Let's start by redrawing and simplifying the Beta data path so that it will be easier to reason about when we add pipelining.

The first simplification is to focus on sequential execution and so leave out the branch addressing and PC mux logic.

Our simplified Beta always executes the next instruction from PC+4.

We'll add back the branch and jump logic when we discuss control hazards.

The second simplification is to have the register file appear twice in the diagram so that we can tease apart the read and write operations that occur at different stages of instruction execution.

The top Register File shows the combinational read ports, used to when reading the register operands in the RF stage.

The bottom Register File shows the clocked write port, used to write the result into the destination register at the end of the WB stage.

Physically, there's only one set of 32 registers, we've just drawn the read and write circuity as separate components in the diagram.

If we add pipeline registers to the simplified diagram, we see that execution proceeds through the five stages from top to bottom.

If we consider execution of instruction sequences with no data hazards, information is flowing down the pipeline and the pipeline will correctly overlap the execution of all the instructions in the pipeline.

The diagram shows the components needed to implement each of the five stages.

The IF stage contains the program counter and the main memory interface for fetching instructions.

The RF stage has the register file and operand multiplexers.

The ALU stage uses the operands and computes the result.

The MEM stage handles the memory access for load and store operations.

And the WB stage writes the result into the destination register.

In each clock cycle, each stage does its part in the execution of a particular instruction.

In a given clock cycle, there are five instructions in the pipeline.

Note that data accesses to main memory span almost two clock cycles.

Data accesses are initiated at the beginning of the MEM stage and returning data is only needed just before the end of the WB stage.

The memory is itself pipelined and can simultaneously finish the access from an earlier instruction while starting an access for the next instruction.

This simplified diagram isn't showing how the control logic is split across the pipeline stages.

How does that work?

Note that we've included instruction registers as part of each pipeline stage, so that each stage can compute the control signals it needs from its instruction register.

The encoded instruction is simply passed from one stage to the next as the instruction flows through the pipeline..

Each stage computes its control signals from the opcode field of its instruction register.

The RF stage needs the RA, RB, and literal fields from its instruction register.

And the WB stage needs the RC field from its instruction register.

The required logic is very similar to the unpipelined implementation, it's just been split up and moved to the appropriate pipeline stage.

We'll see that we will have to add some additional control logic to deal correctly with pipeline hazards.

Our simplified diagram isn't so simple anymore!

To see how the pipeline works, let's follow along as it executes this sequence of six instructions.

Note that the instructions are reading and writing from different registers, so there are no potential data hazards.

And there are no branches and jumps, so there are no potential control hazards.

Since there are no potential hazards, the instruction executions can be overlapped and their overlapped execution in the pipeline will work correctly.

Okay, here we go!

During cycle 1, the IF stage sends the value from the program counter to main memory to fetch the first instruction (the green LD instruction), which will be stored in the RF-stage instruction register at the end of the cycle.

Meanwhile, it's also computing PC+4, which will be the next value of the program counter.

We've colored the next value blue to indicate that it's the address of the blue instruction in the sequence.

We'll add the appropriately colored label on the right of each pipeline stage to indicate which instruction the stage is processing.

At the start of cycle 2, we see that values in the PC and instruction registers for the RF stage now correspond to the green instruction.

During the cycle the register file will be reading the register operands, in this case R1, which is needed for the green instruction.

Since the green instruction is a LD, ASEL is 0 and BSEL is 1, selecting the appropriate values to be written into the A and B operand registers at the end of the cycle.

Concurrently, the IF stage is fetching the blue instruction from main memory and computing an updated PC value for the next cycle.

In cycle 3, the green instruction is now in the ALU stage, where the ALU is adding the values in its operand registers (in this case the value of R1 and the constant 4) and the result will be stored in Y_MEM register at the end of the cycle.

In cycle 4, we're overlapping execution of four instructions.

The MEM stage initiates a memory read for the green LD instruction.

Note that the read data will first become available in the WB stage - it's not available to CPU in the current clock cycle.

In cycle 5, the results of the main memory read initiated in cycle 4 are available for writing to the register file in the WB stage.

So execution of the green LD instruction will be complete when the memory data is written to R2 at the end of

cycle 5.

Meanwhile, the MEM stage is initiating a memory read for the blue LD instruction.

The pipeline continues to complete successive instructions in successive clock cycles.

The latency for a particular instruction is 5 clock cycles.

The throughput of the pipelined CPU is 1 instruction/cycle.

This is the same as the unpipelined implementation, except that the clock period is shorter because each pipeline stage has fewer components.

Note that the effects of the green LD, i.e., filling R2 with a new value, don't happen until the rising edge of the clock at the end of cycle 5.

In other words, the results of the green LD aren't available to other instructions until cycle 6.

If there were instructions in the pipeline that read R2 before cycle 6, they would have gotten an old value!

This is an example of a data hazard.

Not a problem for us, since our instruction sequence didn't trigger this data hazard.

Tackling data hazards is our next task.