Here's a failed attempt at pipelining a circuit.

For what value of K is the circuit a K-pipeline?

Well, let's count the number of registers along each path from system inputs to system outputs.

The top path through the A and C components has 2 registers.

As does the bottom path through the B and C components.

But the middle path through all three components has only 1 register.

Oops, this not a well-formed K-pipeline.

Why do we care?

We care because this pipelined circuit does not compute the same answer as the original unpipelined circuit.

The problem is that successive generations of inputs get mixed together during processing.

For example, during cycle i+1, the B module is computing with the current value of the X input but the previous value of the Y input.

This can't happen with a well-formed K-pipeline.

So we need to develop a technique for pipelining a circuit that guarantees the result will be well-formed.

Here's our strategy that will ensure if we add a pipeline register along one path from system inputs to system outputs, we will add pipeline registers along every path.

Step 1 is to draw a contour line that crosses every output in the circuit and mark its endpoints as the terminal points for all the other contours we'll add.

During Step 2 continue to draw contour lines between the two terminal points across the signal connections between modules.

Make sure that every signal connection crosses the new contour line in the same direction.

This means that system inputs will be one side of the contour and system outputs will be on the other side.

These contours demarcate pipeline stages.

Place a pipeline register wherever a signal connection intersects the pipelining contours.

Here we've marked the location of pipeline registers with large black dots.

By drawing the contours from terminal point to terminal point we guarantee that we cross every input-output path, thus ensuring our pipeline will be well-formed.

Now we can compute the system's clock period by looking for the pipeline stage with the longest register-to-register or input-to-register propagation delay.

With these contours and assuming ideal zero-delay pipeline registers, the system clock must have a period of 8 ns to accommodate the operation of the C module.

This gives a system throughput of 1 output every 8 ns.

Since we drew 3 contours, this is a 3-pipeline and the system latency is 3 times 8 ns or 24 ns total.

Our usual goal in pipelining a circuit is to achieve maximum throughput using the fewest possible registers.

So our strategy is to find the slowest system component (in our example, the C component) and place pipeline registers on its inputs and outputs.

So we drew contours that pass on either side of the C module.

This sets the clock period at 8 ns, so we position the contours so that longest path between any two pipeline registers is at most 8.

There are often several choices for how to draw a contour while maintaining the same throughput and latency.

For example, we could have included the E module in the same pipeline stage as the F module.

Okay, let's review our pipelining strategy.

First we draw a contour across all the outputs.

This creates a 1-pipeline, which, as you can see, will always have the same throughput and latency as the original combinational circuit.

Then we draw our next contour, trying to isolate the slowest component in the system.

This creates a 2-pipeline with a clock period of 2 and hence a throughput of 1/2, or double that of the 1-pipeline.

We can add additional contours, but note that the 2-pipeline had the smallest possible clock period, so after that additional contours add stages and hence increase the system's latency without increasing its throughput.

Not illegal, just not a worthwhile investment in hardware.

Note that the signal connection between the A and C module now has two back-to-back pipelining registers.

Nothing wrong with that; it often happens when we pipeline a circuit where the input-output paths are of different lengths.

So our pipelining strategy will be to pipeline implementations with increased throughput, usually at the cost of increased latency.

Sometimes we get lucky and the delays of each pipeline stage are perfectly balanced, in which case the latency will not increase.

Note that a pipelined circuit will NEVER have a smaller latency than the unpipelined circuit.

Notice that once we've isolated the slowest component, we can't increase the throughput any further.

How do we continue to improve the performance of circuits in light of these performance bottlenecks?

One solution is to use pipelined components if they're available!

Suppose we're able to replace the original A component with a 2-stage pipelined version A-prime.

We can redraw our pipelining contours, making sure we account for the internal pipeline registers in the A-prime component.

This means that 2 of our contours have to pass through the A-prime component, guaranteeing that we'll add pipeline registers elsewhere in the system that will account for the two-cycle delay introduced by A-prime.

Now the maximum propagation delay in any stage is 1 ns, doubling the throughput from 1/2 to 1/1.

This is a 4-pipeline so the latency will be 4 ns.

This is great!

But what can we do if our bottleneck component doesn't have a pipelined substitute.

We'll tackle that question in the next section.