The most straightforward way to improve performance is to reduce the propagation delay of a circuit.

Let's look at a perennial performance bottleneck: the ripple-carry adder.

To fix it, we first have to figure out the path from inputs to outputs that has the largest propagation delay, i.e., the path that's determining the overall t_PD.

In this case that path is the long carry chain following the carry-in to carry-out path through each full adder module.

To trigger the path add -1 and 1 by setting the A inputs to all 1's and the B input to all 0's except for the low-order bit which is 1.

The final answer is 0, but notice that each full adder has to wait for the carry-in from the previous stage before it produces 0 on its sum output and generates a carry-out for the next full adder.

The carry really does ripple through the circuit as each full adder in turn does its thing.

To total propagation delay along this path is N-1 times the carry-in to carry-out delay of each full adder, plus the delay to produce the final bit of the sum.

How would the overall latency change if we, say, doubled the size of the operands, i.e., made N twice as large?

It's useful to summarize the dependency of the latency on N using the "order-of" notation to give us the big picture.

Clearly as N gets larger the delay of the XOR gate at the end becomes less significant, so the order-of notation ignores terms that are relatively less important as N grows.

In this example, the latency is order N, which tells us that the latency would be expected to essentially double if we made N twice as large.

The order-of notation, which theoreticians call asymptotic analysis, tells us the term that would dominate the result as N grows.

The yellow box contains the official definition, but an example might make it easier to understand what's happening.

Suppose we want to characterize the growth in the value of the equation n^2 + 2n + 3 as n gets larger.

The dominant term is clearly n^2 and the value of our equation is bounded above and below by simple multiples of n^2, except for finitely many values of n.

The lower bound is always true for n greater than or equal to 0.

And in this case, the upper bound doesn't hold only for n equal to 0, 1, 2, or 3.

For all other positive values of n the upper inequality is true.

So we'd say this equation was "order N^2".

There are actually two variants for the order-of notation.

We use the Theta notation to indicate that g(n) is bounded above AND below by multiples of f(n).

The "big O" notation is used when g(n) is only bounded above by a multiple of f(n).

Here's a first attempt at improving the latency of our addition circuit.

The trouble with the ripple-carry adder is that the high-order bits have to wait for the carry-in from the low-order bits.

Is there a way in which we can get high half the adder working in parallel with the low half?

Suppose we wanted to build a 32-bit adder.

Let's make two copies of the high 16 bits of the adder, one assuming the carry-in from the low bits is 0, and the other assuming the carry-in is 1.

So now we have three 16-bit adders, all of which can operate in parallel on newly arriving A and B inputs.

Once the 16-bit additions are complete, we can use the actual carry-out from the low-half to select the answer from the particular high-half adder that used the matching carry-in value.

This type of adder is appropriately named the carry-select adder.

The latency of this carry-select adder is just a little more than the latency of a 16-bit ripple-carry addition.

This is approximately half the latency of the original 32-bit ripple-carry adder.

So at a cost of about 50% more circuitry, we've halved the latency!

As a next step, we could apply the same strategy to halve the latency of the 16-bit adders.

And then again to halve the latency of the 8-bit adders used in the previous step.

At each step we halve the adder latency and add a MUX delay.

After log2(N) steps, N will be 1 and we're done.

At this point the latency would be some constant cost to do a 1-bit addition, plus log2(N) times the MUX latency to select the right answers.

So the overall latency of the carry-select adder is order log(N).

Note that log2(N) and log(N) only differ by a constant factor, so we ignore the base of the log in order-of notation.

The carry-select adder shows a clear performance-size tradeoff available to the designer.

Since adders play a big role in many digital systems, here's a more carefully engineered version of a 32-bit carry-select adder.

You could try this in your ALU design!

The size of the adder blocks has been chosen so that the trial sums and the carry-in from the previous stage arrive at the carry-select MUX at approximately the same time.

Note that since the select signal for the MUXes is heavily loaded we've included a buffer to make the select signal transitions faster.

This carry-select adder is about two-and-a-half times faster than a 32-bit ripple-carry adder at the cost of about twice as much circuitry.

A great design to remember when you're looking to double the speed of your ALU!