We've been designing our processing pipelines to have all the stages operate in lock step, choosing the clock period to accommodate the worst-case processing time over all the stages.

This is what we'd call a synchronous, globally timed system.

But what if there are data dependencies in the processing time, i.e., if for some data inputs a particular processing stage might be able to produce its output in a shorter time?

Can we design a system that could take advantage of that opportunity to increase throughput?

One alternative is to continue to use a single system clock, but for each stage to signal when it's ready for a new input and when it has a new output ready for the next stage.

It's fun to design a simple 2-signal handshake protocol to reliably transfer data from one stage to the next.

The upstream stage produces a signal called HERE-IS-X to indicate that is has new data for the downstream stage.

And the downstream stage produces a signal called GOT-X to indicate when it is willing to consume data.

It's a synchronous system so the signal values are only examined on the rising edge of the clock.

The handshake protocol works as follows: the upstream stage asserts HERE-IS-X if it will have a new output value available at the next rising edge of the clock.

The downstream stage asserts GOT-X if it will grab the next output at the rising edge of the clock.

Both stages look at the signals on the rising edge of the clock to decide what to do next.

If both stages see that HERE-IS-X and GOT-X are asserted at the same clock edge, the handshake is complete and the data transfer happens at that clock edge.

Either stage can delay a transfer if they are still working on producing the next output or consuming the previous input.

It's possible, although considerably more difficult, to build a clock-free asynchronous self-timed system that uses a similar handshake protocol.

The handshake involves four phases.

In phase 1, when the upstream stage has a new output and GOT-X is deasserted, it asserts its HERE-IS-X signal

and then waits to see the downstream stage's reply on the GOT-X signal.

In phase 2, the downstream stage, seeing that HERE-IS-X is asserted, asserts GOT-X when it has consumed the available input.

In phase 3, the downstream stage waits to see the HERE-IS-X go low, indicating that the upstream stage has successfully received the GOT-X signal.

In phase 4, once HERE-IS-X is deasserted, the downstream stage deasserts GOT-X and the transfer handshake is ready to begin again.

Note that the upstream stage waits until it sees the GOT-X deasserted before starting the next handshake.

The timing of the system is based on the transitions of the handshake signals, which can happen at any time the conditions required by the protocol are satisfied.

No need for a global clock here!

It's fun to think about how this self-timed protocol might work when there are multiple downstream modules, each with their own internal timing.

In this example, A's output is consumed by both the B and C stages.

We need a special circuit, shown as a yellow box in the diagram, to combine the GOT-X signals from the B and C stages and produce a summary signal for the A stage.

Let's take a quick look at the timing diagram shown here.

After A has asserted HERE-IS-X, the circuit in the yellow box waits until both the B and the C stage have asserted their GOT-X signals before asserting GOT-X to the A stage.

At this point the A stage deasserts HERE-IS-X, then the yellow box waits until both the B and C stages have deasserted their GOT-X signals, before deasserting GOT-X to the A stage.

Let's watch the system in action!

When a signal is asserted we'll show it in red, otherwise it's shown in black.

A new value for the A stage arrives on A's data input and the module supplying the value then asserts its HERE-IS-X signal to let A know it has a new input.

At some point later, A signals GOT-X back upstream to indicate that it has consumed the value, then the upstream stage deasserts HERE-IS-X, followed by A deasserting its GOT-X signal.

This completes the transfer of the data to the A stage.

When A is ready to send a new output to the B and C stages, it checks that its GOT-X input is deasserted (which it is), so it asserts the new output value and signals HERE-IS-X to the yellow box which forwards the signal to the downstream stages.

B is ready to consume the new input and so asserts its GOT-X output.

Note that C is still waiting for its second input and has yet to assert its GOT-X output.

After B finishes its computation, it supplies a new value to C and asserts its HERE-IS-X output to let C know its second input is ready.

Now C is happy and signals both upstream stages that it has consumed its two inputs.

Now that both GOT-X inputs are asserted, the yellow box asserts A's GOT-X input to let it know that the data has been transferred.

Meanwhile B completes its part of the handshake, and C completes its transaction with B and A deasserts HERE-IS-X to indicate that it has seen its GOT-X input.

When the B and C stages see their HERE-IS-X inputs go low, they their finish their handshakes by deasserting their GOT-X outputs, and when they're both low, the yellow box lets A know the handshake is complete by deserting A's GOT-X input.

Whew!

The system has returned to the initial state where A is now ready to accept some future input value.

This an elegant design based entirely on transition signaling.

Each module is in complete control of when it consumes inputs and produces outputs, and so the system can process data at the fastest possible speed, rather than waiting for the worst-case processing delay.