Strategies for pipelining logic

Aug 14, 2017

One of the things that new FPGA students struggle with is the fact that *everything* in digital logic takes place in parallel.

Many of these students come from a computer science background. They understand how an algorithm works, and how one thing must take place after another in a specific sequence. They tend to struggle, though, with the idea that every step in an algorithm occupies a piece of digital logic that will act on *every* clock tick–whether used or not.

One solution to sequencing operations is to create a giant state machine. The reality, though, is that an FPGA tends to create all the logic for every state at once, and then only select the correct answer at the end of each clock tick. In this fashion, a state machine can be very much like the simple ALU we've discussed.

On the other hand, if the FPGA is going to implement all of the logic for the operation anyway, why not arrange each of those operations into a sequence, where each stage does something useful? This approach rearranges the algorithm into a pipeline. Pipelining tends to be faster than the state machine approach for accomplishing the same algorithm, and it can even be more resource efficient, although it isn't necessarily so.

The difficult part of a digital logic pipeline is that the pipeline runs and produces outputs even when the inputs to the pipeline are not (yet) valid.

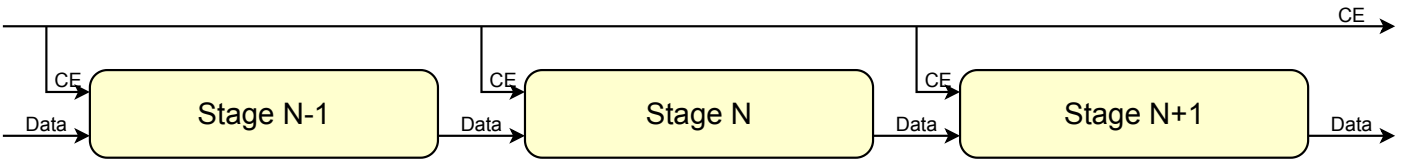
So, let's discuss several different strategies for handling the signaling associated with pipeline logic. In general, there's no one size fits all strategy. The strategy you pick will depend upon the needs of your algorithm, and its data source (input) and destination (output).

We'll work our way through several different strategies from the simplest to most complex.

The global valid signal for sampled data

The first strategy for handling pipelining that we'll discuss is to use a global valid signal. At each stage, the data coming into the pipeline is valid when the global valid signal is true. Likewise, each stage may take no more clocks to finish then there are between valid signals. I like to use the `CE` or "clock enable" signal to represent this valid logic. Hence, Fig 1 shows a block diagram of this sort communication.

Fig 1: Pipelining with a global valid signal

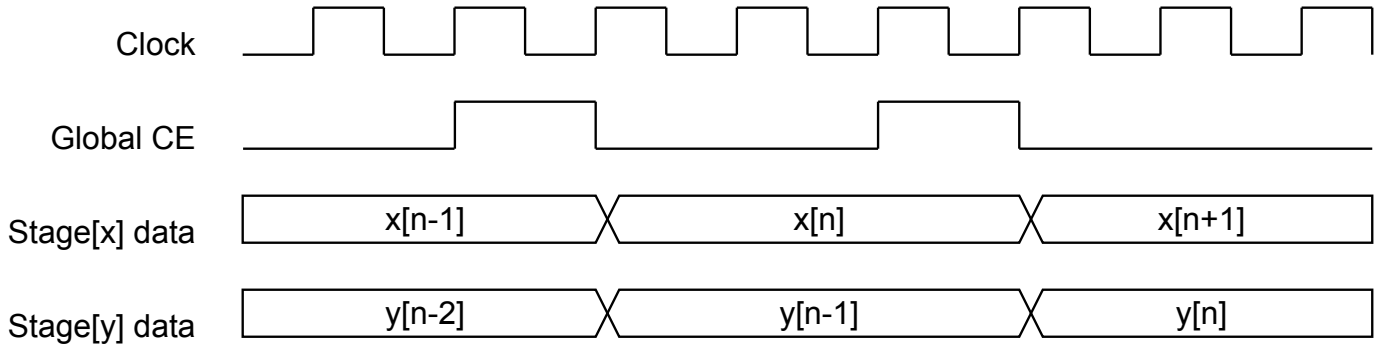


The basic rules are this:

1. There is a global `CE` line, synchronous with the clock. This line is true anytime new data is ready.
2. Logic is allowed to transition *only* when this `CE` line is true

This leads to a trace diagram similar to Fig 2 below.

Fig 1: Pipelining with a global valid signal



The neat thing about this approach is that there's no real pipelining logic required per se. Each stage just waits for the global valid signal to be true, and then applies its logic.

```
always @(posedge i_clk)
        if (i_ce)
        begin
                output <= (some function of)(i_input);
        end // else *NOTHING*.  Nothing is allowed to change without CE=1
```

You may find very quickly, though, that this approach doesn't handle all pipelining needs. However, while it doesn't handle all pipelining needs, it does handle the needs of a crucial problem set: *signal processing*.

In a typical signal processing application, data either comes into an FPGA from an analog to digital (A/D) converter, or it goes out of the FPGA to a digital to analog (D/A) converter, or perhaps it does both. The sample rate defines when the `CE` signal needs to be high, and any transient states are flushed through the system. The digital logic engineer's job is to manipulate the samples along the way.

Because the data moves through the FPGA at a fixed rate, and because it never suddenly changes speeds, the global valid signal works quite well for it.

Applications include digital filtering, digital phase lock loops, numerically controlled oscillators, and more. Indeed, anything that works at a *fixed data rate* is usually a good candidate for this method of pipelining.

Why, even our resamplers have worked off of the concept of a global valid signal, they've just had to deal with two different valid signals: one that holds for one clock per input sample, and another that holds for one clock for every output sample. [1] [2]

The traveling CE to reduce latency

The global valid signal we discussed above, though, has two basic problems. The first problem is that there's no way to know if an output sample is "valid" or not". The second is that the whole operation depends upon a uniform clock creating the `CE` signal. What happens if the data is produced in a bursty fashion, and you want to know not only *when* the output is valid but also *if* the output is valid? In this case, another approach is required.

I'm going to call this second approach the "traveling CE" approach. Basically, each stage in the pipeline propagates the CE forward, as in Fig 3.

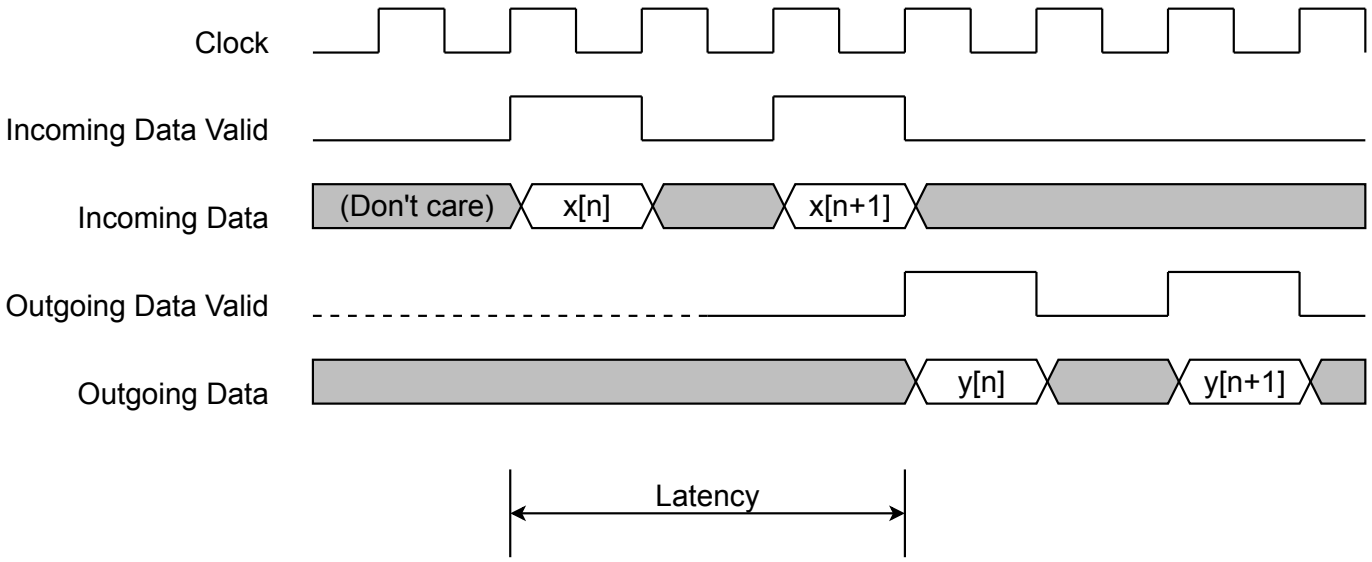
Fig 3: Pipelining with a traveling CE

The basic rules to this approach are:

1. Whenever the `CE` signal is true, the data associated with it must also be valid.
2. At the end of every stage of processing, a `CE` signal must be produced, together with the output data for that stage.
3. The `CE` signal *must* be initialized to zero. Further, if any reset is to be used, the `CE` *must* be set to zero on any reset. (The data is a don't care on reset, but the `CE` line *must* be set to zero.)
4. Nothing is allowed to change *except* on a `CE` signal. Hence, the only time the incoming data is referenced is when `i_ce` (the input `CE` line to a pipeline stage) is high.
5. Finally, every piece of logic must be ready, at all times, for a new value to enter into the pipeline. This particular pipeline strategy cannot handle stalls.

A trace of this type of logic might look like Fig 4.

Fig 4: Pipelining with a traveling CE



In Verilog, the approach looks like:

```
initial o_ce = 1'b0;
always @(posedge i_clk)
        if (i_reset)
                o_ce <= 1'b0;
        else
                o_ce <= i_ce;
always @(posedge i_clk)
        if (i_ce)
                o_output <= ... // some function of i_input;
        // else *NOTHING*.  Nothing is allowed to change except on
        // a reset or an i_ce
```

This approach works very well when the pipeline can be separated into stages that take no more than a single input valid signal. Likewise, it works well when none of the stages depends upon any feedback from future results. In other words, if nothing ever needs to wait, then this approach to pipelining works fairly well.

Applications of this pipelining approach include logic multiplies, Fourier transform processing (example), video processing, gear-boxes, and more. Indeed, we used this approach within our hexbus debugging bus to hook the input processing chain together. You may notice, however, that this approach didn't work on the output processing chain. The problem there was that the UART transmitter took longer than a single clock to transmit a character, and so another pipeline signaling approach was needed–one that allowed the end of the pipeline to control the rate of the pipeline.

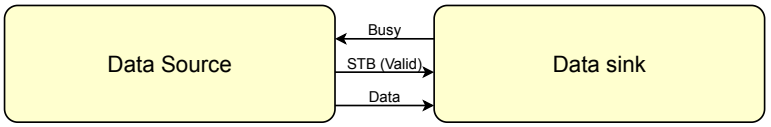
The simple handshake

The biggest problem with the traveling CE approach to pipelining is that there's no way to handle the case where the listener isn't ready. To use a UART transmitter as an example, you can create a

pipeline to fill the transmitter, but what do you do when the transmitter is busy? This requires a simple handshake approach, one that I will describe here in this section.

The basic hand shake relies on a pair of signals–one from the current device and another from the next one in the pipeline. We'll call these signals `STB` (or valid) and `BUSY`, although the wires go by a variety of other names depending upon the interface. Fig 5 shows a simple pipeline, having only two stages, with the handshaking signals working through it.

Fig 5: Block diagram of a simple handshake



The basic rules associated with the simple handshake are:

1. A transaction takes place any time the `STB` line is true and the `BUSY` line is false.
2. The receiving pipeline stage needs to be careful *not* to ever lower the `BUSY` line, unless it is ready to receive data on the next clock.
3. The `STB` line should be raised any time data is ready to send. The data source *must not* wait for `BUSY` to be false before raising the `STB` line.

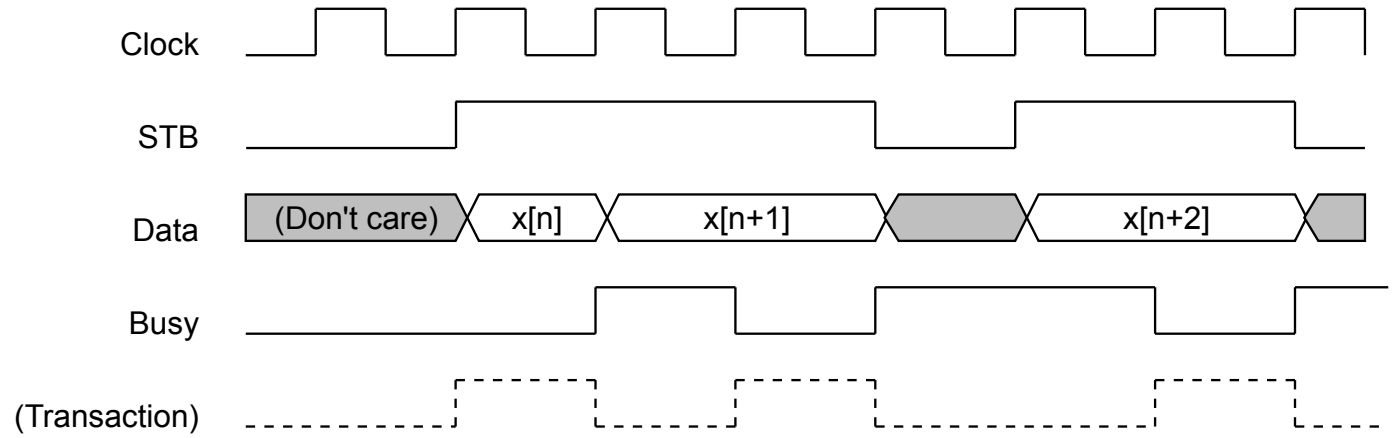
   The issue with not waiting for `BUSY` to be true has to do with avoiding deadlocks. By setting `STB` independent of `BUSY`, the dependence between the two is removed.
4. Likewise, the `BUSY` line *should idle* in the not busy condition.

   While many AXI demonstration implementations idle with the `AXI_*READY` line false (their equivalent of a `BUSY` line being true), this will only slow down your interaction by an unnecessary clock. Remember, one of the goals of pipelining logic is speed. Making `BUSY` true when it doesn't need to be will slow down the pipeline.
5. Once `STB` is raised, the data being transferred cannot be changed until the clock after the transaction takes place. That is, use `(STB)&&(!BUSY)` to determine if things need to change.
6. The data lines are in a "don't care" condition any time `STB` is false.
7. The `STB` and `BUSY` lines must be initialized to zero. If you have a need for a reset or a clear pipeline operation, these signals need to be returned to zero on either of these signals.

   Since the data lines will be placed into a "don't care" condition, they don't need to have any value on reset.

If you look at this handshake from the standpoint of the logic involved, a trace would look like Fig 6.

Fig 6: A simple handshake pipeline signal



Pay close attention to the “(Transaction)” line. This line is the key to understanding the trace. It is formed from the combinational result of `(STB)&&(!BUSY)`. This line is the analog of the `CE` line in the traveling `CE` approach from before. When the (Transaction) line is high, the data is valid (since `STB` was high), and the processing can step forward one further step.

I’ve used this approach many times when building controllers for slow hardware. In those cases, the receiver generally looks like:

```
initial o_busy = 1'b0;
always @(posedge i_clk)
        if (i_reset)
        begin
                o_busy <= 1'b0;
                state  <= IDLE_STATE;
        end if ((i_ce)&&(!o_busy)) begin
                // We just accepted an input sample into this controller
                // Turn o_busy on, and start processing this input.
                o_busy <= 1'b1;
                state  <= START_STATE;
                data   <= i_data;
                // etc.
        end else case(state)
                // A state machine is used to handle an interaction
                // with the hardware now that a request has been made.
                // ....
                FINAL_STATE: begin
                        o_busy <= 1'b0;
                        state  <= IDLE_STATE;
                        // ... other logic
                        end
                // default:
                endcase
```

As you might have guessed by now, my UART transmitter uses this approach. You can also find several examples of interacting with such a transmitter among the bench tests for the UART

transmitter. Perhaps closer to home, you may find this approach to pipelining used by the transmit half of the hexbus module.

The problem with the UART example is that it doesn't really capture the logic required in any mid-point pipeline stage, only the final stage.

At the midpoint, there are two choices for how to handle things. You can either register the `BUSY` signal and suffer a pipeline stall in between any two transactions, or you can create a `BUSY` signal using combinational logic. The combinational `BUSY` (shown in the example code below) has the problem that the time required for `BUSY` determination accumulates as you move backwards through the pipeline. This can slow down your logic, so when this combinational path approaches your clock period it becomes undesirable. On the other hand, if it's a UART or any other slow peripheral at the end of the logic pipeline (flash, ICAPE2 OLEDrgb, etc), then you might not care about any clocks lost in the `BUSY` signal calculation.

All that is to say, here's an example of how to build a pipeline component with this handshake as both the input and the output to the component:

```
initial r_busy = 1'b0;
initial o_stb  = 1'b0;
always @(posedge i_clk)
        if (i_reset)
        begin
                // busy and stb must be cleared on any reset
                // Data is a don't care
                r_busy <= 1'b0;
                o_stb  <= 1'b0;
        end if (!o_busy)
        begin
                o_stb <= 0;
                if (i_stb)
                begin
                        // An incoming transaction has just taken place
                        r_busy <= 1'b1;
                        // begin your logic here ...
                        //
                end
                // else we remain in an idle condition
        end else if ((o_stb)&&(!i_busy))
        begin
                // An output transaction just took place
                r_busy <= 1'b0;
                o_stb  <= 1'b0;
        end else if (!o_stb) begin
                // o_busy is true, so you can perform any necessary logic he
```

```
                if (your logic is complete)
                        o_stb <= 1'b1;
        end // else we have to wait for our output data to be accepted
        // by the next stage before we can move on.
```

The final step is to set the output busy line, `o_busy`. We use `r_busy` to record any time our own component is busy. The final busy is set up so that no empty delay cycles will be necessary, even though it requires some combinational logic (i.e. borrowed clock time) to do.

```
assign  o_busy = (r_busy) && (!o_stb || i_busy);
```

This example is easily modified to to remove the combinational accumulation by simply setting the `o_busy` line via the `r_busy` logic.

```
assign  o_busy = r_busy;
```

This will create an idle cycle between pipeline stages, but it will also fix the combinational time accumulation problem.

Examples of this type of handshaking abound. For example, the Wishbone bus has a form of interaction that uses this form of handshaking, although it changes the signal names a touch. While the `STB` name remains the same, the Wishbone bus uses `STALL` as the name for its `BUSY` line. Likewise, the AXI bus specification uses this form of handshaking. Indeed, it uses this form across *five separate* hand-shaking channels. AXI uses the terms `*AXI_*VALID` for `STB` and `*AXI_*READY` instead of `!BUSY`. We've also already discussed the transmit half, i.e. return processing chain, of the main dbgbus module as another example.

The buffered handshake

If you are trying to go for high speed, for example if you wished to run the ZipCPU at 200MHz instead of its more natural 100MHz speed, then the simple handshake method can suffer from severe timing problems as the pipeline grows in length. This problem is twofold. First, any time a `BUSY` signal needs to pass combinationally from the `N` th stage back to the first stage, the time required increases at each stage. This can slow down your logic. The logic chain line can cross large sections of digital logic, incurring timing delays from one end of the chip to another, while the logic elements along the way that this signal needs to pass through just contribute to the pain. Second, while it is possible to slow the `BUSY` signals down, by inserting a stall between pipeline stages, this can slow the pipeline down by a factor of two.

Another approach is needed.

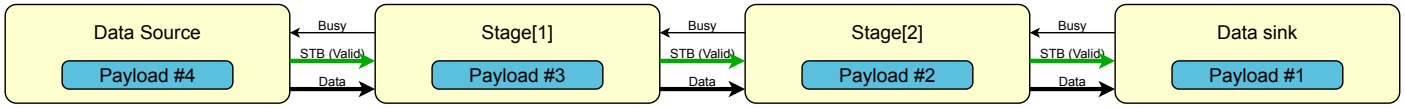
The way to mitigate this problem to set the `BUSY` value with a clocked register. This means that when a subsequent pipeline stage isn't ready (a pipeline stall), it will cost a clock until the `BUSY` line can be true. To avoid losing any data, the data that arrived before the `BUSY` signal could go high will need to be stored in a buffer.

For this reason, I'm going to call this a "buffered handshake".

The "buffered handshake" is going to use the exact same signals as we showed in Fig 5 above. Further, although Fig 6 shows what the signaling might look like, it doesn't capture the concept of the `BUSY` signal propagating from the end to the beginning, with the data bunching up in the middle like an accordion.

Perhaps some pictures would help this explanation. Consider a four stage pipeline, such as the one shown in Fig. 7.

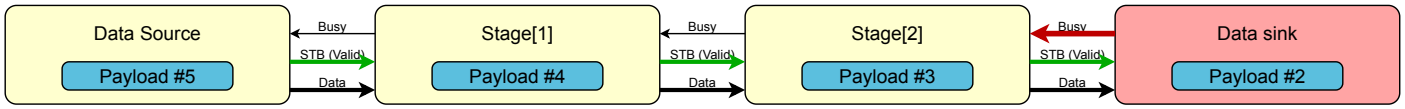
Fig 7: A four stage pipeline using a buffered handshake



This figure shows four separate pipeline stages, from the data source or generator, to the ultimate consumer of the data. The pipeline in this figure is currently full, with each stage having a payload value within it. As a result, each of the `STB` signals is valid going into the next block.

Fig 8 shows what happens if the final stage in this pipeline stalls on the next clock.

Fig 8: Final pipeline stage stalls

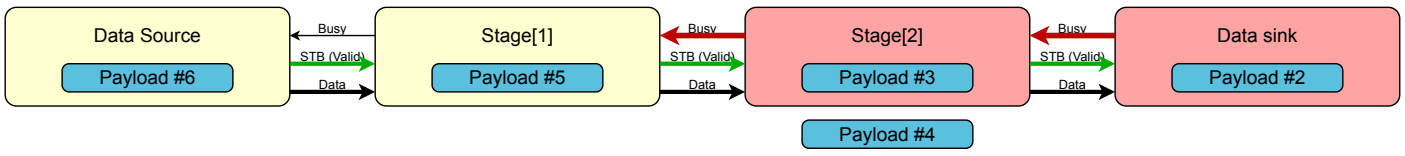


Were this the simple handshake we discussed above, all of the `BUSY` flags would go true at once.

In the case of the buffered handshake, only one `BUSY` flag becomes true.

This means that `Stage[2]` hasn't had an opportunity to set it's `BUSY` flag. It has no choice but to ingest `payload #4` or risk dropping it (this would be bad). Therefore, `Stage[2]` has a buffer which it uses to store `payload #4` on the next clock, as shown in Fig 9.

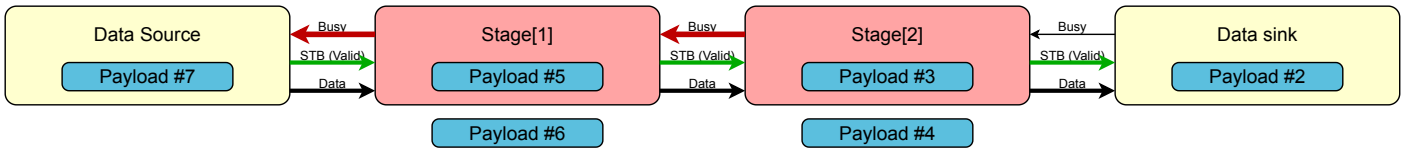
Fig 9: The stall propagates



This leaves `Stage[1]` in the position `Stage[2]` was in on the last clock. It cannot push `payload #5` forward, and yet it hasn't had the opportunity to set its `BUSY` line. It *must* accept `payload #6`. It does so by placing it into its buffer.

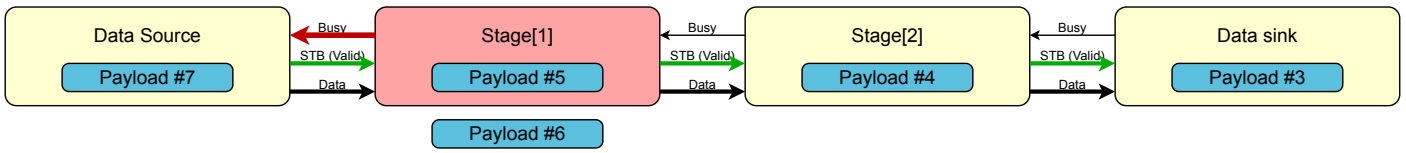
If at this time the data sink now becomes available, it will lower its busy line, yielding an image looking like Fig 10.

Fig 10: The pipeline starts to clear



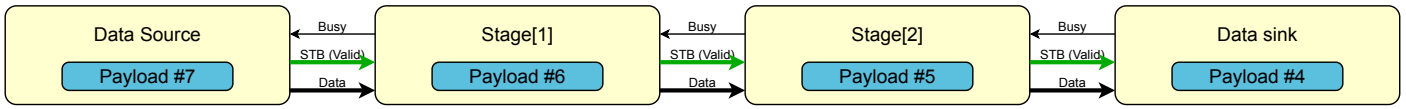
As this pipeline clears, `Stage[2]` transmits `payload #3` and clears its `BUSY` flag. It's now ready to transmit `payload #4` on the next clock, as well as to received `payload #5` on that same clock, as in Fig 11.

Fig 11: The pipeline continues to clear



At this point, `Stage[1]` can now flush its buffer and the pipeline is clear again as in Fig 12.

Fig 12: The pipeline finally clears



You may notice that this pipeline uses the same `STB` and `BUSY` signals that we've used for the simple handshaking approach to pipelining. The difference with this approach is that the `BUSY` signal is registered, and must wait on a clock to propagate.

The rules defining this behavior are very similar to those for the simple handshake above:

1. A data transfer takes place any time `(STB)&&(!BUSY)` whether the given pipeline stage is ready for it or not.

2. If the output `(STB)&&(BUSY)` are true, but the input `(STB)&&(!BUSY)` is true, the data must be stored into a buffer.

3. If `BUSY` is true on the input, but `BUSY` isn't true on the output, then the buffer's values can be released and sent forwards and we can set `BUSY` for the incoming data to be false.

So, now that you know the concept, how shall we set up the logic necessary to implement this? We'll look at how to design the logic for one stage in this pipeline only, since the other stages will use similar logic.

The following logic is rather confusing when it comes to naming, since both input and output ports share names. I'll use the `i_` prefix to reference a wire coming into a stage, whether the `i_stb` line coming from the previous stage or the `i_busy` line coming from the subsequent stage. In a similar manner, I'll use the `o_` prefix to reference logic leaving this stage, whether the `o_stb`

sent to the subsequent pipeline stage to indicate that this stage has something to pass on, or the `o_busy` line to send to the previous stage to indicate that this stage is now busy. We'll also use the `r_` prefix to reference values within our register, both `r_stb` to indicate that something valid is in it as well as `r_data` to indicate the value of what's in it.

The first requirement is that the pipeline be empty on any reset.

```
initial r_stb  = 1'b0;
initial o_stb  = 1'b0;
initial o_busy = 1'b0;
always @(posedge i_clk)
begin
        if (i_reset)
        begin
                r_stb  <= 1'b0;
                o_stb  <= 1'b0;
                o_busy <= 1'b0;
                // Data is a don't care
        end
```

Next, let's deal with the case where the next or subsequent stage isn't `BUSY`. This should be the normal pipeline flow case. Under normal flow, we'll want to copy the input strobe `i_stb` to the output strobe, `o_stb`. Further, some data may need to be applied to `i_data` to create the output `o_data`. We'll use the notation `logic(i_data)` to indicate this. The `logic()` function is not intended to be valid Verilog, but rather to convey the concept of what's taking place.

If, on the other hand, some data was in the buffer, then `o_busy` must also have been true on this clock. `o_stb` must also be true and `o_data` valid. Since `(o_stb)&&(!i_busy)`, a transaction has taken place and `r_stb` can be copied to `o_stb`, and `r_data` to `o_data`–flushing our buffer.

```
// Always block continued ... (i_reset) is false
        else if (!i_busy) // the next stage is not busy
        begin
                if (!r_stb)
                begin
                        // Nothing is in the buffer, so send the input
                        // directly to the output.
                        o_stb   <= i_stb;

                        // This logic() function is arbitrary, and specific
                        // the what this stage is supposed to do.
                        o_data  <= logic(i_data);
                end else begin
```

```
                        // o_busy is true and something is in our buffer.
                        // Flush the buffer to the output port.
                        o_stb  <= 1'b1;
                        o_data <= r_data; // This is the buffered data

                        // We can ignore the input in this case, since
                        // we'll only be here if `o_busy` is also true.
                end

                // We can also clear any stall condition
                o_busy  <= 1'b0;

                // And declare the register to be empty.
                r_stb   <= 1'b0;
        end
```

The next case is the case where `o_stb` is false (and `i_busy` is true). This case wasn't shown in the diagram series above. It's basically the case where a pipeline stage has no data payload within it at all. In that case, we'll keep `o_busy` false, we'll accept any data, and then set the output `o_stb` value to indicate to the next stage that we have something ready to be read. Who knows, the `i_busy` flag might be de–asserted on the next clock and we might not need to stall.

```
// Always block continued ... (i_reset) is false, (i_busy) is true
//
        else if (!o_stb)
        begin
                o_stb  <= i_stb;
                o_busy <= 1'b0;

                // Keep the buffer empty
                r_stb <= 1'b0;

                // Apply the logic to the input data, and set the output dat
                o_data <= logic(i_data);
        end
```

The last case to deal with is the case where `i_busy` is true, `o_stb` indicates we have a payload loaded, and we now need to store our input into our buffer. Hence, we'll set `r_stb` and mark this stage of the pipeline as `BUSY` with `o_busy`.

```
// Always block continued ... (i_reset) is false, (i_busy) and (o_stb) are b
// true.
        else if ((i_stb)&&(!o_busy))
```

```
        begin
                // If the next stage *is* busy, though, and we haven't
                // stalled yet, then we need to accept the requested value
                // from the input.  We'll place it into a termporary
                // location.
                r_stb  <= (i_stb)&&(o_stb);
                o_busy <= (i_stb)&&(o_stb);
        end
end
```

That ends our giant always block, but we still have one value that we haven't set: `r_data`. `r_data` needs to be set based upon the input data, `i_data`. If you need to apply a `logic()` transform to `i_data`, you can do that here to `r_data`.

```
always @(posedge i_clk)
        if (!o_busy)
                r_data <= logic(i_data);
```

That was a lot harder than the simple handshake, now, wasn't it?

Be aware, the code above hasn't been tested. Although I copied it from a (working) data width bus translation module, I found some bugs and made some changes along the way. Hence, if you try this and find any bugs, then please please write me at the e-mail address in the postscript below.

You may also notice that `r_stb` and `o_busy` above are the same signal. I've kept them separate for conceptual understanding, but these two can be combined into a single signal.

The overall approach, though, is a clear example of how logic resources can be traded to achieve pipeline speed and throughput. Indeed, it is only one of many examples, but its a worthwhile lesson to take away from this exercise. While I haven't done so, I think that if you count the LUTs used by this routine, you'll find that all this extra logic has at least doubled the number of LUTs required.

Although this is a useful approach to pipelining, it may easily be more logic than your problem requires. Indeed, very few of my own routines have ever needed to use this buffered handshaking approach. The routines that have needed to use it tend to be wishbone peripherals with complex logic within them–such as my SDRAM controller, my attempt at a DDR3 SDRAM controller, or a wishbone bus width expansion module that I put together for an HDMI video project.

Conclusion

We've now walked through several examples of the signaling associated with pipeline logic. These examples have gone from the simple global `CE` approach, all the way to a buffered handshake

approach. Which type of pipeline signaling you use will be specific to your problem and your needs. However, these approaches should handle most of the problems you might have.

Let's come back to this topic of pipelining at least one more time, though, and look at how the ZipCPU handles its pipeline signaling within the CPU proper. This is a more complicated environment, as lots of events can stall the CPU along the way. Indeed, handling the pipeline needs of a CPU can be quite a challenge.

That will then be our next post on this topic.

For the time is come that judgment must begin at the house of God: and if it first begin at us, what shall the end be of them that obey not the gospel of God? (1Pet 4:17)