Gisselquist Technology, LLC

Building a Skid Buffer for AXI processing

May 22, 2019

I'm currently in the process of building and verifying several AXI cores, primarily for the purpose of proving that my AXI *formal* Verification IP core works. Some examples of these cores include:

- Xilinx's demonstration core
- My own AXI slave core
- My own AXI crossbar
- A Bus Fault Isolator
- A WB to AXI converter, and
- An AXI to WB converter. This latter one will actually be two converters, AXI-write to WB bridge and AXI read to WB bridge just like the AXI-lite to WB converter was also split into an AXI-lite (write) to WB converter and an AXI-lite (read) to WB converter. The two will then be connected together with a WB arbiter, just like it was with the AXI-lite to WB converter.

Most of these cores have already passed a formal verification check. None of these, however, have passed the FPGA check save perhaps the Xilinx core–which I'm assuming others have used although I haven't used it myself.

These cores are all currently kept in my Wishbone to AXI bridge repository. They aren't there because they really belong there, but rather for a lack of a better place.

I've already blogged about formally verifying Xilinx's AXI demonstration core. I've even blogged about formally verifying Xilinx's AXI-lite demonstration core, as well as demonstrating how to build a bug free AXI-lite core. I'd like to do the same for my AXI (full) slave core as well.

Indeed, I'd like to blog about some or all of these other cores. They each have some very fascinating and useful features.

- For example, the AXI slave core is designed to be able to sustain 100% throughput on both read and write channels. Xilinx's core, for comparison, was only able to achieve just less than a 50% read throughput, and something close to 100% on the write channel, although it didn't quite get there.

- The AXI crossbar is unusual in several respects. First, it is unusual in that it is a public, open source, *formally verified* *formally verified* crossbar. Every tried to simulate both halves of a design, both the master and the slave side of Xilinx's interconnect? An open source AXI crossbar, capable of being Verilated, would be very powerful for this purpose.

  Further, if either Xilinx's AXI demo core, or their AXI-lite demo core is any indication, then this crossbar will have over twice the throughput. Similarly, both of these cores had latent bugs within them, having not been formally verified. Xilinx's crossbar may have similar latent bugs or limitations within it. While I'd love to know, I don't have access to the logic within their crossbar to find out.

- Finally, my brand new bus fault isolator will allow you to connect an unverified AXI design to a larger system, knowing that the bus fault isolator will identify any descrepancies between your core's AXI interface and the formal properties we've discussed that can be used to verify and arbitrary slave, and return a bus error in the case of any error.

  Just think about that for a moment. When I worked with the Cyclone-V, I had a bug in my own design where two Wishbone bus responses got collapsed into one. The ARM on the Cyclone-V then *hung* waiting for that response. It never came. No matter what I tried, I couldn't get access into the design to see what had happened. Had I had this bus fault isolator, the fault in my broken design would have been detected and a bus error returned. I could have then used logic to dig into what was going wrong to find the bug. Even better, the bus fault isolator now has a recovery mode, allowing access to the slave after a reset period.

  Does this sound like a "get out to FPGA-Hell for free" card at all?

There is one key component, however, in all of these designs. Without this key, I wouldn't be able to make any high performance AXI designs. That key component is the *skid buffer*.

I know, I called these "double buffers" some time ago, but I am really starting to like the term "skid buffer". It captures the idea much better, and so I'm going to switch terminology and start calling these things "skid buffer" from now on.

If you are going to build or otherwise work with an AXI design, you really need to understand a basic skid buffer. Indeed, that's really the whole occasion for this post: I was going to post about my AXI slave core, and I realized that I was either going to need to spend a long section explaining skid buffers, or I would need to separate that material into its own post.

The basic concept of the skid buffer

So, just what is a skid buffer? A skid buffer arises from the need to create a stall signal in a *registered data only* context.

Just to illustrate with an example, the ZipCPU doesn't use skid buffers. As a result, if the CPU needs to wait on a long instruction, such as a memory load or a divide, then the read-operands stage needs to stall lest an instruction be lost. If the read-operands stage stalls, then the decode stage needs to stall. If the decode stage needs to stall, the prefetch needs to stall and so on.

```
always @(*)
begin
	div_stall = div_busy;
	op_stall  = div_stall | mem_busy | cpu_halt | // other things
	dcd_stall = op_stall | pipeline_hazard;
	pf_stall  = .. ///
end
```

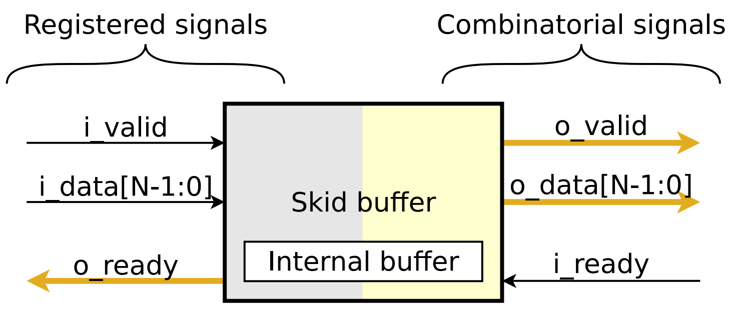
Because the ZipCPU uses *combinatorial* stall signals internally, by the time the stall signal gets to the prefetch stage there's not much slack left in the signal before the next clock edge. Indeed, this was one of the problems I had when I tried to run the ZipCPU at higher clock rates. (It wasn't the only problem …)

If I had used skid buffers instead, the stall signal could have then been *registered*, breaking the timing accumulation.

The problem is that if the stall signal is registered, then the previous processing stage in the pipeline doesn't know about the stall until it finishes its processing and registers its values at the next set of flip-flops. At this point, the data needs to go somewhere or get dropped.

Enter a skid buffer, such as the one shown in Fig. 1 at the right.

Fig 1. A basic skid buffer block diagram



The goal of the skid buffer in Fig. 1 is to bridge the divide between combinatorial logic on the one side and the registered logic on the other–given that the outgoing stall signal (i.e. `!o_ready`) can only be a registered signal.

In this case, I've used the AXI signaling convention, so this skid buffer has `VALID` and `READY` signals on both incoming and outgoing interfaces.

There are two big criteria this skid buffer must meet. First, if ever `VALID & !READY`, the respective data values must remain constant into the next clock cycle. Second, no piece of data may be lost along the way.

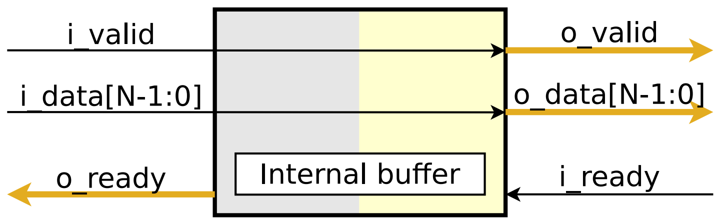
The first of these is very easy to express formally.

```
        // First, we assume this property of the input
        assume property (@(posedge i_clk)
                disable iff (i_reset)
                (i_valid && !o_ready) |=> i_valid && $stable(i_data));

        // Then we assert it when describing the output
        assert property (@(posedge i_clk)
                disable iff (i_reset)
                (o_valid && !i_ready) |=> o_valid && $stable(o_data));
```

In all of my AXI cores, I have wanted to use the data as soon as possible. This means that I don't want to add any more buffers or path logic to my incoming data than necessary.

Fig 2. With no stalls, the buffer acts like a pass through device



Hence, when all is going well and nothing is stalled, then the skid buffer needs to operate like a pass through device, as illustrated in Fig. 2.

In this case, both the incoming valid and data signals pass through the core and go directly to the output.

If the output port is stalled however, then we need to copy everything to an internal buffer, i.e. the "skid" buffer, lest the input data values get lost on the next cycle. This is the meaning of Fig. 3 on the right. In this figure, the incoming valid and data lines are copied directly to the buffer. This gives the buffer its own internal valid and data lines. Further down, when we start discussing the implementation of this core we'll name these `r_valid` and `r_data` respectively–but I'm getting ahead of myself.

Fig 3. Copying the incoming data to an internal buffer

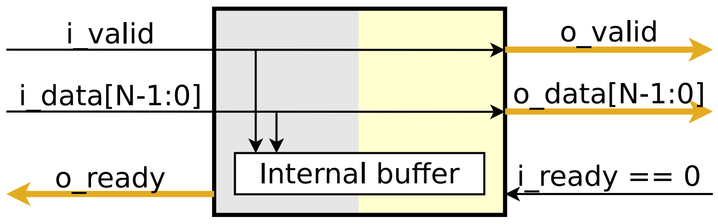
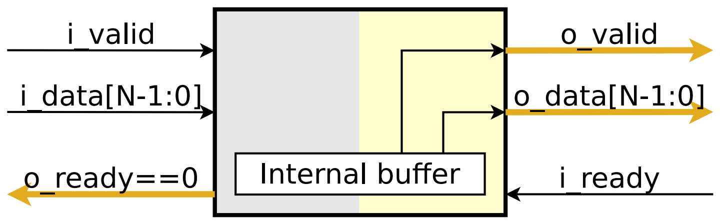


Fig 4. The stall signal propagates upstream



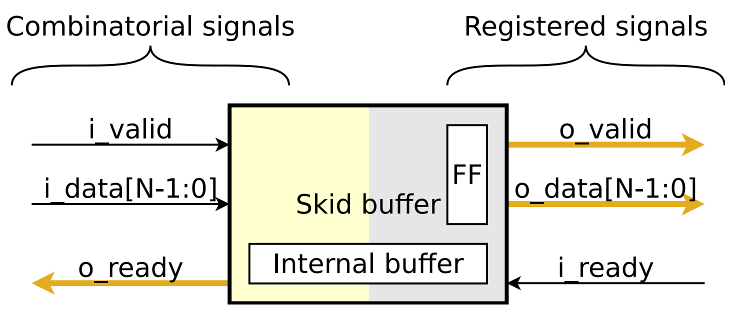
On the next clock cycle, the core can output the incoming values from the last cycle–the ones it just buffered. Further, the incoming interface may move on to its next value, but by now the incoming `o_ready` signal has fallen, so the module feeding this one now knows that it needs to wait. This is shown in Fig. 4 on the left.

The neat thing about all of this is that the logic necessary to implement a skid buffer is fairly straightforward, so let's take a peek at what it takes build one of these today.

Optional Enhancements

After building this first skid buffer, I quickly realized there was a need for a similar skid buffer, but with the output sides reversed. That is, could a skid buffer be created where the incoming side was combinatorial and the outgoing side was registered?

Fig 5. The Skid buffer can be made to register outputs



Our skid buffer might then look like Fig. 5.

In this case, the biggest difference are the set of flip-flops used to register the outputs.

That sounds simple enough. But what if I wanted both? What if I could use combinatorial logic to create the outgoing interface whenever the skid buffer was used on an incoming AXI channel, and combinatorial logic on the incoming interface whenever the skid buffer was used to drive an outgoing AXI channel? To support both purposes, I created a parameter which could be used to select between them, `OPT_OUTREG`. When this parameter is set, the skid buffer will register all outputs. When it isn't set, the outputs can then be combinatorially driven. In all cases, the upstream stall signal will be registered.

Surely that's simple enough to build, right?

Not quite. I wanted more.

I've noticed that with a lot of my Wishbone bus designs, that the Wishbone signals have a very high fan out. They send data all across the chip. When such high fanout signals get set in a manner so as to minimize LUTs, they might end up transitioning often–even when the valid signal (`STB` for Wishbone) is low and so no reader is listening on the other end. Further, every one of these high-fanout transitions takes power.

What if I instead wanted to force any unused data lines to be zero when the valid signal indicated the output was inactive? This might prevent unnecessary transitions, and perhaps even lower the power usage of my designs. (This remains to be determined, but this design is part of my preparation for an experiment or two to see if this is so.)

For this, I assigned another options, `OPT_LOWPOWER`. `OPT_LOWPOWER` is really defined by a set of formal properties. If `OPT_LOWPOWER` is set, then anytime `!o_valid` is true then `o_data` should be zero.

```
	assert property (@(posedge i_clk)
		!o_valid |-> (o_data == 0));
```

Supporting both of these options also means that we'll be essentially designing four separate skid buffer designs, depending upon the options chosen. These options will be aintermingled throughout the design to control logic generation as well. Therefore, when it comes to verification we'll also need to make certain that we verify all four of the designs found within this code.

Verilog code

As is my normal practice, I'll skip most of the introductory code, and jump right into the skid buffer example itself.

```
module skidbuffer(i_clk, i_reset,
                i_valid, o_ready, i_data,
                o_valid, i_ready, o_data);
        parameter       [0:0]   OPT_LOWPOWER = 0;
        parameter       [0:0]   OPT_OUTREG = 1;
```

Or perhaps we'll just mostly skip this introductory code.

I would like to point out that both `OPT_LOWPOWER` and `OPT_OUTREG` are single bit parameters, making true/false logical tests simpler and keeping us from accidentally trying to set a value to something other than true (1) or false (0).

```
        parameter               DW = 8;
        input   wire                    i_clk, i_reset;
        input   wire                    i_valid;
        output  reg                     o_ready;
        input   wire    [DW-1:0]        i_data;
        output  reg                     o_valid;
        input   wire                    i_ready;
        output  reg     [DW-1:0]        o_data;

        //
        // We'll start with skid buffer itself
        //
        reg                     r_valid;
        reg     [DW-1:0]        r_data;
```

The internal skid buffer itself is captured by two signals. The first, `r_valid`, just indicates that the internal buffer has valid data within it. This is shown in Fig. 3 above, which shows data going into the internal buffer, as well as in Fig. 4 above, which shows data coming out of the internal buffer.

The basic logic for this `r_valid` signal is that we want it to go high any time there's a valid incoming signal but the outgoing path is stalled.

```
	initial		r_valid = 0;
	always @(posedge i_clk)
	if (i_reset)
		r_valid <= 0;
	else if ((i_valid && o_ready) && (o_valid && !i_ready))
		// We have incoming data, but the output is stalled
		r_valid <= 1;
```

We can then return to normal operation once the incoming ready signal returns to normal, acting again as a pass through device.

```
	else if (i_ready)
		r_valid <= 0;
```

The data logic is even simpler. Any time the outgoing combinatorial side is ready, we'll just quietly copy the incoming value into our buffer.

```
	always @(posedge i_clk)
	if (o_ready)
		r_data <= i_data;
```

However, the logic above doesn't preserve our `OPT_LOWPOWER` property, shown below, that `r_data` should be zero whenever `r_valid` is false.

```
	assert property (@(posedge i_clk)
		!r_valid |-> r_data == 0);
```

To implement this low power logic, we'll need to make certain that `r_data` starts at zero. Not only that, but any time the design is reset then `r_valid` will be reset to zero, and so we'll need to set `r_data` to zero in both of those cases–but only if `OPT_LOWPOWER` is set.

```
	initial		r_data = 0;
	always @(posedge i_clk)
	if (OPT_LOWPOWER && i_reset)
		r_data <= 0;
```

This also means that any time the outgoing side *isn't* stalled we'll need to hold `r_data` at zero as well.

```
	else if (OPT_LOWPOWER && (!o_valid || i_ready))
		r_data <= 0;
```

Finally, we can copy the data any time the outgoing/upstream side isn't stalled, just like before.

```
	else if ((!OPT_LOWPOWER || i_valid) && o_ready)
		r_data <= i_data;
```

Or rather, we can't because that's not quite right. If we are in both `OPT_LOWPOWER` mode, and we are registering our output, then we need to make certain we *only* set this value when `i_valid` is true. Otherwise if `OPT_LOWPOWER` is true, the input and `r_valid` properties would force the output to be zero.

While I could write this as,

```
	else if ((!(OPT_LOWPOWER && OPT_OUTREG) || i_valid) && o_ready)
```

I prefer expanding the logic out using De Morgan's laws. The condition below, therefore, captures the same logic.

```
	else if ((!OPT_LOWPOWER || !OPT_OUTREG || i_valid) && o_ready)
		r_data <= i_data;
```

There's also one very profound key feature to a skid buffer implementation that I missed for the first several years I used them: the output stall signal is given by the internal buffer's valid signal. The two are signals completely equivalent. Ok, I'll admit I didn't believe it myself until I ran the formal proof, but that's beside the point. In this case, since we are using AXI READY/VALID notation, this means that the outgoing READY (not stalled) signal is the opposite of our VALID signal.

```
	always @(*)
		o_ready = !r_valid;
```

I built and implemented many skid buffers before realizing this. Even once I first saw this equivalency, it still took some time (and formal proofs) to believe it. That said, it nicely simplifies any implementation.

Now that we've dealt with the internal buffer, we can move on to the outgoing interface. We'll need to split this logic into two sections, though: One section for the simpler case where the outgoing registers are not buffered, and another for the case where they are.

```
	generate if (!OPT_OUTREG)
	begin
```

In the unregistered case, we'll want our output port to be valid any time either the input port is valid, or if there's data in our skid buffer.

```
		always @(*)
			o_valid = (i_valid || r_valid);
```

This is also the combinatorial side of the interface, so you may note the use of the `always @(*)`.

As for our output data, we'll want that to come from the buffer any time the buffer is active, or be a pass through otherwise.

```
		always @(*)
		if (r_valid)
			o_data = r_data;
		else
			o_data = i_data;
```

Well, almost. What if the incoming `i_data` didn't observe the low-power property? In that case we'd need to only set `o_data` to the incoming `i_data` value if `i_valid` were also set, otherwise we'd want to force the output to be zero.

```
		always @(*)
		if (r_valid)
			o_data = r_data;
		else if (!OPT_LOWPOWER || i_valid)
			o_data = i_data;
		else
			o_data = 0;
```

Otherwise the outgoing interface logic seems simple enough. But what about the case where we register the outgoing data?

That one is just a touch trickier.

Perhaps the valid line isn't any different, save for the reality that it can be reset.

```
	end else begin

		initial	o_valid = 0;
		always @(posedge i_clk)
		if (i_reset)
			o_valid <= 0;
		else if (!o_valid || i_ready)
			o_valid <= (i_valid || r_valid);
```

That said, I've been burned by this kind of logic before, so I've gotten to the point where I always build it with the structure above. Notice the `if (!o_valid || i_ready)` condition. This is the piece that's caught me up a couple of times. It's basically the same as saying `if (!(o_valid && !i_ready))` but rewritten using De Morgan's laws, and so it describes any time the outgoing interface is not stalled.

My problem is that I keep wanting to add other logic to channels like this, much like we discussed in the article about the most common AXI mistake.

Be forwarned: if you play with registered signals using this basic handshake, you will want to use this pattern and no more! How do I know this? Because every time I do something different, the formal tools correct me. This seems to be the only valid approach to signals subject to the rules of this type of handshake.

Does that mean that this format will apply to the `o_data` signal as well? Absolutely!

We start with resetting `o_data` anytime `OPT_LOWPOWER` is set, and then we refuse any further logic if the output is stalled.

```
		initial		o_data = 0;
		always @(posedge i_clk)
		if (OPT_LOWPOWER && i_reset)
			o_data <= 0;
		else if (!o_valid || i_ready)
		begin
```

The key to making this work in a registered context is now found within the next tidbit of logic. First, if there's something in the buffer, then that needs to move to the output port. If not, but if something is coming in on the input port, then we'll set to that output instead.

```
			if (r_valid)
				o_data <= r_data;
			else if (!OPT_LOWPOWER || i_valid)
				o_data <= i_data;
			else
				o_data <= 0;
		end

	end endgenerate
```

As before, though, there's an optimization we can take, but not if we are in the `OPT_LOWPOWER` mode.

Before I leave this topic, notice the key feature of how I've used `OPT_LOWPOWER`: If it *isn't* set, then all of the `OPT_LOWPOWER` logic (save the initial statement) just goes away. Since `OPT_LOWPOWER` is a constant, the synthesizer can handle optimizing this logic away if `OPT_LOWPOWER` is ever clear. The same is also basically true of `OPT_OUTREG`, but there's more going on with that signal.

That's all there is to the implementation of a skid buffer. As you can see, the logic is *really* simple, and there are really only two internal registers associated with it: `r_valid` and `r_data`. If the outputs are registered as well, then `o_valid` and `o_data` will also be registered.

That leads us to the next step: proving that this implementation works and that it does what it is supposed to.

Formal Verification

We've already seen several of the formal properties above. I'll repeat those again below in a moment. For now, let's start with the reset properties.

Following any reset, all of the valid lines need to be cleared. We can assume this of our input signal.

```
`ifdef FORMAL
	// Reset properties
	property RESET_CLEARS_IVALID;
		@(posedge i_clk) i_reset |=> !i_valid;
	endproperty
```

In this case, I declared this as a named property–a feature of the SystemVerilog Assertion language. I'll come back to this in a moment and either assert or assume it.

We'll also want to assume that any time the incoming interface is stalled, that is any time there's valid data at the input but `o_ready` is low, the valid signal needs to continue into the next clock cycle and the data isn't allowed to change.

```
	property IDATA_HELD_WHEN_NOT_READY;
		@(posedge i_clk) disable iff (i_reset)
		i_valid && !o_ready |=> i_valid && $stable(i_data);
	endproperty
```

Now here's why I'm declaring these as named properties: when I went to verify my AXI slave core using this buffer, I realized that the assumptions might void the proof. They had to be converted to assertions for that proof, while left as assumptions within.

To handle this, I created a `SKIDBUFFER` macro to determine if the properties should be assumed or asserted. Using this macro, I can choose to assume or assert as necessary.

```
`ifdef SKIDBUFFER
	assume property (RESET_CLEARS_IVALID);
	assume property (IDATA_HELD_WHEN_NOT_READY);
`else
	assert RESET_CLEARS_IVALID;
```

```
		assert  IDATA_HELD_WHEN_NOT_READY;
	`endif
```

Those are the only two assumptions describing the incoming interface.

On the outgoing side, we'll quickly repeat the reset property: following any reset, both valid signals need to be cleared.

```
		assert property (@(posedge i_clk)
			i_reset |=> !r_valid && !o_valid);
```

We can now start walking through both our internal and output signals.

The big rule we want to preserve is that any time there's an outstanding request on the output port that's stalled, i.e. `o_valid && !i_ready`, then the request must remain the same on the next clock.

```
		// Rule #1:
		//	Once o_valid goes high, the data cannot change until the
		//	clock after i_ready
		assert property (@(posedge i_clk)
			disable iff (i_reset)
			o_valid && !i_ready
			|=> (o_valid && $stable(o_data)));
```

The `disable iff (i_reset)` just means that we won't check this test if the reset is ever high. Personally, I think this goes without saying, however, the formal tools have been known to disagree with me from time to time.

The second rule tries to capture the "no data shall be dropped" policy. Specifically, if there's data on the incoming port, then it either needs to go to the output or it needs to be buffered.

```
		// Rule #2:
		//	All incoming data must either go directly to the
		//	output port, or into the skid buffer
		assert property (@(posedge i_clk)
			disable iff (i_reset)
			(i_valid && o_ready
				&& (!OPT_OUTREG || o_valid) && !i_ready)
				|=> (r_valid && r_data == $past(i_data)));
```

What about the other cases? Well, if either `!i_valid` or `i_valid && !o_ready`, then nothing happens on the input port that we need to worry about. Since `r_valid` is equivalent to `!o_ready`, we know that the only interesting case is the one in which `r_valid` is low. If `r_valid` is low and `i_ready` is high, the core is a simple pass through and a quick code inspection will prove that works. That leaves the case where `r_valid` is low and `i_ready` is also low–the case we covered above.

This doesn't quite capture everything, though. We've now discussed how information should flow through this design, but not how the design should return to idle. That's important, and I've been burned by not checking the return to idle before. Hence we want to make certain that the design will return to idle.

So any time `i_ready` is true on the outgoing interface then everything should be cleared. On the next clock, `o_valid` should only be true if `i_valid` is also true.

```
	// Rule #3:
	//	After the last transaction, o_valid should become idle
	generate if (!OPT_OUTREG)
	begin

		assert property (@(posedge i_clk)
			disable iff (i_reset)
			i_ready |=> (o_valid == i_valid));
```

But what if we are registering the ports on our outgoing interface?

In that case two rules shall apply. First, any time an input is accepted, then `o_valid` should be high on the next clock.

```
	end else begin

		assert property (@(posedge i_clk)
			disable iff (i_reset)
			i_valid && o_ready |=> o_valid);
```

Second, any time `i_ready` is true and there's nothing on either the input or in the buffer, then `o_valid` should clear on the next clock.

```
		assert property (@(posedge i_clk)
			disable iff (i_reset)
			!i_valid && !r_valid && i_ready |=> !o_valid);

	end endgenerate
```

That checks both the rise and the fall of `o_valid`. Seems simple enough.

But what about `r_valid`?

Well, if `r_valid` is ever true while the outgoing port is `i_ready`, then the skid buffer gets copied to the outgoing port and `r_valid` must be deasserted on the next clock. This is the case from Fig. 4 above.

```
	// Rule #4
	//	Same thing, but this time for r_valid
	assert property (@(posedge i_clk)
		r_valid && i_ready |=> !r_valid);
```

What if something was also coming in on the incoming interface? It won't happen. Remember, `o_ready = !r_valid`. Therefore if `r_valid` is high, the incoming interface is stalled and so we can ignore it.

That leaves the two special low power properties that we discussed above. We only want to enforce those if `OPT_LOWPOWER` is set, and we want to ignore them otherwise. Therefore, we'll use a generate block to capture these checks. That means that if `OPT_LOWPOWER` isn't set, the synthesis tool (i.e. yosys) won't even create the logic to support these checks.

```
	generate if (OPT_LOWPOWER)
	begin
		//
		// If OPT_LOWPOWER is set, o_data and r_data both need
		// to be zero any time !o_valid or !r_valid respectively
		assert property (@(posedge i_clk)
			!o_valid |-> o_data == 0);

		assert property (@(posedge i_clk)
			!r_valid |-> r_data == 0);

		// else
		//	if OPT_LOWPOWER isn't set, we can lower our logic
		//	count by not forcing these values to zero.

	end endgenerate
```

Those are all of the properties we need to know this works, but does it really work?

For that, we'll turn to cover.

Cover

Unlike the safety (assert/assume) properties above which can be proved true if no trace can be found that makes an assert false while keeping all of the assumes true, cover only succeeds if at least one trace can be found. Cover is very useful for finding faults in your assumptions, proving that particular operations can take place and more.

It's also very valuable when you just want a trace showing that the design works.

So let's build such a trace that starts and ends with the core idle. In the middle, we'll insist that the `i_ready` line toggle from high to low twice before settling back on high.

Oh, and did I mention that we only want to check cover if this unit is verified in isolation? Otherwise it may be that the parent module doesn't ever make this `cover()` statement true –something that wouldn't be a fault.

```
`ifdef SKIDBUFFER
	reg	f_changed_data;

	// Cover test
	cover property (@(posedge i_clk)
		disable iff (i_reset)
		(!o_valid && !i_valid)
		##1 i_valid &&  i_ready [*3]
		##1 i_valid && !i_ready
		##1 i_valid &&  i_ready [*2]
		##1 i_valid && !i_ready [*2]
		##1 i_valid &&  i_ready [*3]
		// Wait for the design to clear
		##1 o_valid && i_ready [*0:5]
		##1 (!o_valid && !i_valid && f_changed_data));
```

The resulting trace is interesting, but it could be better. In particular, the incoming data is all zeros. While that's valid, it's not very revealing. I'd rather be able to "see" from the trace that the various data lines were properly progressing. Perhaps if we cound insist that the incoming data be a counter?

The easy way to do this is to add another register, and some logic associated with it. Let's call this `f_changed_data`, and use it to indicate that our data "properly" changes throughout the cover trace. That is, `f_changed_data` will capture if the incoming data counts up–just so we can visualize the trace and what is going on easier.

In many ways, this isn't your typical "formal" property. It doesn't use any of the formal language features (except `$past()`). However, if the whole world starts to look like a Verilog problem, then the solution is easily a simple piece of Verilog logic.

We'll start by setting this `f_changed_data` flag to true, and the clear the flag if any rule of what we want to see for our cover statement changes.

```
	initial		f_changed_data = 0;
	always @(posedge i_clk)
	if (i_reset)
		f_changed_data <= 1;
```

`i_data` is only allowed to change on the clock following `!i_valid || o_ready`.

```
        else if (i_valid && $past(!i_valid || o_ready))
        begin
```

In this case, we’ll just clear `f_changed_data` any time `i_data` doesn’t increment.

```
                if (i_data != $past(i_data + 1))
                        f_changed_data <= 0;
```

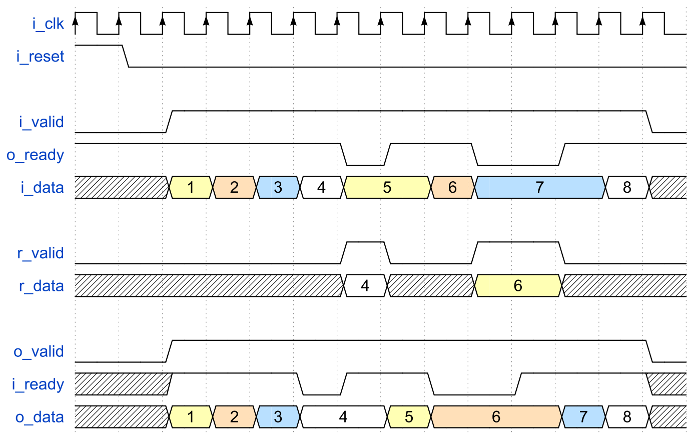
Similarly, we want clear this value any time the input hasn’t been valid constantly.

```
        end else if (!i_valid && i_data != 0)
                f_changed_data <= 0;
```

Voila! We now have a wonderful trace in Fig. 6 showing how this core works.

Yes, this trace has been edited, but only minimally.

Fig 6. A cover trace from this skid buffer



SymbiYosys Script

In most cases the SymbiYosys script used to drive a proof like this is very basic. In this case, however, there are a couple of key features of the script that are worth discussing.

First, you may recall above that I mentioned we would need to do four separate proofs in order to try all of the combinations of our `OPT_LOWLOGIC` and `OPT_OUTREG` parameters. Each of these can be separated into its own proof using the `[tasks]` section of the SymbiYosys configuration file. We’ll also create a single task for our cover proof.

```
[tasks]
prfc prf
prfo prf               opt_outreg
lpc  prf opt_lowpower
lpo  prf opt_lowpower opt_outreg
cvr
```

If you’ve never seen a `[tasks]` section of a SymbiYosys file before, then you are in for a treat. Each line within this section defines a separate formal run. The first identifier on the line gives the name of the run, and the subsequent identifiers are tags that are then applied to the run and useful when configuring it.

The second section, the `[options]` section, shows the first reason to use tasks: All of our induction proofs can be accomplished within 3 steps, whereas the cover pass will take 20. Here you'll notice that every task above had either the `prf` tag or the `cvr` name (also a tag). This allows us to set different depths for each of the passes.

```
[options]
prf: mode prove
prf: depth 3
cvr: mode cover
cvr: depth 20
```

This design is simple enough that it doesn't really matter what engine we use, so we'll use the default.

```
[engines]
smtbmc
```

The real action, however, is in the `[script]` section. This section contains a series of commands to be given to yosys to control how the design is processed. That also means that you can use the yosys `help` command if you come across anything confusing.

We'll start the script off by defining the `SKIDBUFFER` macro, and then reading our code into yosys.

```
[script]
read -define SKIDBUFFER
read -formal skidbuffer.v
```

Often my designs consist of many parts. For these designs, I'd spend one line here reading in each of the input files.

The next step, however, is to control the parameters externally. yosys has a new extension to the `hierarchy` command to make this easier. Basically, the `hierarchy` command finds the top level module and instantiates all of the logic below it. In our case, we want to instantiate specific logic depending upon the proof. Therefore we are going to use the `chparam` option to `hierarchy` to set these parameters.

This is different from the approach I discussed before. Before, I would have written,

```
opt_outreg:    chparam -set OPT_OUTREG 1   skidbuffer
~opt_outreg:   chparam -set OPT_OUTREG 0   skidbuffer
opt_lowpower:  chparam -set OPT_LOWPOWER 1 skidbuffer
~opt_lowpower: chparam -set OPT_LOWPOWER 0 skidbuffer
```

This usage had some problems, and so it has now been deprecated. One of those problems was that yosys would re-elaborate the module with every call to `chparam`. Any elaboration errors due to incompatible parameter settings along the way might cause the whole process to halt.

Instead, using the `hierarchy` command, every parameter is set at once. Hence we might want to use,

```
prfc: hierarchy -top skidbuffer -chparam OPT_LOWPOWER 0 -chparam OPT_OUTREG 0
prfo: hierarchy -top skidbuffer -chparam OPT_LOWPOWER 0 -chparam OPT_OUTREG 1
lpc:  hierarchy -top skidbuffer -chparam OPT_LOWPOWER 1 -chparam OPT_OUTREG 0
lpo:  hierarchy -top skidbuffer -chparam OPT_LOWPOWER 1 -chparam OPT_OUTREG 1
```

The problem with this approach is simple: what if you have twenty different tasks, all with different repeats of the same options?

In this case, SymbiYosys' python interface can come to our rescue.

The following script will check our two parameters independently, and create a string variable, called `cmd`, containing the `hierarchy` line with the appropriate values in it. Then, when the `output(cmd);` call is issued at the end, the `cmd` string will be written into the individual yosys scripts driving each of their respective proofs.

```
--pycode-begin--
cmd = "hierarchy -top skidbuffer"
cmd += " -chparam OPT_LOWPOWER %d" % (1 if "opt_lowpower" in tags else 0)
cmd += " -chparam OPT_OUTREG   %d" % (1 if "opt_outreg"   in tags else 0)
output(cmd);
--pycode-end--
prep -top skidbuffer
```

The final `[files]` section is fairly unremarkable. It just lists the files used in this proof. In this case, it is only the skidbuffer.v file.

```
[files]
skidbuffer.v
```

SymbiYosys will copy this file into its processing directory prior to running the proof.

The whole proof can now be run using,

```
% sby -f skidbuffer.sby
```

Alternatively, you can integrate it into your formal verification Makefile, and just run

```
% make
```

Feel free to take a loot at the Makefile I'm using for these AXI projects should you need an example.

Conclusion

Skid buffers are very powerful, and very useful, especially when using AXI. Indeed, I'm using skid buffers throughout almost all of my various AXI designs. Over and over it's the same logic, so it makes sense to create a single file to capture this logic and simplify the design. They really are just that useful.

I just wish I'd separated this logic into its own module earlier, since now I have many, many copies of the same logic that need to be maintained. To this end, I'd like to thank Eric LaForest for setting a better example for me to follow. I would also commend his blog article on skid buffers to you for further reading.

For those of you who are not familiar with the SystemVerilog's concurrent assertion language, or who do not have access to the commercial SymbioticEDA Suite, you might find this discussion of alternatives and equivalents to concurrent assertions valuable.

The desire accomplished is sweet to the soul: but it is abomination to fools to depart from evil. (Prov 13:19)
