Crossing clock domains with an Asynchronous FIFO

Jul 6, 2018

My first VGA video project was for a Basys3 board, following Digilent's instructions. The system ran at 100MHz with a 25MHz pixel clock that I could create by dividing the 100MHz clock down in logic, rather than using a PLL. In that design, I could also divide the 100MHz clock by two in logic to reference flash, and so I had a flash controller running from a 50MHz clock. It was a complex design, partly because in order to get enough bandwidth from flash to video I needed to compress the video images on the flash device. In the end, though, everything ran off of a single 100MHz clock–in spite of the various rates moving through the board.

If only things stayed that easy.

Since then, I've worked on a video project having a 148.5MHz pixel rate, but where the SDRAM memory controller wanted a 100MHz clock. Moving the pixels from the memory clock to the video and back again was a challenge that I never got past–and part of the reason why the design doesn't quite work yet.

I've also wanted to work on an I2S audio system–also on Digilent's Nexys Video board. Like the 148.5MHz video system, the audio system on that board wants a clock that isn't an easy logical division of 100MHz: it wants a 49.152MHz clock.

Both of these designs required having a data stream generated in one clock domain, but consumed in another.

Sure, I tried solving the problem using the techniques I'd discussed earlier in my clock domain crossing article, but the results … never really worked. When passing streaming data around, the approaches described in that article just weren't up to the task. They were the wrong solution for the job. A much more appropriate solution would've been an asynchronous FIFO.

My journey with asynchronous FIFOs started out with the initial belief that they were a wizards concept that I just didn't understand. Then I found a paper by Cliff Cummings on the topic. Not only did this paper describe what an asynchronous FIFO was, but it also made some amazing claims about the FIFO's performance that weren't apparent to me as I examined his code. In particular, I wasn't convinced that his implementation of an asynchronous FIFO wouldn't overflow, wouldn't underflow, or even that the asynchronous reset's would work properly.

At one time I even sat down with Cummings and asked him about his implementation. After chatting together, we both agreed the design might be an ideal design to verify using formal techniques.

So let me dedicate this article to him.

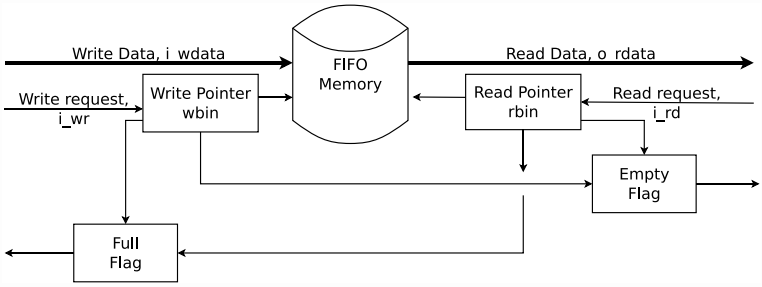
This article is also a gateway article to other articles discussing systems that require clock domain crossings–such as presenting the video simulator I posted on github, and what makes it special.

Since the FIFO we'll be discussing today is asynchronous, I would recommend you first read our discussion of properties associated with an asynchronous reset, as well as the example of the asynchronous clock switch. These will give you a bit of background regarding how we might handle multiple clocks while at the same time working through a design formally.

Basic FIFO

If you've never wrestled with the concept of an asynchronous FIFO before, you might ask yourself what the big deal is? Indeed, in many ways an asynchronous FIFO is just like any other FIFO.

Fig 1. A Basic synchronous FIFO



Let's compare the two FIFOs with each other. Both a synchronous and asynchronous FIFOs have a write pointer. We'll call this `wbin`. Then, on any write, we'll increase this pointer by one–but only if the FIFO isn't already FULL.

```
initial wbin = 0;
always @(posedge i_wclk or negedge i_wrst_n)
if (!i_wrst_n)
 wbin <= 0;
else ((i_wr)&&(!o_wfull))
 wbin <= wbin + 1;
```

Did you notice that this logic takes place using the write clock domain, `i_wclk`? Or that the reset was a negative edge driven asynchronous reset in the write clock domain as well? You'd expect this from an asynchronous FIFO. The write pointer logic within a synchronous FIFO would be the same except that only one clock would be used, likely with a synchronous reset as well.

Cliff Cummings' FIFO is just subtly different from my own earlier presentation of a synchronous FIFO: his FIFO holds a full `2^N` elements. The FIFO I presented earlier only holds `(2^N)-1` elements. I like the

difference, and will probably upgrade my own synchronous FIFO implementations to follow this lead as well. This changes `wbin` slightly–we'll now use `N+1` address bits in `wbin` to hold a pointer into a `2^N` element FIFO.

Moving on with our comparison between asynchronous and synchronous FIFOs, both of them will need to place the incoming data into a memory (block RAM) on any write.

```
always @(posedge i_wclk)
if ((i_wr)&&(!o_wfull))
 mem[wbin[AW-1:0]] <= i_wdata;
```

In the case of the asynchronous FIFO, this is also done specifically using a clock associated with the write channel, `i_wclk`. I've also used `AW` to reflect the *address width* of this memory. I'll probably still refer to this as `N` throughout in this text.

The read logic is very similar to the write logic above. The logic starts by adjusting a read address pointer which we'll call `rbin`. Like `wbin` above, this has one more bit than necessary to actually address a value within the buffer–hence it has `N+1` bits to access `2^N` data points. In a fashion similar to the write pointer, this pointer also needs to increment: anytime there's a read request and the buffer isn't empty.

```
initial rbin = 0;
always @(posedge i_rclk or negedge i_rrst_n)
if (!i_rrst_n)
 rbin <= 0;
else if ((i_rd)&&(!o_rempty))
 rbin <= rbin + 1;
```

As a final step, we'll read from the memory and return the result.

```
assign o_rdata = mem[rbin[AW-1:0]];
```

Voila! That's the basics of any FIFO–synchronous or asynchronous. How much harder can it be?

Much.

As with most digital design problems, the devil lies buried in the details. In this case, look a little closer at the two flags, `o_wfull` indicating that the FIFO is full and `o_rempty` indicating that it is empty.

As a first attempt to calculate these, we might express them with combinatorial logic, as in:

```
// The FIFO is empty when both read and write pointers point to the
// same location.
assign o_rempty = (wbin == rbin);

// It is full when wbin-rbin = 2^N.  In that case, the bottom AW
// address bits are identical, but the top bit is different.
assign o_wfull  = (wbin[AW] != rbin[AW])
   && (wbin[AW-1:0]==rbin[AW-1:0]);
```

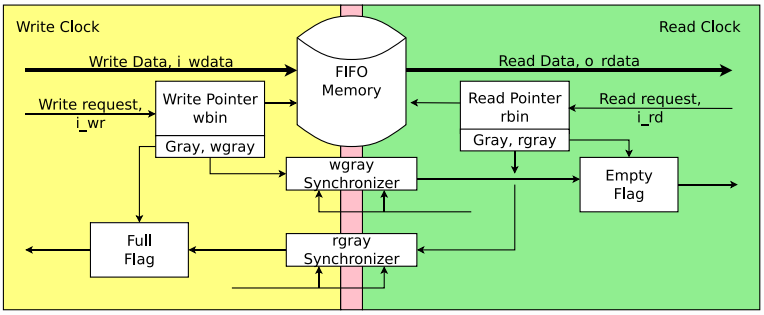
Herein lies the problem.

For a synchronous FIFO, both `AW+1` bit pointers are generated on the same clock, so there isn't an immediately apparent problem. Sure, you might adjust this logic so the `o_rempty` and `o_wfull` flags are registered, but they'll still have these same basic values.

The big problem with these two pointers is specific to any *asynchronous* FIFO design. In an asynchronous design, the read pointer is kept in the read clock domain and the write pointer in a separate write clock domain. Calculating `o_rempty` or `o_wfull` requires crossing clock domains. This invites problems with metastability, where a design might fine work in simulation but not on actual hardware. Indeed, it might work 95% of the time on actual hardware, leaving behind incomprehensible results when it doesn't work.

I've tried to illustrate this problem in Fig 2 below. In this figure, I colored the background based upon which part of the design lies within each clock domain, whether yellow for the write clock, or green for the read clock.

Fig 2. In an Asynchronous FIFO, the pointers need to cross clock domains

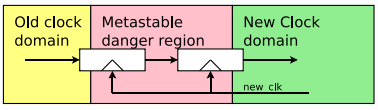


Fixing these two flags is really the focus of how to build an asynchronous FIFO. To do so, we'll build off of our previous work using 2FF or 3FF synchronizers, but this time we'll need to introduce Gray codes as well. Each of these concepts is shown in Fig 2 above.

Gray Codes

When we last discussed clock domain crossings, we shows how it was possible to use a two or a three clock synchronizer to pass a one-bit value from one clock domain to the next. Fig 3 shows an example of this, using two flip flops clocked in the new domain–this would be a two clock synchronizer.

Fig 3. A two clock synchronizer



The code for this operation is shown below.

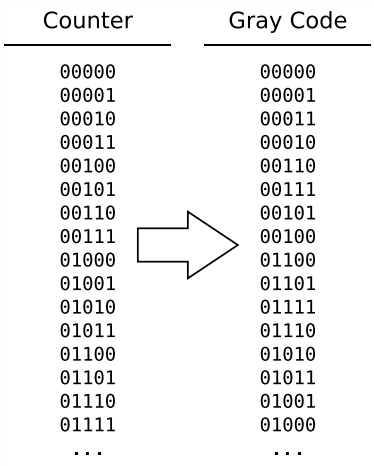
```
always @(posedge new_clock)
```

```
  resynchronized_value <= { unstable_values, unsynchronized_input };
```

In the case of this one bit difference, it doesn't really matter if that one bit change arrives one clock earlier or one clock later–it's just a slow signal with no dependencies.

Sadly, we're not trying to cross a 1-bit signal from one clock domain to another, but rather an `N` bit (i.e. `AW`-bit) signal from one clock domain to the next–whether `wbin` to the read clock side or `rbin` to the write clock side. If we put the whole word into a synchronizer, like the one shown in Fig 3 but with more bits, then the outputs wouldn't suffer from metastability anymore, but they might not be stable anymore either. For example, if `rbin` were all ones and transitioning to all zeros, then some uncontrolled random number of ones might be set at the output of the synchronizer–depending upon how the bits were routed, and thus upon which arrived before the new clock signal and which arrived afterwards. As an example, an `8'hff` transitioning to `8'h00` might be read as `8'h52` (among many, many other possibilities). This is unacceptable.

Fig 4. Counting in Gray Code




```
gray_value = (counter>>1)^counter;
```

The solution is to pass the address from one clock domain to another in a form chosen so that only one bit will ever change at any time. Formally, we can describe such a `word` with the requirement that the difference between any `word` and the `$past()` version of it must be no more than one bit.

```
always @(posedge first_clk)
 word <= // Logic to create the next value;

// Verify that it has no more than one bit difference
always @(posedge first_clk)
 assert((word == $past(word))
  ||($onehot(word ^ $past(word))));
```

`$onehot` is a SystemVerilog function that returns `1` (true) if and only if one bit is set within its argument, and zero otherwise.

Sadly, the open source version of Yosys doesn't (yet) understand `$onehot`. (The commercial version does.) We'll alternatively express this within a loop.

Specifically, if any single bit changes between the new and old values, then all other values must be the same.

```
genvar k;
generate for (k=0; k <= AW; k=k+1)
begin : CHECK_ONEHOT
 always @($global_clock)
  assert(word[k] == $past(word[k])
   ||(word ^ $past(word) ^ (1<<k) == 0));
end
```

Okay, so this is our criteria for success, but what sort of encoding meets this criteria? Gray coding!

If we have a counter such as,

```
always @(posedge i_clk)
 counter <= counter + 1;
```

then a Gray coded version of that same counter will have the property that only one bit changes at a time. We can create this Gray coded by exclusively `OR`ing the counter with itself shifted down by one.

```
assign graycounter = counter ^ (counter >> 1);
```

Using such a Gray coded counter, we can cross clock domains with both read and write address pointers. Not only will the results be stable and so not suffer from metastability, but they will also capture all of the information within the address in an fashion that won't be corrupted if any given bit within the word arrives earlier or later.

This means we'll add Gray coded pointers to our FIFO design in order to bridge them across the clock domain divide. First, the read pointer, now Gray coded and represented as `rgray`, will cross to the write clock domain.

```
initial { wq2_rgray,  wq1_rgray } = 0;
always @(posedge i_wclk or negedge i_wrst_n)
if (!i_wrst_n)
 { wq2_rgray, wq1_rgray } <= 0;
else
 { wq2_rgray, wq1_rgray } <= { wq1_rgray, rgray };
```

This is just an `N`-bit wide, two flip-flop synchronizer, such as we introduced earlier and diagrammed in Fig 3 above.

Second, the write pointer, Gray coded as `wgray`, will cross from the write to the read clock domain.

```
initial { rq2_wgray,  rq1_wgray } = 0;
always @(posedge i_rclk or negedge i_rrst_n)
if (!i_rrst_n)
 { rq2_wgray, rq1_wgray } <= 0;
else
 { rq2_wgray, rq1_wgray } <= { rq1_wgray, wgray };
```

This brings our two pointers, `rbin` and `wbin`, into the other clock domain as `wq2_rgray` and `rq2_wgray`, but these values are no longer counters. How shall we use them in our comparisons? Do we need to convert them back to more traditional counters?

Comparing Gray coded pointers

Let's examine how we might use these two Gray pointers. Remember, we need to determine when the FIFO is empty and when it is full. Before, we had the two lines,

```
assign o_rempty = (wbin == rbin);
assign o_wfull  = (wbin[AW] != rbin[AW])
   && (wbin[AW-1:0]==rbin[AW-1:0]);
```

Perhaps you noticed the `r` in `o_rempty`, or the `w` in `o_wfull`. These are from Cummings' notation, and used to remind the reader that `o_rempty` is calculated in the read clock domain, whereas `o_wfull` is calculated in the write clock domain.

If you examine the Gray coded counters illustrated in Fig 4, you'll notice that the Gray coded values are unique–just like the counters they represent. Further, an `N` bit counter can be represented with an `N` bit Gray code. In other words, if you want to check whether or not two pointers are identical, you only need to check whether the two Gray coded pointers are identical.

This works great for testing whether or not the FIFO is empty.

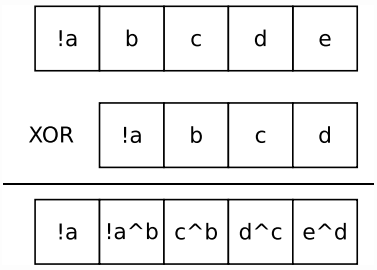
```
assign o_rempty = (rq2_wgray == rgray);
```

But how shall we test if the FIFO is full?

In this case, the math is more complicated. We want to test whether or not `wbin - rbin == 2^N`. If this is true, then the FIFO is full. Notice that any time this comparison is true, the bottom `N-1` bits will be constant between `wbin` and `rbin`, and the top bit will be flipped.

To see how this comparison changes once converted to Gray code, consider the example of `16` element FIFO. Such a FIFO will require 5-bit read and write pointers. We'll allow 5-bits of the read pointer to have the arbitrary value, `{a, b, c, d, e}`. When the FIFO is full, the associated write pointer will be `{ !a, b, c, d, e }`. Now, consider what will happen to this pointer when converted to Gray code, as shown in Fig 5 for the write pointer.

Fig 5. If the top bit of an address changes, then both the top two bits of the Gray coded address change



Remember how the only bit that differed for the write pointer was the most significant bit. From Fig 5, you can see that all but the top two bits will be identical between the read and write pointers following the conversion to Gray code. Hence, we can test whether the two pointers are identical in all but their top bits by testing whether the top two bits are opposites, but the rest of the bits are identical.

```
assign o_wfull  = (wgray[AW:AW-1] == ~wq2_rgray[AW:AW-1])
   && (wgray[AW-2:0]==wq2_rgray[AW-2:0]);
```

The solution Cummings' presents is almost identical. The difference is that he creates `o_rempty` and `o_wfull` using clocked logic instead of combinatorial for better performance. As a result, his solution looks more like,

```
always @(posedge i_rclk or negedge i_rrst_n)
if (!i_rrst_n)
 o_rempty <= 1'b0;
else
 o_rempty <= (rq2_wgray == rgray);
```

and

```
always @(posedge i_wclk or negedge i_wrst_n)
if (!i_wrst_n)
 o_wfull <= 1'b0;
else
 o_wfull <= (wgray[AW:AW-1] == ~wq2_rgray[AW:AW-1])
   && (wgray[AW-2:0]==wq2_rgray[AW-2:0]);
```

However, at this point in the development … I got lost. Sure, these pointers are now Gray coded and so they'll pass from one clock domain to another without problems, but how shall I get some confidence that this design actually works now that it is so different from the synchronous FIFO I am familiar with? I mean, what confidence can I have that these two flags, already delayed by two clocks, are going to keep the FIFO from underruns or overruns? Cummings confidently declares that his solution works, citing the works of those who have proved these properties. However, even after I read his document, I remained unconvinced, and not certain of whether or not I wanted to trust my professional designs to his hand waving. (Sorry Cliff!)

What I'd like is to have the confidence that can only come from some form of *formal proof* that this whole thing works–even after crossing clock domains.

Therefore, let's start looking at the components necessary to formally verify this design.

Proof Outline

Let's consider what we might wish to prove.

1. We'd like to make certain that the FIFO pointers are "within bounds" at all times. Since we used `AW+1` (address width) bits to represent a `2^(AW)` element FIFO, we'll need to measure the fill at all times and make certain it remains within `0` and `2^(AW)`.

One of the neat things about formal methods is that the variables and tests used within the formal section need not be synthesizable, neither are they susceptible to metastability. As a result, we can violate all principles of good synthesizable design, and we can measure the actual FIFO fill directly at any instant (well, at any formal timestep).

This fill should start out at zero, and it should always be less than or equal to the total amount of space in the FIFO.

```
wire [AW-1:0] f_fill;
assign f_fill = (wbin - rbin);

initial assert(f_fill == 0);
always @(*)
 assert(f_fill <= { 1'b1, {(AW){1'b0}} });
```

2. Looking at the empty flag, we want to assert that any time the FIFO is actually empty, that the `o_rempty` flag is also high.

```
always @(*)
 if (f_fill == 0)
  assert(o_rempty);
```

3. We'll want to do the same thing for `o_wfull`: any time the FIFO is truly full, we'll want to assert that `o_wfull` is high.

```
always @(*)
 if (f_fill == { 1'b1, {(AW){1'b0}}})
  assert(o_rempty);
```

4. Just to make certain that these values won't get stuck, let's also use a `cover` property to cover going from not-empty to empty, and from full to not full.

5. Finally, the classic means of proving that a FIFO works is to accept two arbitrary values into the FIFO, at an arbitrary but sequential pair of locations, and then to verify that those same two values may be read out of the FIFO some time later.

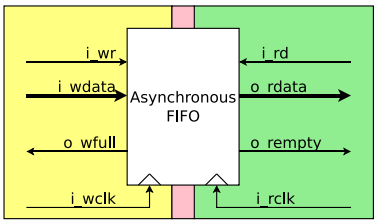
   Let's add that to our proof requirements as well.

This is only a high level view of the formal properties we'd like to use. We'll add some other properties below as well–if for no other reason than to make certain our design can pass induction.

Cliff Cummings' Asynchronous FIFO

Cummings' FIFO has the basic interface shown on the right in Fig 6. Operation starts in the write domain, where `i_wdata` is written to the FIFO anytime `i_wr` is true and the `o_wfull` flag is false. The data can then be read later from `o_rdata` any time `i_rd` is true and `o_rempty` is false, in the values `o_rdata`. Likewise, there's a clock and an asynchronous reset associated with each clock domain.

Fig 6. Asynchronous FIFO block diagram



Indeed, the interface is very straight-forward.

Cummings' FIFO, however, is built using a series of separate modules: one module for each clock synchronizer, one for the memory itself, and another two modules for the read pointer and the write pointer respectively. Since a design with any hidden states within it may struggle to pass induction, I took the liberty of rearranging Cummings' FIFO a little bit. Specifically, I placed all the parts and pieces into a single file. This will make it easier to reference values within the design from within a single formal property section below.

The design begins with some declarations. Note that this FIFO will use a parameterized data width of two bits, and an address width of 4 bits–and so this FIFO will hold `2^4=16` elements. While the widths are arbitrary, I've chosen smaller widths to help deal with the combinatorial explosion associated with using formal methods.

```
module afifo(i_wclk, i_wrst_n, i_wr, i_wdata, o_wfull,
  i_rclk, i_rrst_n, i_rd, o_rdata, o_rempty);
 parameter DSIZE = 2,
   ASIZE = 4;
 localparam DW = DSIZE,
   AW = ASIZE;
 input wire   i_wclk, i_wrst_n, i_wr;
 input wire [DW-1:0] i_wdata;
 output reg   o_wfull;
 input wire   i_rclk, i_rrst_n, i_rd;
 output wire [DW-1:0] o_rdata;
 output reg   o_rempty;

 wire [AW-1:0] waddr, raddr;
 wire   wfull_next, rempty_next;
 reg [AW:0]  wgray, wbin, wq2_rgray, wq1_rgray,
    rgray, rbin, rq2_wgray, rq1_wgray;
 //
 wire [AW:0]  wgraynext, wbinnext;
 wire [AW:0]  rgraynext, rbinnext;

 reg [DW-1:0] mem [0:((1<<AW)-1)];
```

These declarations are followed by the write logic of the FIFO. This includes bringing the read Gray pointers into the write clock domain,

```
 //
```

```
// Cross clock domains
//
// Cross the read Gray pointer into the write clock domain
initial { wq2_rgray,  wq1_rgray } = 0;
always @(posedge i_wclk or negedge i_wrst_n)
if (!i_wrst_n)
 { wq2_rgray, wq1_rgray } <= 0;
else
 { wq2_rgray, wq1_rgray } <= { wq1_rgray, rgray };
```

maintaining the write pointer and its Gray code equivalent,

```
// Calculate the next write address, and the next graycode pointer.
assign wbinnext  = wbin + { {(AW){1'b0}}, ((i_wr) && (!o_wfull)) };
assign wgraynext = (wbinnext >> 1) ^ wbinnext;

assign waddr = wbin[AW-1:0];

// Register these two values--the address and its Gray code
// representation
initial { wbin, wgray } = 0;
always @(posedge i_wclk or negedge i_wrst_n)
if (!i_wrst_n)
 { wbin, wgray } <= 0;
else
 { wbin, wgray } <= { wbinnext, wgraynext };

assign wfull_next = (wgraynext == { ~wq2_rgray[AW:AW-1],
   wq2_rgray[AW-2:0] });
```

and writing values into the asynchronous FIFO on the write clock.

```
//
// Calculate whether or not the register will be full on the next
// clock.
initial o_wfull = 0;
always @(posedge i_wclk or negedge i_wrst_n)
if (!i_wrst_n)
 o_wfull <= 1'b0;
else
 o_wfull <= wfull_next;

//
// Write to the FIFO on a clock
always @(posedge i_wclk)
if ((i_wr)&&(!o_wfull))
 mem[waddr] <= i_wdata;
```

The write section is followed by the read section, having almost exactly the same format. First the Gray coded write address crosses clocks into the read clock domain,

```
//
// Cross clock domains
//
// Cross the write Gray pointer into the read clock domain
initial { rq2_wgray,  rq1_wgray } = 0;
always @(posedge i_rclk or negedge i_rrst_n)
if (!i_rrst_n)
 { rq2_wgray, rq1_wgray } <= 0;
else
 { rq2_wgray, rq1_wgray } <= { rq1_wgray, wgray };
```

then the read pointer is adjusted.

```
// Calculate the next read address,
assign rbinnext  = rbin + { {(AW){1'b0}}, ((i_rd)&&(!o_rempty)) };
// and the next Gray code version associated with it
assign rgraynext = (rbinnext >> 1) ^ rbinnext;

// Register these two values, the read address and the Gray code version
// of it, on the next read clock
//
initial { rbin, rgray } = 0;
always @(posedge i_rclk or negedge i_rrst_n)
if (!i_rrst_n)
 { rbin, rgray } <= 0;
else
 { rbin, rgray } <= { rbinnext, rgraynext };

// Memory read address Gray code and pointer calculation
assign raddr = rbin[AW-1:0];
```

Finally, an determination is made as to whether or not the FIFO is empty and then a value is (may be) read from the buffer.

```
// Determine if we'll be empty on the next clock
assign rempty_next = (rgraynext == rq2_wgray);

initial o_rempty = 1;
always @(posedge i_rclk or negedge i_rrst_n)
if (!i_rrst_n)
 o_rempty <= 1'b1;
else
 o_rempty <= rempty_next;
```

```
//
// Read from the memory--a clockless read here, clocked by the next
// read FLOP in the next processing stage (somewhere else)
//
assign o_rdata = mem[raddr];

endmodule
```

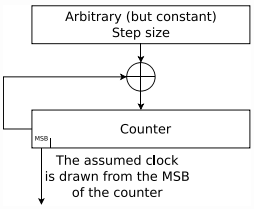
This is all straight forward with the exception of our pending question: will this work?

Let's write some formal properties together to find out.

Assuming two clocks

We've already discussed how to assume a clock using SystemVerilog properties and SymbiYosys. Basically, this involves allowing the solver to pick two arbitrary step sizes, `f_wclk_step` and `f_rclk_step`, creating counters that step by these amounts but with arbitrary initial states, and then assuming the incoming clock is identical to the most-significant bits of these counters. All of this logic is shown pictorially in Fig 7 to the right.

Fig 7. Assuming an Arbitrary Clock Rate



Let's walk through those steps again, this time in Verilog.

We'll start with the arbitrary clock steps. These are just constants, chosen by the solver. They are subject to assumed constraints, but nothing more. In past articles, I would've used the magic value `$anyconst` to describe a value with these properties.

```
localparam F_CLKBITS=5;
wire [F_CLKBITS-1:0] f_wclk_step, f_rclk_step;

assign f_wclk_step = $anyconst;
assign f_rclk_step = $anyconst;
```

However, in an effort to make these random constant value declarations more uniform, and particularly to be able to support both SystemVerilog *and* VHDL, Yosys now has the option of declaring these clock step amounts using an attribute, `(* anyconst *)`.

```
(* anyconst *) wire [F_CLKBITS-1:0] f_wclk_step, f_rclk_step;
```

We'll assume that both of these step sizes are greater than zero. This will guarantee that each clock moves forward–and that they are never stuck.

```
always @(*)
 assume(f_wclk_step != 0);
always @(*)
 assume(f_rclk_step != 0);
```

We can now use these steps sizes in a counter.

```
reg [F_CLKBITS-1:0] f_wclk_count, f_rclk_count;

always @($global_clock)
 f_wclk_count <= f_wclk_count + f_wclk_step;
always @($global_clock)
 f_rclk_count <= f_rclk_count + f_rclk_step;
```

Notice the reference to `$global_clock` here. This is a reference to the time-step within the formal solver.

Although `$global_clock` is a SystemVerilog concept, SystemVerilog actually defines this value differently. In SystemVerilog, the `$global_clock` needs to be defined before it can be used. It's similar, but not quite the same as the global formal timestep. To bridge this gap, the commercial version of Yosys allows,

```
(* gclk *) wire gbl_clock;
global clocking @(posedge gbl_clock); endclocking;
```

In this case, `gbl_clock` is defined to be the global simulation clock. Transitions on any edge of this clock will reference a formal timestep. The `global clocking` declaration just defines the SystemVerilog identifier, `$global_clock`, to be a reference to a transition of this formal timestep clock, `gbl_clock`.

Now, with all this background, we can finally assume our incoming clocks at their various speeds. Specifically, we'll assume that the read and write clocks are synonymous with the most significant bit of these counters.

```
always @(*)
begin
 assume(i_wclk == f_wclk_count[F_CLKBITS-1]);
 assume(i_rclk == f_rclk_count[F_CLKBITS-1]);
end
```

This will give them each a rough 50% duty cycle.

If you look at either clock within a trace, you'll notice that the edges will appear to jitter back and forth. For example, the clock might take two time periods in one cycle and three in the next. This is normal. It is a byproduct of how the clocks are defined. However, these formal clocks don't need to be so stable that you can drive a hardware PLL from them. They only need to be representative of two separate clock rates, and they will be that for us.

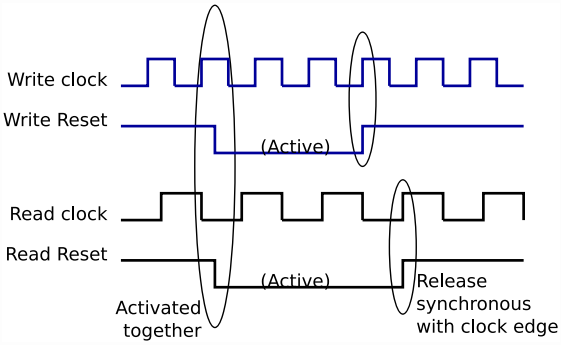
The Asynchronous Reset

Cummings' design includes two negative logic asynchronous reset signals, `i_wrst_n`, and `i_rrst_n`.

These two resets are related to each other. Specifically, we'll assume that the two resets will always be asserted at the same time together, but that they are only ever de-asserted with their respective clocks. This latter criteria is sometimes called asynchronous assertion with a synchronous release.

You can see this concept drawn out in Fig 8 below.

Fig 8. Asynchronous assertion, Synchronous Release



This is what we are going to insist, or rather assume, of our reset. (Remember: assume inputs, assert local state and outputs.)

But first, let's discuss the initial state of the reset. Initially, I wanted to assume that both resets started asserted.

```
initial assume(!i_wrst_n);
initial assume(!i_rrst_n);
```

What happens, though, with this approach when you want to implement a design where the resets are both hard-wired high (inactive)? This should be an allowed configuration for an FPGA. Instead, let's only assume that both resets start out the same. That is, either the design starts with both resets active or with no resets active.

```
initial assume(i_wrst_n == i_rrst_n);
```

We'll also assume that both resets are driven from one source. That means that if one reset line falls, i.e. that reset is asserted, then the other must fall as well–just as we illustrated in Fig 8 above. Neither reset should be asserted without the other.

```
always @($global_clock)
 assume($fell(i_wrst_n)==$fell(i_rrst_n));
```

This assumption has a sad consequence: because it is a clocked assumption, which it needs to be in order to evaluate `$fell()`, it will take a clock edge before this assumption is applied. As a result, many of our assertions, particularly those that depend upon results from both resets, which would otherwise have depended on `@(*)` are instead asserted on `@($global_clock)`.

We'll also assume a synchronous release from reset, also shown in Fig 8 above. Hence, if the associated clock doesn't rise, then neither should the reset.

```
always @($global_clock)
if (!$rose(i_wclk))
 assume(!$rose(i_wrst_n));

always @($global_clock)
if (!$rose(i_rclk))
 assume(!$rose(i_rrst_n));
```

Before we leave the discussion of the reset, there's one more item to check. Following any reset, the read and write pointers should be zero. If the write side deasserts the reset first, the write pointer may start incrementing before the read pointer. On the other hand, if the read side leaves the reset state first, it shouldn't be allowed to read anything until something has been written into the FIFO. Hence, we'll assert that anytime the write reset is low that the read address must point to the beginning of the FIFO.

```
always @($global_clock)
if (!i_wrst_n)
 assert(rbin == 0);
```

Assuming Synchronous Inputs

Whenever you build a set of formal properties to describe logic in multiple clock domains, you'll want to assume that the inputs associated with each clock domain remain synchronous to that domain. While technically whether or not this takes place depends upon the problem, I personally find it disconcerting to watch values change arbitrarily within the generated trace–values that should've only changed on a clock edge. Going one step further, we should also assert that the outputs are synchronous with each domain as well.

The basic form of asserting that something is synchronous to a positive edged clock is to assert that if the clock doesn't rise, then the value should be stable. This needs to be done with the `$global_clock`, since it's describing the minimum sub-clock time interval within the design.

```
always @($global_clock)
if (!$rose(clk))
 assert($stable(value));
```

You also need an `f_past_valid` flag to describe this as well.

```
reg f_past_valid_gbl;
initial f_past_valid_gbl = 1'b0;
always $($global_clock)
```

```
  f_past_valid_gbl <= 1'b1;
```

In our case, we can use three separate `f_past_valid` types of flags: one for each of the clocks in our design, `$global_clock`, `i_wclk` (not shown), and `i_rclk` (not shown). Without these values, the solver will try to reference an undefined value before time began and all assertions based upon this value would fail. For this reason, all references to `$past()` or in this case to `$rose()`, `$fell()`, or `$stable()` need to be qualified by an `f_past_valid` piece of logic. In this case, by `f_past_valid_gbl` –the `f_past_valid` signal we created for the formal time-step.

```
always @($global_clock)
if (f_past_valid_gbl)
begin
```

In the case of the write clock, if the write clock hasn't risen than the write request line, `i_wr`, and the write data, `i_wdata`, should both be assumed constant.

```
 if (!$rose(i_wclk))
 begin
  assume($stable(i_wr));
  assume($stable(i_wdata));
```

In a similar manner, the `o_wfull` flag should also be constant. Since this latter flag is an output, we'll *assert* that it is constant rather than *assuming* it. The only difficulty being the reset. Specifically, if the write reset is ever asserted, then `o_wfull` will drop asynchronously. Hence, we'll need to check for that.

```
  assert($stable(o_wfull)||(!i_wrst_n));
 end
```

The read logic is almost identical as well, so it's not that remarkable.

```
 if (!$rose(i_rclk))
 begin
  assume($stable(i_rd));
  assert((o_rempty)||($stable(o_rdata)));
  assert((!i_rrst_n)||($stable(o_rempty)));
 end
end
```

This is just one of those rather tedious parts of an asynchronous proof.

Verifying the Reset

My readers should understand why I use `f_past_valid`: if you make an assertion using the `$past` operator, one were the `$past()` operator references a value before the initial time began, then the solver can immediately declare that assertion to be invalid. Not only that, there will be no trace associated with that past assertion. To keep this from happening, I use a register I call `f_past_valid`. It's initialized to zero, and then set immediately to one on the first clock tick.

This is all straightforward. It's something I've been doing for many proofs, and I've posted about the reasons for it already on this blog.

One time, however, I was caught by surprise when I examined a trace describing a formal proof that crossed multiple files. In this trace, the `f_past_valid` signals didn't line up! Some were true, others were false. This should've never happened in real life. What made it happen was induction, where the solver gets to pick the initial states for all values.

To keep this from happening, I will often add statements to my design forcing the design to be in reset any time `f_past_valid` is false.

```
always @(*)
if (!f_past_valid)
 assume(i_reset);
```

I also use `f_past_valid` to double check any initialization statements. For synchronous logic, this usually looks something like:

```
always @(posedge i_clk)
if ((!f_past_valid)||($past(i_reset)))
begin
 assert(value == 0);
 // ....
end
```

In this manner, I can insist that anytime one `f_past_valid` is false, the entire design is in its reset state, *and* that the reset state is identical to the initial state.

That's not how I verified this design, though. In particular, within this design I wanted to allow the reset, line(s) to be tied high if the designer so chose.

In this design, I used the fact that the reset was asynchronous, to do things a little differently. Instead of checking whether the reset was true in the past, we'll do this second check based upon whether the reset is currently asserted.

Hence, anytime the write reset is asserted, or until the first write clock, all of our write values should be in their initial states.

```
always @(*)
if ((!f_past_valid_wr)||(!i_wrst_n))
begin
  `ASSUME(i_wr == 0);
  //
  `ASSERT(wgray == 0);
  `ASSERT(wbin == 0);
```

```
  `ASSERT(!o_wfull);
  //
  `ASSERT(wq1_rgray == 0);
  `ASSERT(wq2_rgray == 0);
  `ASSERT(rq1_wgray == 0);
  `ASSERT(rq2_wgray == 0);
```

While this section is all about the *write* side of the interface, we'll make an exception to this rule for two values associated with the read clock: the FIFO read address, `rbin`, and its associated `o_rempty` signal. Anytime the registers in the write clock are in their reset, state, these read values must also remain in their reset values–since the write side hasn't yet written anything to be read, and the read pointer isn't allowed to move forward from an empty FIFO while it's still empty, etc. You get the idea.

```
  //
  `ASSERT(rbin == 0);
  `ASSERT(o_rempty);
end
```

The read side is less remarkable. It also includes roughly the same logic.

```
always @(*)
if ((!f_past_valid_rd)||(!i_rrst_n))
begin
  `ASSUME(i_rd == 0);
  //
  `ASSERT(rgray == 0);
  `ASSERT(rbin == 0);
  `ASSERT(rq1_wgray == 0);
  `ASSERT(rq2_wgray == 0);
  `ASSERT(wq1_rgray == 0);
  `ASSERT(wq2_rgray == 0);
end
```

At this point, we know our design starts in a reset configuration either at the beginning of time, or following any reset.

Verifying the Fill Levels

All of the steps so far have been preliminary, set up sorts of things. None of them have actually impacted the proof requirements we listed above. Let's now move into the actual FIFO properties associated with its operation. We'll start with the pointers in this section, and then verify the two element write test in the next section. In between these two sections, we'll skip the proof of the Gray codes. It follows directly from the discussion above, and it is in the file if you wish to reference it. Indeed, many of the properties associated with the fill of this FIFO we've already discussed above, but they are important enough to discuss one more time here.

Will start with a measure of how full the FIFO is.

```
wire [AW:0]  f_fill;

assign f_fill = (wbin - rbin);
```

I like to prefix values, like this one, that are only used in the formal properties with a `f_`. It helps me quickly recognize which values are used for the formal proof, and which are used within the design itself.

As mentioned above, because `f_fill` is a value only defined in our formal properties, it can depend upon values that cross clock domains.

As we indicated above, we'll start out asserting that the FIFO is initially empty, and that it never has more than `2^(AW)` elements within it.

```
initial `ASSERT(f_fill == 0);
always @($global_clock)
  `ASSERT(f_fill <= { 1'b1, {(AW){1'b0}} });
```

Now let's look at `o_wfull`. In particular, anytime the FIFO is full, the `o_wfull` flag should be asserted. This property should help to assure users of this asynchronous FIFO that it actually works–at least, that this flag works.

```
// Any time the FIFO is full, o_wfull should be true.  It may take a
// clock or two to clear, though, so this is an implication and not
// an equals.
always @($global_clock)
if (f_fill == {1'b1,{(AW){1'b0}}})
  `ASSERT(o_wfull);
```

We'll also check how things work on the clock before everything is full. Specifically, if the FIFO has `2^(AW)-1` elements in it, it should then be full on the next write. That means that either the `o_wfull` flag must be true, there's no request to write, or `o_wfull` will be set to true on the next clock.

```
always @($global_clock)
if (f_fill == {1'b0,{(AW){1'b1}}})
  `ASSERT((wfull_next)||(!i_wr)||(o_wfull));
```

This was one of those extra properties necessary to pass induction.

We'll make a similar assertion about the FIFO being empty: Any time the FIFO is truly empty, the `o_rempty` must be true. In this design, it will also be asserted at other times as well (i.e. there's a lag before its cleared), so as with the write check above it, this is a one-way implication only, and not an equals.

```
always @($global_clock)
if (f_fill == 0)
  `ASSERT(o_rempty);
```

We'll add another assertion in here for good measure and to help with induction. Specifically, we'll assert that if the `f_fill` isn't quite zero, then either the empty flag is true, or the empty flag will be true on the next clock, or nothing is being read. Put simply, let's check the logic before the fill becomes empty just like we did with the logic before the FIFO became full.

```
// If the FIFO is about to be empty, the logic should be able
// to detect that condition as well.
always @($global_clock)
if (f_fill == 1)
  `ASSERT((rempty_next)||(!i_rd)||(o_rempty));
```

Consistency requires that the Gray pointers always match their respective address pointers. Let's insist that be the case here, lest the induction engine find some way to defeat this proof using an invalid state. This applies for both read and write Gray pointers.

```
always @(*)
  `ASSERT(wgray == ((wbin>>1)^wbin));

always @(*)
  `ASSERT(rgray == ((rbin>>1)^rbin));
```

Finally, let's double check that our Gray pointer arithmetic truly does match our address arithmetic. Remember, in the design above, we pushed the empty and full logic from combinatorial logic into clocked logic. The combinatorial logic should still hold, so let's check it here.

First, is the FIFO full? If so, the Gray pointers should differ in their top bit but be identical for the rest of their bits.

```
always @(*)
  `ASSERT( (rgray == { ~wgray[AW:AW-1], wgray[AW-2:0] })
   == (f_fill == { 1'b1, {(AW){1'b0}} }) );
```

Second, if the FIFO is empty, and only if the FIFO is empty, then the Gray pointers should be identical.

```
always @(*)
  `ASSERT((rgray == wgray) == (f_fill == 0));
```

These are the basic properties we want to prove about the FIFO's fill amount. If all you want is a bounded model check, you can skip to the next section.

On the other hand, if you'd like a full induction check, and hence a proof for all time, then we need to check that the values and registers associated with crossing clock domains are valid as well.

To do this, we'll create 2-FF synchronizers within the formal properties section to delay the binary pointer values so they'll have an image, in the new clock domain, synchronous with the value of their Gray versions. Again, since this isn't synthesizable code, there's no danger of metastability here, as there would be in the synthesis section above.

```
reg [AW:0] f_w2r_rbin, f_w1r_rbin,
  f_r2w_wbin, f_r1w_wbin;
wire [AW:0] f_w2r_fill, f_r2w_fill;

initial { f_w2r_rbin, f_w1r_rbin } = 0;
always @(posedge i_wclk or negedge i_wrst_n)
if (!i_wrst_n)
 { f_w2r_rbin, f_w1r_rbin } <= 0;
else
 { f_w2r_rbin, f_w1r_rbin } <= { f_w1r_rbin, rbin };

initial { f_r2w_wbin, f_r1w_wbin } = 0;
always @(posedge i_rclk or negedge i_rrst_n)
if (!i_rrst_n)
 { f_r2w_wbin, f_r1w_wbin } <= 0;
else
 { f_r2w_wbin, f_r1w_wbin } <= { f_r1w_wbin, wbin };
```

The next step is to force these helper variables to be properly matched to their Gray coded counter parts. (No pun intended.) That is, we should be able to convert these values to Gray code and they should match.

```
always @(*)
  `ASSERT(rq1_wgray == ((f_r1w_wbin>>1)^f_r1w_wbin));
always @(*)
  `ASSERT(rq2_wgray == ((f_r2w_wbin>>1)^f_r2w_wbin));

always @(*)
  `ASSERT(wq1_rgray == ((f_w1r_rbin>>1)^f_w1r_rbin));
always @(*)
  `ASSERT(wq2_rgray == ((f_w2r_rbin>>1)^f_w2r_rbin));
```

Finally, we can calculate the fill at the other end of this synchronizer chain. These two fill values, one using `wbin` and the re-synchronized `rbin`, the other using `rbin` and the re-synchronized `wbin`, should clearly describe our `o_wfull` and `o_rempty` flags.

```
assign f_w2r_fill = wbin - f_w2r_rbin;
assign f_r2w_fill = f_r2w_wbin - rbin;


// And assert that the fill is always less than or equal to full.
// This catches underrun as well as overflow, since underrun will
// look like the fill suddenly increases
always @(*)
  `ASSERT(f_w2r_fill <= { 1'b1, {(AW){1'b0}} });
always @(*)
```

```
  `ASSERT(f_r2w_fill <= { 1'b1, {(AW){1'b0}} });

// From the writers perspective, anytime the Gray pointers are
// equal save for the top bit, the FIFO is full and should be asserted
// as such.  It is possible for the FIFO to be asserted as full at
// some other times as well.
always @(*)
if (wgray == { ~wq2_rgray[AW:AW-1], wq2_rgray[AW-2:0] })
  `ASSERT(o_wfull);

// The same basic principle applies to the reader as well.  From the
// readers perspective, anytime the Gray pointers are equal the FIFO
// is empty, and should be asserted as such.
always @(*)
if (rgray == rq2_wgray)
  `ASSERT(o_rempty);
```

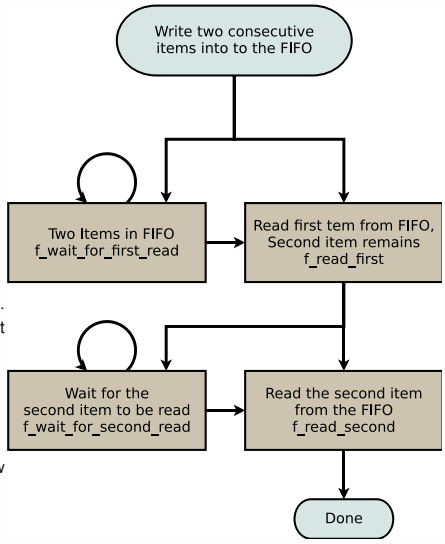
At this point, we now have full confidence in our two flags, `o_wfull` and `o_rempty`.

When I first formally verified this FIFO, this was where I stopped. Then I was told of additional FIFO property, which we'll discuss in the next section.

FIFO Contract

I've now been a part of and read several discussions about how to formally verify a FIFO–whether synchronous or asynchronous. I've even found one reference to this method as well, although there must be more. These discussions all revolve around two arbitrary values written to the FIFO in succession, that then need to be able to be read out of the in succession.

Fig 9. Two consecutive items written to a FIFO, must be able to be read out consecutively at a later time



This was a new criteria for me when I first heard it, so I decided to try it on one of my own FIFOs to see if it would work. It didn't pass (originally)–I was overwriting the FIFO's tail on any write request during overrun. Oops. Hence, while I used to think this criteria wasn't all that necessary, I no longer hold that view.

If you read Doc Formal's article on invariants, or the slides I saw at DVCON 2018, they all made this proof look quite easy. When I actually tried it myself, I couldn't figure out how to keep it as simple as the slides presented it. In particular, how can you constrain the formal engine in such a way that the parts of the proof don't get out of hand and into an inconsistent state?

As a result, we'll apply this criteria below in such a constrained way that there's no way the solver could get it wrong. Specifically, if the solver starts in the middle of the two element sequence described by this contract, we'll make certain below that the state the solver finds itself within matches our own internal state in every way and in every step along the way.

So let's start out by picking an arbitrary location within the FIFO,

```
(* anyconst *) wire [AW:0]  f_const_addr;
```

and a location immediately following it.

```
wire [AW:0]  f_const_next_addr;

assign f_const_next_addr = f_const_addr + 1;
```

Let's also pick two arbitrary values which will eventually will be placed in those locations.

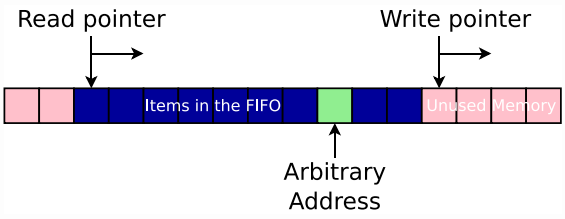
```
(* anyconst *) reg [DW-1:0] f_const_first, f_const_next;
```

Let's also create an expression that will be true anytime the first address, `f_const_addr`, is within the valid set of FIFO values. This is tricky because the read and write pointers will wrap around the end of the FIFO, so we'll have to break this out by stages. Since it's non-intuitive, we'll show this graphically as we go along.

```
reg   f_addr_valid, f_next_valid;
```

Looking at the first address, `f_const_addr`, if the read pointer follows the write pointer in order, and `f_const_addr` is between the two, then `f_const_addr` references an item within the FIFO. This case is shown in Fig 10 below.

Fig 10. The Arbitrary Address lies between the Read and Write Pointers



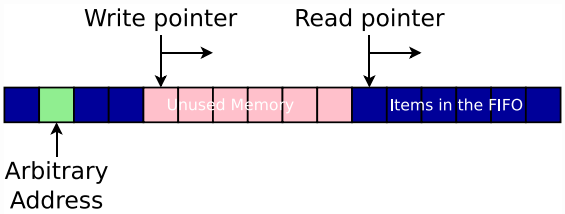
Given the picture above, the test should start to make sense.

```
always @(*)
```

```
begin
 f_addr_valid = 1'b0;
 if((wbin > rbin)&&(wbin > f_const_addr)
    &&(rbin <= f_const_addr))
  // Order rbin <= addr < wbin
  f_addr_valid = 1'b1;
```

In a similar fashion, if `wbin` has wrapped around so that the write pointer is now less than the read pointer, `rbin`, but `f_const_addr` remains less than the write pointer, then `f_const_addr` is within the FIFO. This is shown in Fig 11 below.

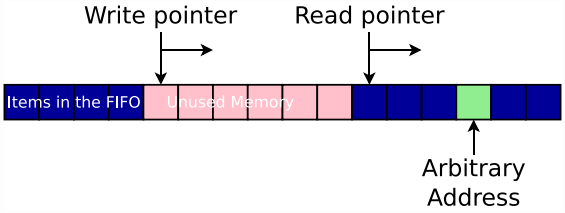
Fig 11. The Arbitrary Address lies between the Read and Write Pointers




```
 else if ((wbin < rbin)&&(f_const_addr < wbin))
  // addr < wbin < rbin
  f_addr_valid = 1'b1;
```

Finally, if the write pointer is less than the read pointer, but `f_const_addr` lies after the read pointer, then the address is in the FIFO.

Fig 12. The Arbitrary Address lies between the Read and Write Pointers




```
 else if ((wbin < rbin)&&(rbin <= f_const_addr))
  // wbin < rbin < addr
  f_addr_valid = 1'b1;
end
```

I'll skip the similar, though identical, comparison for the next address following, `f_const_next_addr`. You can find it within the verilog code for the FIFO if you like.

The next step is to implement the logic shown in Fig 9 above. With all the associated details, this becomes rather complex, so let's break it down a bit. Specifically, we'll create some logic to determine if the first, second, or both values are in the FIFO. That is, not only is their address a valid reference to an item within the FIFO, but the data at that address must match as well.

For the first value to be in the FIFO, 1) it's address must lie within the FIFO, and 2) the value at that address must match the first of our two values.

```
reg f_first_in_fifo, f_second_in_fifo, f_both_in_fifo;

always @(*)
 f_first_in_fifo = (f_addr_valid)
   &&(mem[f_const_addr[AW-1:0]]==f_const_first);
```

The same logic applies to the second value within our FIFO, only we'll check against the next address, `f_const_next_addr`, and the value against `f_const_next`.

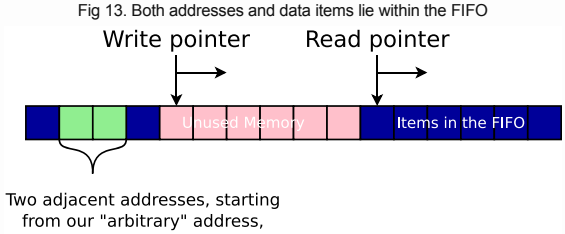
```
always @(*)
 f_second_in_fifo = (f_next_valid)
   &&(mem[f_const_next_addr[AW-1:0]]==f_const_next);
```

Finally, we'll set `f_both_in_fifo` to only be true if both values are within the FIFO, at their respective address locations.

```
always @(*)
 f_both_in_fifo = (f_first_in_fifo)&&(f_second_in_fifo);
```

Conceptually, this might look like Fig 13 below.

Fig 13. Both addresses and data items lie within the FIFO



At this point we want to assert that if both values are in the FIFO, then we should at some point later in time be able to read them both out one by one.

If you read other articles on FIFOs, you'll often see them simplified using a concurrent assertion so that the logic reads something like:

```
assert property (@(posedge i_clk)
 disable iff (i_rrst_n)
```

```
 f_both_in_fifo |->
  f_wait_for_first_read [*0:$]
  ##1 f_read_first
  ##1 f_wait_for_second_read [*0:$]
  ##1 f_read_second);
```

Decoding this, it means that if both values are in the FIFO, then the FIFO can remain in that state indefinitely, or alternatively a read request can read the first value from the FIFO. Then, the FIFO can wait with the second value in memory indefinitely or (ultimately) it can be read out.

The problem with this simplified notation is all the details. Worse, in my case I wanted to be able to support an induction length shorter than the entire (potentially infinite) sequence length. To make certain the solver could handle this case, I wanted to make certain that every state within the sequence was constrained unambiguously. All registers associated with each of the states needed to be fully constrained so there was no room for the solver to place the design into an invalid state.

Again, feel free to use Fig 9 above as a reference in this discussion below.

For example, waiting for the first read means that both values must be in the FIFO, and that neither are being read at that time.

```
reg f_wait_for_first_read, f_read_first, f_read_second,
 f_wait_for_second_read;

always @(*)
 f_wait_for_first_read = (f_both_in_fifo)
   &&((!i_rd)||(f_const_addr != rbin)||(o_rempty));
```

Notice the `o_rempty` flag. I wasn't expecting this one. If both items are in the FIFO, then `o_rempty` should be zero already, right? Not quite. When I ran SymbiYosys, the produced trace reminded me that while `o_rempty` will be raised any time the design is empty, it may take a couple of clock cycles after the design is no longer empty in order to be lowered. Hence, if `i_rd` is true but `o_rempty` is also true, then no read is taking place and we'll need to keep waiting for that first read.

Reading the first value from the FIFO means that there must be a read request, the FIFO must not be empty (finally), and the read request must be of the first value. Further, at the time of this read, both values must still be in the FIFO.

```
always @(*)
 f_read_first = (i_rd)&&(o_rdata == f_const_first)&&(!o_rempty)
  &&(rbin == f_const_addr)&&(f_both_in_fifo);
```

We'll then wait for our second read. This may take between zero and an infinite number of clock cycles. While waiting, the second value must be still in the FIFO, the read pointer must point to the second of our two addresses, and we can't allow any reads from the FIFO.

```
always @(*)
 f_wait_for_second_read = (f_second_in_fifo)
   &&((!i_rd)||(o_rempty))
   &&(f_const_next_addr == rbin);
```

Finally, the last stage in our test is the one where the second value is read from the FIFO. In this case, there must be a read request, the read data must match the data we started with, the FIFO cannot be empty, the address must match, etc. It's a mouthful!

```
always @(*)
 f_read_second = (i_rd)&&(o_rdata == f_const_next)&&(!o_rempty)
   &&(rbin == f_const_next_addr)
   &&(f_second_in_fifo);
```

You may have noticed that I've just assigned values to variables. These values describe the various states the FIFO may be in, but they don't (yet) string those states together via an assertion of any type. That's coming next.

If you ever find yourself needing to do this, let me share with you what not to do. Do not place all of these criteria into one big huge property like the one shown below.

```
assert property (@(posedge i_clk)
 (f_first_in_fifo)&&(f_second_in_fifo);
 |-> (f_first_in_fifo)&&(f_second_in_fifo)
  &&((!i_rd)||(f_const_addr != rbin)||(o_rempty)) [*0:$]
 ##1 (i_rd)&&(o_rdata == f_const_first)&&(!o_rempty)
  &&(rbin == f_const_addr)
  &&(f_first_in_fifo)&&(f_second_in_fifo);
 ##1 (f_second_in_fifo)
   &&((!i_rd)||(o_rempty))
   &&(f_const_next_addr == rbin) [*0:$]
 ##1 (i_rd)&&(o_rdata == f_const_next)&&(!o_rempty)
   &&(rbin == f_const_next_addr)
   &&(f_second_in_fifo));
```

(My initial attempt was even worse–I didn't use `f_first_in_fifo` or `f_sceond_in_fifo`.)

The problem with such a complicated property is, what happens when it fails? In my case, SymbiYosys would say that the assertion failed, and then give me the line of this assertion and a trace showing me that the assertion had failed. It never said which part of the assertion, which line within the assertion, or which step the assertion was within, was the one that failed. The line number was always the line number of the whole assertion sequence.

You can guess how I solved this problem using the state definitions we just defined above. By creating wires to contain these complicated logic components, these logic components describing which state we are in then show up on the trace. This made it possible for me to understand the trace, and thus to see what was going wrong.

On the other hand, if you are using the open source version of SymbiYosys, then you'll have to describe this

check using immediate assertions alone. Again, the wires defining the various states above will help us simplify the state machine we'll need to write.

Starting at the top, we'll create a quick state transition checker that will be disabled any time `i_wrst_n` gets asserted.

```
always @($global_clock)
if ((f_past_valid_gbl)&&(i_wrst_n))
begin
```

The state starts with both items in the FIFO, or equivalently with `f_both_in_fifo` true. Following `f_both_in_fifo`, we can either read the first value or continue waiting for the first value to be read.

```
  if ((!$past(f_read_first))&&(($past(f_both_in_fifo))))
   assert((f_wait_for_first_read)
    || (($rose(i_rclk))&&(f_read_first)));
```

Let me point you attention to the `$rose(i_rclk)` condition for a moment. The first value cannot be read without a read that starts on the rising read clock edge.

Once we've read that first value, we'll need to stay in that state until the next clock edge. On the next read clock edge, we can either read the second value, or start waiting for the second value to be read.

```
  if ($past(f_read_first))
   assert(
    ((!$rose(i_rclk))&&(f_read_first))
    ||($rose(i_rclk)&&((f_read_second)
      ||(f_wait_for_second_read))));
```

Finally, if we were waiting for the second value to be read on the last time step, then we can continue waiting on this time step, or if the clock rises we can actually start reading the second value.

```
  if ($past(f_wait_for_second_read))
   assert((f_wait_for_second_read)
    ||(($rose(i_rclk))&&(f_read_second)));
end
```

There you have it! That's the majority of this proof.

In hindsight, I probably didn't need all of those free variables, those using `(* anyconst *)`, to make this work. If I have to come back to this later, I may remove them–instead reflecting that any input value is by default a free variable of its own right.

Cover Properties

We're not quite done yet. We'd still like to know about whether or not some things that might actually take place. This was the fourth part in our proof outline above. So far, we've skipped it. We've proved parts one through five except for part four. Part four requires us to demonstrate that the FIFO can actually fill, and that it can actually empty.

Why might we need this? Consider, what would happen if we had somehow accidentally assumed the FIFO would remain empty, or that it would never reach it's fill? Just to make certain we didn't mess things up, let's make sure several states are reachable.

First, let's make sure we can enter the empty state. This is good since we start up in the empty state. In order to return to empty, the FIFO must be able to both receive a value into it and to have a value read out of it.

```
always @($global_clock)
if (f_past_valid_gbl)
 cover((o_rempty)&&(!$past(o_rempty)));
```

The same basic logic applies to the full state. The design starts out non-full, and we'd like to have some assurance that the FIFO can be filled, and also that it can leave its full condition.

```
always @(posedge i_wclk)
if (f_past_valid_wr)
 cover($past(o_wfull)&&(!o_wfull));
```

If you check out the design, you'll find there are several other cover properties as well. Feel free to examine and comment on whether I may have missed any.

```
`endif
endmodule
```

Conclusions

Voila! All of what you need to know to formally verify an asynchronous FIFO. Yes, it took a lot of work, but the good thing is that this work only needed to be done once. Now you can use this FIFO to cross clock domains with all kinds of things like … bus requests even!

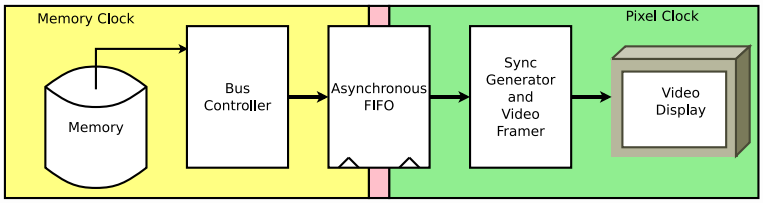
Still not convinced? Feel free to examine the code and the properties yourself. You can even add your own properties, in case I've missed any–whatever it takes to convince you that this FIFO actually works.

Now, what shall we use this for?

Perhaps we could use this to cross clock domains using a bus? For example, what if you had two wishbone buses, one in each of two clock domains. Crossing clock domains with between those buses can be a challenge. Not only that, but it's a particularly challenge required by just about any video system. Why? Well, it just so happens that in every video design I've ever done (there haven't been that many), the pixel clock ends up being at one frequency and the memory (and system) clock is at another. Hence, I'd like to come back to this FIFO design later and modify it so that it produces an output fill level for a frame buffer controller. Such a controller will wait until there is room for another burst length in the buffer before trying to fill the buffer

with a burst.

Fig 14. Asynchronous FIFO within a Video Application



Indeed, that's part of the VGA simulator I recently posted on line. That project includes simulation code for both a source and a sink video signal, taking the source from a location on the screen of the simulator's host, and then displaying the result in a window on that same screen as well. Of course, there's still plenty of room for improvement in the project. In particular, I have need of an HDMI simulation–something that would be an easy adjustment to make to the VGA simulation project.

Perhaps we should come back to that project in a future article, and show how you can use it?

For now, let me thank Cliff Cummings for his excellent article on how to build an asynchronous FIFO, and further for his encouragement to write this article!

But, beloved, be not ignorant of this one thing, that one day is with the Lord as a thousand years, and a thousand years as one day. (2 Pet 3:8)