Gisselquist
Technology, LLC

10. Adding a FIFO

Daniel E. Gisselquist, Ph.D.

Lesson Overview

Serial ports can easily get overloaded with information

- What if the receiver is faster than the transmitter?
  - Perhaps you are bridging two separate serial channels, and each channel has a different baud rate
- If the serial port feeds a CPU, the CPU might not be able to keep up

Let's build a FIFO to address these problems!

Objectives

- Know how to build and verify a FIFO

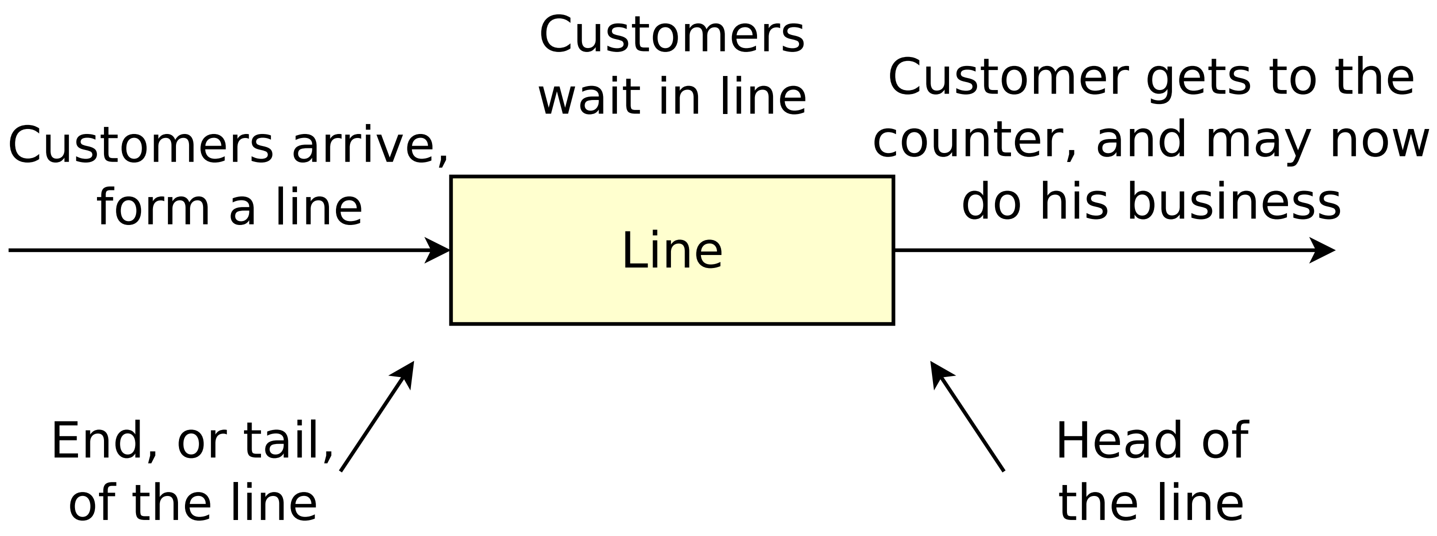
Design Goal

Let’s build a design to buffer a line before transmit

- The design will first read a line of data
- Then write it out sometime later, either . . .
  - After the design receives a newline, or alternatively
  - After the buffer fills
- We’ll use a FIFO to hold the intermediate data

What is a FIFO?

You are probably familiar with waiting in line

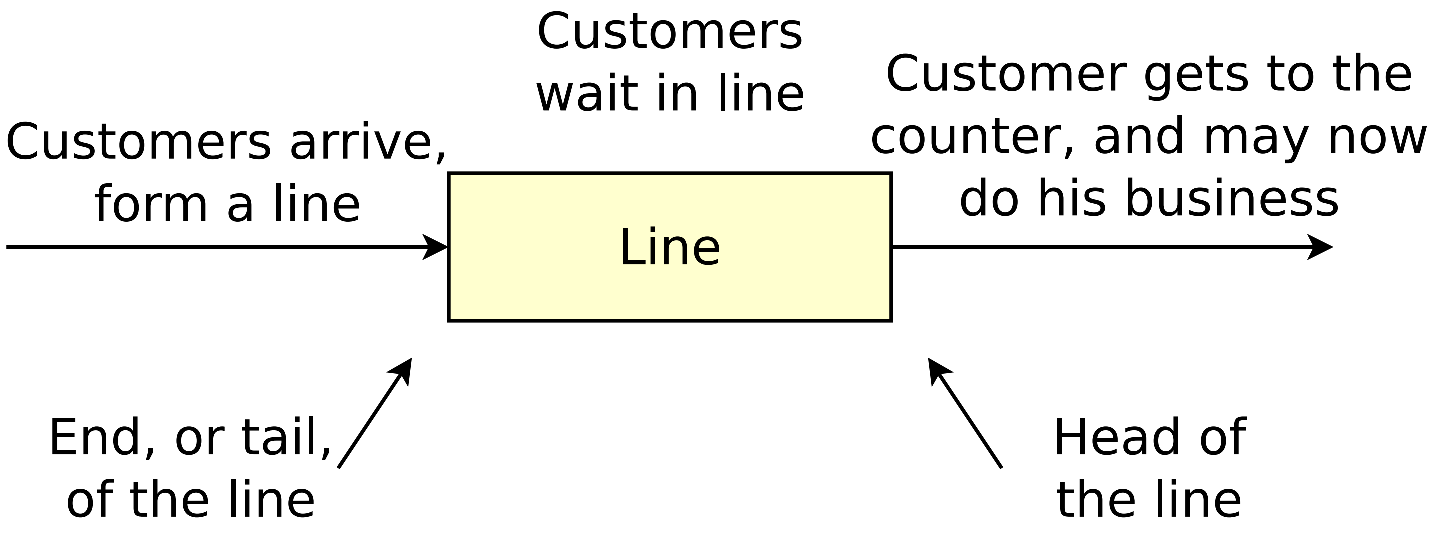

Customers wait in line
Customers arrive, form a line
Line
Customer gets to the counter, and may now do his business
End, or tail, of the line
Head of the line

This could describe . . .

- Waiting in line to purchase something
- Waiting in line to see the doctor
- Waiting in line to vote
- Any number of things

What is a FIFO?

You are probably familiar with waiting in line



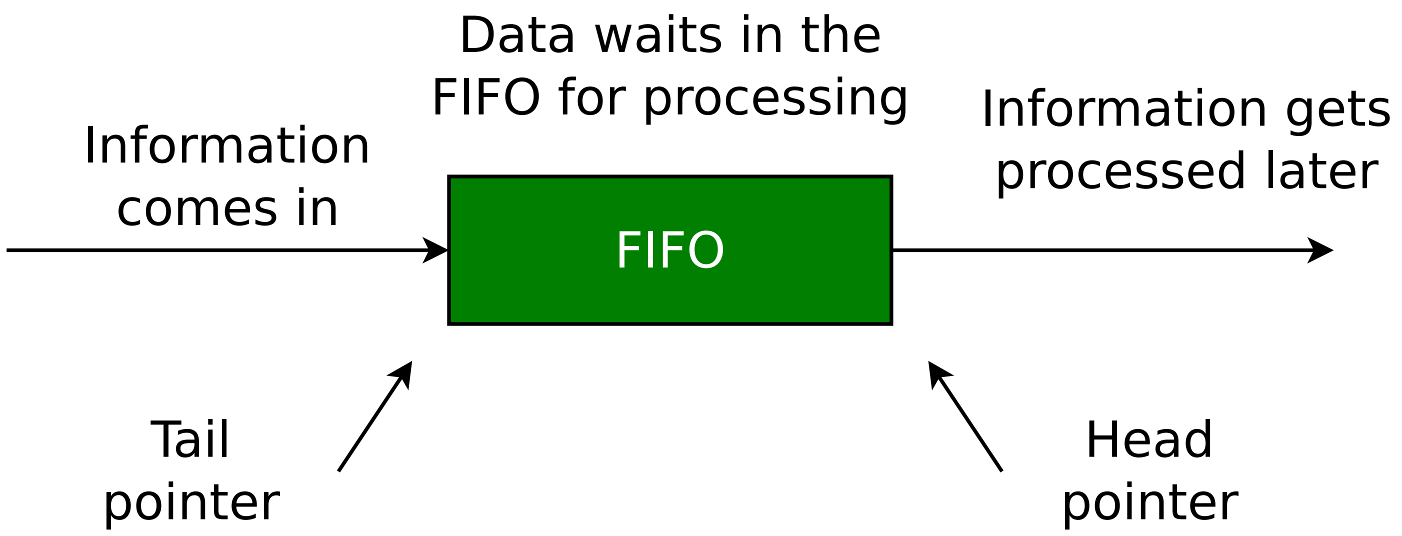
Customers wait in line
Customer gets to the counter, and may now do his business
Customers arrive, form a line
Line
End, or tail, of the line
Head of the line

There are rules to waiting in line

- You always join and the end of the line
- You get service at the front or head of the line
- "Cutting" in line is frowned upon

What is a FIFO?

A FIFO is nothing more than data waiting in line

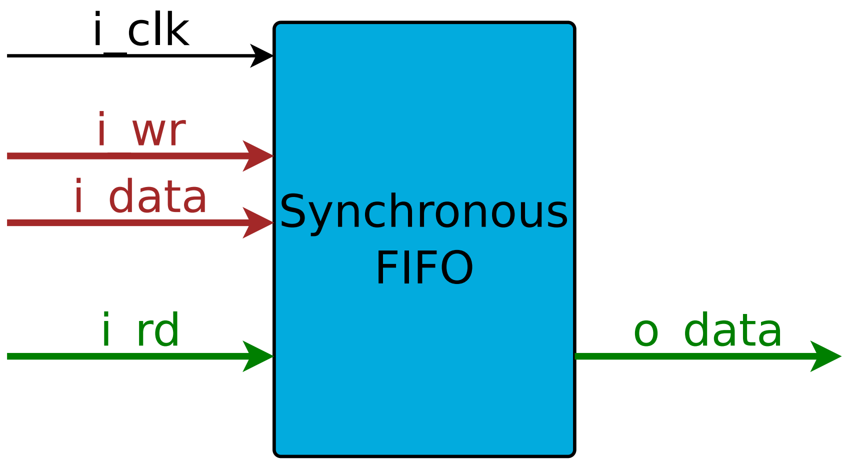


Data waits in the FIFO for processing
Information comes in
FIFO
Information gets processed later
Tail pointer
Head pointer

- Data enters the FIFO at the tail
- Data gets processed from the head
- Data in the FIFO is stored in block RAM

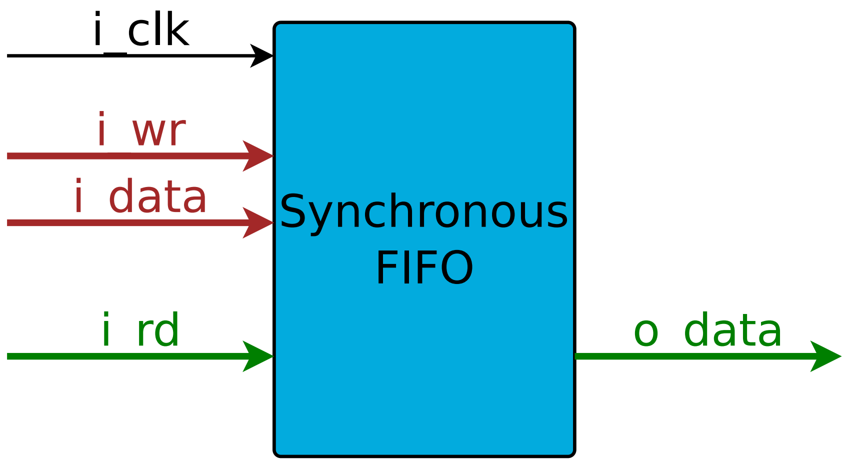
It's really just that simple

Basic FIFO Design



Let's spend a moment looking at the I/O ports of a FIFO

Basic FIFO Design

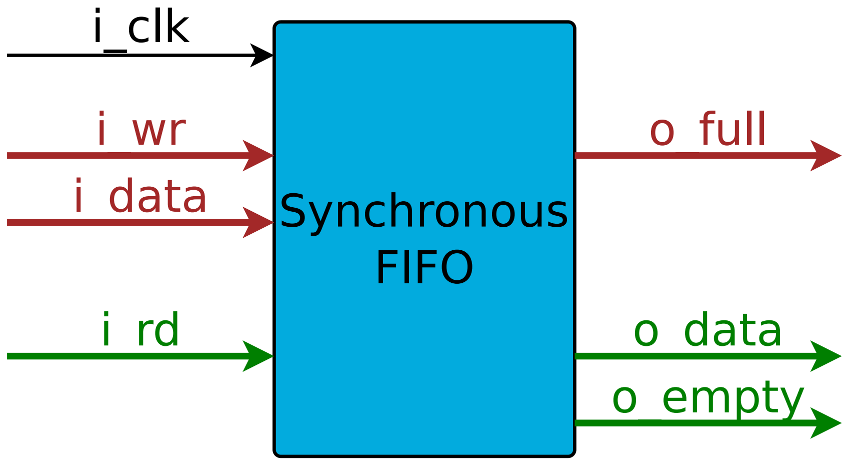


- On any `i_wr`, we'll write `i_data` to the FIFO
- On any `i_rd`, we'll return `o_data` from the FIFO

Not quite . . .

- What if there's nothing in the FIFO, should the read succeed?
- What if the FIFO is full, should a write succeed?

FIFO Interface

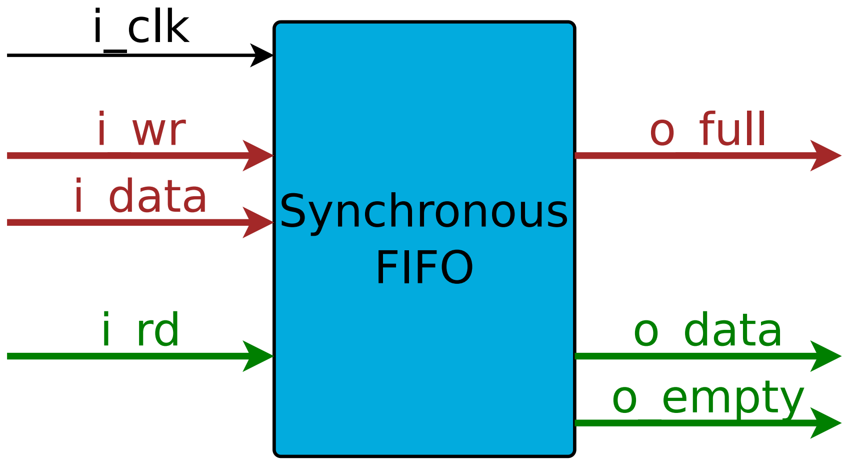


- On any `i_wr && !o_full`, we'll write `i_data` to memory
- On any `i_rd && !o_empty`, we'll read and return `o_data` from memory

We can simplify this by defining:

```
assign w_wr = i_wr && !o_full;
assign w_rd = i_rd && !o_empty;
```

FIFO



- On any `w_wr`, we'll write `i_data` to our internal memory
- On any `w_rd`, we'll read and return `o_rdata` from memory

This is how we'll handle overflows

- It should work much like our `i_wb_stb && !o_wb_stall` lesson
- The surrounding context must handle any over- or underflows

FIFO Memory

The two memory accesses constrain much of our logic

- Writes to the FIFO memory

```
// Maintain a write pointer
initial wr_addr = 0;
always @(posedge i_clk)
if (w_wr) // Increment on any write
        wr_addr <= wr_addr + 1;

// On any write, update the memory
always @(posedge i_clk)
if (w_wr)
        fifo_mem[wraddr] <= fifo_mem[i_data];
```

FIFO Memory

Our memories constrain much of our logic

- Writes to the FIFO memory
- Reads from the FIFO memory

```
// Maintain a read pointer
initial rd_addr = 0;
always @(posedge i_clk)
if (w_rd) // Increment on any read
        rd_addr <= rd_addr + 1;

// Return the value from the FIFO
//   found at the read address
always @(*)
        o_data <= fifo_mem[rd_addr];
```

Did you notice that this read violates our memory rules? We'll need to come back to this in a bit.

Address pointers

There's a trick to the addresses . . .

- For a memory of 2^N elements . . .
- With an N bit array index

  . . . you can only use $2^N - 1$ elements

  – Remember, you need to be able to tell the difference between empty (`wr_addr == rd_addr`) and full
- With an $N+1$ bit array index, you can use all 2^N elements

```
parameter BW = 8; // Bits per element
parameter LGFLEN = 8; // Memory size of 2^8
// Define the memory
reg [(BW-1):0]  mem [0:(1<<LGFLEN)-1];
// Give the pointers one extra bit
reg [LGFLEN:0]  wr_addr, rd_addr;
```

Address pointers

- With an $N+1$ bit array index, you can use all 2^N elements

```
reg  [(BW-1):0]   mem  [0:(1<<LGFLEN)-1];
reg  [LGFLEN:0]   wr_addr, rd_addr;
```

- The number of items in the FIFO is the address difference

```
always @(*)
        o_fill = wr_addr - rd_addr;
```

- We can now calculate our empty and full outputs

```
always @(*)
        o_empty = (o_fill == 0);
always @(*)
        o_full  = (o_fill == { 1'b1,
                        {(LGFLEN){1'b0}} });
```

Formal Checks

You should get in the habit of writing formal properties as you write your code

- Can you think of any appropriate properties from just these definitions?

Formal Checks

You should get in the habit of writing formal properties as you write your code

- Can you think of any appropriate properties from just these definitions?

Example: our fill can never exceed 2^N elements, so let's keep the solver from trying such a fill

```
always @(*)
        assert(o_fill <= { 1'b1,
                        {(LGFLEN){1'b0}} });
```

Formal Checks

You should get in the habit of writing formal properties as you write your code

- We might want to adjust our calculations of `o_fill`, `o_empty`, and `o_full` later
- Writing an assertion now might help make sure any rewrite later doesn't fundamentally change anything

```
always @(*)
        assert(o_fill  == wr_addr - rd_addr);
always @(*)
        assert(o_empty == (o_fill == 0));
always @(*)
        assert(o_full  == (o_fill == { 1'b1,
                      {(LGFLEN){1'b0}} });
```

Hint: one of the exercises in this lesson is to rewrite this logic

Not there yet

Our design isn't there yet

- We broke our memory rules
  - Our design will only work on *some architectures*

```
always @(*)
        o_data = fifo_mem[rd_addr];
```

  - This will work fine on any Xilinx FPGA
  - This won't map to block RAM on an iCE40
- Our logic all depends upon `w_wr` and `w_rd`
  - These values depend upon `o_full` and `o_empty`
  - These depend upon an N bit subtract and comparison
  - This will limit the total speed of our design

Let's come back to these issues after we go through simulation

Formal Verification

Review: Memory verification

- Let the solver pick a ~~random~~ address
- Follow the value at that address
- Verify the design works as intended

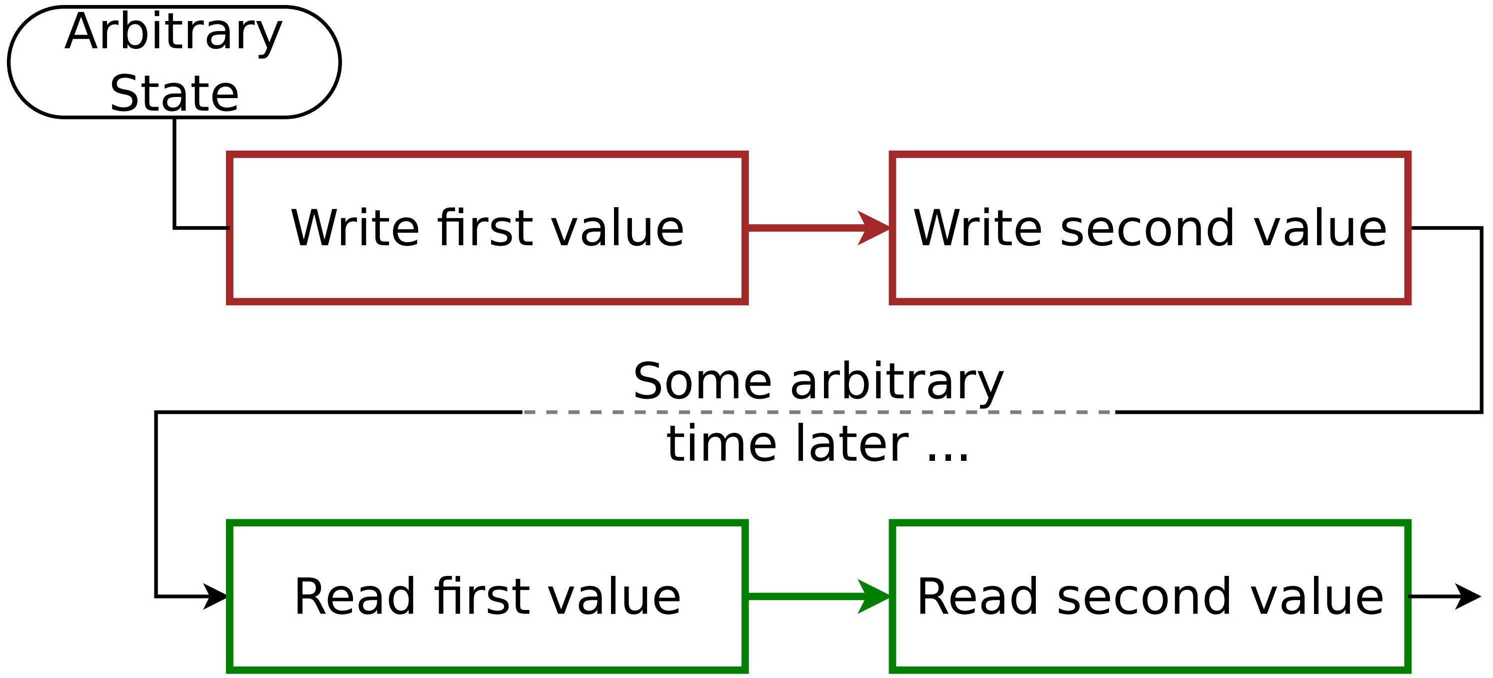
  for that address only

FIFO's offer a slight twist off of this strategy

FIFO Verification

To verify a FIFO

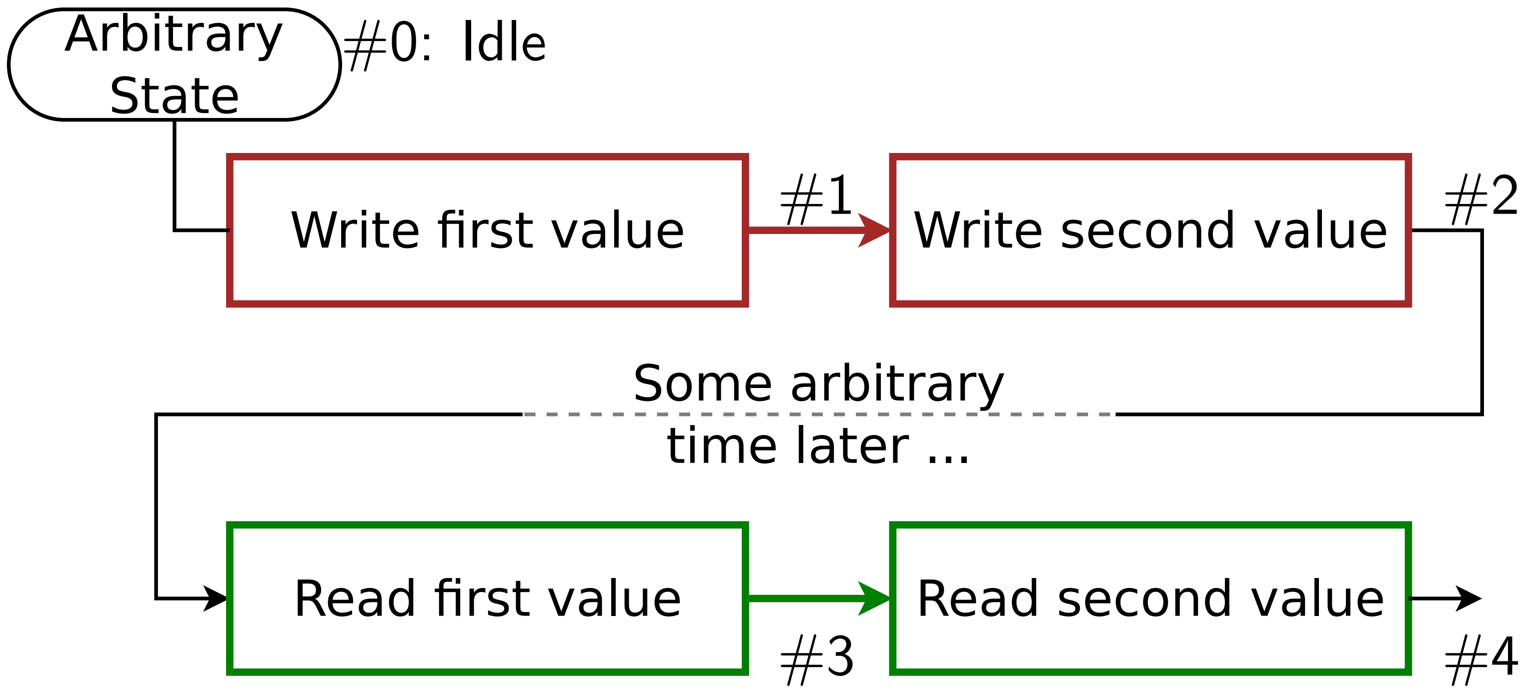
- Write two arbitrary values to it in succession
- Prove that you can then read those same values back later



FIFO Verification

To verify a FIFO

- Write two arbitrary values to it in succession
- Prove that you can then read those same values back later



Let's assign states!

FIFO Verification

We'll need two consecutive addresses

```
(* anyconst *) reg [LGFLEN:0] f_first_addr;
               reg [LGFLEN:0] f_second_addr;

always @(*)
        f_second_addr = f_first_addr + 1;
```

We'll also need two arbitrary values

```
(* anyconst *) reg [BW-1:0] f_first_data,
                        f_second_data;
```

FIFO Verification

Here's our basic state transitions:

```
always @(posedge i_clk)
case(f_state)
2'h0: // This is the IDLE state
        //
        // Our process starts when we write our
        // first value to the FIFO, at the
        // first of our chosen two addresses
        if (w_wr && (wr_addr == f_first_addr)
                && (i_data == f_first_data))
        f_state <= 2'h1;
// ...
endcase
```

FIFO Verification

Here's our basic state transitions:

```
always @(posedge i_clk)
case(f_state)
2'h0: // ...
2'h1: // If we read the first value out at
        // this stage, then abort our check
        if (w_rd && rd_addr == f_first_addr)
                f_state <= 2'h0;
        else if (w_wr)
        // Otherwise if we write the second
        // value then move to the next state.
        // If it's the wrong value then abort
        // the check
        f_state <= (i_data == f_second_data)
                ? 3'h2 : 2'h0;
// ...
endcase
```

FIFO Verification

Here's our basic state transitions:

```
always @(posedge i_clk)
case(f_state)
2'h0: // ...
2'h1: // ...
2'h2:   // Wait until we read the first value
        // back out of the FIFO
        if (i_rd && rd_addr == f_first_addr)
                // Then move forward by
                // one state
                f_state <= 2'h3;
// ...
endcase
```

FIFO Verification

Here's our basic state transitions:

```
always @(posedge i_clk)
case(f_state)
2'h0: // ...
2'h1: // ...
2'h2: // ...
2'h3:     // Finally, return to idle when the
          // last item is read
          if (i_rd) f_state <= 2'h0;
endcase
```

FIFO Verification

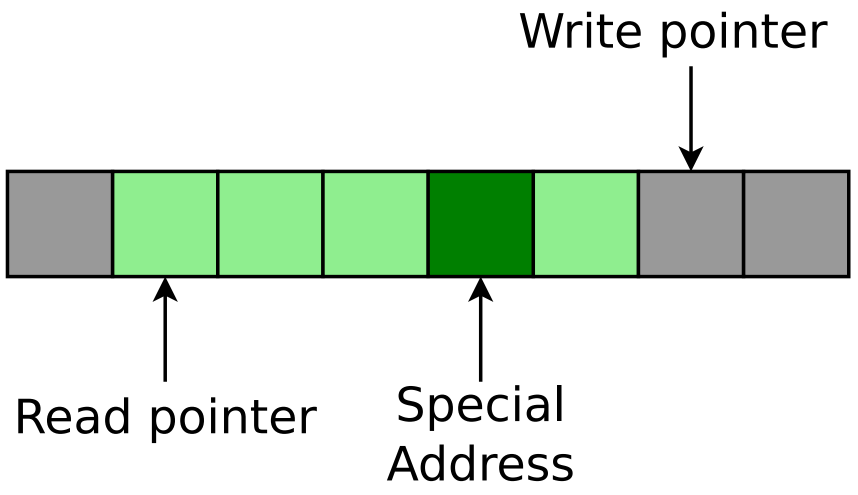
Basic proof:

- If we are in state one,
  - The first address must point to something within the FIFO
  - The memory at that location must be our special value
  - The write address must point to the second special address
- If we are in state two,
  - Both the first and second addresses must point to valid locations within the FIFO
  - The values at both locations must match our special values
- . . .

This is actually harder than it sounds

FIFO Verification

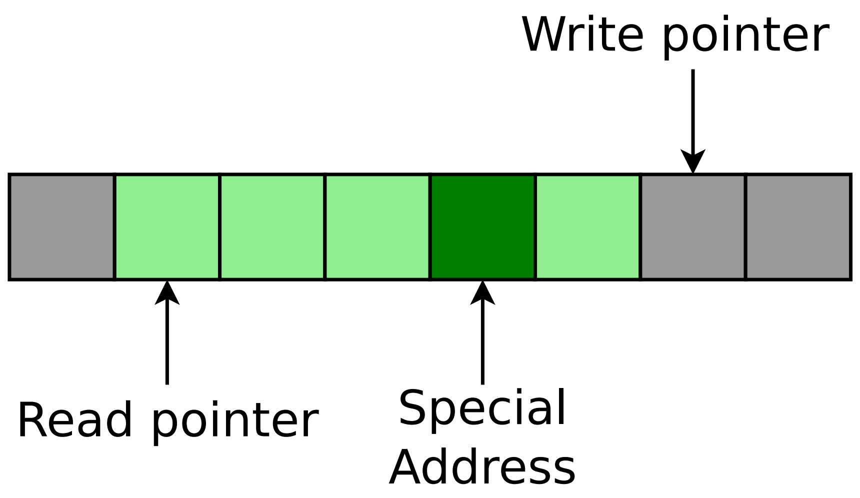
How shall we determine if our special address is within the FIFO?



- It must be greater or equal to the read address
- It must be less than the write address

FIFO Verification

How shall we determine if our special address is within the FIFO?



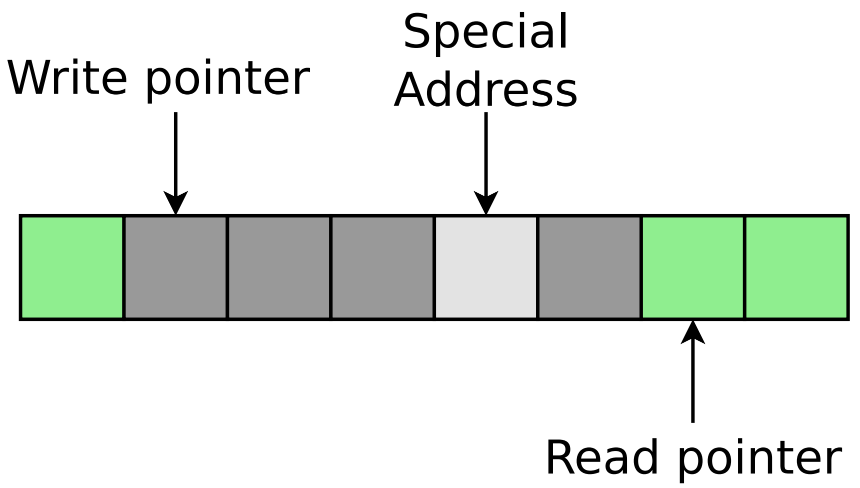
- It must be greater or equal to the read address
- It must be less than the write address

This is actually harder than it sounds

- *The pointers can wrap!*

FIFO Verification

How shall we determine if our special address is within the FIFO?



The pointers can wrap!

- What then does greater or less than mean?

Solution:

- Unwrap all the pointers by subtracting the read pointer

FIFO Verification

To determine if `f_first_addr` is within the active addresses of the FIFO:

```
reg [LGFLEN:0] f_distance_to_first;
always @(*)
begin
        // Unwrap by subtracting the distance
        // to the read address
        f_distance_to_first
                = (f_first_addr - rd_addr);
        // ...
end
```

`f_distance_to_first` can now be checked against the FIFO fill, to determine if the special address references a valid location within the FIFO

FIFO Verification

To determine if `f_first_addr` is within the active addresses of the FIFO:

```
reg [LGFLEN:0] f_distance_to_first;
always @(*)
begin
        // Check the distance into the FIFO
        // against the FIFO's fill level
        if ((!o_empty)
                &&(f_distance_to_first<o_fill))
                f_first_addr_in_fifo = 1;
        else
                f_first_addr_in_fifo = 0;
end
```

We'll need to do this to check both addresses

First state

Now we can create assertions for the first state

```
always @(*)
if (f_state == 2'b01)
begin
        // First value has been written
        assert(f_first_addr_in_fifo);
        assert(fifo_mem[f_first_addr]
                == f_first_data);
```

Don't forget, we must be waiting at the second address to write the second piece of data!

```
        assert(wr_addr == f_second_addr);
end
```

We now need to repeat this for the other two states

Second state

Here's the second state:

```
always @(*)
if (f_state == 2'b10)
begin
        // First value must be in the FIFO
        assert(f_first_addr_in_fifo);
        assert(fifo_mem[f_first_addr]
                == f_first_data);

        // Second value too!
        assert(f_second_addr_in_fifo);
        assert(fifo_mem[f_second_addr]
                == f_second_data);

        if (rd_addr == f_first_addr)
                assert(o_data == f_first_data);
end
```

Last state

Here's the third and last state

```
always @(*)
if (f_state == 2'b11)
begin
        // Only the second value need be
        // in the FIFO
        assert(f_second_addr_in_fifo);
        assert(fifo_mem[f_second_addr]
                == f_second_data);

        // The output data must match our
        // second value until the next w_rd
        assert(o_data == f_second_data);
end
```

SymbiYosys

You should now be able to prove this design via induction

- Does it pass?

Cover

You should also create some cover properties. Here are some of mine:

```
initial f_was_full = 0;
always @(posedge i_clk)
if (o_full)
        f_was_full <= 1;

always @(posedge i_clk)
        cover(f_was_full && f_empty);

always @(posedge i_clk)
        cover($past(o_full,2)
                &&(!$past(o_full))&&(o_full));
```

Of course, these will only pass in a reasonable time if the memory size is small

Cover Lesson Learned

I was once burned by going through all the motions of formal verification, only to have the design fail in simulation

- The design was a data cache
- I had verified both interfaces
  - All the bus properties were met
  - The CPU could depend upon the resulting interface
- I covered the return from a request
  - The cache went to the bus to get the requested value
  - The values returned by the bus were properly placed into the cache
  - The core then returned the right value

The resulting code failed in practice

Cover Lesson Learned

I was once burned by going through all the motions of formal verification, only to have the design fail in simulation

- The design was a data cache
- I had verified both interfaces
  - All the bus properties were met
  - The CPU could depend upon the resulting interface
- I covered the return from a request
  - The cache went to the bus to get the requested value
  - The values returned by the bus were properly placed into the cache
  - The core then returned the right value

The resulting code failed in practice

- What went wrong?

Cover Lesson Learned

The problem?

- The cache never raised its ready line to indicate it was ready for the next request
- Once it became busy, it never returned to idle
- The CPU froze on its second memory access

Cover Lesson Learned

The problem?

- The cache never raised its ready line to indicate it was ready for the next request
- Once it became busy, it never returned to idle
- The CPU froze on its second memory access

Ever since this painful lesson, I've made a point to cover the return to an idle state following the covered state

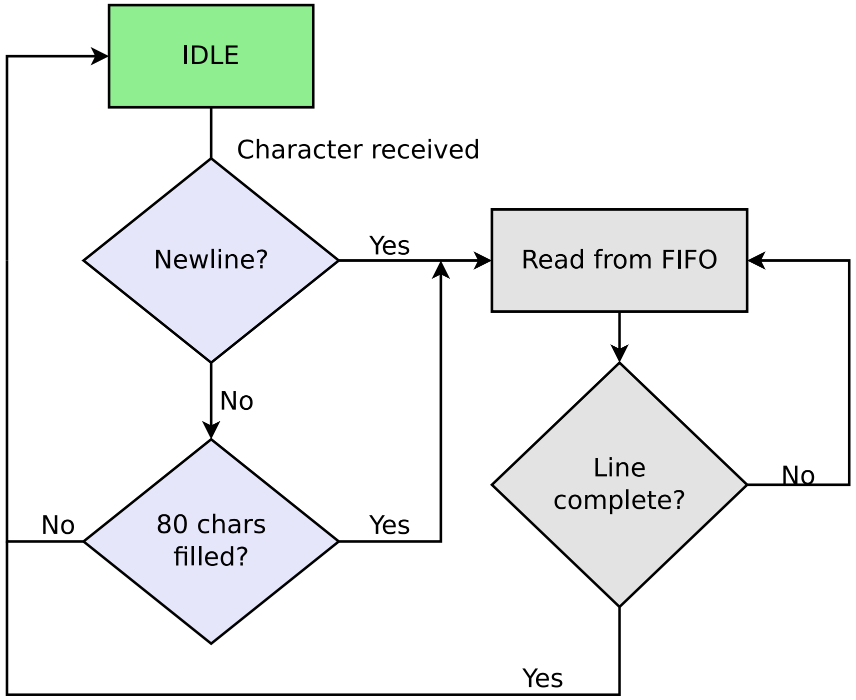
- That's the reason for the `f_empty` check in the cover statement below

```
always @(posedge i_clk)
        cover(f_was_full && f_empty);
```

Line Capturer

We now have a FIFO, what shall we do with it?

- Let's build a line capturer



Using the FIFO

We now have a FIFO, what shall we do with it?

- Let's build a line capturer

1. When it receives a character from the serial port, it places it into the FIFO
2. Once either the line has reached (or past 80) characters, we'll dump the FIFO to the serial port transmitter
3. Likewise, on any new line, we'll dump the FIFO to the serial port transmitter.

Think of this like an old fashioned printer:

- Once the print head starts moving from left to right, it moves across the page at a constant speed
- You don't want to start the head moving until a whole line is available

Rx + FIFO

Step one: the receiver must feed the FIFO

```
// Serial port Receiver
rxuart  #(.CLOCKS_PER_BAUD(CLOCKS_PER_BAUD))
        receiver(i_clk, i_uart_rx, rx_stb,
                rx_data);

// Our synchronous FIFO
//    Fed directly from the receiver
sfifo   #(.BW(8), .LGFLEN(8))
        fifo(i_clk, rx_stb, rx_data, fifo_full,
                fifo_fill,
                fifo_rd, tx_data, fifo_empty);
```

Rx + FIFO

Step two: Build a basic FSM to control the FIFO reads

- We'll use `run_tx` to say we are currently transmitting
- `line_count` captures the number of items left to write

```
initial run_tx = 0;
initial line_count = 0;
always @(posedge i_clk)
if (rx_stb && (rx_data == 8'ha
                || rx_data == 8'hd))
begin // Start reading on any received newline
        // or carriage return
        run_tx <= 1'b1;
        line_count <= fifo_fill[7:0];
end else if (!run_tx)
begin
// ....
```

Rx + FIFO

Step two: Build a basic FSM to control the FIFO reads

- We'll use `run_tx` to say we are currently transmitting
- `line_count` captures the number of items left to write

```
// ...
end else if (!run_tx)
begin // If we're not currently running
        if (fifo_fill >= 9'd80)
        begin // Start running when a
                // full line has been
                run_tx <= 1'b1; // received
                line_count <= 80;
// ...
```

Rx + FIFO

Step two: Build a basic FSM to control the FIFO reads

- We'll use `run_tx` to say we are currently transmitting
- `line_count` captures the number of items left to write

```
// ...
end else if (!fifo_empty && !tx_busy)
begin // If we are running, then keep going
        // until our line_count gets down
        // to zero
        line_count <= line_count - 1;
        if (line_count == 1)
                run_tx <= 0;
end
```

Rx + FIFO

Last steps:

- Read from the FIFO any time we send to the transmitter

```
always @(*)
        fifo_rd = (tx_stb && !tx_busy);
```

- Activate the transmitter anytime `run_tx` is true

```
always @(*)
        tx_stb = (run_tx && !fifo_empty);
```

Rx + FIFO

Last steps:

- Read from the FIFO any time we send to the transmitter

```
always @(*)
        fifo_rd = (tx_stb && !tx_busy);
```

- Activate the transmitter anytime `run_tx` is true

```
always @(*)
        tx_stb = (run_tx && !fifo_empty);
```

Q: Can the check for whether the FIFO is empty be removed?

Rx + FIFO

Last steps:

- Read from the FIFO any time we send to the transmitter

```
always @(*)
        fifo_rd = (tx_stb && !tx_busy);
```

- Activate the transmitter anytime `run_tx` is true

```
always @(*)
        tx_stb = (run_tx && !fifo_empty);
```

Q: Can the check for whether the FIFO is empty be removed?
Q: How would you know? Would a formal check help?

Verifying the FSM

You should be able to formally verify that this works

- Don't reverify the FIFO
- Consider letting the solver pick the output of the FIFO

```
(* anyseq *) reg formal_data;
always @(*)
        o_data = formal_data;
```

- Focus on the FIFO flags

What properties would you use to verify this FSM design?

- Don't forget to abstract the serial ports
- You may need to assume the receiver is slower than the transmitter

Simulation

At this point, you know all that is needed to build a simulation

- We have a serial port transmitter, and simulation
- We have a serial port receiver, and simulation
- We've learned to interact with our design over an O/S pipe that communicates with a child process running the Verilator based simulation
- Indeed, the simulation should look very similar to the one from our last lesson

What more might we need?

Simulation

At this point, you know all that is needed to build a simulation

- We have a serial port transmitter, and simulation
- We have a serial port receiver, and simulation
- We've learned to interact with our design over an O/S pipe that communicates with a child process running the Verilator based simulation
- Indeed, the simulation should look very similar to the one from our last lesson

What more might we need?
There's one problem: the simulation trace reveals that . . .

- The last simulation doesn't really exercise our design

Let's fix this!

Random Delay

Here's the key: let's add a random delay between incoming bytes

- We'll use **m_delay** to capture this delay
- We'll drain it to zero before sending a new character

```
int UARTSIM::operator()(const int i_tx) {
        // ...
        if (m_tx_state == TXIDLE)
                if (m_delay > 0) {
                // Wait for a delay to complete
                // before checking for a new
                // data byte
                        m_delay--;
                } else {
                // Continue as before and ask
                //  the O/S for new data byte
```

Random Delay

Here's the key: let's add a random delay between incoming bytes

- We'll use **m_delay** to capture this delay
- We'll create a random delay at the end of every transmitted character

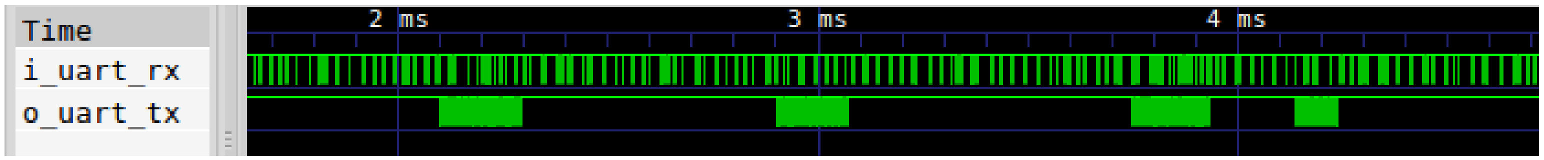
```
int UARTSIM::operator()(const int i_tx) {
        // ...
        } else if (m_tx_baudcounter <= 0) {
            if (!m_tx_busy) {
                // Return to idle
                m_tx_state = TXIDLE;
                if ((rand() & 0x1f)>12)
                    m_delay = (rand() & 0x07f)
                            * m_baud_counts;
        // ...
```

Simulation

You should now be able to run the simulation

Sim Trace

Here's the trace I got from running the simulation



- Notice the pseudorandom incoming byte stream, and
- The very bursty transmit stream

This was exactly what we wanted!

Fixing the read

What we've built is a *first word fall through* FIFO

- That won't work on an iCE40
  - Because all block RAM reads on an iCE40 *must* be registered
  - The hardware doesn't exist on an iCE40 to do otherwise
- This was our problem code

```
always @(*)
        o_data <= fifo_mem[rd_addr];
```

How shall we fix this?

Fixing the read

Solution one: bypass the memory

- On a write, store the value in memory and in a register
- On the next clock, offer the register value as a result
- We'll also need to adjust the read address
  - The memory read address needs to be the address it will be on the *next clock*

Let's see what this would look like

Bypass memory

Our basic logic will be to capture two memory values, and select between them

- The first is placed in a register

```
always @(posedge i_clk)
        bypass_data <= i_data;
```

- The second is read from the memory

```
always @(*)
        rd_next = rd_addr[LGFLEN-1:0] + 1;

always @(posedge i_clk)
        rd_data <= mem[(w_rd)?rd_next
                : rd_addr[LGFLEN-1:0]];
```

Bypass memory

Our basic logic will be to capture two memory values, and select between them

- The first is placed in a register
- The second is read from the memory
- Then we select between them

```
always @(*)
        o_data = (bypass_valid) ? bypass_data
                : rd_data;
```

Bypass memory

The trick is the selector, `bypass_valid`, that tells us which value to return

```
initial bypass_valid = 0;
always @(posedge i_clk)
begin
        bypass_valid <= 1'b0;
        if (!i_wr)
                // If we haven't written to the
                // FIFO in the last cycle, the
                // memory read will be good
                bypass_valid <= 1'b0;
        // ...
end
```

Bypass memory

The trick is the selector, `bypass_valid`, that tells us which value to return

```
initial bypass_valid = 0;
always @(posedge i_clk)
begin
        bypass_valid <= 1'b0;
        if (!i_wr)
                bypass_valid <= 1'b0;
        else if (o_empty ||(i_rd&&(o_fill == 1)))
                // Otherwise if we read, and the
                // memory is now empty, use the
                // register value
                bypass_valid <= 1'b1;
        // Remember, the last assignment wins
end
```

You can use formal methods to prove the result is the same

Fixing the read

Solution two: Work with the registered output

- We could just use the registered data

```
always @(posedge i_clk)
        o_data <= mem[(w_rd)?rd_next
                : rd_addr[LGFLEN-1:0]];
```

- The biggest problem would be our empty FIFO logic
  - It would need to be delayed by one clock

```
initial o_empty = 1;
always @(posedge i_clk)
if ((o_fill > 1)||((o_fill == 1)&&(!w_rd)))
        o_empty <= 1'b0;
else
        o_empty <= 1'b1;
```

Exercise #1

Pick a solution to our memory problem and

- Formally verify it
- Prove that it still meets the two write/two read criteria of a FIFO

Exercise #2

All of our `o_fill` and `o_full` logic is combinatorial

- This will prevent keep us from using our FIFO in a high speed design
- Solution: Rewrite this logic so that it's registered

```
always @(posedge i_clk)
        o_fill <= // Your logic here

always @(posedge i_clk)
        o_full <= // Your logic here
```

- Use the formal solver to formally prove that it still meets the properties required of `o_fill` and `o_full`

Exercise

Are you ready? Let's build this!

- Create this design, and place it on your FPGA
- You've already formally verified and simulated it, right?
  - So . . . it should work on the hardware the first time, right?

(Guilty admission: Mine still didn't work the first time . . .)

Questions

- Many serial port implementations use RTS and CTS signals
  - How might you use the FIFO level to stop an upstream transmitter when the FIFO was (nearly) full? Don't forget these things don't stop on a dime.
  - How might a downstream receiver signal to this design to stop transmitting, since its FIFO was (nearly) full?
- Most packet format end with a CRC
  - How would you modify this design to add and check a CRC?
- Many packet formats have either a fixed length, or a length specifier
  - How would a variable packet length change things?

Conclusion

What did we learn this lesson?

- What a FIFO is, and why you might use one
- How to formally verify a FIFO
- Some of the problems associated with reading the data from a FIFO on different pieces of hardware
- How to eliminate combinatorial logic, while making sure that the design functionality doesn't change