



Gisselquist  
Technology, LLC

## 5. Serial Port Transmitter

Daniel E. Gisselquist, Ph.D.





# Lesson Overview



- ▷ Lesson Overview
- Serial Protocol
- Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

Let's see if we can do Hello World

- If you can do the LED sequencer, you can do this project
- We'll be building a two module design
- And some awesome simulation capability

## Objectives

- Be able to transmit Hello World!
- Clean up our Verilator work
- Simulate a serial port receiver



# Serial Protocol



Lesson Overview

➤ Serial Protocol

Implementation

Submodules

Top Level

Philosophy

Simulation

Main simulation file

Cosimulation

Make Foo

Formal Verification

Verifying txuart

Formal Contract

Exercise #1

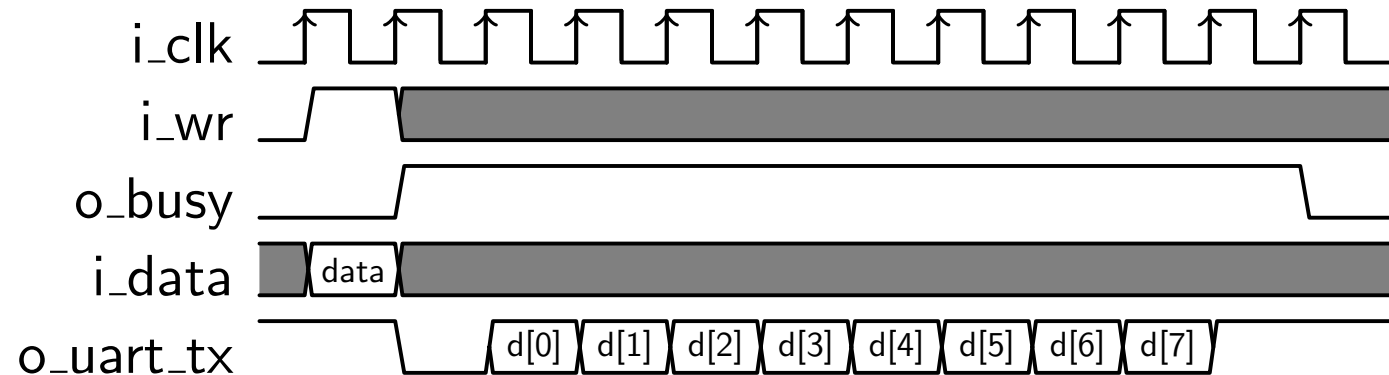
Hello World

Exercise #2

Hardware!

Conclusion

Let's transmit a character



A serial transmission ...



# Serial Protocol



Lesson Overview

➤ Serial Protocol

Implementation

Submodules

Top Level

Philosophy

Simulation

Main simulation file

Cosimulation

Make Foo

Formal Verification

Verifying txuart

Formal Contract

Exercise #1

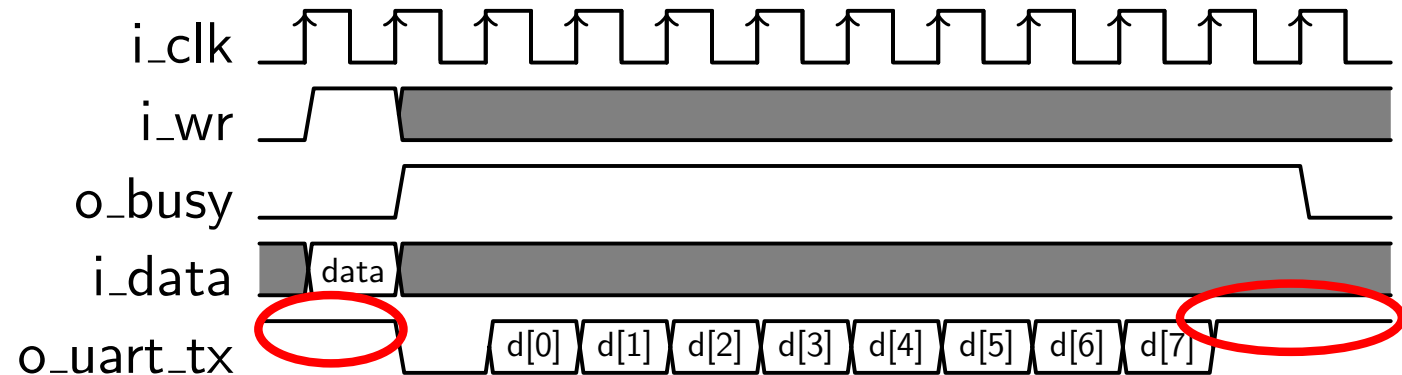
Hello World

Exercise #2

Hardware!

Conclusion

Let's transmit a character



A serial transmission ...

- Idles high



# Serial Protocol



Lesson Overview

➤ Serial Protocol

Implementation

Submodules

Top Level

Philosophy

Simulation

Main simulation file

Cosimulation

Make Foo

Formal Verification

Verifying txuart

Formal Contract

Exercise #1

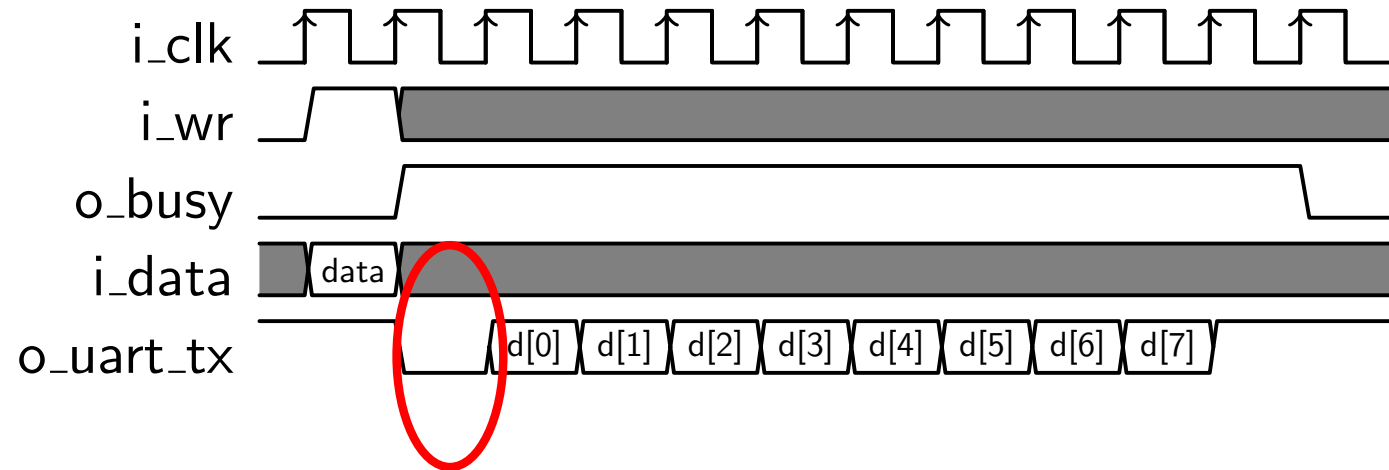
Hello World

Exercise #2

Hardware!

Conclusion

Let's transmit a character



A serial transmission ...

- Idles high
- Begins with a start bit (low)



# Serial Protocol



Lesson Overview

➤ Serial Protocol

Implementation

Submodules

Top Level

Philosophy

Simulation

Main simulation file

Cosimulation

Make Foo

Formal Verification

Verifying txuart

Formal Contract

Exercise #1

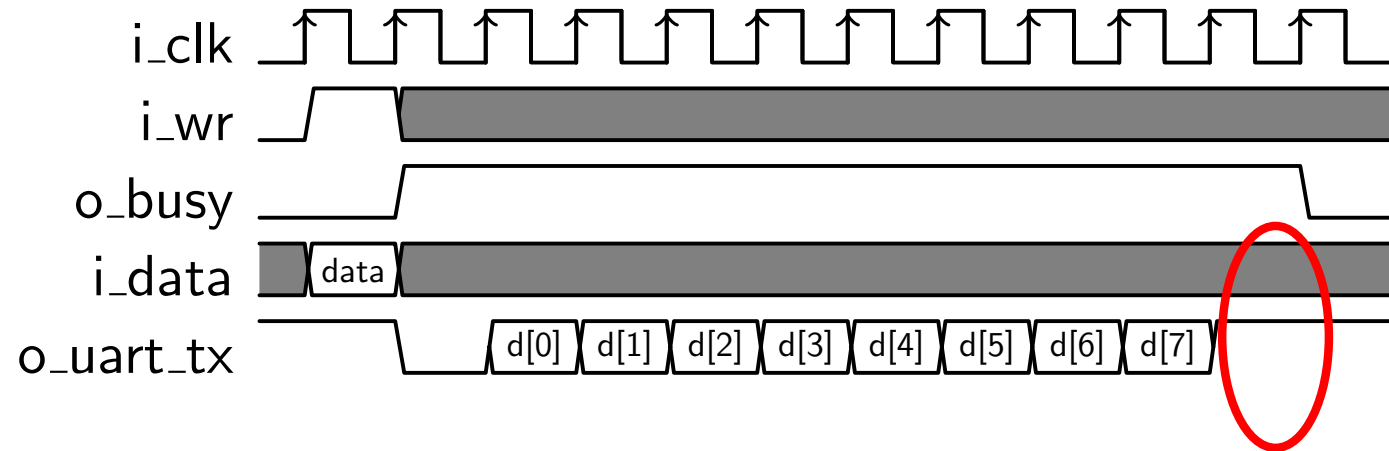
Hello World

Exercise #2

Hardware!

Conclusion

Let's transmit a character



A serial transmission ...

- Idles high
- Begins with a start bit (low), ends with a stop bit (high)



# Serial Protocol



Lesson Overview

➤ Serial Protocol

Implementation

Submodules

Top Level

Philosophy

Simulation

Main simulation file

Cosimulation

Make Foo

Formal Verification

Verifying txuart

Formal Contract

Exercise #1

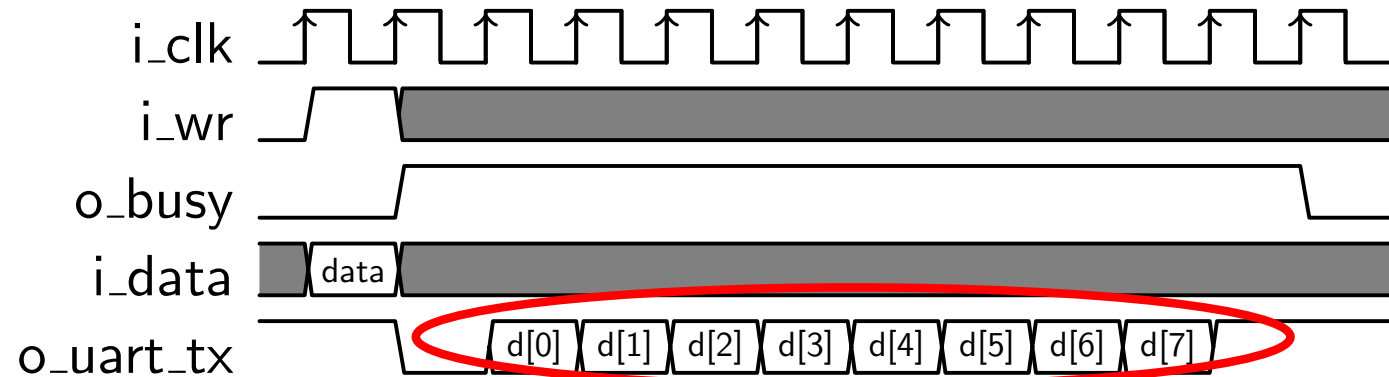
Hello World

Exercise #2

Hardware!

Conclusion

Let's transmit a character



A serial transmission ...

- Idles high
- Begins with a start bit (low), ends with a stop bit (high)
- Sends a byte of data, LSB first

Do this, and you will have a serial port transmitter



# Goal



Lesson Overview

➤ Serial Protocol

Implementation

Submodules

Top Level

Philosophy

Simulation

Main simulation file

Cosimulation

Make Foo

Formal Verification

Verifying txuart

Formal Contract

Exercise #1

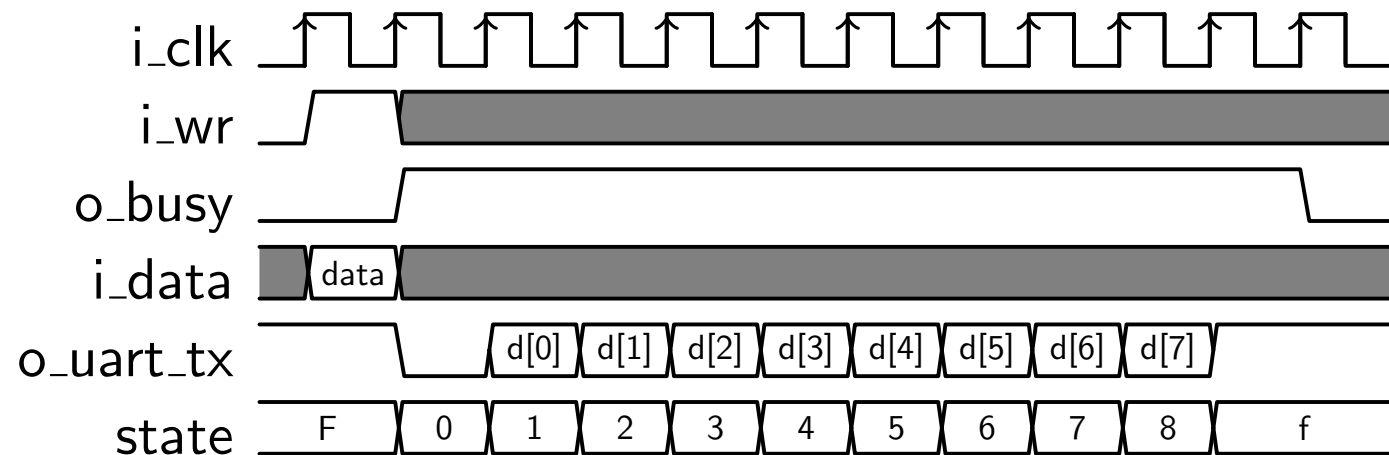
Hello World

Exercise #2

Hardware!

Conclusion

Let's add state ID's to this diagram



This will work for now

- Ten states to our state machine
- We'll still need to slow it down later





# State Variable



We can set o\_busy together with our state

```
initial { o_busy, state } = { 1'b0, IDLE };  
always @(posedge i_clk)  
if ((i_wr)&&(!o_busy))  
    // Start a new byte  
    { o_busy, state } <= { 1'b1, START };  
else if (state == IDLE)  
    // Stay in IDLE  
    { o_busy, state } <= { 1'b0, IDLE };  
else if (state < LAST)  
begin  
    o_busy <= 1'b1;  
    state <= state + 1;  
end else // Return to IDLE  
    { o_busy, state } <= { 1'b1, IDLE };
```

Is this a Mealy or a Moore FSM?

- Lesson Overview
- Serial Protocol
  - ▷ Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion



# Outgoing Data



- Lesson Overview
- Serial Protocol
  - ▷ Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

The outgoing data is just a shift register

```
initial lcl_data = 8'hff;
always @(posedge i_clk)
if ((i_wr)&&(!o_busy))
    // Load the register
    // Start outputting a zero
    lcl_data <= { i_data, 1'b0 };
else
    // Shift right for more data
    // Shift 1'b1 in from the left
    lcl_data <= { 1'b1, lcl_data };

assign o_uart_tx = lcl_data[0];
```

The output depends upon state only



# Outgoing Data



- Lesson Overview
- Serial Protocol
  - ▷ Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

The outgoing data is just a shift register

```
initial lcl_data = 8'hff;
always @(posedge i_clk)
if ((i_wr)&&(!o_busy))
    // Load the register
    // Start outputting a zero
    lcl_data <= { i_data, 1'b0 };
else
    // Shift right for more data
    // Shift 1'b1 in from the left
    lcl_data <= { 1'b1, lcl_data };

assign o_uart_tx = lcl_data[0];
```

The output depends upon state only

- *This is a Moore FSM*



# Clock divider



- Lesson Overview
- Serial Protocol
  - ▷ Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

All that remains is an integer clock divider!

- We'll adjust our logic above to only change on `baud_stb`
- ...or (if idle) on `(i_wr)&&(!o_busy)`

```
initial counter = 0;
always @(posedge i_clk)
if ((i_wr)&&(!o_busy))
    counter <= CLOCKS_PER_BAUD - 1;
else if (counter > 0)
    counter <= counter - 1;
else if (state != IDLE)
    counter <= CLOCKS_PER_BAUD - 1;

assign baud_stb = (counter == 0);
```

Is counter a state variable?



# Clock divider



- Lesson Overview
- Serial Protocol
  - ▷ Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

All that remains is an integer clock divider!

- We'll adjust our logic above to only change on `baud_stb`
- ...or (if idle) on `(i_wr)&&(!o_busy)`

```
initial counter = 0;
always @(posedge i_clk)
if ((i_wr)&&(!o_busy))
    counter <= CLOCKS_PER_BAUD - 1;
else if (counter > 0)
    counter <= counter - 1;
else if (state != IDLE)
    counter <= CLOCKS_PER_BAUD - 1;

assign baud_stb = (counter == 0);
```

Is counter a state variable? *Yes, even if it isn't so named*



# A Common Mistake



- Lesson Overview
- Serial Protocol
  - ▷ Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

All that remains is an integer clock divider!

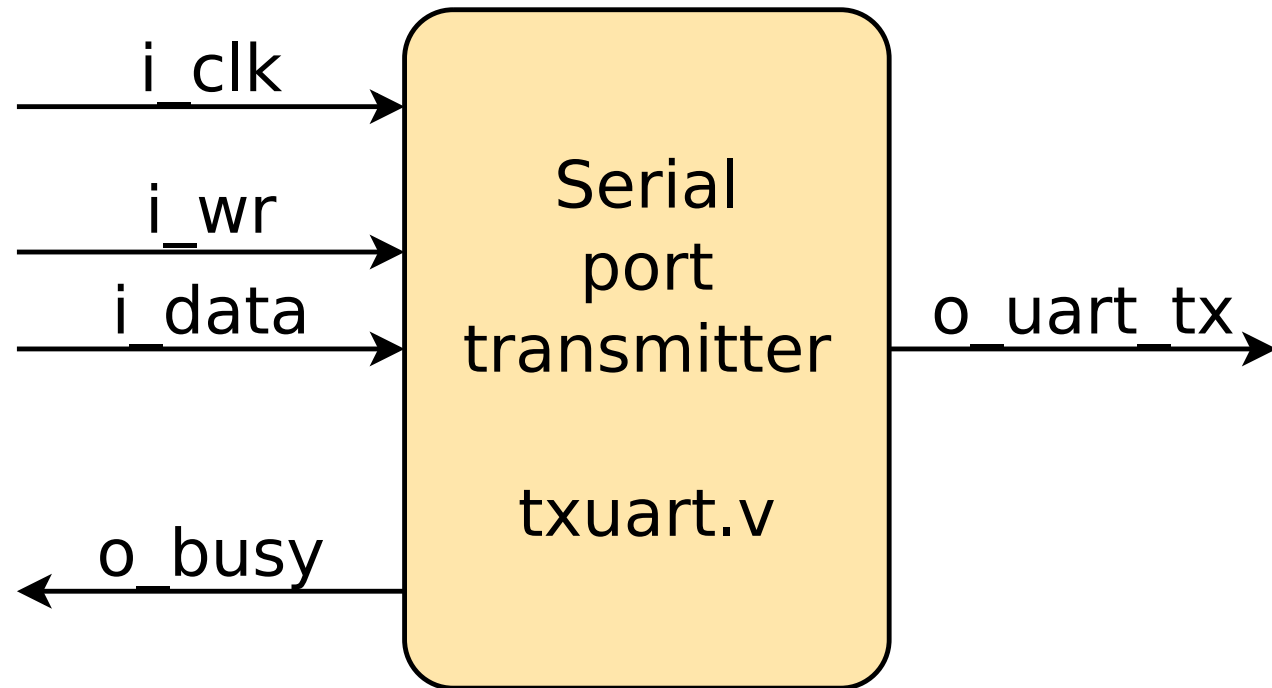
- We'll adjust our logic above to only change on `baud_stb`
- ... or (if idle) on `(i_wr)&&(!o_busy)`

A common mistake is to condition the first transition on more than `(i_wr)&&(!o_busy)`

- This risks another condition taking priority over `(i_wr)&&(!o_busy)`
- Result is that the transmitter doesn't notice the transmit request
- This mistake can usually be caught using formal methods.



# A Component

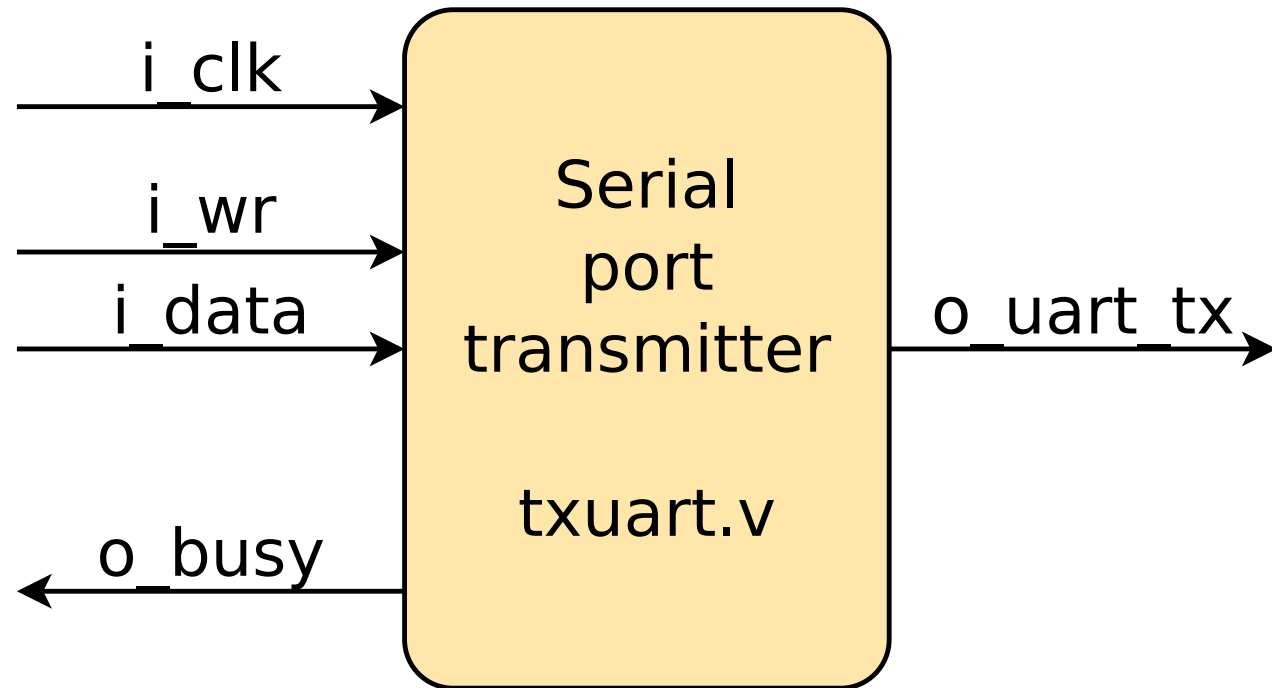


- `i_wr` requests a character (`i_data`) be transmitted
- Whenever `o_busy` is true, `i_wr` is ignored
- `i_data` is queued for transmission when `(i_wr && !o_busy)`

- Lesson Overview
- Serial Protocol
  - ▷ Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion



# A Component



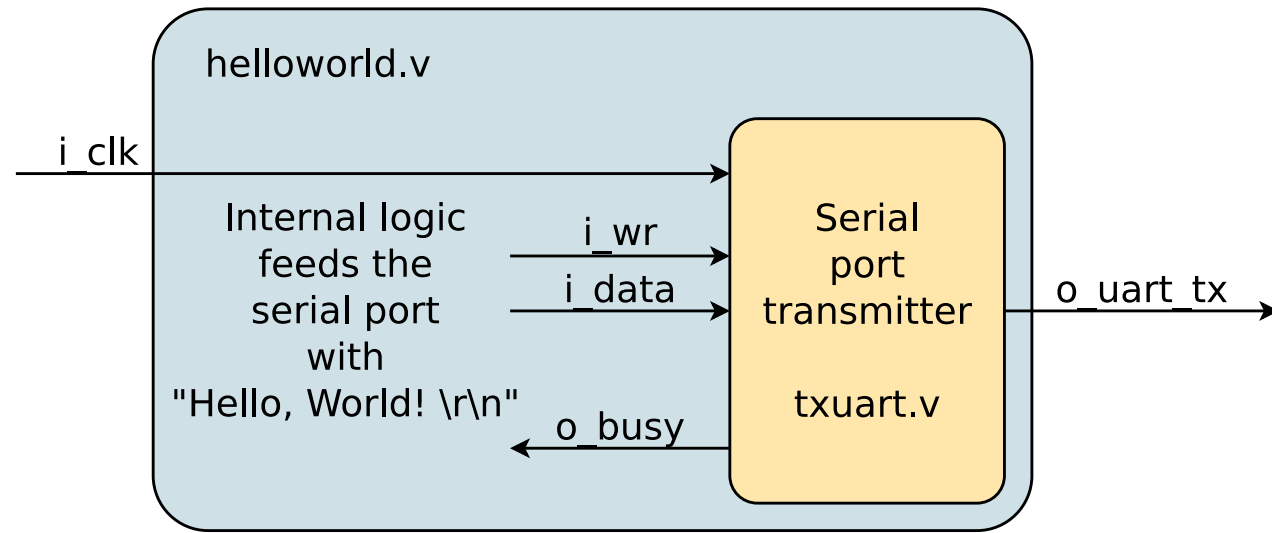
A good serial port

- Can be used again and again
- From one design to the next





# Submodules



Just like a printed circuit board (PCB)

- Logic from one component can be used within another
- Akin to placing multiple chips on a PCB
- Each module is typically called a *core*
- It's possible to have multiple copies of the same module
- You can also place cores within cores within cores, etc.



# Modules



- Lesson Overview
- Serial Protocol Implementation
  - ▷ Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

Two methods to use one module within another

## 1. Pass by ordered-list

```
txuart    #(CLOCK_RATE_HZ / MYBAUDRATE)
          mytxuart(clk, tx_stb, tx_data, o_uart,
                  tx_busy);
```

- Ports must be given in order, and cannot be skipped
- The name of your new module, `mytxuart` must be unique within its context
- Inputs to the module can come from be either wires or registers
- Outputs from the module must be placed into wires
- Optionally, parameters within the module can be overridden

These are found in the `#(...)` block

Like the portlist, these can be done in matching order



# Modules



- Lesson Overview
- Serial Protocol Implementation
  - ▷ Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

Two methods to use one module within another

1. Pass by port-order
2. Pass by port name

```
txuart    #(.CLOCKS_PER_BAUD(CLOCK_RATE_HZ
              / MYBAUDRATE))
          mytxuart(.i_clk(clk),
                  .i_wr(tx_stb),      .i_data(tx_data),
                  .o_busy(tx_busy),  .o_uart_tx(o_uart));
```

- Ports and parameters may now be in any order
- They may also (optionally) be skipped
- You cannot mix calling conventions
  - Either pass by port-order, or pass by port-name
  - Never both



# Top Level



- Lesson Overview
- Serial Protocol Implementation
- Submodules
  - ▷ Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
  - Verifying txuart
  - Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

We'll need a message.

```
always @(posedge i_clk)
case(tx_index)
4'h0: tx_data <= "H"; // Could also use a memory
4'h1: tx_data <= "e"; // here
4'h2: tx_data <= "l";
4'h3: tx_data <= "l"; // Because this case is so
4'h4: tx_data <= "o"; // small, it is equivalent
4'h5: tx_data <= ","; // to a memory
4'h6: tx_data <= "_";
4'h7: tx_data <= "W";
4'h8: tx_data <= "o";
// ...
4'he: tx_msg <= "\r";
4'hf: tx_msg <= "\n";
endcase
```



# Hello World



Lesson Overview

Serial Protocol

Implementation

Submodules

▷ Top Level

Philosophy

Simulation

Main simulation file

Cosimulation

Make Foo

Formal Verification

Verifying txuart

Formal Contract

Exercise #1

Hello World

Exercise #2

Hardware!

Conclusion

If we want our serial port to run Hello World, ...

- it needs a driver, helloworld.v

```
// tx_index tells us what character to send next
always @(posedge i_clk)
  if ((tx_wr)&&(!tx_busy))
    tx_index <= tx_index + 1'b1;

// tx_stb requests a character be sent
always @(posedge i_clk)
  if (tx_restart)
    tx_stb <= 1'b1;
  else if ((tx_stb)&&(!tx_busy)&&(tx_index == 4'hf))
    tx_stb <= 1'b0; // Wait for next second
```

We'll need to restart this periodically



# Hello World



Lesson Overview

Serial Protocol

Implementation

Submodules

▷ Top Level

Philosophy

Simulation

Main simulation file

Cosimulation

Make Foo

Formal Verification

Verifying txuart

Formal Contract

Exercise #1

Hello World

Exercise #2

Hardware!

Conclusion

If we want our serial port to run Hello World

- it needs a driver, helloworld.v
- It needs to be periodically restarted

```
// Integer clock divider
initial hz_counter = 28'h16;
always @(posedge i_clk)
if (counter == 0)
    hz_counter <= CLOCK_RATE_HZ - 1'b1;
else
    hz_counter <= hz_counter - 1'b1;

// And the once / sec restart signal
initial tx_restart = 0;
always @(posedge i_clk)
    tx_restart <= (hz_counter == 1);
```



# Philosophy



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- ▷ Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

Most HDL/FPGA courses stop here

- You have no way of knowing if you did it right other than hardware test
- You can only debug using LED's
- When it doesn't work, you'll never know why not
- They don't teach you to use
  - Simulation, or
  - Formal methodsto find latent bugs in your design

The result is a lesson in frustration, rather than a celebration of success



# Philosophy



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- ▷ Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

Most HDL/FPGA courses stop here

- You have no way of knowing if you did it right other than hardware test
- You can only debug using LED's
- When it doesn't work, you'll never know why not
- They don't teach you to use
  - Simulation, or
  - Formal methodsto find latent bugs in your design

The result is a lesson in frustration, rather than a celebration of success

*We can do better!*





# Philosophy



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- ▷ Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

Most HDL/FPGA courses stop here. We'll keep going.

- Let us continue, and learn how
  1. Simulate, then
  2. Formally verifythis design



# Simulation



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
  - ▷ Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

Our simulation is getting so big it is becoming annoying

- On every tick, we need to keep track of
  - Current time (i.e. the number of clock ticks so far)
  - The pointer to the Verilated Verilog code
  - The pointer to our C++ trace object
- This means we either
  - Pass lots of parameters around
  - Keep multiple global variables
  - Use a C++ class that keeps variables with the methods that use them

Solution: a reusable Verilator template class!



# Verilator template



Most of this task is just rearranging our simulation code

```
template <class VA> class TESTB {
public:
    VA                                *m_core;
    VerilatedVcdC                    *m_trace;
    uint64_t                          *m_tickcount;

    TESTB(void) : m_trace(NULL),
                  m_tickcount(0) {
        m_core = new VA;
        Verilated::traceEverOn(true);
        m_core->i_clk = 0;
        eval();
    }
    // ...
}
```



# Verilator template



Most of this task is just rearranging our simulation code

```
template <class VA> class TESTB {  
public:  
    VA *m_core;  
    VerilatedVcdC *m_trace;  
    uint64_t *m_tickcount;  
  
    TESTB(void) : m_trace(NULL),  
                  m_tickcount(0) {  
        m_core = new VA;  
        Verilated::traceEverOn(true);  
        m_core->i_clk = 0;  
        eval();  
    }  
    // ...  
};
```

Use a template class to only do this once



# Verilator template



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- ▷ Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

Most of this task is just rearranging our simulation code

```
template <class VA> class TESTB {
public:
    VA                *m_core;
    VerilatedVcdC     *m_trace;
    uint64_t          *m_tickcount;

    TESTB(void) : m_trace(NULL),
                  m_tickcount(0) {
        m_core = new VA;
        Verilated::traceEverOn(true);
        m_core->i_clk = 0;
        eval();
    }
    // ...
}
```

Put our three trace variables here



# Verilator template



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- ▷ Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

Most of this task is just rearranging our simulation code

```
template <class VA> class TESTB {
public:
    VA                *m_core;
    VerilatedVcdC     *m_trace;
    uint64_t          *m_tickcount;

    TESTB(void) : m_trace(NULL),
                  m_tickcount(0) {
        m_core = new VA;
        Verilated::traceEverOn(true);
        m_core->i_clk = 0;
        eval();
    }
    // ...
}
```

Initialize these values in the constructor



# Verilator template



That's the constructor, here's the destructor

```
// ...  
virtual ~TESTB(void) {  
    closetrace();  
    delete m_core;  
    m_core = NULL;  
}  
// ...
```

- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
  - ▷ Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion



# Verilator template



Create a trace. Should look familiar.

```
// ...
virtual void opentrace(const char *vcdname) {
    // Open a VCD file
    m_trace = new VerilatedVcdC;
    m_core->trace(m_trace, 99);
    m_trace->open(vcdname);
}

virtual void closetrace(void) {
    // Close the already opened VCD file
    m_trace->close();
    delete m_trace;
    m_trace = NULL;
}
// ...
```





# Verilator template



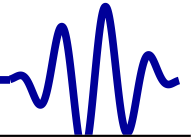
Finally, our operations. These haven't fundamentally changed.

```
// ...  
virtual void eval(void) {  
    m_core->eval();  
}  
  
virtual void tick(void) {  
    // ...  
    // This is the same as what we  
    // introduced in our last  
    // lesson ...  
}  
// ...  
};
```

See past lessons, and the current project file(s) for more detail here.



# Main simulation file

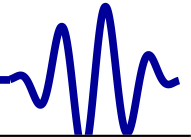


- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
  - ▷ Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

```
#include <Vhelloworld.h> // our top level
#include "uartsim.h" // A co-simulator
// ...
int main(int argc, char **argv) {
    Verilated::commandArgs(argc, argv);
    TESTB<Vhelloworld> *tb
        = new TESTB<Vhelloworld>;
    UARTSIM *uart // cosim object
        = new UARTSIM();
    // ...
    for(int clocks=0;
        clocks < 16*32*baudclocks;
        clocks++) {
        tb->tick();
        (*uart)(tb.o_uart_tx);
    }
}
```



# Main simulation file



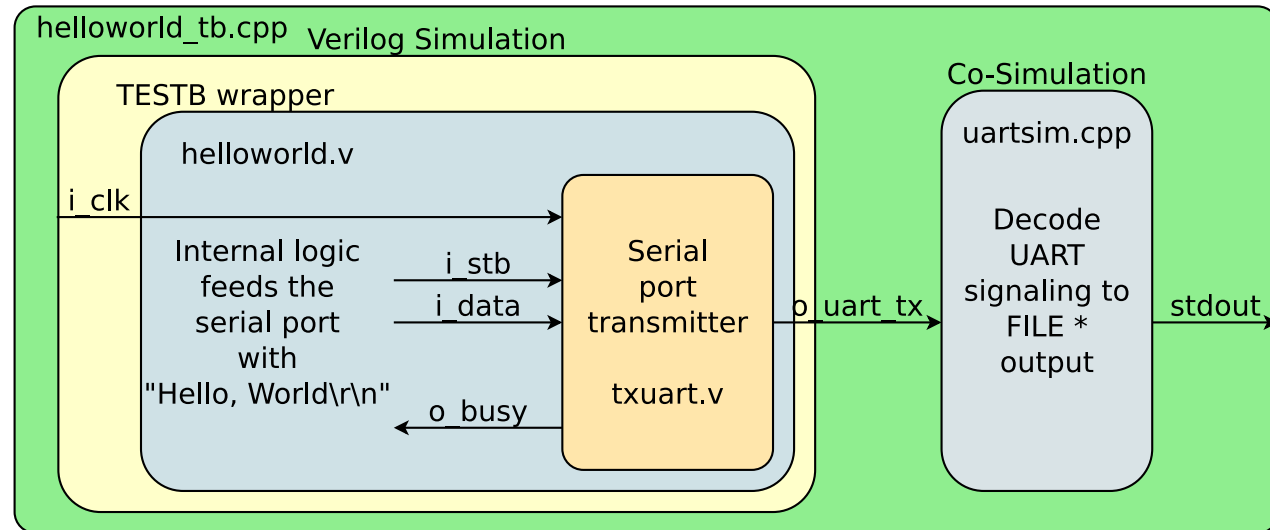
```
#include <Vhelloworld.h> // our top level
#include "uartsim.h" // A co-simulator
// ...
int main(int argc, char **argv) {
    Verilated::commandArgs(argc, argv);
    TESTB<Vhelloworld> *tb
        = new TESTB<Vhelloworld>;
    UARTSIM          *uart // cosim object
        = new UARTSIM();
    // ...
}
```

The secret key to success lies in the **UARTSIM** co-simulator

- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
  - Main simulation
    - ▷ file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion



# What is cosimulation?



A cosimulator is a separate simulation

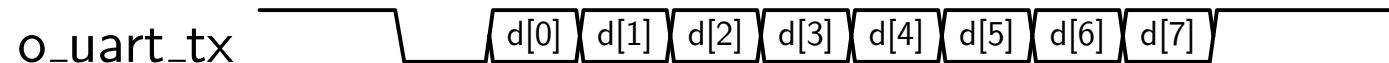
- Simulates the hardware components we are connected to
- In this case, the serial port
- Can use C++ **assert()** statements liberally



# Serial Decoding



Our co-simulation will need to decode this serial signal



Steps to decode a serial port:

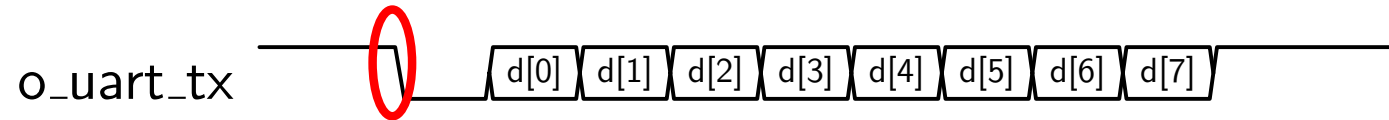
- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
  - ▷ Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion



# Serial Decoding



Our co-simulation will need to decode this serial signal



Steps to decode a serial port:

1. Detect the start bit
  - This determines the timing of everything to follow

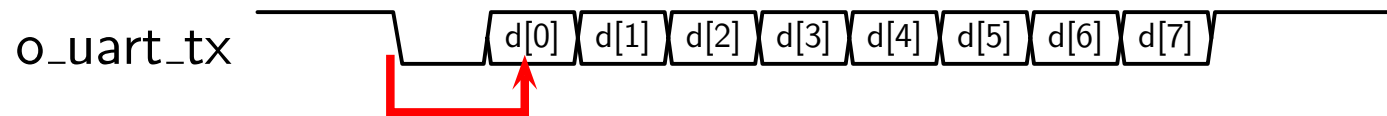
- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
  - ▷ Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion



# Serial Decoding



Our co-simulation will need to decode this serial signal



Steps to decode a serial port:

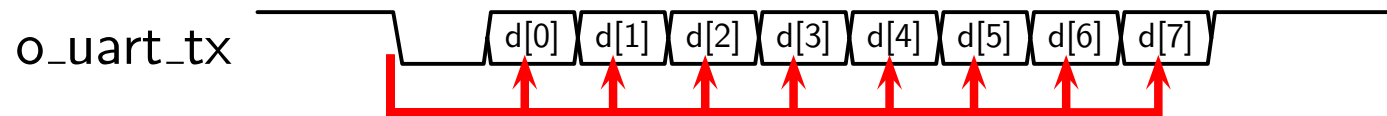
1. Detect the start bit
  - This determines the timing of everything to follow
2. Wait a baud and a half
  - Centers our sample mid baud-interval



# Serial Decoding



Our co-simulation will need to decode this serial signal



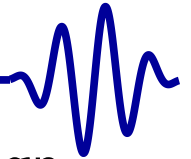
Steps to decode a serial port:

1. Detect the start bit
  - This determines the timing of everything to follow
2. Wait a baud and a half
  - Centers our sample mid baud-interval
3. Sample each remaining data bit mid-baud
  - Known baud rate determines the separation





# UART Co-simulator



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
  - ▷ Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

The first step is to make certain the cosimulator and design share the same baud rate

- First, adjust the design

```
module helloworld(i_clk ,
'ifdef VERILATOR
            o_setup ,
'endif
            o_uart_tx );
// ...
    parameter INITIAL_UART_SETUP
                = (CLOCK_RATE_HZ/BAUD_RATE);
'ifdef VERILATOR
    output wire [31:0] o_setup;
    assign o_setup = INITIAL_UART_SETUP;
'endif
```



# UART Co-simulator



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
  - ▷ Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

The first step is to make certain the cosimulator and design share the same baud rate

- First, adjust the design
- Then read the value from C++

```
int      main(int argc , char **argv) {  
    // ...  
    unsigned      baudclocks ;  
  
    baudclocks = tb->m_core->o_setup ;  
    uart->setup(baudclocks) ;  
    // ...  
}
```

Now the cosimulator and design share the same baud rate



# UART Co-simulator



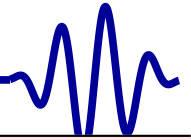
All the co-sim work is done on a clock tick

```
int    UARTSIM::operator()(const int i_tx) {
    m_last_tx = i_tx;

    if (m_rx_state == RXIDLE) {
        // Detect start bit
        if (!i_tx) {
            m_rx_state = RXDATA;
            // Wait a baud and a half
            m_rx_baudcounter = m_baud_counts
                               + m_baud_counts/2-1;
            m_rx_bits        = 0; // bit counter
            m_rx_data        = 0; // a shift reg
        }
        // continued ...
    }
}
```



# UART Co-simulator



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
  - ▷ Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

```
// ... continued
} else if (m_rx_baudcounter <= 0) {
    // Middle of a data bit interval
    if (m_rx_bits >= 8) {
        // Last data bit: post the result
        m_rx_state = RXIDLE;
        putchar(m_rx_data);
        fflush(stdout);
    } else {
        m_rx_bits++;
        m_rx_data = ((i_tx & 1) ? 0x80 : 0)
                    | (m_rx_data >> 1);
    } // Restart the baud counter
    m_rx_baudcounter = m_baud_counts - 1;
} else // Wait for next mid-bit interval
    m_rx_baudcounter--;
}
```



# Make Foo



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
  - ▷ Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

When command lines get complicated, I turn to make

- A makefile consists of a list of targets, dependencies, and instructions

```
target: dependency files
```

```
# Instructions for creating the target
```

```
touch target # Just one example
```

- Now, if any of the dependency files change, make will rebuild the target
- Make will also now rebuild all targets depending upon this one



# Make Foo



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- ▷ Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

You can set a Makefile variable

```
TOPMOD := helloworld
```

and then reference it later

```
$(VERIFIL) := $(TOPMOD).v
```

If we do this right,

- Our Makefile logic can be reused



# Make Foo



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
  - ▷ Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

## Example of re-use

```
TOPMOD := helloworld
VLOGFIL := $(TOPMOD).v      # Our Verilog file
VCDFILE := $(TOPMOD).vcd  # Our VCD trace file
SIMPROG := $(TOPMOD)_tb   # Simulation executable
SIMFILE := $(SIMPROG).cpp # Simulation top lvl
```

Now redefining \$(TOPMOD) will change this Makefile from one purpose/project to another



# Make Foo



- Lesson Overview
- Serial Protocol
- Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- ▷ Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

With -Wall, Verilator will fail on a warning

- It will leave its build products behind
- A second make will finish building the erroneous code
- The **.DELETE\_ON\_ERROR**: makefile target prevents this

**.DELETE\_ON\_ERROR:**





# Make Foo



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
  - ▷ Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

Verilator will build dependency files for you with `-MMD`

- We can include these into our Makefile with

```
DEPS := $(wildcard obj_dir/*.d)  
  
ifneq $(DEPS),  
include $(DEPS)  
endif
```

- Now, if `txuart.v` changes, make will call Verilator again
- This keeps us from needing to list all the Verilog files in the Makefile



# Make Clean



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
  - ▷ Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

We can create a special “clean” target

- To remove all build products

```
clean :  
      rm -rf obj_dir / $(TOPMOD) _tb
```

- `clean` isn't really a file, but a target that should always be built upon request

```
.PHONY: clean
```

This will tell make to ignore any file named “clean” that might be in your directory



# Make Clean



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
  - ▷ Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

We can create a special “clean” target

- To remove all build products

```
clean :  
        rm -rf obj_dir/ helloworld_tb
```

- This will fail if we delete our Verilator dependency files
- Simple fix:

```
ifneq $(MAKECMDGOALS), clean)  
ifneq $(DEPS), )  
include $(DEPS)  
endif  
endif
```



# Simulation



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- ▷ Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

Try running the simulation now

```
% ./helloworld_tb
Hello, World!

Simulation complete
%
```

Things to note:

- Simulation is slow
  - 8,680 clocks required to simulate each character
- The VCD file is large (14M)
  - This is actually quite small relatively
  - Simulations can take up 50GB or more
  - Keep an eye on disk space usage



# Formal Verification



- Lesson Overview
- Serial Protocol
- Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
  - Formal
    - ▷ Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

The entire design needs to be simplified

- Split into two separate proofs
  - TX UART itself
  - The Hello World wrapper
- When verifying the Hello World wrapper
  - Can't keep the assumptions of the TX UART!
  - If we define TXUART only for txuart.v ...
  - We can create a macro redefining assume
  - ...and turning it into an assert for helloworld.v

```
'ifdef TXUART
'define ASSUME assume
'else
'define ASSUME assert
'endif
```



# Formal Verification



- Lesson Overview
- Serial Protocol
- Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
  - Formal
    - ▷ Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

The entire design needs to be simplified

- Split into two separate proofs
- When verifying the Hello World wrapper
  - Need to define TXUART now
  - Requires adjusting our SymbiYosys script

```
[ script ]
```

```
read -DTXUART -formal txuart.v  
prep -top txuart
```

- The `-DTXUART` defines the TXUART macro
- The rest is the same as before



# Verifying txuart



- Lesson Overview
- Serial Protocol
- Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
  - ▷ Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

Some useful properties:

- Input requests should remain constant until they are serviced

```
always @(posedge i_clk)
if ((f_past_valid)
      &&($past(i_wr))&&($past(o_busy)))
begin
    'ASSUME(i_wr == $past(i_wr));
    'ASSUME(i_data == $past(i_data));
end
```



# Verifying txuart



- Lesson Overview
- Serial Protocol
- Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
  - ▷ Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

Some useful properties:

- Baud counter should always be less than CLOCKS\_PER\_BAUD

```
always @(*)  
    assert(counter < CLOCKS_PER_BAUD);
```

- If the baud counter is nonzero, it should be counting down

```
always @(posedge i_clk)  
if ((f_past_valid)&&($past(counter) != 0))  
    assert(counter == $past(counter - 1'b1));
```





# Verifying txuart



- Lesson Overview
- Serial Protocol
- Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
  - ▷ Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

Some useful properties:

- If the counter is non-zero, the busy output should be true

```
always @(*)  
if (counter > 0)  
    assert (o_busy);
```

These assertions are all good and nice, but ...

- They do nothing to assure me that this design even works



# Formal Contract



- Lesson Overview
- Serial Protocol
- Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- ▷ Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

Any set of formal properties should include a *contract*

- Describes the required black-box behavior
- Describes how the core will be seen by the world

This is in addition to any assertions about local register values



# Formal Contract



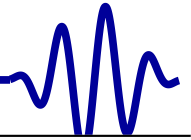
- Lesson Overview
- Serial Protocol
- Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
  - Verifying txuart
    - ▷ Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

Our contract:

```
always @(posedge i_clk)
if ((i_wr)&&(!o_busy))
    fv_data <= i_data;
always @(posedge i_clk)
case(state)
IDLE:      assert(o_uart_tx);
START:     assert(o_uart_tx == 1'b0);
BIT_ZERO:  assert(o_uart_tx == fv_data[0]);
BIT_ONE:   assert(o_uart_tx == fv_data[1]);
BIT_TWO:   assert(o_uart_tx == fv_data[2]);
BIT_THREE: assert(o_uart_tx == fv_data[3]);
BIT_FOUR:  assert(o_uart_tx == fv_data[4]);
BIT_FIVE:  assert(o_uart_tx == fv_data[5]);
BIT_SIX:   assert(o_uart_tx == fv_data[6]);
BIT_SEVEN: assert(o_uart_tx == fv_data[7]);
default:  assert(0); // Should never be here
```



# Formal Contract



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
  - Verifying txuart
    - ▷ Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

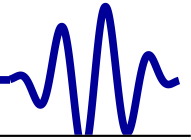
```
% sby -f txuart.sby
```

```
SBY 8:03:12 [txuart] engine_0.induction: ## 0:00:00 Temporal induction failed!
SBY 8:03:12 [txuart] engine_0.basecase: ## 0:00:00 Checking assumptions in step 3..
SBY 8:03:12 [txuart] engine_0.basecase: ## 0:00:00 Checking assertions in step 3..
SBY 8:03:12 [txuart] engine_0.induction: ## 0:00:00 Assert failed in txuart: txuart.v:227
SBY 8:03:12 [txuart] engine_0.induction: ## 0:00:00 Writing trace to VCD file: engine_0/trace_induct.vcd
SBY 8:03:12 [txuart] engine_0.basecase: ## 0:00:00 Checking assumptions in step 4..
SBY 8:03:12 [txuart] engine_0.basecase: ## 0:00:00 Checking assertions in step 4..
SBY 8:03:12 [txuart] engine_0.induction: ## 0:00:00 Writing trace to Verilog testbench: engine_0/trace_induct_tb.v
SBY 8:03:12 [txuart] engine_0.basecase: ## 0:00:00 Status: PASSED
SBY 8:03:12 [txuart] engine_0.basecase: finished (returncode=0)
SBY 8:03:12 [txuart] engine_0: Status returned by engine for basecase: PASS
SBY 8:03:12 [txuart] engine_0.induction: ## 0:00:00 Writing trace to constraints file: engine_0/trace_induct.smtc
SBY 8:03:12 [txuart] engine_0.induction: ## 0:00:00 Status: FAILED (!)
SBY 8:03:12 [txuart] engine_0.induction: finished (returncode=1)
SBY 8:03:12 [txuart] engine_0: Status returned by engine for induction: FAIL
SBY 8:03:12 [txuart] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)
SBY 8:03:12 [txuart] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00 (0)
SBY 8:03:12 [txuart] summary: engine_0 (smtbmc yices) returned PASS for basecase
SBY 8:03:12 [txuart] summary: engine_0 (smtbmc yices) returned FAIL for induction
SBY 8:03:12 [txuart] DONE (UNKNOWN, rc=4)

/ex-05-hello$
```



# Formal Contract



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
  - Verifying txuart
    - ▷ Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

```
% sby -f txuart.sby
```

```
SBY 8:03:12 [txuart] engine_0.induction: ## 0:00:00 Temporal induction failed!
SBY 8:03:12 [txuart] engine_0.basecase: ## 0:00:00 Checking assumptions in step 3..
SBY 8:03:12 [txuart] engine_0.basecase: ## 0:00:00 Checking assertions in step 3..
SBY 8:03:12 [txuart] engine_0.induction: ## 0:00:00 Assert failed in txuart: txuart.v:227
SBY 8:03:12 [txuart] engine_0.induction: ## 0:00:00 Writing trace to VCD file: engine_0/trace_induct.vcd
SBY 8:03:12 [txuart] engine_0.basecase: ## 0:00:00 Checking assumptions in step 4..
SBY 8:03:12 [txuart] engine_0.basecase: ## 0:00:00 Checking assertions in step 4..
SBY 8:03:12 [txuart] engine_0.induction: ## 0:00:00 Writing trace to Verilog testbench: engine_0/trace_induct_tb.v
SBY 8:03:12 [txuart] engine_0.basecase: ## 0:00:00 Status: PASSED
SBY 8:03:12 [txuart] engine_0.basecase: finished (returncode=0)
SBY 8:03:12 [txuart] engine_0: Status returned by engine for basecase: PASS
SBY 8:03:12 [txuart] engine_0.induction: ## 0:00:00 Writing trace to constraints file: engine_0/trace_induct.smtc
SBY 8:03:12 [txuart] engine_0.induction: ## 0:00:00 Status: FAILED (!)
SBY 8:03:12 [txuart] engine_0.induction: finished (returncode=1)
SBY 8:03:12 [txuart] engine_0: Status returned by engine for induction: FAIL
SBY 8:03:12 [txuart] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)
SBY 8:03:12 [txuart] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00 (0)
SBY 8:03:12 [txuart] summary: engine_0 (smtbmc yices) returned PASS for basecase
SBY 8:03:12 [txuart] summary: engine_0 (smtbmc yices) returned FAIL for induction
SBY 8:03:12 [txuart] DONE (UNKNOWN, rc=4)
```

What happened?



# Formal Contract



- Lesson Overview
- Serial Protocol
- Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
  - Verifying txuart
    - ▷ Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

Our contract: *Failed Induction! Why?*

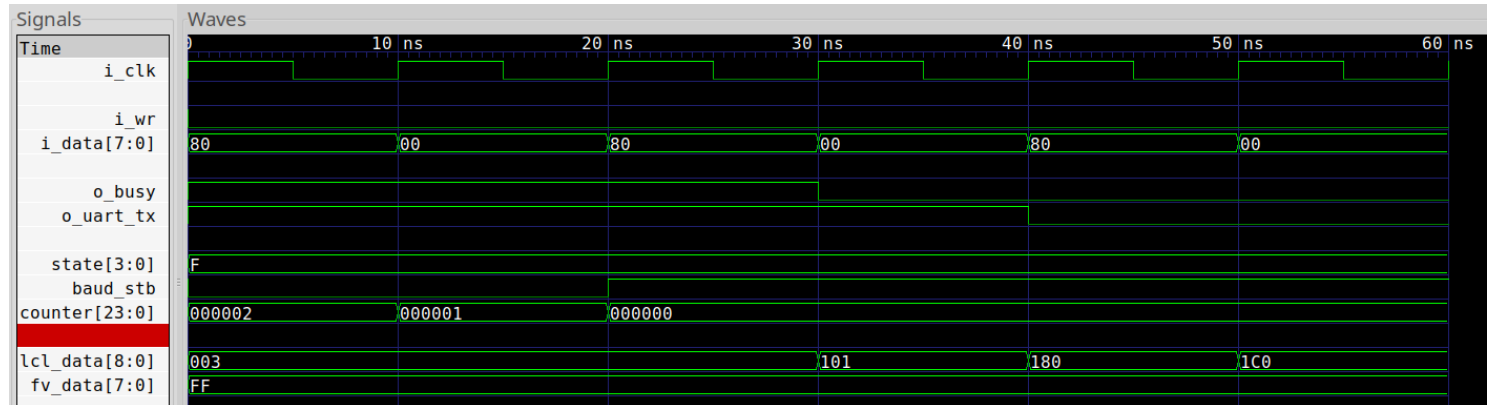
```
always @(posedge i_clk)
  if ((i_wr)&&(!o_busy))
    fv_data <= i_data;
always @(posedge i_clk)
  case(state)
    IDLE:      assert(o_uart_tx);
    START:    assert(o_uart_tx == 1'b0);
    BIT_ZERO: assert(o_uart_tx == fv_data[0]);
    BIT_ONE:  assert(o_uart_tx == fv_data[1]);
    BIT_TWO:  assert(o_uart_tx == fv_data[2]);
    BIT_THREE: assert(o_uart_tx == fv_data[3]);
    BIT_FOUR: assert(o_uart_tx == fv_data[4]);
    BIT_FIVE: assert(o_uart_tx == fv_data[5]);
    BIT_SIX:  assert(o_uart_tx == fv_data[6]);
    BIT_SEVEN: assert(o_uart_tx == fv_data[7]);
    default:  assert(0); // Should never be here
```



# Formal Contract



Need to look at the trace

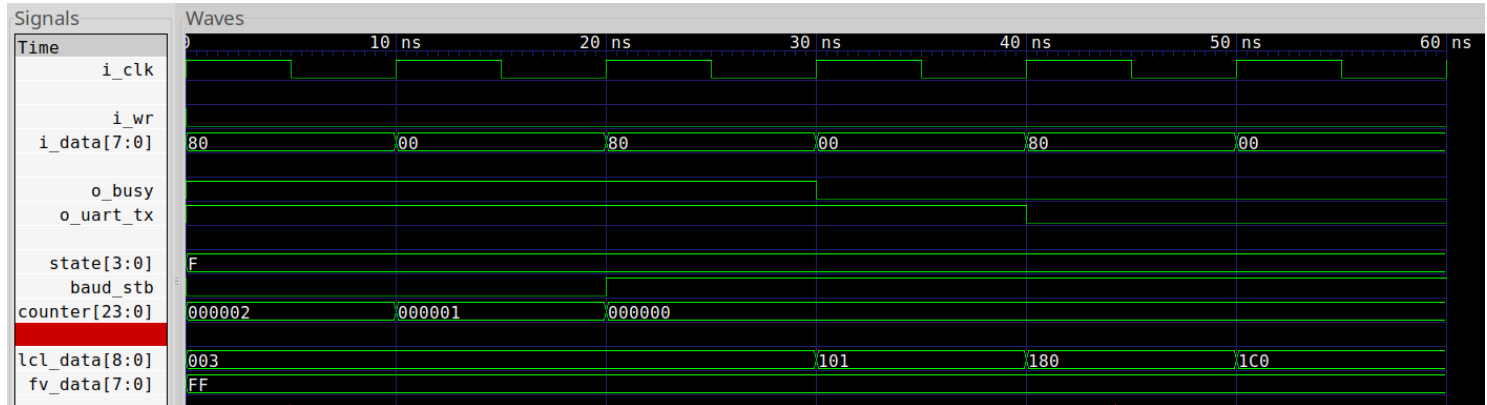




# Formal Contract



Need to look at the trace



The problem  
starts back here

Our assertion  
failed here

Why was lcldata set to 003 on start?

- It should have been 9'h1ff!





# Formal Contract



- Lesson Overview
- Serial Protocol
- Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
  - Verifying txuart
    - ▷ Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

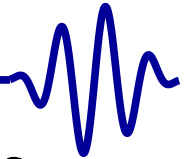
The issue revolves around how  $k$ -induction works

- During the induction step, ...
- Initial values are constrained by assumptions and assertions only
- If your design isn't fully constrained, it may start in an unreachable state

Induction typically requires more assertions to pass



# Passing Induction



Fixing an induction problem always follows the same steps

- Look for something amiss in the first  $N - 1$  steps  
... the steps *before* the assertion failure
- **assert**() something appropriate to keep it from happening
- If the **assert**() is inappropriate
  - Your design will fail at the (second to) last step of a trace
  - Don't be surprised if BMC fails during this process
- Repeat until you find a bug, or until your design passes

Let's apply this to our design



# Passing Induction



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
  - ▷ Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

lcldata should be 9'h1ff whenever state == IDLE

```
always @(*)  
case(state)  
IDLE: assert(lcl_data == 9'h1ff);  
default :  
endcase
```

The rest of the missing assertions are left as an exercise.



# Exercise #1



- Lesson Overview
- Serial Protocol
- Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- ▷ Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

Your turn!

- Modify txuart.v as necessary until it passes formal verification



# Hello World



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
  - ▷ Hello World
- Exercise #2
- Hardware!
- Conclusion

What properties would be appropriate for helloworld.v?

```
always @(*)
if ((tx_stb)&&(!tx_busy))
begin
    case(tx_index)
        4'h0: assert(tx_data <= "H" );
        4'h1: assert(tx_data <= "e" );
        4'h2: assert(tx_data <= "l" );
        4'h3: assert(tx_data <= "l" );
        //
        // ...
    endcase
end
```

We could check that the right letters are sent



# Hello World



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- ▷ Hello World
- Exercise #2
- Hardware!
- Conclusion

What properties would be appropriate for helloworld.v?

```
always @(*)  
if (tx_index != 4'h0)  
    assert(tx_stb);
```

We could assert the request is high throughout the message  
Can you think of any other properties to check?



# Exercise #2



Lesson Overview  
Serial Protocol  
Implementation  
Submodules  
Top Level  
Philosophy  
Simulation  
Main simulation file  
Cosimulation  
Make Foo  
Formal Verification  
Verifying txuart  
Formal Contract  
Exercise #1  
Hello World  
▷ Exercise #2  
Hardware!  
Conclusion

Your turn!

- Simulate this Hello World
- Formally verify the top level



# Hardware!



Lesson Overview

Serial Protocol

Implementation

Submodules

Top Level

Philosophy

Simulation

Main simulation file

Cosimulation

Make Foo

Formal Verification

Verifying txuart

Formal Contract

Exercise #1

Hello World

Exercise #2

▷ Hardware!

Conclusion

This is the exercise you've been waiting for:

- Run Hello World on your hardware!

You'll need some parameters for your terminal program

- Adjust `CLOCK_RATE_HZ` to match your board
- Your terminal should be set to
  - 8 data bits
  - No parity
  - One stop bit
  - No hardware flow control
  - A baud rate of `BAUD_RATE` (115.2kb)

I encourage you to look up these terms

- You should see a repeating “Hello, World!” pattern

Don't forget to make sure you connect to the right serial port





# Conclusion



- Lesson Overview
- Serial Protocol Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- ▷ Conclusion

What did we learn this lesson?

- How to build a UART transmitter!
- How cosimulation works
- How to make the simulation driver simpler
- A little about Makefile's
- What a “contract” is when using formal verification
- The realities of working with induction

We learned how to do our debugging *before touching the hardware!*