9. Serial Port Receiver
Lesson Overview

Let’s build a **Serial Port Receiver**

- Like the transmitter, it will have
  - A constant baud rate,
  - 8 data bits, no parity, and one stop bit

- Building the **serial port** is not tremendously more complex than the transmitter
  - Verifying the **serial port** will be our biggest challenge

- Also build a UARTSIM transmitter in C++ for Verilator

Objectives

- Know how to coordinate verification across files
- Experience the power of **induction**
- Gain more experience building Verilator **Co-simulators**
- Learn how to work with a Verilator **serial port** simulator
We discussed building a serial port receiver before.

The basic processing steps are:
We discussed building a serial port receiver before.

```
```

The basic processing steps are:
1. Detect the start bit
   - This determines the timing of everything to follow
We discussed building a **serial port** receiver before.

The basic processing steps are:

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
We discussed building a serial port receiver before.

The basic processing steps are:

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
We discussed building a **serial port** receiver before.

The basic processing steps are:

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
4. Report our result when done
We discussed building a **serial port** receiver before.

This also means that we’ll be done halfway through the stop bit.

- The transmitter will still be busy, even though
- The receiver is already looking for the next start bit
Since our last discussion (about simulation)

```
// Initialize everything to one (idle)
initial { ck_uart, q_uart } = -1;
always @(posedge i_clk)
  { ck_uart, q_uart }
  <= { q_uart, i_uart_rx };
```

- This should be negligible to the rest of the algorithm
- It will impact our formal verification properties
The receiver logic is just another state machine

- Each state will require multiple clocks
- States 2-9 are exactly one baud in length
- States 1 is half again as long
  - To account for the start bit, and
  - To make sure we timeout mid-baud interval
- The \( o_{\text{stb}} \) signal will be one clock wide
- When \( o_{\text{stb}} \) is high, \( o_{\text{data}} \) contains the received data
  - It is a don’t care value otherwise
Let’s work through timing all these transitions

- A counter, `baud_counter`, will count down the time until the next state transition
- While we are in idle, it will remain at zero

```vhdl
initial baud_counter = 0;
always @(posedge i_clk)
  if (state == IDLE)
    begin
      baud_counter <= 0;
      // ...
```
Let’s work through timing all these transitions

- A counter, baud_counter, will count down the time until the next state transition
- While we are in idle, it will remain at zero
- On a start bit, it will start counting a baud and a half

```
initial baud_counter = 0;
always @(posedge i_clk)
if (state =!= IDLE)
begin
  baud_counter <= 0;
  if (!ck_uart)
    baud_counter <= CLOCKS_PER_BAUD - 1'b1
    + CLOCKS_PER_BAUD / 2;
  // ...
```
Let’s work through timing all these transitions

- A counter, `baud_counter`, will count down the time until the next state transition
- While we are in idle, it will remain at zero
- On a start bit, it will start counting a baud and a half
- When it is not zero, it will count down to zero

```verilog
always @(posedge i_clk)
  if (state == IDLE)
    begin
      // ...
    end
  else if (baud_counter == 0)
    begin
      // ...
    end
  else
    baud_counter <= baud_counter - 1'b1;
```
Let’s work through timing all these transitions

- A counter, `baud_counter`, will count down the time until the next state transition
- While we are in idle, it will remain at zero
- On a start bit, it will start counting a baud and a half
- When it is not zero, it will count down to zero
- When it reaches zero, we count down the next baud

```verilog
initial baud_counter = 0;
always @(posedge i_clk)
if (state == IDLE)
begin
    // ...
end else if (baud_counter == 0)
begin
    baud_counter <= Clocks_PER_BAUD - 1'b1;
```

Let’s work through timing all these transitions

- A counter, `baud_counter`, will count down the time until the next state transition
- While we are in idle, it will remain at zero
- On a start bit, it will start counting a baud and a half
- When it is not zero, it will count down to zero
- When it reaches zero, we count down the next baud
- Unless we reach the end of the word

```vhdl
// ...
end else if (baud_counter == 0)
begin
    baud_counter <= CLOCKS_PER_BAUD - 1'b1;
    if (state >= STOP)
        baud_counter <= 0;
```

- ...where it will remain at zero
The receiver state follows the same conditions

- We start in IDLE, and remain in IDLE while ck_uart is high

```vhdl
initial state = IDLE;
always @(posedge i_clk)
if (state == IDLE) // 0
begin
    // Wait until ck_uart goes low
    state <= IDLE;
    if (!ck_uart)
        // ...
```

The receiver state follows the same conditions:

- We start in IDLE, and remain in IDLE while `ck_uart` is high.
Receiver State

The receiver state follows the same conditions

- We start in IDLE, and remain in IDLE while $ck_{\text{uart}}$ is high
- When $ck_{\text{uart}}$ goes low, we switch states

```verilog
initial state = IDLE;
always @(posedge i_clk)
if (state == IDLE) // 0
begin
    // Wait until $ck_{\text{uart}}$ goes low
    state <= IDLE;
    if (!ck_uart)
        state <= BIT_ZERO;
end else // ...
```
Receiver State

Once we’ve seen a start bit

- We cycle through and receive each bit following, and
- Return to idle when we get to the stop bit

```verilog
always @(posedge i_clk)
//
end else if (baud_counter == 0)
begin
    state <= state + 1;
    if (state >= STOP_BIT)
        state <= IDLE;
end
```

See any assertions you might need to make about the state?
On every state change

- Shift in one more bit of the answer

```verilog
always @(posedge i_clk)
if ((baud_counter == 0) && (state != STOP_BIT))
o_data <= { ck_uart, o_data[7:1] };
```
Return Strobe

On the last and final transition

- Notify our environment of a received bit
- Just as we return to IDLE

```verilog
initial o_wr = 1'b0;
always @(posedge i_clk)
o_wr <= (baud_counter == 0) && (state == STOP_BIT);
```
On the last and final transition

- Notify our environment of a received bit
- Just as we return to IDLE

```verilog
initial o_wr = 1'b0;
always @(posedge i_clk)
    o_wr <= (baud_counter == 0) && (state == STOP_BIT);
```

This should all be quite straightforward

- This isn’t really any harder than the transmitter
Formally verifying this receiver . . . that’s harder
Let’s reflect upon the two basic types of properties we’ve created

- **Contract properties**
  - Verify that a design does what it was intended to do
  - These can be **black-box properties**

- **Induction** properties
  - Verify that a design remains within the set of legal states
  - These will always be **white-box properties**

And our two rules

- **assume** any input properties
- **assert** any local state and output properties
The contract for a **serial port** is straightforward

- If you send it a known transmission
- It should set $o_{wr}$ when done, and
- $o_{data}$ should match any expected result
- We can use our transmitter to send a known transmission

That’s the contract. That’s the easy part
The difficult part is setting up the **induction** properties

- We need to make certain our design remains in a consistent state
  - That includes not only the state of the receiver, and
  - The state of the transmitter, but
  - *The two states must match!*

- That means the transmitter can’t be sending bit two while we are receiving bit six

- That also means that after the transmitter has sent four bits the receiver must have received those same four bits

Coordinating the state between the receiver and the transmitter is the challenging part
Adjusting the transmitter

We’ll add two output ports to our transmitter for this purpose

- **f_data**
  - This is the data the transmitter is sending
  - We’ll need to match our received data with this at every step of the way

- **f_counter**
  - This will count clocks since the beginning of transmission
  - We’ll use this to match the receiver’s state

We’ll call this adjusted transmitter \texttt{f\_txuart}

- Since these extra ports are only necessary for formally verifying the receiver
- They are inappropriate for an independent transmitter
Capturing the data being sent is easy

```verilog
always @(posedge i_clk)
  if (((i_wr) && (!o_busy)))
    f_data <= i_data;
```

It’s even easier, since . . .

- The transmitter already contained this value internally
- The transmitter verified its internal state against this value
- The transmitter finishes after the receiver
  - So this value should be valid when we examine it
- We’ll just make this value an output
The transmit counter is conceptually simple

```verilog
always @(posedge i_clk)
  if (!o_busy)
    f_counter <= 0;
  else
    f_counter <= f_counter + 1;
```

Only we must now assert that

- This counter matches our transmitter’s internal state
Matching \texttt{f\_counter} to the transmitter’s count-down counter

\begin{verbatim}
always @(*) // In the transmitter
  case(state)
    START: assert(f_counter == CLOCKS_PER_BAUD-1-counter);
    BIT_ZERO: assert(f_counter == 2*CLOCKS_PER_BAUD-1-counter);
    BIT_ONE: assert(f_counter == 3*CLOCKS_PER_BAUD-1-counter);
    // ...
endcase
\end{verbatim}

Let’s look at this a little deeper
Let’s discuss these assertions

\[ \text{BIT_ZERO: } \text{assert} \left( f\_\text{counter} == \frac{2}{1} \times \text{CLOCKS\_PER\_BAUD} - 1 - \text{counter} \right); \]

You may find this easier to understand if you draw it out

- \( f\_\text{counter} \) starts at the beginning of time and counts up
- Our baud interval counter, \( \text{counter} \), counts down each interval
Let’s discuss these assertions

\[
\text{BIT\_ZERO: assert}(f\_counter = 2 \times \text{CLOCKS\_PER\_BAUD} - 1 - \text{counter});
\]

Notice the multiply for a moment

- Multiplying are normally bad
  - Formal tools struggle to verify multiplies
  - This multiplies two constants, so the result is constant
  - So this works

That handles the internal values

- What about the inputs to \text{f\_txuart}?
Our receiver doesn’t have inputs for the formal transmitter

- We need to generate inputs for $f_{\text{txuart}}$
- $(\ast \text{anyseq} \ast)$ can be used for that purpose
- $(\ast \text{anyseq} \ast)$ is like $(\ast \text{anyconst} \ast)$
  - The solver gets to pick the values
- Only $(\ast \text{anyseq} \ast)$ values can change from clock to clock
  - $(\ast \text{anyconst} \ast)$ values are required to be constant
- Both types of values may be constrained by assumptions
- We’ll pass two inputs to the transmitter

```vhdl
// The write request input
(* anyseq *) reg f_tx_iwr;

// The write data input
(* anyseq *) reg [7:0] f_tx_idata;
```
Here's our transmitter instantiation

```verilog
(* anyseq *) reg f_tx_iwr;

(* anyseq *) reg [7:0] f_tx_idata;

wire f_tx_uart;

// Assume our input matches the txuart's output
always @(*)
assume (i_uart_tx == f_tx_uart);

(* ignored *)
wire f_tx_busy;

wire [7:0] f_txdata;

wire [24:1:0] f_tx_counter;

f_txuart #(CLOCKS_PER_BAUD)
tx(i_clk, f_tx_iwr, f_tx_idata,
    f_tx_uart, f_tx_busy,
    f_txdata, f_tx_counter);

We'll be working with f_txdata and f_tx_counter
```
We can now assert our receiver contract

- o_wr goes high once at the end of every word

```verilog
always @(*)
assert (o_wr == (f_tx_counter == 9 * CLOCKS_PER_BAUD + CLOCKS_PER_BAUD / 2));
```

- o_data has a copy of the transmitted information

```verilog
always @(*)
if (o_wr)
    assert (o_data == f_txdata);
```

Problem: that’s not enough to pass induction!
Now we need to synchronize our partial results

```verilog
always @(*)
case(state)
  4'h2: assert(o_data[7] == f_txdata[0]);
  4'h3: assert(o_data[7:6] == f_txdata[1:0]);
  4'h4: assert(o_data[7:5] == f_txdata[2:0]);
  4'h5: assert(o_data[7:4] == f_txdata[3:0]);
  // ... etc
  4'h9: assert(o_data[7:0] == f_txdata[7:0]);
endcase
```

Even this isn’t enough, we need to match counters as well
Induction

Matching the two counters is harder

- Following the end condition, the transmitter may have a half clock period left
- After the transmitter starts, it can go two clocks through the synchronizer before we leave IDLE

```verilog
always @(*)
  case(state)
  4'h0: begin
    if (f_tx_uart)
      assert((f_tx_counter == 0)
        || f_tx_counter > 9 * CLOCKS_PER_BAUD
          + CLOCKS_PER_BAUD / 2);
    else
      assert(f_tx_counter < 3);
  end
```
Matching the two counters is harder

- While waiting for the first bit, the two counters should be off by a baud and a half

```verilog
always @(*)
case(state)
  // ...
4'h1: begin // Start state
  assert (CLOCKS_PER_BAUD+CLOCKS_PER_BAUD/2 - baud_counter == f_tx_counter - 2);
endcase
```

- Remember the two stage FF synchronizer, and
- The receiver is off cut from the transmitter by half a baud
Matching the two counters is harder

- While waiting for the first bit, the two counters should be off by a baud and a half
- The rest of the bits/states follow the same pattern

```verilog
always @(*)
case(state)
  // ...
  4'h2: begin // Start state
    assert(2*CLOCKS_PER_BAUD+CLOCKS_PER_BAUD/2 - baud_counter == f_tx_counter -2);
endcase
```

- Don’t forget that baud_counter counts down,
- While f_tx_counter counts up
Beginners often struggle to understand how the transmitter and receiver can get out of synch during induction

- This gives them no end of trouble
- This doesn’t happen in a bounded check, but
- A bounded check can’t handle 10 periods of 868 clocks
- Induction is the key to verifying our contract
- Several extra assertions were required to get there

Synchronizing the two modules is key to success

- We’ll discuss cover() next
- Then you should be able to finish the proof
We should cover our solution as well

```verbatim
always @(posedge i_clk)
  cover(o_wr);
```

- But how shall we cover something that takes $10 \times 868$ clocks?
- Solution: Lower the clocks per baud, but just for cover
- This can be done in the SymbiYosys file
Remember tasks?

- You can use tasks to selectively adjust parameter values

```plaintext
[tasks]
cvr
prf

[options]
prf: mode prove
cvr: mode cover
cvr: depth 192
prf: depth 4

[script]
read -formal f_txuart.v
read -formal rxuart.v
cvr: hierarchy -top rxuart \
    -chparam CLOCKS_PER_BAUD 8
prep -top rxuart
```
Remember tasks?

- You can use tasks to selectively adjust parameter values

```plaintext
[ tasks ]
cvr
prf

[ options ]
prf: mode prove
cvr: mode cover
cvr: depth 192
prf: depth 4

[ script ]
read -formal f_txuart.v
read -formal rxuart.v
cvr: hierarchy -top rxuart
   -chparam CLOCKS_PER_BAUD 8
prep -top rxuart
```

This changes our CLOCKS_PER_BAUD parameter to 8, but only for our cover task.
Remember tasks?

- You can use tasks to selectively adjust parameter values

```bash
[ tasks ]
cvr
prf

[ options ]
prf: mode prove
 cvr: mode cover
 cvr: depth 192
 prf: depth 4

[ script ]
read –formal f_txuart.v
read –formal rxuart.v
cvr: hierarchy –top rxuart \ 
    –chparam CLOCKS_PER_BAUD 8
prep –top rxuart
```

This adjusts our depth to 192, but again only for the cover task.
What might we want to cover?

- A successful result

```verilog
always @(posedge i_clk)
  cover(o_wr);
```

- A second successful result? Two 8’hf9s received in a row?

```verilog
initial f_first_hit = 1'b0;
always @(posedge i_clk)
  if ((o_wr)&&(o_data == 8’hf9))
    f_first_hit <= 1'b1;

always @(posedge i_clk)
  cover((f_first_hit)&&(o_wr)
    &&(o_data == 8’hf9));
```
Cover is important, don’t skip it!

- Using `cover()` on this project, I discovered a bug in our transmitter.
- The transmitter should be able to transmit two characters in `20*CLOCKS_PER_BAUD`.
- Our original transmitter took one clock too long.
  - Two characters took `20*CLOCKS_PER_BAUD + 1` at first.
- I found the bug by examining the cover trace.

You now know enough to finish the rest of the formal proof on your own.
Formally verify that your receiver works!

- As always, some bugs have been hidden in the example code

Then, make it better

- Create a register called `zero_baud_counter`

```verilog
reg zero_baud_counter;
```

- Make it change on `@posedge i_clk` clock only
- Verify that it is true only if `baud_counter == 0`

```verilog
always @(*)
assert (zero_baud_counter == (baud_counter == 0));
```

You may start with the (mostly correct) receiver in exercise 9
Question for thought:

- Imagine you wanted to build a receiver that could handle multiple baud rates
  - For example, all 24-bit divisions of your clock rate
- How would you verify such a receiver?
Simulation outline

- We’ll read from one file
- “Transmit” the data to our **serial port**
  - The UARTSIM accepts data to transmit on STDIN
- Read the results from the port
  - We’ll dump these out STDOUT, and
- Verify the result matches the original file
Let’s dig into this UART **Co-simulator**

- Anytime we are idle,
- Check for a character to transmit on STDIN

```c
if (m_tx_state == TXIDLE) {
    ch = getchar();
    // ...
}
```

- Problem: this will hang our simulation if no character is available
- We need to check if there’s a character available first
- But without stopping if not
The `poll()` system call provides what we need

```c
if (m_tx_state == TXIDLE) {
    struct pollfd pb;
    pb.fd = STDIN_FILENO;
    pb.events = POLLIN;
    if (poll(&pb, 1, 0) < 0)
        perror("Polling error:");

    if (pb.revents & POLLIN) {
        char buf[1];

        nr = read(STDIN_FILENO, buf, 1);
        if (1 == nr) {
            // ...
        }
```

- This solves the hanging problem
- Now we just need to transmit the character to our receiver
The transmit logic follows what we’ve rehearsed already:

- On new data, set two shift registers
  - One containing the data plus a stop bit
  - One containing a bit mask of 10 busy intervals (One interval is implied, so 0x1ff)
  - Then clear the start bit and start a baud counter

```c
if (m_tx_state == TXIDLE) {
    // on start
    m_tx_data = 0x100 | (buf[0] & 0xff);
    m_tx_busy = 0x1ff; // Busy reg
    m_tx_state = TXDATA; // New state
    o_rx = 0; // Clear UART signal
    m_tx_baudcounter = m_baud_counts - 1;
}
```
The transmit logic follows what we’ve rehearsed already

- Whenever our timer runs out
  - Shift everything over, and
  - Restart the counter

```c
} else if (m_tx_baudcounter <= 0) {
    m_tx_data >>= 1;
    m_tx_busy >>= 1;
    m_tx_baudcounter = m_baud_counts - 1;
    o_rx = m_tx_data & 1;
```

- But ... how do we leave this loop?
The transmit logic follows what we’ve rehearsed already

- Except …
  - When we are no longer busy, and
  - When restarting the last counter

```c
} else if (m_tx_baudcounter <= 0) {
    if (!m_tx_busy)
        m_tx_state == IDLE;
    else {
        // ...
        if (m_tx_busy == 1)
            m_tx_baud_counter--;
    } else {
        // ...
```

- Wait, why is there one less clock on the last step?
The transmit logic follows what we’ve rehearsed already

- One less clock on the last step is required because
  - It takes a clock to recognize the idle, and then to
  - Return \texttt{m\_tx\_state} to IDLE

```c
} else if (m\_tx\_baudcounter <= 0) {
    if (!m\_tx\_busy)
        m\_tx\_state == IDLE;
    else {
        // ...
        if (m\_tx\_busy == 1)
            m\_tx\_baud\_counter--;
    }
else {
    // ...

- The last piece is simple
The transmit logic follows what we’ve rehearsed already

- Finally, if we are not IDLE, then the counter is not zero
  - Decrement the baud counter
  - Return a bit to the simulation

```c
if (m_tx_state == TXIDLE) {
    // ...
} else if (m_tx_baudcounter <= 0) {
    // ...
} else {
    // ...
    m_tx_baudcounter--;  
    o_rx = m_tx_data & 1;
}
return o_tx;
```

- That’s the logic in the (simulated) transmitter
We need a test bench that can

- Create a known input stream into our receiver
  - We can use another `psalm.txt` file for this
- Produce an output
- Compare the output with the input

The fact that UARTSIM uses stdin will make this problematic
Let’s create two pipes, then split our test bench into two:

- This will allow us to write to the UARTSIM, and
- Read and verify the result
Let’s create two pipes, then split our test bench into two:

1. The first process, the parent, will
   - Read the test data from a file
   - Write it into the pipe, sending it to the child’s stdin
   - Read the results back from the pipe
   - Compare the results with the original file

2. The second process will run our simulation
   - Accept data from stdin
   - Write it to the serial port via the UARTSIM
   - Receive the results from the receiver
   - Write the results out to the parent via stdout

It’s time to learn about the fork() system call
The `fork()` system call splits a process into two

- One process will be called the parent
  - This process maintains the identity of the original process
- The other process is the child

```c
if ((child_pid == fork()) != 0) {
    // Code to run in the parent
    // (the original process)
} else {
    // Code to run in the child
}
```

Before we `fork()`, we’ll need to create two `pipe()`s to communicate between processes
The `pipe()` system call creates a pipe

- We’ll need two: one for each direction

```c
int childs_stdin[2], childs_stdout[2];

if ((pipe(childs_stdin) != 0) || (pipe(childs_stdout) != 0)) {
    // Deal with any errors
    exit(EXIT_FAILURE);
}

// Now we can call fork()
```

- We’ll replace the child’s `stdin/stdout` with these pipes
- The parent will thus control the child’s `stdin/stdout`
The `pipe()` system call creates a unidirectional pipe

- Data written to `childs_stdin [1]` can be read from `childs_stdin [0]`, same for `childs_stdout`

- The parent closes the read end of the `childs_stdin`
  
  ```
  close(childs_stdin[0]);
  ```

  - Only the child will read from this pipe

- The parent also closes the write end of `childs_stdout`
  
  ```
  close(childs_stdout[1]);
  ```

  - Only the child will write to this pipe

- The child will do the opposite
The child also needs to map these pipes to stdin/stdout

- First, map `childs_stdin [0]` to stdin
- Done by first closing the file descriptor to be replaced
- Then duplicating the pipe’s file descriptor

```c
close(STDIN_FILENO);
dup(childs_stdin[0]);
```

- The duplicated file descriptor naturally replaces the one that was just closed
- We’ll repeat this for stdout

```c
close(STDOUT_FILENO);
dup(childs_stdout[1]);
```

We can now build and run our test!
The setup

This is what we’ve just created

- Two processes, where the child’s stdin/stdout are controlled by the parent
- These will be inter-process pipes
- The parent’s stdin/stdout will remain unchanged
In the parent, we send the message to the slave

```
write(childs_stdin[1], string, flen);
```

And read it back out

```
rd = read(childs_stdout[0], rdbuf, flen);
for (i=0; i<rd; i++) {
    putchar(rdbuf[i]);
    if (rdbuf[i] != string[i]) {
        fail=i;
        break;
    }
}
```

Don’t forget to check for errors!
The slave’s code looks like what we’ve done with Verilator before

- First the setup

```cpp
// Create a test bench
tb = new TESTB<Vrxuart>;
// Start a VCD trace
tb->opentrace("rxuart.vcd");
// Create a UART simulator
uart = new UARTSIM();
// Set the baud rate
// ...
// and make sure the port starts idle
tb->m_core->i_uart_rx = 1;
```

- Now we can build our test
The slave matches what we’ve done with Verilator before

- First the setup
- Then run the testbench

```c
while ((testcount++ < LARGE_NUMBER) && (num_received < flen)) {
    tb->tick();
    tb->m_core->i_uart_rx = (*uart)(1);
    // Any time we receive a character
    if (tb->m_core->o_wr) {
        num_received++;
        // Send it to stdout, and
        // thus to the parent via
        // the pipe
        putchar(tb->m_core->o_data);
    }
} exit(EXIT_SUCCESS);
```
Exercise!

Does your component simulation work?
- If not, debug as necessary

Once you get to real hardware
- You will no longer have access to every internal signal
  - You might only ever get an LED, sometimes not even that
- Debugging only gets harder in the next step

Many student’s have asked, why doesn’t my serial port work?
- The secret they were missing?
  - Avoid debugging on the hardware! Formal first, then simulation, then hardware once the bugs are gone
- If you know your design works, that will eliminate many possible causes of error in hardware
Let’s build a design and get it to work with your hardware

Debugging this design in hardware can be a challenge!

- A lot of things can go wrong—even if our code works
  - Subtle clock differences can be a challenge
  - Terminal setup can be an issue
- We’ll can now use the button, the LED, and the UART output to debug
- You should also know how to fully simulate this design
Common problems include:

- The wrong baud rate
  - You may receive either nothing or perhaps garbage

- Setting hardware flow control (turn it off for now)
  - Nothing comes through at all

- Missing carriage returns
  - You’ll see all the data, but it quickly vanishes off the edge of the screen

The message was carefully chosen to use the full 80 character width

- This will make it easier to spot missing characters
The rarer ugly problem

- One student saw only every other character of the message
- This was traced to a faster transmitter than the receiver
- ...and the following fragile logic

```vhdl
always @(*)
begin
  tx_wr = rx_stb;
  tx_data = rx_data;
end
```

- If the transmitter was still busy when `rx_stb` was true
  - It would miss the incoming data
  - Remember: `o_wr (rx_stb above)` is only high for a single cycle

- One solution: Adjust your terminal to produce two stop bits
A better solution to the rare but ugly problem

- A register between RX and TX will help smooth over subtle clock rate differences

```verilog
initial tx_wr = 1'b0;
always @(posedge i_clk)
  if (rx_stb)
    begin
      tx_wr <= 1'b1;
      tx_data <= rx_data;
    end else if (!tx_busy)
      tx_wr <= 1'b0;
```

- Can you see any lingering problems with this solution?
You can also set the LED on some internal condition:

- `if (rx_stb)` for example, or
- `if (rx_stb && (rx_data == 'P'))` as another

```verilog
reg [25:0] led_counter;
initial { o_led = 0, led_counter } = 0;
always @(posedge i_clk)
if (condition)
begin
    led_counter <= 0;
    o_led <= 1'b1;
end else if (&led_counter)
    o_led <= 1'b0;
else
    led_counter <= led_counter + 1'b1;
```

- This can help determine if your problem is a transmitter or receiver issue
You can also use our transmit word design, \texttt{txdata}:

- Using our button counter design, you can replace the transmitters output with any (useful) internal 32-bit value
- You did test the transmitter design and get it running, right?
- You should be able to guess and confirm potential problems
- This includes finding the cause of any missing characters
Other means of debugging include:

- **Sending internal logic wires to external ports**
  - And examining them with logic analyzer, oscilloscope, or even another FPGA

- **Connecting your device to another serial port / terminal**

- **Swapping USB cables**
  - Much to my surprise, USB cables can and do break
  - If things aren’t working, don’t forget to try another cable
    - That this solution works sometimes has surprised more than one skeptic designer
What did we learn this lesson?

- How to build and verify a **serial port** receiver
  - How to connect a formal-only transmitter to check if the receiver truly does work
  - A **serial port** requiring 868 clocks per baud will take 8,680 clocks per character. With **induction**, we can verify the **serial port** in less than 5 clocks

- How to build a simulated **serial port** transmitter
  - How to control items sent to the **serial port co-simulator** via **stdin** and **stdout**

- How important the fundamentals are to hardware debugging
  - Counters, LEDs, Buttons, hex data output, etc.