6. Transmitting Data Words

Daniel E. Gisselquist, Ph.D.
Debugging is one of the hardest parts of digital logic design

- You can’t see what’s happening *inside* the FPGA
- LED’s are one solution
  - FPGA’s operate 50MHz+
  - Your eye operates at < 60Hz
- The serial port can be a second solution

Let’s learn to send data through our serial port!

Objectives

- Transform Hello World into a debugging output
- Learn about formal abstraction
- Experiment with using *ncurses* with Verilator
- Extract internal design variables from within Verilator
Data Transmitter

Let’s transmit a word of data

Each word will . . .

- Start with 0x
- Contain the number sent, but in hexadecimal

  *this is much easier than doing decimal!*

  Four bits can be encoded at a time

- End with a carriage return / line-feed pair
You should know how to build this design already

Remember how we’ve built state machines before

- In this case, you have two triggers
  - One trigger, `i_stb`, starts the process
  - A busy line from the serial port, `tx_busy` (not shown), controls the movement from one character to the next

- This design will be the focus of this lesson
You should know how to build this design already

- **i_clk**: requests sending a word of data
- **i_stb**: requests a char be transmitted
- **o_busy**: unable to accept another word
- **i_data**: data
- **tx_stb**: controls the movement from one character to the next
- **tx_data**: 0 × h h h h h h h h h r n

Remember how we’ve built state machines before

- In this case, you have two triggers
  - One trigger, **i_stb**, starts the process
  - A busy line from the serial port, **tx_busy** (not shown), controls the movement from one character to the next

This design will be the focus of this lesson
Our overall design will look like this:

- Some event will trigger a counter
- A second module will detect that the counter has changed
- Finally we’ll output the result
- We’ll use `txuart.v` from the last exercise

Let’s take a quick look at `counter.v` and `chgdetector.v`
Creating a Counter

You should already know how to make an event counter

```verilog
module counter(i_clk, i_event, o_counter);
    input wire i_clk, i_event;
    output reg [31:0] o_counter;

    initial o_counter = 0;
    always @(posedge i_clk)
        if (i_event)
            o_counter <= o_counter + 1'b1;
endmodule
```

Feel free to add a reset if you would like
Detecting a change in the counter is also pretty easy

```verilog
module chgdetecttor(i_clk, i_data,
                     o_stb, o_data, i_busy);

    // ...

    initial { o_stb, o_data } = 0;

    always @(posedge i_clk)
        if (!i_busy)
            begin
                stb <= 0;
                if (o_data != i_data)
                    begin
                        stb <= 1'b1;
                        o_data <= i_data;
                    end
            end

endmodule
```
Detecting a change in the counter is also pretty easy

```verilog
module chgdetector(i_clk, i_data, o_stb, o_data, i_busy);
    // ...
    initial { o_stb, o_data } = 0;
    always @(posedge i_clk)
    if (!i_busy)
    begin
        stb <= 0;
        if (o_data != i_data)
            stb <= 1'b1;
            o_data <= i_data;
    end
endmodule
```

Nothing is allowed to change if `i_busy` is true. That's the case where a request has been made, but it has yet to be accepted.
Detecting a change in the counter is also pretty easy

```verilog
class chgdetector
module chgdetector(i_clk, i_data, o_stb, o_data, i_busy);

// ...
initial { o_stb, o_data } = 0;
always @(posedge i_clk)
if (!i_busy)
begin
    stb <= 0;
    if (o_data != i_data)
begin
        stb <= 1'b1;
        o_data <= i_data;
    end
end
endmodule
```

Otherwise, anytime the data changes, we set up a request to transmit the new data.
What formal properties might we use here?

- Any output value should remain unchanged until accepted

```verilog
// Remember this property?
always @(posedge i_clk)
if ((f_past_valid)
    &&($past(o_stb))&&($past(i_busy)))
    assert ((o_stb)&&($stable(o_data)));
```

Remember how this works? This says that ...
What formal properties might we use here?

- Any output value should remain unchanged until accepted

```vhdl
// Remember this property?
always @(posedge i_clk)
  if ((f_past_valid)
      &($past(o_stb))&($past(i_busy)))
    assert ((o_stb) && ($stable(o_data)));
```

Remember how this works? This says that . . .

- If both o_stb and i_busy are true on the same clock cycle (i.e., the interface is stalled)
- Then request should remain outstanding on the next cycle
- ... and the data should be the same on that next cycle
- $stable(o_data) is shorthand for o_data == $past(o_data)
What formal properties might we use here?

- Any output value should remain unchanged until accepted

```verilog
always @(posedge i_clk)
    if ((f_past_valid) & ($past(o_stb)) & ($past(i_busy)))
        assert ((o_stb) & ($stable(o_data)));
```

- When o_stb rises, o_data should reflect the input

```verilog
always @(posedge i_clk)
    if ((f_past_valid) & ($rose(o_stb)))
        assert (o_data == $past(i_data));
```

$\textit{\$rose(o_stb)}$ is shorthand for $(o_{\text{stb}}[0] \&\& \neg$\textit{\$past(o_{\text{stb}}[0])\text{)}}$
What formal properties might we use here?

- Any output value should remain unchanged until accepted

```verilog
// Remember this property?
always @(posedge i_clk)
if ((f_past_valid)
  &($past(o_stb))&($past(i_busy)))
  assert((o_stb)&($stable(o_data)));
```

- When o_stb rises, o_data should reflect the input

```verilog
always @(posedge i_clk)
if ((f_past_valid)&&($rose(o_stb)))
  assert(o_data == $past(i_data));
```

$rose(o_stb)$ is shorthand for (o_stb[0] && !$past(o_stb[0]))

- Can you think of any other properties we might need?
Our focus: txdata

This lesson will focus on txdata.v

- We’ve already built txuart.v
- You should have no problems designing counter.v or chgdetector.v
  
  You are encouraged to do so on your own
  - If not, you can find counter.v and chgdetector.v in the course handouts

You should also have a good idea how to start on txdata.v.

- It’s not all that different from txuart.v or helloworld.v
- The example in the course handouts is broken
Here’s the port list(s) we’ll design to

```verilog
module txdata(i_clk, i_stb, i_data, o_busy, o_uart_tx);

// ...
    txuart #(UART_SETUP[23:0]) txuarti(i_clk, tx_stb, tx_data, o_uart_tx, tx_busy);

// ...
endmodule
```
Here’s the port list(s) we’ll design to

```vhdl
module txdata(i_clk, i_stb, i_data, o_busy, o_uart_tx);
// ...
```

```vhdl
txuart #(UART_SETUP[23:0]) txuarti(i_clk, tx_stb, tx_data, o_uart_tx, tx_busy);
// ...
endmodule
```

- If `i_stb` is true, we have a new value to send
- `i_data` will then contain that 32-bit value
- `o_busy` means we cannot accept data
- `o_uart_tx` is the 1-bit serial port output
Here’s the port list(s) we’ll design to

```verilog
module txdata(i_clk, i_stb, i_data, o_busy, o_uart_tx);
// ...

endmodule
```

- `tx_stb` requests data be transmitted
- `tx_data` is the 8-bit character to transmit
- `tx_busy` means the serial port transmitter is busy and cannot accept data
We can create a state diagram for this state machine too.
We can create a state diagram for this state machine too.

We’ll start sending our message upon request (i_stb is true), and advance to the next character any time the transmitter is not busy (tx_busy is false).
We can create a state diagram for this state machine too.

In this chart, data is the 32-bit word we are sending, and \( \text{hex} \{ \} \) just references the fact that we need to convert the various nibbles to hexadecimal before outputting them.
We can create a state diagram for this state machine too.

Remember, input data such as `i_data` are only valid as long as the incoming request is valid (`i_stb` is high). We'll need to make a copy of that data once the request is made, `(i_stb) && (!o_busy)`, and then work off of that copy.
We can even annotate this with state ID numbers

- State 0
- State 1
- State 2
- State 3
- State 4
- State 5
- State 6
- State 7
- State 8
- State 9
- State 10
- State 11
- State 12

- tx_stb is low, state = 0 for idle
- tx_stb is high for all these states
The state machine should remind you of `helloworld.v`

```verilog
always @(posedge i_clk)
if (!o_busy)
begin
  if (i_stb)
  begin
    state <= 1;
    tx_stb <= 1;
  end // else state already == 0
end else if ((tx_stb) && (!tx_busy))
begin
  state <= state + 1;
  if (state >= 4'd1)
  begin
    tx_stb <= 1'b0;
    state <= 0;
  // ...

```
Outgoing Data

The outgoing data is just a shift register

```markdown
initial sreg = 0;
always @(posedge i_clk)
if (!o_busy) // && (i_stb)
sreg <= i_data;
else if ((!tx_busy)&&(state > 4'h1))
  // Hold constant until read
  sreg <= { i_data[27:0], 4'h0 };
```

Question:

Why aren’t we conditioning our load on i_stb as well?
Converting to hex is very straightforward.

```verilog
always @(posedge i_clk)
  case(sreg[31:28])
  4'h0: hex <= "0";
  4'h1: hex <= "1";
  4'h2: hex <= "2";
  4'h3: hex <= "3";
  // ...
  4'h9: hex <= "9";
  4'ha: hex <= "a";
  4'hb: hex <= "b";
  4'gc: hex <= "c";
  4'hd: hex <= "d";
  4'he: hex <= "e";
  4'hf: hex <= "f";
  default: begin end
endcase
```
Converting to hex is very straightforward.

```verilog
always @(posedge i_clk)
  case (sreg[31:28])
    4'h0: hex <= "0"; // Values in quotation
    4'h1: hex <= "1"; // marks specify literal
    4'h2: hex <= "2"; // 8-bit values with an
    4'h3: hex <= "3"; // ASCII encoding
    // ...
    4'h9: hex <= "9";
    4'hA: hex <= "a";
    4'hB: hex <= "b";
    4'hC: hex <= "c";
    4'hD: hex <= "d";
    4'hE: hex <= "e";
    4'hF: hex <= "f";
    default: begin end
  endcase
```
Converting to hex is very straightforward

```verilog
always @ (posedge i_clk)
    case (sreg[31:28])
        4'h0: hex <= "0"; // Values in quotation
        4'h1: hex <= "1"; // marks specify literal
        4'h2: hex <= "2"; // 8-bit values with an
        4'h3: hex <= "3"; // ASCII encoding
        // ...
        4'h9: hex <= "9"; // Strings work similarly
        4'ha: hex <= "a"; // with the only difference
        4'hb: hex <= "b"; // being that string
        4'hc: hex <= "c"; // literals may be much
        4'hd: hex <= "d"; // longer than 8-bits
        4'he: hex <= "e";
        4'hf: hex <= "f"; // Example: A <= "1234"
    default: begin end
endcase
```
Outgoing Data

Put together, here’s our code to transmit a byte

```verilog
always @(posedge i_clk)
  case(state)
    if (!tx_busy)
      case(state)
        4'h1: tx_data <= "0"; // These are the
        4'h2: tx_data <= "x"; // values we’ll
        4'h3: tx_data <= hex; // want to output
        4'h4: tx_data <= hex; // at each state
          // ...
        4'h9: tx_data <= hex;
        4'ha: tx_data <= hex;
        4'hb: tx_data <= "\r"; // Carriage return
        4'hc: tx_data <= "\n"; // Line-feed
      default: tx_data <= "Q"; // A bad value
      endcase
  endcase
```

Let’s do simulation *after* formal verification

- It’s easier to get a trace from formal
- Formal methods are often done faster
- etc.
Our design is getting large

- We’ve already verified txuart.v
- It would be nice not to have to do it again

Let’s simplify things instead!

- Let’s replace txuart.v with something that . . .
  - Might or might not act like txuart.v
Our design is getting large

- We’ve already verified `txuart.v`
- It would be nice not to have to do it again

Let’s simplify things instead!

- Let’s replace `txuart.v` with something that . . .
  - Might or might not act like `txuart.v`
  - . . . at the solver’s discretion
Our design is getting large

- We’ve already verified `txuart.v`
- It would be nice not to have to do it again

Let’s simplify things instead!

- Let’s replace `txuart.v` with something that . . .
  - Might or might not act like `txuart.v`
  - . . . at the solver’s discretion
  - Acting like `txuart.v` must remain a possibility

This is called *abstraction*
Here’s how we’ll do it:

```verilog
ifdef FORMAL
   txuart #(UART_SETUP[23:0]) txuarti(i_clk, tx_stb, tx_data, o_uart_tx, tx_busy);
elsif
   (* anyseq *) wire serial_busy, serial_out;
aassign o_uart_tx = serial_out;
aassign tx_busy = serial_busy;
```

- (*) allows the solver to pick the values of serial_busy and serial_out
- (*) values can change from one clock to the next
- They might match what txuart would’ve done
Here’s how we’ll do it:

```vhdl
‘ifdef FORMAL
  txuart #(UART_SETUP[23:0]) txuarti(i_clk, tx_stb, tx_data, o_uart_tx, tx_busy);
‘else
  (* anyseq *) wire serial_busy, serial_out;
  assign o_uart_tx = serial_out;
  assign tx_busy = serial_busy;
```

- (* anyseq *) allows the solver to pick the values of `serial_busy` and `serial_out`
- (* anyseq *) values can change from one clock to the next
- They might match what `txuart` would’ve done, or they might not
Here’s how we’ll do it:

```verilog
ifndef FORMAL

  txuart #(UART_SETUP[23:0]) txuarti(i_clk,
  tx_stb, tx_data, o_uart_tx, tx_busy);

else

  (* anyseq *) wire serial_busy, serial_out;
  assign o_uart_tx = serial_out;
  assign tx_busy = serial_busy;

```

- (*) anyseq *) allows the solver to pick the values of serial_busy and serial_out
- (*) anyseq *) values can change from one clock to the next
- They might match what txuart would’ve done, or they might not
- If our design passes in spite of what this abstract txuart does
Formal Verification

Here’s how we’ll do it:

```
ifndef FORMAL
    txuart #(UART_SETUP[23:0]) txuarti(i_clk, tx_stb, tx_data, o_uart_tx, tx_busy);
else
    (* anyseq *) wire serial_busy, serial_out;
    assign o_uart_tx = serial_out;
    assign tx_busy = serial_busy;
```

- (* anyseq *) allows the solver to pick the values of serial_busy and serial_out
- (* anyseq *) values can change from one clock to the next
- They might match what txuart would’ve done, or they might not
- If our design passes *in spite of what this abstract txuart does*, then it will pass if txuart acts like it should
We’ll insist that our abstract UART is busy following any request

```verilog
reg [1:0] f_minbusy;

initial f_minbusy = 0;
always @(posedge i_clk)
if ((tx_stb)&&(tx_busy))
    f_minbusy <= 2'b01;
else if (f_minbusy != 2'b00)
    f_minbusy <= f_minbusy + 1'b1;
```

We can use `f_minbusy` to force any transmit request to take at least four cycles before dropping the busy line

- `f_minbusy` is just a 2-bit counter
- After passing 3, it waits at zero for the next byte
We’ll insist that our abstract UART is busy following any request.

```verilog
reg [1:0] f_minbusy;

initial f_minbusy = 0;
always @(posedge i_clk)
if ((tx_stb) && (!tx_busy))
    f_minbusy <= 2'b01;
else if (f_minbusy != 2'b00)
    f_minbusy <= f_minbusy + 1'b1;

always @(*)
if (f_minbusy != 0)
    assume(tx_busy);
```

Since (* anyseq * ) values act like inputs to our design, constraining them by an assumption is appropriate.
We’ll also insist it doesn’t become busy on its own

```vhdl
initial assume(!tx_busy); // Starts idle
always @(posedge i_clk)
  if ($past(i_reset)) // Becomes idle after reset
    assume(!tx_busy);
  else if (($past(tx_stb)) && (!$past(tx_busy)))
    // Must become busy after a new request
    assume(tx_busy);
  else if (!$past(tx_busy))
    // Otherwise, it cannot become busy
    // without a request
    assume(!tx_busy);
```

Now we can build a proof without re-verifying txuart.v!
Let's see if this design works:

```verilog
// Don't forget to set the mode to cover
// in your SBY file!
always @(posedge i_clk)
if (f_past_valid)
    cover($fell(o_busy));
```

This would yield a trace with a reset

- It works, but it's not very informative
What if we except the reset?

```verilog
// Don’t forget to set the mode to cover
// in your SBY file!
always @(posedge i_clk)
if ((f_past_valid) & & (!$past(i_reset)))
    cover($fell(o_busy));
```

We can now get a useful trace

- The trace starts with a request
- Works through the whole sequence
- Stops when the state machine is ready to start again
What if we look for 0x12345678\r\n?

```verilog
reg f_seen_data;
initial f_seen_data = 0;
always @(posedge i_clk)
if (i_reset)
    f_seen_data <= 1'b0;
else if ((i_stb) && (!o_busy) && (i_data == 32'h12345678))
    f_seen_data <= 1'b1;

always @(posedge i_clk)
if ((f_past_valid) && (!$past(i_reset)) && (f_seen_data))
    cover($fell(o_busy));
```

Check out the trace.
What if we look for 0x12345678\r\n? 

```verilog
reg f_seen_data;
initial f_seen_data = 0;
always @(posedge i_clk)
if (i_reset)
    f_seen_data <= 1'b0;
else if ((i_stb)&&(o_busy)
    &&(i_data == 32'h12345678))
    f_seen_data <= 1'b1;
always @(posedge i_clk)
if ((f_past_valid)&&(!$past(i_reset))
    &&(f_seen_data))
    cover($fell(o_busy));
```

Check out the trace. Does your design work?
What if we look for `0x12345678`?

```verilog
reg f_seen_data;
initial f_seen_data = 0;
always @(posedge i_clk)
  if (i_reset)
    f_seen_data <= 1'b0;
  else if ((i_stb) && (!o_busy) && (i_data == 32'h12345678))
    f_seen_data <= 1'b1;
```

Caution: It’s a snare to use something like `f_seen_data` outside of a cover context

- We aren’t doing directed simulation
- The great power of formal is that it applies to all inputs
- We’re just picking an interesting input for a trace
Now, what assertions would be appropriate?

- We can assert state is legal
- That $tx_{-}stb != (state == 0)$
- Can we assert that the first data output is a "0"?
- That the second output is a "1"?

Your turn: what would make the most sense here?
Yes, we can assert a sequence takes place!

```vhdl
reg [12:0] f_p1reg; // Property s_reg

initial f_p1reg = 0;
always @(posedge i_clk)
if (i_reset)
    f_p1reg <= 0;
else if ((i_stb) && (!o_busy))
begin
    f_p1reg <= 1;
    assert(f_p1reg == 0);
end else if (!tx_busy)
    f_p1reg <= { f_p1reg[11:0], 1'b0 };

f_p1reg[x] will now be true for stage x of any output sequence
```
But what is f_p1reg? It’s a shift register

- f_p1reg[x] is true anytime we are in stage x of our sequence
- We can use this when constructing formal properties
Using $f_{p1\text{reg}}[x]$ we can make assertions about the different states in our sequence.

```verilog
always @(posedge i_clk)
if (!tx_busy) || (f_minbusy == 0)
begin
    // If the serial port is ready for
    // the next character, or while we are
    // waiting for the next character, ...
    if (f_p1reg[0])
        assert((tx_data == "0") && (state == 1));
    if (f_p1reg[1])
        assert((tx_data == "x") && (state == 2));
    // etc.
end
```
Sequence

Why use a shift register for f_p1reg[x]?

- A counter would also work for this sequence
- A shift register is more general and powerful
  - A shift register can represent states in a sequence that might overlap itself
  - Perhaps such a sequence may be entered on every clock cycle
  - An example would be a peripheral that always responds to any request in \( N \) cycles, yet never stalls

\( f_p1reg[x] \) allows us to represent general sequence states
Concurrent Assertions

Full System Verilog support would make this easier

```verilog
sequence SEND(A,B);
    (tx_stb)&&(state == A)&&(tx_data == B)
throughout
    (tx_busy) [0:$] ###1 (!tx_busy)
endsequence
```

This defines a sequence where

- `(tx_stb) && ...` must be true
- while `tx_busy` is true, and then
- until (and including) the clock where `tx_busy` is false
Sequence

Full System Verilog support would make this easier

```verilog
sequence SEND(A, B);
// ....
```

We could then string such sequences together in a **property** that could be asserted

```verilog
assert property (@(posedge i_clk))
  disable iff (i_reset)
  (i_stb) && (!o_busy)
  |=> SEND(1, "0") // First state
  ##1 SEND(2, "x") // Second, etc
  // ...
```

- A |=> B means if A, then B is asserted true on the next clock
- ##1 here means one clock later
Sequence

Full System Verilog support would make this easier

```verilog
sequence SEND(A,B);
// ....
```

We could then string such sequences together in a property that could be asserted

```verilog
assert property (@(posedge i_clk))
  disable iff (i_reset)
  (i_stb)&&(!o_busy)
  |=> SEND(1, "0") // First state
  ##1 SEND(2, "x") // Second, etc
  // ...
```

SymbiYosys support for sequences requires a license
**Sequence**

Full System Verilog support would make this easier

```verilog
sequence SEND(A,B);
// ....
```

We could then string such sequences together in a `property` that could be asserted

```verilog
assert property (@(posedge i_clk))
  disable iff (i_reset)
  (i_stb) && (!o_busy)
|=> SEND(1, "0") // First state
##1 SEND(2, "x") // Second, etc
// ...
```

*SymbiYosys support for sequences requires a license*

- `f_p1reg` let’s us do roughly the same thing
Exercise #1

Your turn!
Take a moment now to . . .

- Create your txdata.v, or
- Download my broken one, and then
- Formally verify it
  - Add such assertions as you deem fit
  - Make sure you get a trace showing it working

Does your design work?
Let’s move on to simulation

- Let’s use the simulator to count key presses
- `ncurses` + Verilator offers a quick debugging environment
  - Every time a key is pressed, output a new count value
  - We’ll use `getch()` to get key presses immediately
    - You may need to download and install `ncurses-dev`
  - We’ll adjust `uartsim()` to print to the screen
- You can also examine internal register values with Verilator
  - While the design is running

Let’s look at how we’d do these things
ncurses is an old-fashioned text library

- It allows us easy access to key press information
- We can write to various locations of the screen
- etc.
- The original ZipCPU debugger was written with ncurses

We’ll only scratch the surface here
Starting **ncurses** requires some boilerplate

```c
#include <ncurses>

// ... int main(int argc, char **argv) {
    // ...
    initscr();
    raw();
    noecho();
    keypad(stdscr, true);
    halfdelay(1);
}
```

- This initializes the curses environment
- Turns off line handling and echo
- Decodes special keys (like escape) for us
- **halfdelay**(1) – Doesn’t wait for keypresses
Our inner loop will start by checking for keypresses

```c
    do {
        done = false;
        tb->m_core->i_event = 0;
        // Ket a keypress
        chv = getch();
        if (chv == KEY_ESCAPE)
            // Exit on escape
            done = true;
        else if (chv != ERR)
            // Key was pressed
            tb->m_core->i_event = 1;
            tb->tick();
            (*uart)(tb->m_core->o_uart_tx);
    } while (!done);
```
We can speed this up too:

```c
do {
    // ...
    for (int k=0; k<1000; k++) {
       .tb->tick();
        (*uart)(tb->m_core->o_uart_tx);
        tb->m_core->i_event = 0;
    }
} while (!done);
```

- `getch()` waits $\frac{1}{10^{th}}$ of a second for a keypress
  - This is because we called `halfdelay(1)`;
- This will run 1000 simulation ticks per `getch()` call
We can also count given keypresses

```c
do {
    // ...
    for (int k=0; k<1000; k++) {
        tb->tick();
        (*uart)(tb->m_core->o_uart_tx);
        keypresses += tb->m_core->i_event;
        tb->m_core->i_event = 0;
    }
} while (!done);
```

We’ll print this number out before we are done
We’ll also need to replace the putchar() in uartsim.cpp

- **ncurses** requires we use **addch()**

```cpp
// if character received
if (m_rx_data != '\r')
    addch(m_rx_data);
```

- No flush is necessary, **getch()** handles that
- '\r' would clear our line, so we keep from printing it
endwin() ends the ncurses environment

```c
endwin();
printf("\n\nSimulation complete\n");
printf("%4d keypresses sent\n", keypresses);
```

This is nice, but

- wouldn't you also like a summary of keypresses the design counted?
Verilator data

Verilator maintains your entire design in a C++ object

- With a little work, we can find our variables
- A quick grep through Vthedesign.h reveals ...
- \texttt{v__DOT__counterv} contains our counter’s value
  - I use an older version of Verilator
  - Modern versions place this in \texttt{thedesign__DOT__counterv}
  - Supporting both requires a little work

- You can often find other values like this
  - Grep on your variables name
  - Be aware, Verilator will pick which of many names to give a value
  - Output wires may go by the name of their parent’s value
This little adjustment will allow us to simplify the reference to our counter

```c
#ifdef OLD_VERILATOR
#define VVAR(X) v__DOT_ ## A
#else
#define VVAR(X) thedesign__DOT_ ##A
#endif

#define counterv VVAR(_counterv)
```

- If **OLD_VERILATOR** is defined (my old version)
  - counterv evaluates to `v__DOT__counterv`
- Otherwise counterv is replaced by
  - `thedesign__DOT__counterv`
We can now output our current counter

```c
endwin();

printf("\n\nSimulation complete\n");
printf("%4d key presses sent\n", keypresses);
printf("%4d key presses registered\n", tb->m_core->counter);
```
Two changes are required to our build script

- If you want to define **NEW_VERILATOR** or **OLD_VERILATOR** ...
  - You’ll need to do some processing on Verilator’s version
  - The `vversion.sh` file does this, returning either `-DOLD_VERILATOR` or `-DNEW_VERILATOR`
  - We can use this output in our `g++` command line
  - Alternatively, you can just adjust the file for your version

- We need to reference `-lncurses` in our Makefile when building our executable
Exercise #2

Your turn!
Build and experiment with the simulation

- Using your txdata.v
- **main()** is found in thedesign_tb.cpp in the handouts
- Experiment with …
  - Adjusting the number of **tb->tick()** calls between calls to **getch()**
  - Does this speed up or slow down your design?
  - Are all of your keypresses recognized?
  - What happens when you press the key while the design is busy?
Exercise #3

Only now is it time to test this in hardware

- You’ll need to test for button changes

```verilog
always @(posedge i_clk)
    last_btn <= i_btn;

assign w_event = (i_btn) & & (!last_btn);
```

- Does it work?
  - Does it count once per keypress?
  - Does the counter look reasonable?

My implementation experienced several anomalies.

- We’ll discuss those in the next lesson
Conclusion

What did we learn this lesson?

- How to formally verify a part of a design, and not just the leaf modules
- Creating interesting traces with cover
- Subtle timing differences can be annoying
- How to use Verilator with ncurses
- Extracting an internal design value from within a Verilator simulation

We learned how to get information back out from within the hardware

- We’ll discuss the hazards of asynchronous inputs more in the next lesson