



Gisselquist
Technology, LLC

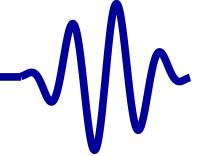
0. Preface

Daniel E. Gisselquist, Ph.D.





Lesson Overview



▷ Lesson Overview

V-

Verilog subset

Style

Co-sim

Formal

Debugging

Which board?

Minimum Board

Clear your desk

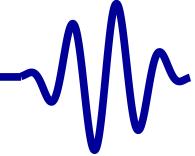
Test your board

Conclusion

Objectives

- Understand the Course philosophy
- Check the Pre-requisites
- Getting Started

Clear your desk, it's time to get started!



Lesson Overview

▷ V-

Verilog subset

Style

Co-sim

Formal

Debugging

Which board?

Minimum Board

Clear your desk

Test your board

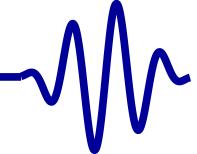
Conclusion

Verilog is a large language

- Only some verilog is necessary for design
 - Simulation verilog gets confused with synthesizable verilog
 - Programmers turn Verilog into a programming language.
It's not
 - Verilog testbench language is inadequate for bug finding when compared with formal methods
We'll be using SymbiYosys for formal verification
 - Verilog testbenches are a poor substitute for a good simulation language, such as C++
We'll be using verilator and C++ for simulation

A better solution is needed!

- Let's call it V--



Lesson Overview

V-

▷ Verilog subset

Style

Co-sim

Formal

Debugging

Which board?

Minimum Board

Clear your desk

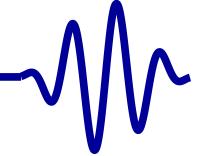
Test your board

Conclusion

Verilog is a large language

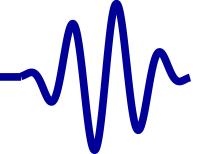
- Only some verilog is necessary for design
- We'll use synthesizable code only
 - No `A <= #10 B;` statements
 - No `@posedge` statements
 - No **\$display**, **\$monitor**, or **\$final** statements, etc.
 - No '`x`' values
 - Only toplevel ports can be **inouts**
 - We'll use restricted forms for multiply and memory
 - Avoid teaching loops as long as possible
 - We will use **initial** statements

initial statements are appropriate for FPGA's



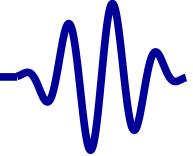
Verilog is a large language

- Only some verilog is necessary for design
- We'll use synthesizable code only
- Safe style guide
 - One clock (initially)
 - No logic generated clocks
 - We'll use **initial** statements
 - Reset values must match initial values



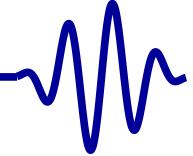
Verilog is a large language

- Only some verilog is necessary for design
- We'll use synthesizable code only
- Safe style guide
- **Co-simulation** is a *must*
 - External hardware peripheral simulations will be built in C++
 - Goal is to create a design that looks, acts, and works as though it were on the FPGA



Verilog is a large language

- Only some verilog is necessary for design
- We'll use synthesizable code only
- Safe style guide
- Co-simulation is a *must*
- Formal verification is great for bench testing
 - We'll scratch the surface here

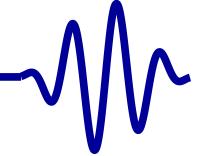


Verilog is a large language

- Only some verilog is necessary for design
 - We'll use synthesizable code only
 - Safe style guide
 - **Co-simulation** is a *must*
 - Formal verification is great for bench testing
 - Verilog instruction must include
 - Formal methods, and
 - Simulation
- ... from the beginning!*



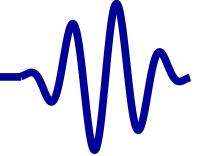
Which board?



Lesson Overview
V-
Verilog subset
Style
Co-sim
Formal
Debugging
▷ Which board?
Minimum Board
Clear your desk
Test your board
Conclusion

This course is intended to be board-agnostic

- We'll cover the basics and the mechanics
- We'll use Verilator extensively
- You don't need a board to take this course
- You may enjoy the course more with a board
- "Board bonus chapter appendices" may eventually accompany the course



Lesson Overview

V-

Verilog subset

Style

Co-sim

Formal

Debugging

Which board?

▷ Minimum Board

Clear your desk

Test your board

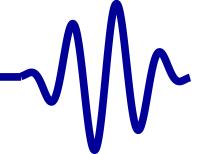
Conclusion

Course designs depend upon a minimum capability

- One button/switch, one LED
 - Many exercises use multiple LEDs
 - While not necessary, if you want to build these in hardware you'll need more than one LED on your board
- Serial port, both transmit/receive



Clear your desk



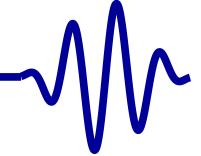
Lesson Overview
V–
Verilog subset
Style
Co-sim
Formal
Debugging
Which board?
Minimum Board
▷ Clear your desk
Test your board
Conclusion

If you have an FPGA board, then

- Find and download the schematic, . . .
- The data sheets for all of the components, . . .
- The board vendor's master constraint file, and
- The board vendor's demo code

Put these files in a project reference directory

Do this before any project with a new FPGA board!



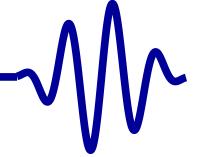
Your board vendor should provide you with

- A demonstration design, and
- The instructions necessary to build and load it

This design should verify that your hardware works

If you will be using hardware for this course, please verify that your hardware passes this test first

Conclusion



- Digital design can be hard, let's not make it harder
- Teach debugging tools with the language
- Are you ready to learn?

Lesson Overview

V–
Verilog subset

Style

Co-sim

Formal

Debugging

Which board?

Minimum Board

Clear your desk

Test your board

▷ Conclusion



Gisselquist
Technology, LLC

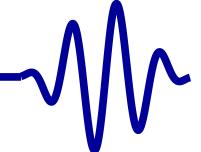
1. Wires

Daniel E. Gisselquist, Ph.D.





Lesson Overview



▷ Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

Conclusion

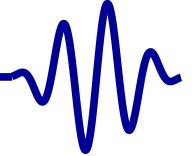
- What is a **wire**?
- What can I do with it?
- How do I build a design?

Objectives

- To get an initial, basic familiarization with combinatorial logic
- To learn how to run the tools to build a design
- To get an initial design running on an FPGA board



First design

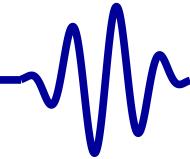


Lesson Overview
▷ First design
Schematic
Constraints
PCF
Build the design
First Success!
Simulation
Verilator Driver
Bus Signals
Bit Select
Internal Signals
Schematic
Circular Logic
Dual Assignment
Sim Result
Examples
Exercise
Conclusion

Let's build a simple Verilog design

```
module thruwire(i_sw, o_led);
    input wire i_sw;
    output wire o_led;

    assign o_led = i_sw;
endmodule
```



Lesson Overview

▷ First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

Conclusion

Let's build a simple Verilog design

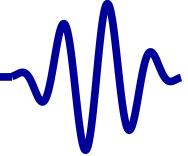
```
module thruwire(i_sw, o_led);
    input wire i_sw;
    output wire o_led;

    assign o_led = i_sw;
endmodule
```

- Verilog files contain modules
- This module is named `thruwire`
- While Verilog allows more than one module per file,
I recommend only one module per file.



First design



Lesson Overview

▷ First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

Conclusion

Let's build a simple Verilog design

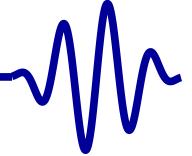
```
module thruwire(i_sw, o_led);
    input wire i_sw;
    output wire o_led;

    assign o_led = i_sw;
endmodule
```

- The **module** keyword marks the beginning
- **endmodule** marks the end of the module



First design



Lesson Overview
▷ First design
Schematic
Constraints
PCF
Build the design
First Success!
Simulation
Verilator Driver
Bus Signals
Bit Select
Internal Signals
Schematic
Circular Logic
Dual Assignment
Sim Result
Examples
Exercise
Conclusion

Let's build a simple Verilog design

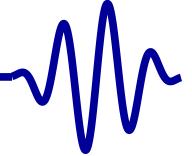
```
module thruwire(i_sw, o_led);
    input wire i_sw;
    output wire o_led;

    assign o_led = i_sw;
endmodule
```

- This module declares two ports, `i_sw` and `o_led`
- The first is declared to be an `input`
- The second is declared as an `output`
- Both are `wire`'s, but we'll get to that later



First design



Lesson Overview

▷ First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

Conclusion

Let's build a simple Verilog design

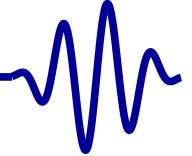
```
module thruwire(i_sw, o_led);
    input wire i_sw;
    output wire o_led;

    assign o_led = i_sw;

endmodule
```

- Our one piece of logic sets `o_led` to be the same as `i_sw`

First design



Lesson Overview
▷ First design
Schematic
Constraints
PCF
Build the design
First Success!
Simulation
Verilator Driver
Bus Signals
Bit Select
Internal Signals
Schematic
Circular Logic
Dual Assignment
Sim Result
Examples
Exercise
Conclusion

Let's build a simple Verilog design

```
module thruwire(i_sw, o_led);
    input wire i_sw;
    output wire o_led;

    assign o_led = i_sw;
endmodule
```

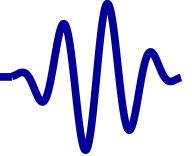
FPGA's are commonly used as:

- Traffic cops

A programmable/adjustable wire fabric

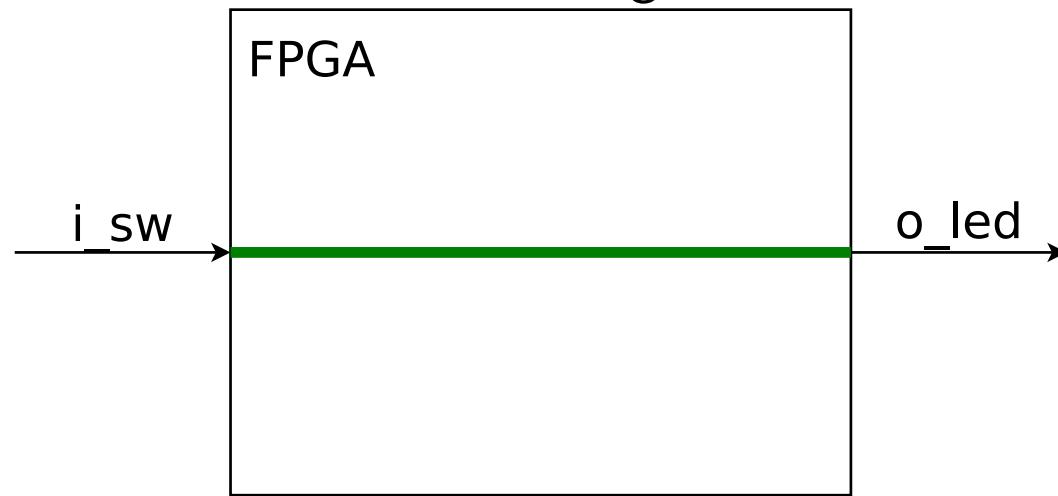
- Voltage level shifters
- This logic would be appropriate for each
... it generates a simple “wire” through the chip

GT Schematic



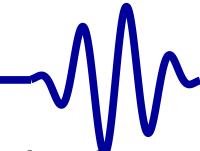
Lesson Overview
First design
▷ Schematic
Constraints
PCF
Build the design
First Success!
Simulation
Verilator Driver
Bus Signals
Bit Select
Internal Signals
Schematic
Circular Logic
Dual Assignment
Sim Result
Examples
Exercise
Conclusion

Here's what a schematic of this design would look like



All from this assign statemnt

```
assign o_led = i_sw;
```



Lesson Overview

First design

▷ Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

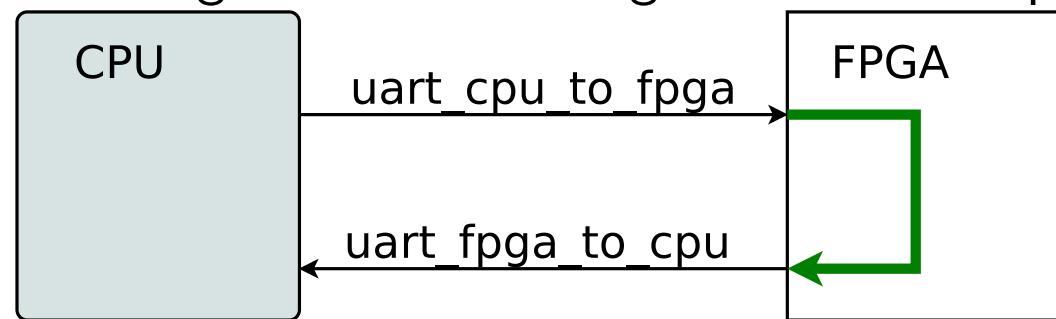
Sim Result

Examples

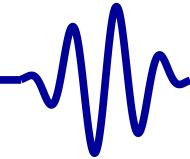
Exercise

Conclusion

A very similar design would make a good first serial port test

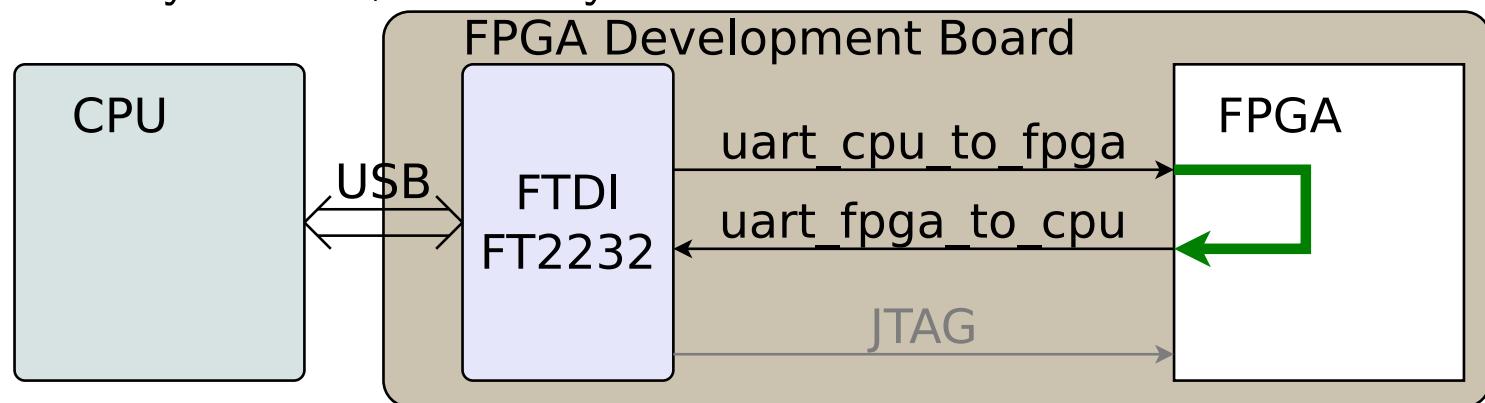


- Your circuit board should pass this test before you try to implement your own serial port within it



Lesson Overview
First design
▷ Schematic
Constraints
PCF
Build the design
First Success!
Simulation
Verilator Driver
Bus Signals
Bit Select
Internal Signals
Schematic
Circular Logic
Dual Assignment
Sim Result
Examples
Exercise
Conclusion

For many boards, this may look more familiar



- FTDI FT2232H provides access to both
 - UART (i.e. serial port), and
 - JTAG, to load your design into the FPGA in the first place
- Other solutions exist, such as
 - A STM chip, as used by the BlackIce, or
 - Direct USB, as used by the TinyFPGA BX

Constraints

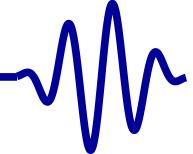


Lesson Overview
First design
Schematic
▷ Constraints
PCF
Build the design
First Success!
Simulation
Verilator Driver
Bus Signals
Bit Select
Internal Signals
Schematic
Circular Logic
Dual Assignment
Sim Result
Examples
Exercise
Conclusion

A fundamental part of any FPGA design maps your ports to the pins

- This is the purpose of a *Constraint File*
- Different vendors use different forms for their constraint files
 - PCF: Used by Arachne-PNR and NextPNR
 - UCF: Used by ISE for older Xilinx designs
 - XDC: Used by Vivado for newer Xilinx designs
 - QSF: Used by Quartus for Altera Intel chips
- Your board vendor should provide you with a master constraint file
- You'll still need to
 - Comment-out pins you aren't using
 - Rename pins to match your Verilog

GT PCF File



Lesson Overview

First design

Schematic

Constraints

▷ PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

Conclusion

If you are using `nextpnr`, you'll need a PCF file

```
set_io i_sw P13
set_io o_led C8
```

- Maps top-level ports to pins
- You'll find P13 and C8 on the schematic
 - Find the FPGA pins connected to the switch
... and the LED output
 - If your design has no switches, you can use buttons
(for now)
Buttons also bounce, but we'll get to that later



UCF File



Lesson Overview

First design

Schematic

Constraints

▷ PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

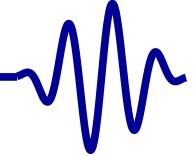
Conclusion

If you are using ISE, you'll need a UCF file

```
NET "i_sw" LOC = "P9" | IO_STANDARD = LVCMOS33;  
NET "o_led" LOC = "N3" | IO_STANDARD = LVCMOS33;
```

- This would be for the older Xilinx FPGA's
- Make sure you actually look up the correct pins
 - P13 for one board might be something else on another
 - On this board, the switch is on pin P9
- Most development boards use the 3.3V LVCMOS standard
 - Pins are typically grouped in banks
 - All pins in a bank use the same voltage
 - This voltage is usually fixed
 - The master constraint file will help here

GT XDC File



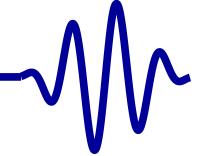
Lesson Overview
First design
Schematic
Constraints
▷ PCF
Build the design
First Success!
Simulation
Verilator Driver
Bus Signals
Bit Select
Internal Signals
Schematic
Circular Logic
Dual Assignment
Sim Result
Examples
Exercise
Conclusion

If you are using Vivado, you'll need a XDC file

```
set_property -dict {PACKAGE_PIN E22
                    IOSTANDARD LVCMOS12} [get_ports {i_sw}]
set_property -dict {PACKAGE_PIN T14
                    IOSTANDARD LVCMOS25} [get_ports {o_led}]
```

- This would be for the newer Xilinx FPGA's
- Usually, the vendor will provide a "master XDC" file
- From there, you should be able to
 - Rename the appropriate ports to `i_sw` and `o_led`
 - Comment out every other I/O port

Build the design



Lesson Overview

First design

Schematic

Constraints

PCF

▷ Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

Conclusion

For an iCE40 design, this will look like:

```
% yosys -p 'synth_ice40 -json thruwire.json' \
    thruwire.v
% nextpnr-ice40 --hx8k --package ct256 \
    --pcf thruwire.pcf --json thruwire.json
% icepack thruwire.asc thruwire.bin
```

You'll need to do this for every project—get used to this flow.

- A makefile can drastically simplify this process

You should now have a file `thruwire.bin` that you can load onto your board.

- If you aren't using an iCE40, follow your chip vendor's instructions



First Success!



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

▷ First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

Conclusion

Follow your board vendor's instructions for loading this file onto your board.

Notice now that every time you flip the switch, the LED responds



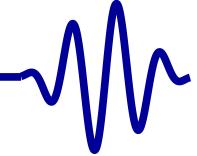
First Success!



Lesson Overview
First design
Schematic
Constraints
PCF
Build the design
▷ First Success!
Simulation
Verilator Driver
Bus Signals
Bit Select
Internal Signals
Schematic
Circular Logic
Dual Assignment
Sim Result
Examples
Exercise
Conclusion

Follow your board vendor's instructions for loading this file onto your board.

Notice now that every time you flip the switch, the LED responds
Yaaaayyyyy!!! Your first FPGA design.



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

▷ Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

Conclusion

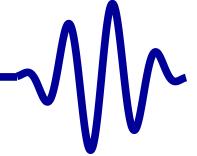
Simulation is an important part of design

| Simulation | Hardware |
|------------------------|---------------------------|
| Can trace all signals | Can only see some signals |
| Extended tests cost GB | Extended tests are simple |
| Easy to debug | <i>Very hard</i> to debug |

Because hardware is so hard to debug, simulation is vital

- A successful complex project
... requires simulation!

GT Simulation



Lesson Overview
First design
Schematic
Constraints
PCF
Build the design
First Success!
▷ Simulation
Verilator Driver
Bus Signals
Bit Select
Internal Signals
Schematic
Circular Logic
Dual Assignment
Sim Result
Examples
Exercise
Conclusion

Simulation is an important part of design

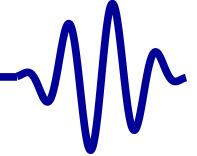
| Simulation | Hardware |
|------------------------|---------------------------|
| Can trace all signals | Can only see some signals |
| Extended tests cost GB | Extended tests are simple |
| Easy to debug | <i>Very hard</i> to debug |

Because hardware is so hard to debug, simulation is vital

- A successful complex project
... requires simulation!

Do it the easy way:

GT Simulation



Lesson Overview
First design
Schematic
Constraints
PCF
Build the design
First Success!
▷ Simulation
Verilator Driver
Bus Signals
Bit Select
Internal Signals
Schematic
Circular Logic
Dual Assignment
Sim Result
Examples
Exercise
Conclusion

Simulation is an important part of design

| Simulation | Hardware |
|------------------------|---------------------------|
| Can trace all signals | Can only see some signals |
| Extended tests cost GB | Extended tests are simple |
| Easy to debug | <i>Very hard</i> to debug |

Because hardware is so hard to debug, simulation is vital

- A successful complex project
... requires simulation!

Do it the easy way: *use the simulator!*



Verilator



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

▷ Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

Conclusion

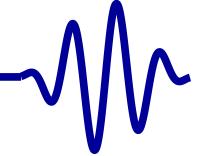
Let's now build our design using Verilator

```
% verilator -Wall -cc thruwire.v  
% cd obj_dir/  
% make -f Vthruwire.mk
```

- Verilator compiles Verilog into C++ placed into `obj_dir/`
- The make command then builds this converted C++ file into a shared object file we can now use



Verilator Driver



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

▷ Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

Conclusion

You'll need a main simulation driver too.

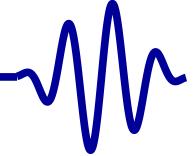
- The code below comes from a file named `thruwire.cpp`

```
#include <stdio.h>
#include <stdlib.h>
#include "Vthruwire.h"
#include "verilated.h"

int main(int argc, char **argv) {
    // Your logic here
}
```



Verilator Driver



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

▷ Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

Conclusion

You'll need a main simulation driver too.

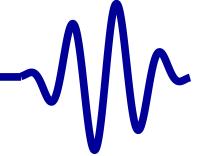
```
// ...
int main(int argc, char **argv) {
    // Call commandArgs first!
    Verilated::commandArgs(argc, argv);

    // Instantiate our design
    Vthruwire *tb = new Vthruwire;

    // ...
}
```



Verilator Driver



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

▷ Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

Conclusion

You'll need a main simulation driver too.

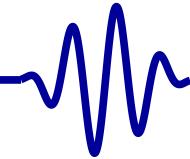
```
int main(int argc, char **argv) {
    // ...
    // Now run the design thru 20 timesteps
    for(int k=0; k<20; k++) {
        // We'll set the switch input
        // to the LSB of our step
        tb->i_sw = k&1;

        tb->eval();

        // ...
    }
}
```



Verilator Driver



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

▷ Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

Conclusion

You'll need a main simulation driver too.

```
int main(int argc, char **argv) {
    // ...
    for(int k=0; k<20; k++) {
        // We'll set the switch input
        // to the LSB of our counter
        tb->i_sw = k&1;

        tb->eval();

        // Now let's print our results
        printf("k=%d\n", k);
        printf("sw=%d\n", tb->i_sw);
        printf("led=%d\n", tb->o_led);
    }
}
```



Building it all



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

▷ Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

Conclusion

Last step, let's put it all together:

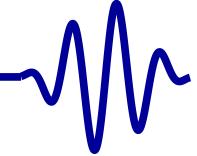
```
% g++ -I /usr/share/verilator/include \
      -I obj_dir/
      /usr/share/verilator/include/verilated.cpp \
      thruwire.cpp obj_dir/Vthruwire__ALL.a \
      -o thruwire
```

(Double check the location of Verilator in your own installation, it might be located in another directory.)

Wow, that's pretty complicated.

You should have a Makefile in your ex-01-thruwire directory with both the code and the build instructions.

```
% cd ex-01-thruwire/
% make
# (Make output skipped for brevity)
%
```



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

▷ Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

Conclusion

We can now run our simulator!

```
% thruwire
k = 0, sw = 0, led = 0
k = 1, sw = 1, led = 1
k = 2, sw = 0, led = 0
k = 3, sw = 1, led = 1
k = 4, sw = 0, led = 0
k = 5, sw = 1, led = 1
k = 6, sw = 0, led = 0
k = 7, sw = 1, led = 1
k = 8, sw = 0, led = 0
k = 9, sw = 1, led = 1
# .... (Lines skipped for brevity)
%
```



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

▷ Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

Conclusion

Many Verilog problems can be avoided by some simple steps

1. Make '**default_nettype none**' the first line of your Verilog file

- Before your **module** declaration
- Otherwise mis-spelled identifiers will be quietly turned into wires

```
module thruwire(i_sw, o_led);
    input wire i_sw;
    output wire o_led;

    assign o_led = sw;
endmodule
```

Without '**default_nettype none**', this design would pass without error

Good habits



Lesson Overview
First design
Schematic
Constraints
PCF
Build the design
First Success!
Simulation
▷ Verilator Driver
Bus Signals
Bit Select
Internal Signals
Schematic
Circular Logic
Dual Assignment
Sim Result
Examples
Exercise
Conclusion

Many Verilog problems can be avoided by some simple steps

1. Make '**default_nettype** none' the first line of your Verilog file
2. Fix any errors when you verilator -Wall your design
3. Run your design in a simulator
 - Attempt to recreate any hardware bugs . . . *in the simulator*

These three rules will save you a lot of heartache!

. . . *Get in the habit of using them!*

[Lesson Overview](#)[First design](#)[Schematic](#)[Constraints](#)[PCF](#)[Build the design](#)[First Success!](#)[Simulation](#)[Verilator Driver](#)[▷ Bus Signals](#)[Bit Select](#)[Internal Signals](#)[Schematic](#)[Circular Logic](#)[Dual Assignment](#)[Sim Result](#)[Examples](#)[Exercise](#)[Conclusion](#)

That was one single wire. We can also declare values consisting of many bits.

```
input  wire [8:0] i_sw;
output wire [8:0] o_led;
```

This defines

- `i_sw` to be 9-input wires, and
- `o_led` to be 9-output wires



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

▷ Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

Conclusion

- Select bits of interest from a bus

```
assign o_led[7] = i_sw[0];  
assign o_led[6:5] = i_sw[5:4];
```

- Bit 7 of o_led is set to bit 0 of i_sw
 - Bits 5 and 6 of o_led are set to bits 4 and 5 of i_sw

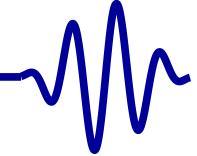
- Concatenate bits together

```
assign o_led[4:0] = { i_sw[2:0], i_sw[7:6] };
```

- The $\{ \cdot, \cdot \}$ operator composes a new bit vector from other vectors



Internal Signals



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

▷ Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

Conclusion

You can also declare and work with internal wires

```
wire [8:0] w_internal;
```

- Internal wires are neither **input** nor **output**
- These wires can now be used in logic

```
assign w_internal = 9'h87;  
assign o_led = i_sw ^ w_internal;
```

Literals



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

▷ Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

Exercise

Conclusion

A Verilog literal is defined as

- A width
- An apostrophe
- An optional sign indication, s

Defaults to unsigned

- A numeric type: h (hex), d (decimal), o (octal), b (binary), sd (signed decimal)
- The value: a series of digits, possibly containing underscores
Underscores can be *very useful* for longer numbers

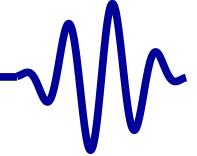
Examples include:

1'b0 1'b1 2'b01 4'b0101 4'h5 -7'sd124

32'hdead_beef 32'd100_000_000

Place a '-' in front of the width for negative numbers

Sign Extension



Lesson Overview
First design
Schematic
Constraints
PCF
Build the design
First Success!
Simulation
Verilator Driver
Bus Signals
Bit Select
▷ Internal Signals
Schematic
Circular Logic
Dual Assignment
Sim Result
Examples
Exercise
Conclusion

If the literal is smaller than the context . . .

- If there is no 's', the number is unsigned and it is zero extended
- Any literal with an 's' is sign extended
 - . . . to fit the width

If the literal is too big for the context . . .

- It is truncated to fit the context

Many tools will create a warning for width mismatches

Operators



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

▷ Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

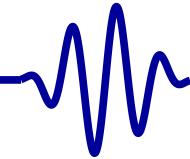
Exercise

Conclusion

The Verilog language supports the following operators

| | | | |
|------------------------|-------------------|--------------------|------------------------|
| + | Addition | - | Subtraction |
| << | Left Shift | >> | Right shift |
| - | Unary negation | ?: | Tertiary operator |
| ~ | Bit-wise negation | ^ | Bit-wise XOR |
| | Bitwise OR | & | Bitwise AND |
| | Logical OR | && | Logical and |
| ! | Logical negation | >>> | Arithmetic right shift |
| == | Equality | != | Inequality |
| <, <= | Less than (Equal) | >, >= | Greater than (Equal) |
| Limited, use with care | | Avoid within logic | |
| * | Multiplication | / | Division |
| | | % | Remainder |

- Some FPGA's support native multiplication
- None support a single clock divide or remainder



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

▷ Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

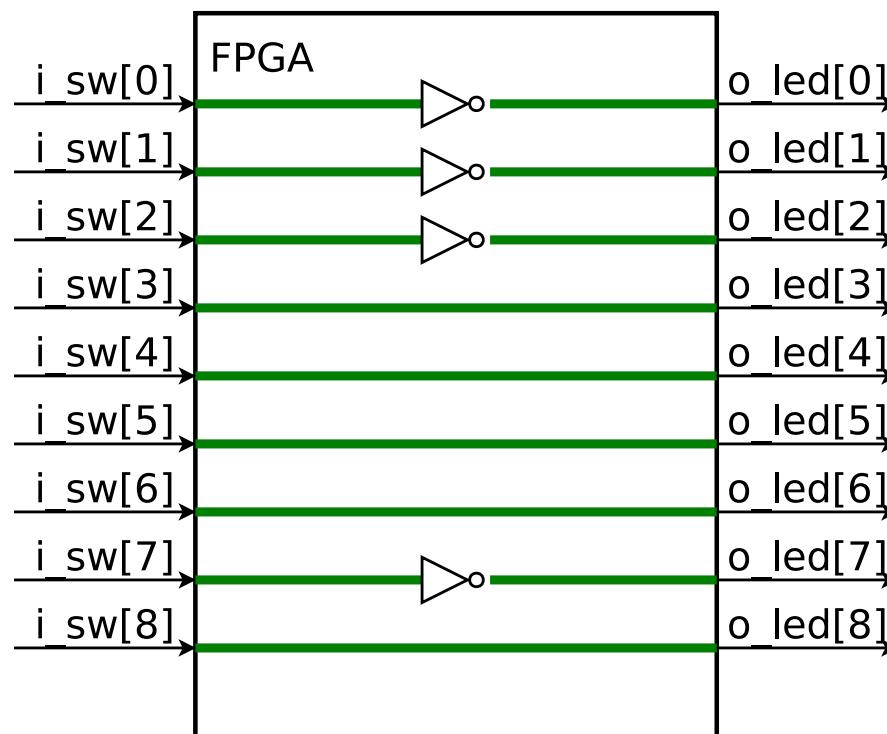
Exercise

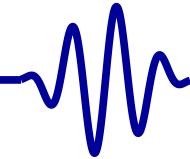
Conclusion

From this code:

```
assign w_internal = 9'h87;  
assign o_led = i_sw ^ w_internal;
```

Get this internal structure:





Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

▷ Circular Logic

Dual Assignment

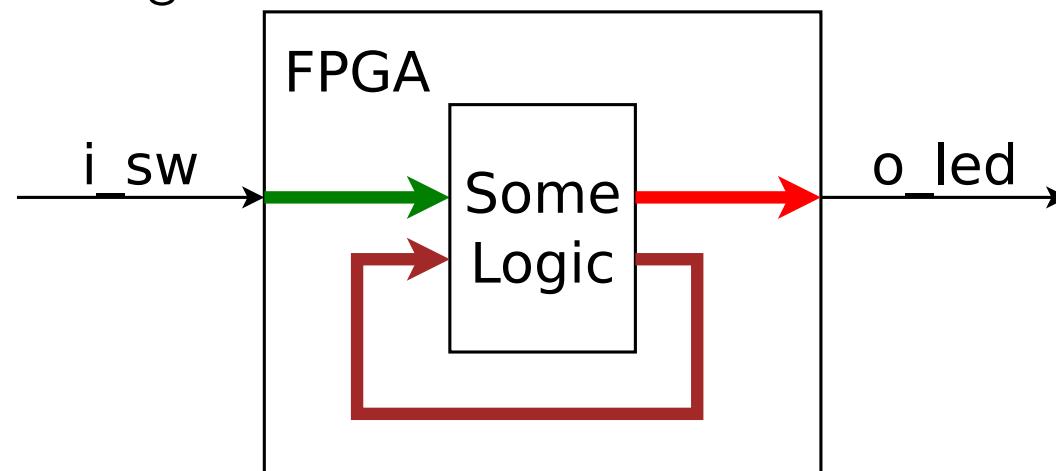
Sim Result

Examples

Exercise

Conclusion

Avoid circular logic!



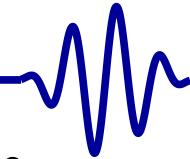
Example:

```
assign o_led = i_sw + o_led;
```

- This doesn't work in hardware like it might in software
- This is roughly equivalent to creating a short circuit
- Most tools will fail to build such designs
This include Verilator



Dual Assignment



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual

▷ Assignment

Sim Result

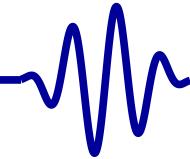
Examples

Exercise

Conclusion

You are designing hardware: A value can only be set once
This is an error:

```
assign o_led = i_sw | 9'h87;  
assign o_led = i_sw + 1;
```

[Lesson Overview](#)[First design](#)[Schematic](#)[Constraints](#)[PCF](#)[Build the design](#)[First Success!](#)[Simulation](#)[Verilator Driver](#)[Bus Signals](#)[Bit Select](#)[Internal Signals](#)[Schematic](#)[Circular Logic](#)[Dual](#)[▷ Assignment](#)[Sim Result](#)[Examples](#)[Exercise](#)[Conclusion](#)

Let's build this design:

```
'default_nettype none

module maskbus( i_sw , o_led );
    input wire [8:0] i_sw;
    output wire [8:0] o_led;

    wire [8:0] w_internal;

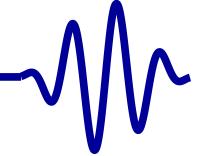
    assign w_internal = 9'h87;
    assign o_led = i_sw ^ w_internal;

endmodule
```

... using Verilator



Updated Driver



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual

▷ Assignment

Sim Result

Examples

Exercise

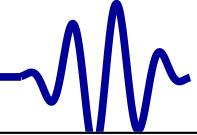
Conclusion

Let's update our driver for this wire bus design

```
int main(int argc, char **argv) {
    // ...
    for(int k=0; k<20; k++) {
        // ...
        // Bottom 9 bits of counter
        tb->i_sw = k & 0x1ff;

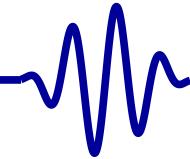
        tb->eval();

        // Now let's print our results
        printf("k=%d\n", k);
        printf("sw=%3x\n", tb->i_sw);
        printf("led=%3x\n", tb->o_led);
    }
}
```



Lesson Overview
First design
Schematic
Constraints
PCF
Build the design
First Success!
Simulation
Verilator Driver
Bus Signals
Bit Select
Internal Signals
Schematic
Circular Logic
Dual Assignment
▷ Sim Result
Examples
Exercise
Conclusion

```
% . /maskbus
k = 0, sw = 0, led = 87
k = 1, sw = 1, led = 86
k = 2, sw = 2, led = 85
k = 3, sw = 3, led = 84
k = 4, sw = 4, led = 83
k = 5, sw = 5, led = 82
k = 6, sw = 6, led = 81
k = 7, sw = 7, led = 80
k = 8, sw = 8, led = 8f
k = 9, sw = 9, led = 8e
# .... (Lines skipped for brevity)
%
```



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

▷ Examples

Exercise

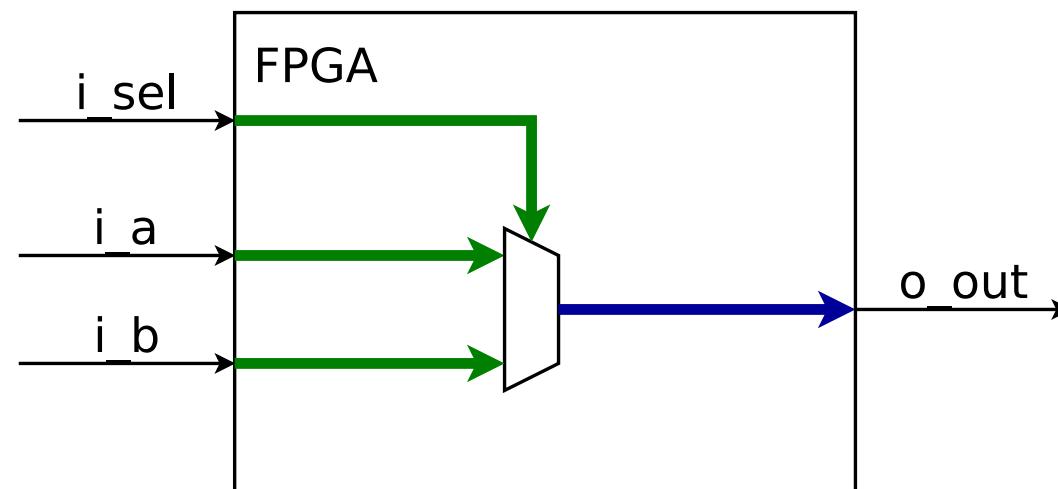
Conclusion

What can you do with wires and wire logic?

Example: Multiplexer

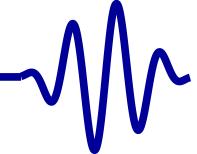
```
input    wire    i_a, i_b, i_sel;
output   wire    o_out;

assign   o_out = (i_sel) ? i_a : i_b;
```





Examples



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

▷ Examples

Exercise

Conclusion

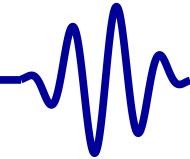
What can you do with wires and wire logic?

Example: Multiplexer

```
input    wire    i_a, i_b, i_sel;  
output   wire    o_out;  
  
assign   o_out = (i_sel) ? i_a : i_b;
```

- This is a good example of the tertiary operator
- Interested in making a connection to one of two serial ports?
- How about connecting one of two bus masters to an interconnect?

We'll get to these examples later.



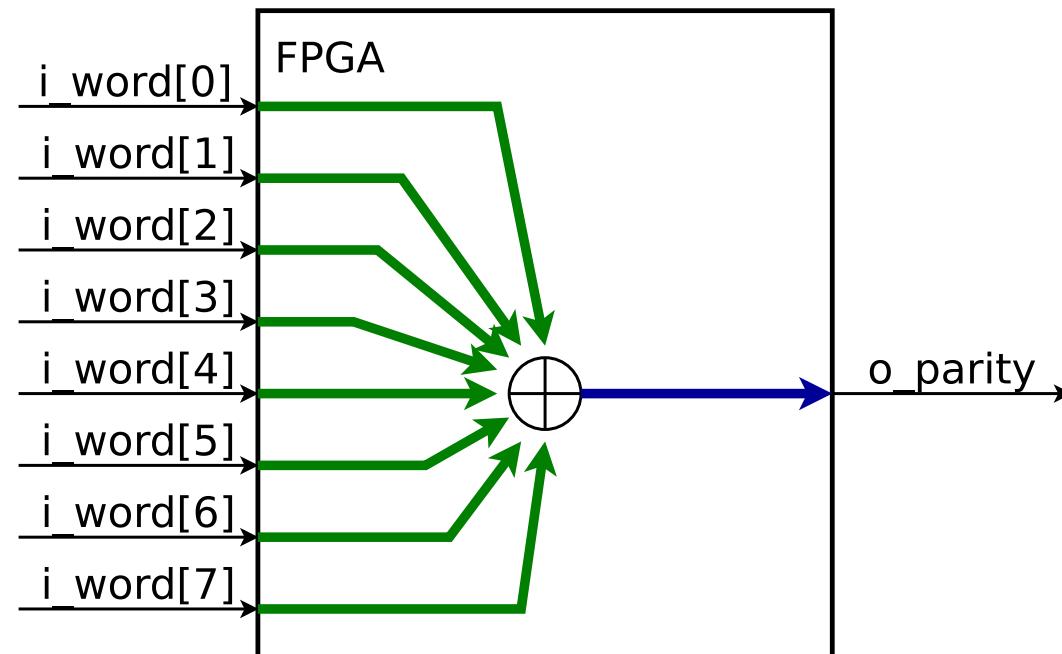
Lesson Overview
First design
Schematic
Constraints
PCF
Build the design
First Success!
Simulation
Verilator Driver
Bus Signals
Bit Select
Internal Signals
Schematic
Circular Logic
Dual Assignment
Sim Result
▷ Examples
Exercise
Conclusion

What can you do with wires and wire logic?

Example: Parity check

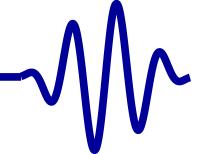
```
input    wire [7:0] i_word;
output   wire          o_parity;

assign   o_parity = ^ i_word;
```





Examples



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

▷ Examples

Exercise

Conclusion

What can you do with wires and wire logic?

Example: Parity check

```
input    wire    [7:0]    i_word;
output   wire
assign   oParity = ^i_word;
```

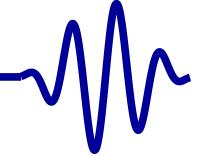
This form of XOR is a *reduction operator*

- It XORs all the word's bits together
- Other reduction operators include | and &

Error Correction Code (ECC) creation logic is very similar



Examples



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

▷ Examples

Exercise

Conclusion

What can you do with wires and wire logic?

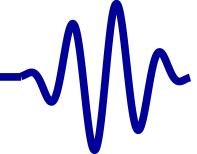
Example: Interrupt detector

```
input    wire    [7:0]    i_irq_source;
output   wire
assign   o_irq = | i_irq_source;
```

- `i_irq_source` contains eight interrupt sources
- `o_irq` is true if any interrupt source is true



Examples



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

▷ Examples

Exercise

Conclusion

What can you do with wires and wire logic?

Example: CPU stall determination

```
assign dcd_stall = (dcd_valid)&&(op_stall);
```

From the ZipCPU, the decode stage must stall if

- It has produced a valid result, and
- The next stage, read operands, is stalled for some reason
These stalls can back up through the CPU
Ex. Read operands might be stalled if the ALU is stalled



Examples



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

▷ Examples

Exercise

Conclusion

What can you do with wires and wire logic?

Example: Determining if there's a phase error in a phase lock loop

```
assign phase_err = (output_phase != input_phase);
```

In this case, the loop will adjust if there are any errors



Exercise



Lesson Overview

First design

Schematic

Constraints

PCF

Build the design

First Success!

Simulation

Verilator Driver

Bus Signals

Bit Select

Internal Signals

Schematic

Circular Logic

Dual Assignment

Sim Result

Examples

▷ Exercise

Conclusion

This section has two exercises:

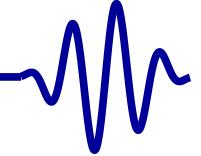
1. Build and try the `thruwire` demo.

- Toggle the switch.
- Verify that toggling your switch will toggle the LED
- Build and run the Verilator simulation

2. Create a test of your serial port connection

- Connecting the input serial port wire to the output
Beware: These wires are often marked “TX” and “RX”,
but not always from the perspective of the FPGA
- Turn off any ‘local echo’
- Turn off any hardware flow control
- Verify that characters typed into your terminal program
show up on the screen

Conclusion



- Wires represent connections within the design
- Wires can also represent the outputs of combinatorial logic
- Wires have no memory, circular logic or feedback is illegal
- You know how to create constraints for your project!

You can now build and load a design onto an FPGA!

Lesson Overview
First design
Schematic
Constraints
PCF
Build the design
First Success!
Simulation
Verilator Driver
Bus Signals
Bit Select
Internal Signals
Schematic
Circular Logic
Dual Assignment
Sim Result
Examples
Exercise
▷ Conclusion



Gisselquist
Technology, LLC

2. Registers
Daniel E. Gisselquist, Ph.D.





Lesson Overview



▷ Lesson Overview

Registers

Combinatorial

Latches

Flip Flops

Blocking

All in Parallel

Feedback

Blinky

Broken Blinky

Verilator

Parameters

Sim Result

Trace Generation

GTKWave

Strobe

PPS-I

PPS-II

Stretch

Too Slow

Dimmer

Exercises

- What is a register (**reg**)?
- How do things change with time?
- Discover the system clock

Objectives

- Learn how to create combinatorial logic with registers
- Learn to create clocked (synchronous) logic
- Understand that registers can “remember” things
- Understand where your System Clock comes from
- Timing Checks, and why they are important

Registers



Lesson Overview
▷ Registers
Combinatorial
Latches
Flip Flops
Blocking
All in Parallel
Feedback
Blinky
Broken Blinky
Verilator
Parameters
Sim Result
Trace Generation
GTKWave
Strobe
PPS-I
PPS-II
Stretch
Too Slow
Dimmer
Exercises

Why use registers?

- Wires have no memory
- Only registers can hold state (data)

Two basic types, both set with an **always**

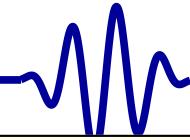
1. Combinatorial: Like wires

```
always @(*)  
    A = B;
```

This form can be easier to read when the logic becomes complex

2. Synchronous: Only changes values on a clock

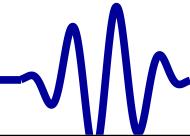
```
always @ (posedge i_clk)  
    A <= B;
```



```
always @(*)  
    A = 9'h87;
```

- Registers can only be assigned in **always** blocks.
- Always blocks may consist of one statement, or
- Many statements between a **begin** and **end** pair

```
always @(*)  
begin  
    o_led = A ^ i_sw;  
    o_led = o_led + 7;  
    if (i_reset)  
        o_led = 0;  
end
```

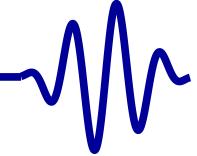


```
always @(*)
begin
    o_led = A ^ i_sw;
    o_led = o_led + 7;
    if (i_reset)
        o_led = 0;
end
```

This block

- Looks like software
- Acts like you would expect in a simulator
- Takes no time at all in hardware
 - The hardware acts as if all statements were done at once

Only use “=” in a combinatorial always block



Lesson Overview

Registers

Combinatorial

▷ Latches

Flip Flops

Blocking

All in Parallel

Feedback

Blinky

Broken Blinky

Verilator

Parameters

Sim Result

Trace Generation

GTKWave

Strobe

PPS-I

PPS-II

Stretch

Too Slow

Dimmer

Exercises

What happens here?

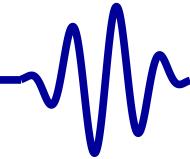
```
input    wire          i_S ;
input    wire [7:0]    i_V ;
output   reg [7:0]    o_R ;

always @(*)
  if (i_S)
    o_R = i_V ;
```

This is called a latch

- It requires memory
- May do one thing in simulation, another in hardware
- Most FPGA's don't support latches
- Can have subtle timing problems in hardware

Avoid using latches!



Lesson Overview

Registers

Combinatorial

▷ Latches

Flip Flops

Blocking

All in Parallel

Feedback

Blinky

Broken Blinky

Verilator

Parameters

Sim Result

Trace Generation

GTKWave

Strobe

PPS-I

PPS-II

Stretch

Too Slow

Dimmer

Exercises

What happens here?

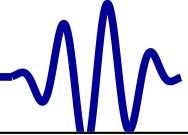
```
always @(*)
begin
    o_R = 0;
    if (i_S)
        o_R = i_V;
end
```

No latch is inferred

- This is a very useful pattern!
- o_R now has a default value
This prevents a latch from being inferred
- No memory is required
- The last assignment gives o_R its final value



Flip Flops

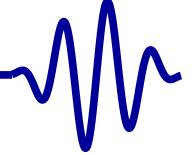


Lesson Overview
Registers
Combinatorial
Latches
▷ Flip Flops
Blocking
All in Parallel
Feedback
Blinky
Broken Blinky
Verilator
Parameters
Sim Result
Trace Generation
GTKWave
Strobe
PPS-I
PPS-II
Stretch
Too Slow
Dimmer
Exercises

```
reg [9:0] A;  
  
always @ (posedge i_clk)  
    A <= A + 1'b1;
```

- Any registers set within an **always @(posedge i_clk)** block will transition to their new values on the next clock edge only
 - *Only a bonafide clock edge should be used*
 - Do not transition on anything you create in logic
- Note that we are using \leq for assignment
 - This is a *non-blocking* assignment
 - Most, if not all, clocked register should be set with \leq

Blocking



Lesson Overview
Registers
Combinatorial
Latches
Flip Flops
▷ Blocking
All in Parallel
Feedback
Blinky
Broken Blinky
Verilator
Parameters
Sim Result
Trace Generation
GTKWave
Strobe
PPS-I
PPS-II
Stretch
Too Slow
Dimmer
Exercises

- This is a non-blocking assignment

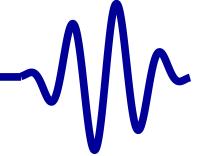
```
always @(posedge i_clk)
    A <= A + 1'b1;
```

- Blocking assignment

```
always @(posedge i_clk)
    A = A + 1'b1;
```

- A blocking assignment's value may be referenced again before the clock edge
 - Creates the appearance of time passing within the block
 - *It may also cause simulation-hardware mismatch*
 - *Use with caution*
- In this case, both generate the same logic

Non-Blocking



What value will be given for A?

- Assume it starts at zero
- What will it be after one clock tick?

```
always @(posedge i_clk)
begin
    A <= 5;
    A <= A + 1'b1;
end
```

- The assignment only takes place on the clock edge
- Last assignment wins
- A is set to 1, then 2 on the next clock, 3 on the clock after, etc.

Blocking



Lesson Overview
Registers
Combinatorial
Latches
Flip Flops
▷ Blocking
All in Parallel
Feedback
Blinky
Broken Blinky
Verilator
Parameters
Sim Result
Trace Generation
GTKWave
Strobe
PPS-I
PPS-II
Stretch
Too Slow
Dimmer
Exercises

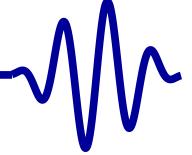
Now what value will be given for A?

- Assume it starts at zero
- What will it be after one clock tick?

```
always @(posedge i_clk)
begin
    A = 5;
    A = A + 1'b1;
end
```

- Again, the assignment only takes place on the clock edge
- It appears as though it took several steps
- A is set to 6

Blocking



Lesson Overview
Registers
Combinatorial
Latches
Flip Flops
▷ Blocking
All in Parallel
Feedback
Blinky
Broken Blinky
Verilator
Parameters
Sim Result
Trace Generation
GTKWave
Strobe
PPS-I
PPS-II
Stretch
Too Slow
Dimmer
Exercises

What if something depends upon A in another block?

- Assume A=0 before the clock tick

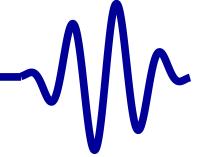
```
always @(posedge i_clk)
begin
    A = 5;
    A = A + 1'b1;
end

always @(posedge i_clk)
    B <= A;
```

- This result is *simulation dependent!*
- B may be set to 0, or it may be set to 6

Don't do this! Use `<=` within an `always @(posedge i_clk)`

Non-Blocking



Lesson Overview
Registers
Combinatorial
Latches
Flip Flops
▷ Blocking
All in Parallel
Feedback
Blinky
Broken Blinky
Verilator
Parameters
Sim Result
Trace Generation
GTKWave
Strobe
PPS-I
PPS-II
Stretch
Too Slow
Dimmer
Exercises

Now what will B be set to?

- Assume A=0 before the clock tick

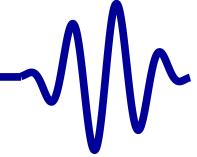
```
always @(posedge i_clk)
begin
    A <= 5; // Ignored!
    A <= A + 1'b1;
end

always @(posedge i_clk)
    B <= A;
```

- A will be set to 1, and B will be set to 0
- On the next clock, A will be set to 2 and B to 1, etc.

Now simulation matches hardware

All in Parallel



Lesson Overview

Registers

Combinatorial

Latches

Flip Flops

Blocking

▷ All in Parallel

Feedback

Blinky

Broken Blinky

Verilator

Parameters

Sim Result

Trace Generation

GTKWave

Strobe

PPS-I

PPS-II

Stretch

Too Slow

Dimmer

Exercises

- A design may contain multiple always blocks
- The hardware will execute all at once
- The simulator will execute one at a time

Rules: When using the simulator, . . .

- Make sure your design can be synthesized
- Make sure it fits within your chosen device
 - This is not a simulator task
 - Requires using the synthesizer periodically
- Make sure it maintains an appropriate clock rate
 - We'll get to timing checks in a moment



Feedback



Lesson Overview
Registers
Combinatorial
Latches
Flip Flops
Blocking
All in Parallel
▷ Feedback
Blinky
Broken Blinky
Verilator
Parameters
Sim Result
Trace Generation
GTKWave
Strobe
PPS-I
PPS-II
Stretch
Too Slow
Dimmer
Exercises

- Wires in a loop created circular logic
- Clocked registers in a loop creates feedback

```
assign err = i_actual - o_command;  
always @ (posedge i_clk)  
begin  
    o_command <= o_command + (err >> 5);  
end
```

Feedback is used commonly in control systems

Blinky



Let's make an LED blink!

```
module blinky(i_clk, o_led);
    input wire i_clk;
    output wire o_led;

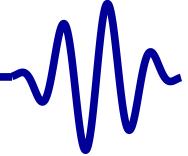
    reg [26:0] counter;
    initial counter = 0;
    always @(posedge i_clk)
        counter <= counter + 1'b1;

    assign o_led = counter[26];
endmodule
```

Feel free to synthesize and try this

- The LED should blink at a steady rate
- Rate is determined by the 26 above

Broken Blinky



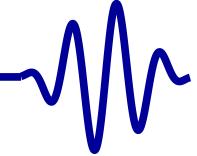
Lesson Overview
Registers
Combinatorial
Latches
Flip Flops
Blocking
All in Parallel
Feedback
Blinky
▷ Broken Blinky
Verilator
Parameters
Sim Result
Trace Generation
GTKWave
Strobe
PPS-I
PPS-II
Stretch
Too Slow
Dimmer
Exercises

Here's a common beginner mistake

```
reg      counter;  
  
always @(posedge i_clk)  
    counter <= counter + 1'b1;  
  
assign o_led = counter;
```

Don't make this mistake

- Notice that counter is only 1-bit
- This will blink at half the i_clk frequency
- Result is typically way too fast to see any changes
- LED may glow dimly
- Need to slow it down



Lesson Overview

Registers

Combinatorial

Latches

Flip Flops

Blocking

All in Parallel

Feedback

Blinky

Broken Blinky

▷ Verilator

Parameters

Sim Result

Trace Generation

GTKWave

Strobe

PPS-I

PPS-II

Stretch

Too Slow

Dimmer

Exercises

Simulating our design (blinky) now requires a clock:

```
void      tick(Vblinky *tb) {  
    // The following eval() looks  
    // redundant ... many of hours  
    // of debugging reveal its not  
    tb->eval();  
    tb->i_clk = 1;  
    tb->eval();  
    tb->i_clk = 0;  
    tb->eval();  
}
```

- We'll need to toggle the clock input for anything to happen
- This operation is so common, it deserves its own function,
tick()

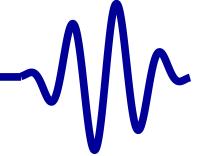
[Lesson Overview](#)[Registers](#)[Combinatorial](#)[Latches](#)[Flip Flops](#)[Blocking](#)[All in Parallel](#)[Feedback](#)[Blinky](#)[Broken Blinky](#)[▷ Verilator](#)[Parameters](#)[Sim Result](#)[Trace Generation](#)[GTKWave](#)[Strobe](#)[PPS-I](#)[PPS-II](#)[Stretch](#)[Too Slow](#)[Dimmer](#)[Exercises](#)

We can now simplify our main loop a touch

```
int main(int argc, char **argv) {
    int last_led;
    // .... Setup

    last_led = tb->o_led;
    for(int k=0; k<(1<<20); k++) {
        // Toggle the clock
        tick(tb);

        // Now let's print the LEDs value
        // anytime it changes
        if (last_led != tb->o_led) {
            printf("k=%d, ", k);
            printf("led=%d\n", tb->o_led);
        } last_led = tb->o_led;
    }
}
```



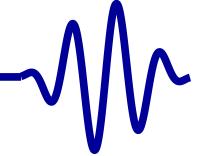
Can we simulate this? Not easily

- Counting to 2^{27} may take seconds in hardware, but . . .
- It's extreme slow in simulation.
- Let's speed blinky up—just for simulation
- We can do this by adjusting the width of the counter

We'll use a parameter to do this

```
parameter      WIDTH=27;  
reg          [WIDTH-1:0]      counter;  
// . . .  
assign     o_led = counter[WIDTH-1];
```

Parameters



Lesson Overview
Registers
Combinatorial
Latches
Flip Flops
Blocking
All in Parallel
Feedback
Blinky
Broken Blinky
Verilator
▷ Parameters
Sim Result
Trace Generation
GTKWave
Strobe
PPS-I
PPS-II
Stretch
Too Slow
Dimmer
Exercises

Parameters are very powerful! They allow us to

- Reconfigure a design, after it's been written
- Examples:
 - ZipCPU cache sizes can be adjusted by parameters
 - Internal memory sizes, implement the divide instruction or not, specify the type of multiply
 - Default serial port speed, number of GPIO pins supported by a GPIO controller, and more

Verilator argument `-GWIDTH=12` sets the `WIDTH` parameter to 12

```
% verilator -Wall -GWIDTH=12 -cc blinky.v
```



Sim Result



Lesson Overview
Registers
Combinatorial
Latches
Flip Flops
Blocking
All in Parallel
Feedback
Blinky
Broken Blinky
Verilator
Parameters
▷ Sim Result
Trace Generation
GTKWave
Strobe
PPS-I
PPS-II
Stretch
Too Slow
Dimmer
Exercises

```
% ./blinky
k = 2047, led = 1
k = 4095, led = 0
k = 6143, led = 1
k = 8191, led = 0
k = 10239, led = 1
k = 12287, led = 0
k = 14335, led = 1
k = 16383, led = 0
k = 18431, led = 1
k = 20479, led = 0
# .... (Lines skipped for brevity)
%
```



Trace Generation



Lesson Overview
Registers
Combinatorial
Latches
Flip Flops
Blocking
All in Parallel
Feedback
Blinky
Broken Blinky
Verilator
Parameters
Sim Result
 Trace
▷ Generation
GTKWave
Strobe
PPS-I
PPS-II
Stretch
Too Slow
Dimmer
Exercises

This is easy. For more complex designs, we'll need a trace

- That means writing to a trace file on every clock

Steps

1. Add the --trace option to the Verilator command line

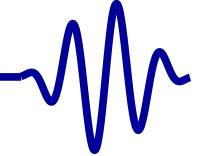
```
% verilator -Wall --trace -GWIDTH=12 \
              -cc blinky.v
```

2. Create a trace from your .cpp file

```
#include "verilated_vcd_c.h"
// ...
int main(int argc, char **argv) {
    // ...
    unsigned tickcount = 0;
    // ...
```



Trace Generation



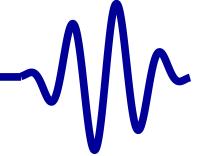
Lesson Overview
Registers
Combinatorial
Latches
Flip Flops
Blocking
All in Parallel
Feedback
Blinky
Broken Blinky
Verilator
Parameters
Sim Result
 Trace
▷ Generation
GTKWave
Strobe
PPS-I
PPS-II
Stretch
Too Slow
Dimmer
Exercises

Create the trace file within C++

```
// ...  
int main(int argc, char **argv) {  
    // ...  
    // Generate a trace  
    Verilated::traceEverOn(true);  
    VerilatedVcdC* tfp = new VerilatedVcdC;  
    tb->trace(tfp, 99);  
    tfp->open("blinkytrace.vcd");  
  
    // ...  
    for(int k=0; k<(1<<20); k++) {  
        tick(++tickcount, tb, tfp);  
        // ...  
    }  
}
```



Trace Generation



Lesson Overview
Registers
Combinatorial
Latches
Flip Flops
Blocking
All in Parallel
Feedback
Blinky
Broken Blinky
Verilator
Parameters
Sim Result
 Trace
▷ Generation
GTKWave
Strobe
PPS-I
PPS-II
Stretch
Too Slow
Dimmer
Exercises

3. Write trace data on every clock

```
void tick(int tickcount, Vblinky *tb,
          VerilatedVcdC* tfp) {
    tb->eval();
    if (tfp) // dump 2ns before the tick
        tfp->dump(tickcount * 10 - 2);
    tb->i_clk = 1;
    tb->eval();
    if (tfp) // Tick every 10ns
        tfp->dump(tickcount * 10);
    tb->i_clk = 0;
    tb->eval();
    if (tfp) { // Trailing edge dump
        tfp->dump(tickcount * 10 + 5);
        tfp->flush();
    }
}
```



Trace Generation



Lesson Overview
Registers
Combinatorial
Latches
Flip Flops
Blocking
All in Parallel
Feedback
Blinky
Broken Blinky
Verilator
Parameters
Sim Result
 Trace
▷ Generation
GTKWave
Strobe
PPS-I
PPS-II
Stretch
Too Slow
Dimmer
Exercises

- You'll need to add `verilated_vcd_c.cpp` to your `g++` build command in order to support generating a trace as well

```
% export VINC=/usr/share/verilator/include
% g++ -I${VINC} -I obj_dir
    ${VINC}/verilated.cpp
    ${VINC}/verilated_vcd_c.cpp blinky.cpp
    obj_dir/Vblinky__ALL.a -o blinky
```

- Now, running `blinky` will generate a trace

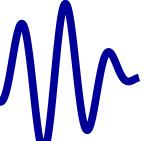
```
% ./blinky
# ...
```

- You can view it with GTKwave

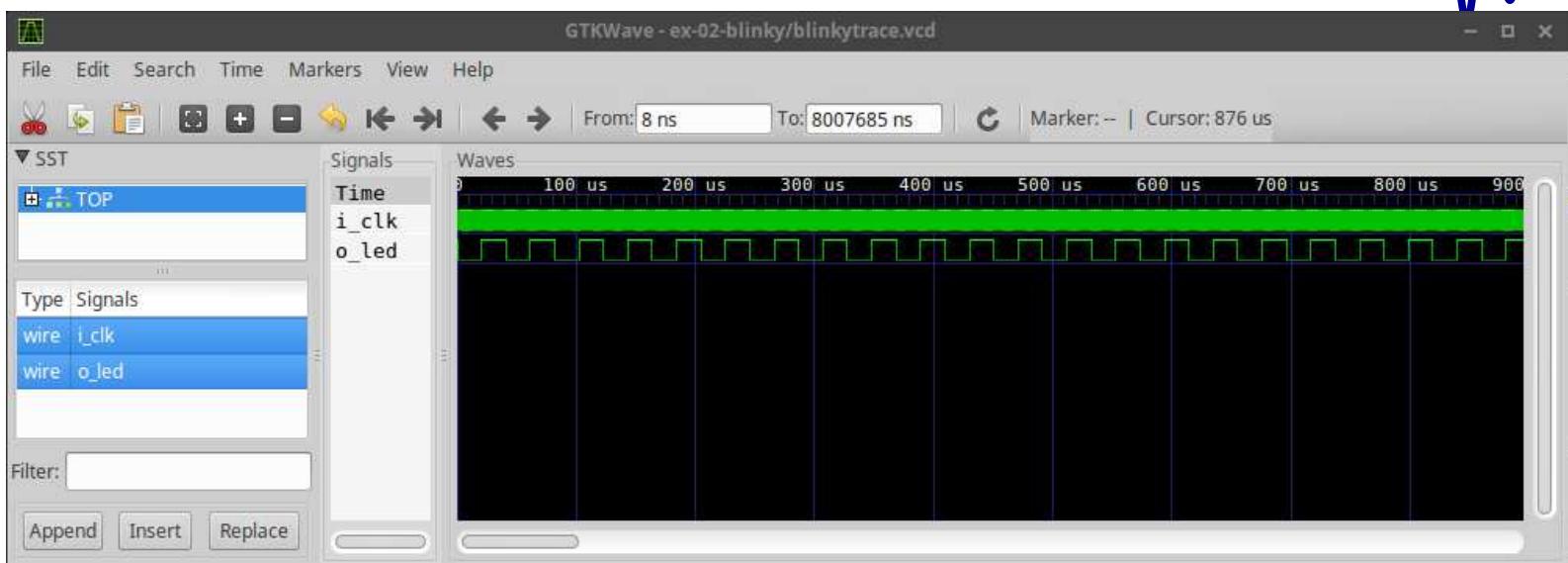
```
% gtkwave blinkytrace.vcd
```



GTKWave



Lesson Overview
Registers
Combinatorial
Latches
Flip Flops
Blocking
All in Parallel
Feedback
Blinky
Broken Blinky
Verilator
Parameters
Sim Result
Trace Generation
▷ GTKWave
Strobe
PPS-I
PPS-II
Stretch
Too Slow
Dimmer
Exercises

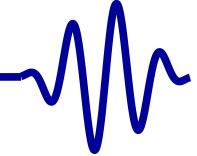


This is how logic debugging is done

- The simulator trace shows you every register's value
 - ... at every clock tick
 - You can zoom in to find any bugs



Strobe



Lesson Overview

Registers

Combinatorial

Latches

Flip Flops

Blocking

All in Parallel

Feedback

Blinky

Broken Blinky

Verilator

Parameters

Sim Result

Trace Generation

GTKWave

▷ Strobe

PPS-I

PPS-II

Stretch

Too Slow

Dimmer

Exercises

How is this design different from blinky?

```
module strobe(i_clk, o_led);
    input wire i_clk;
    output wire o_led;

    reg [26:0] counter;

    always @(posedge i_clk)
        counter <= counter + 1'b1;

    assign o_led = &counter[26:24];
endmodule
```

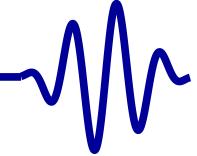
[Lesson Overview](#)[Registers](#)[Combinatorial](#)[Latches](#)[Flip Flops](#)[Blocking](#)[All in Parallel](#)[Feedback](#)[Blinky](#)[Broken Blinky](#)[Verilator](#)[Parameters](#)[Sim Result](#)[Trace Generation](#)[GTKWave](#)[Strobe](#)[▷ PPS-I](#)[PPS-II](#)[Stretch](#)[Too Slow](#)[Dimmer](#)[Exercises](#)

Can we get an LED to blink once per second?

```
always @(posedge i_clk)
  if (counter >= CLOCK_RATE_HZ/2-1)
    begin
      counter <= 0;
      o_led <= !o_led;
    end else
      counter <= counter + 1;
```

When CLOCK_RATE_HZ/2 ticks have passed, the LED will toggle

- This structure is known as an integer clock divider
- It offers an exact division

[Lesson Overview](#)[Registers](#)[Combinatorial](#)[Latches](#)[Flip Flops](#)[Blocking](#)[All in Parallel](#)[Feedback](#)[Blinky](#)[Broken Blinky](#)[Verilator](#)[Parameters](#)[Sim Result](#)[Trace Generation](#)[GTKWave](#)[Strobe](#)[PPS-I](#)[▷ PPS-II](#)[Stretch](#)[Too Slow](#)[Dimmer](#)[Exercises](#)

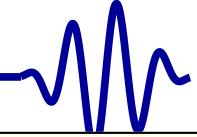
Can we get an LED to blink once per second?

```
parameter CLOCK_RATE_HZ = 100_000_000;
parameter [31:0] INCREMENT
              = (1<<30)/(CLOCK_RATE_HZ / 4);
input     wire      i_clk;
output    wire      o_led;

reg       [31:0]   counter;

initial counter = 0;
always @ (posedge i_clk)
          counter <= counter + INCREMENT;

assign   o_led = counter[31];
```



Lesson Overview
Registers
Combinatorial
Latches
Flip Flops
Blocking
All in Parallel
Feedback
Blinky
Broken Blinky
Verilator
Parameters
Sim Result
Trace Generation
GTKWave
Strobe
PPS-I
▷ PPS-II
Stretch
Too Slow
Dimmer
Exercises

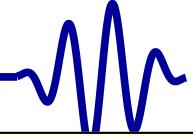
```
parameter CLOCK_RATE_HZ = 100_000_000;  
parameter [31:0] INCREMENT  
          = (1<<30)/(CLOCK_RATE_HZ / 4);  
always @(*posedge i_clk)  
    counter <= counter + INCREMENT;
```

- After CLOCK_RATE_HZ clock edges, the counter will roll over
- The divide by four above, on both numerator and denominator, is just to keep this within 32-bit arithmetic

$$\text{INCREMENT} = \frac{2^{32}}{\text{CLOCK_RATE_HZ}}$$

- This is called a *fractional clock divider*
 - The division isn't exact
 - It's often good enough

Stretch



Lesson Overview
Registers
Combinatorial
Latches
Flip Flops
Blocking
All in Parallel
Feedback
Blinky
Broken Blinky
Verilator
Parameters
Sim Result
Trace Generation
GTKWave
Strobe
PPS-I
PPS-II
▷ Stretch
Too Slow
Dimmer
Exercises

```
module stretch(i_clk, i_event, o_led);
    input wire i_clk, i_event;
    output wire o_led;

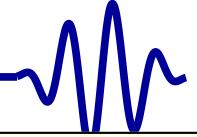
    reg [26:0] counter;

    always @(posedge i_clk)
        if (i_event)
            counter <= 0;
        else if (!counter[26])
            counter <= counter + 1;

    assign o_led = !counter[26];
endmodule
```

FPGA signals are often too fast to see

Stretch

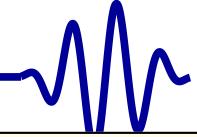


Lesson Overview
Registers
Combinatorial
Latches
Flip Flops
Blocking
All in Parallel
Feedback
Blinky
Broken Blinky
Verilator
Parameters
Sim Result
Trace Generation
GTKWave
Strobe
PPS-I
PPS-II
▷ Stretch
Too Slow
Dimmer
Exercises

```
module stretch(i_clk, i_event, o_led);
    // ...
    reg [26:0] counter;
    always @(posedge i_clk)
        if (i_event)
            counter <= 0;
        else if (!counter[26])
            counter <= counter + 1;
        assign o_led = !counter[26];
endmodule
```

FPGA signals are often too fast to see

- This slows them down to eye speed
- Only works for a single event though
- Multiple events would overlap, and be no longer distinct



Lesson Overview
Registers
Combinatorial
Latches
Flip Flops
Blocking
All in Parallel
Feedback
Blinky
Broken Blinky
Verilator
Parameters
Sim Result
Trace Generation
GTKWave
Strobe
PPS-I
PPS-II
Stretch
▷ Too Slow
Dimmer
Exercises

```
module tooslow(i_clk, o_led);
    input wire i_clk;
    output wire o_led;

    parameter NBITS = 1024;
    reg [NBITS-1:0] counter;

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

    assign o_led = counter[NBITS-1];
endmodule
```

This is guaranteed to fail a timing check

- It's now time to learn how to check timing
- This design should fail, for reasonable clock speeds

Too Slow

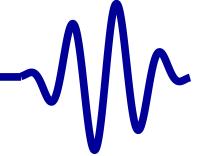


Follow your chip vendor's instructions to do a timing check

- Use your system clock frequency
 - For now, that's the clock frequency coming into your board
 - We'll adjust it later
- Make sure this design fails
 - The carry chain takes time to propagate
 - Extra long carry chains take extra long
 - If the propagation doesn't complete before the next clock . . . your design will fail (like this one)
- From now on, *always* check timing for a design
 - Before loading it onto a board
 - Every now and then while simulating



Dimmer



Lesson Overview

Registers

Combinatorial

Latches

Flip Flops

Blocking

All in Parallel

Feedback

Blinky

Broken Blinky

Verilator

Parameters

Sim Result

Trace Generation

GTKWave

Strobe

PPS-I

PPS-II

Stretch

Too Slow

▷ Dimmer

Exercises

Can you tell me what this will do?

```
module dimmer(i_clk, o_led);
    input wire i_clk;
    output wire o_led;

    reg [27:0] counter;

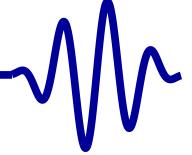
    always @(posedge i_clk)
        counter <= counter + 1;

    assign o_led = (counter[7:0]
                    < counter[27:20]);

endmodule
```



Exercises



Lesson Overview
Registers
Combinatorial
Latches
Flip Flops
Blocking
All in Parallel
Feedback
Blinky
Broken Blinky
Verilator
Parameters
Sim Result
Trace Generation
GTKWave
Strobe
PPS-I
PPS-II
Stretch
Too Slow
Dimmer
▷ Exercises

- Implement blinky on your hardware
- Implement one of the two PPS designs
 - Using a stopwatch, verify the blink rate of 1Hz
 - Make the blinks shorter, but at the same frequency
- Verify that the 1024 bit `tooslow` counter will fail timing
- Implement the dimmer



Gisselquist
Technology, LLC

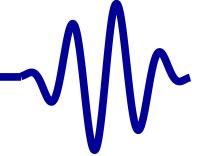
3. Finite State Machines

Daniel E. Gisselquist, Ph.D.





Lesson Overview

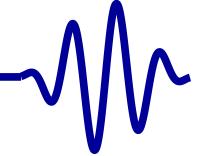


▷ Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

- What is a Finite State Machine?
- Why do I need it?
- How do I build one?

Objectives

- Learn the concatenation operator
- Be able to explain a shift register
- To get basic understanding of Finite State Machines
- To learn how to build and use Finite State Machines

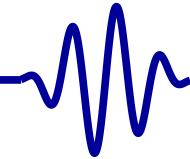


Lesson Overview
▷ Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

The concatenation operator

```
always @(posedge i_clk)
    o_led <= { o_led[6:0], o_led[7] };
```

Composes a new bit-vector from other pieces



The concatenation operator

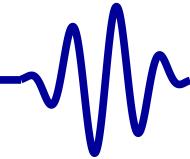
```
always @ (posedge i_clk)
    o_led <= { o_led[6:0], o_led[7] };
```

Simplifies what otherwise would be quite painful

```
always @ (posedge i_clk)
begin
    o_led[0] <= o_led[7];
    o_led[1] <= o_led[0];
    o_led[2] <= o_led[1];
    o_led[3] <= o_led[2];
    o_led[4] <= o_led[3];
    o_led[5] <= o_led[4];
    o_led[6] <= o_led[5];
    o_led[7] <= o_led[6];
end
```



Shift Register



Lesson Overview
▷ Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

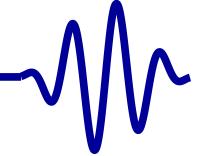
A shift register shifts bits through a register

- Can shift from LSB to MSB

```
always @ (posedge i_clk)
    o_led <= { o_led [6:0] , i_input };
```

- or from MSB to LSB

```
always @ (posedge i_clk)
    o_led <= { i_input , o_led [7:1] };
```



You can use this to create a neat LED display as well

- You just need to mix the shift register

```
initial o_led = 8'h1;
always @(posedge i_clk)
if (stb)
    o_led <= { o_led[6:0], o_led[7] };
```

- With a counter to slow it down

```
reg [26:0] counter;
reg stb;
initial { stb, counter } = 0;
always @(posedge i_clk)
{ stb, counter } <= counter + 1'b1;
```

- stb here is a *strobe* signal. A *strobe* signal is true for one clock only, whenever an event takes place

Shift Register



Lesson Overview
▷ Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

You can use this to create a neat LED display as well

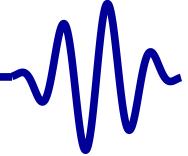
- You just need to mix the shift register

```
initial o_led = 8'h1;
always @(posedge i_clk)
if (stb)
    o_led <= { o_led[6:0], o_led[7] };
```

- With a counter to slow it down

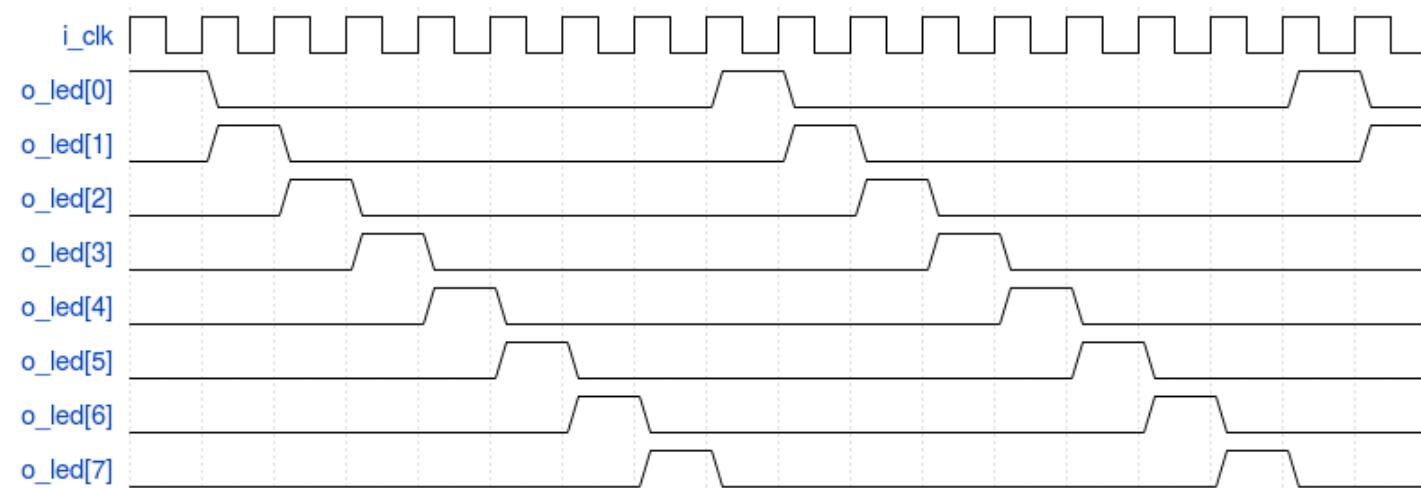
```
reg [26:0] counter;
reg stb;
initial { stb, counter } = 0;
always @(posedge i_clk)
{ stb, counter } <= counter + 1'b1;
```

- Note that you can *assign* to a concatenation as well



Lesson Overview
Shift Register
▷ Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

If you've never seen [Wavedrom](#), it is an awesome tool!
Here's a waveform description of our shift register



```
initial o_led = 8'h1;  
always @ (posedge i_clk)  
    o_led <= { o_led[6:0], o_led[7] };
```

What would it take to make the LED's go *back and forth*?



LED Walker



Lesson Overview
Shift Register
Wavedrom
▷ LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

Let's build an LED walker!

- Active LED should walk across valid LED's and back

We'll assume 8 LEDs

Shift registers don't naturally go both ways

- Only one LED should be active at any time
- One LED should always be active at any given time

Most of this project can be done in simulation



Lesson Overview

Shift Register

Wavedrom

LED Walker

▷ Wavedrom

The Need

Case Statement

The Need

The addresses

Simulation

Finite State Machine

Simple

Mealy

Moore

One Process FSM

Two Process FSM

Which to use?

Formal Verification

Assertion

SymbiYosys

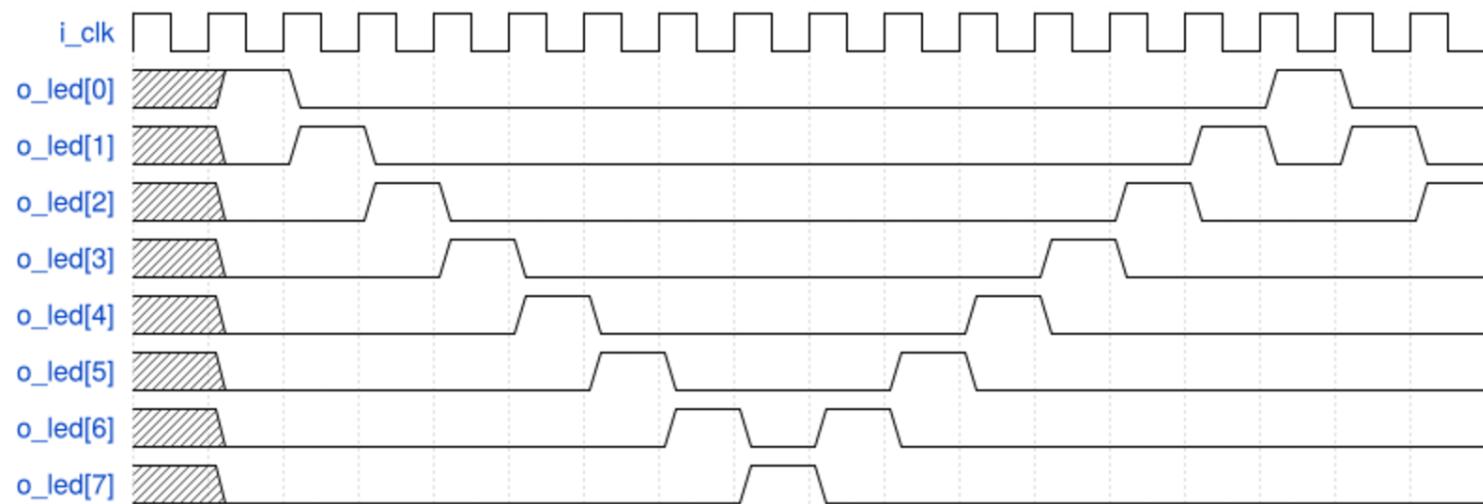
Integer Clock

Divider

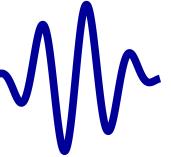
Exercise

Conclusion

Here's a waveform description of what I want this design to do

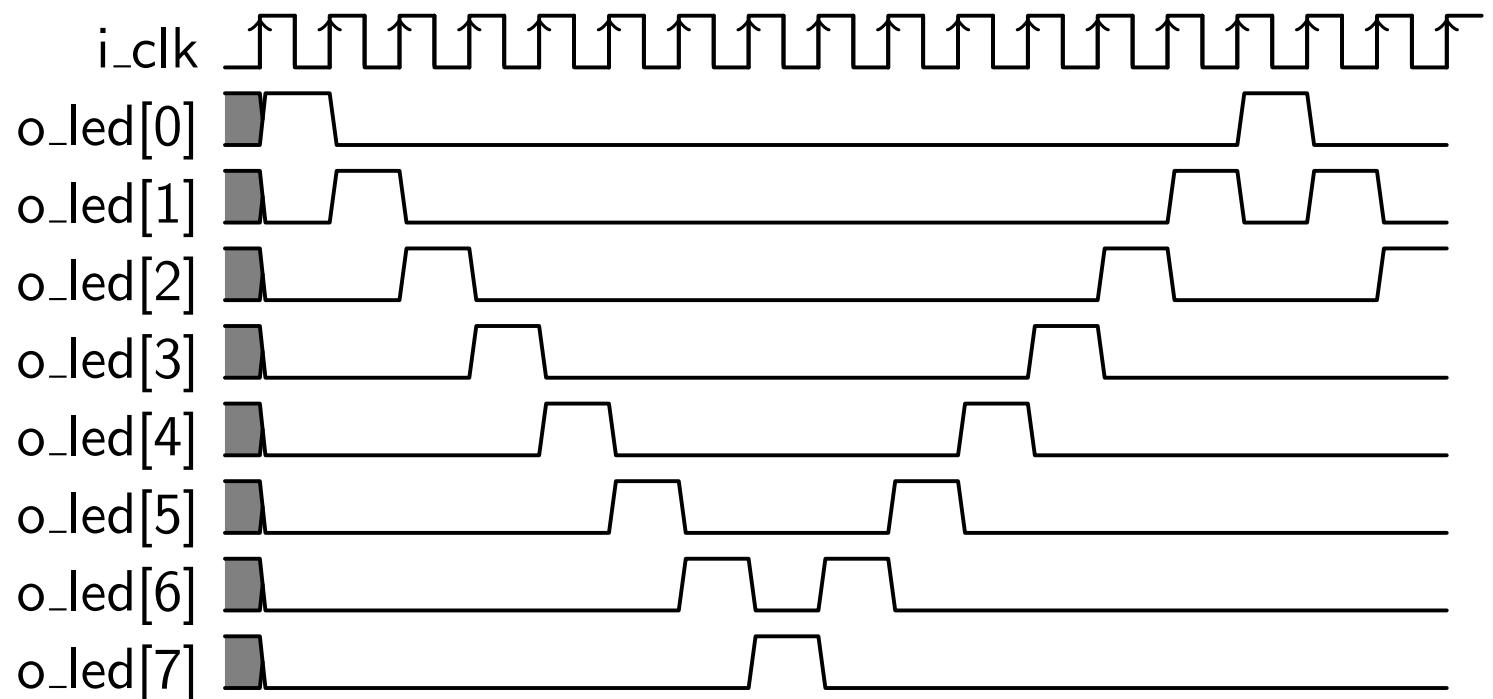


- This “goal” diagram can help mitigate complexity



Lesson Overview
Shift Register
Wavedrom
LED Walker
▷ Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

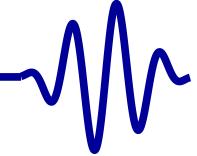
Tikz-timing also works nicely for L^AT_EX users



- Our goal will be to create a design with these outputs
- If successful, you'll see this in GTKwave



The Need



Lesson Overview

Shift Register

Wavedrom

LED Walker

Wavedrom

▷ The Need

Case Statement

The Need

The addresses

Simulation

Finite State Machine

Simple

Mealy

Moore

One Process FSM

Two Process FSM

Which to use?

Formal Verification

Assertion

SymbiYosys

Integer Clock

Divider

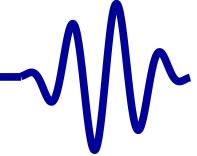
Exercise

Conclusion

Were we building in C, this would be our program

```
while (1) {  
    o_led = 0x01;  
    o_led = 0x02;  
    o_led = 0x04;  
    // ...  
    o_led = 0x80;  
    o_led = 0x40;  
    // ...  
    o_led = 0x04;  
    o_led = 0x02;  
}
```

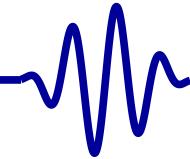
How do we turn this code into Verilog?

[Lesson Overview](#)[Shift Register](#)[Wavedrom](#)[LED Walker](#)[Wavedrom](#)[The Need](#)[▷ Case Statement](#)[The Need](#)[The addresses](#)[Simulation](#)[Finite State Machine](#)[Simple](#)[Mealy](#)[Moore](#)[One Process FSM](#)[Two Process FSM](#)[Which to use?](#)[Formal Verification](#)[Assertion](#)[SymbiYosys](#)[Integer Clock](#)[Divider](#)[Exercise](#)[Conclusion](#)

We could use a giant cascaded **if** statement

```
always @(posedge i_clk)
  if (o_led == 8'b0000_0001)
    o_led <= 8'h02;
  else if (o_led == 8'b0000_0010)
    o_led <= 8'h04;
  else if (o_led == 8'b0000_0100)
    o_led <= 8'h08;
  else if (o_led == 8'b0000_1000)
    o_led <= 8'h08;

  // ...
  // Don't forget a final else!
  else // if (o_led == 8'b0000_0010)
    o_led <= 8'h01
```



Lesson Overview

Shift Register

Wavedrom

LED Walker

Wavedrom

The Need

▷ Case Statement

The Need

The addresses

Simulation

Finite State Machine

Simple

Mealy

Moore

One Process FSM

Two Process FSM

Which to use?

Formal Verification

Assertion

SymbiYosys

Integer Clock

Divider

Exercise

Conclusion

We could use a giant case statement

```
always @(posedge i_clk)
  case(o_led)
    8'b0000_0001: o_led <= 8'h02;
    8'b0000_0010: o_led <= 8'h04;
    // ...
    8'b0010_0000: o_led <= 8'h40;
    8'b0100_0000: o_led <= 8'h80;
    8'b1000_0000: o_led <= 8'h40;
    // ...
    8'b0000_0100: o_led <= 8'h02;
    8'b0000_0010: o_led <= 8'h01;
    default: o_led <= 8'h01;
  endcase
```

Can anyone see a problem with these two approaches?



The Need



Lesson Overview

Shift Register

Wavedrom

LED Walker

Wavedrom

The Need

▷ Case Statement

The Need

The addresses

Simulation

Finite State Machine

Simple

Mealy

Moore

One Process FSM

Two Process FSM

Which to use?

Formal Verification

Assertion

SymbiYosys

Integer Clock

Divider

Exercise

Conclusion

A better way: Let's assign an index to each of these outputs

```
// ... using C++ notation again
    o_led = 0x01;      // 1
    o_led = 0x02;      // 2
    o_led = 0x04;      // 3
    //
    o_led = 0x80;      // 8
    o_led = 0x40;      // 9
    //
    o_led = 0x04;      // 13
    o_led = 0x02;      // 14
```

In software, you might think of this as an *instruction address*



Lesson Overview

Shift Register

Wavedrom

LED Walker

Wavedrom

The Need

▷ Case Statement

The Need

The addresses

Simulation

Finite State Machine

Simple

Mealy

Moore

One Process FSM

Two Process FSM

Which to use?

Formal Verification

Assertion

SymbiYosys

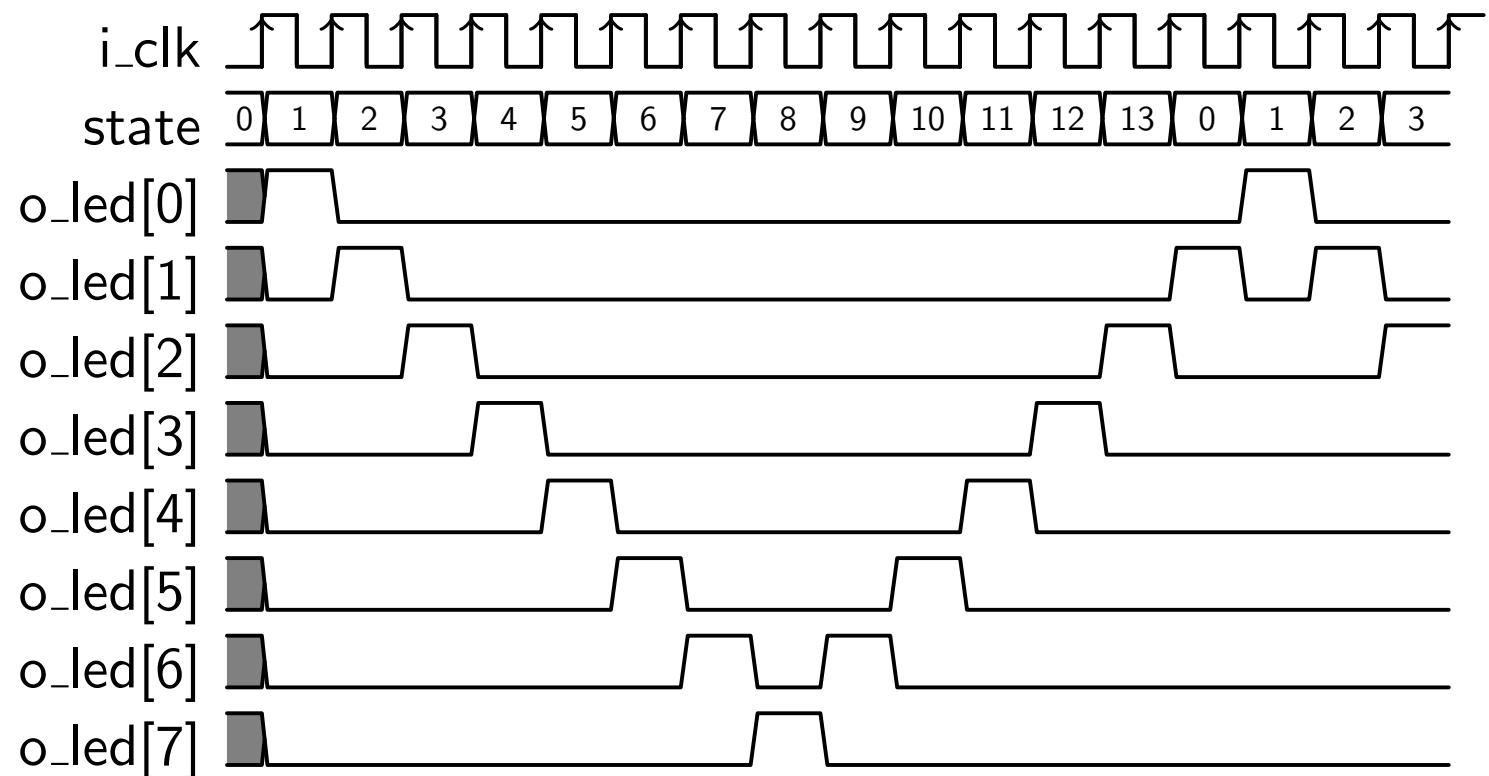
Integer Clock

Divider

Exercise

Conclusion

Here's what an updated waveform diagram might look like



- Our goal will be to create a design with these outputs
- If successful, you'll see this in GTKwave



Lesson Overview

Shift Register

Wavedrom

LED Walker

Wavedrom

The Need

Case Statement

▷ The Need

The addresses

Simulation

Finite State Machine

Simple

Mealy

Moore

One Process FSM

Two Process FSM

Which to use?

Formal Verification

Assertion

SymbiYosys

Integer Clock

Divider

Exercise

Conclusion

We can now set the result based upon the *instruction address*

```
always @(posedge i_clk)
  case(led_index)
    4'h0:   o_led <= 8'h01;
    4'h1:   o_led <= 8'h02;
    4'h2:   o_led <= 8'h04;
    // ...
    4'h7:   o_led <= 8'h80;
    4'h8:   o_led <= 8'h40;
    // ...
    4'hc:   o_led <= 8'h02;
    4'hd:   o_led <= 8'h01;
    default: o_led <= 8'h01;
  endcase
```

- This is an example of a *finite state machine*



The addresses



Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
▷ The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

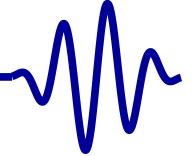
All we need now is something to drive the *instruction address*

- This is known as the *state* of our finite state machine

```
initial led_index = 0; // Our "state" variable
always @(posedge i_clk)
if (led_index >= 4'd13)
    led_index <= 0;
else
    led_index <= led_index + 1'b1;
```



Simulation



Lesson Overview

Shift Register

Wavedrom

LED Walker

Wavedrom

The Need

Case Statement

The Need

The addresses

▷ Simulation

Finite State Machine

Simple

Mealy

Moore

One Process FSM

Two Process FSM

Which to use?

Formal Verification

Assertion

SymbiYosys

Integer Clock

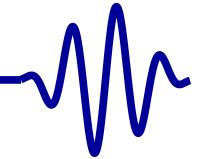
Divider

Exercise

Conclusion

Go ahead and simulate this design

- Does it work as intended?
- Did we miss anything?



Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
 Finite State
 > Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

A finite state machine consists of...

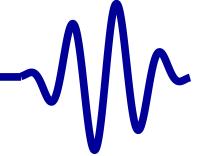
- Inputs
- State Variable,

Finite means there are a limited number of states

- Outputs



Finite State Machine



Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
 Finite State
 > Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

A finite state machine consists of...

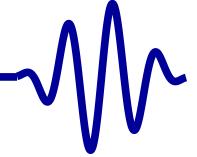
- Inputs // we didn't have any
- State Variable, // led_index , or addr

Finite means there are a limited number of states

- Outputs // o_led

Keep it just that simple.

Simple



Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
▷ Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

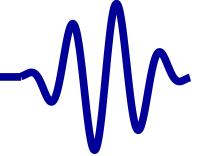
- State machines are conceptually very simple
- We'll ignore the excess math here

Two classical FSM forms

- Mealy
- Moore

Two implementation approaches

- One process
- Two process



Lesson Overview

Shift Register

Wavedrom

LED Walker

Wavedrom

The Need

Case Statement

The Need

The addresses

Simulation

Finite State Machine

Simple

▷ Mealy

Moore

One Process FSM

Two Process FSM

Which to use?

Formal Verification

Assertion

SymbiYosys

Integer Clock

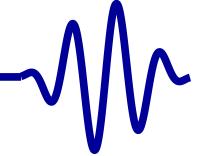
Divider

Exercise

Conclusion

Outputs depend upon the current state *plus inputs*

```
always @(*)
  if (!i_display_enable)
    o_led = 0;
  else
    case(led_index)
      4'h1: o_led = 8'h01;
      4'h2: o_led = 8'h02;
      4'h3: o_led = 8'h04;
      4'h4: o_led = 8'h08;
      // ...
    endcase
```



Lesson Overview

Shift Register

Wavedrom

LED Walker

Wavedrom

The Need

Case Statement

The Need

The addresses

Simulation

Finite State Machine

Simple

Mealy

► Moore

One Process FSM

Two Process FSM

Which to use?

Formal Verification

Assertion

SymbiYosys

Integer Clock

Divider

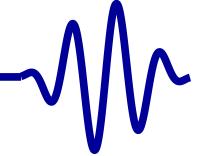
Exercise

Conclusion

Outputs depend upon the *current state only*

```
// Update the state
always @(posedge i_clk)
    enabled <= i_display_enable;
// Create the outputs
always @(*)
if (!enabled)
    o_led = 0;
else
    case(led_index)
        4'h1: o_led = 8'h01;
        4'h2: o_led = 8'h02;
        // ...
    endcase
```

The inputs are then used to determine the next state



A one process state machine

- Created with *synchronous* always block(s)

```
initial led_index = 0; // Our "state" variable
always @(posedge i_clk)
begin
    if (led_index >= 4'hE)
        led_index <= 0;
    else
        led_index <= led_index + 1'b1;

    case(led_index)
        4'h0: o_led <= 8'h01;
        // ...
    endcase
end
```

[Lesson Overview](#)[Shift Register](#)[Wavedrom](#)[LED Walker](#)[Wavedrom](#)[The Need](#)[Case Statement](#)[The Need](#)[The addresses](#)[Simulation](#)[Finite State Machine](#)[Simple](#)[Mealy](#)[Moore](#)[One Process FSM](#)[Two Process](#)[▷ FSM](#)[Which to use?](#)[Formal Verification](#)[Assertion](#)[SymbiYosys](#)[Integer Clock](#)[Divider](#)[Exercise](#)[Conclusion](#)

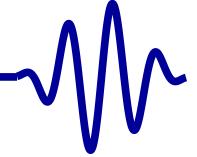
Two Process FSM uses both synchronous and *combinatorial* logic

```
always @(*)
begin
    if (led_index >= 4'hE)
        next_led_index = 0;
    else
        next_led_index
            = next_led_index + 1'b1;
    case(led_index)
        4'h0: o_led = 8'h01;
        // ...
    endcase
end

always @(posedge i_clk)
    led_index <= next_led_index;
```



Which to use?



Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
▷ Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

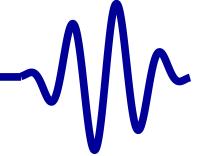
Pick whichever finite state machine form . . .

- . . . you are most comfortable with

There is no right answer here



Which to use?



Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
▷ Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

Pick whichever finite state machine form . . .

- . . . you are most comfortable with

There is no right answer here

but people still argue about it!

- Tastes great
- Less Filling

I tend to use one process FSM's



Formal Verification



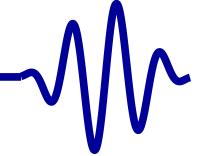
Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
 Formal
 ▷ Verification
Assertion
SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

Formal Verification is a process to prove your design “works”

- Fairly easy to use
- Can be faster and easier than simulation
- Most valuable
 - Early in the design process
 - For design *components*, and not entire designs



Formal Verification



Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
 Formal
 ▷ Verification
Assertion
SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

Formal Verification

- You specify properties your design must have
- A solver attempts to prove if your design has them
- If the solver fails
 - It will tell you what property failed
 By line number
 - It will generate a trace showing the failure
- These traces tend to be much shorter than simulation failure traces

Assertion



Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
▷ Assertion
SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

The free version of Yosys supports immediate assertions

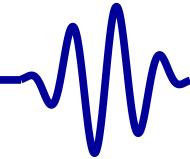
Two types

- Clocked – only checked on clock edges

```
// Remember how we only
// used some of the states?
always @(posedge i_clk)
    assert(led_state <= 4'd13);
```

- Combinational – always checked

```
always @(*)
    assert(led_state <= 4'd13);
```



Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
▷ SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

To verify this design using SymbiYosys,

- You'll need a script

```
[options]
```

```
mode prove
```

```
[engines]
```

```
smtbmc
```

```
[script]
```

```
read -formal ledwalker.v
```

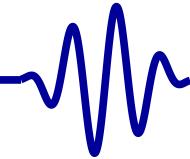
```
# ... other files would go here
```

```
prep -top ledwalker
```

```
[files]
```

```
# List all files and relative paths here
```

```
ledwalker.v
```



Lesson Overview

Shift Register

Wavedrom

LED Walker

Wavedrom

The Need

Case Statement

The Need

The addresses

Simulation

Finite State Machine

Simple

Mealy

Moore

One Process FSM

Two Process FSM

Which to use?

Formal Verification

Assertion

▷ SymbiYosys

Integer Clock

Divider

Exercise

Conclusion

1. BMC (Bounded Model Checking)

[**options**]

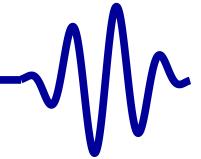
mode bmc

depth 20

- Examines the first N steps (20 in this case)
- ... looking for a way to break your assertion(s)
- Can find property (i.e. **assert**) failures
- An **assert** is a *safety* property
 - Succeeds only if *no trace* can be found that makes any one of your assertions fail



Three Basic FV Modes



Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
▷ SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

1. BMC (Bounded Model Checking)
2. Cover

```
[ options ]
mode cover
depth 20
```

- Examines the first N steps (20 in this case)
- ... looking for a way to make any cover statement *pass*

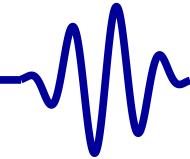
```
always @(posedge i_clk)
    cover(led_state == 4'he);
```

- No trace will be generated if no way is found
- **cover** is a *liveness* property

Succeeds if one trace, any trace, can be found to make the statement *true*



Three Basic FV Modes



Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
▷ SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

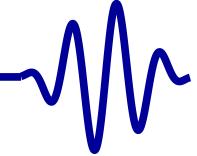
1. BMC (Bounded Model Checking)
2. Cover
3. Full proof using k -induction

```
[ options ]  
mode prove  
depth 20
```

- Examines the first N steps (20 in this case)
- Also examines an arbitrary N steps
starting in an arbitrary state
The induction step will ignore your **initial** statements
Correct functionality must be guaranteed using **assert** statements
- Can prove your properties hold for all time
- This is also a *safety* property check



Example property

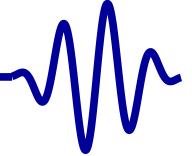


Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
▷ SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

Assert the design can only contain one of eight outputs

```
always @(*)
begin
    f_valid_output = 0;
    case(o_led)
        8'h01: f_valid_output = 1'b1;
        8'h02: f_valid_output = 1'b1;
        8'h04: f_valid_output = 1'b1;
        8'h08: f_valid_output = 1'b1;
        8'h10: f_valid_output = 1'b1;
        8'h20: f_valid_output = 1'b1;
        8'h40: f_valid_output = 1'b1;
        8'h80: f_valid_output = 1'b1;
    endcase
    assert(f_valid_output);
end
```

It doesn't work



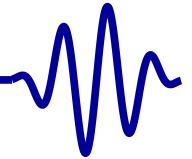
Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
▷ SymbiYosys
Integer Clock
Divider
Exercise
Conclusion

If you try implementing this design as it is now,

- You'll be disappointed
- All the LED's will light dimly

The LED's will toggle so fast you cannot see them change

We need a way to slow this down.



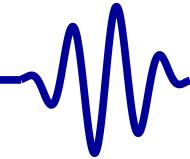
Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
 Integer Clock
 ▷ Divider
Exercise
Conclusion

You may remember the integer clock divider

- Let's use it here

```
always @ (posedge i_clk)
if (wait_counter == 0)
    wait_counter <= CLK_RATE_HZ - 1;
else
    wait_counter <= wait_counter - 1'b1;

always @ (posedge i_clk)
begin
    stb <= 1'b0;
    if (wait_counter == 0)
        stb <= 1'b1;
end
```



This wait_counter is limited in range

- It will only range from 0 to CLK_RATE_HZ-1
- Don't forget the assertion that wait_counter remains in range!

```
always @(posedge i_clk)
    assert(wait_counter <= CLK_RATE_HZ - 1);
```

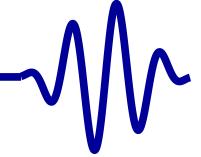
If your state variable can only take on some values, always make an assertion to that affect

- Let's also make sure the stb matches the wait_counter too

```
always @(posedge i_clk)
    assert(stb == (wait_counter == 0));
```



Integer Clock Divider



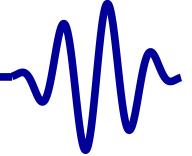
Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
 Integer Clock
 ▷ Divider
Exercise
Conclusion

Now we can use stb to tell us when to adjust our state

```
initial led_index = 0;
always @(posedge i_clk)
if (stb)
begin
    // The logic inside is just
    // what it was before
    // Only the if(stb) changed
    if (led_index >= 4'd13)
        led_index <= 0;
    else
        led_index <= led_index + 1'b1;
end // else nothing changes
// wait for stb to be true before changing state
```



Exercise



Lesson Overview

Shift Register

Wavedrom

LED Walker

Wavedrom

The Need

Case Statement

The Need

The addresses

Simulation

Finite State Machine

Simple

Mealy

Moore

One Process FSM

Two Process FSM

Which to use?

Formal Verification

Assertion

SymbiYosys

Integer Clock

Divider

▷ Exercise

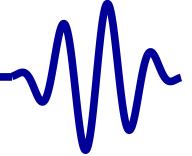
Conclusion

Try out the tools

1. Recreate this waveform using Wavedrom



Exercise



Lesson Overview

Shift Register

Wavedrom

LED Walker

Wavedrom

The Need

Case Statement

The Need

The addresses

Simulation

Finite State Machine

Simple

Mealy

Moore

One Process FSM

Two Process FSM

Which to use?

Formal Verification

Assertion

SymbiYosys

Integer Clock

Divider

▷ Exercise

Conclusion

Try out the tools

1. Recreate this waveform using Wavedrom
2. Simulate this design

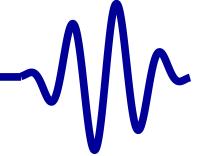
- **printf** o_led anytime it changes
- Look at the trace in gtkwave

Does it match our design goal?

Don't forget to slow it down!



Exercise



Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
▷ Exercise
Conclusion

Try out the tools

1. Recreate this waveform using Wavedrom
2. Simulate this design
3. Run SymbiYosys

Does this design pass?

If it passes, try **assert(led_index <= 4);**

Examine the resulting waveform



Exercise



Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
▷ Exercise
Conclusion

Try out the tools

1. Recreate this waveform using Wavedrom
2. Simulate this design
3. Run SymbiYosys

Does this design pass?

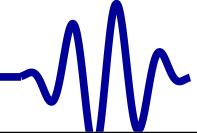
If it passes, try **assert**(led_index <= 4);

Examine the resulting waveform

Let's do this one together



Running Verilator

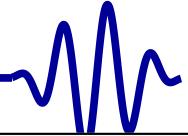


Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
▷ Exercise
Conclusion

```
% verilator -Wall -cc ledwalker.v
%Error: ledwalker.v:61: Can't find definition
          of variable: o_leed
%Error: Exiting due to 1 error(s)
%Error: Command Failed /usr/bin/verilator_bin
          -Wall -cc ledwalker.v
%
```

- Oops, we misspelled o_led in our case statement
- We also forgot to start our file with '**default_nettype** none'
- Once fixed, we pass the Verilator check

```
% verilator -Wall -cc ledwalker.v
%
```

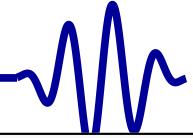


Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
▷ Exercise
Conclusion

```
% sby -f ledwalker.sby
```

```
/ex-03-walker$ sby -f ledwalker.sby
SBY 21:11:51 [ledwalker] Removing direcory 'ledwalker'.
SBY 21:11:51 [ledwalker] Copy 'ledwalker.v' to 'ledwalker/src/ledwalker.v'.
SBY 21:11:51 [ledwalker] engine_0: smtbmc
SBY 21:11:51 [ledwalker] base: starting process "cd ledwalker/src; yosys -ql ../model/design.log ../model/design.ys"
SBY 21:11:51 [ledwalker] base: ledwalker.v:71: ERROR: Identifier `led state' is implicitly declared and `default_nettpe is set to none.
SBY 21:11:51 [ledwalker] base: finished (returncode=1)
SBY 21:11:51 [ledwalker] base: job failed. ERROR.
SBY 21:11:51 [ledwalker] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)
SBY 21:11:51 [ledwalker] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00 (0)
SBY 21:11:51 [ledwalker] DONE (ERROR, rc=16)
/ex-03-walker$
```

- Another syntax error, mislabeled led_index as led_state
- Let's try again



Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
▷ Exercise
Conclusion

```
% sby -f ledwalker.sby
```

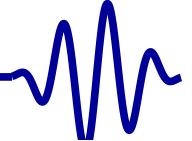
```
SBY 21:14:15 [ledwalker] engine_0.induction: ## 0:00:00 Trying induction in step 5..  
SBY 21:14:15 [ledwalker] engine_0.induction: ## 0:00:00 Temporal induction successful.  
SBY 21:14:15 [ledwalker] engine_0.induction: ## 0:00:00 Status: PASSED  
SBY 21:14:15 [ledwalker] engine_0.induction: finished (returncode=0)  
SBY 21:14:15 [ledwalker] engine_0: Status returned by engine for induction: PASS  
SBY 21:14:15 [ledwalker] engine_0.basecase: ## 0:00:00 Writing trace to constraints file: engine_0/trace.smvc  
SBY 21:14:15 [ledwalker] engine_0.basecase: ## 0:00:00 Status: FAILED (!)  
SBY 21:14:15 [ledwalker] engine_0.basecase: finished (returncode=-1)  
SBY 21:14:15 [ledwalker] engine_0: Status returned by engine for basecase: FAIL  
SBY 21:14:15 [ledwalker] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)  
SBY 21:14:15 [ledwalker] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00 (0)  
SBY 21:14:15 [ledwalker] summary: engine_0 (smtbmc) returned PASS for induction  
SBY 21:14:15 [ledwalker] summary: engine_0 (smtbmc) returned FAIL for basecase  
SBY 21:14:15 [ledwalker] summary: counterexample trace: ledwalker/engine_0/trace.vcd  
SBY 21:14:15 [ledwalker] DONE (FAIL, rc=2)
```

/ex-03-walker\$

It failed, but how? Need to scroll up for the details



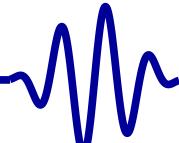
Running SymbiYosys



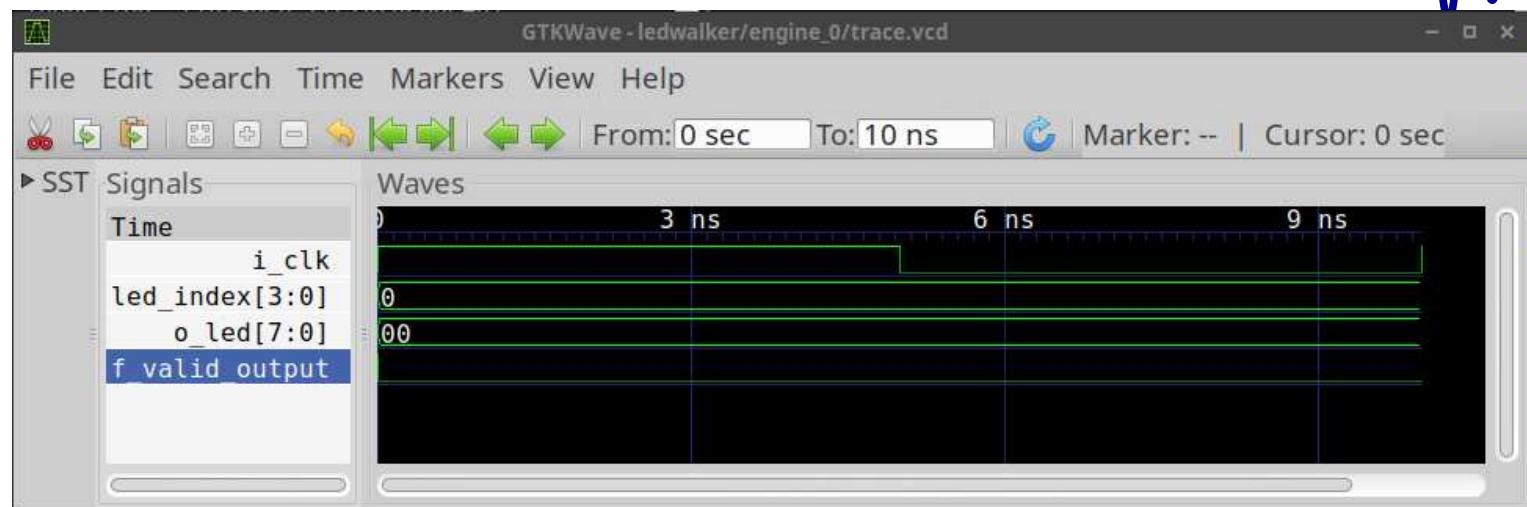
Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
▷ Exercise
Conclusion

```
trace_induct.smtc model/design_smt2.smt2"
SBY 21:14:15 [ledwalker] engine_0.basecase: ## 0:00:00 Solver: yices
SBY 21:14:15 [ledwalker] engine_0.induction: ## 0:00:00 Solver: yices
SBY 21:14:15 [ledwalker] engine_0.basecase: ## 0:00:00 Checking assumptions i
n step 0..
SBY 21:14:15 [ledwalker] engine_0.basecase: ## 0:00:00 Checking assertions in
step 0..
SBY 21:14:15 [ledwalker] engine_0.induction: ## 0:00:00 Trying induction in s
tep 20..
SBY 21:14:15 [ledwalker] engine_0.basecase: ## 0:00:00 BMC failed! ←
SBY 21:14:15 [ledwalker] engine_0.basecase: ## 0:00:00 Assert failed in ledwa
lker: ledwalker.v:96
SBY 21:14:15 [ledwalker] engine_0.basecase: ## 0:00:00 Writing trace to VCD f
ile: engine_0/trace.vcd
SBY 21:14:15 [ledwalker] engine_0.induction: ## 0:00:00 Trying induction in s
tep 19..
```

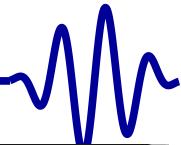
- Fail in line 96
- Trace file in ledwalker/engine_0/trace.vcd
- Open this in GTKWave, compare to line 96



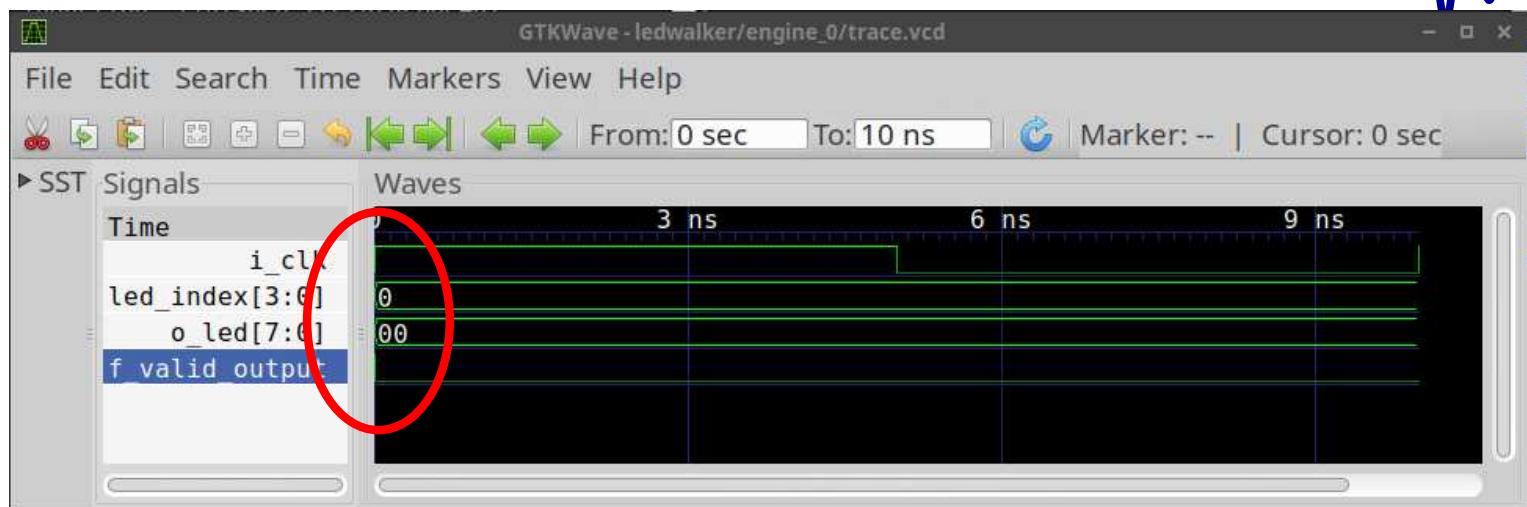
Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
▷ Exercise
Conclusion



- See the bug?



Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
▷ Exercise
Conclusion



- See the bug? `o_led` starts at 8'h00
- We never initialized `o_led` to a valid value
- **initial** `o_led = 8'h01`; fixes this

Running SymbiYosys



Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
▷ Exercise
Conclusion

```
SBY 21:21:37 [ledwalker] engine_0.basecase: ## 0:00:00 Checking assertions in
step 14..
SBY 21:21:37 [ledwalker] engine_0.basecase: ## 0:00:00 BMC failed!
SBY 21:21:37 [ledwalker] engine_0.basecase: ## 0:00:00 Assert failed in ledwa
lker: ledwalker.v:72
SBY 21:21:37 [ledwalker] engine_0.basecase: ## 0:00:00 Writing trace to VCD f
ile: engine_0/trace.vcd
SBY 21:21:37 [ledwalker] engine_0.basecase: ## 0:00:00 Writing trace to Veril
og testbench: engine_0/trace_tb.v
SBY 21:21:37 [ledwalker] engine_0.basecase: ## 0:00:00 Writing trace to const
raints file: engine_0/trace.smtc
SBY 21:21:37 [ledwalker] engine_0.basecase: ## 0:00:00 Status: FAILED (!)
SBY 21:21:37 [ledwalker] engine_0.basecase: finished (returncode=1)
SBY 21:21:37 [ledwalker] engine_0: Status returned by engine for basecase: FAIL
SBY 21:21:37 [ledwalker] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (
0)
SBY 21:21:37 [ledwalker] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00
(0)
SBY 21:21:37 [ledwalker] summary: engine_0 (smtbmc) returned PASS for induction
SBY 21:21:37 [ledwalker] summary: engine_0 (smtbmc) returned FAIL for basecase
SBY 21:21:37 [ledwalker] summary: counterexample trace: ledwalker/engine_0/trace
.vcd
SBY 21:21:37 [ledwalker] DONE (FAIL) rc=2
```

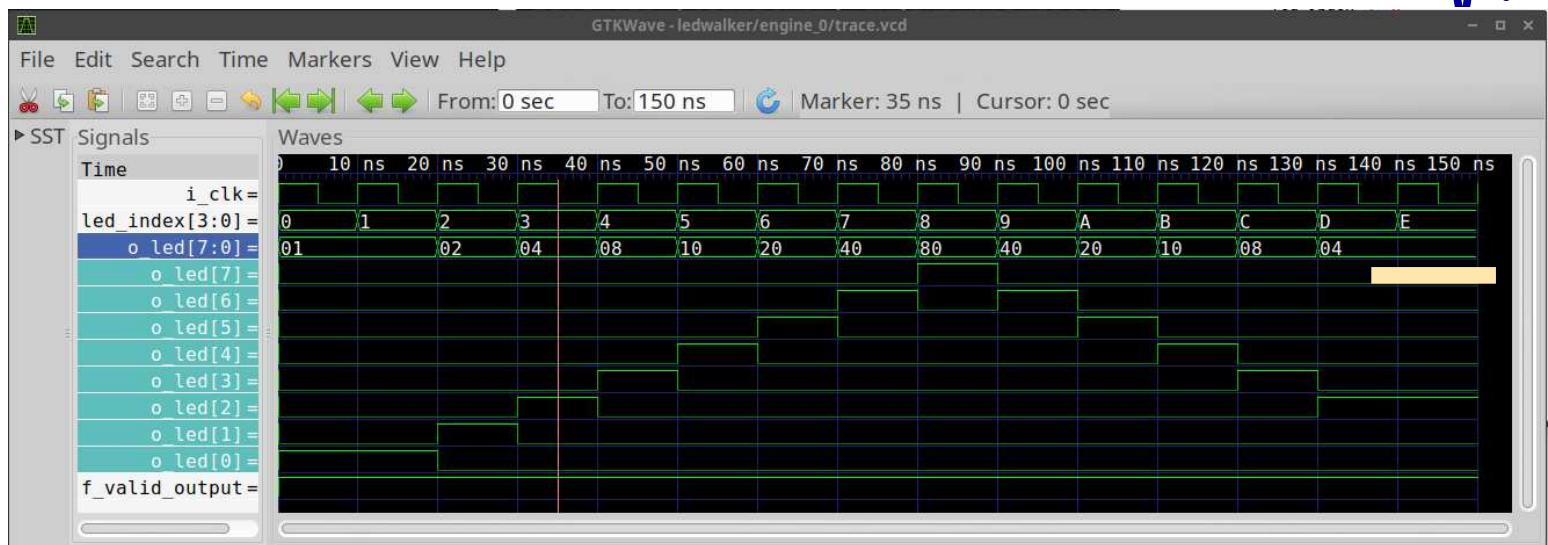
- Same trace file name
- Assertion failed in line 72



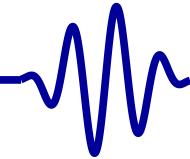
Running SymbiYosys



Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
▷ Exercise
Conclusion



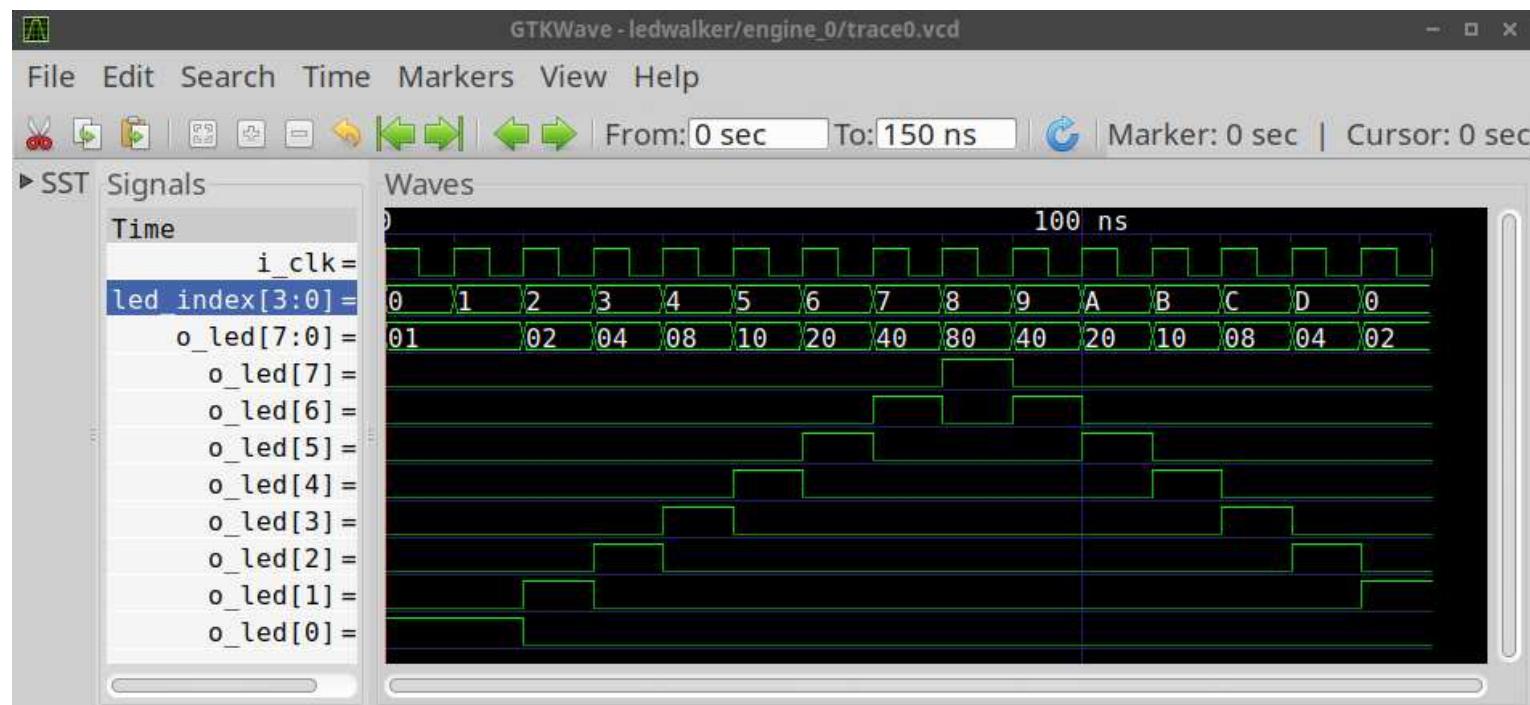
- if (led_index > 4'd12) in line 39 fixes this



Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
▷ Exercise
Conclusion

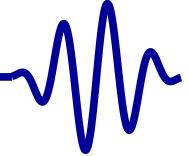
Let's add a quick cover property

```
always @(*)  
    cover((led_index = 0)&&(o_led == 4'h2));
```





Exercise

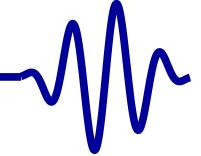


Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
▷ Exercise
Conclusion

Your turn! Use the tools to modify the design

1. Recreate this waveform using Wavedrom
2. Simulate this design
3. Run SymbiYosys
4. Run your device's Synthesis tool
 - Make sure your design . . .
 - Passes a timing check
 - Fits within your device
5. Now repeat with the clock divider

Bonus



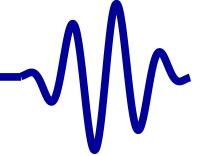
Bonus: If you have hardware and more than one LED

- Adjust this design for the number of LEDs you have
- Implement this on your hardware

Does it work?

Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
▷ Exercise
Conclusion

Conclusion



Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock
Divider
Exercise
▷ Conclusion

What did we learn this lesson?

- What a Finite State Machine (FSM) is
- Why FSM's are necessary
- Verilog **case** statement
- Verilog cascaded **if**
- Formal **assert** statement
- How to run SymbiYosys
- How to run slow down an FSM
- Verilog is fun!



Gisselquist
Technology, LLC

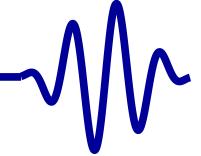
4. Pipeline Control

Daniel E. Gisselquist, Ph.D.





Lesson Overview



▷ Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

Wishbone Bus

Simulation

Unused Logic

Sim Exercise

Past Operator

Formal Verification

SymbiYosys Tasks

Exercise

Bonus

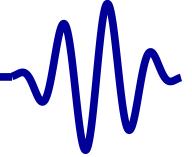
Conclusion

Objectives

- State diagrams
- Pipeline control structures
- Minimal peripherals
- Simulating Wishbone
- **\$past()** operator
- Verifying Wishbone



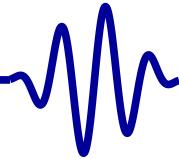
LED Walker



- Lesson Overview
- ▷ LED Walker
- Diagrams
- Pipeline
- Bus
- Wishbone Bus
- Simulation
- Unused Logic
- Sim Exercise
- Past Operator
- Formal Verification
- SymbiYosys Tasks
- Exercise
- Bonus
- Conclusion

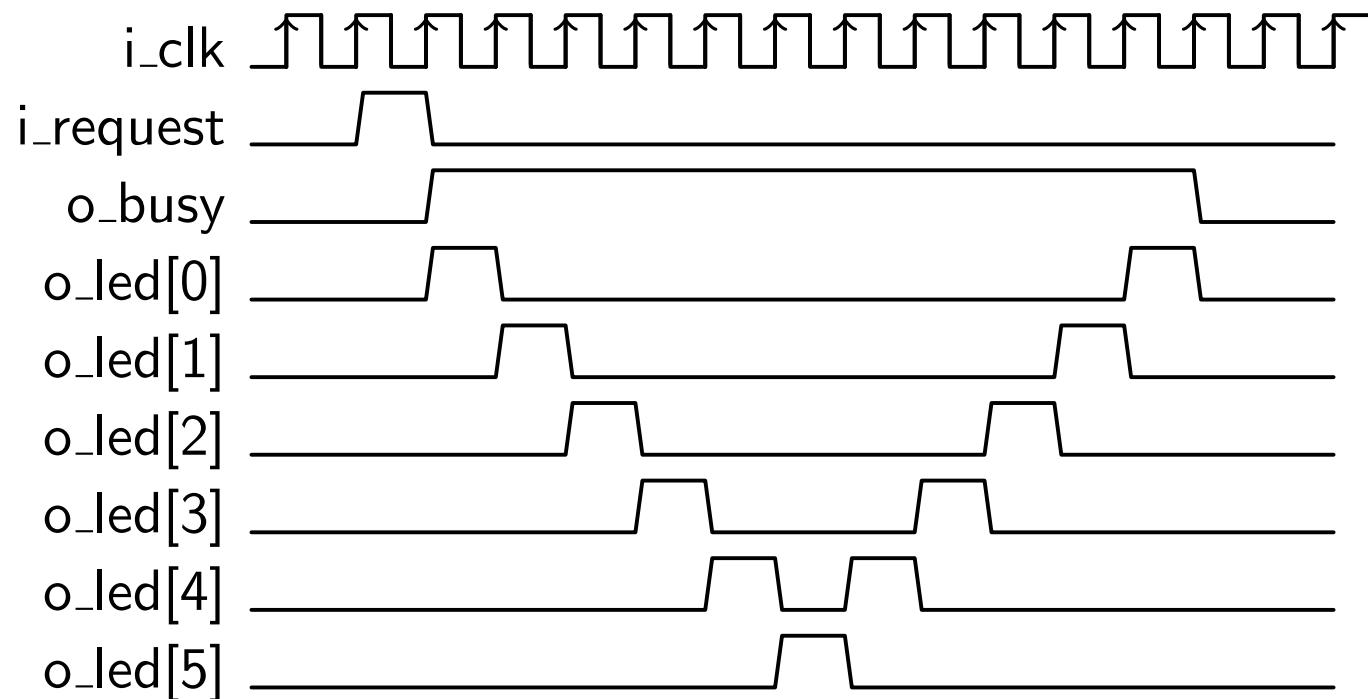
Let's make our LED's walk on command

- Bus requests
- State Diagram

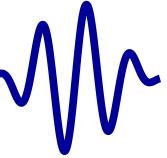


Lesson Overview
▷ LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

Let's adjust our LED sequence to require a request

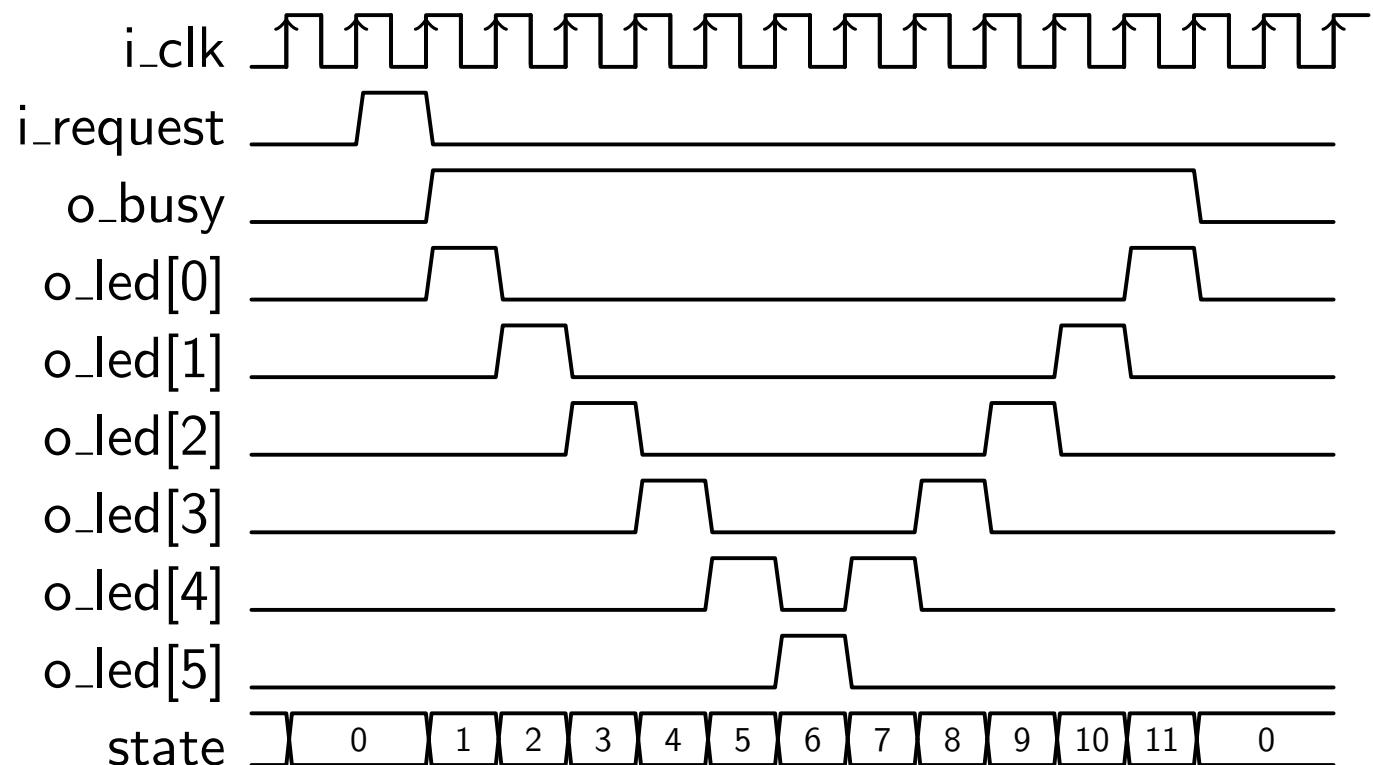


- Our goal will be to create a design with these outputs
- If successful, you'll see this in GTKwave

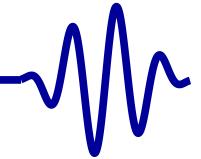


Lesson Overview
▷ LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

We'll add state ID's to this diagram



- Our goal will be to create a design with these outputs
- If successful, you'll see this in GTKwave

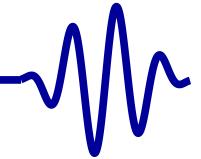


The key to this design is the idle state

- The design waits in state 0 for an `i_request`
- Only responds when it isn't busy

```
initial state = 0;
always @(posedge i_clk)
if ((i_request)&&(!o_busy))
    state <= 4'h1;
else if (state >= 4'hB)
    state <= 4'h0;
else if (state != 0)
    state <= state + 1'b1;

assign o_busy = (state != 0);
```



Lesson Overview

LED Walker

▷ Diagrams

Pipeline

Bus

Wishbone Bus

Simulation

Unused Logic

Sim Exercise

Past Operator

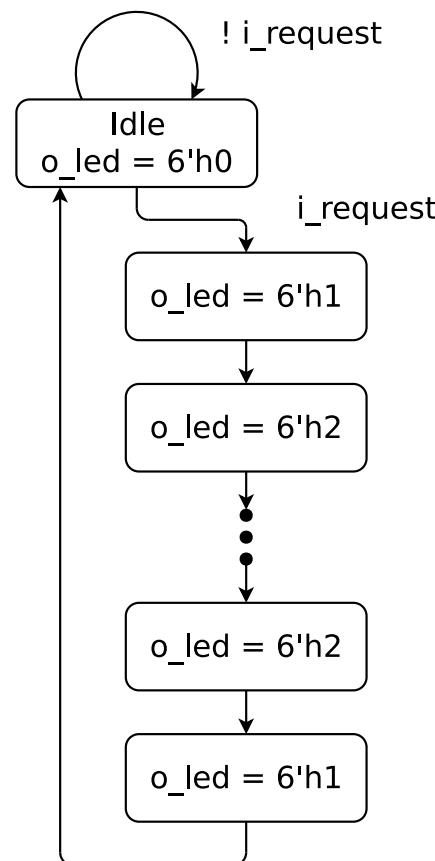
Formal Verification

SymbiYosys Tasks

Exercise

Bonus

Conclusion



□ States

- Shown as named bubbles
- Moore FSM: states include outputs
This FSM is a Moore FSM

□ Transitions

- Arrows between states
- May contain transition criteria
- Mealy FSM: transitions include outputs

Outputs



Lesson Overview
LED Walker
▷ Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

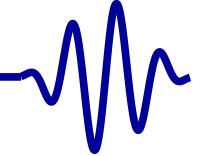
We can use a **case** statement for our outputs

```
always @(posedge i_clk)
  case(state)
    4'h1: o_led <= 6'b00_0001;
    4'h2: o_led <= 6'b00_0010;
    4'h3: o_led <= 6'b00_0100;
    4'h4: o_led <= 6'b00_1000;
    4'h5: o_led <= 6'b01_0000;
    4'h6: o_led <= 6'b10_0000;
    4'h7: o_led <= 6'b01_0000;
    // ...
    4'ha: o_led <= 6'b00_0010;
    4'hb: o_led <= 6'b00_0001;
    default: o_led <= 6'b00_0000;
  endcase
```

Or can we? Does this work?



Pipeline Strategies



Lesson Overview
LED Walker
Diagrams
▷ Pipeline
Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

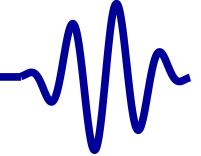
Several approaches to pipeline logic

1. Apply the logic on every clock

```
// From the PPS-II implementation
always @(posedge i_clk)
    counter <= counter + INCREMENT;
```



Pipeline Strategies



Lesson Overview
LED Walker
Diagrams
▷ Pipeline Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

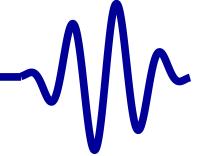
Several approaches to pipeline logic

1. Apply the logic on every clock
2. Wait for a clock enable (CE) signal

```
// From the Integer Clock Divider
always @(posedge i_clk)
if (stb) // this would be the CE signal
begin
    if (led_index >= 4'd13)
        led_index <= 0;
    else
        led_index <= led_index + 1'b1;
end
```



Pipeline Strategies



Lesson Overview
LED Walker
Diagrams
▷ Pipeline Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

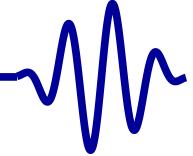
Several approaches to pipeline logic

1. Apply the logic on every clock
2. Wait for a clock enable (CE) signal
3. Move on a request, but only when not busy

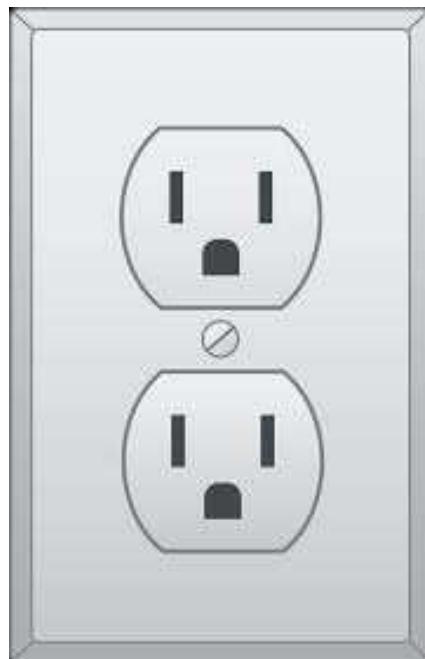
```
// Today's logic: Wait for the request
always @(posedge i_clk)
if ((i_request)&&(!o_busy))
    state <= 4'h1;
else if (state >= 4'hB)
    state <= 4'h0;
else if (state != 0)
    state <= state + 1'b1;
```

Above: A mix of pipeline and state machine logic

This is fairly common

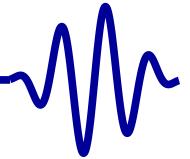


Interface standards simplify plugging things in

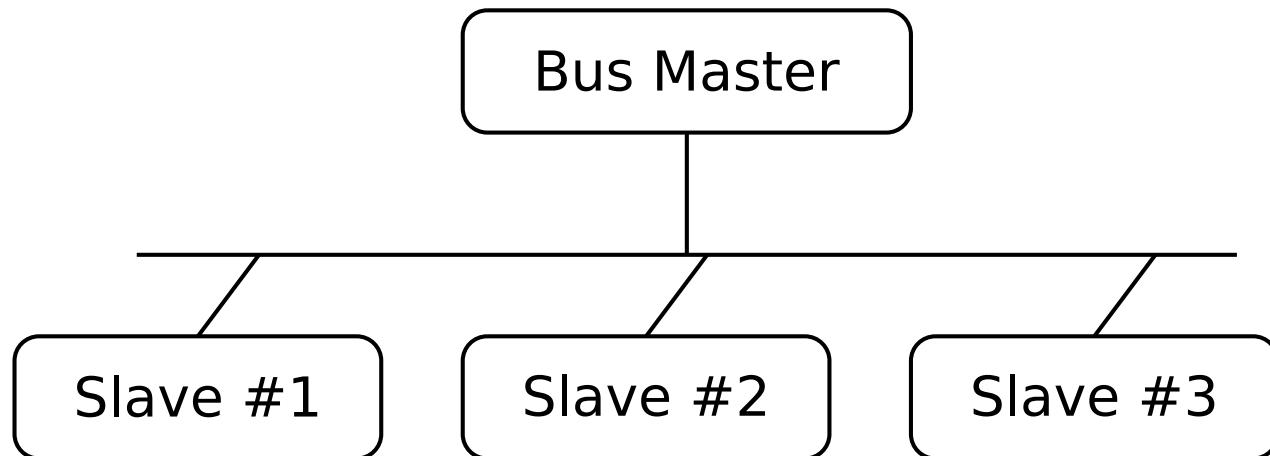


A bus interface can be standardized

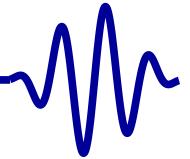
- A master makes requests
A slave responds
- Read request
 - Contains an address
 - Slave responds with a value
- Write request
 - Contains an address
 - Contains a value
 - Slave responds with an acknowledgment



Lesson Overview
LED Walker
Diagrams
Pipeline
▷ Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion



- Every bus has a master
- A Bus may have many slaves
Slaves are differentiated by their address
- All connected via an *interconnect*
- A slave on one bus may be a master on another



Lesson Overview

LED Walker

Diagrams

Pipeline

▷ Bus

Wishbone Bus

Simulation

Unused Logic

Sim Exercise

Past Operator

Formal Verification

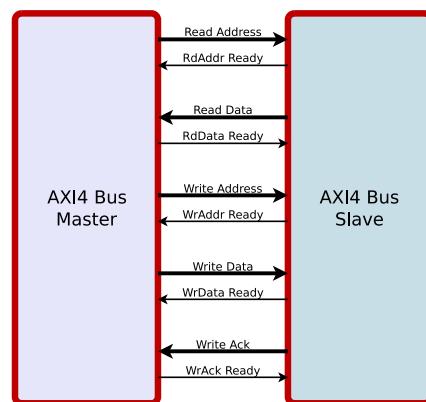
SymbiYosys Tasks

Exercise

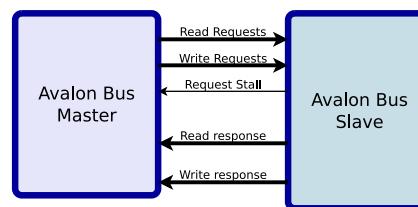
Bonus

Conclusion

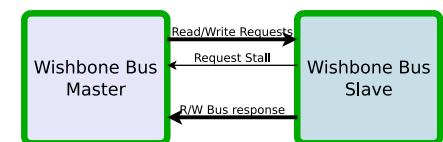
There are many bus standards



AXI



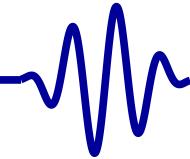
Avalon



Wishbone

I like Wishbone for its simplicity

- Only one request channel
AXI has three, Avalon has two
- Only the request channel can stall
- Acknowledgements are simple



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

▷ Wishbone Bus

Simulation

Unused Logic

Sim Exercise

Past Operator

Formal Verification

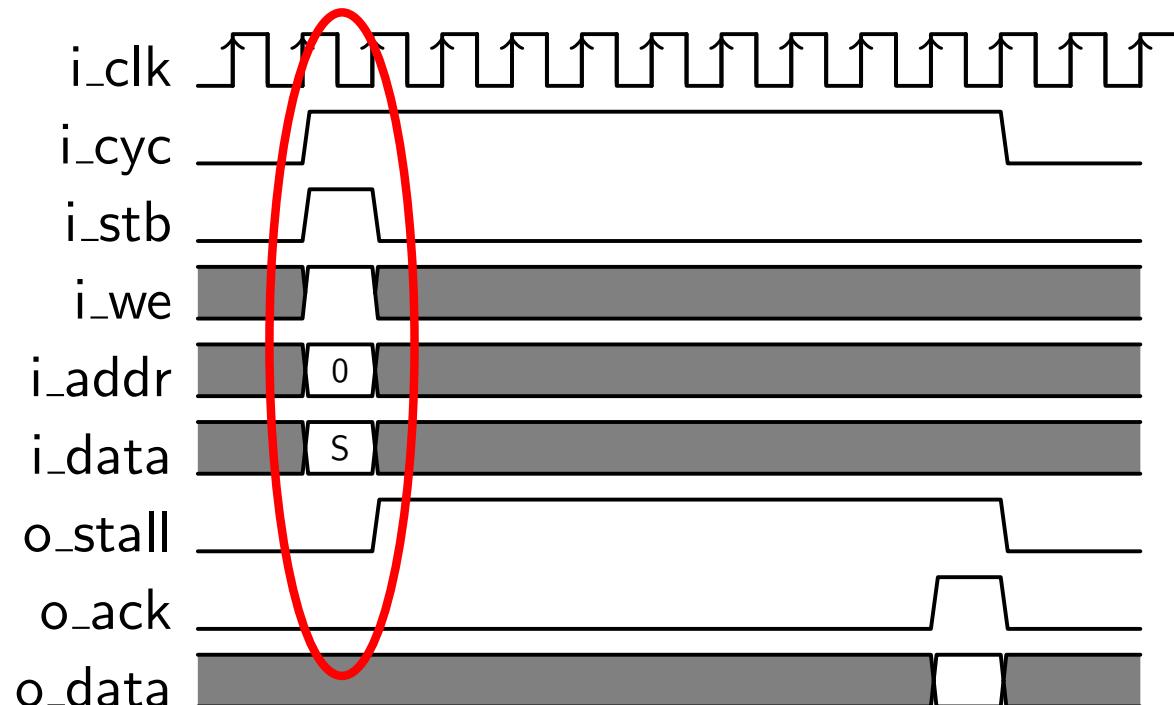
SymbiYosys Tasks

Exercise

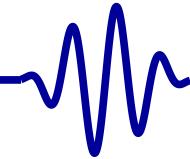
Bonus

Conclusion

I use Wishbone B4, pipelined mode exclusively



- A request takes place any time $(i_stb) \&\& (!o_stall)$
Just like our $(i_request) \&\& (!o_busy)$
- The request details are found in i_we , i_addr , and i_data
- These wires are don't care if i_stb isn't true



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

▷ Wishbone Bus

Simulation

Unused Logic

Sim Exercise

Past Operator

Formal Verification

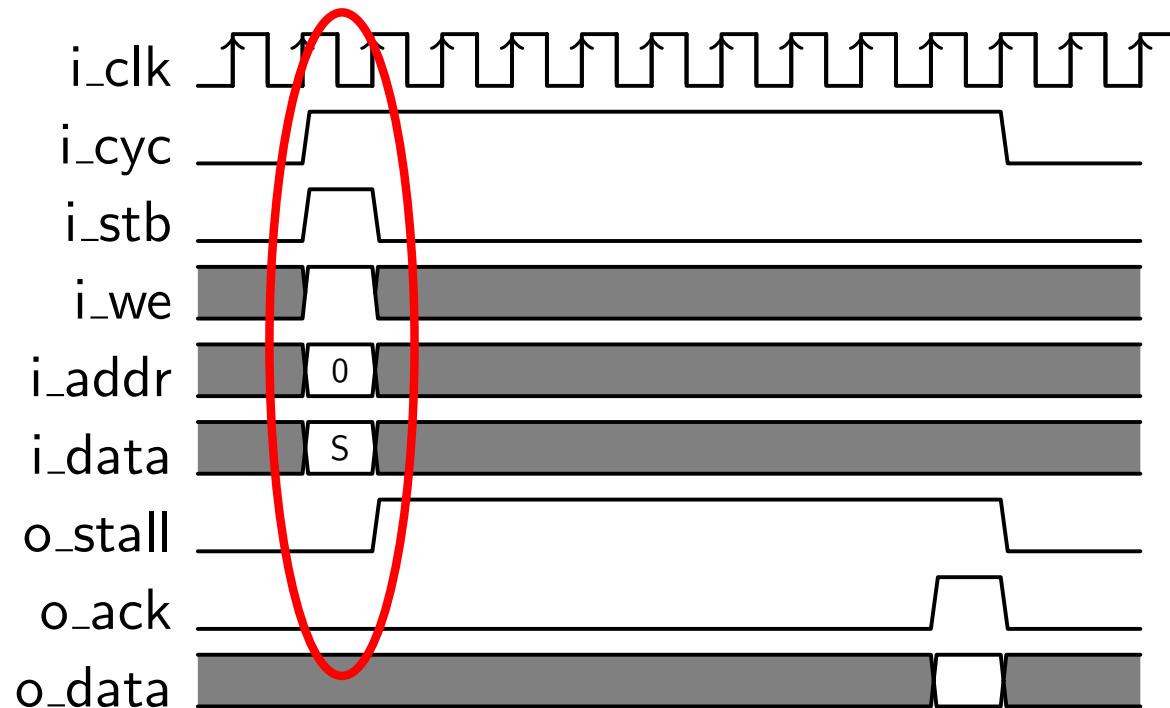
SymbiYosys Tasks

Exercise

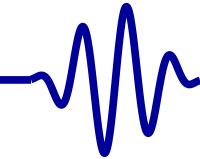
Bonus

Conclusion

I use Wishbone B4, pipelined mode exclusively



- If i_we, this is a write request
- A write request writes i_data to address i_addr
- Read requests ignore i_data



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

▷ Wishbone Bus

Simulation

Unused Logic

Sim Exercise

Past Operator

Formal Verification

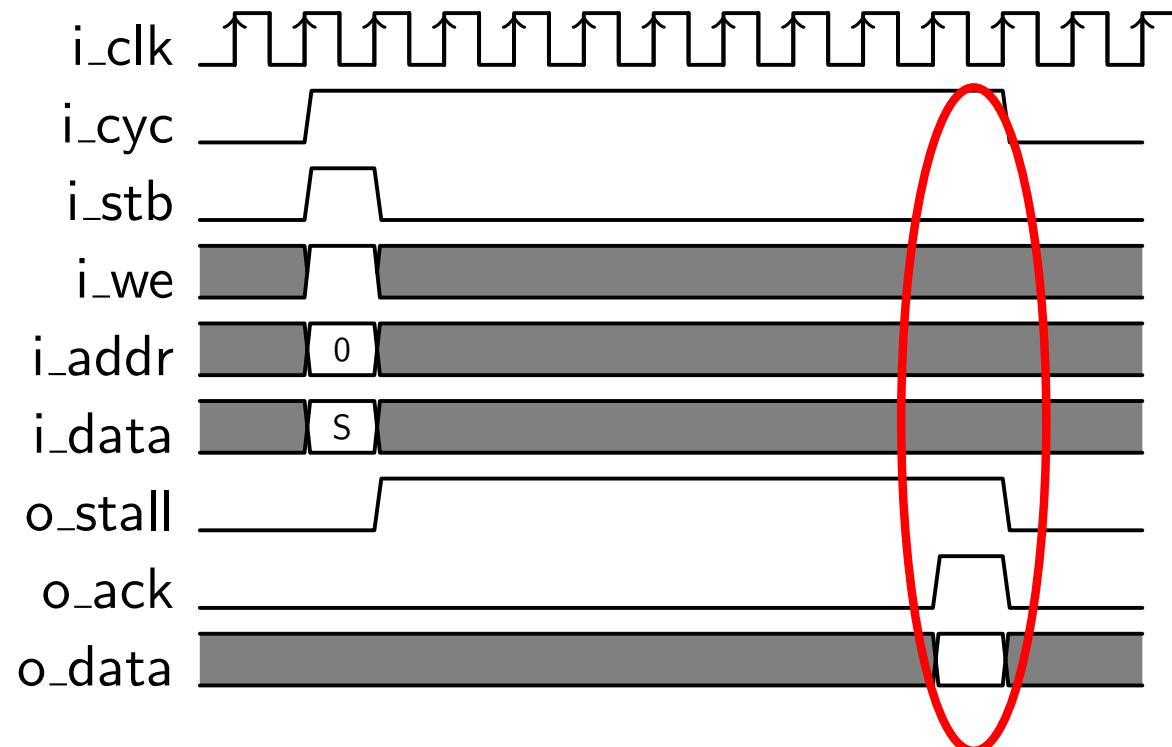
SymbiYosys Tasks

Exercise

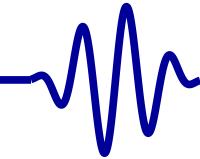
Bonus

Conclusion

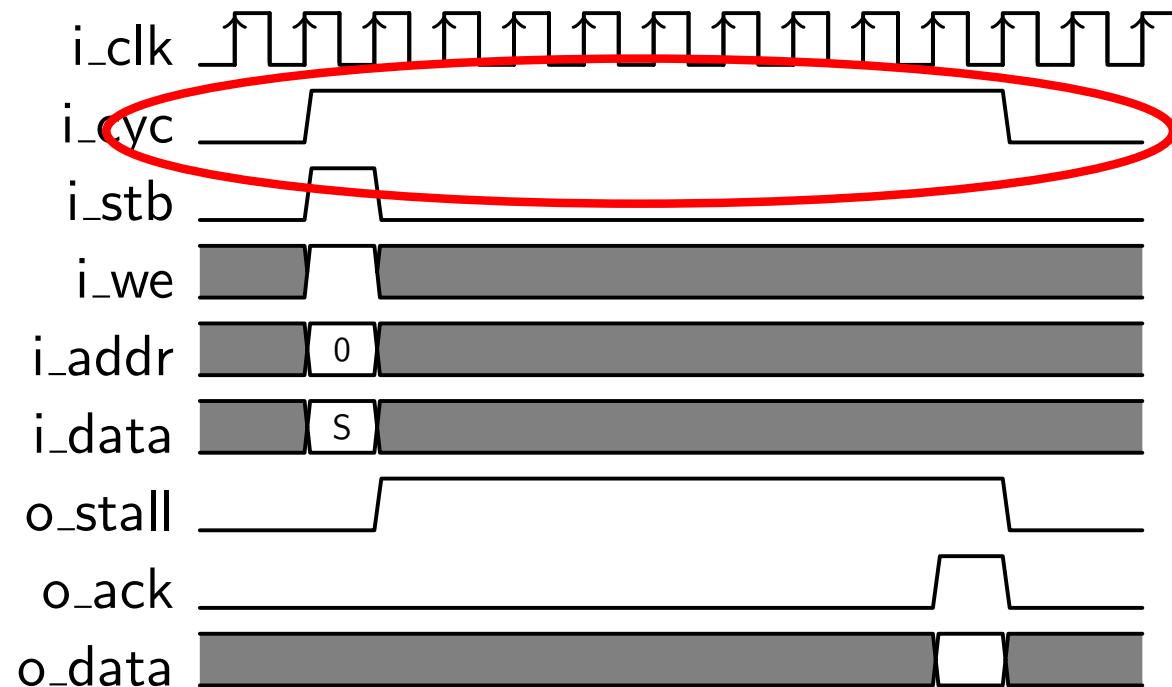
I use Wishbone B4, pipelined mode exclusively



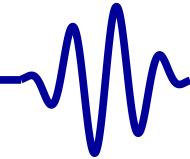
- The response is signaled when o_ack is true
- If this was a read request, o_data would have the result

[Lesson Overview](#)[LED Walker](#)[Diagrams](#)[Pipeline](#)[Bus](#)[▷ Wishbone Bus](#)[Simulation](#)[Unused Logic](#)[Sim Exercise](#)[Past Operator](#)[Formal Verification](#)[SymbiYosys Tasks](#)[Exercise](#)[Bonus](#)[Conclusion](#)

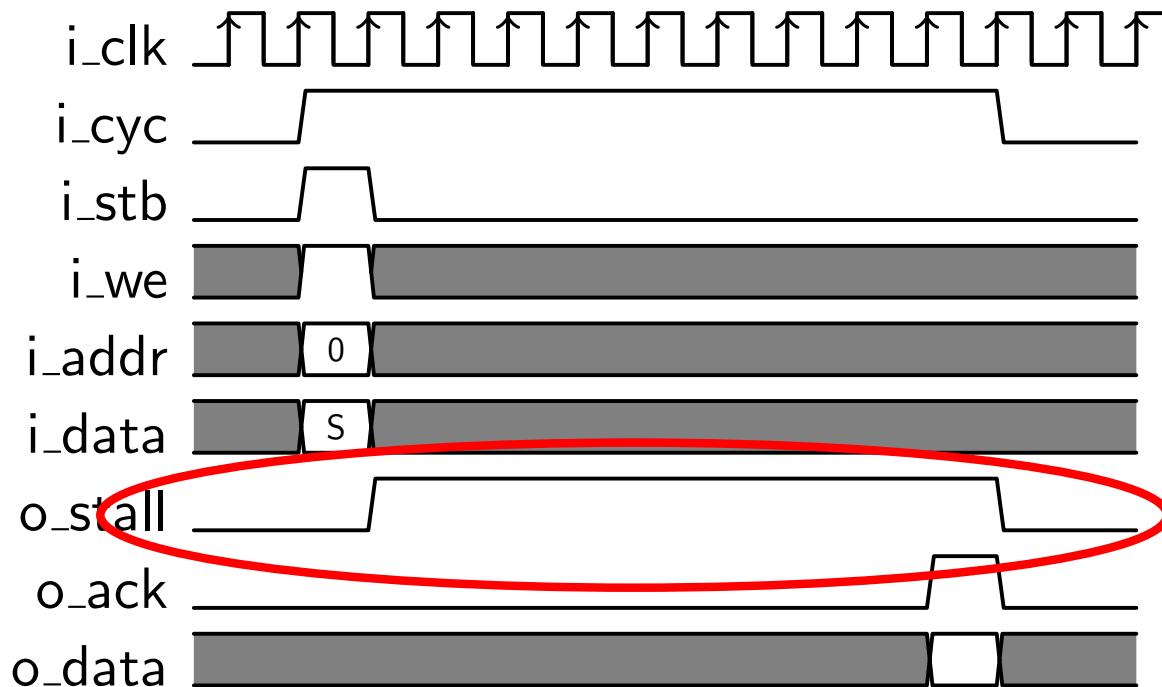
I use [Wishbone B4](#), pipelined mode exclusively



- *i_cyc* will be true from request to ack
- *i_stb* will never be true unless *i_cyc*



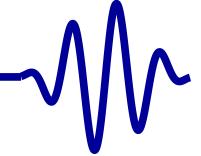
I use Wishbone B4, pipelined mode exclusively



- A slave must respond to every request
- Multiple requests can be made before the slave responds
- This is controlled by the o_stall signal



Wishbone Bus



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

▷ Wishbone Bus

Simulation

Unused Logic

Sim Exercise

Past Operator

Formal Verification

SymbiYosys Tasks

Exercise

Bonus

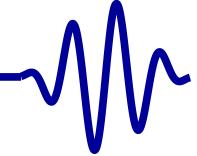
Conclusion

Let's Wishbone enable our core

- We'll start the LED cycling on a write
- Writes will stall if the LED's are busy
- Return our state on a read
- We'll also acknowledge all requests immediately



Wishbone Bus



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

▷ Wishbone Bus

Simulation

Unused Logic

Sim Exercise

Past Operator

Formal Verification

SymbiYosys Tasks

Exercise

Bonus

Conclusion

- We'll immediately acknowledge any transaction

```
initial o_ack = 1'b0;  
always @ (posedge i_clk)  
    o_ack <= (i_stb) && (!o_stall);
```

- Stall if we're busy and another cycle is requested

```
assign o_stall = (busy) && (i_we);
```

- Return state upon any read

```
assign o_data = { 28'h0, state };
```

GT Simulation



Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
▷ Simulation
Unused Logic
Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

It helps to be able to communicate with your wishbone slave during simulation

- Makes simulations easier
- Transaction scripting makes more sense
- Just need to implement two functions
 - One to read from the bus

```
unsigned wb_read(unsigned a);
```

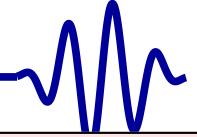
- One to write to the bus

```
void wb_write(unsigned a, unsigned v);
```

- We'll come back later and create high-throughput versions of these



Sim Read



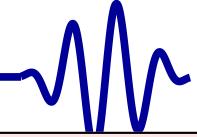
Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
▷ Simulation
Unused Logic
Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

```
unsigned wb_read(unsigned a) {
    tb->i_cyc = tb->i_stb = 1;
    tb->i_we = 0;
    tb->i_addr= a;

    // Make the read request
    while(tb->o_stall)
        tick(tb);
    tick(tb);
    tb->i_stb = 0;
    // Wait for the ACK
    while(!tb->o_ack)
        tick(tb);
    // Idle the bus, and read the response
    tb->i_cyc = 0;
    return tb->o_data;
}
```



Sim Write

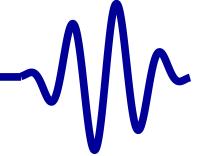


Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
▷ Simulation
Unused Logic
Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

```
void wb_write(unsigned a, unsigned v) {
    tb->i_cyc = tb->i_stb = 1;
    tb->i_we = 1;
    tb->i_addr= a;
    tb->i_data= v;
    // Make the write request
    while(tb->o_stall)
        tick(tb);
    tick(tb);
    tb->i_stb = 0;
    // Wait for the acknowledgement
    while(!tb->o_ack)
        tick(tb);
    // Idle the bus and return
    tb->i_cyc = 0;
}
```



Run Twice



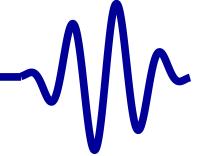
Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
▷ Simulation
Unused Logic
Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

This makes building the sim easy!

- Let's tell our LED's to cycle twice

```
int main(int argc, char **argv) {
    // Setup Verilator (same as before)
    // Read from the current state
    printf("Initial\u005fstate\u005fis:\u005c0x%02x\n",
           wb_read(0));
    for(int cycle=0; cycle<2; cycle++) {
        // Wait five clocks
        for(int i=0; i<5; i++)
            tick();

        // Start the LEDs cycling
        wb_write(0,0);
        tick();
        // ... (next page)
```



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

Wishbone Bus

▷ Simulation

Unused Logic

Sim Exercise

Past Operator

Formal Verification

SymbiYosys Tasks

Exercise

Bonus

Conclusion

This makes building the sim easy!

- Here's the other half

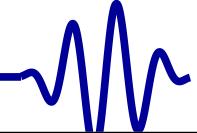
```
// ... (last page)
while((state = wb_read(0))!=0) {
    if ((state != last_state)
        ||(tb->o_led != last_led)) {
        printf(// something useful
            );
    } tick();

    last_state = state;
    last_led = tb->o_led;
}
```

The full example code is available on line



Unused Logic



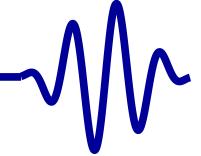
Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
▷ Unused Logic
Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

```
% verilator --trace -Wall -cc reqwalker.v
%Warning-UNUSED: reqwalker.v:37:
    Signal is not used: i_cyc
%Warning-UNUSED: reqwalker.v:38:
    Signal is not used: i_addr
%Warning-UNUSED: reqwalker.v:39:
    Signal is not used: i_data
%Error: Exiting due to 3 warning(s)
%Error: Command Failed /usr/bin/verilator_bin
        --trace -Wall -cc reqwalker.v
%
```

What happened?



Unused Logic



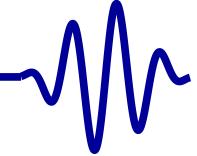
Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
▷ Unused Logic
Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

What happened?

- The -Wall flag to Verilator looks for all kinds things you might not have meant
- It turns warnings into errors
- It found logic we weren't using: `i_cyc`, `i_addr`, and `i_data`
 - These are standard bus interface wires
 - I often include them, even if not used, to keep the interface standardized
- So how do get our design to work?



Unused Logic



Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
▷ Unused Logic
Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

Getting Verilator to ignore unused logic

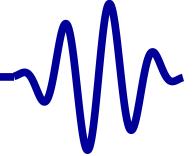
- Use the *// Verilator lint_off UNUSED* command

```
// Verilator lint_off UNUSED
wire unused;
assign unused = &{ 1'b0, i_cyc, i_addr,
                     i_data };
// Verilator lint_on UNUSED
```

- Verilator will now no longer check if unused is used or not



Sim Exercise



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

Wishbone Bus

Simulation

Unused Logic

▷ Sim Exercise

Past Operator

Formal Verification

SymbiYosys Tasks

Exercise

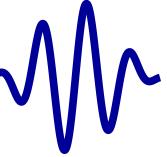
Bonus

Conclusion

Build and run the demo

- Examine the trace
- Examine the output

Does it work like you expected?



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

Wishbone Bus

Simulation

Unused Logic

▷ Sim Exercise

Past Operator

Formal Verification

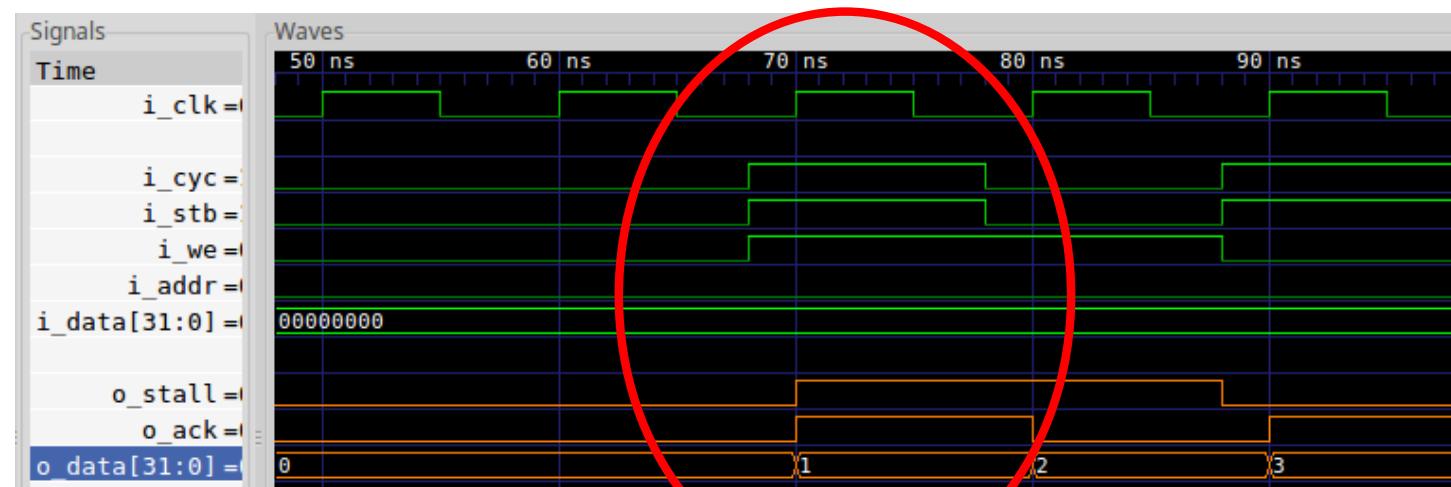
SymbiYosys Tasks

Exercise

Bonus

Conclusion

Look at the trace. Can you explain this?

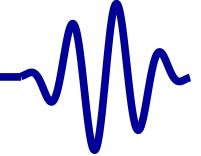


Our inputs aren't clock synchronous!

- Normally, all logic changes on the posedge of i_clk
- i_cyc, i_stb, i_we are changing before the clock



Trace bias



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

Wishbone Bus

Simulation

Unused Logic

▷ Sim Exercise

Past Operator

Formal Verification

SymbiYosys Tasks

Exercise

Bonus

Conclusion

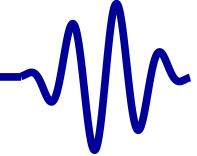
This is a consequence of our **trace()** function

- We set our input values, `i_cyc`, etc *before* calling `tick()`

```
void tick(void) {
    tickcount++;

    tb->eval(); // Adjusted inputs are
    if (tfp)     // recorded here
        tfp->dump(tickcount * 10 - 2);

    tb->i_clk = 1; // <--- posedge i_clk
    tb->eval();   // takes place here!
    if (tfp)
        tfp->dump(tickcount * 10);
    ...
}
```



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

Wishbone Bus

Simulation

Unused Logic

▷ Sim Exercise

Past Operator

Formal Verification

SymbiYosys Tasks

Exercise

Bonus

Conclusion

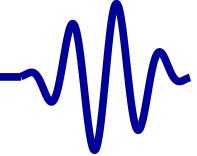
This is a consequence of our **trace()** function

- We set our input values, `i_cyc`, etc *before* calling `tick()`

```
void tick(void) {
    tickcount++;

    tb->eval(); // Adjusted inputs are
    if (tfp) // recorded here
        tfp->dump(tickcount * 10 - 2);

    tb->i_clk = 1; // <--- posedge i_clk
    tb->eval(); // takes place here!
    if (tfp)
        tfp->dump(tickcount * 10);
    // ...
}
```



This is a consequence of our **trace()** function

- We set our input values, `i_cyc`, etc *before* calling `tick()`

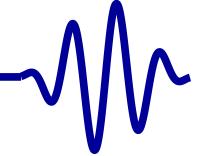
```
void tick(void) {
    tickcount++;

    tb->eval(); // Adjusted inputs are
    if (tfp)      // recorded here
        tfp->dump(tickcount * 10 - 2);
```

- The `tfp->dump(tickcount*10 - 2)` dumps the state of everything just before the positive edge of the clock
- This captures the changes made to `i_cyc`, `i_stb`, `i_we`, etc., in `wb_read()` and `wb_write()`
- The trace accurately reflected these changes taking place before the clock edge



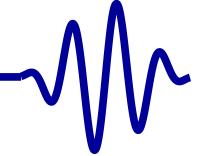
Trace bias



Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
Unused Logic
▷ Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

This is a consequence of our **trace()** function

- We set our input values, `i_cyc`, etc *before* calling `tick()`
- Had we done otherwise, combinatorial logic wouldn't have settled before **posedge i_clk**
- Worse, the trace wouldn't make any sense
- This way, things work. Logic matches the trace.
It just looks strange.



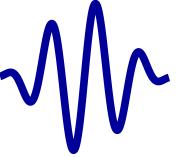
Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
Unused Logic
▷ Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

Is this an output you expected?

```
% ./reqwalker
Initial state is: 0x00
    10: State # 4 --0---
    12: State # 6 -----0-
    14: State # 8 -----0-
    16: State #10 --0---
    27: State # 4 --0---
    29: State # 6 -----0-
    31: State # 8 -----0-
    33: State #10 --0---

%
```

Let's look at the trace again!



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

Wishbone Bus

Simulation

Unused Logic

▷ Sim Exercise

Past Operator

Formal Verification

SymbiYosys Tasks

Exercise

Bonus

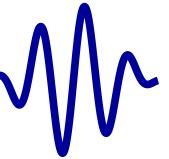
Conclusion

Look at the trace. Can you explain this?



- Why are we getting two acks in a row?
- We never created two adjacent requests!

Double ACKs



Look at the trace. Can you explain this?



- The stall line depends upon `i_we`
- Without a call to `tb->eval()`, it won't update!



Double ACKs



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

Wishbone Bus

Simulation

Unused Logic

▷ Sim Exercise

Past Operator

Formal Verification

SymbiYosys Tasks

Exercise

Bonus

Conclusion

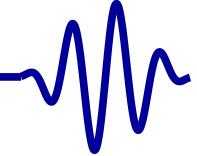
Remember how we defined o_stall?

```
assign o_stall = (busy)&&(i_we);
```

- **wb_write()** and **wb_read()** both adjust **i_we**
- ... without calling Verilator to give it a chance to update **o_stall** before referencing it!
- **o_stall** is still updated before the clock, but not until after we used it in **wb_write()** and **wb_read()**
- We can fix this by calling **tb->eval()** to get Verilator to adjust **o_stall**



Double ACKs



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

Wishbone Bus

Simulation

Unused Logic

▷ Sim Exercise

Past Operator

Formal Verification

SymbiYosys Tasks

Exercise

Bonus

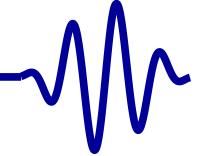
Conclusion

Need to call **tb->eval()**

- o_stall depends upon a Verilator input, i_we
 - Fixing this requires an extra call to **eval()**
 - I don't normally need to do this
- Both **wb_read()** and **wb_write()** need to be updated
- Example update to **wb_read()**:

```
unsigned wb_read(unsigned a) {
    tb->i_cyc = tb->i_stb = 1;
    tb->i_we = 0; tb->eval();
    tb->i_addr= a;
    // Make the request
    // ...
}
```

Exercise



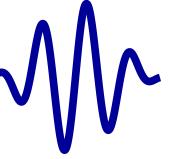
Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
Unused Logic
▷ Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

Rebuild and run again. Is this better?

```
% ./reqwalker
Initial state is: 0x00
    9: State # 3 -0-----
    11: State # 5 -----0--
    13: State # 7 -----0
    15: State # 9 ----0--
    17: State #11 -0-----
    27: State # 3 -0-----
    29: State # 5 -----0--
    31: State # 7 -----0
    33: State # 9 ----0--
    35: State #11 -0-----

%
```

But, why are we reading every other trace?



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

Wishbone Bus

Simulation

Unused Logic

▷ Sim Exercise

Past Operator

Formal Verification

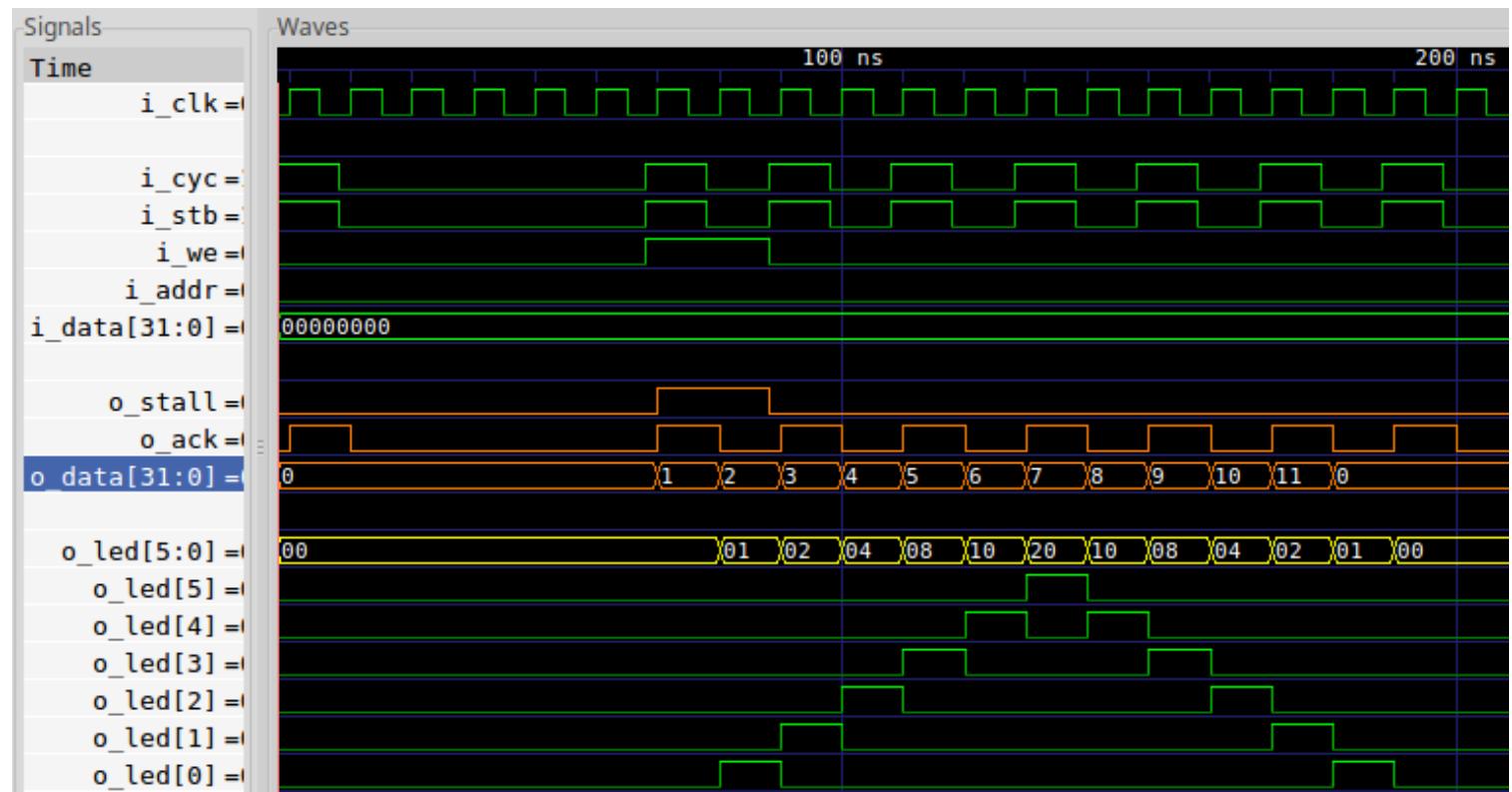
SymbiYosys Tasks

Exercise

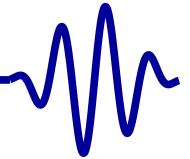
Bonus

Conclusion

Look at the ACK's



- Pattern: i_stb, o_ack repeats
- Lesson: The clock ticks twice per read



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

Wishbone Bus

Simulation

Unused Logic

► Sim Exercise

Past Operator

Formal Verification

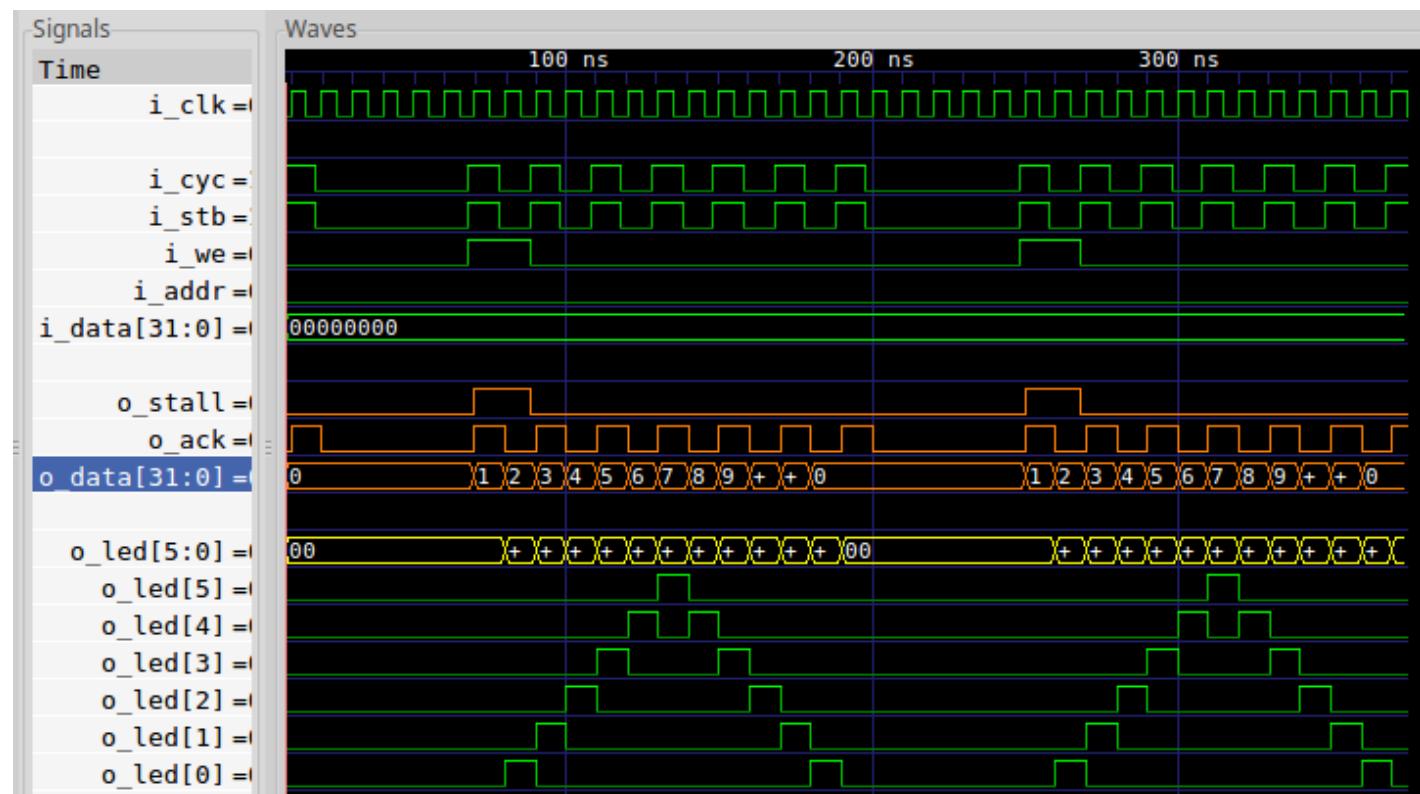
SymbiYosys Tasks

Exercise

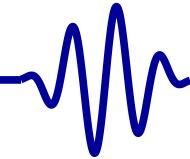
Bonus

Conclusion

Here's the full and final simulation



Here you can see both LED walks, as expected



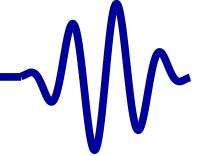
Pipeline logic needs to reason in passing time

- **\$past(X)** returns the value of X one clock ago
- **\$past(X,N)** returns the value of x N clocks ago
- Both require a clock

```
always @(posedge i_clk)
if ($past(c))
    assert(x == y);
```

- It's illegal to use **\$past(X)** without a clock

```
// This is an error: there's no clock
always @(*)
if ($past(c))
    assert(x);
```

[Lesson Overview](#)[LED Walker](#)[Diagrams](#)[Pipeline](#)[Bus](#)[Wishbone Bus](#)[Simulation](#)[Unused Logic](#)[Sim Exercise](#)[▷ Past Operator](#)[Formal Verification](#)[SymbiYosys Tasks](#)[Exercise](#)[Bonus](#)[Conclusion](#)

\$past(X) has one disadvantage

- On the initial clock, **\$past(X)** is undefined
 - Assertions referencing **\$past(X)** will always fail
 - Assumptions referencing **\$past(X)** will always succeed
- I guard against this with `f_past_valid`

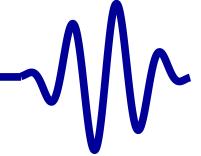
```
reg      f_past_valid;  
initial f_past_valid = 0;  
always @(posedge i_clk)  
    f_past_valid = 1'b1;
```

- To use, place `f_past_valid` in an **if** condition

```
always @(posedge i_clk)  
if ((f_past_valid)&&($past(some_condition)))  
    assert(this_must_then_be_true);
```



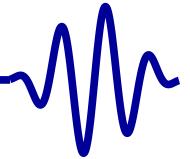
Formal Verification



Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
 Formal
 ▷ Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

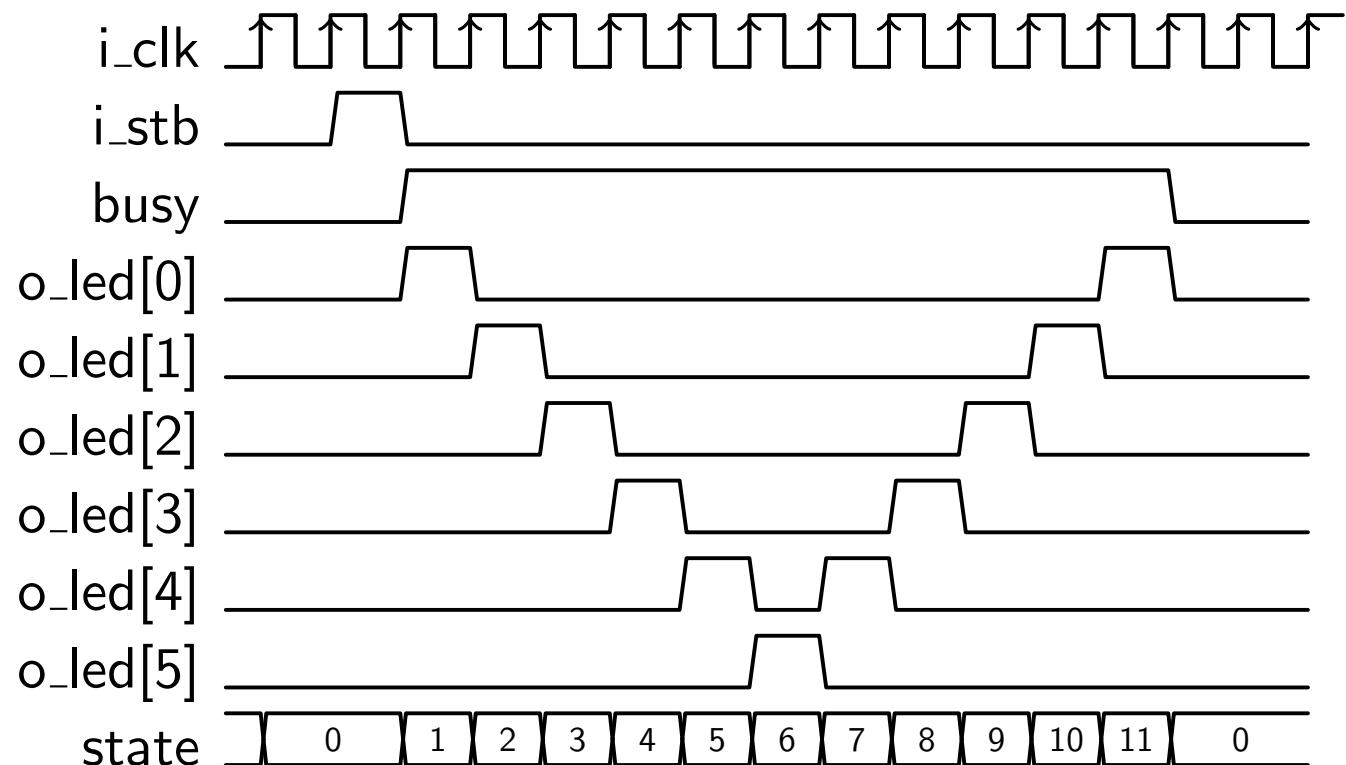
What properties might we use?

- **assume** properties of the inputs
- **assert** properties of local states and outputs



Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
 Formal
▷ Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

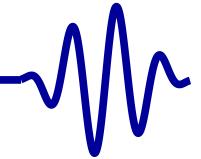
What properties might we use?



The goal waveform diagram should give you an idea



Formal Verification



Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
 Formal
▷ Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

What properties might we use?

- For our state machine

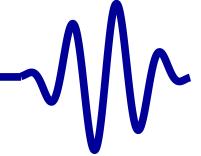
```
always @(*)
case(state)
    4'h0: assert(o_led == 0);
    4'h1: assert(o_led == 6'h1);
    4'h2: assert(o_led == 6'h2);
    //
    4'hb: assert(o_led == 6'h1);
endcase

always @(*)
    assert(busy != (state == 0));

always @(*)
    assert(state <= 4'hb);
```



Formal Verification



Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
 Formal
 ▷ Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

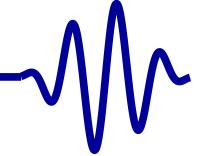
What properties might we use?

- For our state machine, using **\$past(X)**
- An accepted write should start our cycle

```
always @ (posedge i_clk)
if ((f_past_valid)&&($past(i_stb))
    &&($past(i_we))&&(!$past(o_stall)))
begin
    assert(state == 1);
    assert(busy);
end
```



Formal Verification



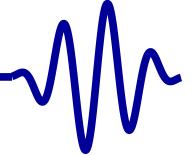
Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
 Formal
 ▷ Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

What properties might we use?

- During the cycle, the state should increment

```
always @ (posedge i_clk)
  if ((f_past_valid) && ($past(busy))
      && ($past(state < 4'hb)))
    assert (state == $past(state)+1);
```

Formal Verification



Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
 Formal
▷ Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

What properties might we use?

- For our bus interface?

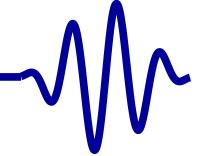
```
// Bus should be idle initially
initial assume(!i_cyc);

// i_stb is only allowed if i_cyc
always @(*)
if (!i_cyc)
    assume(!i_stb);

// When i_cyc goes high, so too does i_stb
always @(posedge i_clk)
if ((!$past(i_cyc))&&(i_cyc))
    assume(i_stb);
```



Formal Verification



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

Wishbone Bus

Simulation

Unused Logic

Sim Exercise

Past Operator

Formal

▷ Verification

SymbiYosys Tasks

Exercise

Bonus

Conclusion

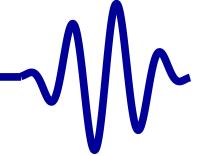
What properties might we use?

- For our bus interface?

```
always @(posedge i_clk)
if ((f_past_valid)&&($past(i_stb))
    &&($past(o_stall)))
begin
    // Request is stalled
    // It shouldn't change
    assume(i_stb);
    assume(i_we == $past(i_we));
    assume(i_addr == $past(i_addr));
    if (i_we)
        assume(i_data == $past(i_data));
end
```



Formal Verification



Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
 Formal
▷ Verification
SymbiYosys Tasks
Exercise
Bonus
Conclusion

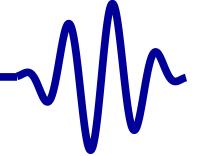
What properties might we use?

- For our bus interface?

```
always @(posedge i_clk)
if ((f_past_valid)&&($past(i_stb))
    &&(!$past(o_stall)))
    assert(o_ack);
```



Cover Property



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

Wishbone Bus

Simulation

Unused Logic

Sim Exercise

Past Operator

Formal

▷ Verification

SymbiYosys Tasks

Exercise

Bonus

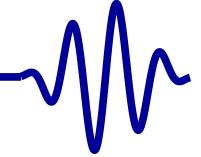
Conclusion

You can also use **\$past** with **cover**

```
always @(posedge i_clk)
if (f_past_valid)
    cover ((!busy)&&($past(busy)));
```



SymbiYosys Tasks



Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
Formal Verification
 ▷ SymbiYosys
 Tasks
Exercise
Bonus
Conclusion

Constantly editing our SymbiYosys file is getting old

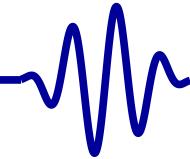
- Running cover, then
- Editing our script, then
- Running induction, then . . .
- Can we do this with one file?

Yes, using SymbiYosys tasks!

- SymbiYosys allows us to define multiple different scripts
- . . . all in the same file
- It does this using tasks



SymbiYosys Tasks



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

Wishbone Bus

Simulation

Unused Logic

Sim Exercise

Past Operator

Formal Verification

SymbiYosys

▷ Tasks

Exercise

Bonus

Conclusion

Let's define two tasks

- cvr to run cover
- prf to run induction

SymbiYosys lines prefixed by a task name are specific to that task

[**tasks**]

prf

cvr

[**options**]

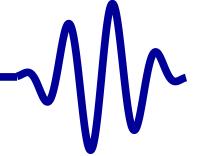
cvr: **mode** cover

prf: **mode** prove

The full reqwalker.sby file is with the course handouts



SymbiYosys Tasks



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

Wishbone Bus

Simulation

Unused Logic

Sim Exercise

Past Operator

Formal Verification

SymbiYosys

▷ Tasks

Exercise

Bonus

Conclusion

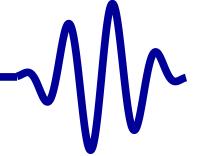
We can now run a named task

```
% sby -f reqwalker.sby prf
```

... or all tasks in sequence

```
% sby -f reqwalker.sby
```

SymbiYosys Tasks



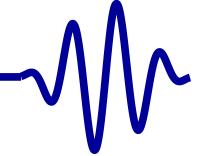
Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
Formal Verification
 SymbiYosys
▷ Tasks
Exercise
Bonus
Conclusion

I use this often with the ZipCPU

- Using the yosys command hierarchy I can describe multiple configurations to verify
 - With/Without the pipeline
 - With/Without the instruction cache
 - With/Without the data cache
 - . . . , etc.
- SymbiYosys tasks are very useful!



Exercise



Lesson Overview

LED Walker

Diagrams

Pipeline

Bus

Wishbone Bus

Simulation

Unused Logic

Sim Exercise

Past Operator

Formal Verification

SymbiYosys Tasks

▷ Exercise

Bonus

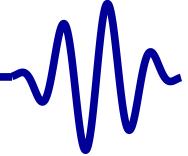
Conclusion

Your turn! Formally verify this design

- Build and create a SymbiYosys script
- Apply to the example design
- Adjust the design until it passes
 - Did you find any bugs?
 - Why weren't these bugs caught in simulation?



Exercise

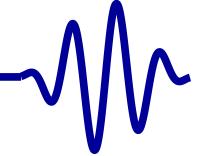


Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
▷ Exercise
Bonus
Conclusion

Your turn to design

- *Add the integer clock divider to this design*
(Otherwise you'd never see the LED's change on real hardware)
- Adjust both simulator and formal properties
- Create a simulation trace
- Create a cover trace
Do they match?

Bonus



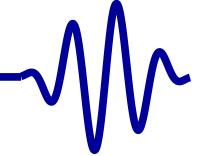
Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
▷ Bonus
Conclusion

Bonus: If you have hardware with more than one LED ...

- Adjust the number of LED's to match your hardware
- Create an `i_btn` input and connect it to a button
- Replace the `i_stb` input with the logic below

```
reg      stb;
initial stb = 0;
always @(posedge i_clk)
  if (i_btn)
    stb <= 1'b1;
  else if (!busy)
    stb <= 1'b0;
```

Bonus

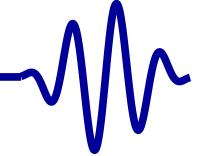


Lesson Overview
LED Walker
Diagrams
Pipeline Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
▷ Bonus
Conclusion

Bonus: If you have hardware with more than one LED

- Adjust the number of LED's to match your hardware
- Create an `i_btn` input and connect it to a button
- Replace the `i_stb` input with the given logic
- Tie `i_we` high
- Ignore `o_stall`, `i_cyc`, etc.
You'll need to adjust the formal properties
You should still be able to simulate it
- Simulate this updated design
- Implement it on your hardware
 - Did it do what you expected? Why or why not?
 - Does the LED walk back and forth when you press the button?
It should!
It might not work reliably . . . yet

Conclusion



Lesson Overview
LED Walker
Diagrams
Pipeline
Bus
Wishbone Bus
Simulation
Unused Logic
Sim Exercise
Past Operator
Formal Verification
SymbiYosys Tasks
Exercise
Bonus
▷ Conclusion

What did we learn this lesson?

- Pipeline handshaking, `i_request && !o_busy`
- State transition diagrams
- Definition of a bus
- Logic involved in processing the wishbone bus
- How to make a wishbone slave
- How to make wishbone bus calls from your Verilator C++ driver
- How to ignore unused logic in Verilator
- Verilator requires a call to `eval()` for combinatorial logic to settle
- The **\$past** operator in formal verification
- SymbiYosys tasks



Gisselquist
Technology, LLC

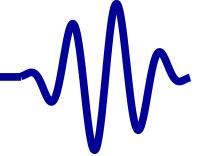
5. Serial Port Transmitter

Daniel E. Gisselquist, Ph.D.





Lesson Overview



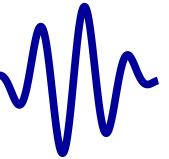
- ▷ Lesson Overview
- Serial Protocol
- Implementation
- Submodules
- Top Level
- Philosophy
- Simulation
- Main simulation file
- Cosimulation
- Make Foo
- Formal Verification
- Verifying txuart
- Formal Contract
- Exercise #1
- Hello World
- Exercise #2
- Hardware!
- Conclusion

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

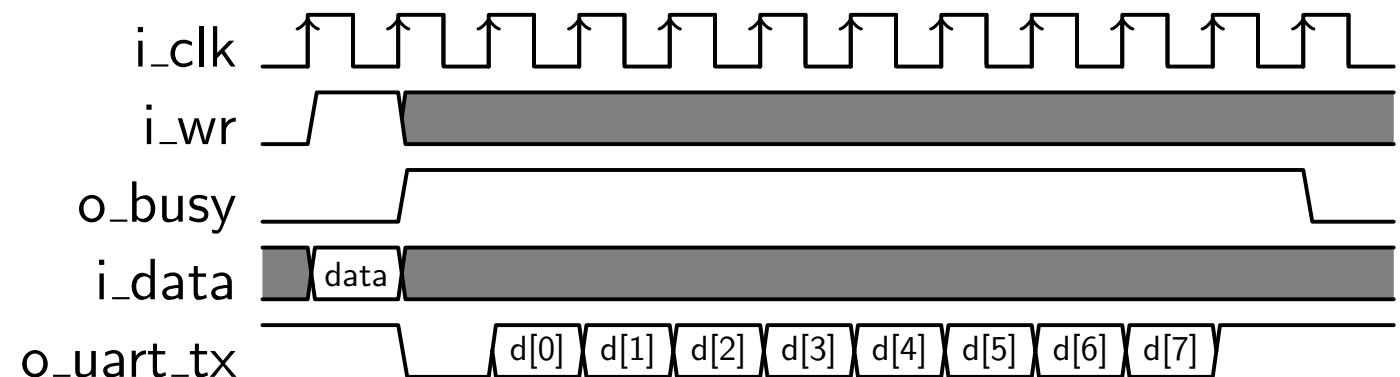
Objectives

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

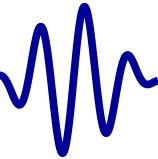


Lesson Overview
▷ Serial Protocol
Implementation
Submodules
Top Level
Philosophy
Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

Let's transmit a character

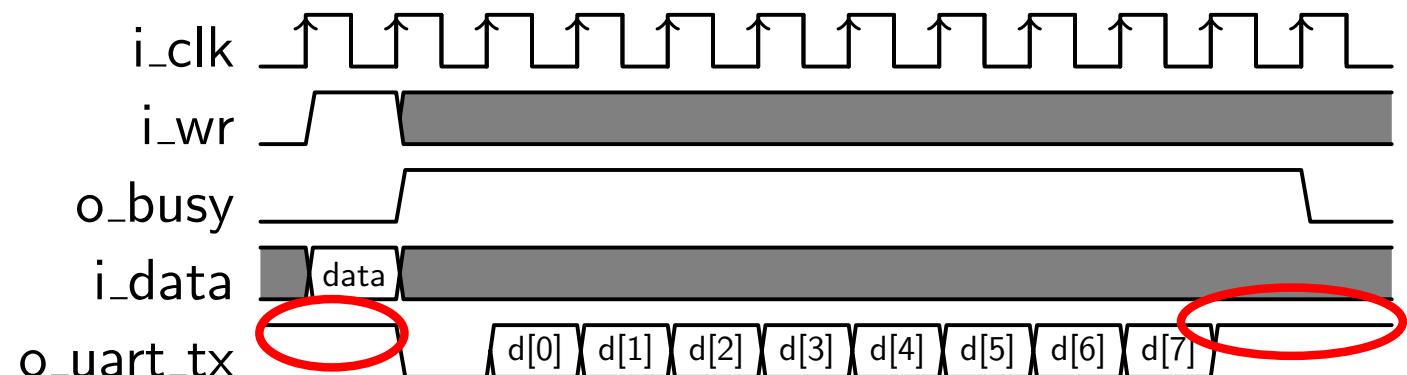


A serial transmission ...



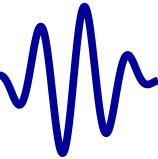
Lesson Overview
▷ Serial Protocol
Implementation
Submodules
Top Level
Philosophy
Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

Let's transmit a character



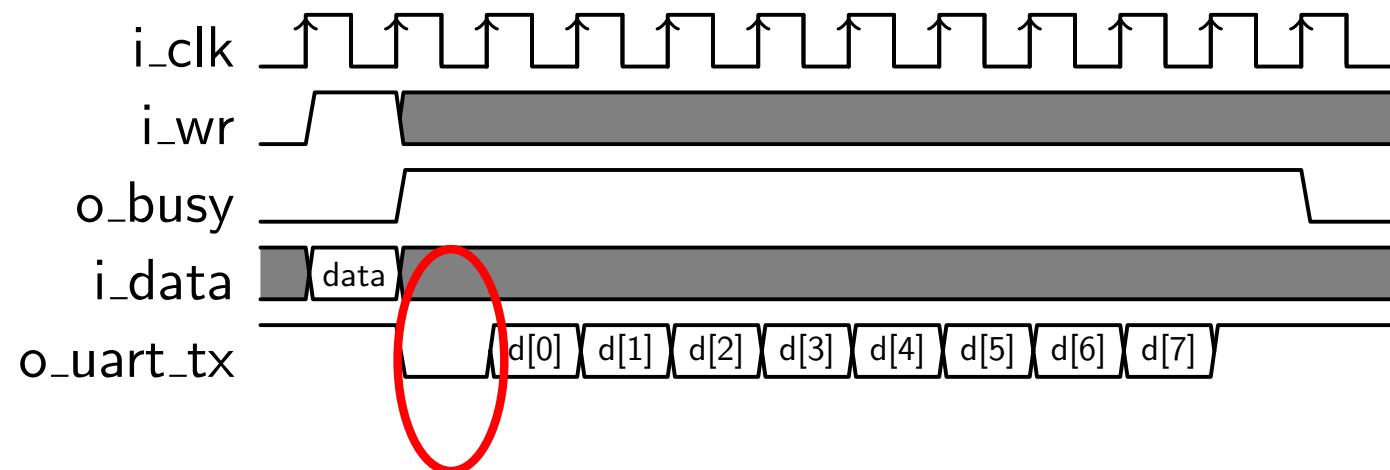
A serial transmission ...

- Idles high



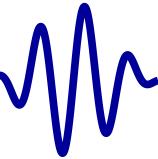
Lesson Overview
▷ Serial Protocol
Implementation
Submodules
Top Level
Philosophy
Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

Let's transmit a character



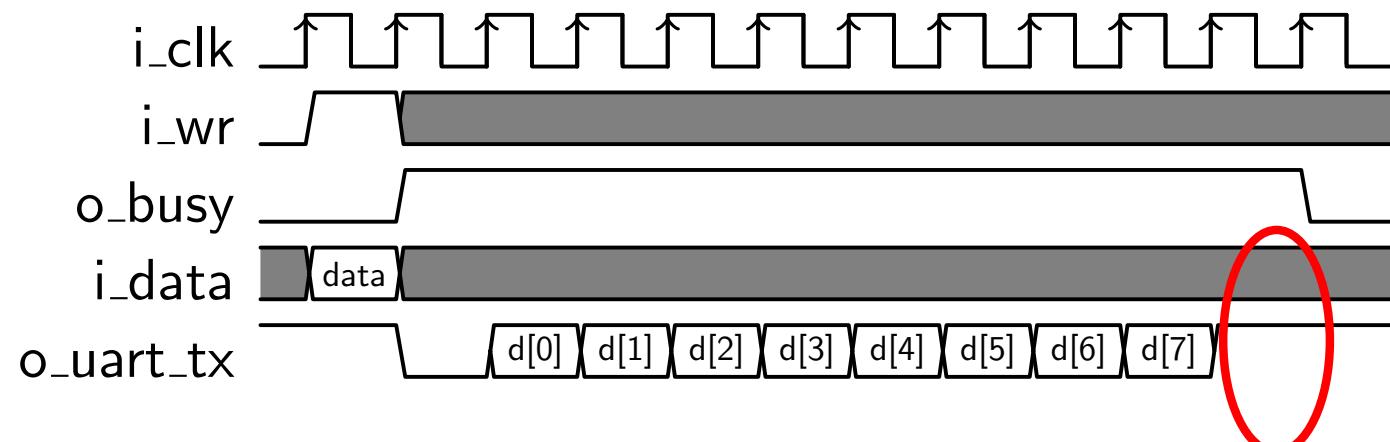
A serial transmission ...

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



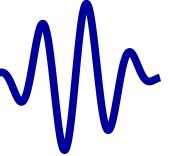
Lesson Overview
▷ Serial Protocol
Implementation
Submodules
Top Level
Philosophy
Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

Let's transmit a character



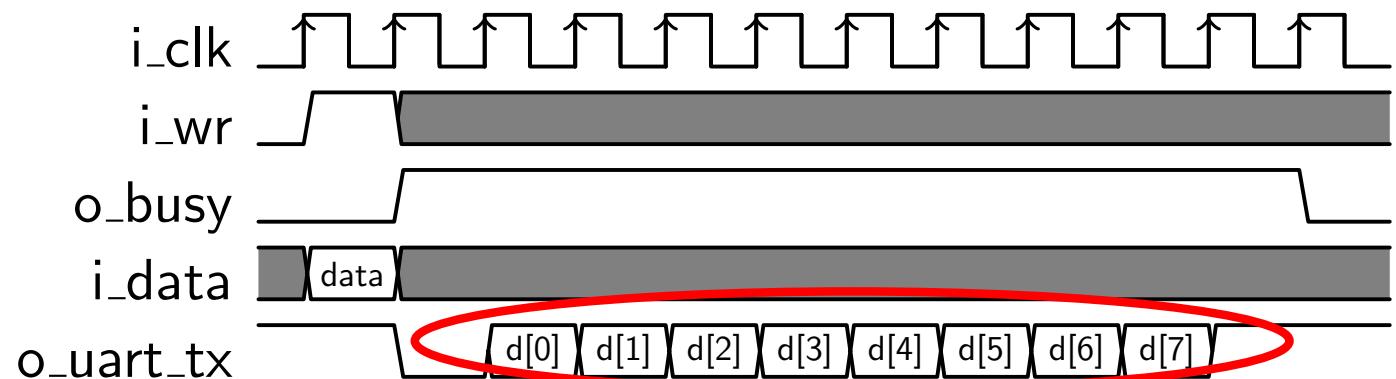
A serial transmission ...

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



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



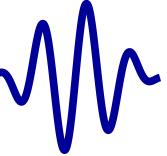
A serial transmission ...

- 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

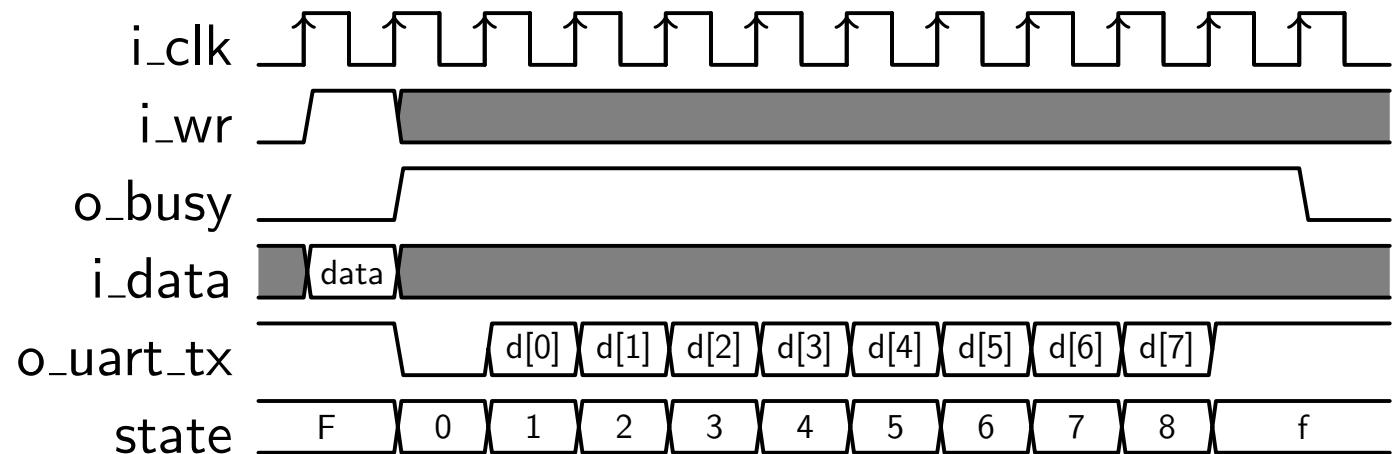


Goal



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

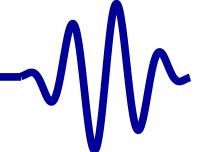
Let's add state ID's to this diagram



This will work for now

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

State Variable



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

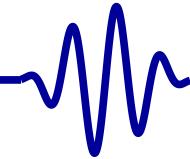
Hardware!

Conclusion

We can set o_busy together with our state

```
initial { o_busy, state } = { 1'b0, IDLE }; //=15
always @(posedge i_clk)
if ((i_wr)&&(!o_busy))
    // Start a new byte, state START=0
    { o_busy, state } <= { 1'b1, START };
else if (state == IDLE)
    // Stay in IDLE = 15 or 0x0f
    { 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?



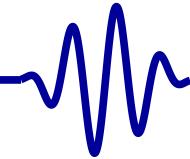
Lesson Overview
Serial Protocol
▷ Implementation
Submodules
Top Level
Philosophy
Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

The outgoing data is just a shift register

```
initial lcl_data = 9'h1ff;
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[8:1] };

assign o_uart_tx = lcl_data[0];
```

The output depends upon state only



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

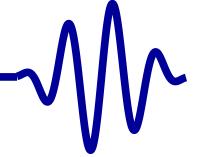
```
initial lcl_data = 9'h1ff;
always @(posedge i_clk)
if ((i_wr)&&(!o_busy))
    // Load the register
    // Start outputting a zero
    lcl_data <= { i_data, 1'b0 };
else
    // Shift right for more data
    // Shift 1'b1 in from the left
    lcl_data <= { 1'b1, lcl_data[8:1] };

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

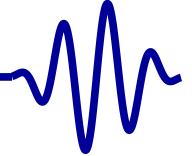
All that remains is an integer clock divider!

- 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?



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!

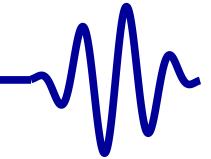
- We'll adjust our logic above to only change on baud_stb
- ... or (if idle) on (i_wr)&&(!o_busy)

```
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



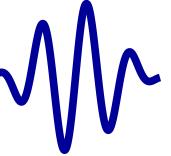
Lesson Overview
Serial Protocol
▷ Implementation
Submodules
Top Level
Philosophy
Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

All that remains is an integer clock divider!

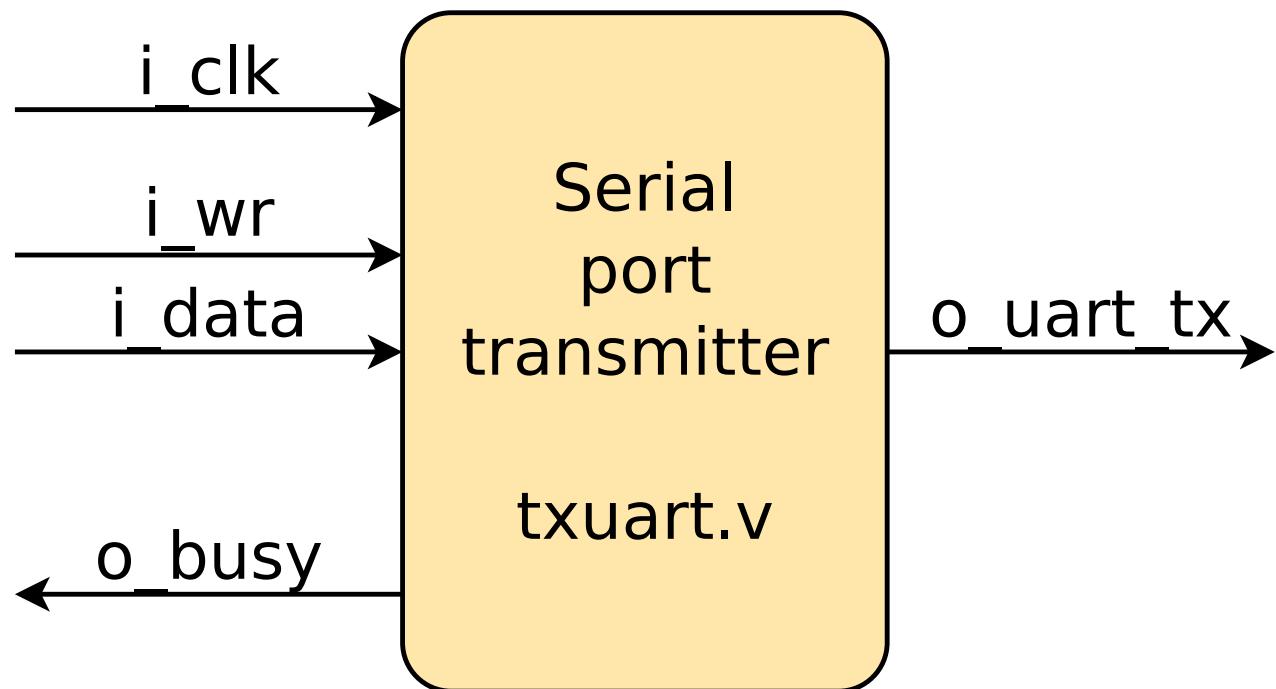
- We'll adjust our logic above to only change on `baud_stb`
- ... or (if idle) on `(i_wr)&&(!o_busy)`

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

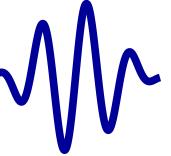
- This risks another condition taking priority over `(i_wr)&&(!o_busy)`
- Result is that the transmitter doesn't notice the transmit request
- This mistake can usually be caught using formal methods.



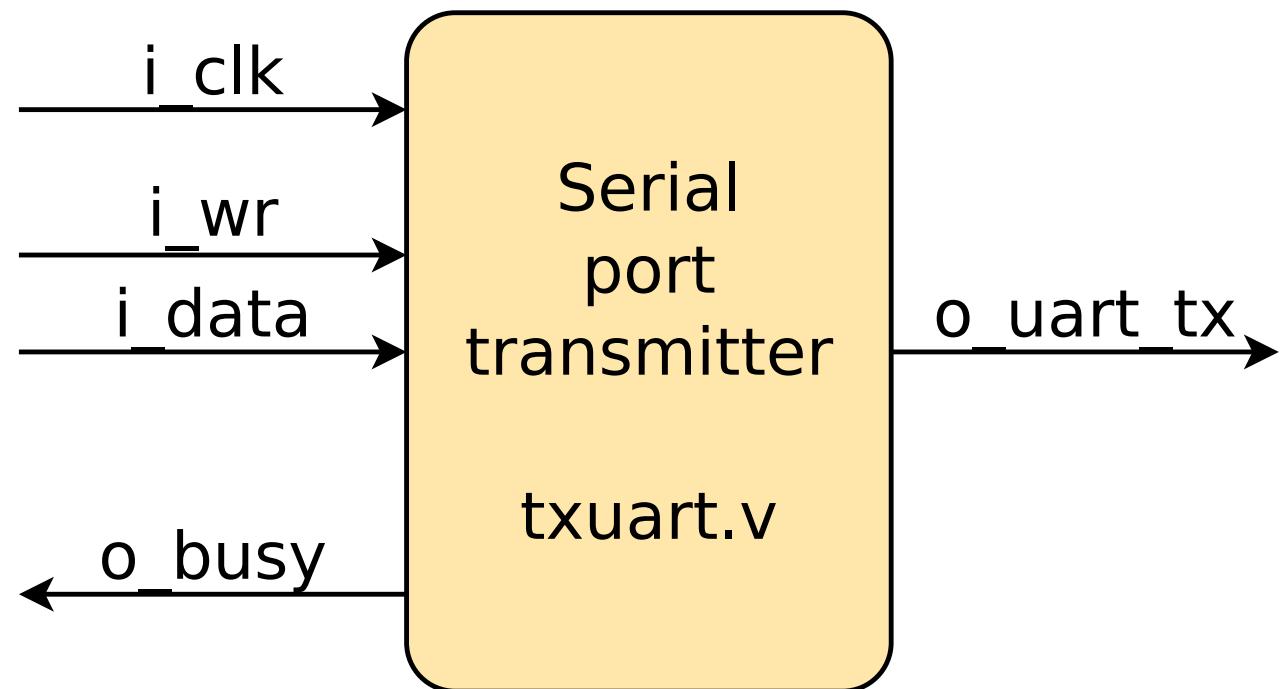
Lesson Overview
Serial Protocol
▷ Implementation
Submodules
Top Level
Philosophy
Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion



- `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`)

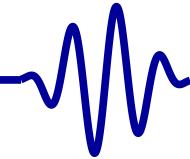


Lesson Overview
Serial Protocol
▷ Implementation
Submodules
Top Level
Philosophy
Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

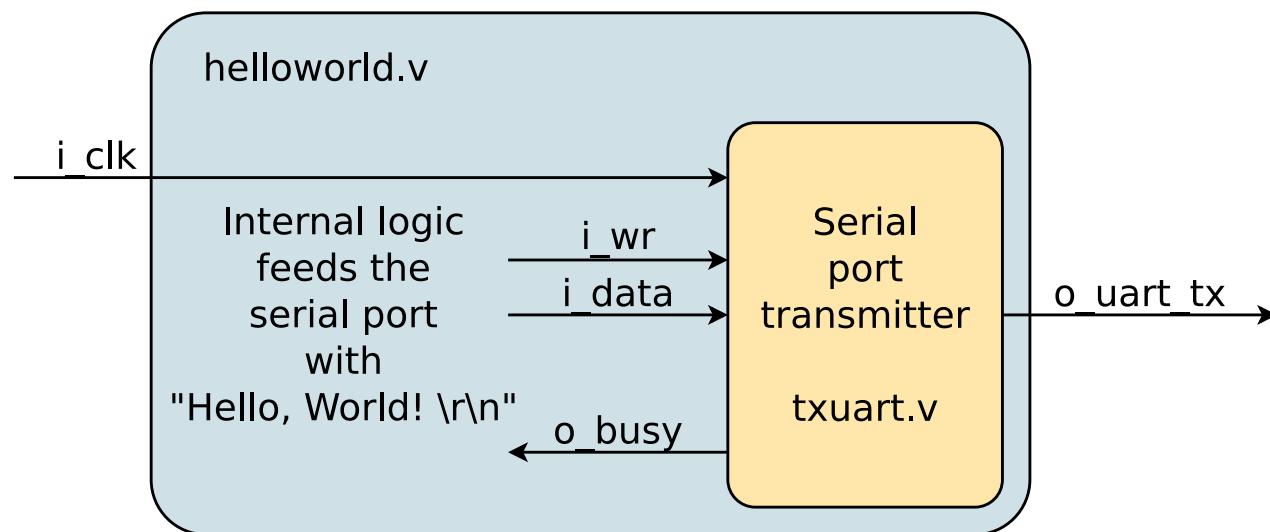


A good serial port

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



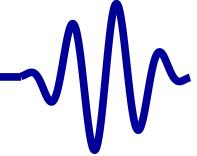
Lesson Overview
Serial Protocol
Implementation
▷ Submodules
Top Level
Philosophy
Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion



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.

GT 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

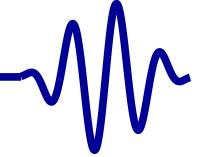
```
txuart #(CLOCK_RATE_HZ / MYBAUDRATE)
          mytxuart(clk, tx_stb, tx_data, o_uart,
                    tx_busy);
```

- 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 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

GT 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

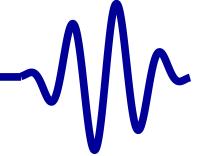
1. Pass by port-order
2. Pass by port name

```
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

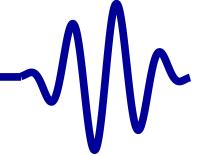


Lesson Overview
Serial Protocol
Implementation
Submodules
▷ Top Level
Philosophy
Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

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 <= "I";
    4'h3: tx_data <= "I"; // 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"; // Carriage return
    4'hf: tx_msg <= "\n"; // Line feed
  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

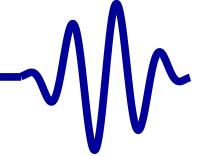
```
// tx_index tells us what character to send next
always @(posedge i_clk)
if ((tx_stb)&&(!tx_busy))
    tx_index <= tx_index + 1'b1;

// 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

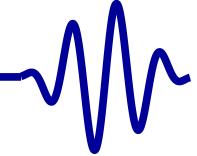
- it needs a driver, helloworld.v
- It needs to be periodically restarted

```
// Integer clock divider
initial hz_counter = 28'h16;
always @(posedge i_clk)
if (hz_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

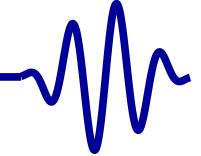
- You have no way of knowing if you did it right other than hardware test
- You can only debug using LED's
- When it doesn't work, you'll never know why not
- They don't teach you to use
 - Simulation, or
 - Formal methodsto find latent bugs in your design

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

We can do better!



Philosophy



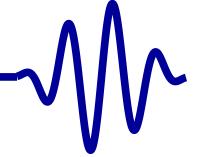
Lesson Overview
Serial Protocol
Implementation
Submodules
Top Level
▷ Philosophy
Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

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

- Let us continue, and learn how to
 1. Simulate, then
 2. Formally verify

this design

GT Simulation

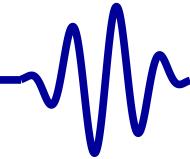


Lesson Overview
Serial Protocol
Implementation
Submodules
Top Level
Philosophy
▷ Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

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 pointers around
 - Keep multiple global variables
 - Use a C++ class that keeps variables with the methods that use them

Solution: a reusable Verilator template class!

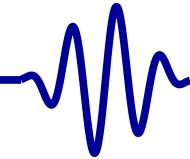


Lesson Overview
Serial Protocol
Implementation
Submodules
Top Level
Philosophy
▷ Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

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();
    }
    // ...
}
```

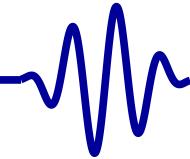


Lesson Overview
Serial Protocol
Implementation
Submodules
Top Level
Philosophy
▷ Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

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();  
    }  
    // ...
```

Use a template class to only do this once



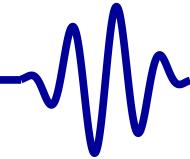
Lesson Overview
Serial Protocol
Implementation
Submodules
Top Level
Philosophy
▷ Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

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



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

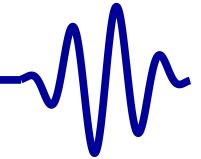
```
template <class VA> class TESTB {
public:
    VA                      *m_core;
    VerilatedVcdC           *m_trace;
    uint64_t                 m_tickcount;

    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



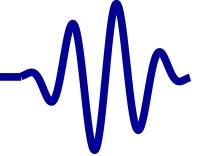
Lesson Overview
Serial Protocol
Implementation
Submodules
Top Level
Philosophy
▷ Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

That's the constructor, here's the destructor

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



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

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



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

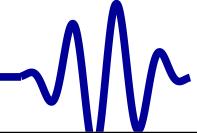
Finally, our operations. These haven't fundamentally changed.

```
// ...
virtual void eval(void) {
    m_core->eval();
}

virtual void tick(void) {
    // ...
    // This is the same as what we
    // introduced in our last
    // lesson ...
}

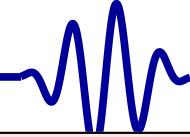
// ...
```

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



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



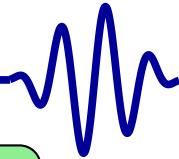
Lesson Overview
Serial Protocol
Implementation
Submodules
Top Level
Philosophy
Simulation
 Main simulation
 ▷ file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

```
#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();
    // ...
```

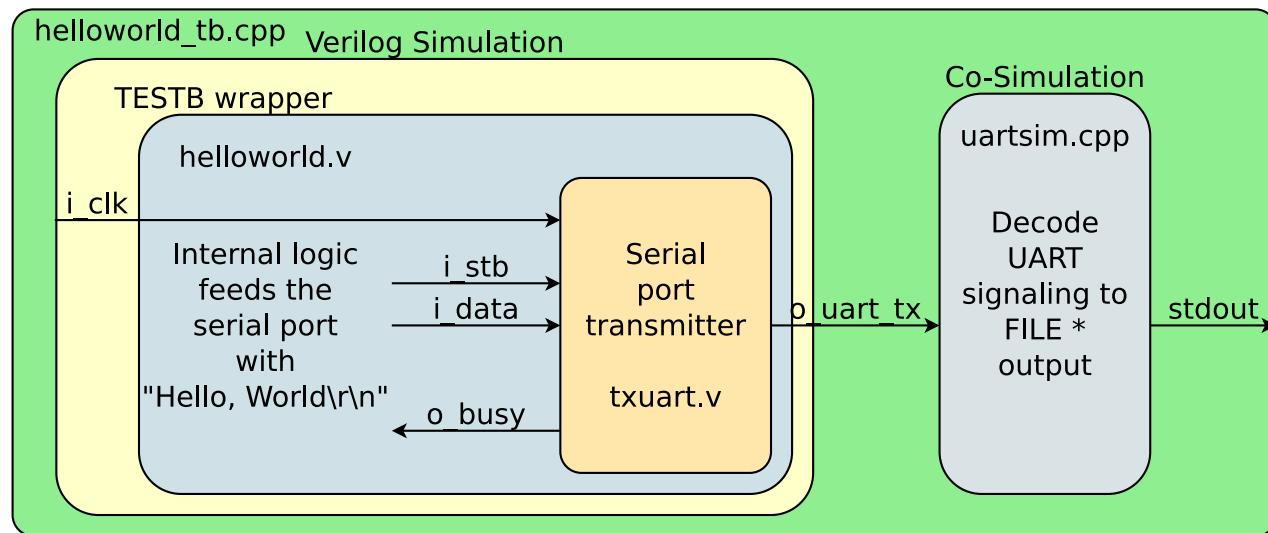


The secret key to success lies in the **UARTSIM** co-simulator

What is cosimulation?



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

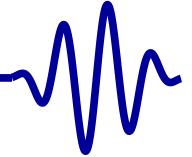


A cosimulator is a separate simulation

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

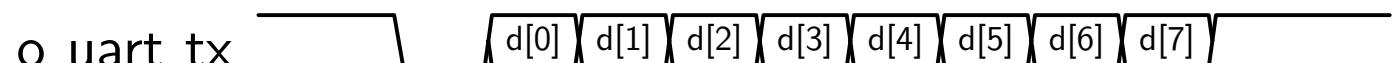


Serial Decoding

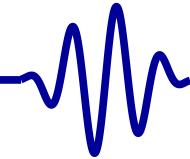


Lesson Overview
Serial Protocol
Implementation
Submodules
Top Level
Philosophy
Simulation
Main simulation file
▷ Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

Our co-simulation will need to decode this serial signal

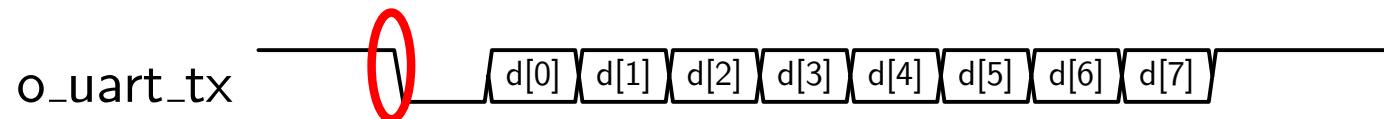


Steps to decode a serial port:



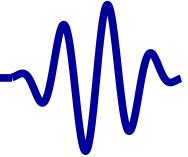
Lesson Overview
Serial Protocol
Implementation
Submodules
Top Level
Philosophy
Simulation
Main simulation file
▷ Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

Our co-simulation will need to decode this serial signal



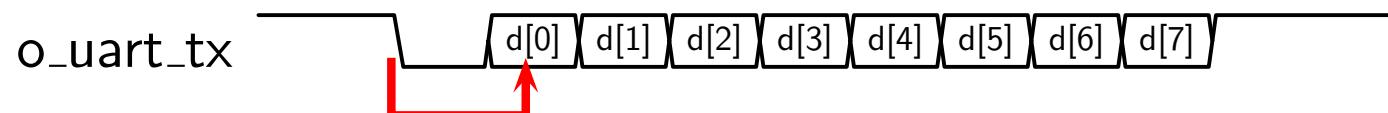
Steps to decode a serial port:

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



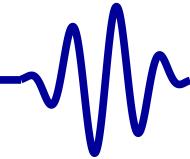
Lesson Overview
Serial Protocol
Implementation
Submodules
Top Level
Philosophy
Simulation
Main simulation file
▷ Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

Our co-simulation will need to decode this serial signal



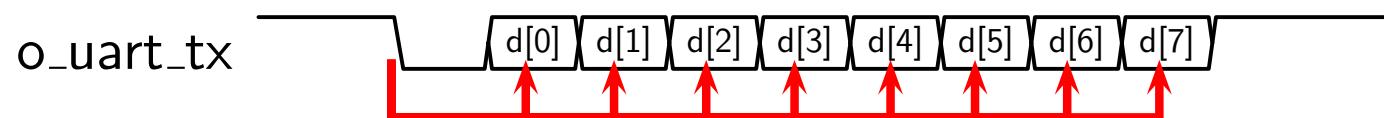
Steps to decode a serial port:

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



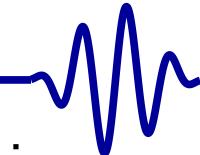
Lesson Overview
Serial Protocol
Implementation
Submodules
Top Level
Philosophy
Simulation
Main simulation file
▷ Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

Our 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



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
```



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

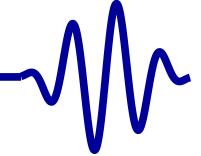
The first step is to make certain the cosimulator and design share the same baud rate

- 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);
    // ...
}
```

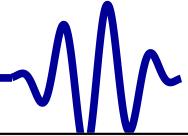
Now the cosimulator and design share the same baud rate



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 the co-sim work is done on a clock tick

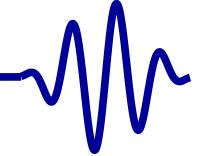
```
int      UARTSIM::operator()(const int i_tx) {  
  
    if (m_rx_state == RXIDLE) {  
        // Detect start bit  
        if (!i_tx) {  
            m_rx_state = RXDATA;  
            // Wait a baud and a half  
            m_rx_baudcounter = m_baud_counts  
                + m_baud_counts/2-1;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



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

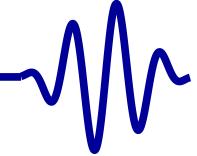
Example of re-use

```
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!

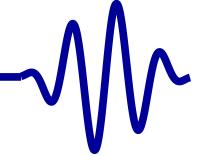
Conclusion

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`

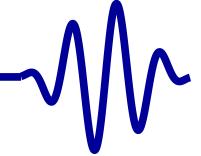
- We can include these into our Makefile with

```
DEPS := $(wildcard obj_dir/*.d)

ifeq ($(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 :

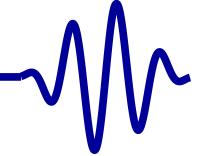
```
rm -rf obj_dir / $(TOPMOD)_tb
```

- 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

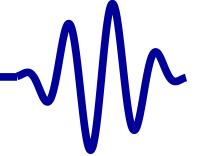
```
clean :
```

```
    rm -rf obj_dir/helloworld_tb
```

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

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

GT 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

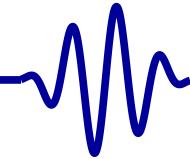
```
% ./helloworld_tb  
Hello , World !
```

```
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



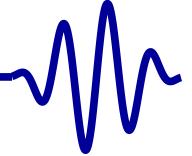
Lesson Overview
Serial Protocol
Implementation
Submodules
Top Level
Philosophy
Simulation
Main simulation file
Cosimulation
Make Foo
Formal
▷ Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

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

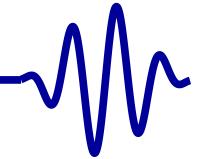
The entire design needs to be simplified

- 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

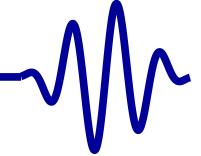


Lesson Overview
Serial Protocol
Implementation
Submodules
Top Level
Philosophy
Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
▷ Verifying txuart
Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

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
```



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

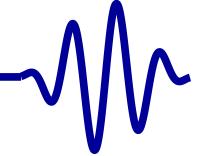
Some useful properties:

- 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));
```



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:

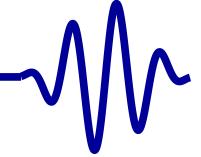
- If the counter is non-zero, the busy output should be true

```
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

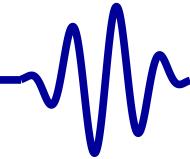
Any set of formal properties should include a *contract*

- Describes the required black-box behavior
- Describes how the core will be seen by the world
- Depends primarily on the outputs
- Shouldn't need to change if the underlying implementation changes

This is in addition to any assertions about local register values



Formal Contract



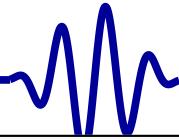
Lesson Overview
Serial Protocol
Implementation
Submodules
Top Level
Philosophy
Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
▷ Formal Contract
Exercise #1
Hello World
Exercise #2
Hardware!
Conclusion

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
```



Running SymbiYosys



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

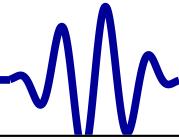
```
% sby -f txuart.sby
```

```
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.smvc
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$
```



Running SymbiYosys



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
```

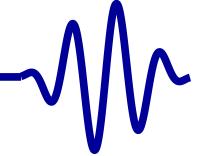
```
SBY 8:03:12 [txuart] engine_0.induction: ## 0:00:00 temporal induction failed!
SBY 8:03:12 [txuart] engine_0.basecase: ## 0:00:00 Checking assumptions in step 3..
SBY 8:03:12 [txuart] engine_0.basecase: ## 0:00:00 Checking assertions in step 3..
SBY 8:03:12 [txuart] engine_0.induction: ## 0:00:00 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.smvc
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\$

What happened?



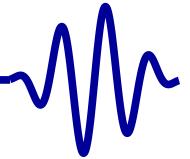
Formal Contract



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

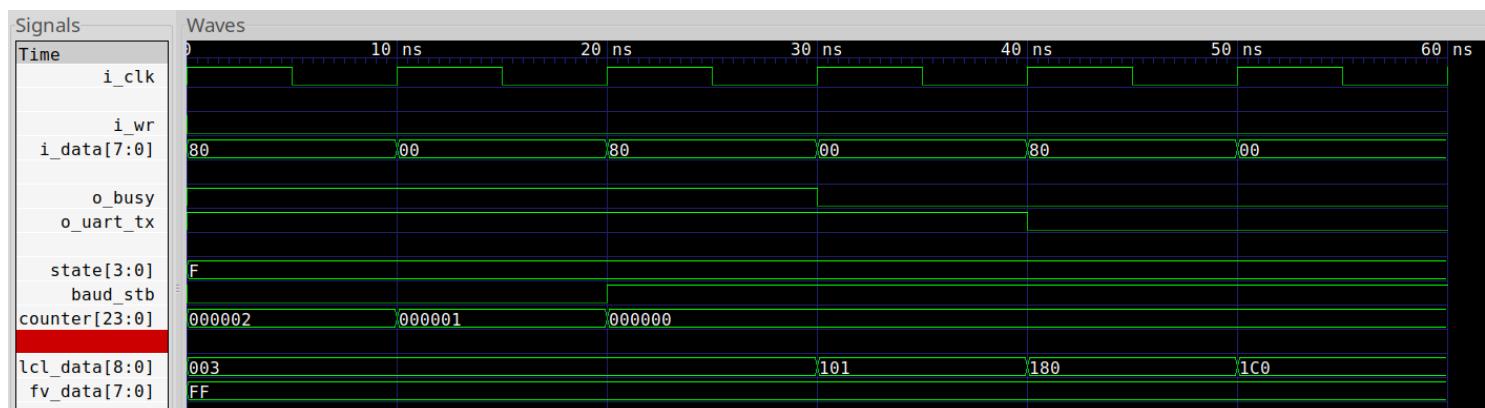
Our contract: *Failed Induction! Why?*

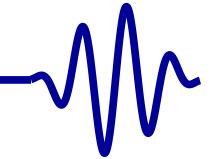
```
always @(posedge i_clk)
if ((i_wr)&&(!o_busy))
    fv_data <= i_data;
always @(posedge i_clk)
case(state)
IDLE:      assert(o_uart_tx);
START:     assert(o_uart_tx == 1'b0);
BIT_ZERO:  assert(o_uart_tx == fv_data[0]);
BIT_ONE:   assert(o_uart_tx == fv_data[1]);
BIT_TWO:   assert(o_uart_tx == fv_data[2]);
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
```



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

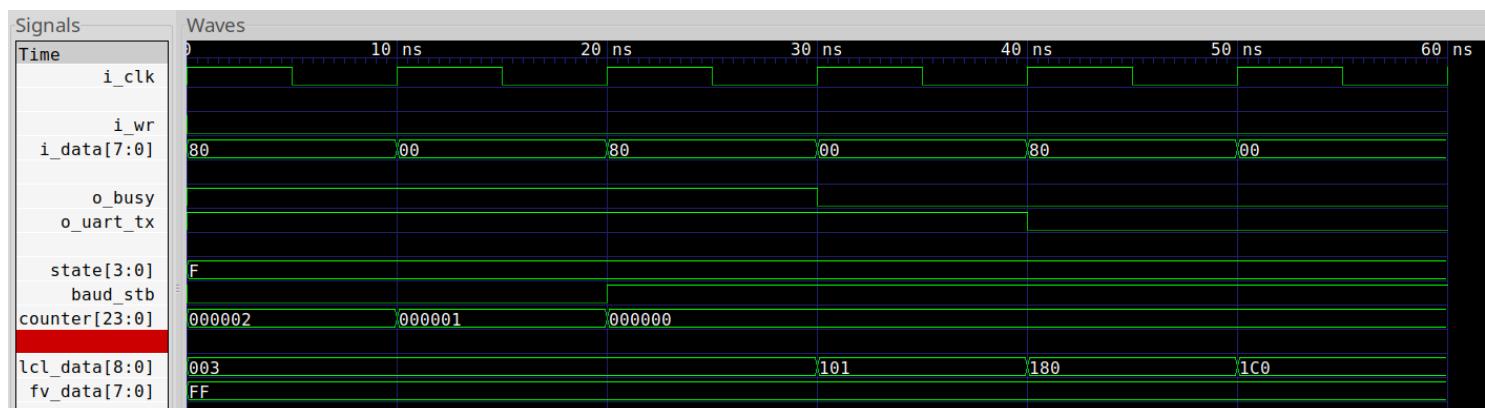
Need to look at the trace





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

Need to look at the trace



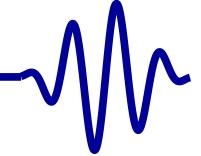
The problem
starts back here

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



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

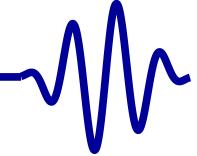
Fixing an induction problem always follows the same steps

- 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

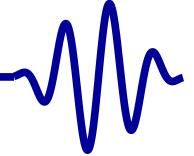
```
always @(*)
  case(state)
    IDLE: assert(lcl_data == 9'h1ff);
    default:
      endcase
```

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

- Hint: there are ten possible values for state, and only one assertion shown above.



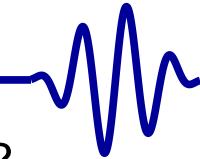
Exercise #1



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



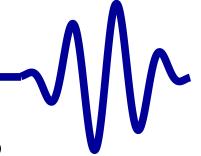
Lesson Overview
Serial Protocol
Implementation
Submodules
Top Level
Philosophy
Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
▷ Hello World
Exercise #2
Hardware!
Conclusion

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



What properties would be appropriate for helloworld.v?

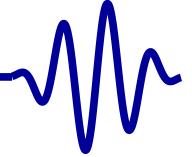
```
always @(*)  
if (tx_index != 4'h0)  
    assert(tx_stb);
```

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

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



Exercise #2

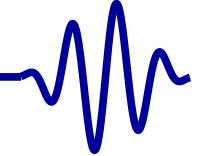


Lesson Overview
Serial Protocol
Implementation
Submodules
Top Level
Philosophy
Simulation
Main simulation file
Cosimulation
Make Foo
Formal Verification
Verifying txuart
Formal Contract
Exercise #1
Hello World
▷ Exercise #2
Hardware!
Conclusion

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

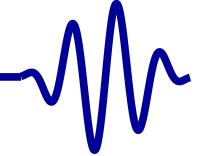
- Adjust CLOCK_RATE_HZ to match your board
- Your terminal should be set to
 - 8 data bits
 - No parity
 - One stop bit
 - No hardware flow control
 - A baud rate of BAUD_RATE (115.2kb)

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
 - and how to build a simulated UART receiver
 - How to make the simulation driver simpler
 - A little about using Makefile's with Verilator
 - What a formal “contract” is
 - The realities of working with induction
- We learned how to do our debugging *before touching the hardware!*



Gisselquist
Technology, LLC

6. Transmitting Data Words

Daniel E. Gisselquist, Ph.D.





Lesson Overview



- ▷ Lesson Overview
- Data Transmitter
- Desired Structure
- Counter
- Change Detection
- txdata
- State diagram
- Outgoing Data
- Formal Verification
- Cover
- Assertions
- Sequence
- Sequence
- Concurrent Assertions
- Simulation
- ncurses
- Verilator data
- Testbench build
- Exercise #2
- Exercise #3
- Conclusion

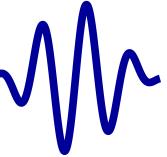
Debugging is one of the hardest parts of digital logic design

- 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!

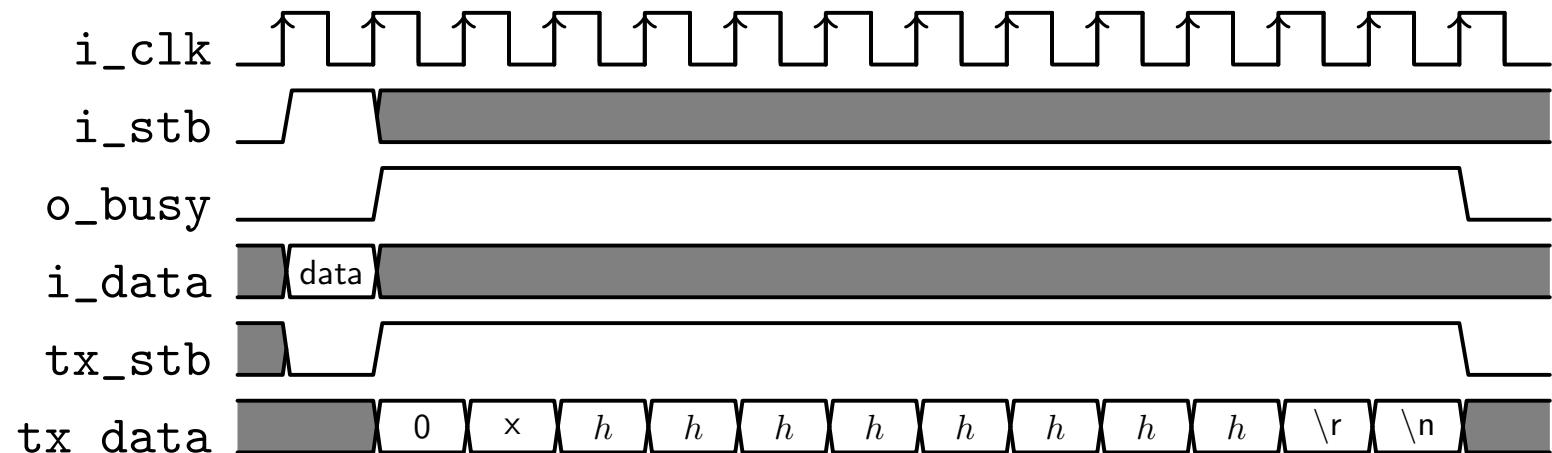
Objectives

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



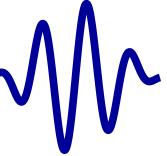
Lesson Overview
Data
▷ Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

Let's transmit a word of data



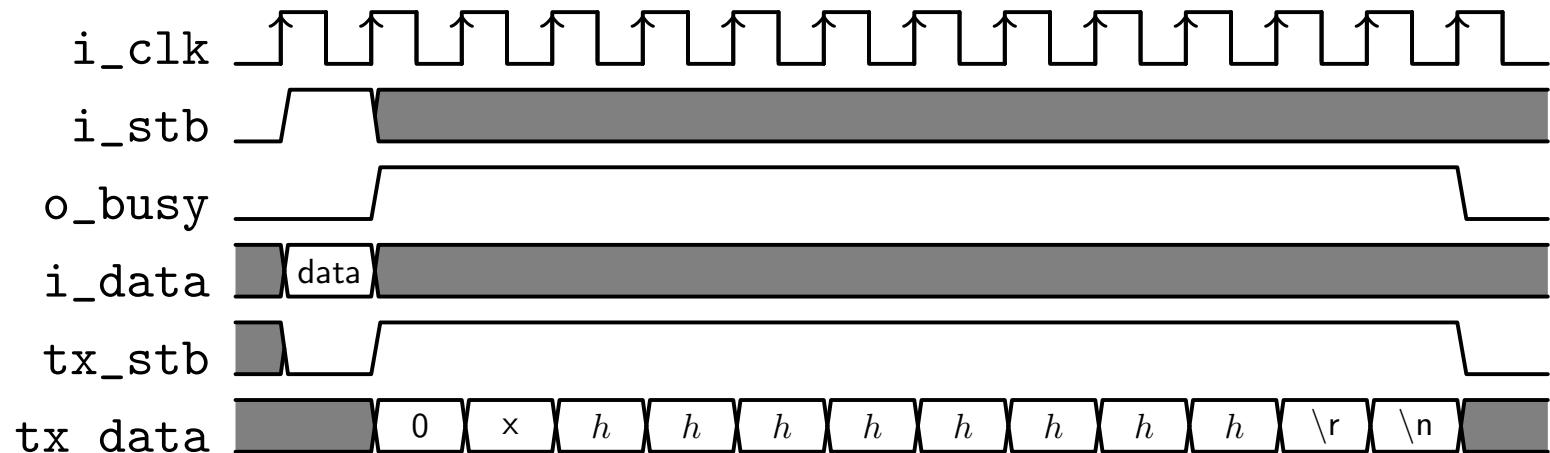
Each word will . . .

- 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



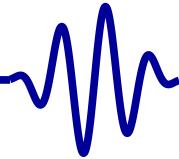
Lesson Overview
Data
▷ Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

You should know how to build this design already



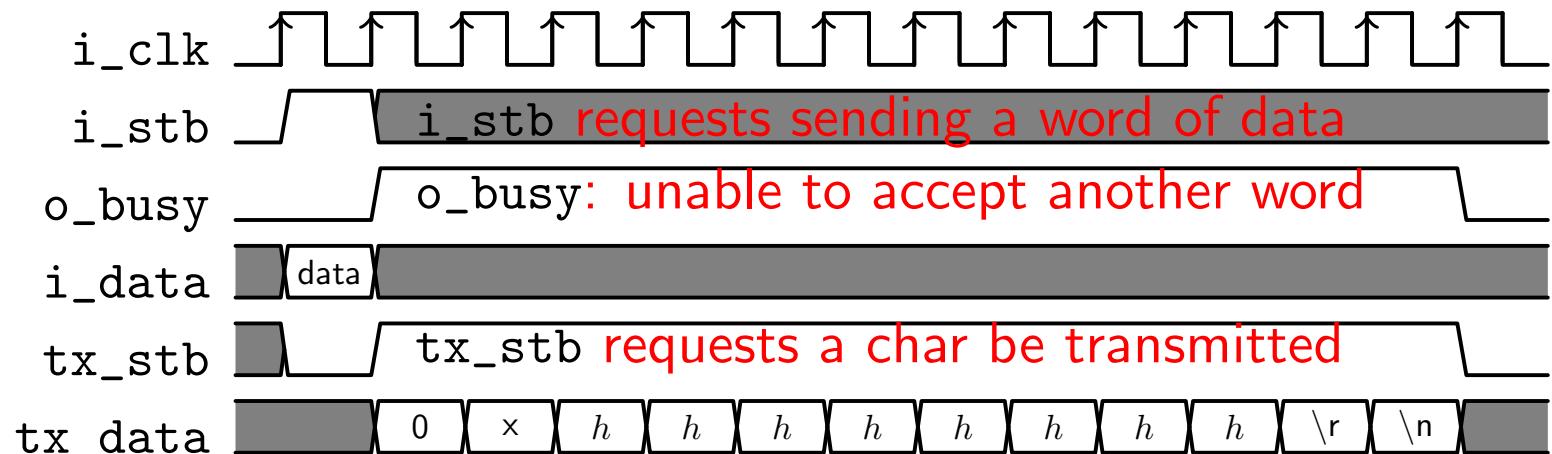
Remember how we've built state machines before

- 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



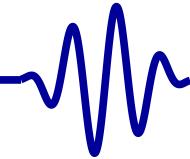
Lesson Overview
Data
Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

You should know how to build this design already



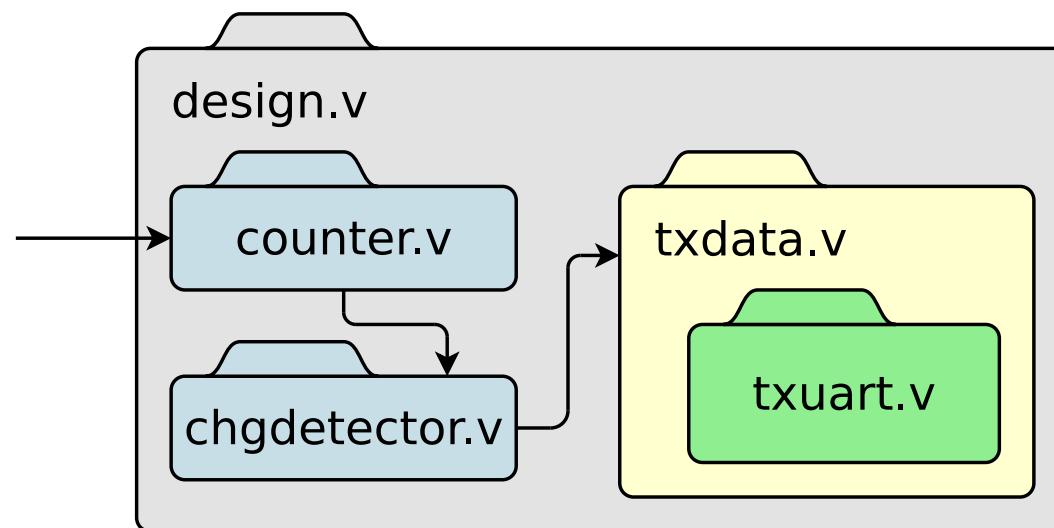
Remember how we've built state machines before

- 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



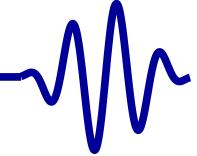
Lesson Overview
Data Transmitter
 Desired
 ▷ Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

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 chgdetect.v



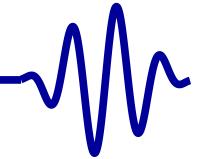
Lesson Overview
Data Transmitter
Desired Structure
▷ Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

You should already know how to make an event counter

```
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
```

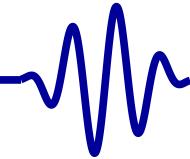
Feel free to add a reset if you would like



Lesson Overview
Data Transmitter
Desired Structure
Counter
 Change
▷ Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

Detecting a change in the counter is also pretty easy

```
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
    end
endmodule
```

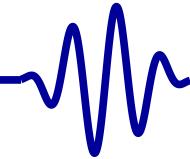


Lesson Overview
Data Transmitter
Desired Structure
Counter
 Change
 ▷ Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

Detecting a change in the counter is also pretty easy

```
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
                // ...
            end
    end
endmodule
```

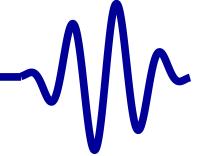
Nothing is allowed to change if `i_busy` is true. That's the case where a request has been made, but it has yet to be accepted.



Lesson Overview
Data Transmitter
Desired Structure
Counter
 Change
 ▷ Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

```
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
    end
endmodule
```

Otherwise, anytime the data changes, we set up a request to transmit the new data.



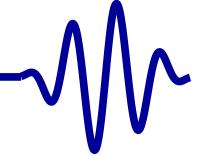
Lesson Overview
Data Transmitter
Desired Structure
Counter
Change
Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

What formal properties might we use here?

- Any output value should remain unchanged until accepted

```
// 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

```
// 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

```
// 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

```
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]))



What formal properties might we use here?

- Any output value should remain unchanged until accepted

```
// 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

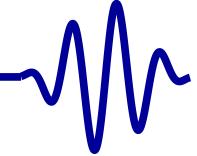
```
always @(posedge i_clk)  
  if ((f_past_valid)&&($rose(o_stb)))  
    assert(o_data == $past(i_data));
```

\$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



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
▷ txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

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

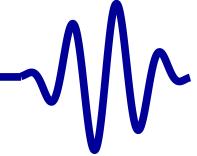
- If not, you can find counter.v and chgdetector.v in the course handouts

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



Our focus: txdata



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
▷ txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

Here's the port list(s) we'll design to

```
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
```



Our focus: txdata



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
▷ txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

Here's the port list(s) we'll design to

```
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
```

- 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

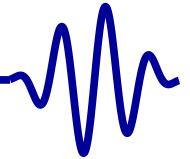


Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
▷ txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

Here's the port list(s) we'll design to

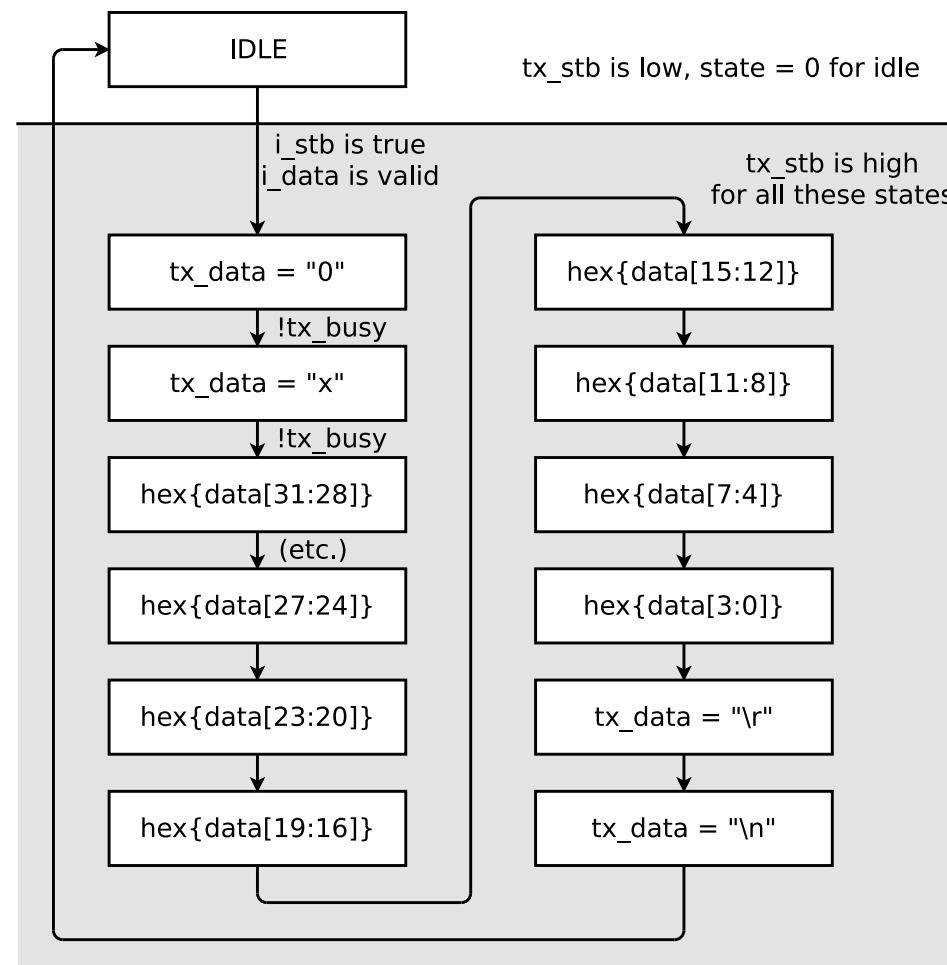
```
module txdata(i_clk, i_stb, i_data, o_busy,
              o_uart_tx);
  // ...
  txuart #(UART_SETUP[23:0]) txuarti(i_clk,
  tx_stb, tx_data, o_uart_tx, tx_busy);
  // ...
endmodule
```

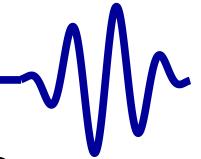
- tx_stb requests data be transmitted
- tx_data is the 8-bit character to transmit
- tx_busy means the serial port transmitter is busy and cannot accept data



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
▷ State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

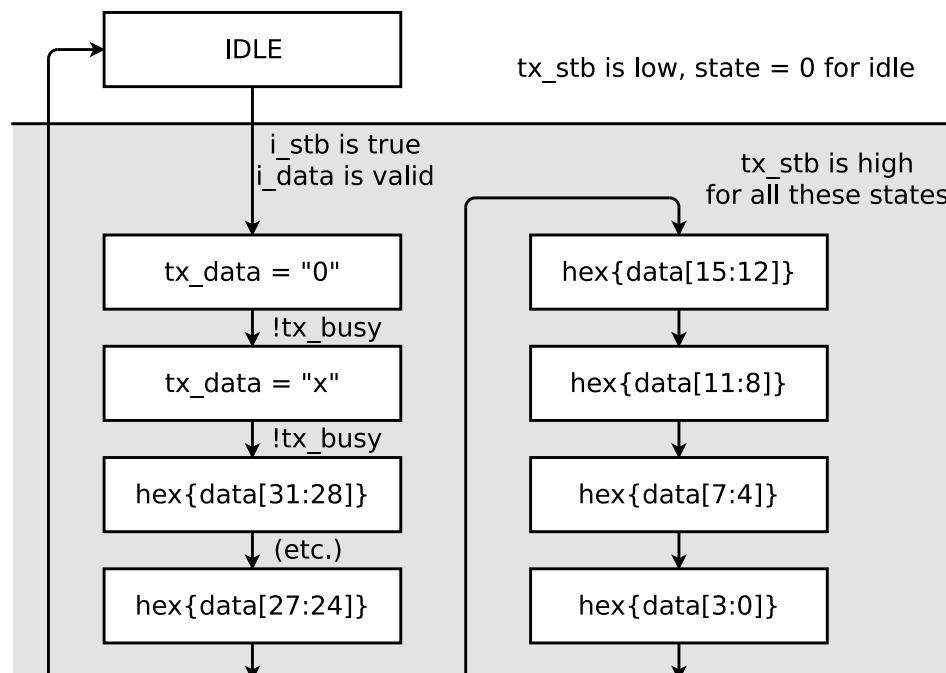
We can create a state diagram for this state machine too



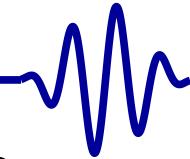


Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
▷ State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

We can create a state diagram for this state machine too

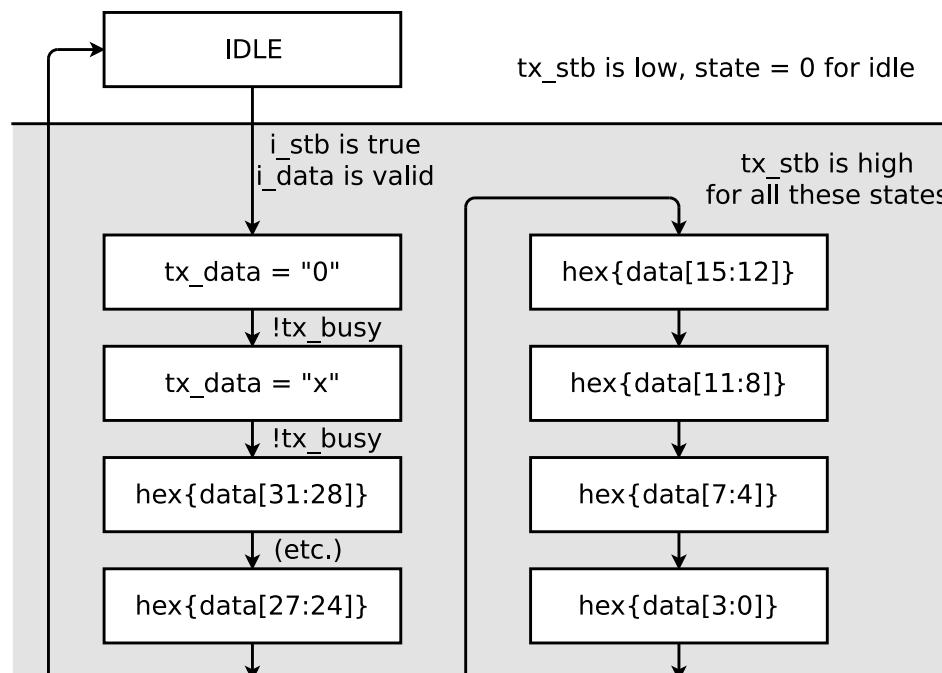


We'll start sending our message upon request (**i_stb** is true), and advance to the next character any time the transmitter is not busy (**tx_busy** is false)

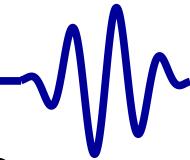


Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
▷ State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

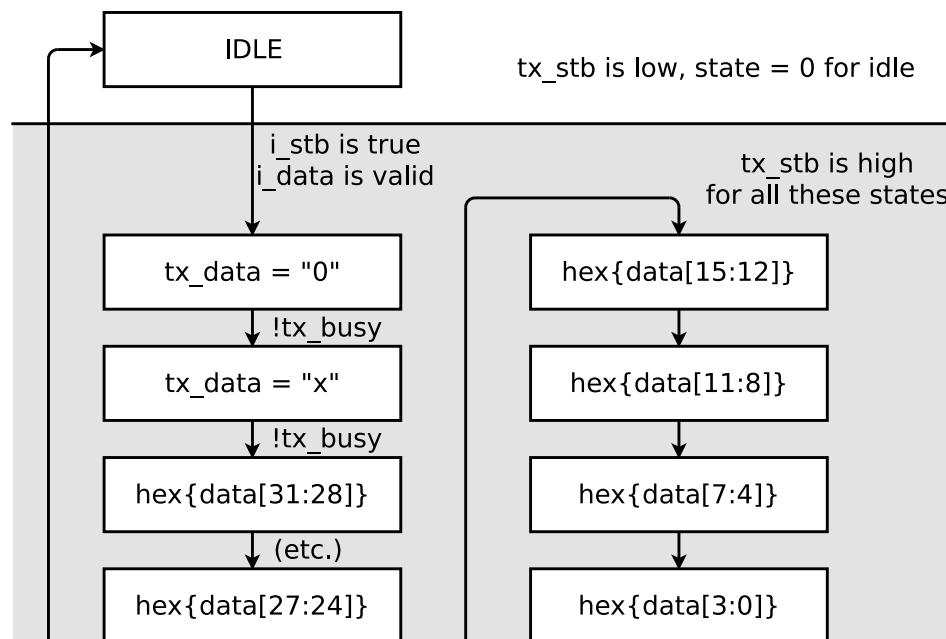
We can create a state diagram for this state machine too



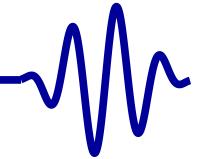
In this chart, data is the 32-bit word we are sending, and `hex{}` just references the fact that we need to convert the various nibbles to hexadecimal before outputting them



We can create a state diagram for this state machine too

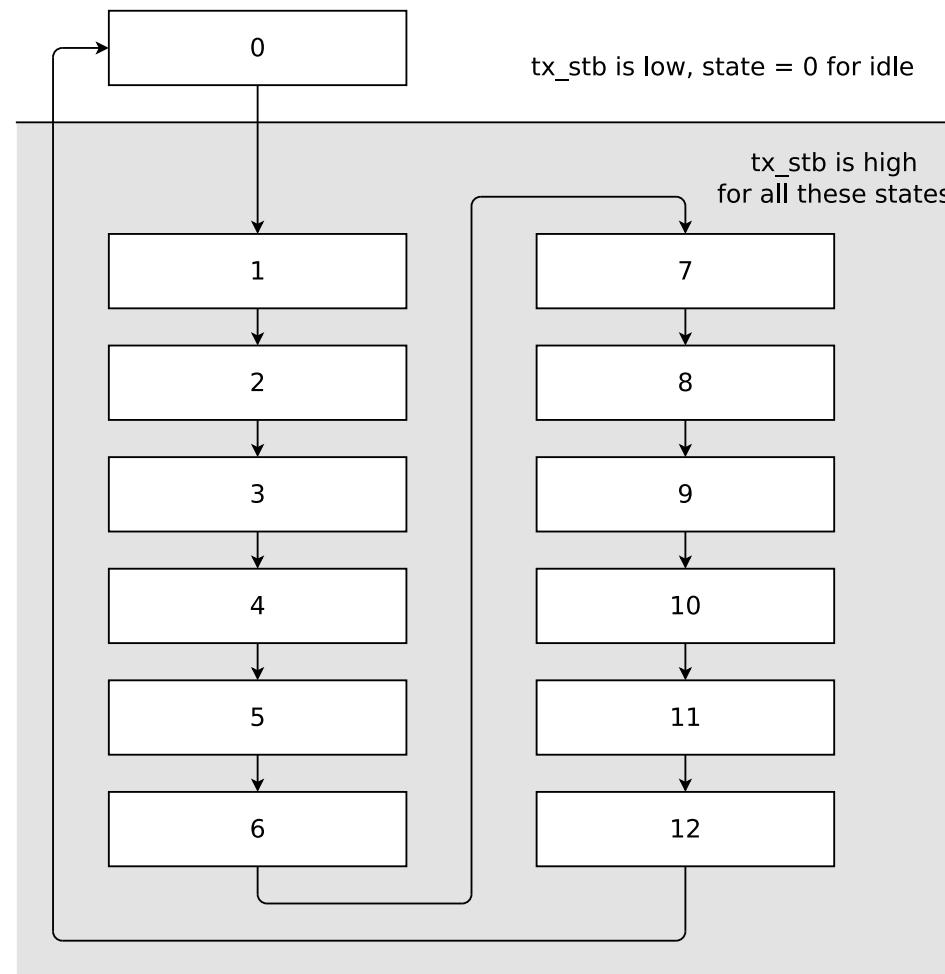


Remember, input data such as `i_data` are only valid as long as the incoming request is valid (`i_stb` is high). We'll need to make a copy of that data once the request is made, `(i_stb) && (!o_busy)`, and then work off of that copy.



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
▷ State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

We can even annotate this with state ID numbers





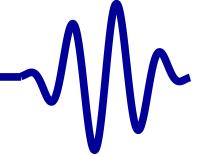
State diagram



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
▷ State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

The state machine should remind you of helloworld.v

```
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'd4)
        begin
            tx_stb <= 1'b0;
            state <= 0;
        end
    end
// ...
```



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
▷ Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

The outgoing data is just a shift register

```
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:

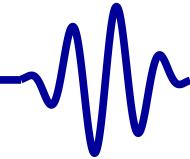
Why aren't we conditioning our load on i_stb as well?



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
▷ Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

Converting to hex is very straight forward

```
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'hc: hex <= "c";
    4'hd: hex <= "d";
    4'he: hex <= "e";
    4'hf: hex <= "f";
  default: begin end
endcase
```



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
▷ Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

Converting to hex is very straight forward

```
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
```

Outgoing Data



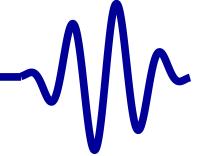
Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
▷ Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

Converting to hex is very straight forward

```
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



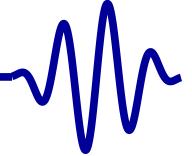
Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
▷ Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

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

```
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
endmodule
```



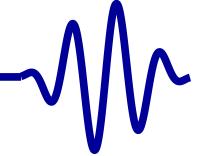
Simulation



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
▷ Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

Let's do simulation *after* formal verification

- It's easier to get a trace from formal
- Formal methods are often done faster
- etc.



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
 Formal
 ▷ Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

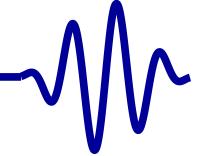
Our design is getting large

- 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

Formal Verification



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
 Formal
 ▷ Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

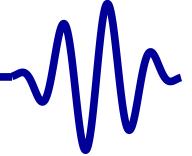
Our design is getting large

- 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

Formal Verification



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
 Formal
 ▷ Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

Our design is getting large

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

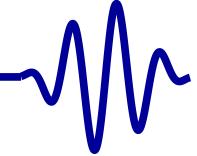
Let's simplify things instead!

- 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*



Formal Verification



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
 Formal
 ▷ Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

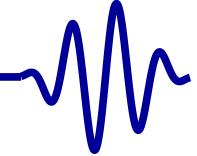
Here's how we'll do it:

```
'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



Formal Verification



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
 Formal
 ▷ Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

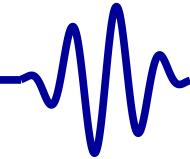
Here's how we'll do it:

```
'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



Formal Verification



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
 Formal
 ▷ Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

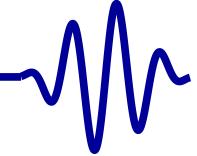
Here's how we'll do it:

```
'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*



Formal Verification



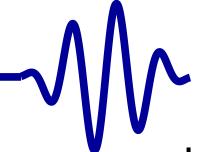
Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
 Formal
 ▷ Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

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

Formal Verification

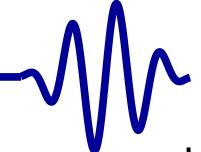


We'll insist that our abstract UART is busy following any request

```
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



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
 Formal
▷ Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

We'll insist that our abstract UART is busy following any request

```
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

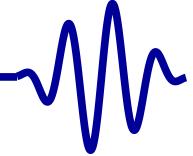


Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
 Formal
 ▷ Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

We'll also insist it doesn't become busy on its own

```
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!



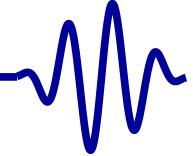
Let's see if this design works:

```
// 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

Cover



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
▷ Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

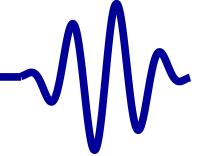
What if we except the reset?

```
// 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

Cover



What if we look for 0x12345678\r\n?

```
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.

Cover

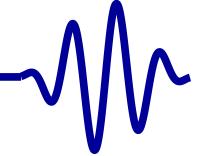


Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
▷ Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

What if we look for 0x12345678\r\n?

```
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?



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
▷ Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

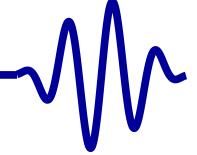
What if we look for 0x12345678\r\n?

```
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

GT Assertions



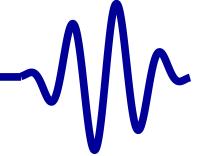
Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
▷ Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

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?

Sequence



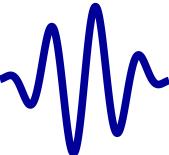
Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
▷ Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

Yes, we can assert a sequence takes place!

```
reg [12:0] f_p1reg; // Property s-reg

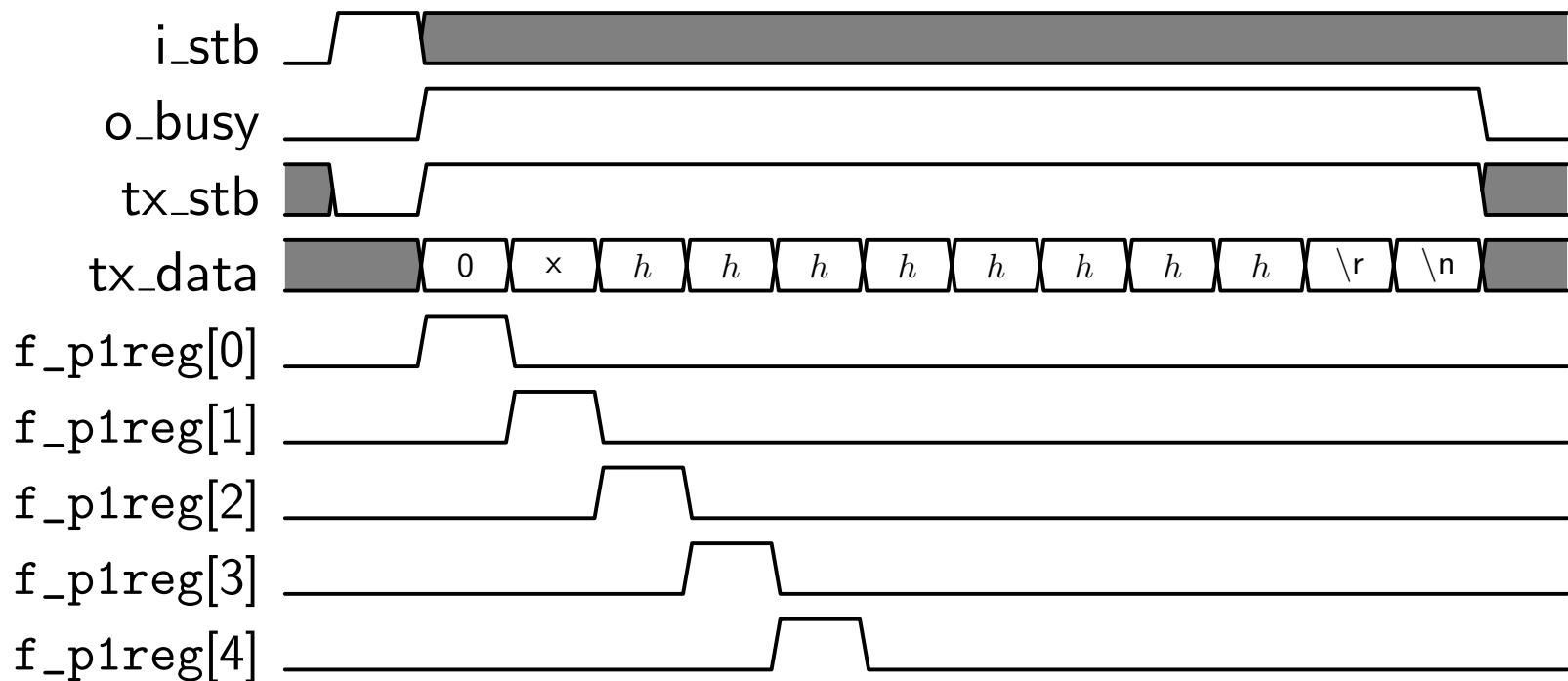
initial f_p1reg = 0;
always @(posedge i_clk)
if (i_reset)
    f_p1reg <= 0;
else if ((i_stb)&&(!o_busy))
begin
    f_p1reg <= 1;
    assert(f_p1reg == 0);
end else if (!tx_busy)
    f_p1reg <= { f_p1reg[11:0], 1'b0 };
```

f_p1reg[x] will now be true for stage x of any output sequence



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
▷ Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

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

Sequence

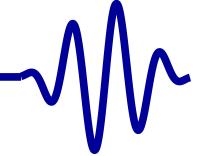


Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
▷ Sequence
Concurrent Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

Using `f_p1reg[x]` we can make assertions about the different states in our sequence

```
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

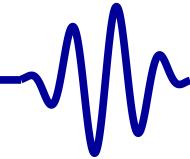


Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
▷ Sequence
Concurrent Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

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



Full System Verilog support would make this easier

```
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



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
▷ Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

Full System Verilog support would make this easier

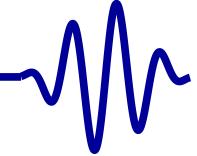
```
sequence SEND(A,B);  
    // . . .
```

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

```
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



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
▷ Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

Full System Verilog support would make this easier

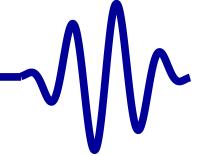
```
sequence SEND(A,B);  
    // . . .
```

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

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

SympyYosys support for sequences requires a license

Sequence



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
▷ Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

Full System Verilog support would make this easier

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

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

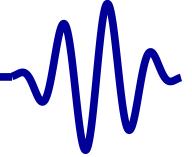
```
assert property (@(posedge i_clk))  
    disable iff (i_reset)  
    (i_stb)&&(!o_busy)  
    |=> SEND(1, "0") // First state  
    ##1 SEND(2, "x") // Second, etc  
    // . . .
```

SymbiYosys support for sequences requires a license

- f_p1reg let's us do roughly the same thing



Exercise #1



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
▷ Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

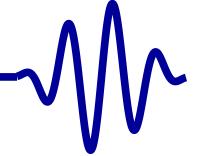
Your turn!

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?

GT Simulation

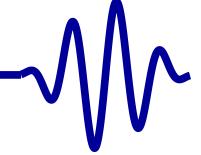


Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
▷ Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

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

ncurses

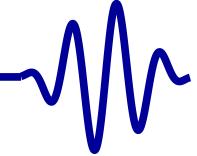


Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
▷ ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

Starting `ncurses` requires some boilerplate

```
#include <ncurses>
// ...
int main(int argc, char **argv) {
    // ...
    initscr();
    raw();
    noecho();
    keypad(stdscr, true);
    halfdelay(1);
```

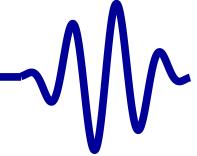
- This initializes the curses environment
- Turns off line handling and echo
- Decodes special keys (like escape) for us
- `halfdelay(1)` – Doesn't wait for keypresses



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
▷ ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

Our inner loop will start by checking for keypresses

```
do {  
    done = false;  
    tb->m_core->i_event = 0;  
    // Get 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:

```
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 $1/10^{th}$ of a second for a keypress
 - This is because we called **halfdelay(1)**;
- This will run 1000 simulation ticks per getch() call



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
▷ ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

We can also count given keypresses

```
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()**

```
// 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

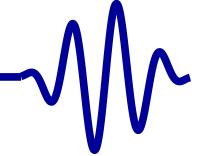


endwin() ends the **ncurses** environment

```
endwin();  
  
printf("\n\nSimulation\u2014complete\n");  
printf("%4d\u2014key\u2014presses\u2014sent\n", keypresses);
```

This is nice, but

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



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
▷ Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

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 ...
- `v__DOT__counterv` contains our counter's value
 - I use an older version of Verilator
 - Modern versions place this in `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



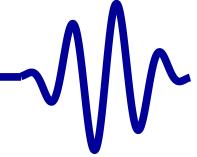
Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
▷ Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

This little adjustment will allow us to simplify the reference to our counter

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

#define counterv VVAR(_counterv)
```

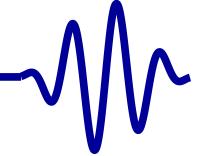
- If **OLD_VERILATOR** is defined (my old version)
 - **counterv** evaluates to **v__DOT__counterv**
- Otherwise **counterv** is replaced by
 - **thedesign__DOT__counterv**



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
▷ Verilator data
Testbench build
Exercise #2
Exercise #3
Conclusion

We can now output our current counter

```
endwin();  
  
printf("\n\nSimulation\u2014complete\n");  
printf("%4d\u2014key\u2014presses\u2014sent\n",  
       keypresses);  
printf("%4d\u2014key\u2014presses\u2014registered\n",  
       tb->m_core->counterv);
```

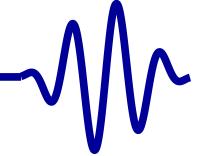


Two changes are required to our build script

- 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

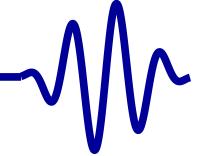


Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
▷ Exercise #2
Exercise #3
Conclusion

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?



Only now is it time to test this in hardware

- You'll need to test for button changes

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

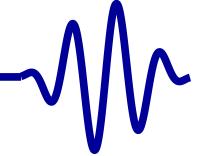
assign w_event = (i_btn) && (!last_btn);
```

- 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



Lesson Overview
Data Transmitter
Desired Structure
Counter
Change Detection
txdata
State diagram
Outgoing Data
Formal Verification
Cover
Assertions
Sequence
Sequence
Concurrent
Assertions
Simulation
ncurses
Verilator data
Testbench build
Exercise #2
Exercise #3
▷ 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



Gisselquist
Technology, LLC

7. Data Coherency

Daniel E. Gisselquist, Ph.D.





Lesson Overview



▷ Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

Co-Simulation

Exercise

Formal Methods

Conclusion

Understanding why the button counter didn't work as expected

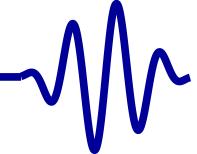
- It double counted button presses
- Sometimes it counted 2-4 times per button press
- Rarer observed effects
 - At one point, the counter counted *down*
 - Another time, it skipped 11 numbers at once

Objectives

- Understand data coherency issues
- Understanding bouncing
- Build and verify a button debouncer



Last Lesson



Lesson Overview

▷ Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

Co-Simulation

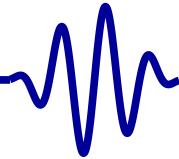
Exercise

Formal Methods

Conclusion

This lesson picks up where the last lesson left off.

- If you didn't build the button counter, or implement it in hardware
 - You missed a valuable lesson
 - Go back and try it
 - Press the button several times, see what happens
- If it didn't work like you expected it should
 - Feel free to start this lesson



Lesson Overview

Last Lesson

▷ Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

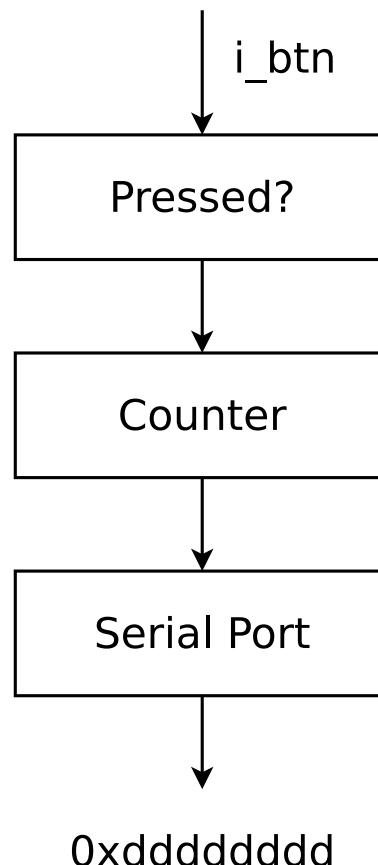
Co-Simulation

Exercise

Formal Methods

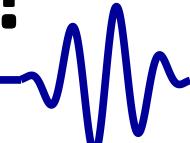
Conclusion

We built a button press counter in the last lesson



1. It detected a button press,
2. Incremented a counter,
3. Sent the value over the serial port as hexadecimal, and was
4. Witnessed at a terminal

An easy way to count button presses, no?



Lesson Overview
Last Lesson
Review
 What
 ▷ happened?
Logic takes time
Setup and Hold
Caving Analogy
No margin
Asynchronous Input
2FF Sync
Bouncing
Debouncing
FSM
Timer
Simulation
Co-Simulation
Exercise
Formal Methods
Conclusion

```
0x00000013  
0x00000014  
0x00000015  
0x00000017  
0x00000018  
0x00000019  
0x0000001a  
0x0000001b  
0x0000001c  
0x0000001d  
0x0000001e  
0x00000029  
0x0000002a  
0x0000002b  
0x0000002c  
0x0000002d  
0x0000002e  
0x0000002f  
0x00000030  
0x00000031  
0x00000032  
0x00000033  
0x00000034  
0x00000035
```

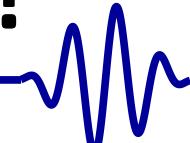
Button Press Counting



Did you notice that pressing the button once often caused the counter to count ... twice??

That's not right

This looks like it could be fixed



Lesson Overview
Last Lesson
Review
 What
 ▷ happened?
Logic takes time
Setup and Hold
Caving Analogy
No margin
Asynchronous Input
2FF Sync
Bouncing
Debouncing
FSM
Timer
Simulation
Co-Simulation
Exercise
Formal Methods
Conclusion

0x00000013
0x00000014
0x00000015
0x00000017
0x00000018
0x00000019
0x0000001a
0x0000001b
0x0000001c
0x0000001d
0x0000001e
0x00000029
0x0000002a
0x0000002b
0x0000002c
0x0000002d
0x0000002e
0x0000002f
0x00000030
0x00000031
0x00000032
0x00000033
0x00000034
0x00000035



Button Press Counting

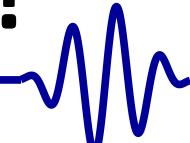
Sometimes the counter jumped significantly, instead of counting up by one



Did you see the jump by 11?

That's not right either

This might take some work to understand



Lesson Overview
Last Lesson
Review
 What
 ▷ happened?
Logic takes time
Setup and Hold
Caving Analogy
No margin
Asynchronous Input
2FF Sync
Bouncing
Debouncing
FSM
Timer
Simulation
Co-Simulation
Exercise
Formal Methods
Conclusion

```
0x00000209  
0x0000020a  
0x0000020b  
0x0000020c  
0x0000020d  
0x0000020e  
0x0000020f  
0x00000210  
0x00000211  
0x00000212  
0x00000213  
0x00000214  
0x00000215  
0x00000214  
0x00000215  
0x00000214  
0x00000215  
0x00000216  
0x00000217  
0x00000218  
0x00000219  
0x0000021a  
0x0000021b
```

Button Press Counting

Counting backwards is definitely
not what I expected!



What's going on?

Our design worked in simulation,
it passed formal verification,
it shouldn't be doing this!

Now I'm really confused! What happened?



Lesson Overview

Last Lesson

Review

What happened?

▷ Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

Co-Simulation

Exercise

Formal Methods

Conclusion

To understand what happened, you need to understand that . . .

- Logic takes time
- It takes time to go through a logic gate
- It takes time to move about the chip

All this work must be done in time for the next clock



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

▷ Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

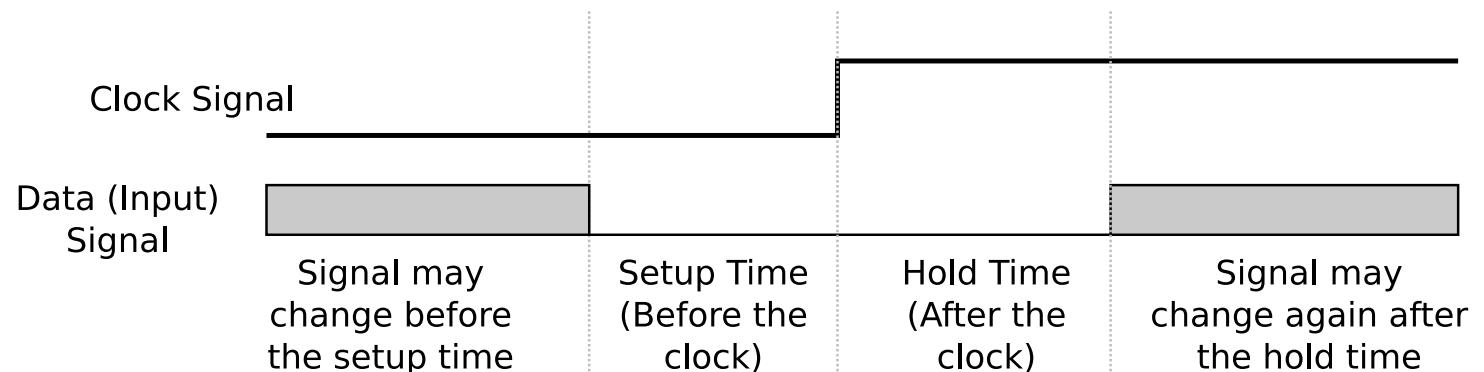
Co-Simulation

Exercise

Formal Methods

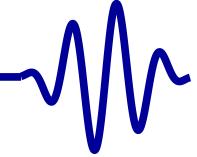
Conclusion

Flip-Flops (FFs) (a.k.a. registers or **regs**) have two requirements



1. The incoming data must be constant for a *setup period* of time before the clock edge
2. It must also be constant for a *hold time* after the clock edge

If these criteria are not met, your design will not function as you expect



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

▷ Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

Co-Simulation

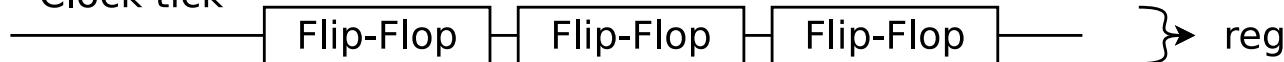
Exercise

Formal Methods

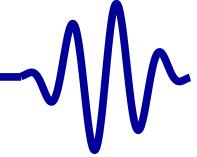
Conclusion

I like to explain clocks using caves as an analogy

Clock tick



It starts with the clock, and the FFs set using that clock



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

▷ Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

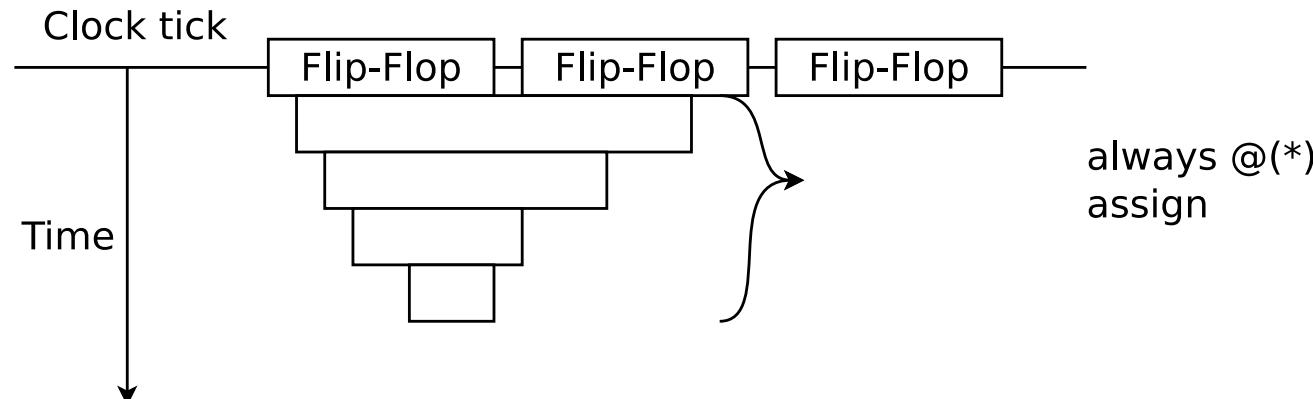
Co-Simulation

Exercise

Formal Methods

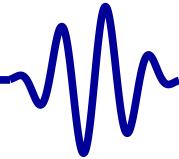
Conclusion

I like to explain clocks using caves as an analogy



Adding logic creates stalactites

- Stalactites in this analogy are formed from **assign** statements and **always @(*)** blocks
 - Their timing is derived from the last clock tick



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

▷ Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

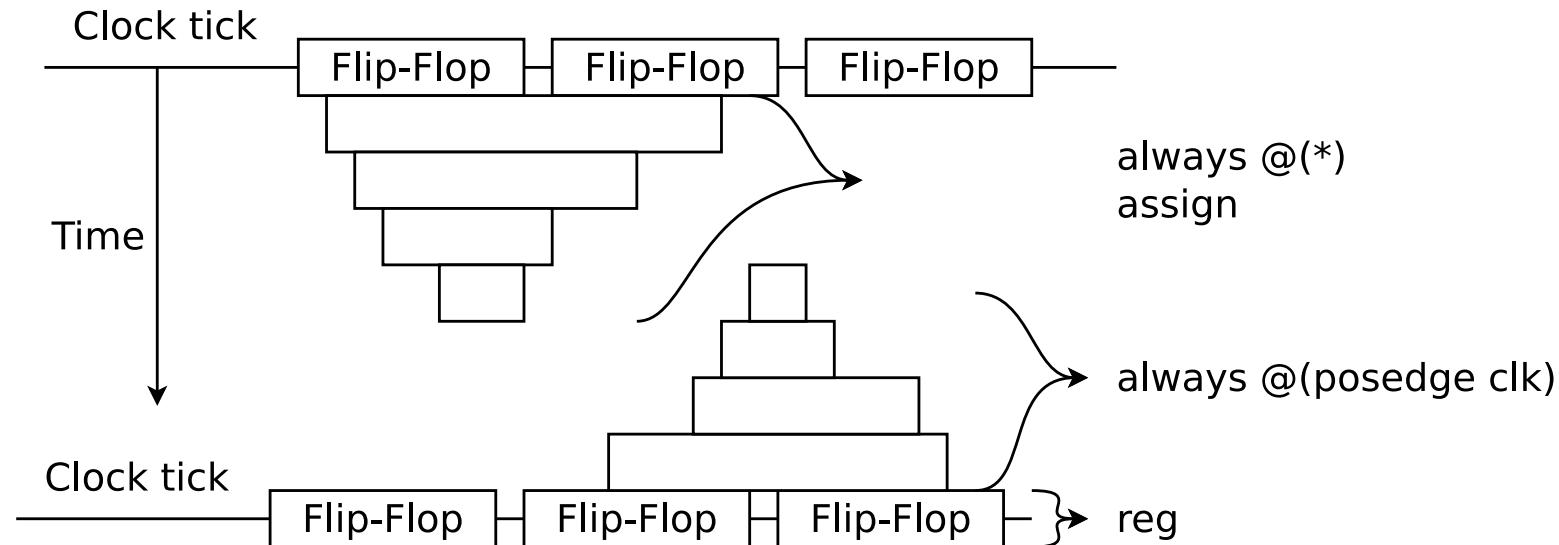
Co-Simulation

Exercise

Formal Methods

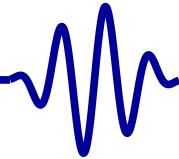
Conclusion

I like to explain clocks using caves as an analogy



Adding logic creates stalactites and stalagmites

- Stalactites in this analogy are formed from **assign** statements and **always @(*)** blocks
- Stalagmites are formed from **always @(posedge i_clk)** blocks
 - Their timing is derived from the next clock tick



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

▷ Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

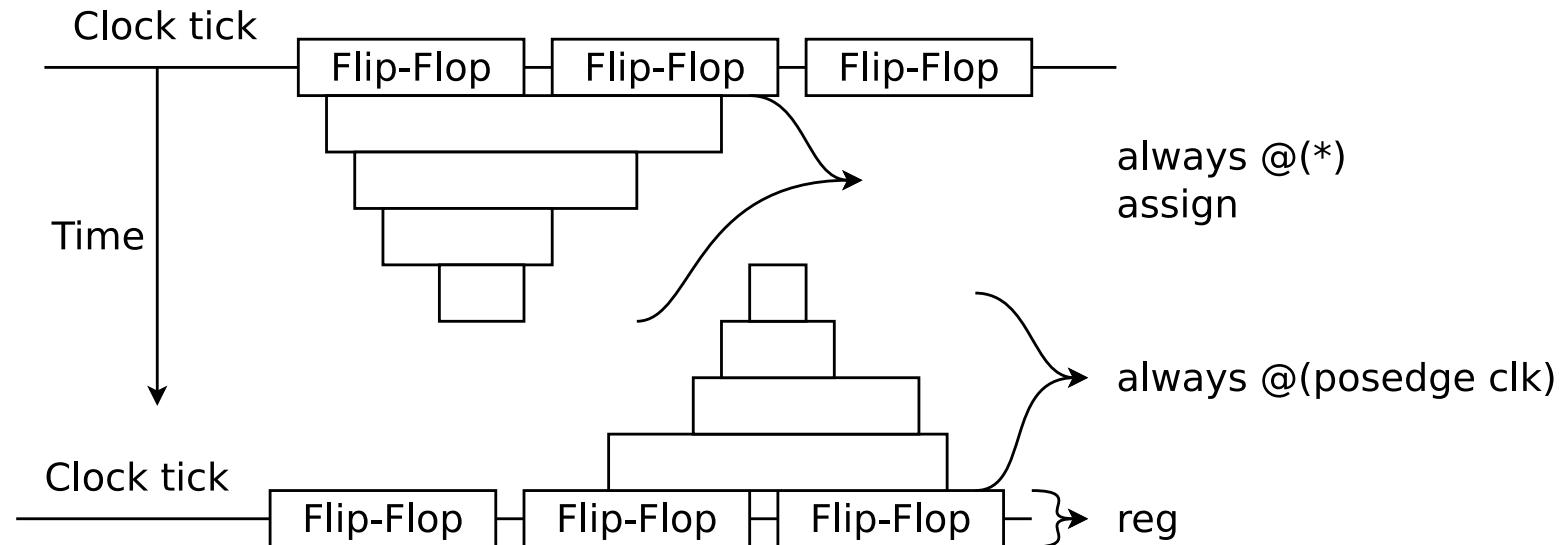
Co-Simulation

Exercise

Formal Methods

Conclusion

I like to explain clocks using caves as an analogy



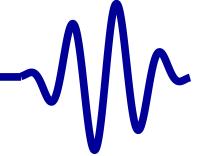
Your goal as the designer is to make certain that there's extra space between stalagmites and the stalactites

- This is your margin
- You need this margin for success

Did we guarantee any margin in our button press design?



What happened



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

▷ Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

Co-Simulation

Exercise

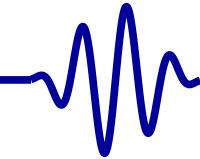
Formal Methods

Conclusion

For reference, here was the basic problematic code:

```
initial o_count = 0;
always @(posedge i_clk)
  if (i_reset)
    o_count <= 0;
  else if ((i_btn)&&(!last_btn))
    o_count <= o_count + 1'b1;
```

See the problem?



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

▷ No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

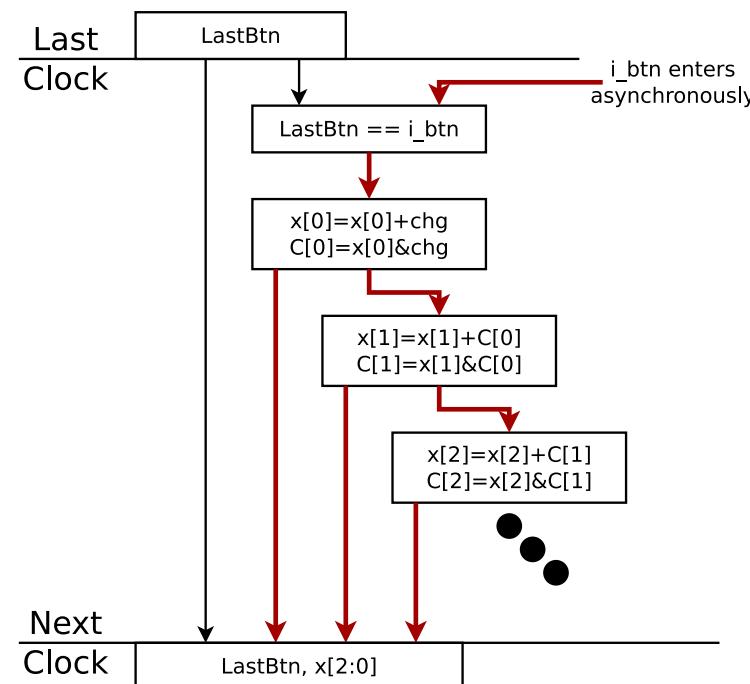
Co-Simulation

Exercise

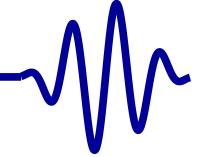
Formal Methods

Conclusion

In our last design, . . .



- Timing analysis was based upon the time between FFs
- The 32-bit carry chain stretched out the logic
- The high clock rate I used just made this worse



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

▷ No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

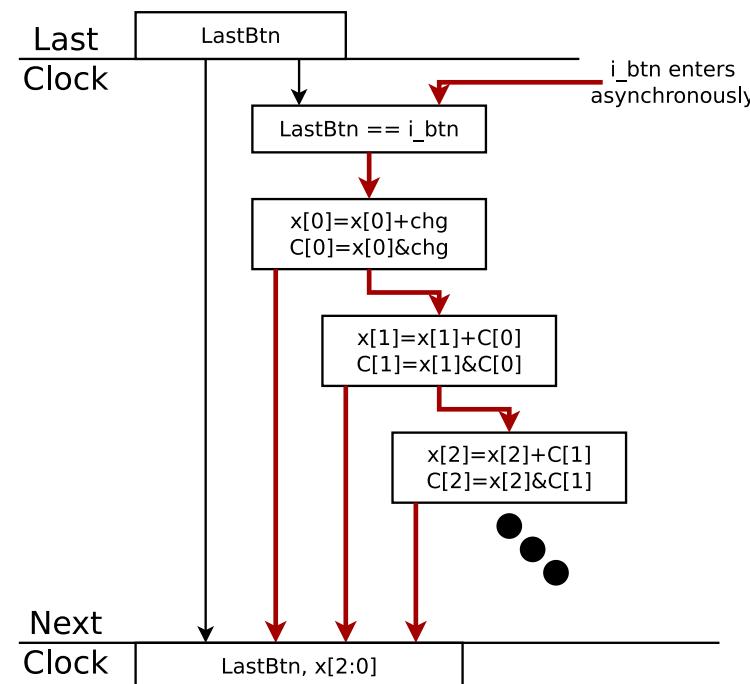
Co-Simulation

Exercise

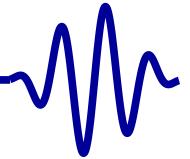
Formal Methods

Conclusion

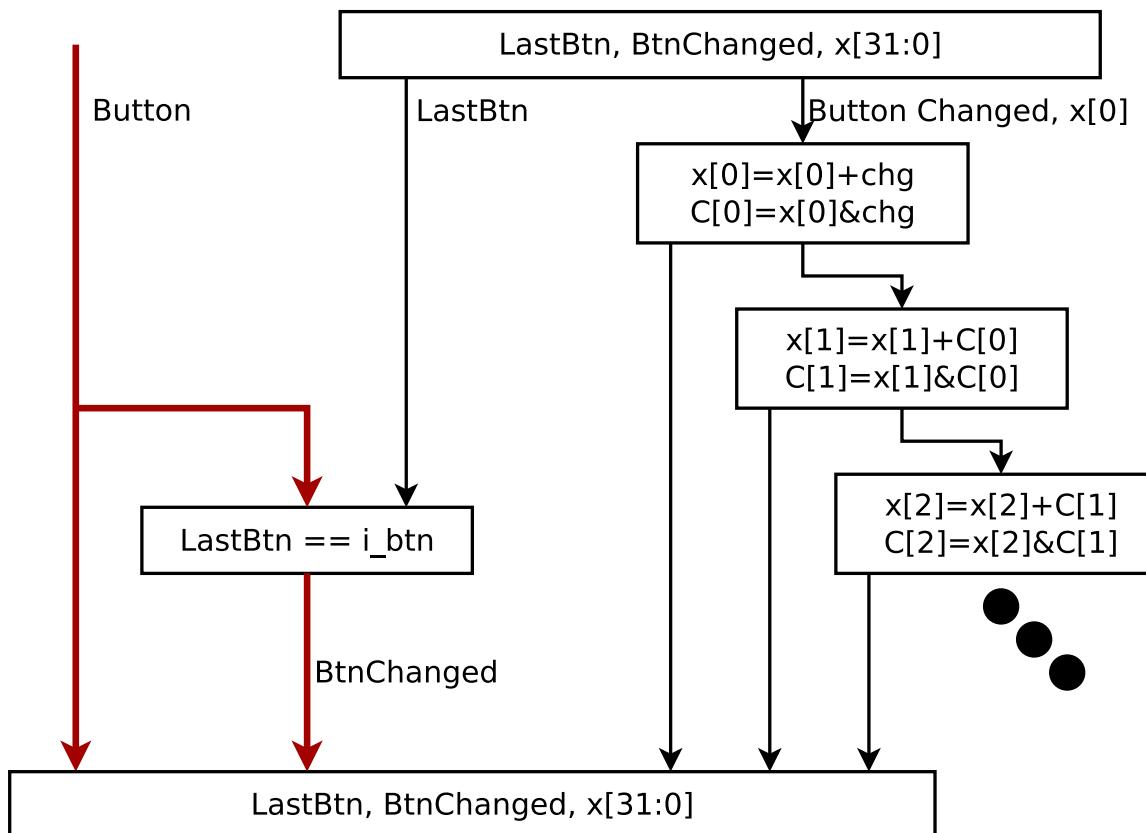
In our last design, . . .



We did *nothing* to guarantee the button press plus our logic would fit between two clock ticks with margin left over

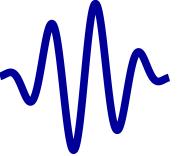


Lesson Overview
Last Lesson Review
What happened?
Logic takes time
Setup and Hold
Caving Analogy
▷ No margin
Asynchronous Input
2FF Sync
Bouncing
Debouncing
FSM
Timer
Simulation
Co-Simulation
Exercise
Formal Methods
Conclusion



Eliminating almost all of the logic is better

- But still not good enough
- The button input must go directly into an FF



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous

▷ Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

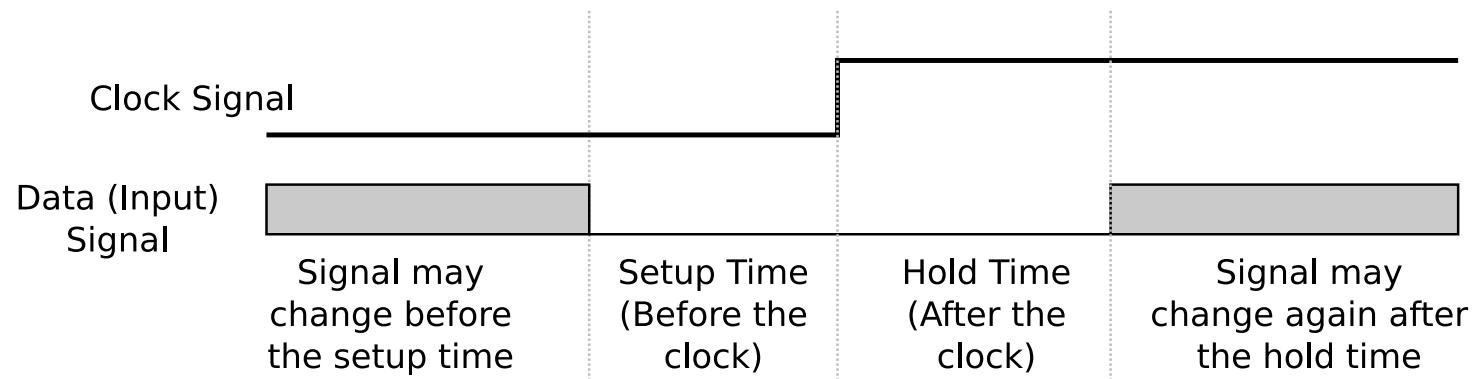
Co-Simulation

Exercise

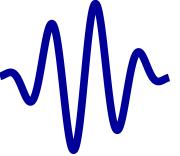
Formal Methods

Conclusion

If we can't control when the button rises, . . .



How can we ensure the setup and hold times are met?



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous

▷ Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

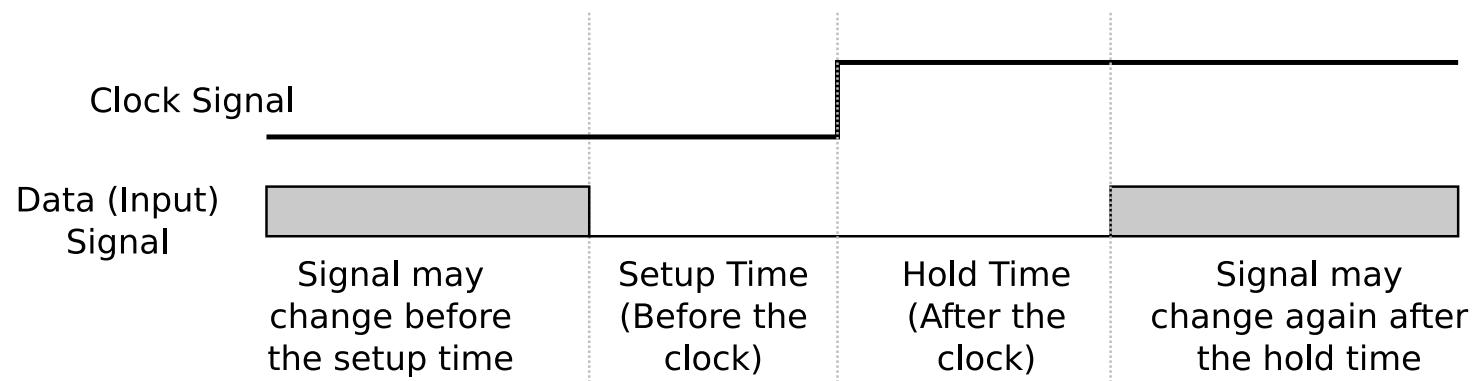
Co-Simulation

Exercise

Formal Methods

Conclusion

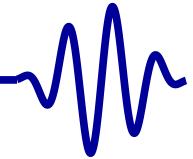
If we can't control when the button rises, . . .



How can we ensure the setup and hold times are met?

- We can't

Asynchronous Input



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous
▷ Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

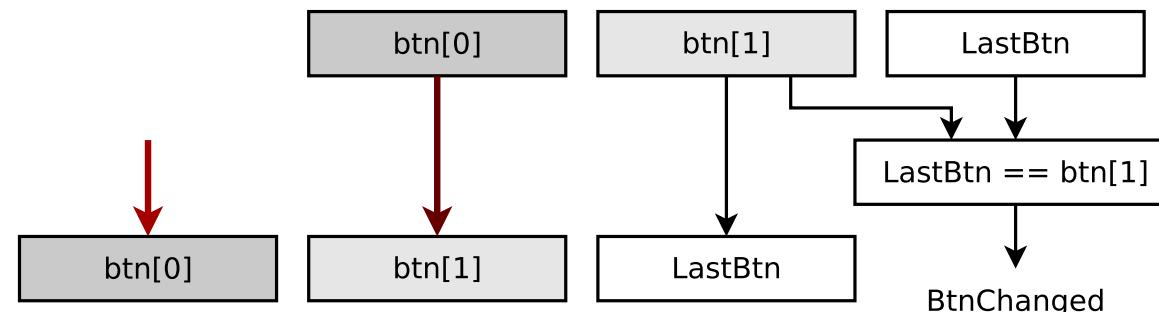
Co-Simulation

Exercise

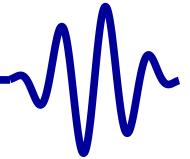
Formal Methods

Conclusion

Rule: All asynchronous inputs must go through a 2FF synchronizer



- Inputs must first go directly into a FF
 - No other logic is allowed
 - The output of this FF *may not (yet) be stable* **Metastability** is the name for when a logic value is neither zero or one. It is a rare result of not meeting setup and hold requirements



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous
▷ Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

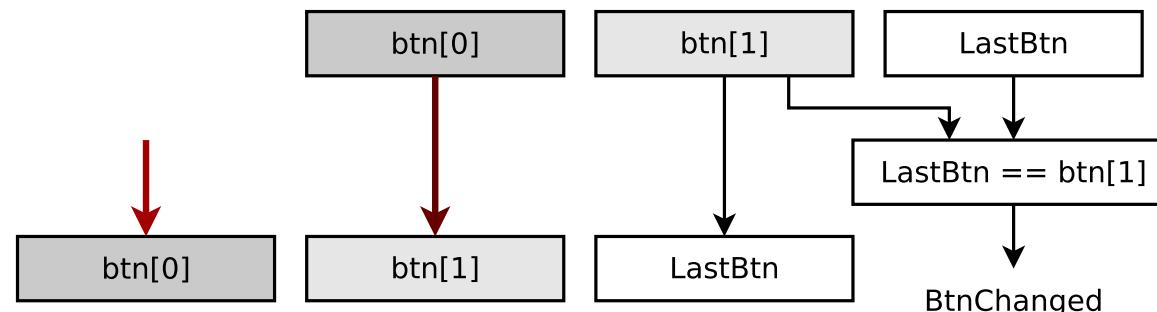
Co-Simulation

Exercise

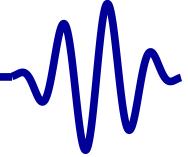
Formal Methods

Conclusion

Rule: All asynchronous inputs must go through a 2FF synchronizer



- Inputs must first go directly into a FF
- To deal with the broken setup and hold times, we go directly into a second *flip-flop*
 - This reduces the likelihood of *metastability*



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous

▷ Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

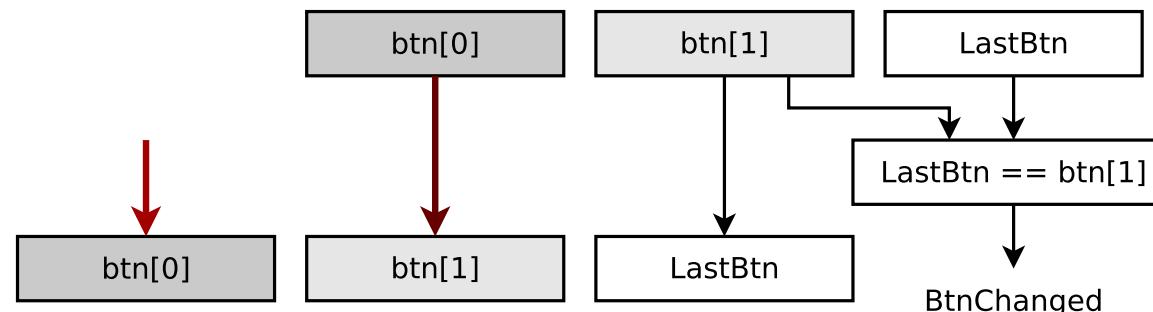
Co-Simulation

Exercise

Formal Methods

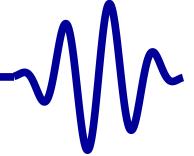
Conclusion

Rule: All asynchronous inputs must go through a 2FF synchronizer

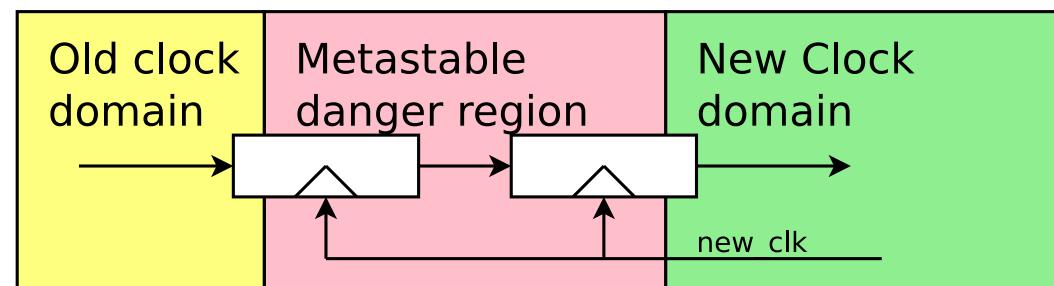


Does this apply to other asynchronous inputs besides buttons?

- Yes! If it is not synchronized to your clock, it *must* go through a **two flip-flop synchronizer**
- Won't this slow signals down? Yes, it will.
 - This is why it is important to provide a clock together with any data signal(s) in low-latency applications

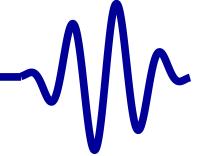
[Lesson Overview](#)[Last Lesson](#)[Review](#)[What happened?](#)[Logic takes time](#)[Setup and Hold](#)[Caving Analogy](#)[No margin](#)[Asynchronous Input](#)[▷ 2FF Sync](#)[Bouncing](#)[Debouncing](#)[FSM](#)[Timer](#)[Simulation](#)[Co-Simulation](#)[Exercise](#)[Formal Methods](#)[Conclusion](#)

This is a 2 Flip-Flop (2FF) synchronizer



Synchronizing our button input would look like

```
reg      r_btn ,  r_aux ;  
  
initial { r_btn ,  r_aux } = 2'b00;  
always @ (posedge i_clk)  
    { r_btn ,  r_aux } <= { r_aux ,  i_btn };
```



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

▷ Bouncing

Debouncing

FSM

Timer

Simulation

Co-Simulation

Exercise

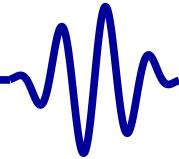
Formal Methods

Conclusion

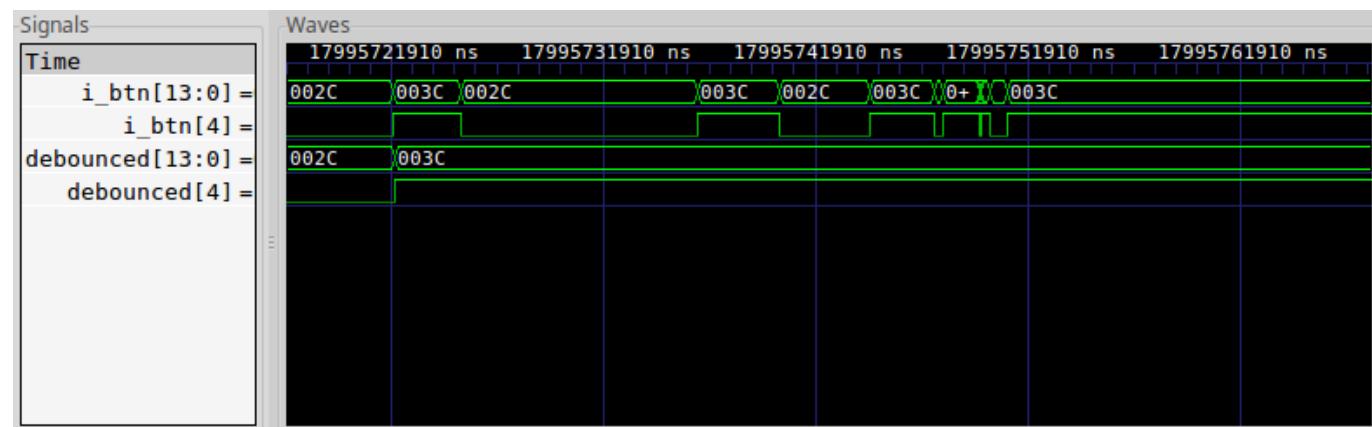
This will fix everything but the double-counts

- Often, pressing a button caused the counter to count *twice*
- The counter wouldn't skip, but one button press generated two counts

This is due to button *bouncing*

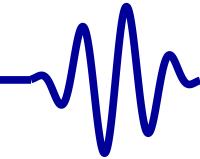
[Lesson Overview](#)[Last Lesson](#)[Review](#)[What happened?](#)[Logic takes time](#)[Setup and Hold](#)[Caving Analogy](#)[No margin](#)[Asynchronous Input](#)[2FF Sync](#)[▷ Bouncing](#)[Debouncing](#)[FSM](#)[Timer](#)[Simulation](#)[Co-Simulation](#)[Exercise](#)[Formal Methods](#)[Conclusion](#)

A trace from within our design might look like this



Look at the trace for `i_btn[4]`

- Notice how the button toggles, or “bounces” before it settles
- This is common
- It is caused by
 - Increased capacitance as the contacts come closer
 - A voltage slowly crossing through the threshold region



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

▷ Bouncing

Debouncing

FSM

Timer

Simulation

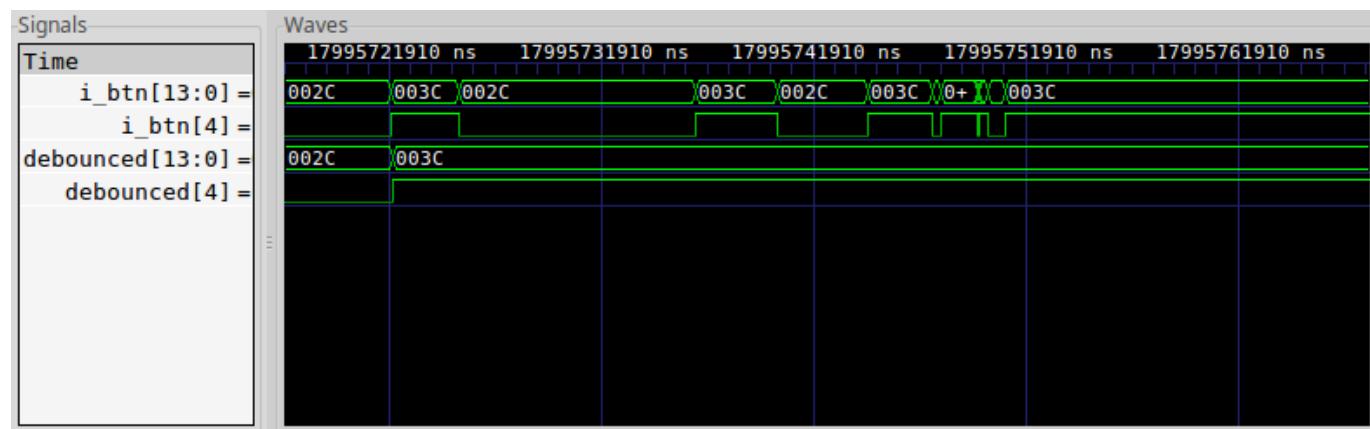
Co-Simulation

Exercise

Formal Methods

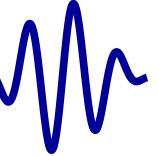
Conclusion

A trace from within our design might look like this



We'll need to simplify this “bouncing” trace

- This is called *debouncing*
- Our goal will be to produce a trace like `debounced[4]` above



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

▷ Debouncing

FSM

Timer

Simulation

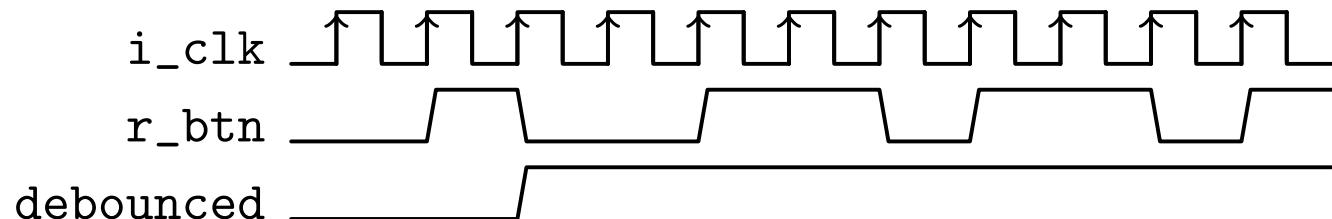
Co-Simulation

Exercise

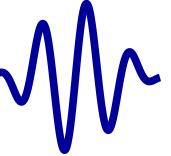
Formal Methods

Conclusion

Our goal:



- Create an output that changes when the button changes
- Not when the button bounces



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

▷ Debouncing

FSM

Timer

Simulation

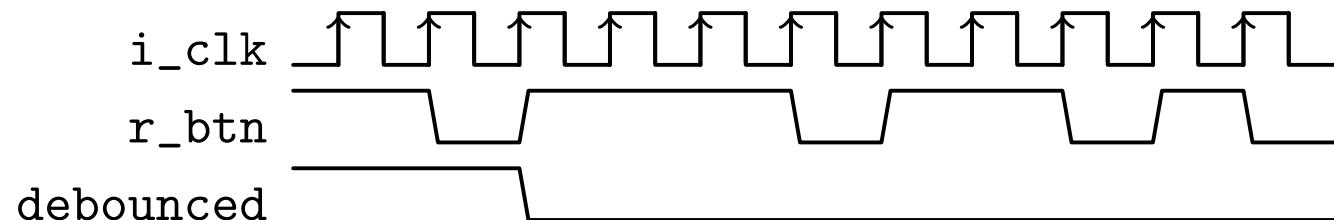
Co-Simulation

Exercise

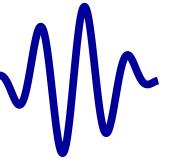
Formal Methods

Conclusion

Our goal:



This applies both to the button press as well as to its release



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

▷ Debouncing

FSM

Timer

Simulation

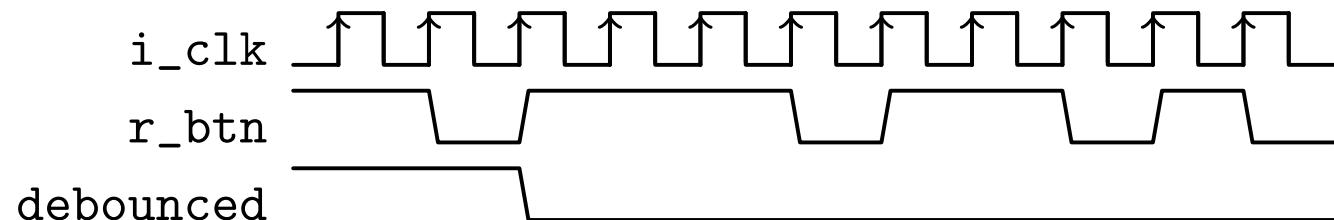
Co-Simulation

Exercise

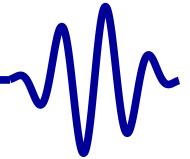
Formal Methods

Conclusion

Our goal:



This applies both to the button press as well as to its release
A state diagram might make more sense of what we need to do



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

▷ FSM

Timer

Simulation

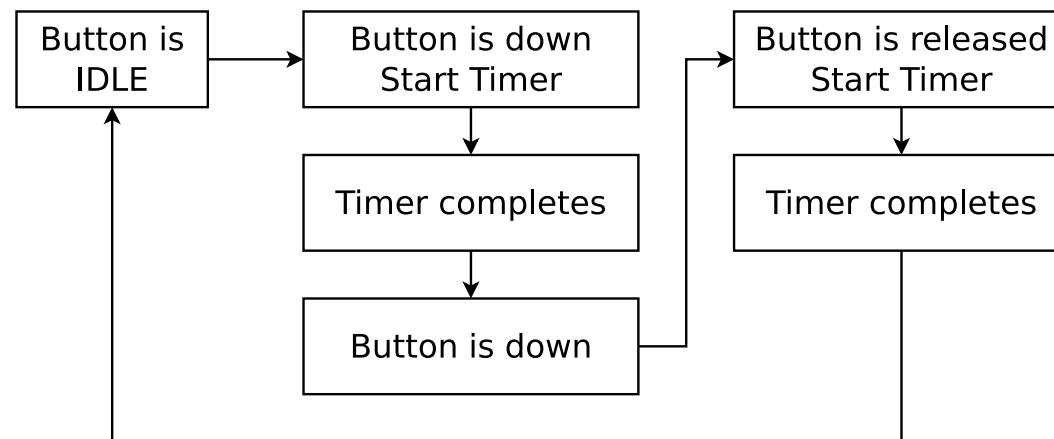
Co-Simulation

Exercise

Formal Methods

Conclusion

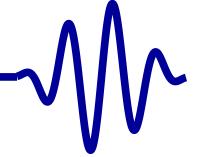
Debouncing requires a timer



We'll respond to the button any time the timer is idle

- This should be starting to look familiar

GT Timer



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

▷ Timer

Simulation

Co-Simulation

Exercise

Formal Methods

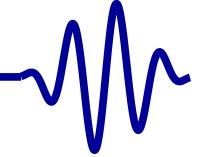
Conclusion

A button **debouncer** has three basic parts

1. The 2FF synchronizer

```
initial { r_btn, r_aux } = 0;  
always @ (posedge i_clk)  
    { r_btn, r_aux } <= { r_aux, i_btn };
```

GT Timer



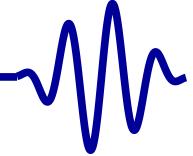
Lesson Overview
Last Lesson
Review
What happened?
Logic takes time
Setup and Hold
Caving Analogy
No margin
Asynchronous Input
2FF Sync
Bouncing
Debouncing
FSM
▷ Timer
Simulation
Co-Simulation
Exercise
Formal Methods
Conclusion

A button debouncer has three basic parts

1. The 2FF synchronizer
2. The count-down timer

```
initial timer = 0;
always @ (posedge i_clk)
  if (timer != 0)
    timer <= timer - 1;
  else if (r_btn != o_debounced)
    timer <= TIME_PERIOD-1;
```

GT Timer



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

▷ Timer

Simulation

Co-Simulation

Exercise

Formal Methods

Conclusion

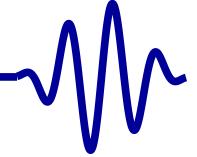
A button debouncer has three basic parts

1. The 2FF synchronizer
2. The count-down timer
3. The output

```
always @ (posedge i_clk)
  if (timer == 0)
    o_debounced <= r_btn;
```

This looks simple enough. Now, how to verify it?

GT Simulation



Lesson Overview
Last Lesson
Review
What happened?
Logic takes time
Setup and Hold
Caving Analogy
No margin
Asynchronous Input
2FF Sync
Bouncing
Debouncing
FSM
Timer
▷ Simulation
Co-Simulation
Exercise
Formal Methods
Conclusion

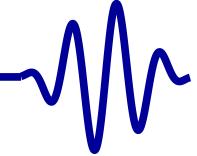
The problem is that our *simulated* button never bounced

- If we can simulate a button bouncing, we'll gain some confidence that our debouncer will work
- Perhaps if we toggled the button input randomly for some period of time, both
 - Following a button press, and
 - Following the button's release
- The simulated button would then stop toggling
 - Remaining in its pressed or released state

Making sure our simulation matches our hardware is an important and critical part of design!



Co-Simulation



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

▷ Co-Simulation

Exercise

Formal Methods

Conclusion

A button co-simulator should ...

- Be able to be pressed

```
class BUTTONSIM {  
    // ...  
    void press(void);
```

- Be able to be released

```
void release(void);
```

- Bounce following any press or release

```
int operator()(void);
```

```
}
```

Let's build out these methods



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

▷ Co-Simulation

Exercise

Formal Methods

Conclusion

Our button class will have two state variables and a constant

```
#define TIME_PERIOD 50000 // 1/2 ms at 10ns
class BUTTONSIM {
    int m_state, m_timeout;
public:
    BUTTONSIM(void) {
        // Start with the button up
        m_state = 0; // Not pressed
        //
        // And begin stable, i.e.
        m_timeout=0;
    } // ...
```

- `m_state` is the current state of the button
- `m_timeout` is a count-down timer. When it reaches zero, our button's value will be stable



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

▷ Co-Simulation

Exercise

Formal Methods

Conclusion

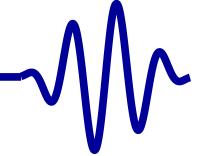
When a button is pressed, we'll change the state and set a timer

```
class BUTTONSIM {  
    // ...  
    void press(void) {  
        m_state = 1; // i.e. down  
        m_timeout = TIME_PERIOD;  
    }  
}
```

The timer will tell us when to stop bouncing



Sim Release



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

▷ Co-Simulation

Exercise

Formal Methods

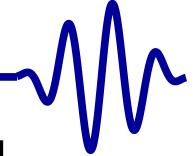
Conclusion

Button release is nearly identical

```
class BUTTONSIM {  
    // ...  
    void release(void) {  
        m_state = 0; // i.e. released  
        m_timeout = TIME_PERIOD;  
    }  
}
```



Sim Release



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

▷ Co-Simulation

Exercise

Formal Methods

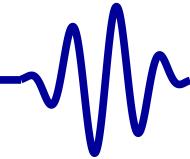
Conclusion

We can also support a test to see if the button is pressed

```
class BUTTONSIM {  
    // ...  
    bool     pressed(void) {  
        return m_state;  
    }  
}
```

While this wasn't part of our initial design outline,

- We are going to need this method below



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

▷ Co-Simulation

Exercise

Formal Methods

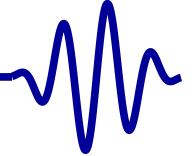
Conclusion

Now, let's make our button bounce

```
int BUTTONSIM::operator()(void) {
    if (m_timeout > 0) // Always count down
        m_timeout--;
    if (m_timeout == TIME_PERIOD-1) {
        // Return any new button
        // state accurately and
        // immediately
        return m_state;
    } else if (m_timeout > 0) {
        // Until we become stable
        // Bounce!
        return rand()&1;
    }
    // Else the button has settled
    return m_state;
}
```



Simulation



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

▷ Co-Simulation

Exercise

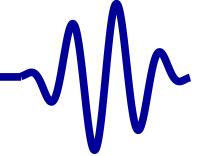
Formal Methods

Conclusion

Adding this to our simulation requires

- Declaring our button

```
BUTTONSIM *btn ;
```



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

▷ Co-Simulation

Exercise

Formal Methods

Conclusion

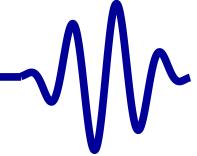
Adding this to our simulation requires

- Declaring our button, and allocating a button object

```
BUTTONSIM *btn ;  
// ...  
btn = new BUTTONSIM() ;
```



Simulation



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

▷ Co-Simulation

Exercise

Formal Methods

Conclusion

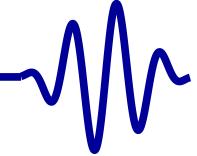
Adding this to our simulation requires

- Declaring our button, and allocating a button object
- Adjusting our button press scheme

```
do {  
    int chv;  
    chv = getch();  
    if (chv == 'r')  
        btn->release();  
    else if ((chv != ERR)  
             && (!btn->pressed())) {  
        keypasses++;  
        btn->press();  
    }  
    // ...  
} while (!done);
```



Simulation



Lesson Overview
Last Lesson
Review
What happened?
Logic takes time
Setup and Hold
Caving Analogy
No margin
Asynchronous Input
2FF Sync
Bouncing
Debouncing
FSM
Timer
Simulation
▷ Co-Simulation
Exercise
Formal Methods
Conclusion

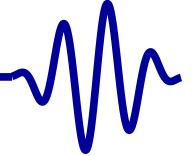
Adding this to our simulation requires

- Declaring our button
- Adjusting our button press scheme
- Adding it to our list of co-sim calls

```
for(int k=0; k<1000; k++) {  
    // Advance the Verilator logic  
    tb->tick();  
    // Serial-port Co-sim  
    (*uart)(tb->m_core->o_uart_tx);  
    // Button co-sim  
    m_core->i_btn = (*btn)();  
}
```



Exercise



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

Co-Simulation

▷ Exercise

Formal Methods

Conclusion

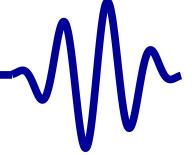
Your turn!

Build and experiment with the simulation

- Create a trace showing the button bouncing
- Make your Verilog timeout longer than the C++
TIME_PERIOD.



Exercise



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

Co-Simulation

▷ Exercise

Formal Methods

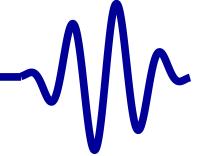
Conclusion

Now build this on your hardware. Does it work?

- Do you ever get multiple counts for a single press?
- Does the counter ever jump?



Formal Methods



Lesson Overview

Last Lesson

Review

What happened?

Logic takes time

Setup and Hold

Caving Analogy

No margin

Asynchronous Input

2FF Sync

Bouncing

Debouncing

FSM

Timer

Simulation

Co-Simulation

Exercise

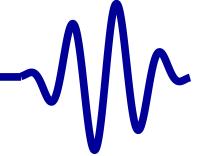
▷ Formal Methods

Conclusion

We haven't discussed formal methods this lesson

- Our debouncing circuit can still be verified
 - Although there's not much there
 - You should have an idea of how to do this from our last lessons
- What formal properties might you include to verify this design?

Conclusion



What did we learn this lesson?

- *Always send asynchronous inputs through a 2FF synchronizer before using them*
 - Failing to do this can result in some inexplicable behavior
 - Simulation and implementation might not match
 - ▷ Bugs of this kind can be very hard to find and fix
- **Buttons *bounce!***
 - A basic **debouncing** circuit is another FSM
 - This time with a counter within it



Gisselquist
Technology, LLC

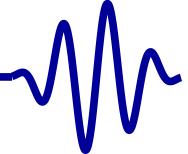
8. Using Block RAM

Daniel E. Gisselquist, Ph.D.





Lesson Overview



- ▷ Lesson Overview
- Design Goal
- On-chip RAM
- Block RAM Rules
- Initializing Memory
- Hex file
- Reset
- Overview
- Restarting
- Mem Address
- Serial Port
- Next Steps
- Formal Verification
- Reset Assertions
- Cover
- Exercise!
- Hardware Bonus
- Conclusion

Three types of FPGA memory

- Flip-flops
- Distributed RAM
- Block RAM

Block RAM is special within an FPGA

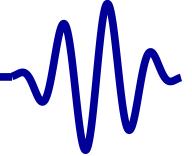
- It is fast and abundant
- Requires one clock to access
- Can only be initialized at startup
- Yet there are some logic requirements to use it

Objectives

- Be able to create block RAM resources
- Understand the requirements of block RAMs
- Learn how to verify a component containing a block RAM



Lesson Overview



- ▷ Lesson Overview
- Design Goal
- On-chip RAM
- Block RAM Rules
- Initializing Memory
- Hex file
- Reset
- Overview
- Restarting
- Mem Address
- Serial Port
- Next Steps
- Formal Verification
- Reset Assertions
- Cover
- Exercise!
- Hardware Bonus
- Conclusion

Let's also take a quick look at synchronous resets

- Learn the two types of resets
- Reset logic follows one of two forms

Extra Objectives

- Know the two forms of synchronous reset logic
- Know how to verify a design with a synchronous reset



Design Goal



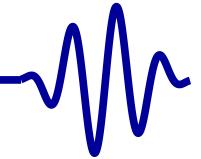
Lesson Overview
▷ Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

Let's rebuild our Hello World design, but make the message longer

- We'll use a memory to capture our longer message

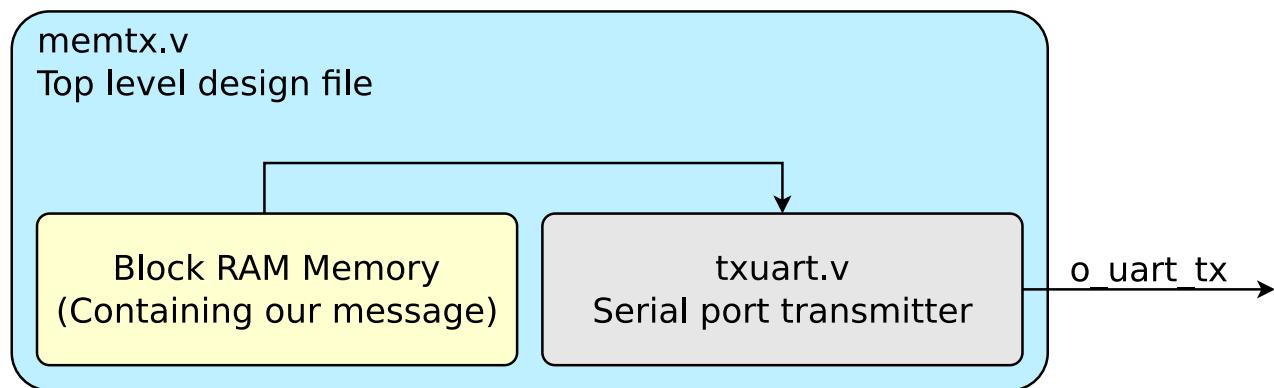
```
% ./memtx_tb
=====
| Psalm 1
|
| Blessed is the man that walketh not in the counsel of the ungodly, nor
| standeth in the way of sinners, nor sitteth in the seat of the scornful.
| But his delight is in the law of the LORD; and in his law doth he meditate
| day and night.
| And he shall be like a tree planted by the rivers of water, that bringeth
| forth his fruit in his season; his leaf also shall not wither; and
| whatsoever he doeth shall prosper.
| The ungodly are not so: but are like the chaff which the wind driveth
| away.
| Therefore the ungodly shall not stand in the judgment, nor sinners in the
| congregation of the righteous.
| For the LORD knoweth the way of the righteous: but the way of the ungodly
| shall perish.
|
=====
Simulation complete
%
```

- Then read from this memory, and . . .
- Transmit it out the serial port



Lesson Overview
▷ Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

Here's a basic block diagram



- We'll re-use the serial port transmitter, `txuart.v`
- We'll capture our message in a block RAM, and ...
- We'll use a top level module to coordinate it all, `memtx.v`
 - We'll infer the block RAM within our `memtx.v` design

But what is on-chip RAM and how shall we declare and use it?



On-Chip Memory



Lesson Overview
Design Goal
▷ On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

There's a special type of declaration for memory in Verilog:

```
reg [W-1:0] ram [0 : MEMLN - 1];
```

- This defines a memory of MEMLN elements,
where each element is w bits long
- Verilog allows MEMLN to be anything
- Practically, MEMLN must only ever be a power of two, 2^N , in
order to avoid simulation/hardware mismatch
- I tend to define my memories as

```
reg [W-1:0] ram [0:(1<<LGMEMSZ ) - 1];
```

- This forces the power of two requirement
- LGMEMSZ can also be used as the width of the address



Declaring On-Chip Memory



Lesson Overview
Design Goal
▷ On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

There's a special type of declaration for memory in Verilog:

```
reg [W-1:0] ram [0:(1<<LGMEMSZ)-1];
```

The synthesis tool will decide how to implement this

- Flip-Flops
 - Useful for small numbers of bits
 - Very inefficient for implementing memory on an FPGA
- Distributed RAM
 - Useful for small, localized RAM needs
 - Typically allocated one-bit at a time for memory sizes of 2^6 elements (Ex. Xilinx's SLICEM)
- Block RAM
 - Useful for larger and wider RAM needs
 - Using block RAM requires that you follow special rules



Block RAM Rules



Lesson Overview

Design Goal

On-chip RAM

Block RAM

▷ Rules

Initializing Memory

Hex file

Reset

Overview

Restarting

Mem Address

Serial Port

Next Steps

Formal Verification

Reset Assertions

Cover

Exercise!

Hardware Bonus

Conclusion

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block

```
always @ (posedge i_clk)
if (write)
    ram[write_addr] <= write_value;
```

```
always @ (posedge i_clk)
if (read)
    read_value <= ram[read_addr];
```

Block RAM Rules



Lesson Overview

Design Goal

On-chip RAM

Block RAM

▷ Rules

Initializing Memory

Hex file

Reset

Overview

Restarting

Mem Address

Serial Port

Next Steps

Formal Verification

Reset Assertions

Cover

Exercise!

Hardware Bonus

Conclusion

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block
2. RAM can only be initialized once

```
always @(posedge i_clk)
if (i_reset)
begin
    // This is illegal! Block
    // RAM cannot be re-initialized
    for(i=0; i<ramsize; i=i+1)
        ram[i] <= 0;
end else if (i_stb)
    ram[addr] <= value;
```

This is often an unexpected frustration for beginners.

- The solution is to rewrite your algorithm so you don't need to do this



Lesson Overview

Design Goal

On-chip RAM

Block RAM

▷ Rules

Initializing Memory

Hex file

Reset

Overview

Restarting

Mem Address

Serial Port

Next Steps

Formal Verification

Reset Assertions

Cover

Exercise!

Hardware Bonus

Conclusion

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block
2. RAM can only be initialized once
3. Don't put a RAM access in a cascaded if

```
always @(posedge i_clk)
  if (A)
    value <= // something;
  else if (B)
    value <= // something else;
  else if (C)
    // Don't do this either!
    value <= ram[addr];
  else if (D)
    // logic continues ...
```

Such logic often ends up being replaced by flip-flops



Block RAM Rules



Lesson Overview

Design Goal

On-chip RAM

Block RAM

▷ Rules

Initializing Memory

Hex file

Reset

Overview

Restarting

Mem Address

Serial Port

Next Steps

Formal Verification

Reset Assertions

Cover

Exercise!

Hardware Bonus

Conclusion

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block
2. RAM can only be initialized once
3. Don't put a RAM access in a cascaded if
4. Don't put a RAM in a port list

```
// Don't do this
output reg [W-1:0] ram [0:(1<<LGMEMSZ)-1];
```

Block RAM Rules



Lesson Overview

Design Goal

On-chip RAM

Block RAM

▷ Rules

Initializing Memory

Hex file

Reset

Overview

Restarting

Mem Address

Serial Port

Next Steps

Formal Verification

Reset Assertions

Cover

Exercise!

Hardware Bonus

Conclusion

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block
2. RAM can only be initialized once
3. Don't put a RAM access in a cascaded if
4. Don't put a RAM in a port list
5. Don't put a RAM in a block with other things

```
// Many synthesizers will turn this into FFs
always @(posedge i_clk)
  if (write_enable)
    begin
      B <= // some logic;
      C <= // something else;
      ram[addr] <= value;
    end
```

Block RAM Rules



Lesson Overview

Design Goal

On-chip RAM

Block RAM

▷ Rules

Initializing Memory

Hex file

Reset

Overview

Restarting

Mem Address

Serial Port

Next Steps

Formal Verification

Reset Assertions

Cover

Exercise!

Hardware Bonus

Conclusion

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block
2. RAM can only be initialized once
3. Don't put a RAM access in a cascaded if
4. Don't put a RAM in a port list
5. Don't put a RAM in a block with other things

Some synthesizers/hardware allow byte enables

```
always @(posedge i_clk)
if (write_enable)
begin
    if (en[1])
        ram[addr][15:8] <= value[15:8];
    if (en[0])
        ram[addr][ 7:0] <= value[7:0];
end
```



Block RAM Rules



Lesson Overview

Design Goal

On-chip RAM

Block RAM

▷ Rules

Initializing Memory

Hex file

Reset

Overview

Restarting

Mem Address

Serial Port

Next Steps

Formal Verification

Reset Assertions

Cover

Exercise!

Hardware Bonus

Conclusion

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block
2. RAM can only be initialized once
3. Don't put a RAM access in a cascaded if
4. Don't put a RAM in a port list
5. Don't put a RAM in a block with other things

Some synthesizers/hardware allow write-through

- Where the value being written may be read on the same clock

```
always @(posedge i_clk)
begin
    if (write_enable)
        mem [addr] = wvalue ;
        rvalue = mem [addr];
end // Note the non-blocking notation !
```



Lesson Overview

Design Goal

On-chip RAM

Block RAM

▷ Rules

Initializing Memory

Hex file

Reset

Overview

Restarting

Mem Address

Serial Port

Next Steps

Formal Verification

Reset Assertions

Cover

Exercise!

Hardware Bonus

Conclusion

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block
2. RAM can only be initialized once
3. Don't put a RAM access in a cascaded if
4. Don't put a RAM in a port list
5. Don't put a RAM in a block with other things

Some synthesizers/hardware allow write-through

- Where the value being written may be read on the same clock
 - This would be ideal for a CPU register file
- It's not uniformly supported across our chosen tools/vendors
- Know your hardware, synthesizer, and simulator
- We'll pretend this feature does not exist in this tutorial



Block RAM Rules



Lesson Overview

Design Goal

On-chip RAM

Block RAM

▷ Rules

Initializing Memory

Hex file

Reset

Overview

Restarting

Mem Address

Serial Port

Next Steps

Formal Verification

Reset Assertions

Cover

Exercise!

Hardware Bonus

Conclusion

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block
2. RAM can only be initialized once
3. Don't put a RAM access in a cascaded if
4. Don't put a RAM in a port list
5. Don't put a RAM in a block with other things

If you fail to follow these rules,

You might get something other than block RAM, or

Your design might fail to synthesize entirely

This is a common reason for synthesis failure



Lesson Overview

Design Goal

On-chip RAM

Block RAM

▷ Rules

Initializing Memory

Hex file

Reset

Overview

Restarting

Mem Address

Serial Port

Next Steps

Formal Verification

Reset Assertions

Cover

Exercise!

Hardware Bonus

Conclusion

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block
2. RAM can only be initialized once
3. Don't put a RAM access in a cascaded if
4. Don't put a RAM in a port list
5. Don't put a RAM in a block with other things

If you fail to follow these rules,

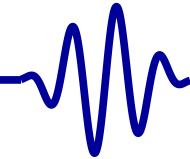
You might get something other than block RAM, or

Your design might fail to synthesize entirely

This is a common reason for synthesis failure

- Always keep an eye on your RAM and LUT usages
- Something out of bounds may be caused by this

If you suspect this is a problem, break your design into smaller and smaller components until you find out what's going on



Lesson Overview

Design Goal

On-chip RAM

Block RAM

▷ Rules

Initializing Memory

Hex file

Reset

Overview

Restarting

Mem Address

Serial Port

Next Steps

Formal Verification

Reset Assertions

Cover

Exercise!

Hardware Bonus

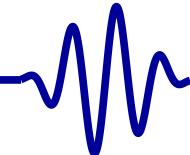
Conclusion

When is distributed RAM used?

- If the memory size is small (32 elements or less)
- If the memory is read without a clock

```
always @(*)  
    rvalue = mem[addr];  
// Or equivalently  
assign rvalue = mem[addr];
```

- Obviously, only if the device has distributed RAM
 - iCE40 devices have no distributed RAM



How might we initialize our RAM?

- We could use assignments within an initial block

```
reg [31:0] ram [0:8191];  
  
integer k;  
initial begin  
    for (k=0; k <8192; k=k+1)  
        ram[k] = 0;  
    // We can also set specific values  
    ram[5] = 7;  
    ram[8190] = 5;  
    // etc.  
end
```

- When using Xilinx's ISE, this is the only way I've managed to initialize RAM



Initializing Memory



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
▷ Initializing
Memory
Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

How might we initialize our RAM?

- We could use assignments within an initial block
 - Verilator (currently) complains about non-blocking **initial** assignments

```
// This will generate a Verilator warning
initial ram[8190] <= 5;
```

- Yosys (currently) complains about blocking **initial** assignments

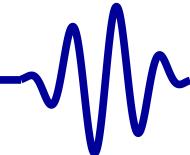
```
// This will generate a Yosys warning
initial ram[8190] = 5;
```

If you don't redefine any values, both will still work

- In this case, you may ignore the warnings



Initializing Memory



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
 Initializing
 Memory
Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

How might we initialize our RAM?

- We could use a \$readmemh function call (recommended)

```
reg [31:0] ram [0:8191];  
initial $readmemh(FILE_NAME, ram);
```

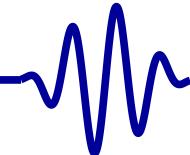
- Each word of the file FILE_NAME has format %0*x

```
012345678  
....
```

- Separate each RAM word by white space
- Number of digits is based upon the width of the RAM word
 - Our example above shows a 32-bit word
- Xilinx's ISE has a known bug that prevents \$readmemh from working. Vivado doesn't have this bug.



Initializing Memory



Lesson Overview

Design Goal

On-chip RAM

Block RAM Rules

▷ Initializing
Memory

Hex file

Reset

Overview

Restarting

Mem Address

Serial Port

Next Steps

Formal Verification

Reset Assertions

Cover

Exercise!

Hardware Bonus

Conclusion

How might we initialize our RAM?

- We could use a \$readmemh function call (recommended)

```
reg [31:0] ram [0:8191];  
initial $readmemh(FILE_NAME, ram);
```

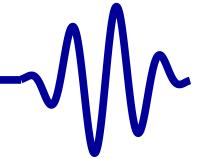
- Alternatively, lines can begin with @ (hexadecimal) addresses

```
@000000e0 2c 20 61 20 6e 65 77 20 6e 61 74 ...  
@000000f0 63 6f 6e 63 65 69 76 65 64 20 69 ...  
....
```

- This example shows a series of 8-bit characters
- Sixteen per line
- This form makes it possible to skip elements
- We'll build one of these files for our project later



Initializing Memory



Lesson Overview

Design Goal

On-chip RAM

Block RAM Rules

▷ Initializing
Memory

Hex file

Reset

Overview

Restarting

Mem Address

Serial Port

Next Steps

Formal Verification

Reset Assertions

Cover

Exercise!

Hardware Bonus

Conclusion

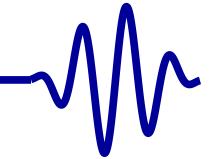
How might we initialize our RAM?

- We could use a \$readmemh function call (recommended)

```
reg [31:0] ram [0:8191];  
initial $readmemh(FILE_NAME, ram);
```

- On-chip RAM can only be initialized in an **initial** block
- Cannot re-initialize a block RAM in this fashion later without reconfiguring (i.e. reloading) the FPGA

Generating the Hex file



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
▷ Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

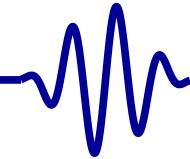
Let's generate a hex file that we can use with \$readmemh

- Use a C++ program
- We'll call this program genhex
- Much of the program is boilerplate and error checking
- We'll skip most of this boilerplate now, and instead focus on the interesting parts

You can find the entire genhex program with the course materials



Generating the Hex file



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
▷ Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

Let's build our hex file

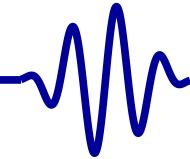
- We'll prefix each line with an address

```
int      linelen = 0;  
int      ch , addr = 0;  
  
fprintf(fout , "@%08x\u202c", addr);  
linelen = 10;
```

- Don't forget that the address begins with an @ sign



Generating the Hex file



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
▷ Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

Let's build our hex file

- We'll prefix each line with an address
- Process one character at a time

```
// Read one character from our file
while((ch = fgetc(fp))!=EOF) {
    // and process it if we read
    // a non-empty character
    // ...
}
```

Let's build our hex file

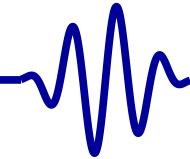
- We'll prefix each line with an address
- Process one character at a time
- The values out are simply hex characters

```
// ...
while((ch = fgetc(fp)) !=EOF) {
    fprintf(fout, "%0*x\u2074",
            (nbits+3)/4, ch & 0x0ff);
    linelen += 3;
    addr++;
    // ...
```

- We can use **nbits** to make the width generic
- In this example, **nbits** = 8 so we only need two hex digits each



Generating the Hex file



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
▷ Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

Let's build our hex file

- We'll prefix each line with an address
- Process one character at a time
- The values are just simply hex characters
- After 56 bytes, start a new line with a new address

```
while((ch = fgetc(fp))!=EOF) {  
    // ...  
    if (linelen >= 56) {  
        // New line starting with  
        // the current address  
        fprintf(fout, "\n@%08x", addr);  
        linelen += 10;  
    }  
} fprintf(fout, "\n");
```

Generating the Hex file



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
▷ Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

One task remains: adding the hexfile generation to our Makefile

- Our target is “memfile.hex”
- It depends upon **genhex**, and our text file, **psalm.txt**

```
memfile .hex : genhex psalm .txt
                . /genhex psalm .txt
```

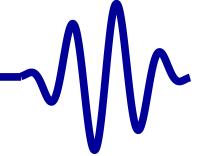
- **genhex** must also be built
 - It depends upon **genhex.cpp**

```
genhex : genhex .cpp
        g++ genhex .cpp -o genhex
```

- Don’t forget to make sure **memfile.hex** is built before it’s needed

Voila! A hex file that will change anytime **psalm.txt** does

Using the hexfile



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
▷ Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

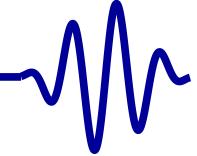
After all that work,

- We can now declare and initialize our memory

```
reg [7:0] tx_memory [0:2047];  
  
initial $readmemh("memfile.hex", tx_memory);
```

Next, we'll need to discuss resets

Reset



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
▷ Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

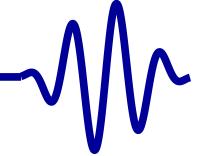
There are two types of resets

- Asynchronous resets

```
initial tx_index = 0;  
always @(posedge i_clk or negedge i_areset)  
  if (i_areset)  
    tx_index <= 0;  
  else begin  
    // The rest of your logic  
  end
```

- These are more complex than their synchronous counterparts
- Often require being asserted for many cycles, and
- Released on a clock edge
- Poor design can lead to radio interference triggering an internal asynchronous reset

Reset



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
▷ Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

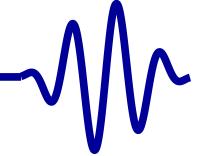
There are two types of resets

- Asynchronous resets

```
initial tx_index = 0;  
always @(posedge i_clk or negedge i_areset)  
  if (i_areset)  
    tx_index <= 0;  
  else begin  
    // The rest of your logic  
  end
```

- These are more complex than their synchronous counterparts
- Often require being asserted for many cycles, and
- Released on a clock edge
- Poor design can lead to radio interference triggering an internal asynchronous reset
 - This is bad.

Reset



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
▷ Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

There are two types of resets

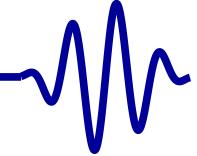
- Asynchronous resets

```
initial tx_index = 0;  
always @(posedge i_clk or negedge i_areset)  
if (i_areset)  
    tx_index <= 0;  
else begin  
    // The rest of your logic  
end
```

- These are more complex than their synchronous counterparts
- Often require being asserted for many cycles, and
- Released on a clock edge
- Poor design can lead to radio interference triggering an internal asynchronous reset
 - This is bad. We will avoid these in this tutorial



Reset



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
▷ Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

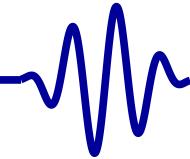
There are two types of resets

- Asynchronous resets, and
- Synchronous resets
 - These are set and released on clock tick

```
initial tx_index = 0;  
always @ (posedge i_clk)  
if (i_reset)  
    tx_index <= 0;  
else begin  
    // The rest of your logic  
end
```

- These are simple to build and use

Let's implement a synchronous reset to this design



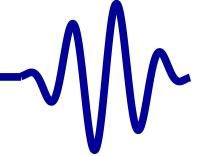
Many designs use a synchronous reset

- Values responsive to a reset should also have an **initial** value
- The **initial** value and the reset value must match

```
initial tx_index = 0;  
always @(posedge i_clk)  
if (i_reset)  
    tx_index <= 0;  
else begin  
    // The rest of your logic  
end
```

- I like this form of a reset, but
- It requires that every register set by this block gets reset as well

The original Hello World design included no reset



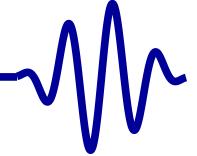
Many designs use a synchronous reset

- Values responsive to a reset should also have an **initial** value
- An alternate form of reset needs to be used if some values need to be reset within the block and others don't

```
initial tx_index = 0;  
always @(posedge i_clk)  
begin  
    // Your logic would come  
    // first, then ...  
  
    if (i_reset)  
        // Overrides the logic above  
        tx_index <= 0;  
end
```

- This is a more generic form, useful for all purposes

Synchronous Reset

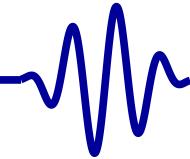


Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
▷ Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

Why might you need a synchronous reset?

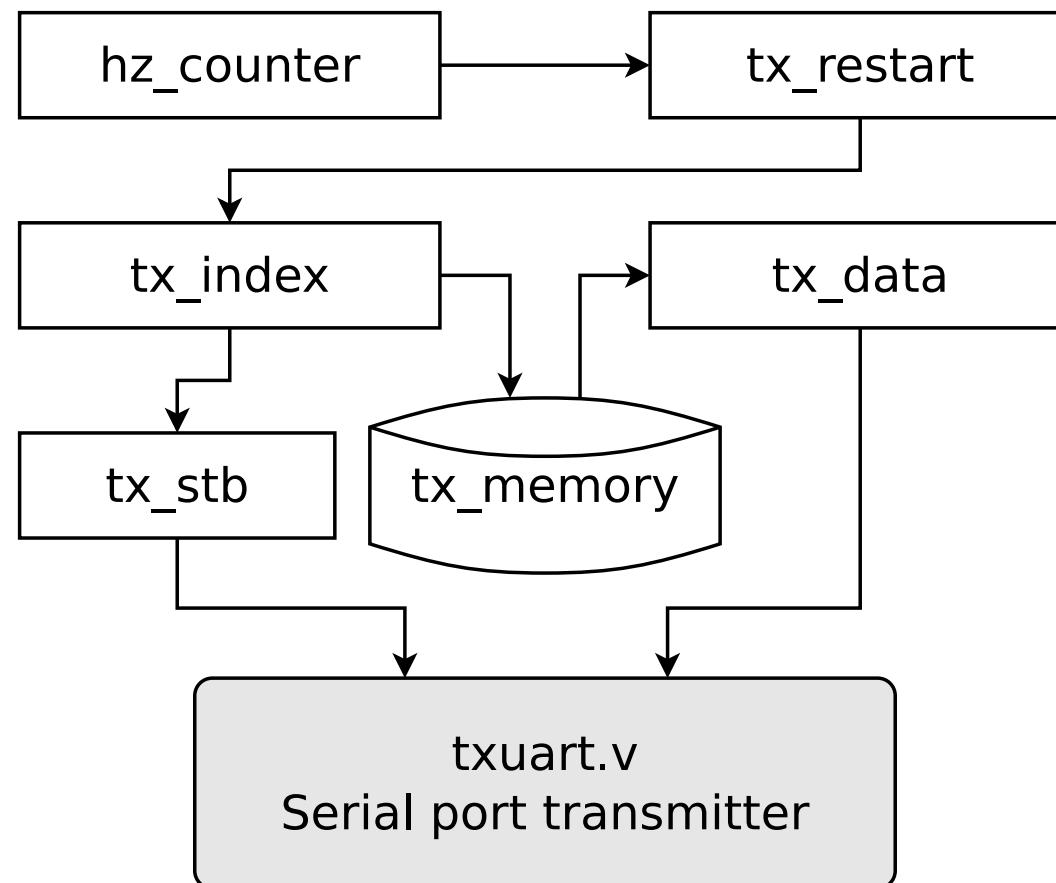
- Sometimes internal or external conditions will require a reset
 - Ex: An embedded CPU crash, or watchdog timer timeout might cause a CPU to need to be reset
- Not all technologies support **initial** values
 - For example, if you want to create FPGA+ASIC support, you design will need a reset
- Sometimes it just helps to start over
- A (debounced) button can be used to create a reset

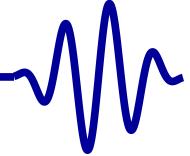
Let's use a synchronous reset in our design



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
▷ Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

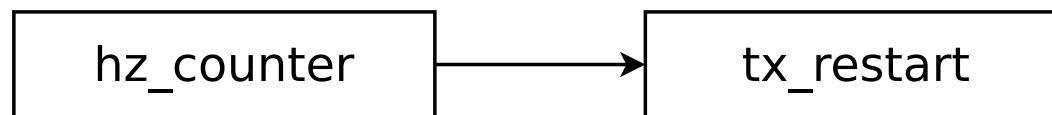
Here's how our design is going to work





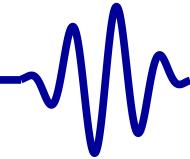
Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
▷ Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

Here's how our design is going to work



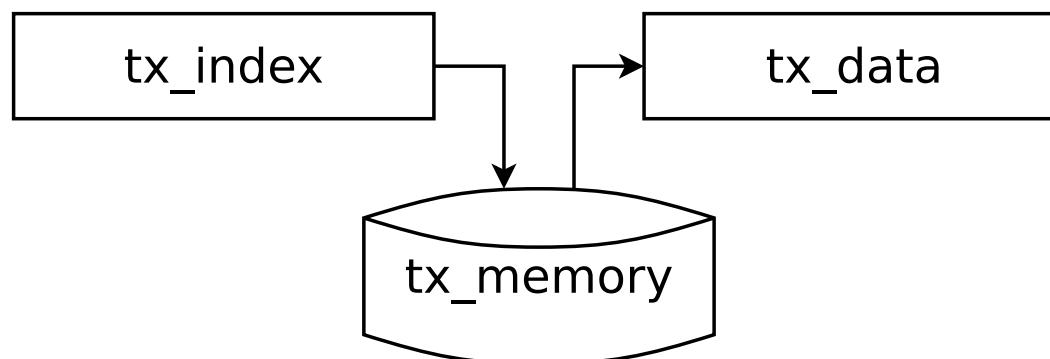
- We'll send our message once per second
- A counter, `hz_counter`, will count each second
- When `hz_counter` reaches zero, `tx_restart` will signal the rest of the design to restart

This much should be fairly familiar

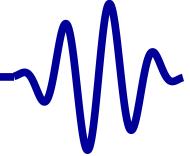


Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
▷ Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

Here's how our design is going to work

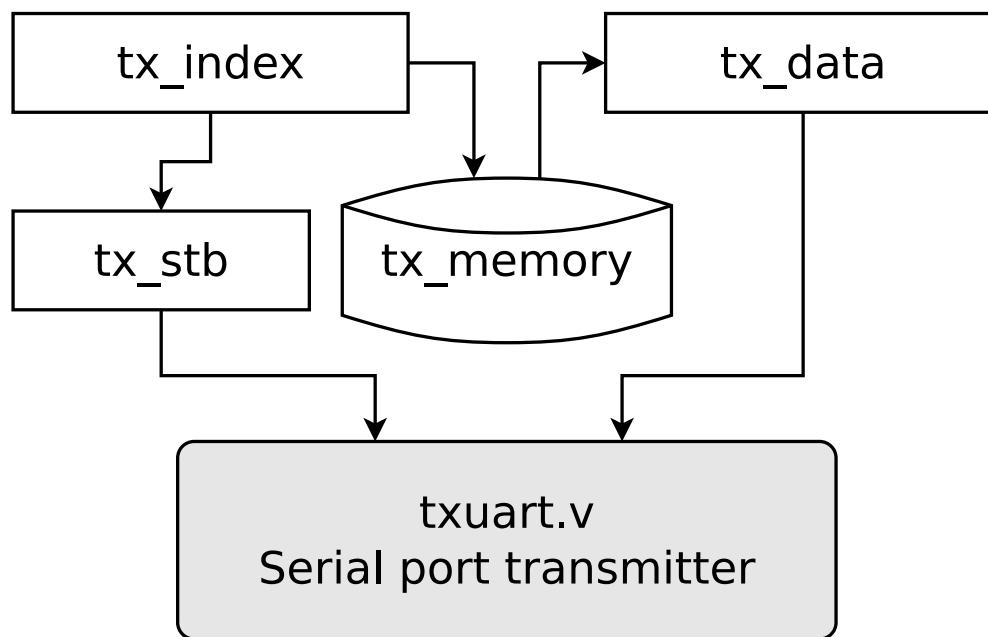


- `tx_index` will capture our position in the message stream
- We'll read `tx_data` from memory, to know what to transmit



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
▷ Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

Here's how our design is going to work



- `tx_stb` will request a byte to be transmitted
- Once the whole message has been transmitted,
- `tx_stb` will deactivate until the next `tx_restart`

Are you ready to examine some Verilog?



Restarting

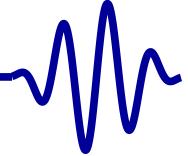


Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
▷ Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

Here's the one-second counter, hz_counter

```
// We'll start our counter just before the
// top of the second, to give everything
// a chance to initialize
initial hz_counter = 28'h16;
always @(posedge i_clk)
if (i_reset)
    hz_counter <= 28'h16;
else if (hz_counter == 0)
    hz_counter <= CLOCK_RATE_HZ - 1'b1;
else
    hz_counter <= hz_counter - 1'b1;
```

Restarting



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
▷ Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

Here's the one-second counter, hz_counter

```
// We'll start our counter just before the
// top of the second, to give everything
// a chance to initialize
initial hz_counter = 28'h16;
always @(posedge i_clk)
if (i_reset)
    hz_counter <= 28'h16;
else if (hz_counter == 0)
    hz_counter <= CLOCK_RATE_HZ - 1'b1;
else
    hz_counter <= hz_counter - 1'b1;
```

- Question: What assertion(s) does this logic require?

Restarting



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
▷ Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

Once a second, we'll set tx_restart

```
initial tx_restart = 0;  
always @(* posedge i_clk)  
    tx_restart <= (hz_counter == 1);
```

Do you see a formal property hiding in here?

```
always @(* )  
    assert(tx_restart == (hz_counter == 0));
```

Practice writing assertions as you see relationships!



Mem Address



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
▷ Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

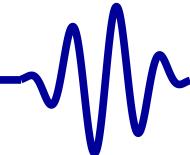
We'll need an address to read from memory

```
// Number of bytes in our message
parameter MSGLEN = 1600;

initial tx_index = 0;
always @(posedge i_clk)
if (i_reset)
    tx_index <= 0;
else if ((tx_stb)&&(!tx_busy))
begin // Advance anytime a character was
      // accepted by the serial port ,
      if (tx_index == MSGLEN-1)
          // End of message
          tx_index <= 0;
else
    tx_index <= tx_index + 1'b1;
end
```



Reading from Memory



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
▷ Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

Memory reads take one clock

```
always @(posedge i_clk)
    tx_data <= tx_memory[tx_index];
```

Remember our rules from earlier?

- We might have also chosen to use a read enable
- It wasn't necessary for this design though

When to transmit



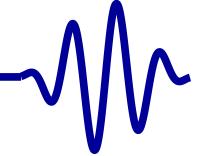
Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
▷ Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

As with Hello World, tx_stb indicates we have a character to transmit

```
initial tx_stb = 1'b0;
always @(posedge i_clk)
if (i_reset)
    tx_stb <= 1'b0;
else if (tx_restart)
    // Start transmitting anytime
    // tx_restart is true
    tx_stb <= 1'b1;
else if ((tx_stb)&&(!tx_busy)
    &&(tx_index >= MSGLEN-1))
    // Stop when we get to the end
    // of the message
    tx_stb <= 1'b0;
```



Serial Port



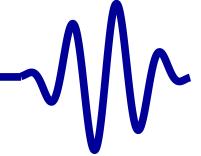
Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
Mem Address
▷ Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

We'll skip the serial port details here

- We built this earlier
- We also showed how to abstract the serial port earlier
- Even our simulation script is nearly identical to Hello World

Feel free to go back and review if you don't remember these

Next Steps



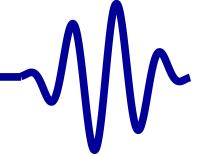
Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
▷ Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

That's the basics of our design!

- We've already built our hex file, so
- We can now move on to formal verification!



Formal Verification



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal
▷ Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

Formally verifying a component using memory requires:

- Assuming a constant address
- Asserting properties for the value at that address
- Usually requires examining no more than a single address

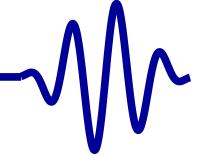
We can assume a constant value using the (* `anyconst` *) attribute

```
(* anyconst *) reg [10:0] f_const_addr;
```

- This allows the solver to pick any value for f_const_addr
- As long as it is constant
- If even one value can make your design fail,
the solver will find it

Let's see how this works

Formal Verification



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
 Formal
 Verification
 Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

Let's create a value to match our memory at

```
(* anyconst *) reg [10:0] f_const_addr;
```

- We'll call this `f_const_value`

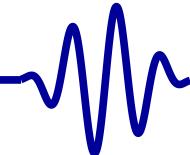
```
reg [7:0] f_const_value;  
  
always @(posedge i_clk)  
if (!f_past_valid)  
    f_const_value <= tx_memory[f_const_addr];  
else  
    assert(f_const_value  
          == tx_memory[f_const_addr]);
```

This value is constant because we are implementing a ROM

Now we can **assert** any properties associated with this address



Formal Verification



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal
▷ Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

This design need only **assert** one memory property

```
(* anyconst *) reg [10:0] f_const_addr;  
                      reg [ 7:0] f_const_value;
```

- When we transmit a value from `f_const_addr`,
- **assert** that it is the right value

```
always @(posedge i_clk)  
if ((tx_stb)&&(!tx_busy)  
    &&(tx_index == f_const_addr))  
    assert(tx_data == f_const_value);
```

We'll come back to this memory verification approach again when we discuss FIFOs



Formal Verification



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
 Formal
 ▷ Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

What other properties might we **assert**?

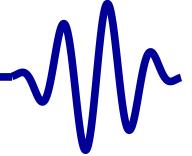
- That our index remains within bounds?
- That any time our index is within the memory bounds, tx_stb is high?

You should be familiar with these

- Let's pause to look at the reset
- Cover might need some attention as well



Reset Assertions



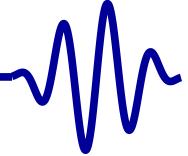
Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
▷ Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

Synchronous reset properties have a basic pattern

- You may (or may not) assume an initial reset

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

Reset Assertions



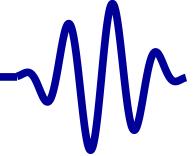
Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
▷ Reset Assertions
Cover
Exercise!
Hardware Bonus
Conclusion

Synchronous reset properties have a basic pattern

- You may (or may not) assume an initial reset
- The **initial** value, held when !f_past_valid, and
The value following a reset, i.e. when **\$past(i_reset)**
Should both be identical

```
// Check for anything with an initial
// or a reset value here
always @(posedge i_clk)
  if ((!f_past_valid)||($past(i_reset)))
    begin
      assert(hz_counter == 28'h16);
      assert(tx_index == 0);
      assert(tx_stb == 0);
    end
```

- This verifies we met the rules of a synchronous reset



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
▷ Cover
Exercise!
Hardware Bonus
Conclusion

Unlike our Hello World design

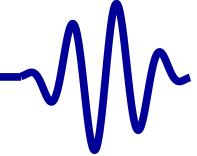
- We can't cover the entire message
 - It's just too long
- We can only cover the first several steps
- Let's cover the first 30 characters

```
always @(posedge i_clk)
cover(tx_index == 30);
```

We'll need to simulate the rest



Simulation

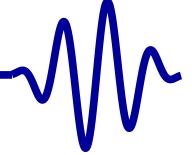


Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
▷ Cover
Exercise!
Hardware Bonus
Conclusion

Our simulation script is nearly identical to Hello World

```
// ...
#include "Vmemtx.h"
// ...
int main(int argc, char **argv) {
    // ...
    TESTB<Vmemtx> *tb = new TESTB<Vmemtx>;
    //
    tb->opentrace("memtx.vcd");
    for(unsigned clocks=0;
        clocks < 16*2000*baudclocks;
        clocks++) {
        //
        tb->tick();
        (*uart)(tb->m_core->o_uart_tx);
    } // ...
}
```

Exercise!



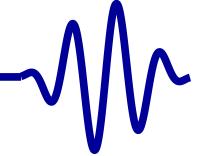
Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
▷ Exercise!
Hardware Bonus
Conclusion

As with all of our designs, let's:

- Formally Verify this design
- Make sure it works in simulation



Hardware Bonus

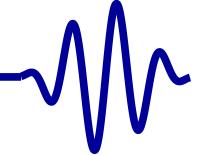


Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
▷ Hardware Bonus
Conclusion

If you have hardware to work with,

- Build this design for your hardware!
 - Be sure to compare the resource usage to Hello World
- Examine the serial port output
 - Does your terminal require carriage returns?
- How hard would it be to change the message?
 - Pick another message to send
 - ▷ Perhaps the Sermon on the Mount from [Matthew 5-7](#)?
 - What changes would need to be made to your design to support a longer message?
 - What's the longest message your hardware will support?
 - ▷ Would [Psalm 119](#) fit?

Conclusion



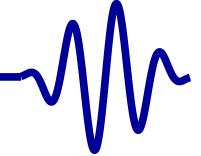
Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
▷ Conclusion

What did we learn this lesson?

- The Rules of using Block RAM
- How to generate a hex file for initializing memory
- Two forms of synchronous reset logic
- How to formally verify ...
 - A component that uses RAM
 - A synchronous reset

Now we just need to build a serial port receiver

Conclusion



Lesson Overview
Design Goal
On-chip RAM
Block RAM Rules
Initializing Memory
Hex file
Reset
Overview
Restarting
Mem Address
Serial Port
Next Steps
Formal Verification
Reset Assertions
Cover
Exercise!
Hardware Bonus
▷ Conclusion

What did we learn this lesson?

- The Rules of using Block RAM
- How to generate a hex file for initializing memory
- Two forms of synchronous reset logic
- How to formally verify ...
 - A component that uses RAM
 - A synchronous reset

Now we just need to build a serial port receiver

- That's next!



Gisselquist
Technology, LLC

9. Serial Port Receiver

Daniel E. Gisselquist, Ph.D.





Lesson Overview



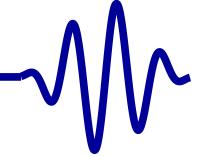
- ▷ Lesson Overview
- Design Goal
- Receiver FSM
- Baud counter
- Receiver State
- Return Data
- Formal Verification
- Formal Contract
- Induction Properties
- Induction
- Cover
- Formal Exercise
- Simulation
- UARTSIM
- Exercise!
- Hardware
- Conclusion

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

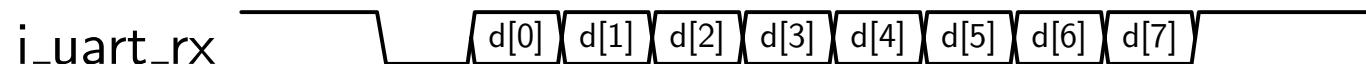
Objectives

- 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

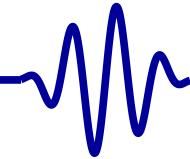


Lesson Overview
▷ Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
Formal Verification
Formal Contract
Induction Properties
Induction
Cover
Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

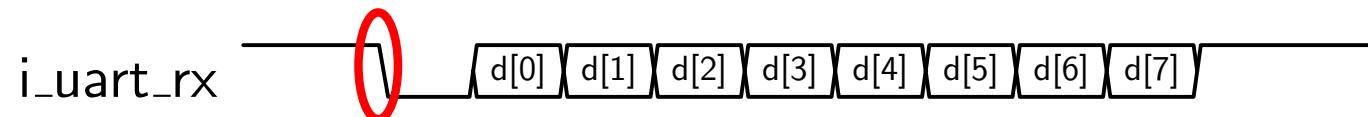
We discussed building a [serial port receiver](#) before



The basic processing steps are:

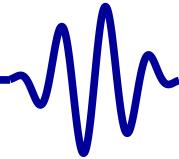


We discussed building a **serial port receiver** before

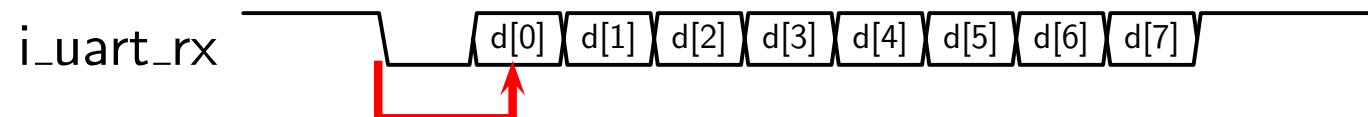


The basic processing steps are:

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

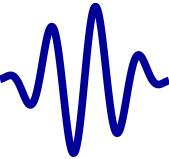


We discussed building a **serial port** receiver before

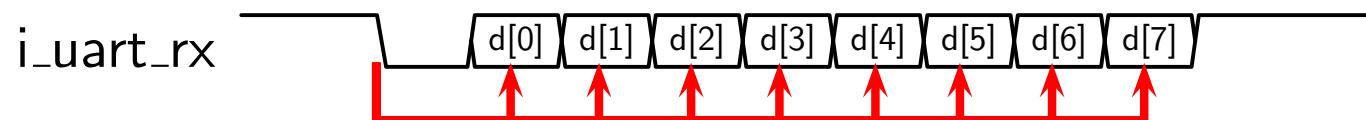


The basic processing steps are:

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

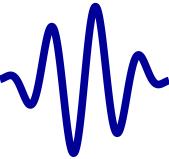


We discussed building a **serial port receiver** before

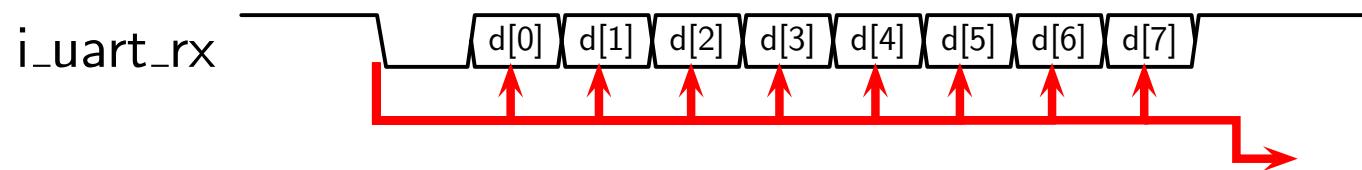


The basic processing steps are:

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

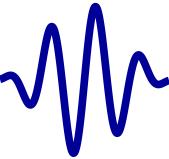


We discussed building a **serial port receiver** before

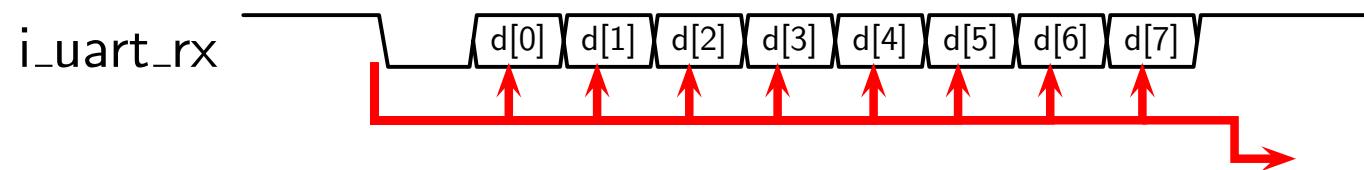


The basic processing steps are:

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

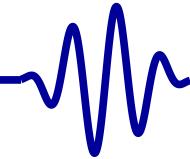


We discussed building a **serial port receiver** before



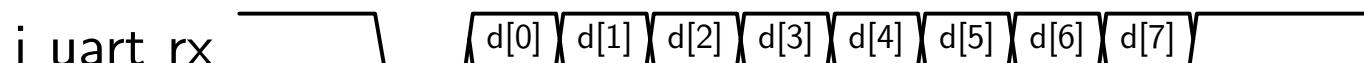
This also means that we'll be done halfway through the stop bit

- The transmitter will still be busy, even though
- The receiver is already looking for the next start bit



Lesson Overview
▷ Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
Formal Verification
Formal Contract
Induction Properties
Induction
Cover
Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

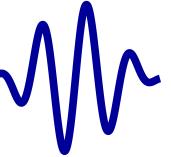
Since our last discussion (about simulation)



We've learned that we need to synchronize the incoming bit

```
// 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



Lesson Overview

Design Goal

► Receiver FSM

Baud counter

Receiver State

Return Data

Formal Verification

Formal Contract

Induction Properties

Induction

Cover

Formal Exercise

Simulation

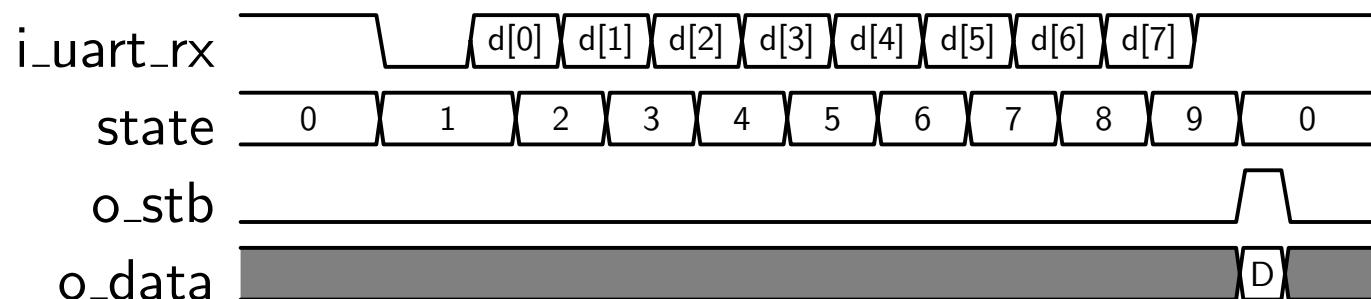
UARTSIM

Exercise!

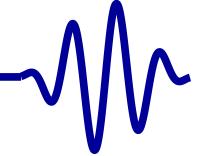
Hardware

Conclusion

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_stb signal will be one clock wide
- When o_stb is high, o_data contains the received data
 - It is a don't care value otherwise



Lesson Overview

Design Goal

Receiver FSM

▷ Baud counter

Receiver State

Return Data

Formal Verification

Formal Contract

Induction Properties

Induction

Cover

Formal Exercise

Simulation

UARTSIM

Exercise!

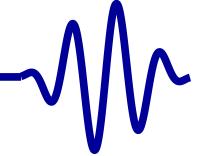
Hardware

Conclusion

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

```
initial baud_counter = 0;
always @(posedge i_clk)
if (state == IDLE)
begin
    baud_counter <= 0;
    // ...
```

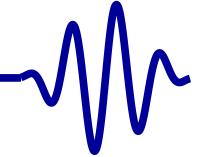


Lesson Overview
Design Goal
Receiver FSM
▷ Baud counter
Receiver State
Return Data
Formal Verification
Formal Contract
Induction Properties
Induction
Cover
Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

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;
    // ...
end
```



Lesson Overview

Design Goal

Receiver FSM

▷ Baud counter

Receiver State

Return Data

Formal Verification

Formal Contract

Induction Properties

Induction

Cover

Formal Exercise

Simulation

UARTSIM

Exercise!

Hardware

Conclusion

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

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



Lesson Overview

Design Goal

Receiver FSM

▷ Baud counter

Receiver State

Return Data

Formal Verification

Formal Contract

Induction Properties

Induction

Cover

Formal Exercise

Simulation

UARTSIM

Exercise!

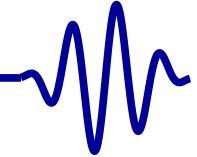
Hardware

Conclusion

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

```
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;
```

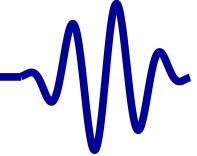


Let's work through timing all these transitions

- 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

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

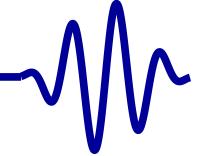
- ... where it will remain at zero



The receiver state follows the same conditions

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

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

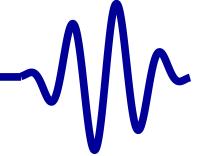


Lesson Overview
Design Goal
Receiver FSM
Baud counter
▷ Receiver State
Return Data
Formal Verification
Formal Contract
Induction Properties
Induction
Cover
Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

The receiver state follows the same conditions

- We start in IDLE, and remain in IDLE while ck_uart is high
- When ck_uart goes low, we switch states

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



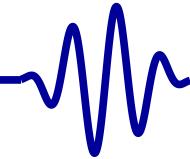
Lesson Overview
Design Goal
Receiver FSM
Baud counter
▷ Receiver State
Return Data
Formal Verification
Formal Contract
Induction Properties
Induction
Cover
Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

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

```
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?

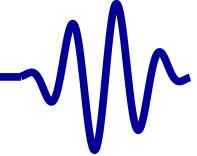


Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
▷ Return Data
Formal Verification
Formal Contract
Induction Properties
Induction
Cover
Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

On every state change

- Shift in one more bit of the answer

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

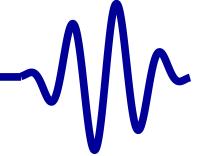


Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
▷ Return Data
Formal Verification
Formal Contract
Induction Properties
Induction
Cover
Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

On the last and final transition

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

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



On the last and final transition

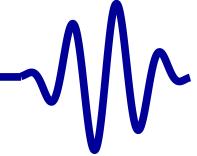
- Notify our environment of a received bit
- Just as we return to IDLE

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

This should all be quite straight forward

- This isn't really any harder than the transmitter

Formal Verification



Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
 Formal
 Verification
Formal Contract
Induction Properties
Induction
Cover
Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

Formally verifying this receiver . . . that's harder

Let's reflect upon the two basic types of properties we've created

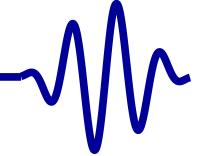
- Contract properties
 - Verify that a design does what it was intended to do
 - These can be **black-box properties**
- Induction properties
 - Verify that a design remains within the set of legal states
 - These will always be **white-box properties**

And our two rules

- **assume** any input properties
- **assert** any local state and output properties



Formal Contract

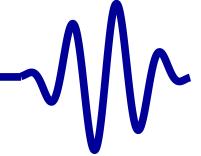


Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
Formal Verification
▷ Formal Contract
Induction Properties
Induction
Cover
Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

The contract for a [serial port](#) is straight forward

- 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

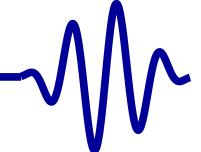
That's the contract. That's the easy part



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

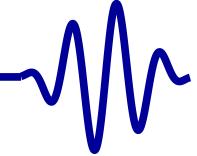


We'll add two output ports to our transmitter for this purpose

- `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 `f_txuart`

- Since these extra ports are only necessary for formally verifying the receiver
- They are inappropriate for an independent transmitter

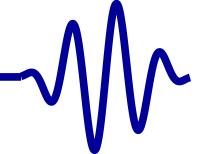


Capturing the data being sent is easy

```
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



Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
Formal Verification
Formal Contract
 Induction
 Properties
 Induction
 Cover
Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

The transmit counter is conceptually simple

```
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

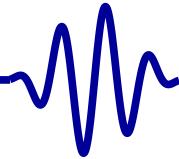


Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
Formal Verification
Formal Contract
 Induction
 ▷ Properties
 Induction
Cover
Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

Matching f_counter to the transmitter's count-down counter

```
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
```

Let's look at this a little deeper



Lesson Overview

Design Goal

Receiver FSM

Baud counter

Receiver State

Return Data

Formal Verification

Formal Contract

Induction

Properties

Induction

Cover

Formal Exercise

Simulation

UARTSIM

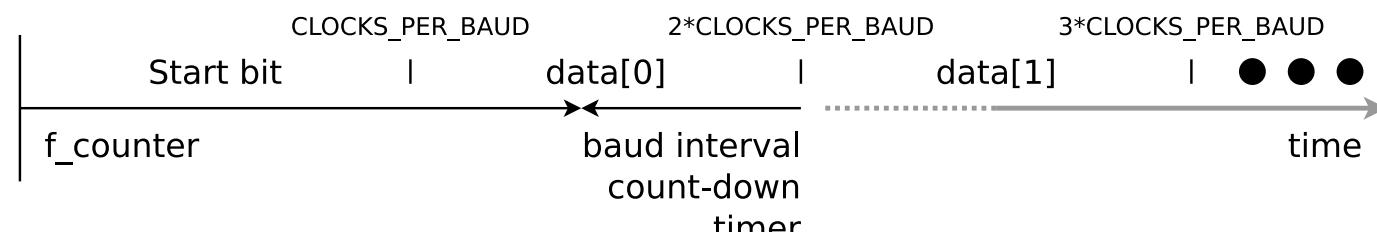
Exercise!

Hardware

Conclusion

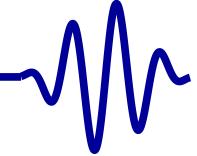
Let's discuss these assertions

```
BIT_ZERO: assert ( f_counter  
                      == 2*CLOCKS_PER_BAUD-1-counter );
```



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

- f_counter starts at the beginning of time and counts up
- Our baud interval counter, counter, counts down each interval



Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
Formal Verification
Formal Contract
 Induction
 Properties
Induction
Cover
Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

Let's discuss these assertions

```
BIT_ZERO: assert ( f_counter  
                  == 2*CLOCKS_PER_BAUD-1-counter );
```

Notice the multiply for a moment

- Multiplies 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

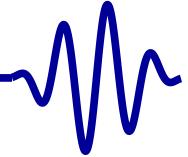
- What about the inputs to f_txuart?



Our receiver doesn't have inputs for the formal transmitter

- We need to generate inputs for `f_txuart`
- `(* anyseq *)` can be used for that purpose
- `(* anyseq *)` is like `(* anyconst *)`
 - The solver gets to pick the values
- Only `(* anyseq *)` values can change from clock to clock
 - `(* anyconst *)` values are required to be constant
- Both types of values may be constrained by assumptions
- We'll pass two inputs to the transmitter

```
// The write request input
(* anyseq *)      reg                      f_tx_iwr;
// The write data input
(* anyseq *)      reg [7:0]                f_tx_idata;
```



Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
Formal Verification
Formal Contract
 Induction
▷ Properties
 Induction
Cover
Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

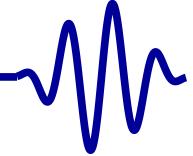
Here's our transmitter instantiation

```
(* anyseq *) reg f_tx_iwr;
(* anyseq *) reg [7:0] f_tx_idata;
wire 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);
// Assume our input matches the txuart's output
always @(*)
    assume(i_uart_tx == f_tx_uart);
```

We'll be working with f_txdata and f_tx_counter

Contract



Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
Formal Verification
Formal Contract
 Induction
 ▷ Properties
 Induction
Cover
Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

We can now assert our receiver contract

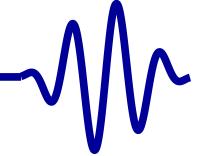
- o_wr goes high once at the end of every word

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

- o_data has a copy of the transmitted information

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

Problem: that's not enough to pass induction!

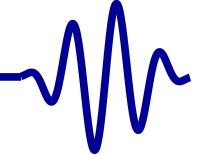


Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
Formal Verification
Formal Contract
Induction Properties
▷ Induction
Cover
Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

Now we need to synchronize our partial results

```
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



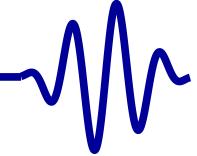
Matching the two counters is harder

- 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

```
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);
          end
        else
          assert(f_tx_counter < 3);
        end
```



Induction



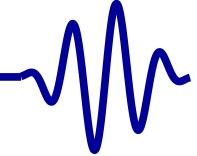
Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
Formal Verification
Formal Contract
Induction Properties
▷ Induction
Cover
Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

Matching the two counters is harder

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

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

- 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

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

- Don't forget that baud_counter counts down,
- While f_tx_counter counts up

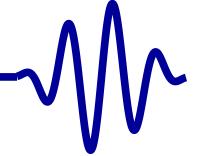


Beginners often struggle to understand how the transmitter and receiver can get out of synch during **induction**

- 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

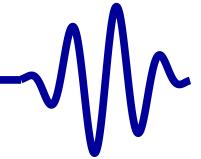
- We'll discuss **cover()** next
- Then you should be able to finish the proof



We should cover our solution as well

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

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

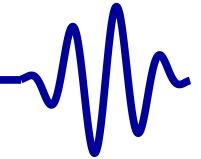


Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
Formal Verification
Formal Contract
Induction Properties
Induction
▷ Cover
Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

Remember tasks?

- You can use tasks to selectively adjust parameter values

```
[ 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
```



Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
Formal Verification
Formal Contract
Induction Properties
Induction
▷ Cover
Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

Remember tasks?

- You can use tasks to selectively adjust parameter values

```
[ tasks ]
```

```
  cvr
```

```
  prf
```

```
  [ options ]
```

```
    prf: mode prov
```

```
    cvr: mode cover
```

```
    cvr: depth 192
```

```
    prf: depth 4
```

```
  [ script ]
```

```
    read -formal f_txuart.v
```

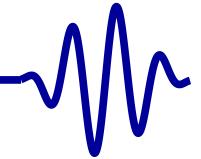
```
    read -formal rxuart.v
```

```
    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



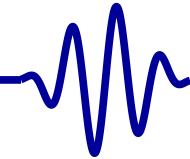
Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
Formal Verification
Formal Contract
Induction Properties
Induction
▷ Cover
Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

Remember tasks?

- You can use tasks to selectively adjust parameter values

```
[ tasks ]
cvr
prf
[ options ]
prf: mode prov
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

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

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

```
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 $20*\text{CLOCKS_PER_BAUD}$
- Our original transmitter took one clock too long
 - Two characters took $20*\text{CLOCKS_PER_BAUD}+1$ 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

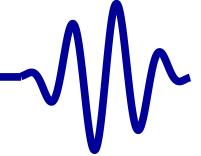
- Create a register called zero_baud_counter

```
reg zero_baud_counter;
```

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

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

You may start with the (mostly correct) receiver in exercise 9

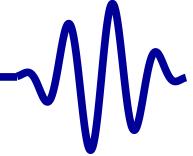


Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
Formal Verification
Formal Contract
Induction Properties
Induction
Cover
▷ Formal Exercise
Simulation
UARTSIM
Exercise!
Hardware
Conclusion

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?

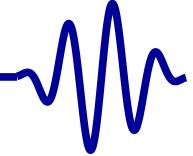
GT Simulation



Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
Formal Verification
Formal Contract
Induction Properties
Induction
Cover
Formal Exercise
▷ Simulation
UARTSIM
Exercise!
Hardware
Conclusion

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

```
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

```

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

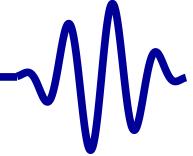


The transmit logic follows what we've rehearsed already

- 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

```
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;
```

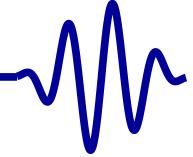


The transmit logic follows what we've rehearsed already

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

```
} 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?

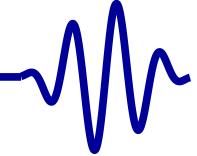


The transmit logic follows what we've rehearsed already

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

```
} 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?

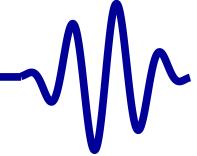


The transmit logic follows what we've rehearsed already

- One less clock on the last step is required because
 - It takes a clock to recognize the idle, and then to
 - Return **m_tx_state** to IDLE

```
} 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

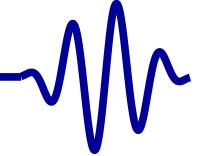


The transmit logic follows what we've rehearsed already

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

```
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



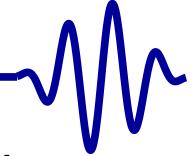
Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
Formal Verification
Formal Contract
Induction Properties
Induction
Cover
Formal Exercise
Simulation
▷ UARTSIM
Exercise!
Hardware
Conclusion

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

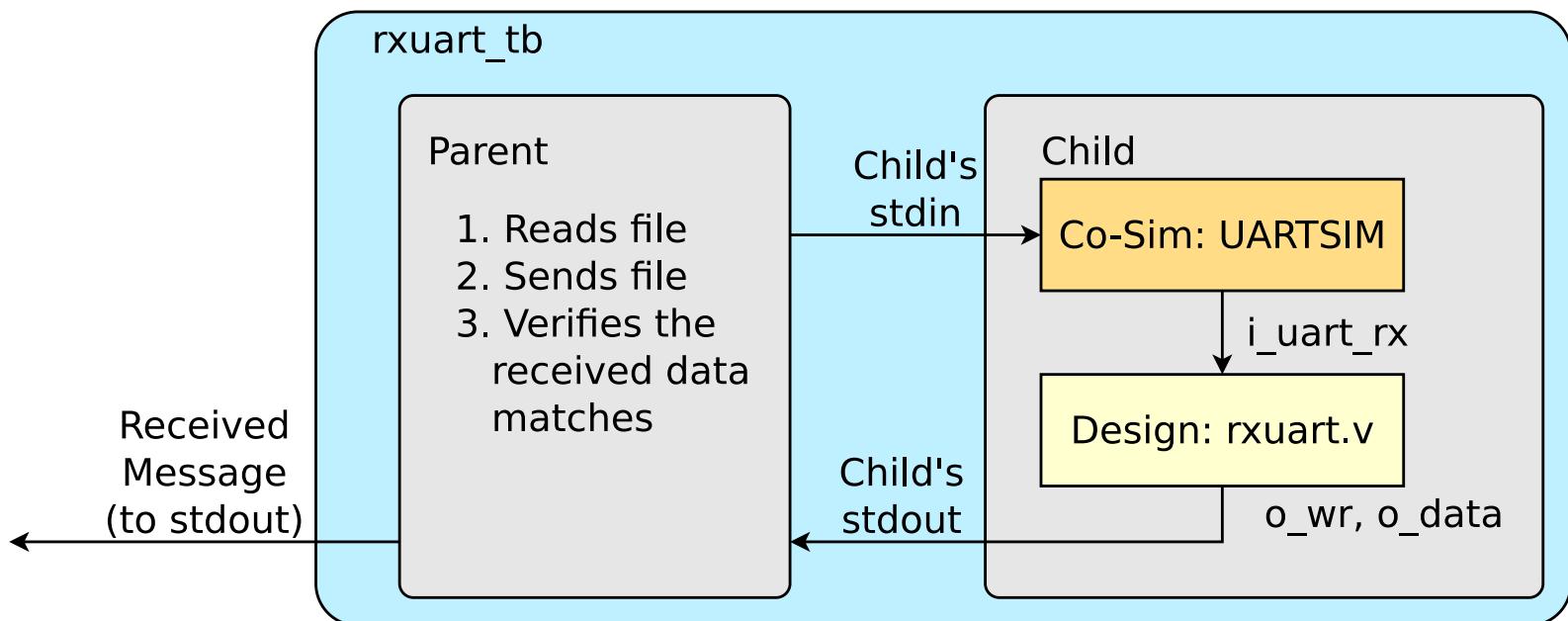
The fact that UARTSIM uses stdin will make this problematic

The setup

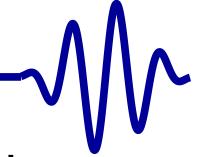


Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
Formal Verification
Formal Contract
Induction Properties
Induction
Cover
Formal Exercise
Simulation
▷ UARTSIM
Exercise!
Hardware
Conclusion

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

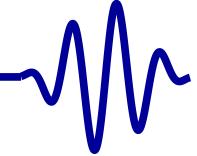


Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
Formal Verification
Formal Contract
Induction Properties
Induction
Cover
Formal Exercise
Simulation
▷ UARTSIM
Exercise!
Hardware
Conclusion

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

```
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

pipe()



Lesson Overview
Design Goal
Receiver FSM
Baud counter
Receiver State
Return Data
Formal Verification
Formal Contract
Induction Properties
Induction
Cover
Formal Exercise
Simulation
▷ UARTSIM
Exercise!
Hardware
Conclusion

The **pipe()** system call creates a pipe

- We'll need two: one for each direction

```
int      child�_stdin [2] , child�_stdout [2];  
  
if ((pipe(child�_stdin) !=0)  
    || (pipe(child�_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**

pipe()



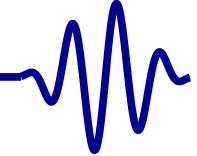
The **pipe()** system call creates a unidirectional pipe

- Data written to **child_stdin [1]** can be read from **child_stdin [0]**, same for **child_stdout**
- The parent closes the read end of the **child_stdin**

```
close(child_stdin[0]);
```

- Only the child will read from this pipe
 - The parent also closes the write end of **child_stdout**
- ```
close(child_stdout[1]);
```
- Only the child will write to this pipe
  - The child will do the opposite

# pipe()



The child also needs to map these pipes to **stdin/stdout**

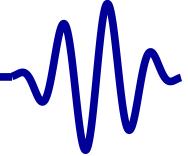
- First, map **childs\_stdin [0]** to stdin
- Done by first closing the file descriptor to be replaced
- Then duplicating the pipe's file descriptor

```
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

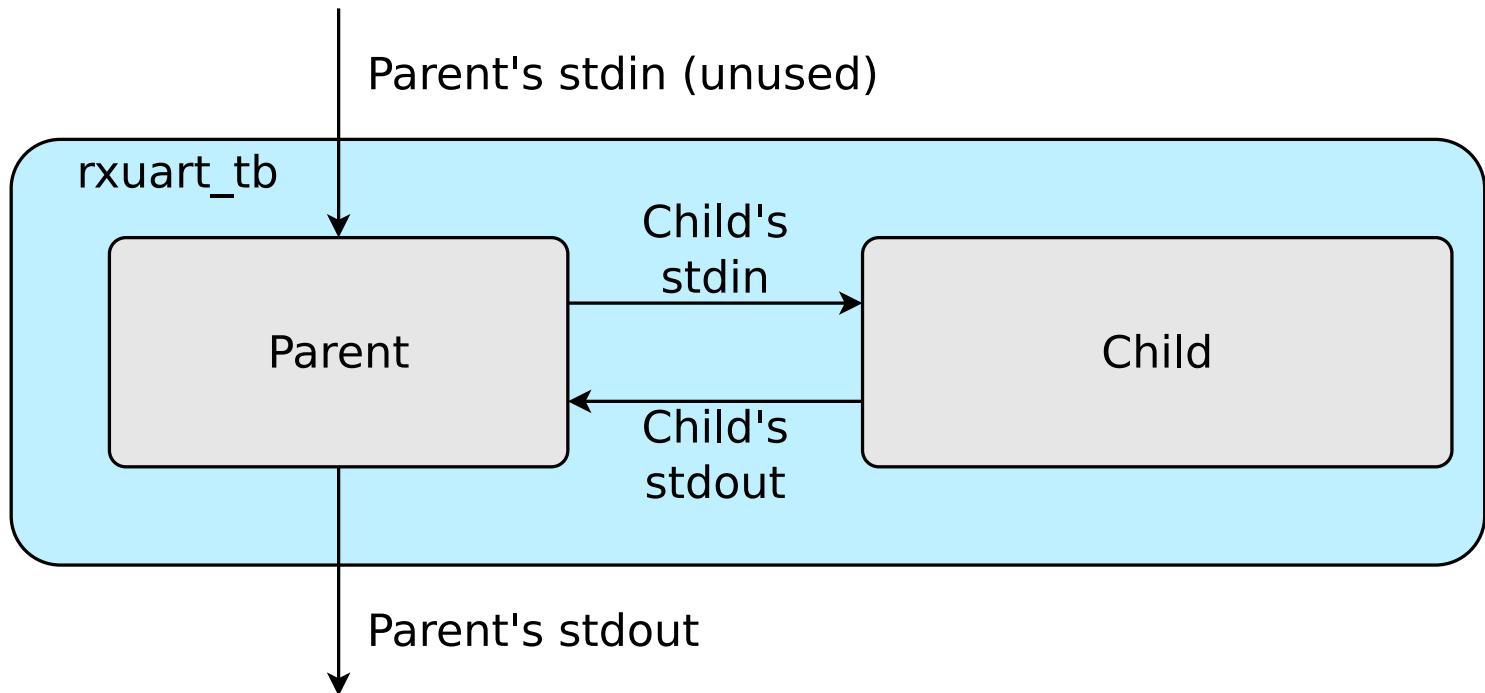
```
close(STDOUT_FILENO);
dup(childs_stdout[1]);
```

We can now build and run our test!

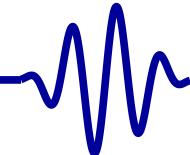


Lesson Overview  
Design Goal  
Receiver FSM  
Baud counter  
Receiver State  
Return Data  
Formal Verification  
Formal Contract  
Induction Properties  
Induction  
Cover  
Formal Exercise  
Simulation  
▷ UARTSIM  
Exercise!  
Hardware  
Conclusion

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



Lesson Overview  
Design Goal  
Receiver FSM  
Baud counter  
Receiver State  
Return Data  
Formal Verification  
Formal Contract  
Induction Properties  
Induction  
Cover  
Formal Exercise  
Simulation  
▷ UARTSIM  
Exercise!  
Hardware  
Conclusion

In the parent, we send the message to the slave

```
write(child�_stdin[1], string, flen);
```

And read it back out

```
rd = read(child�_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!

# In the Slave



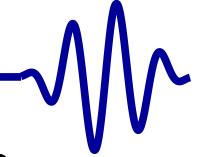
Lesson Overview  
Design Goal  
Receiver FSM  
Baud counter  
Receiver State  
Return Data  
Formal Verification  
Formal Contract  
Induction Properties  
Induction  
Cover  
Formal Exercise  
Simulation  
▷ UARTSIM  
Exercise!  
Hardware  
Conclusion

The slave's code looks like what we've done with Verilator before

- First the setup

```
// 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

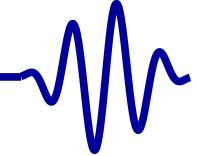


The slave matches what we've done with Verilator before

- First the setup
- Then run the testbench

```
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!



Lesson Overview  
Design Goal  
Receiver FSM  
Baud counter  
Receiver State  
Return Data  
Formal Verification  
Formal Contract  
Induction Properties  
Induction  
Cover  
Formal Exercise  
Simulation  
UARTSIM  
▷ Exercise!  
Hardware  
Conclusion

Does your component simulation work?

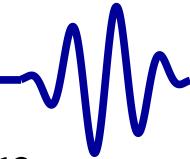
- If not, debug as necessary

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

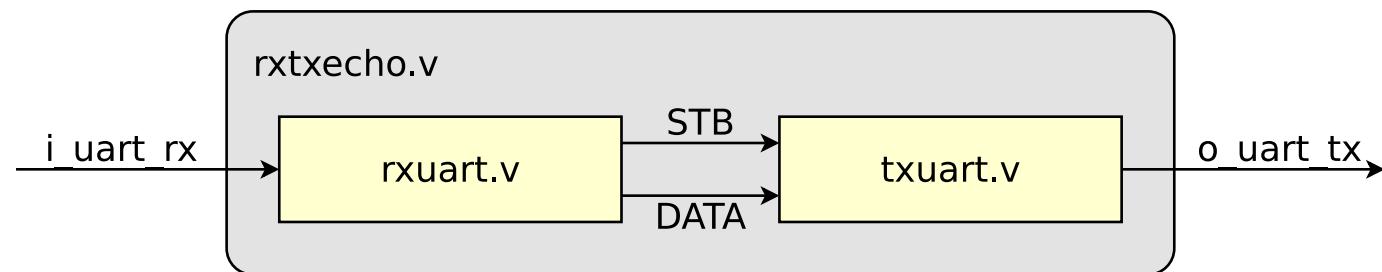
Many student's have asked, why doesn't my [serial port](#) work?

- 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



Lesson Overview  
Design Goal  
Receiver FSM  
Baud counter  
Receiver State  
Return Data  
Formal Verification  
Formal Contract  
Induction Properties  
Induction  
Cover  
Formal Exercise  
Simulation  
UARTSIM  
Exercise!  
▷ Hardware  
Conclusion

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 now use the button, the LED, and the UART output to debug
- You should also know how to fully simulate this design

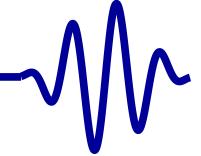


Common problems include:

- 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

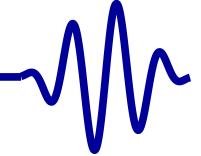


## The rarer ugly problem

- 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

```
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

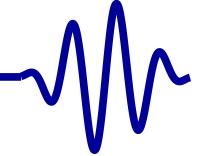


A better solution to the rare but ugly problem

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

```
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:

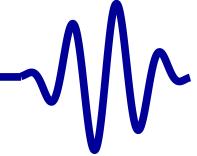
- **if** (rx\_stb) for example, or
- **if** (rx\_stb && (rx\_data == 'P')) as another

```
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



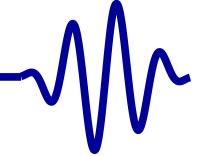
# Debugging



Lesson Overview  
Design Goal  
Receiver FSM  
Baud counter  
Receiver State  
Return Data  
Formal Verification  
Formal Contract  
Induction Properties  
Induction  
Cover  
Formal Exercise  
Simulation  
UARTSIM  
Exercise!  
▷ Hardware  
Conclusion

You can also use our transmit word design, 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

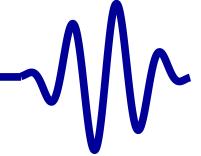


Lesson Overview  
Design Goal  
Receiver FSM  
Baud counter  
Receiver State  
Return Data  
Formal Verification  
Formal Contract  
Induction Properties  
Induction  
Cover  
Formal Exercise  
Simulation  
UARTSIM  
Exercise!  
▷ Hardware  
Conclusion

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

# Conclusion



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.



Gisselquist  
Technology, LLC

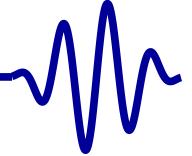
## 10. Adding a FIFO

Daniel E. Gisselquist, Ph.D.





# Lesson Overview



- ▷ Lesson Overview
- Design Goal
- What is a FIFO?
- Basic FIFO Design
  - method
  - FIFO Interface
  - FIFO Memory
  - Addresses
  - Formal Verification
  - FIFO Verification
  - Cover
  - Cover Lesson
  - Line Capturer
  - Using the FIFO
  - Rx + FIFO
  - Verifying the FSM
  - Simulation
  - Random Delay
  - Sim Trace
  - Fixing the read
  - Conclusion

Serial ports can easily get overloaded with information

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

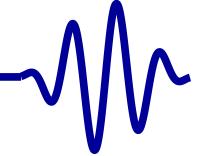
Let's build a FIFO to address these problems!

## Objectives

- Know how to build and verify a FIFO



# Design Goal



Lesson Overview

▷ Design Goal

What is a FIFO?

Basic FIFO Design  
method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

Random Delay

Sim Trace

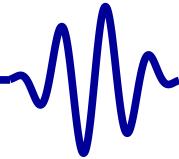
Fixing the read

Conclusion

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

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

# What is a FIFO?



Lesson Overview

Design Goal

▷ What is a FIFO?

Basic FIFO Design

method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

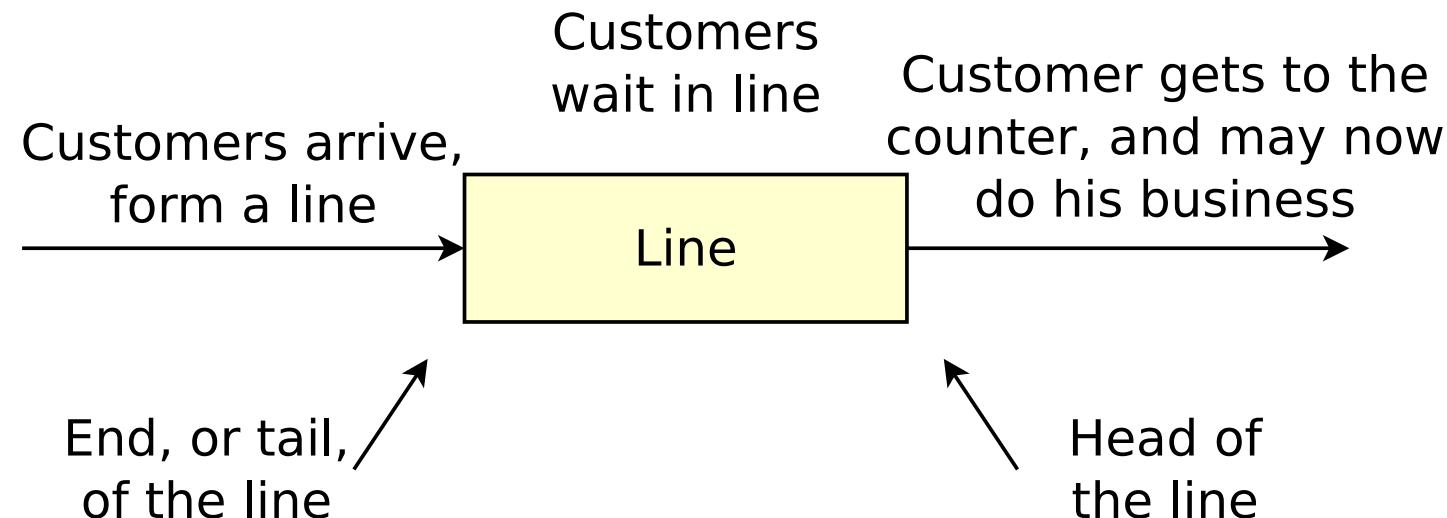
Random Delay

Sim Trace

Fixing the read

Conclusion

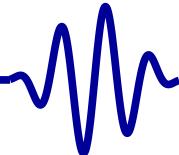
You are probably familiar with waiting in line



This could describe ...

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

# What is a FIFO?



Lesson Overview

Design Goal

▷ What is a FIFO?

Basic FIFO Design  
method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

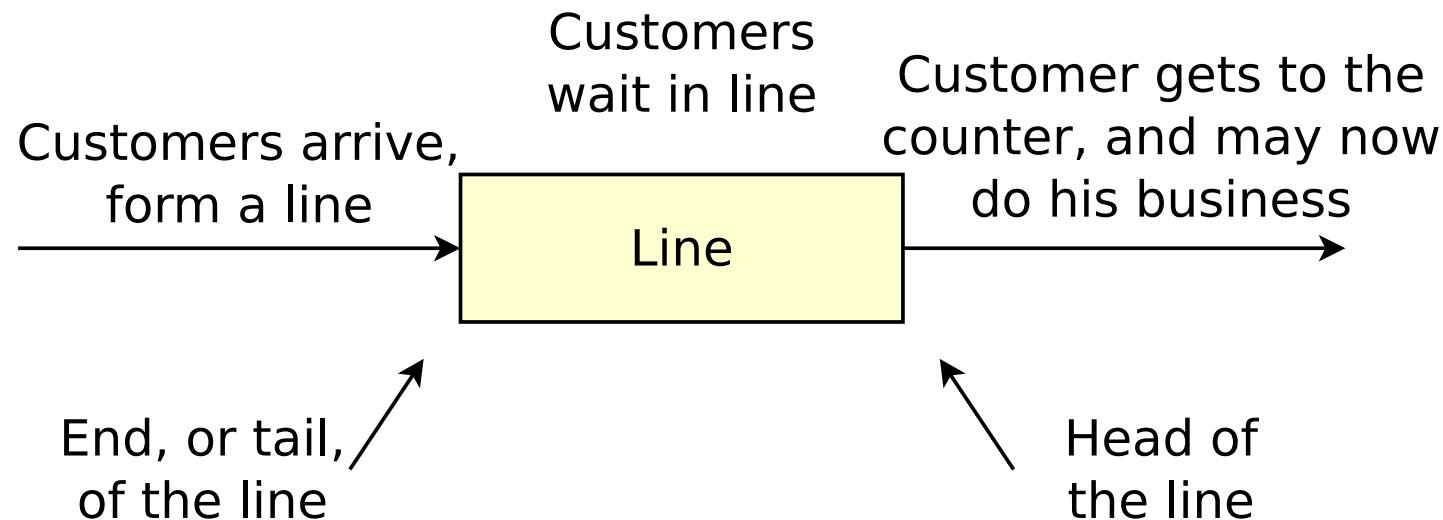
Random Delay

Sim Trace

Fixing the read

Conclusion

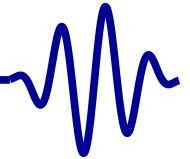
You are probably familiar with waiting in line



There are rules to waiting in line

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

# What is a FIFO?



Lesson Overview

Design Goal

▷ What is a FIFO?

Basic FIFO Design

method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

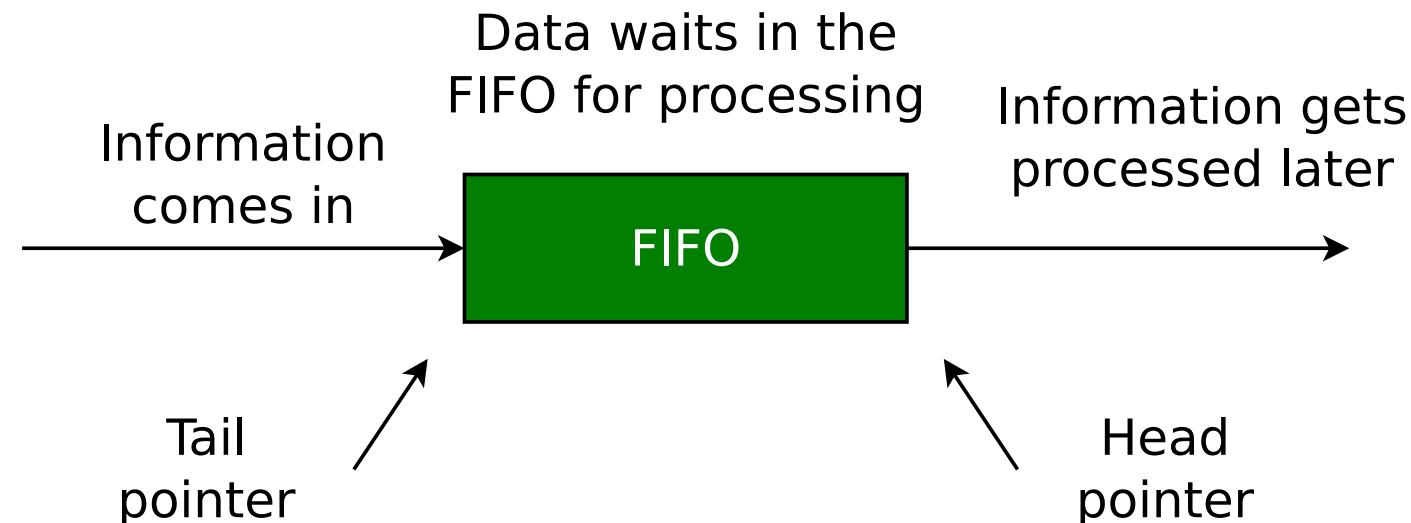
Random Delay

Sim Trace

Fixing the read

Conclusion

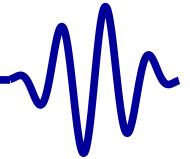
A FIFO is nothing more than data waiting in line



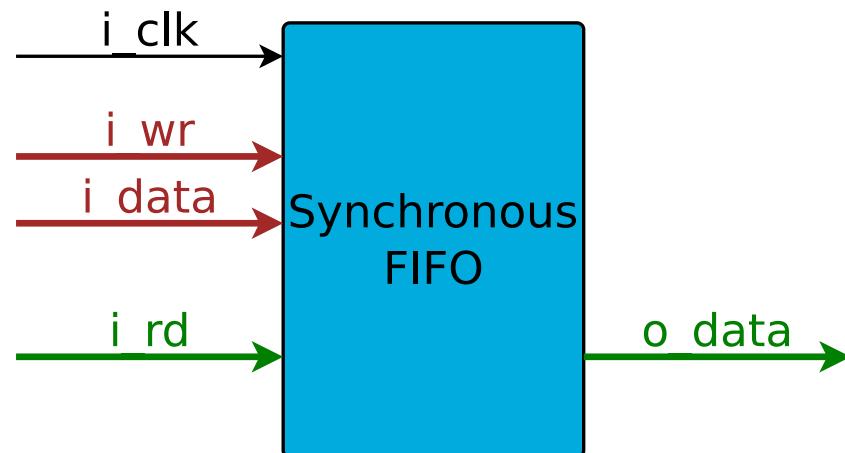
Information gets processed later

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

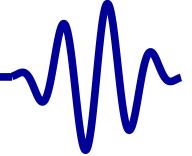
It's really just that simple



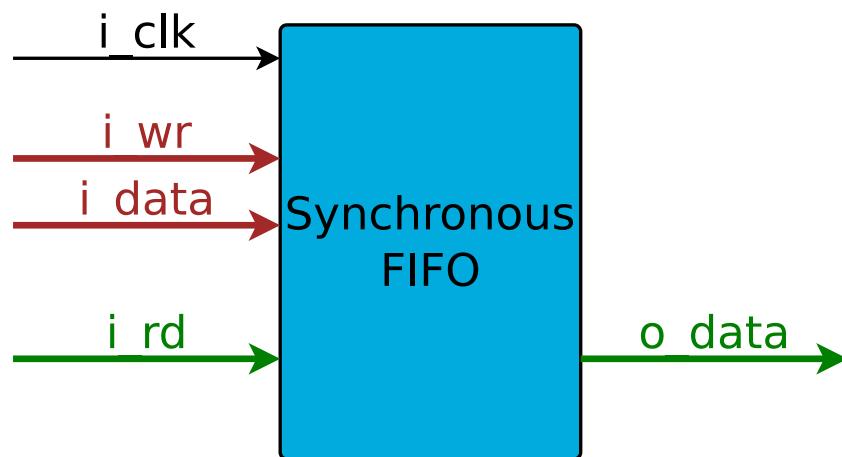
Lesson Overview  
Design Goal  
What is a FIFO?  
    Basic FIFO  
    ▷ Design  
method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion



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



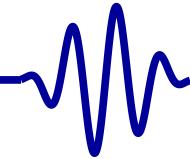
Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
▷ method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion



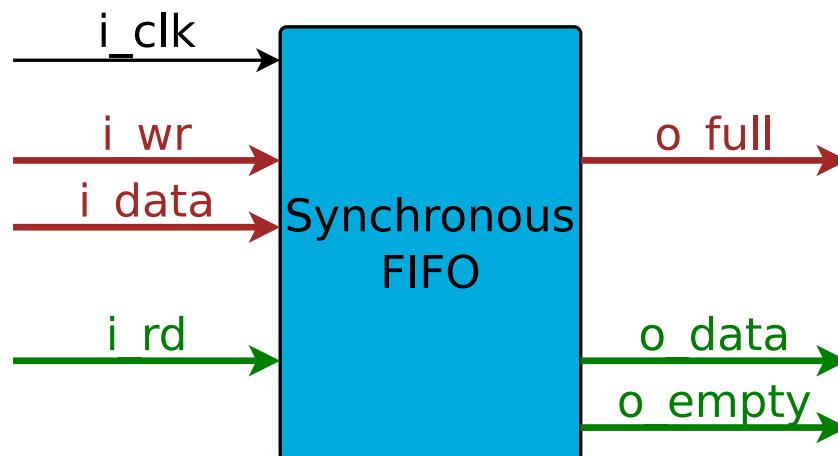
- On any `i_wr`, we'll write `i_data` to the FIFO
- On any `i_rd`, we'll return `o_data` from the FIFO

Not quite . . .

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



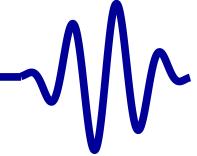
Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
▷ FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion



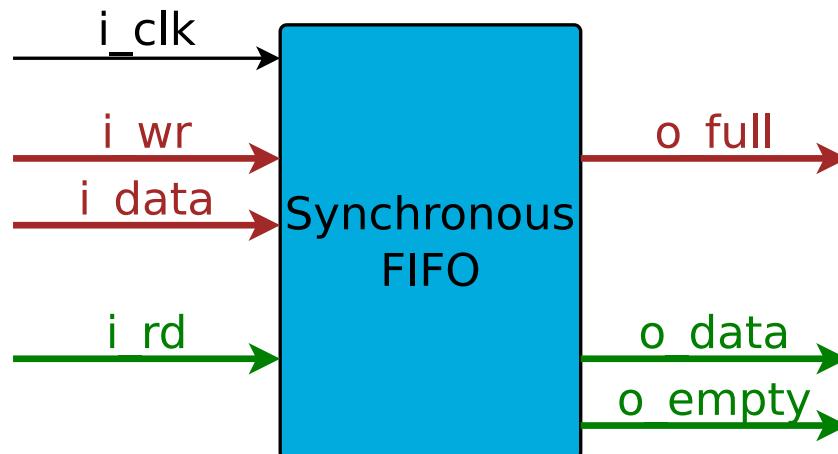
- On any `i_wr && !o_full`, we'll write `i_data` to memory
- On any `i_rd && !o_empty`, we'll read and return `o_data` from memory

We can simplify this by defining:

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



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
▷ FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion



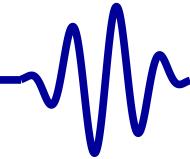
- On any `w_wr`, we'll write `i_data` to our internal memory
- On any `w_rd`, we'll read and return `o_rdata` from memory

This is how we'll handle overflows

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



# FIFO Memory



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
▷ FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

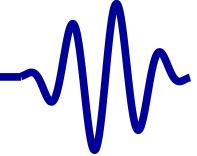
The two memory accesses constrain much of our logic

- Writes to the FIFO memory

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

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

# FIFO Memory



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design method  
FIFO Interface  
▷ FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

Our memories constrain much of our logic

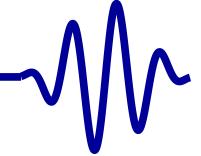
- Writes to the FIFO memory
- Reads from the FIFO memory

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

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

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

# Address pointers

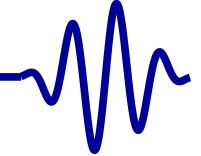


Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design method  
FIFO Interface  
FIFO Memory  
▷ Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

There's a trick to the addresses ...

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

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



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
FIFO Memory  
▷ Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

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

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

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

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

- We can now calculate our empty and full outputs

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

# Formal Checks



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design method  
FIFO Interface  
FIFO Memory  
▷ Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

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

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



# Formal Checks



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design method  
FIFO Interface  
FIFO Memory  
▷ Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

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

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

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

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



# Formal Checks



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design method  
FIFO Interface  
FIFO Memory  
▷ Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

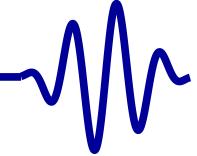
You should get in the habit of writing formal properties as you write your code

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

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

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

# Not there yet



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
FIFO Memory  
▷ Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

Our design isn't there yet

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

**always @(\*)**

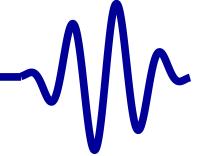
```
o_data = fifo_mem[rd_addr];
```

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

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



# Formal Verification



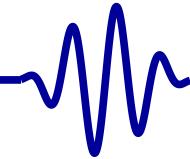
Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal  
▷ Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

## Review: Memory verification

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

*for that address only*

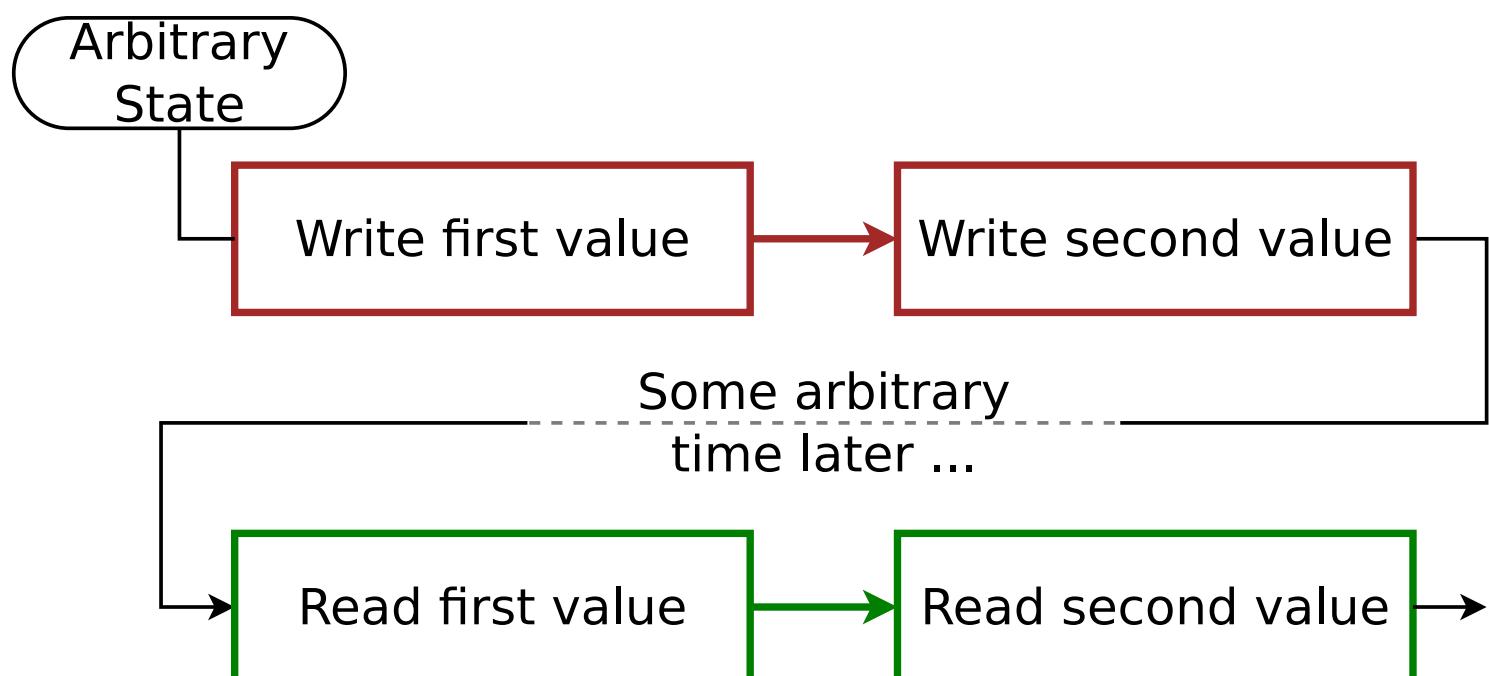
FIFO's offer a slight twist off of this strategy

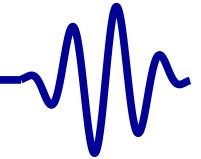


Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design method  
FIFO Interface  
FIFO Memory Addresses  
Formal Verification FIFO  
▷ Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

## To verify a FIFO

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

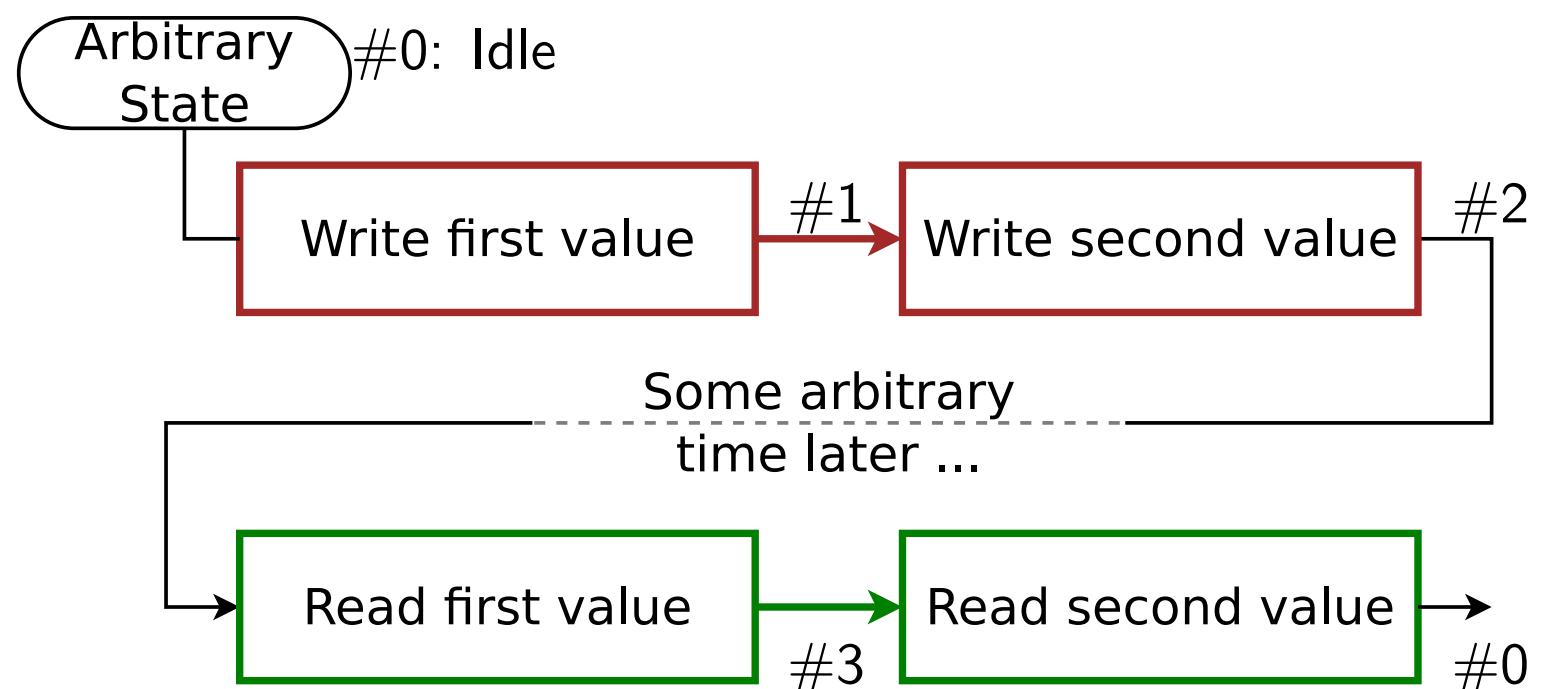




Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design method  
FIFO Interface  
FIFO Memory Addresses  
Formal Verification FIFO  
▷ Verification Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

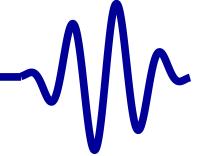
To verify a FIFO

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



Let's assign states!

# FIFO Verification



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
FIFO  
▷ Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

We'll need two consecutive addresses

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

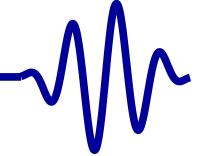
always @(*)
 f_second_addr = f_first_addr + 1;
```

We'll also need two arbitrary values

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



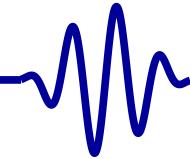
# FIFO Verification



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
  FIFO  
▷ Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

Here's our basic state transitions:

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



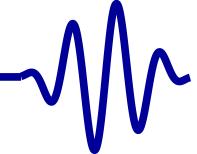
Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
  FIFO  
▷ Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

Here's our basic state transitions:

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



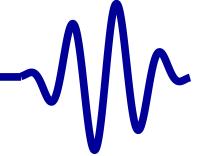
# FIFO Verification



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
  FIFO  
▷ Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

Here's our basic state transitions:

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



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design

method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO

▷ Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

Random Delay

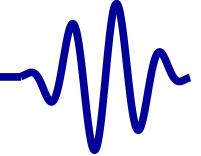
Sim Trace

Fixing the read

Conclusion

Here's our basic state transitions:

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



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design

method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO

▷ Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

Random Delay

Sim Trace

Fixing the read

Conclusion

## Basic proof:

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

This is actually harder than it sounds



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design

method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO

▷ Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

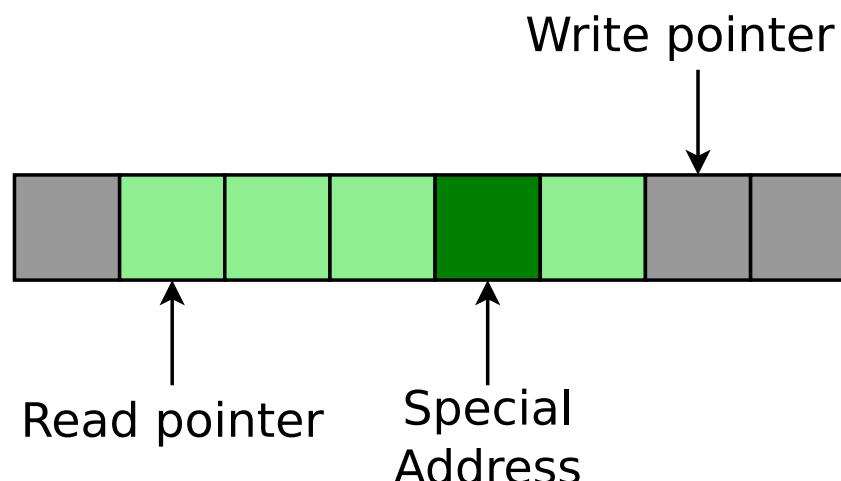
Random Delay

Sim Trace

Fixing the read

Conclusion

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



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



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design

method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO

▷ Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

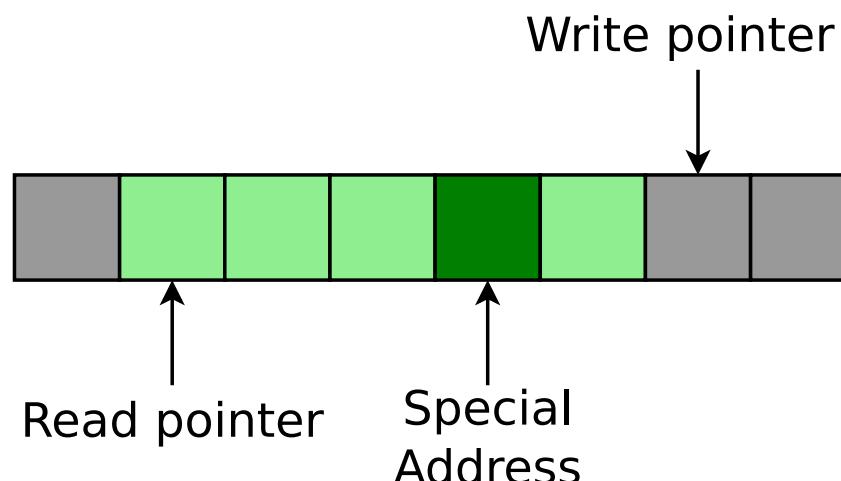
Random Delay

Sim Trace

Fixing the read

Conclusion

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



□ It must be greater or equal to the read address

□ It must be less than the write address

This is actually harder than it sounds

□ *The pointers can wrap!*



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design  
method

FIFO Interface

FIFO Memory

Addresses

Formal Verification  
FIFO

▷ Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

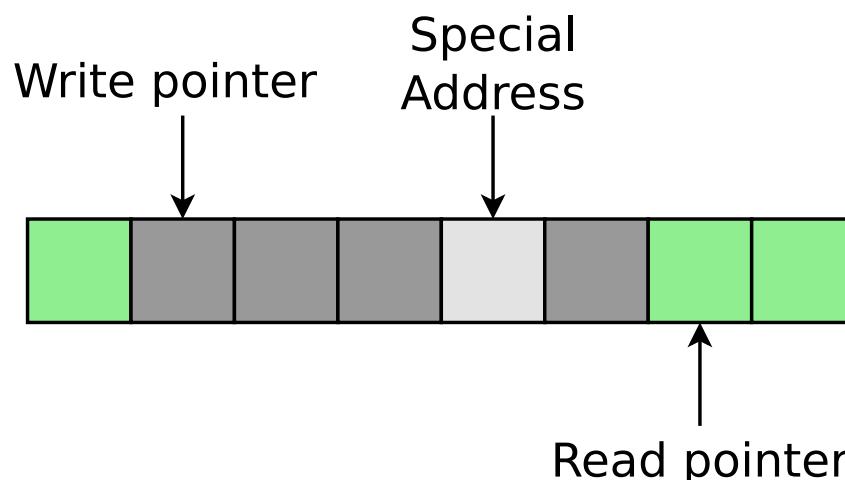
Random Delay

Sim Trace

Fixing the read

Conclusion

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

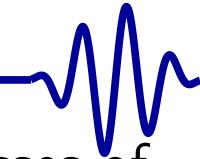


The pointers can wrap!

- What then does greater or less than mean?

Solution:

- Unwrap all the pointers by subtracting the read pointer



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design  
method

FIFO Interface

FIFO Memory

Addresses

Formal Verification  
FIFO

▷ Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

Random Delay

Sim Trace

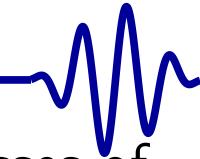
Fixing the read

Conclusion

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

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

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



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design  
method

FIFO Interface

FIFO Memory

Addresses

Formal Verification  
FIFO

▷ Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

Random Delay

Sim Trace

Fixing the read

Conclusion

To determine if f\_first\_addr is within the active addresses of the FIFO:

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

We'll need to do this to check both addresses

# First state



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
  FIFO  
▷ Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

Now we can create assertions for the first state

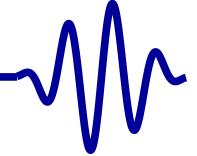
```
always @(*)
 if (f_state == 2'b01)
 begin
 // First value has been written
 assert(f_first_addr_in_fifo);
 assert(fifo_mem[f_first_addr]
 == f_first_data);
```

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

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

We now need to repeat this for the other two states

# Second state



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
  FIFO  
▷ Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

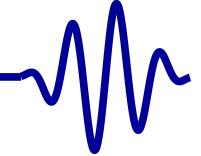
Here's the second state:

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

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

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

# Last state



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
  FIFO  
▷ Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

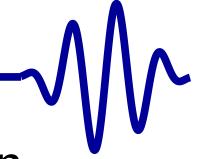
Here's the third and last state

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

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



# SymbiYosys



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design

method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO

▷ Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

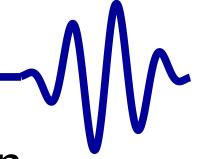
Random Delay

Sim Trace

Fixing the read

Conclusion

You should now be able to prove this design via induction



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design

method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO

▷ Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

Random Delay

Sim Trace

Fixing the read

Conclusion

You should now be able to prove this design via induction

- Does it pass?

# Cover



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
▷ Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

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

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

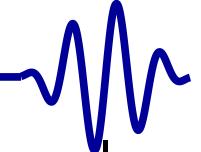
always @(posedge i_clk)
 cover(f_was_full && f_empty);

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

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



# Cover Lesson Learned



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
Cover  
▷ Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

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

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

The resulting code failed in practice



# Cover Lesson Learned



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
Cover  
▷ Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

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

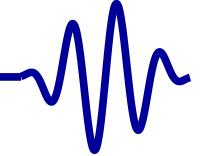
- The design was a data cache
- I had verified both interfaces
  - All the bus properties were met
  - The CPU could depend upon the resulting interface
- I covered the return from a request
  - The cache went to the bus to get the requested value
  - The values returned by the bus were properly placed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

The resulting code failed in practice

- What went wrong?



# Cover Lesson Learned



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design method  
FIFO Interface  
FIFO Memory Addresses  
Formal Verification  
FIFO Verification  
Cover  
▷ Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

## The problem?

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



# Cover Lesson Learned



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design method  
FIFO Interface  
FIFO Memory Addresses  
Formal Verification  
FIFO Verification  
Cover  
▷ Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

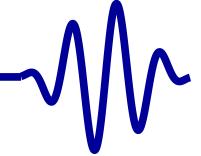
## The problem?

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

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

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

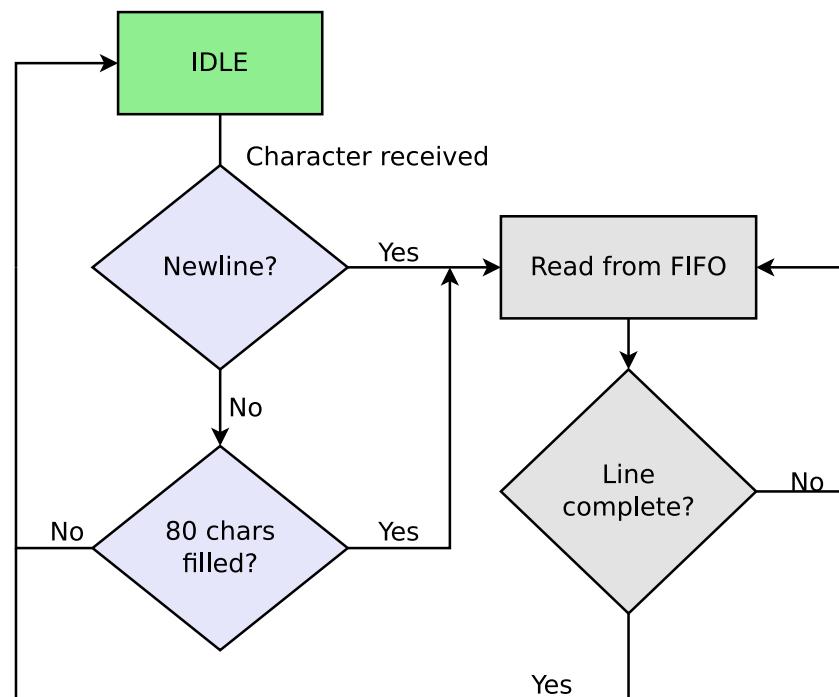
```
always @(posedge i_clk)
 cover(f_was_full && f_empty);
```



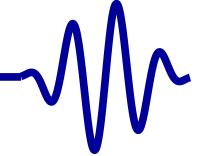
Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
▷ Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

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

- Let's build a line capturer



# Using the FIFO



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design  
method

FIFO Interface

FIFO Memory  
Addresses

Formal Verification

FIFO Verification

Cover

Cover Lesson

Line Capturer

▷ Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

Random Delay

Sim Trace

Fixing the read

Conclusion

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

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

Think of this like an old fashioned printer:

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

# GT Rx + FIFO



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design  
method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

▷ Rx + FIFO

Verifying the FSM

Simulation

Random Delay

Sim Trace

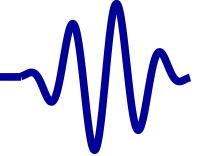
Fixing the read

Conclusion

Step one: the receiver must feed the FIFO

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

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

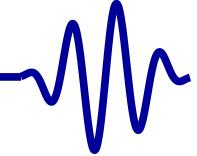


Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
▷ Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

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

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

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



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
▷ Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

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

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

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

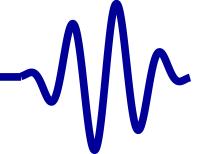


Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
▷ Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

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

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

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



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design

method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

▷ Rx + FIFO

Verifying the FSM

Simulation

Random Delay

Sim Trace

Fixing the read

Conclusion

Last steps:

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

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

- Activate the transmitter anytime run\_tx is true

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



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design

method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

▷ Rx + FIFO

Verifying the FSM

Simulation

Random Delay

Sim Trace

Fixing the read

Conclusion

Last steps:

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

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

- Activate the transmitter anytime run\_tx is true

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

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



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design

method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

▷ Rx + FIFO

Verifying the FSM

Simulation

Random Delay

Sim Trace

Fixing the read

Conclusion

Last steps:

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

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

- Activate the transmitter anytime run\_tx is true

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

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

Q: How would you know? Would a formal check help?



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design  
method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the

▷ FSM

Simulation

Random Delay

Sim Trace

Fixing the read

Conclusion

You should be able to formally verify that this works

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

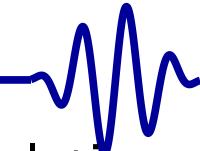
```
(* anyseq *) reg formal_data;
always @(*)
 o_data = formal_data;
```

- Focus on the FIFO flags

What properties would you use to verify this FSM design?

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

# GT Simulation



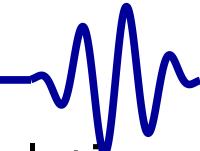
Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design method  
FIFO Interface  
FIFO Memory Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO Rx + FIFO  
Verifying the FSM  
▷ Simulation  
Random Delay  
Sim Trace  
Fixing the read  
Conclusion

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

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

What more might we need?

# GT Simulation



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design method  
FIFO Interface  
FIFO Memory Addresses  
Formal Verification  
FIFO Verification Cover  
Cover Lesson  
Line Capturer  
Using the FIFO Rx + FIFO  
Verifying the FSM  
▷ Simulation  
Random Delay Sim Trace  
Fixing the read Conclusion

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

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

What more might we need?

There's one problem: the simulation trace reveals that . . .

- The last simulation doesn't really exercise our design

Let's fix this!



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
▷ Random Delay  
Sim Trace  
Fixing the read  
Conclusion

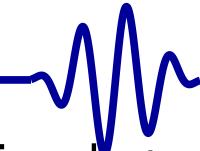
Here's the key: let's add a random delay between incoming bytes

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

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



# Random Delay



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
▷ Random Delay  
Sim Trace  
Fixing the read  
Conclusion

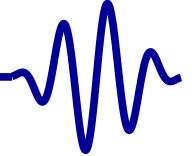
Here's the key: let's add a random delay between incoming bytes

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

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

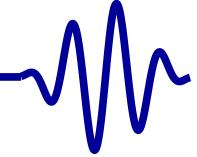


# Simulation



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
▷ Random Delay  
Sim Trace  
Fixing the read  
Conclusion

You should now be able to run the simulation



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design

method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

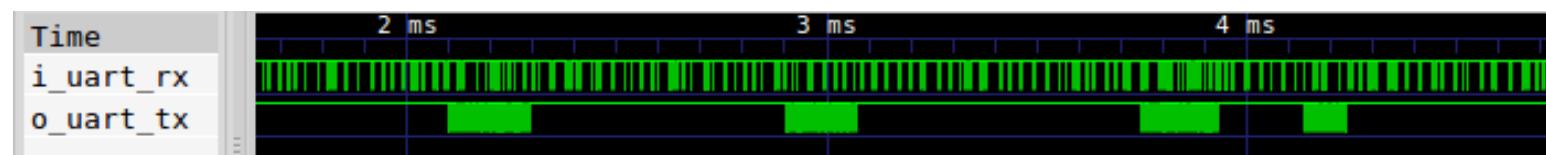
Random Delay

▷ Sim Trace

Fixing the read

Conclusion

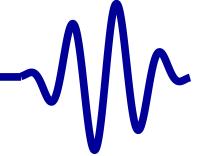
Here's the trace I got from running the simulation



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

This was exactly what we wanted!

# Fixing the read



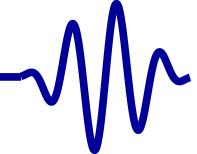
Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design  
method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
▷ Fixing the read  
Conclusion

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

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

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

How shall we fix this?



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
▷ Fixing the read  
Conclusion

## Solution one: bypass the memory

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

Let's see what this would look like



# Bypass memory



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
▷ Fixing the read  
Conclusion

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

- The first is placed in a register

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

- The second is read from the memory

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

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



# Bypass memory



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design  
method

FIFO Interface

FIFO Memory  
Addresses

Formal Verification

FIFO Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

Random Delay

Sim Trace

▷ Fixing the read

Conclusion

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

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

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



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design  
method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

Random Delay

Sim Trace

▷ Fixing the read

Conclusion

The trick is the selector, bypass\_valid, that tells us which value to return

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



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design  
method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

Random Delay

Sim Trace

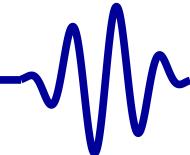
▷ Fixing the read

Conclusion

The trick is the selector, bypass\_valid, that tells us which value to return

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

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



## Solution two: Work with the registered output

- We could just use the registered data

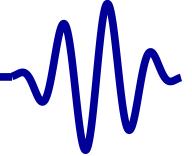
```
always @ (posedge i_clk)
 o_data <= mem[(w_rd)?rd_next
 : rd_addr[LGLEN - 1:0]];
```

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

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



# Exercise #1



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design  
method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

Random Delay

Sim Trace

▷ Fixing the read

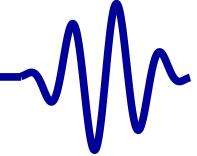
Conclusion

Pick a solution to our memory problem and

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



# Exercise #2



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design method  
FIFO Interface  
FIFO Memory  
Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM Simulation  
Random Delay  
Sim Trace  
▷ Fixing the read  
Conclusion

All of our o\_fill and o\_full logic is combinatorial

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

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

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

- Use the formal solver to formally prove that it still meets the properties required of o\_fill and o\_full



# Exercise



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design  
method

FIFO Interface

FIFO Memory

Addresses

Formal Verification

FIFO Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

Random Delay

Sim Trace

▷ Fixing the read

Conclusion

Are you ready? Let's build this!

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

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

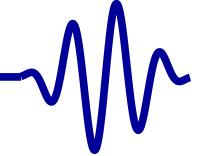
# Questions



Lesson Overview  
Design Goal  
What is a FIFO?  
Basic FIFO Design method  
FIFO Interface  
FIFO Memory Addresses  
Formal Verification  
FIFO Verification  
Cover  
Cover Lesson  
Line Capturer  
Using the FIFO  
Rx + FIFO  
Verifying the FSM  
Simulation  
Random Delay  
Sim Trace  
▷ Fixing the read  
Conclusion

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

# Conclusion



Lesson Overview

Design Goal

What is a FIFO?

Basic FIFO Design  
method

FIFO Interface

FIFO Memory  
Addresses

Formal Verification

FIFO Verification

Cover

Cover Lesson

Line Capturer

Using the FIFO

Rx + FIFO

Verifying the FSM

Simulation

Random Delay

Sim Trace

Fixing the read

► Conclusion

What did we learn this lesson?

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