Gisselquist
Technology, LLC

4. Pipeline Control

Daniel E. Gisselquist, Ph.D.

Lesson Overview

Objectives

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

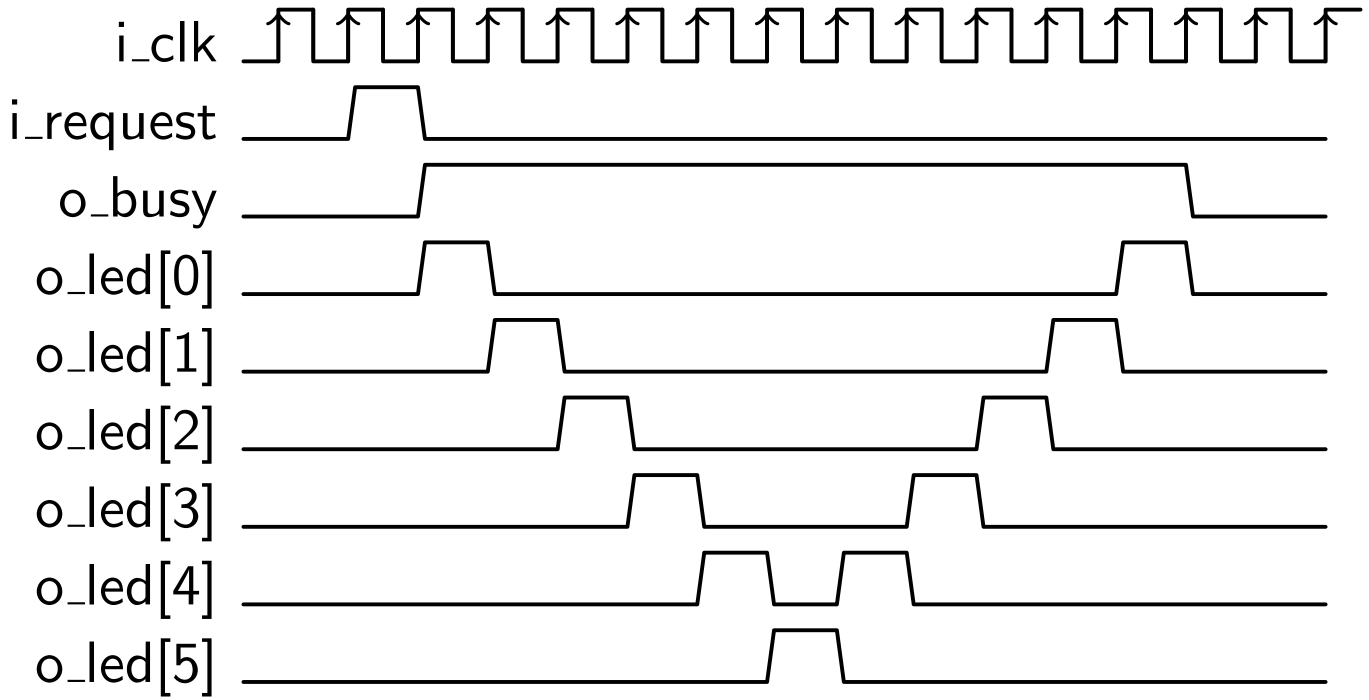
LED Walker

Let's make our LED's walk on command

- Bus requests
- State Diagram

Goal

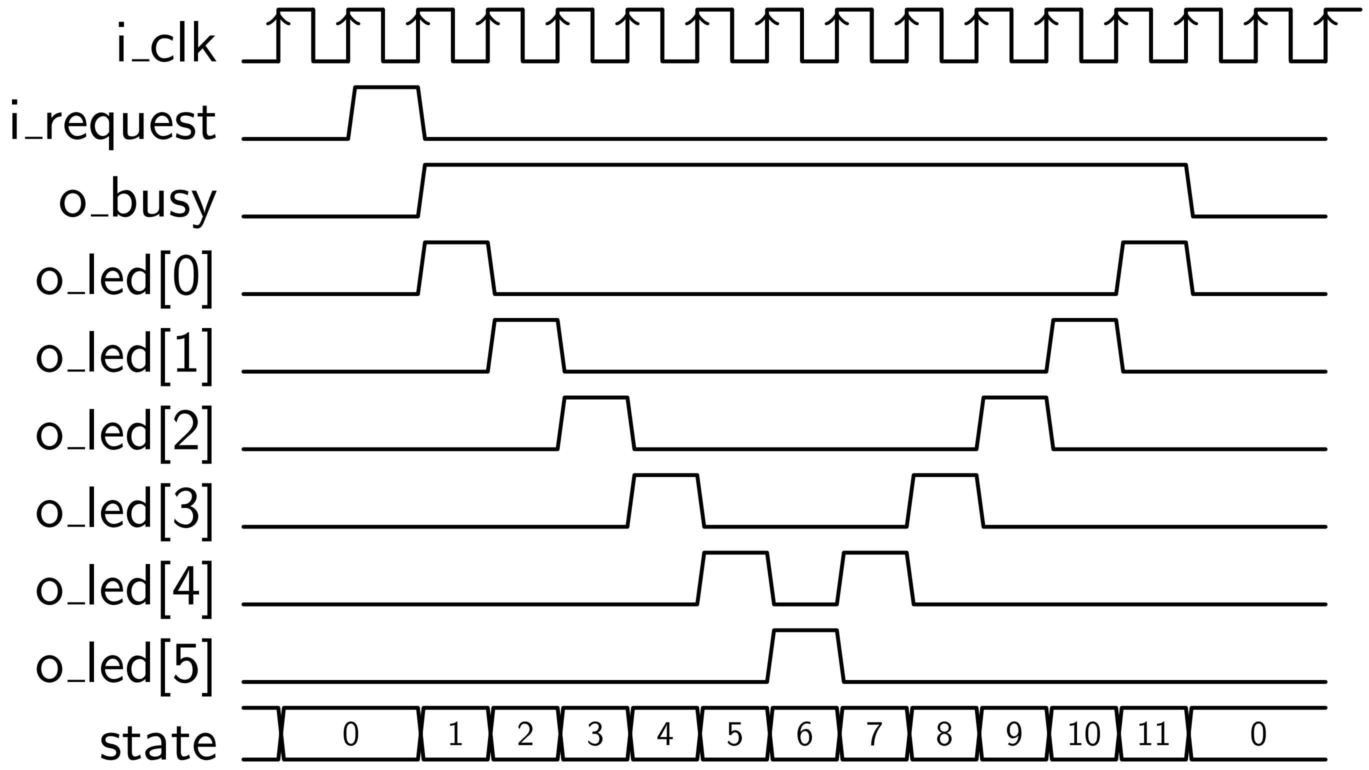
Let's adjust our LED sequence to require a request



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

Goal

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

State Transition

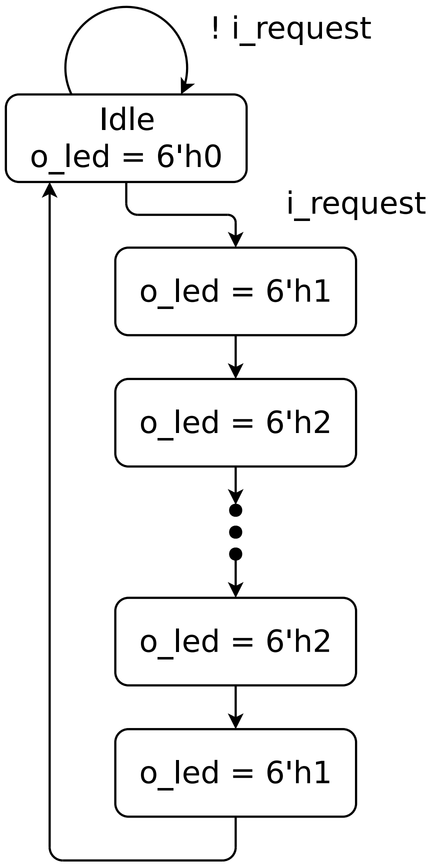
The key to this design is the idle state

- 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);
```

State Transition Diagrams



- 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

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

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

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

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

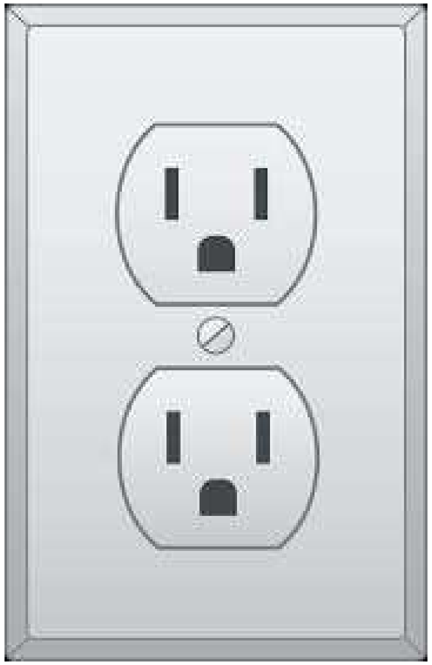
```
// Today's logic: Wait for the request
always @(posedge i_clk)
if ((i_request)&&(!o_busy))
        state <= 4'h1;
else if (state >= 4'hB)
        state <= 4'h0;
else if (state != 0)
        state <= state + 1'b1;
```

Above: A mix of pipeline and state machine logic

This is fairly common

Bus

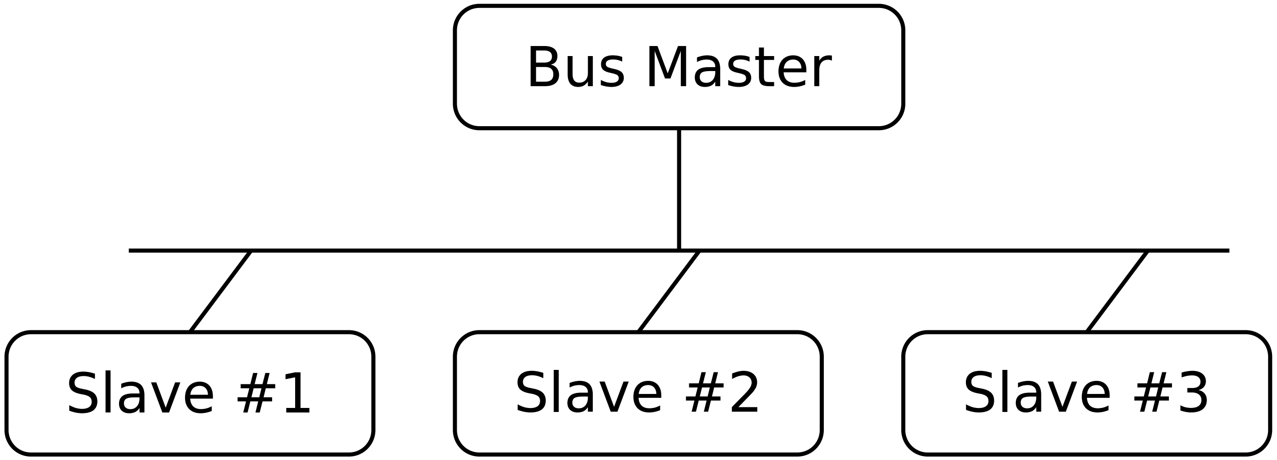
Interface standards simplify plugging things in



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

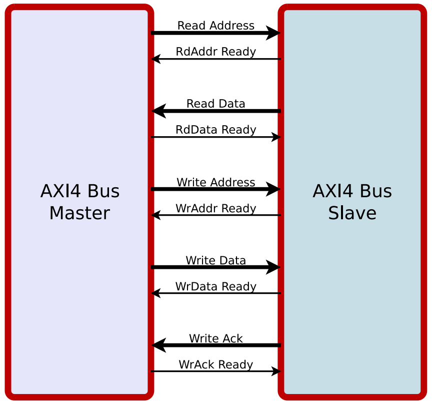
Bus Topology



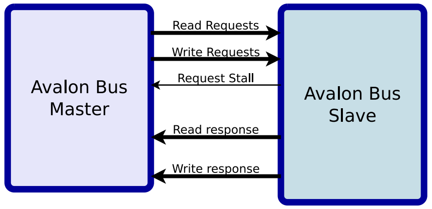
- Every bus has a master
- A Bus may have many slaves
  Slaves are differentiated by their address
- All connected via an *interconnect*
- A slave on one bus may be a master on another

Many Bus Standards

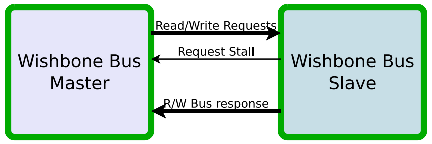
There are many bus standards



AXI



Avalon



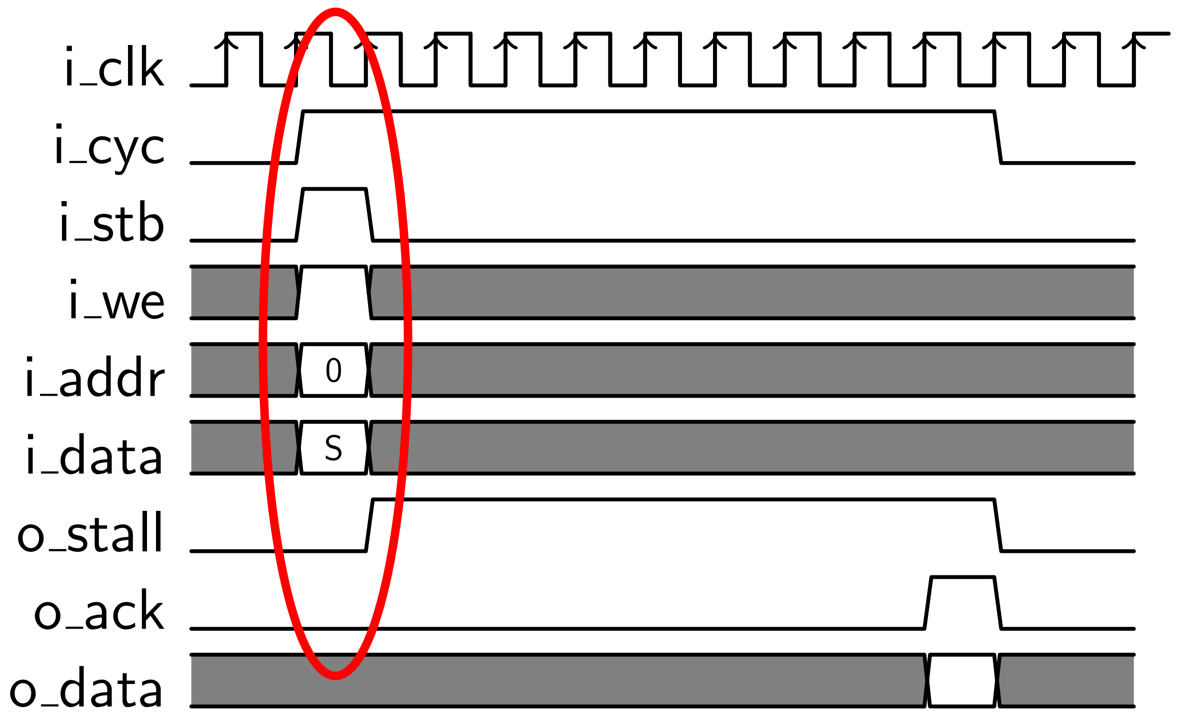
Wishbone

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

Wishbone Bus

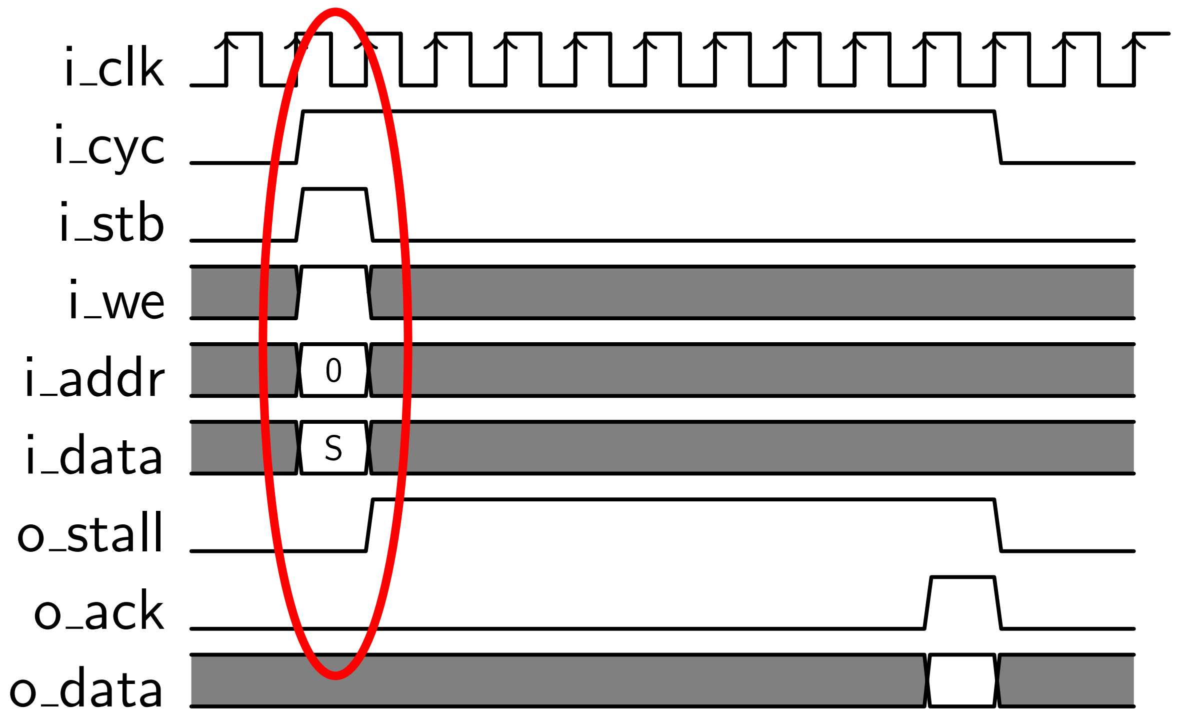
I use Wishbone B4, pipelined mode exclusively



- 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)&&(!o_stall)` isn't true

Wishbone Bus

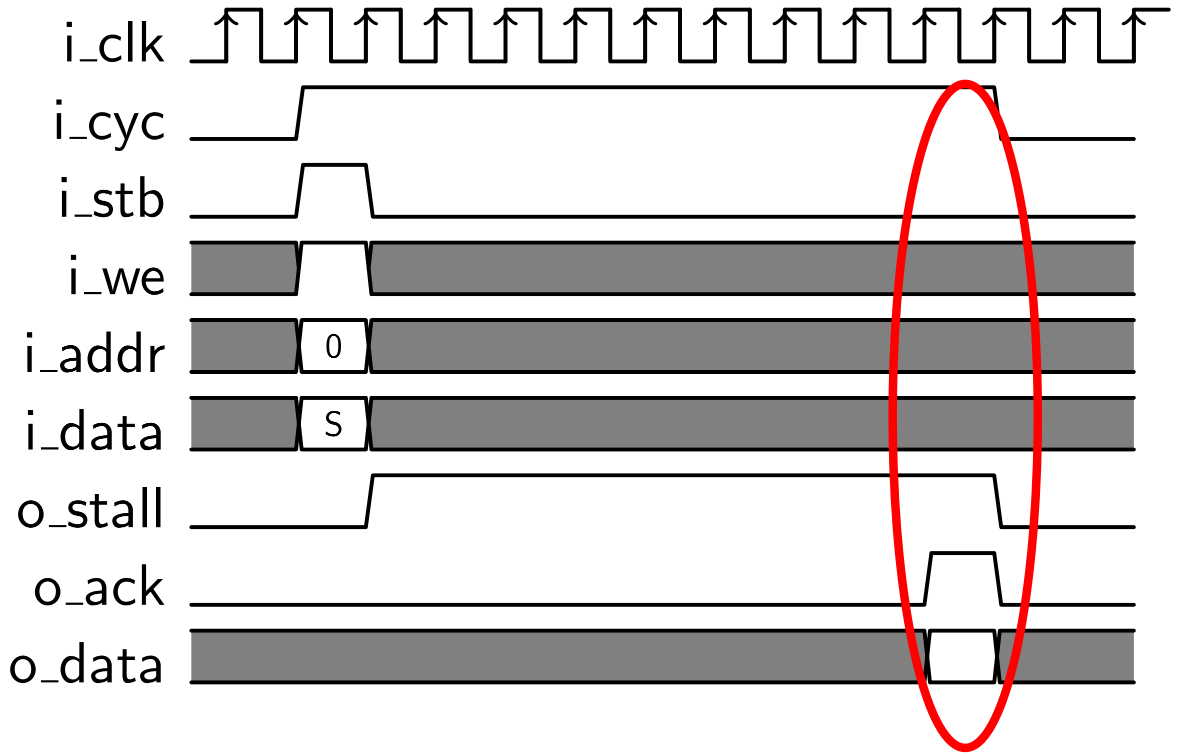
I use Wishbone B4, pipelined mode exclusively



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

Wishbone Bus

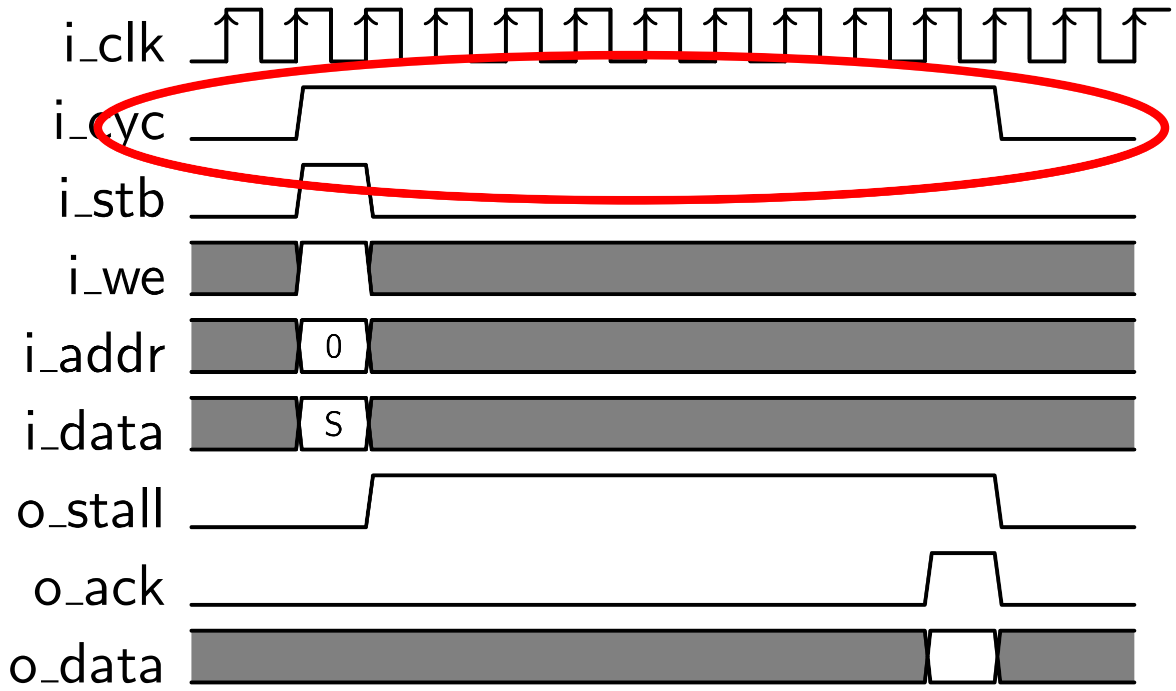
I use Wishbone B4, pipelined mode exclusively



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

Wishbone Bus

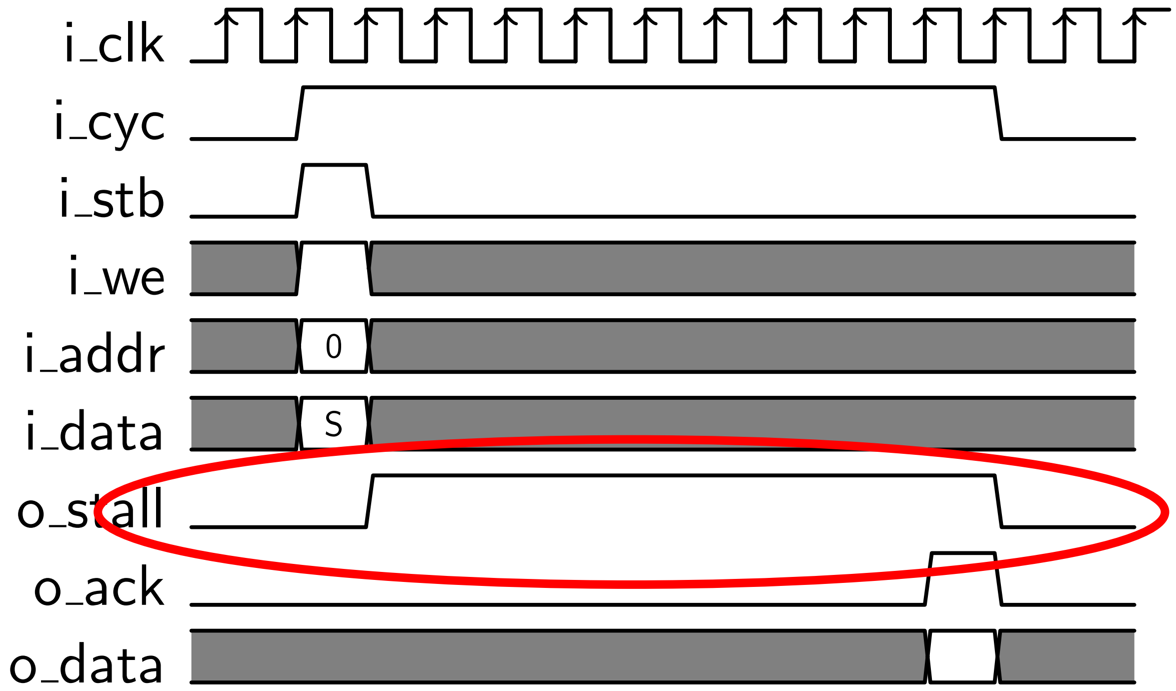
I use Wishbone B4, pipelined mode exclusively



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

Wishbone Bus

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

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

- 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 };
```

Simulation

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

```
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

```
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

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␣state␣is:␣0x%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)
```

Display State

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

```
% 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

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

Getting Verilator to ignore unused logic

- Use the `// Verilator lint_off UNUSED` command

```
// Verilator lint_off UNUSED
wire    [33:0]  unused;
assign  unused = { i_cyc, i_addr, i_data };
// Verilator lint_on  UNUSED
```

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

Sim Exercise

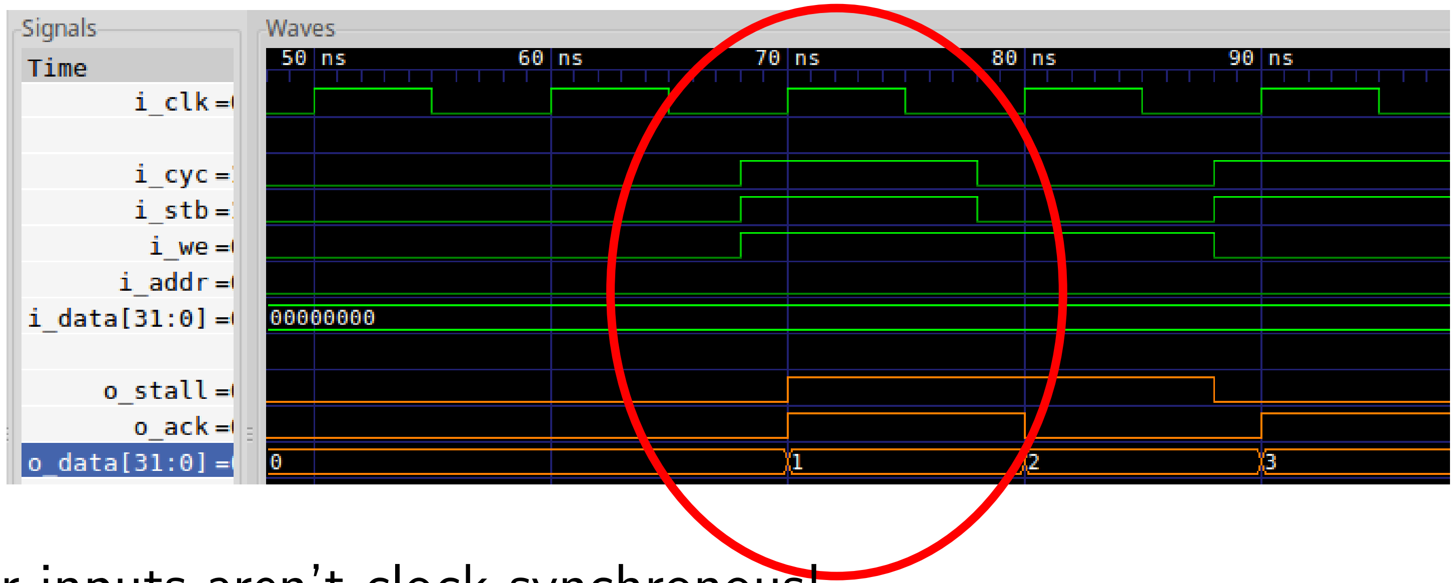
Build and run the demo

- Examine the trace
- Examine the output

Does it work like you expected?

Trace bias

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

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);
      // ...
```

Trace bias

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);
        // ...
```

Trace bias

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

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.

Simulation output

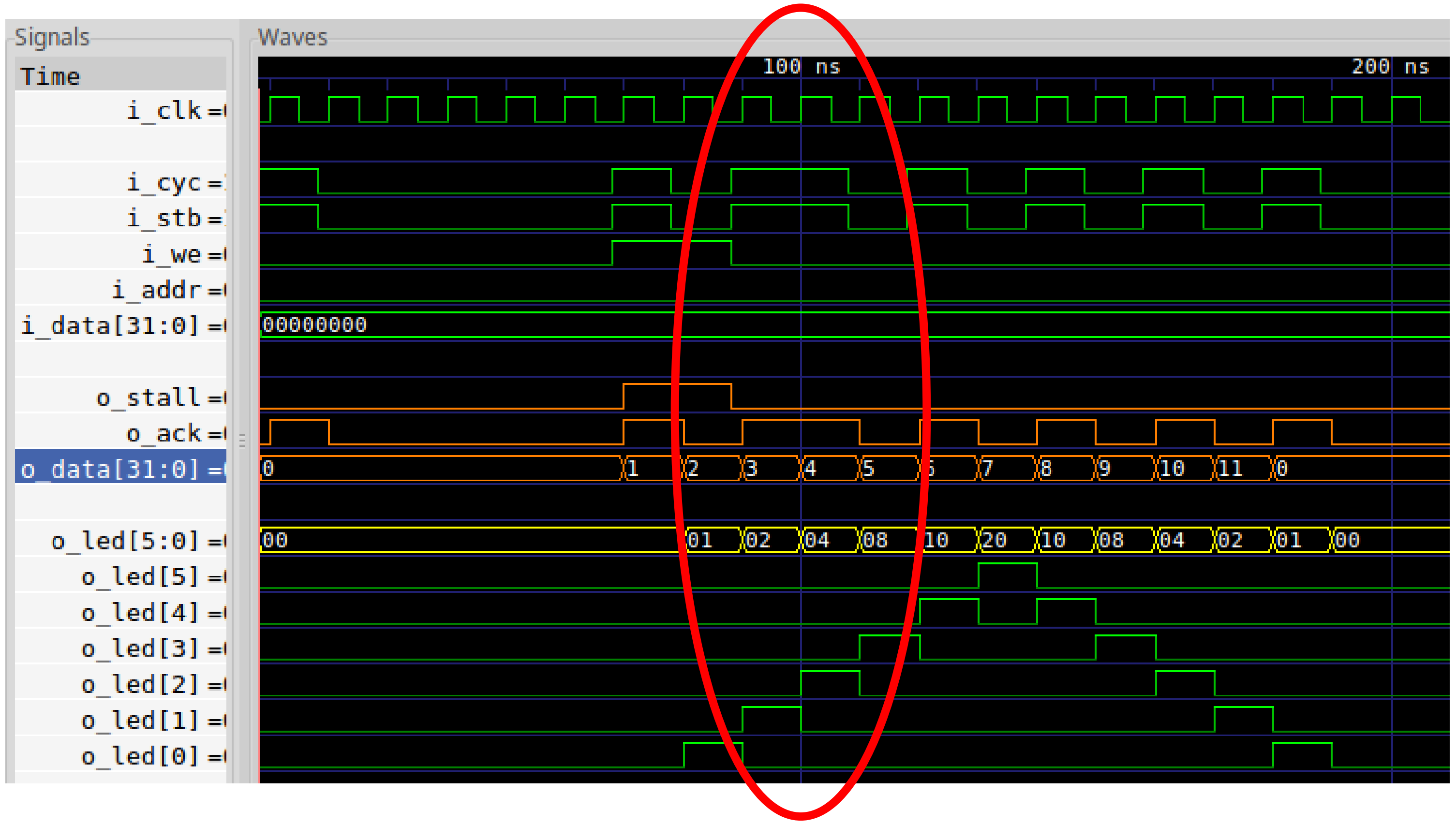
Is this an output you expected?

```
% ./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!

Double ACKs

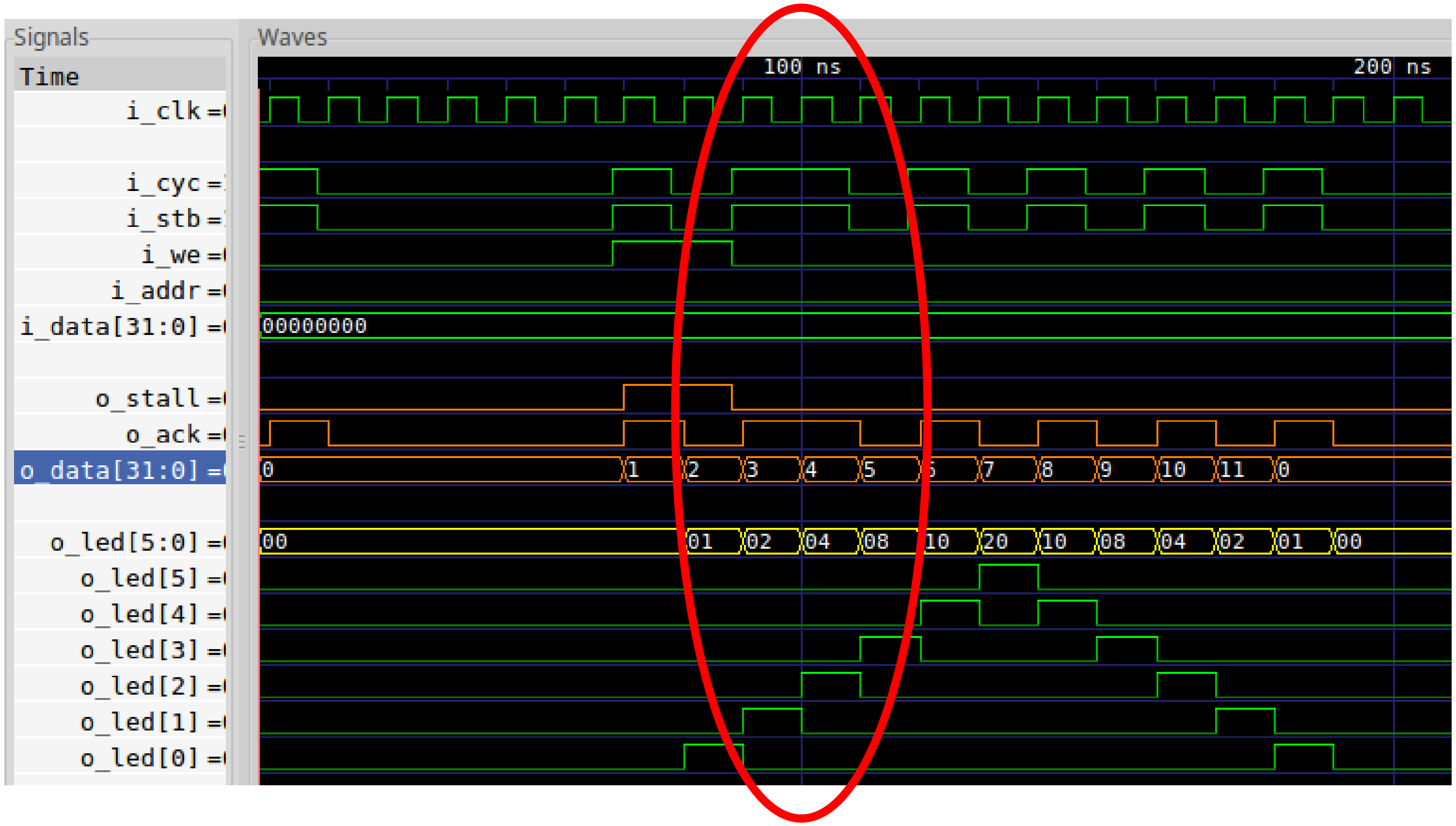
Look at the trace. Can you explain this?



- 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

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

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

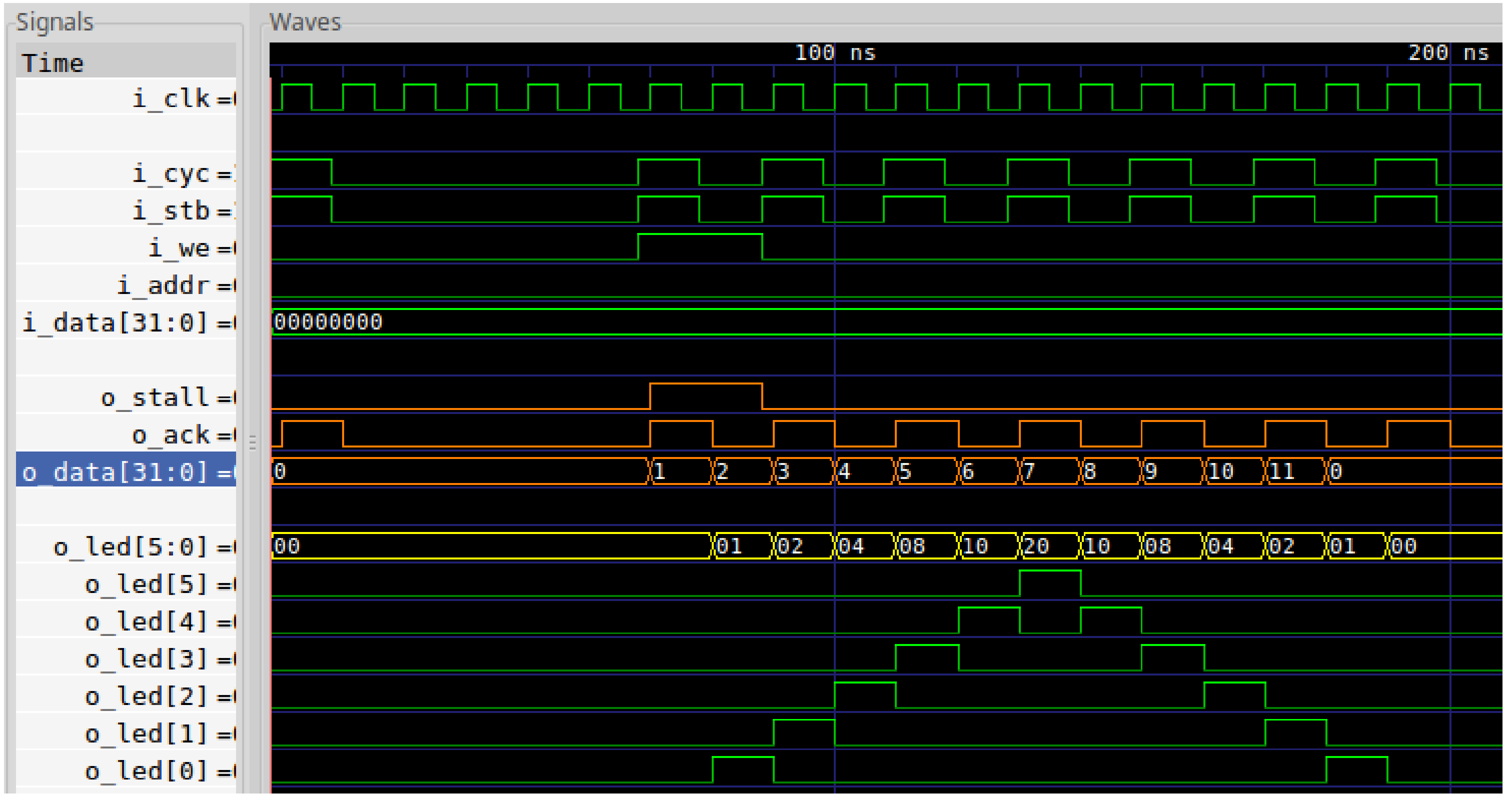
Rebuild and run again. Is this better?

```
% ./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?

Exercise

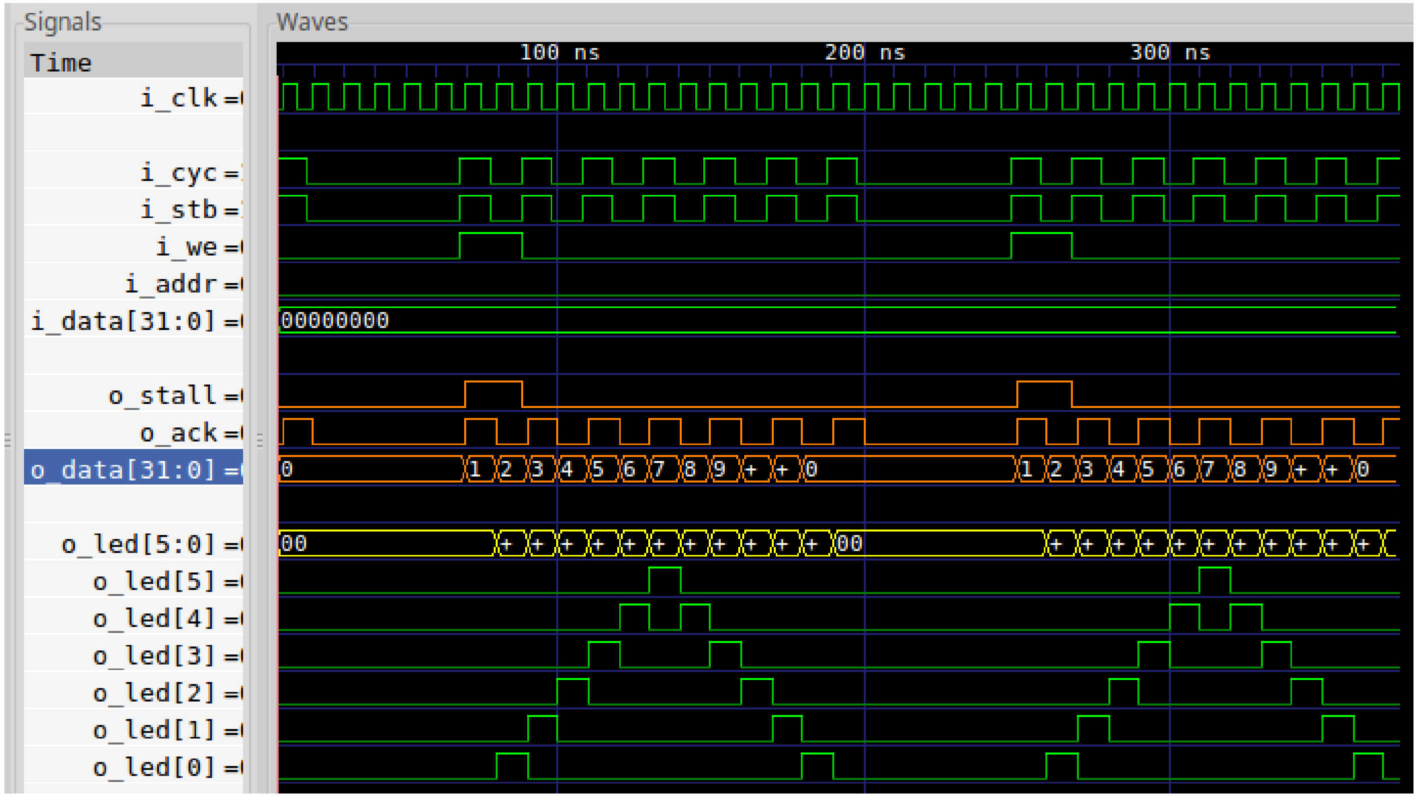
Look at the ACK's



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

Sim Exercise

Here's the full and final simulation



Here you can see both LED walks, as expected

Formal past operator

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);
```

Formal past operator

$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);
```

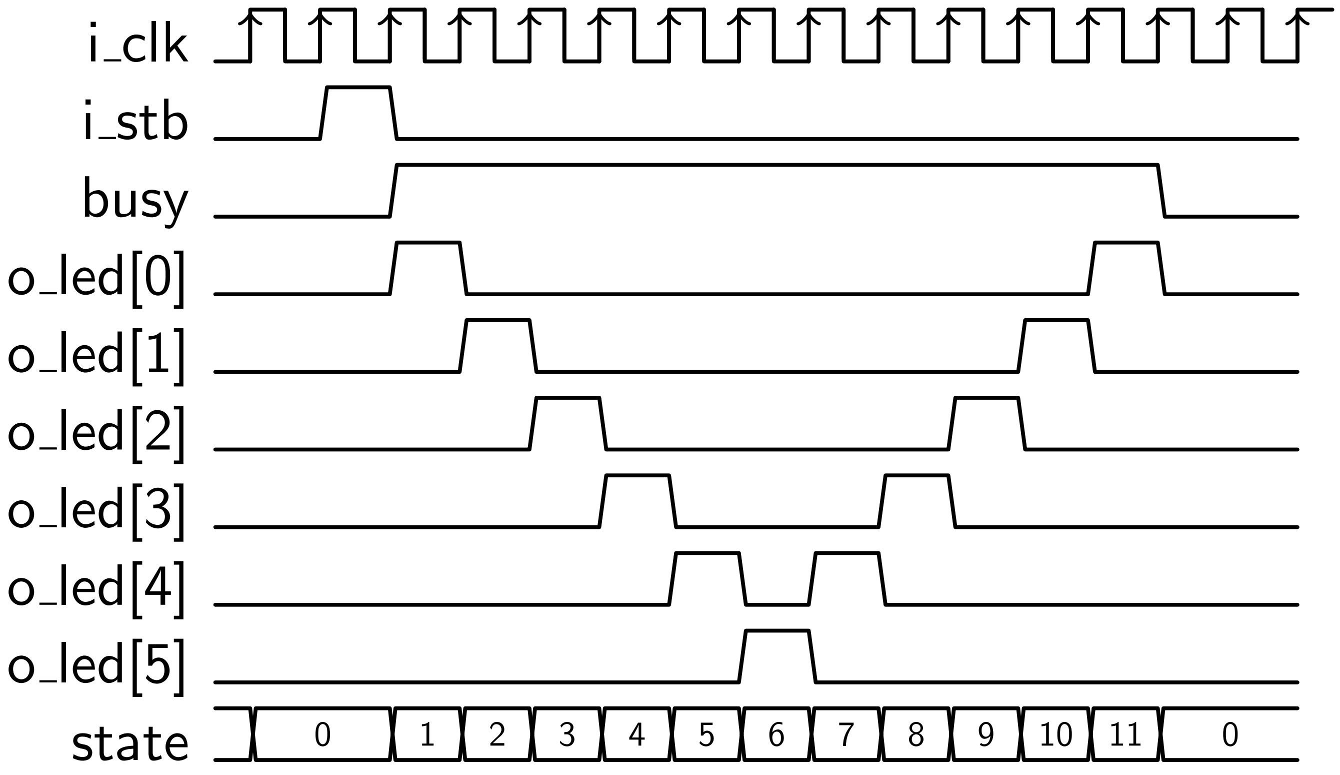
Formal Verification

What properties might we use?

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

Formal Verification

What properties might we use?



The goal waveform diagram should give you an idea

Formal Verification

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

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

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

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

What properties might we use?

- For our bus interface?

```
always @(posedge i_clk)
if ((f_past_valid)&&($past(i_stb))
        &&($past(busy)))
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

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

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

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

SymbiYosys Tasks

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

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

We can now run a named task

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

. . . or all tasks in sequence

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

SymbiYosys Tasks

I use this often with the ZipCPU

- Using the yosys command `chparam` 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

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

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

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

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

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