



Gisselquist
Technology, LLC

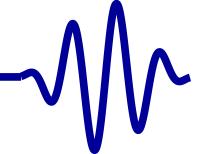
An Introduction to Formal Methods

Daniel E. Gisselquist, Ph.D.





Lessons



▷ Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Day one

1. Motivation
2. Basic Operators
3. Clocked Operators
4. Induction
5. Bus Properties

Day two

6. Free Variables
7. Abstraction
8. Invariants
9. Multiple-Clocks
10. Cover
11. Sequences
12. Final Thoughts



Course Structure



▷ Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

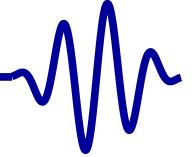
Multiple-Clocks

Cover

Sequences

Quizzes

- We'll be primarily using the *immediate assertion* subset of the full SystemVerilog assertion language
- Each lesson will be followed by an exercise
There are 12 exercises
- My goal is to have 50% lecture, 50% exercises
- Leading up to building a bus arbiter
and testing an synchronous FIFO



Welcome

▷ Motivation

Intro

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

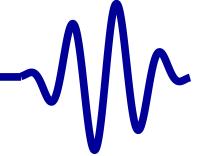
Sequences

Quizzes

Motivation



Lesson Overview



Welcome

Motivation

▷ Intro

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

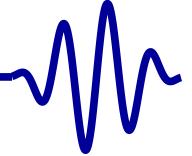
1. Why are you here?
2. What can I provide?
3. What have I learned from formal methods?

Our Objectives

- Get to know a little bit about each other
- Motivate further discussion



Your expectations



Welcome

Motivation

▷ Intro

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

What do you want to learn and get out of this course?



From an ARM dev.



Welcome

Motivation

▷ Intro

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

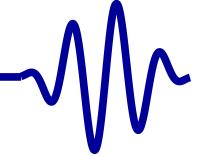
Quizzes

- “I think the main difference between FPGA and ASIC development is the level of verification you have to go through. Shipping a CPU or GPU to Samsung or whoever, and then telling them once they’ve taped out that you have a Cat1 bug that requires a respin is going to set them back \$1M per mask.”
- “... But our main verification is still done *with constrained random test benches written in SV*.
- “Overall, you are looking at 50 man years per project minimum for an average project size.”

[Welcome](#)[Motivation](#)[▷ Intro](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

“If we would not do formal verification, we would
no longer exist.”

– Shahar Ariel, Head of VLSI design at Mellanox



Welcome

Motivation

▷ Intro

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

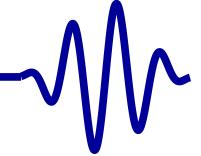
One little mistake . . .

. . . \$475M later.

[Welcome](#)[Motivation](#)[Intro](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

I have proven such things as,

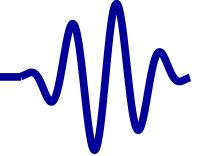
- Formal bus properties (Wishbone, Avalon, AXI, etc.)
- Bus bridges (WB-AXI, Avalon-WB, WB-WB)
- Prefetches, cache controllers, memory controllers, MMU
- SPI based A/D controllers
- SDRAM
- UART, both TX and RX
- FIFO's, signal processing flows, DSP delay
- Display (VGA) Controller
- LFSR's
- Flash controllers
- Formal proof of the ZipCPU

[Welcome](#)[Motivation](#)[Intro](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

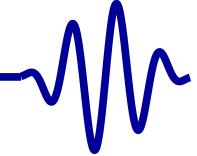
I've found bugs in things I thought were working.

1. FIFO
2. Pre-fetch and Instruction cache
3. SDRAM
4. A peripheral timer

Just how hard can a timer be to get right? It's just a counter!

[Welcome](#)[Motivation](#)▷ [Intro](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

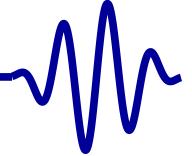
- *It worked in my test bench*
- Failed when reading and writing on the same clock while empty
 - Write first then read worked
 - R+W on full FIFO is okay
 - R+W on an empty FIFO

[Welcome](#)[Motivation](#)▷ [Intro](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

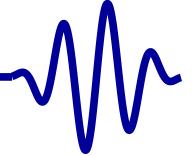
- *It worked in my test bench*
- Failed when reading and writing on the same clock while empty
 - Write first then read worked
 - R+W on full FIFO is okay
 - R+W on an empty FIFO . . . **not so much**
- My test bench didn't check that, formal did

[Welcome](#)[Motivation](#)[▶ Intro](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

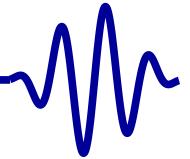
- *It worked in my test bench*
- Ugliest bug I ever came across was in the prefetch cache
It passed test-bench muster, but failed in the hardware with a
strange set of symptoms
- When I learned formal, it was easy to prove that this would
never happen again.
- Low logic has always been one of my goals.
Always asking, “will it work if I get rid of this condition?”
Formal helps to answer that question for me.

[Welcome](#)[Motivation](#)[▷ Intro](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- *It worked in my test bench*
- It passed my hardware testing
 - Test S/W: Week+, no bugs

[Welcome](#)[Motivation](#)[▷ Intro](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- *It worked in my test bench*
- **It passed my hardware testing**
 - Test S/W: Week+, no bugs
 - Formal methods found the bug
 - Full proof took less than < 30 min



Welcome

Motivation

▷ Intro

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

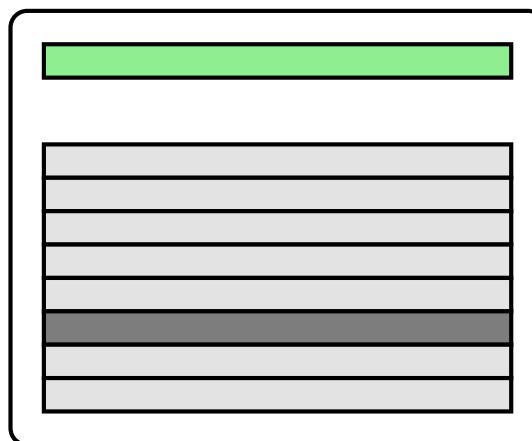
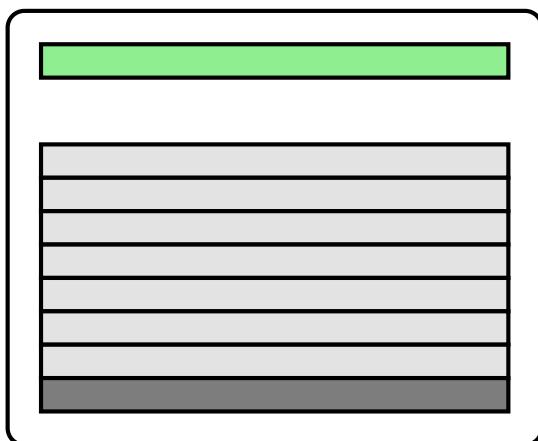
Multiple-Clocks

Cover

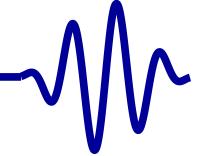
Sequences

Quizzes

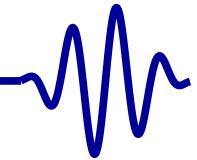
- *It worked in my test bench*
- It passed my hardware testing
- Background



• • •

[Welcome](#)[Motivation](#)[▶ Intro](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- *It worked in my test bench*
- It passed my hardware testing
- Background
 - SDRAM's are organized into separate banks, each having rows and columns
 - A row must be “activated” before it can be used.
 - The controller must keep track of which row is activated.
 - If a request comes in for a row that isn't activated, the active row must be deactivated, and the proper row must be activated.
- A subtle bug in my SDRAM controller compared the active row address against the immediately previous (1-clock ago) required row address, not the currently requested address. This bug had lived in my code for years. Formal methods caught it.



Welcome

Motivation

▷ Intro

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

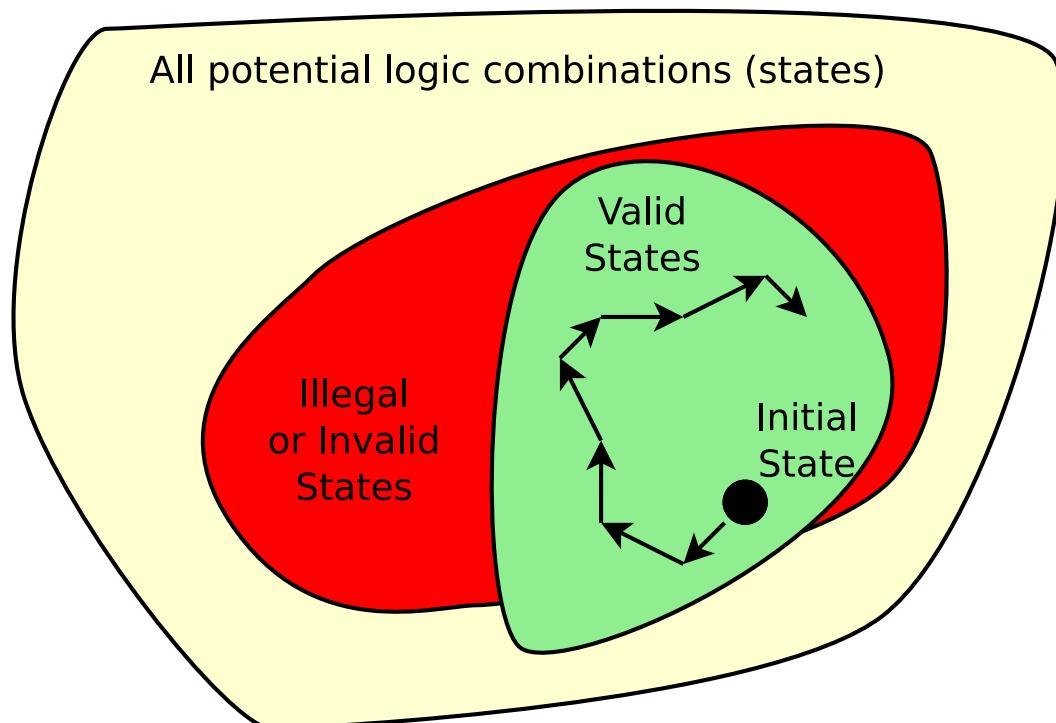
Invariants

Multiple-Clocks

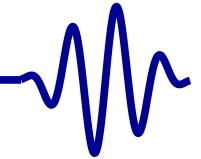
Cover

Sequences

Quizzes



- Only examines a known good branch
- Cannot check for every out of bounds conditions

[Welcome](#)[Motivation](#)[▶ Intro](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- Demonstrate code works
- Through a *normal* working path
 - or a limited number of extraneous paths
- Never rigorous enough to check everything
- Not uniform in rigour

For the FIFO,

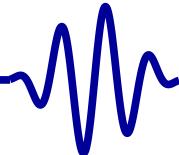
- I only read when I knew it wasn't empty

For the Prefetch,

- I never tested jumping to the last location in a cache line

For the SDRAM,

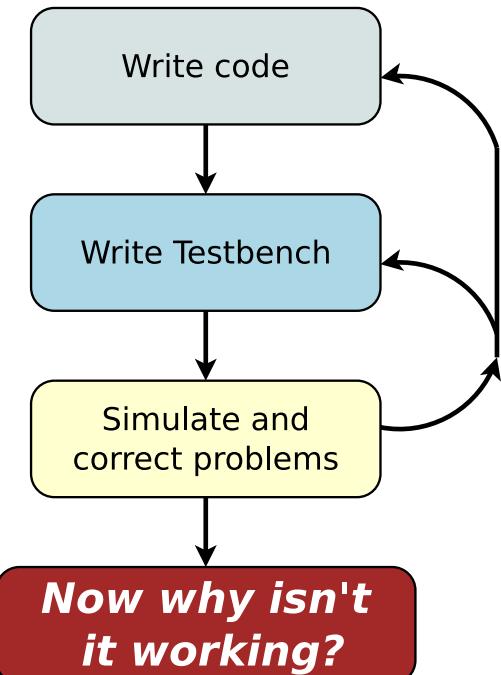
- The error was so obscure, it would be hard to trigger

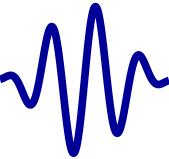
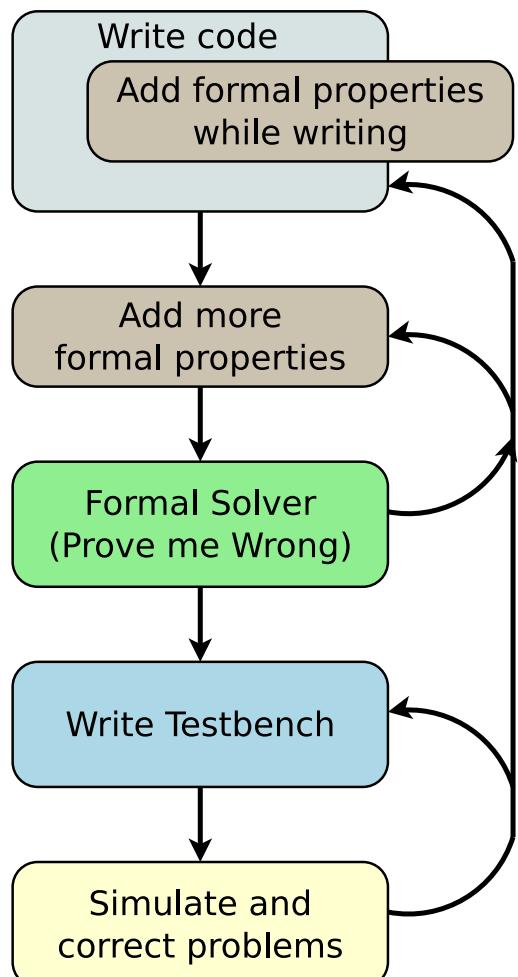
[Welcome](#)[Motivation](#)[Intro](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

This was my method before starting to work with formal.

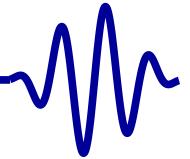
- After . . .
 - Proving my code with test benches
 - Directed simulation
- I was still chasing bugs in hardware

I still use this approach for DSP algorithms.



[Welcome](#)[Motivation](#)▷ [Intro](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- After finding the bug in my FIFO ... I was hooked.
- Rebuilding everything ... now using formal
- Formal found more bugs ... in example after example
- *I'm hooked!*



Welcome

Motivation

▷ Intro

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

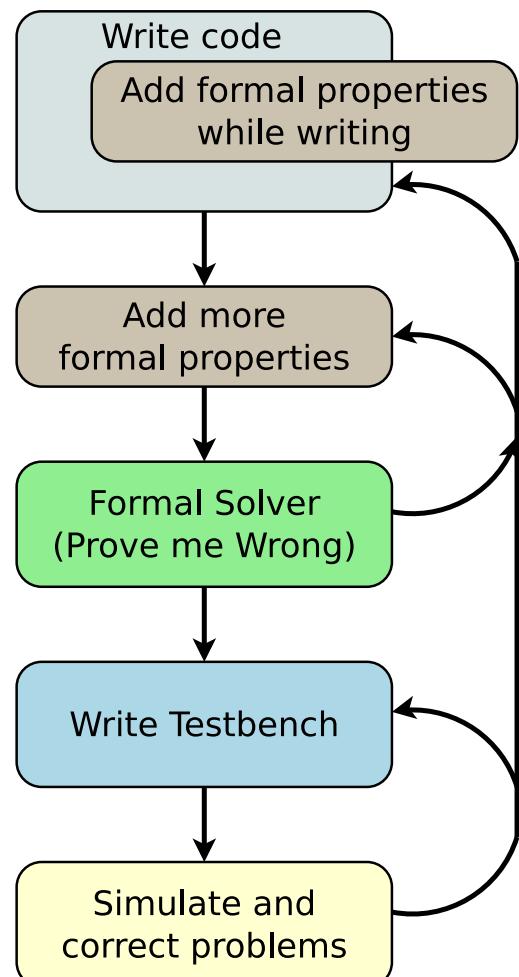
Invariants

Multiple-Clocks

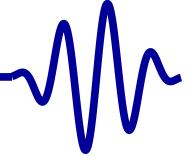
Cover

Sequences

Quizzes



- Bus component
I would not build a bus component without formal any more
- Multiplies
Formal struggles with multiplication



Welcome

Motivation

▷ Basics

Basics

General Rule

Assert

Assume

BMC

Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Formal Verification

Basics: assert and assume



Lesson Overview



Welcome

Motivation

Basics

▷ Basics

General Rule

Assert

Assume

BMC

Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Let's start at the beginning, and look at the very basics of formal verification.

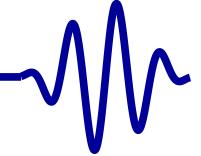
Our Objective:

- To learn the basic two operators used in formal verification,
 - **assert()**
 - **assume()**
- To understand how these affect a design from a state space perspective
- We'll also look at several examples

[Welcome](#)[Motivation](#)[Basics](#)[▷ Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Formal methods are built around looking for redundancies.

- Basic difference between mediocre and excellent:
Double checking your work
- Two separate and distinct fashions
 - First method calculates the answer
 - Second method proved it was right
- Example: Division
 - $89,321/499 = 179$
 - Does it? Let's check: $179 * 499 = 89,321$ — Yes
- Formal methods are similar
 - Your code is the first method
 - Formal properties describe the second

[Welcome](#)[Motivation](#)[Basics](#)[▷ Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

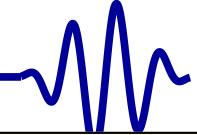
There are really only two basic operators

1. **assume()**

An **assume(X)** statement will limit the state space that the formal verification engine examines.

2. **assert()**

An **assert(X)** statement indicates that X *must* be true, or the design will fail to prove.

[Welcome](#)[Motivation](#)[Basics](#)[▷ Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

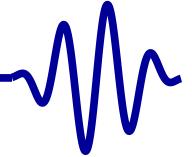
```
always @(*)  
    assert(x);
```

// Use when your property has clock dependencies,
// such as referencing an items value in the past

```
always @(posedge clk)  
    assert(x);
```

As an example,

```
always @(*)  
    assert(counter < 20);
```



Welcome

Motivation

Basics

Basics

▷ General Rule

Assert

Assume

BMC

Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

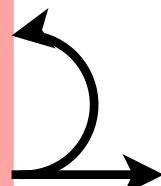
Sequences

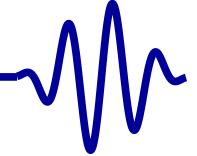
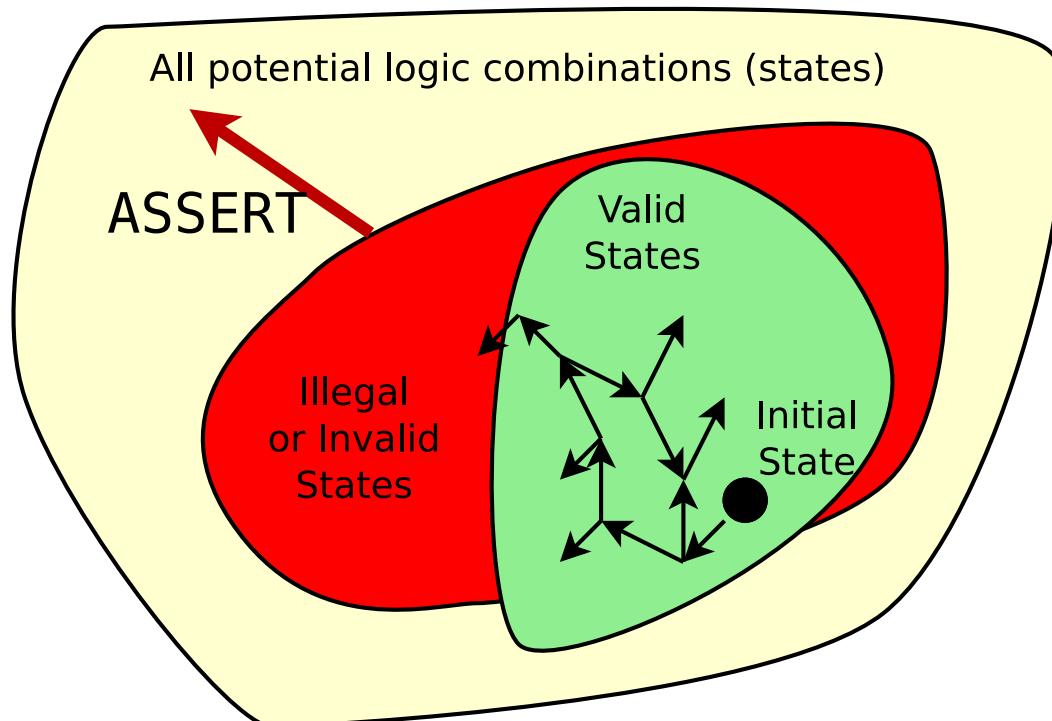
Quizzes

Master FV Rule

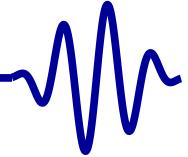
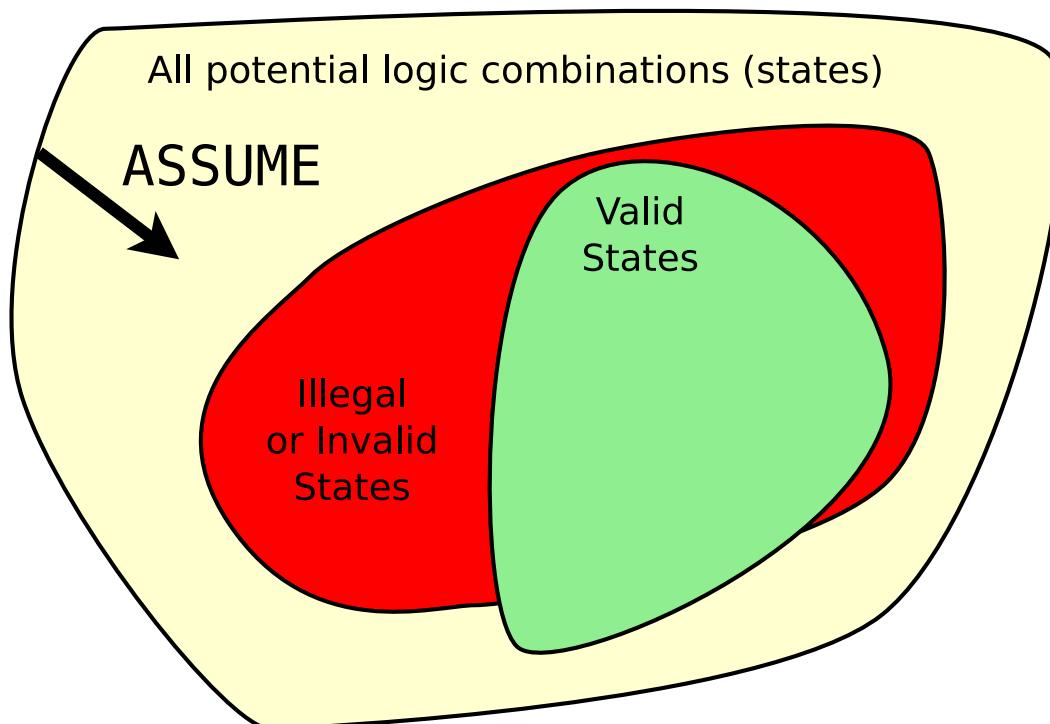
→ assume(inputs);

assert(local state);
assert(outputs);

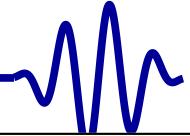


[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[▷ Assert](#)[Assume](#)[BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- Assertions define the *illegal* state space.
- Additional assertions will increase the size of the *illegal* state space.

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[▷ Assume](#)[BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- Assumptions limit the universe of all possibilities
- Additional assumptions will decrease the size of the *total* state space
- *Caution:* One careless assumption can void the proof

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[▷ Assume](#)[BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

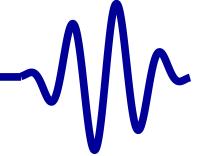
```
reg [15:0] counter;  
  
initial counter = 0;  
always @ (posedge clk)  
    counter <= counter + 1'b1;  
  
always @ (*)  
begin  
    assert(counter <= 100);  
    assume(counter <= 90);  
end
```

Question: Will counter ever reach 120?

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[▷ Assume](#)[BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

restrict () is very similar to **assume()**

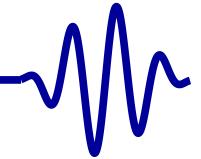
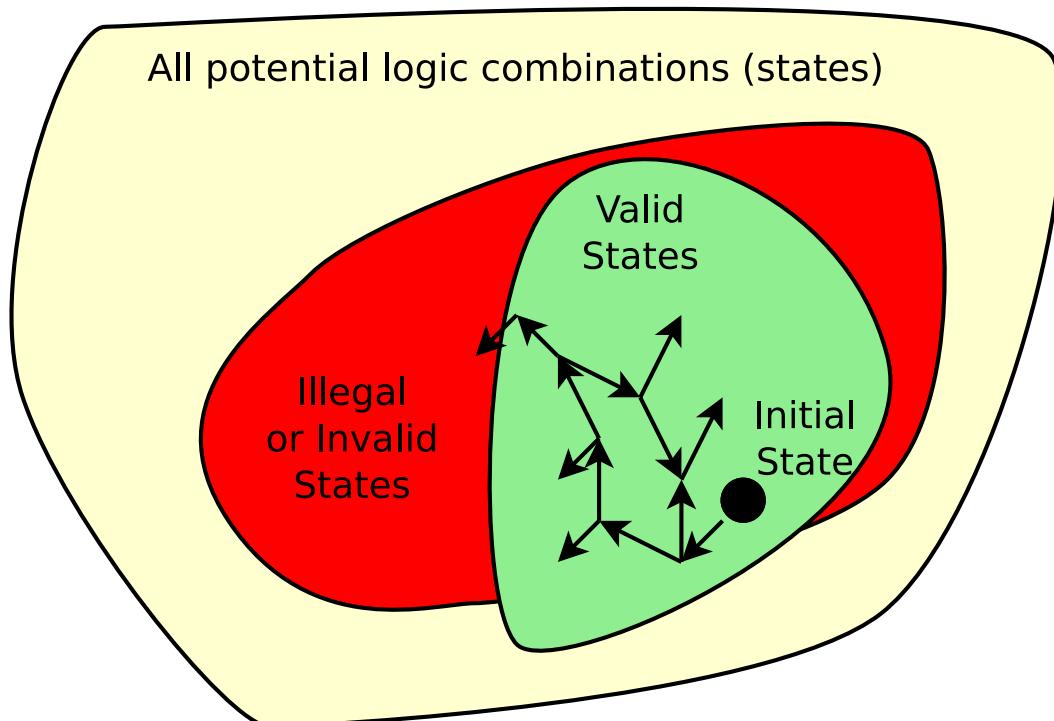
Operator	Formal Verification	Traditional Simulation
restrict ()	Restricts search space	Ignored
assume()		Halts simulation with an error
assert()	Illegal state	

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[▷ Assume](#)[BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

restrict () is very similar to **assume()**

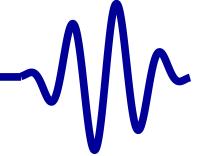
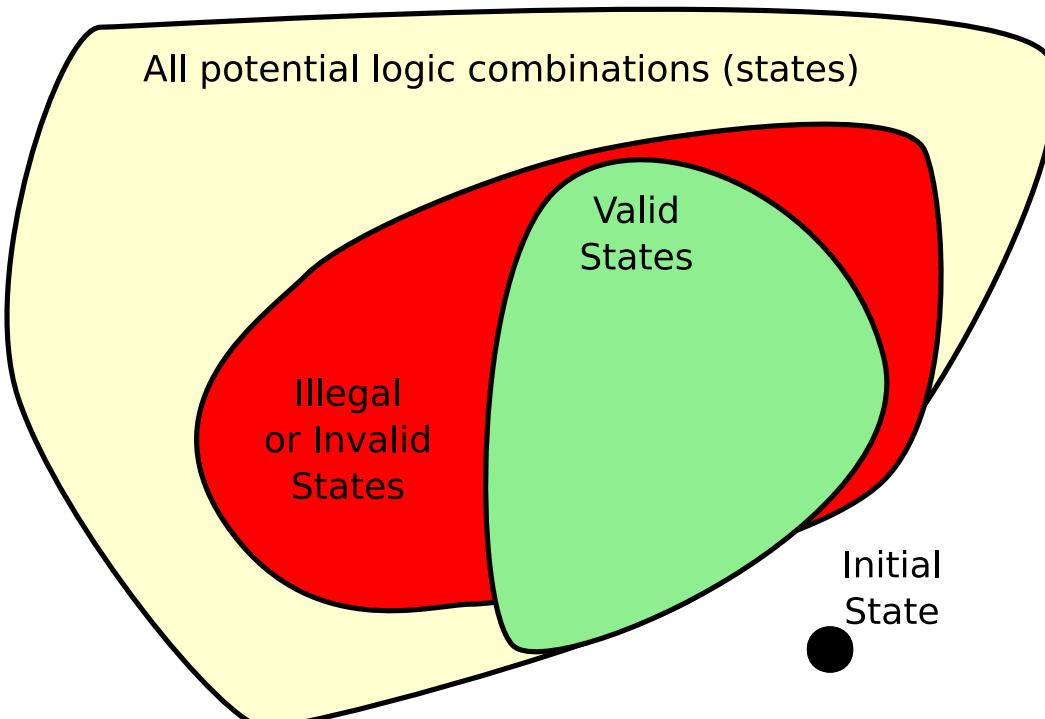
Operator	Formal Verification	Traditional Simulation
restrict ()	Restricts search space	Ignored
assume()		Halts simulation with an error
assert()	Illegal state	

- **restrict ()**: Like **assume(x)**, it also limits the state space
- But in a traditional simulation ...
 - **restrict ()** is ignored
 - **assume()** is turned into an **assert()**

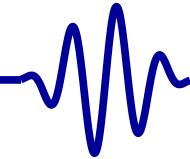
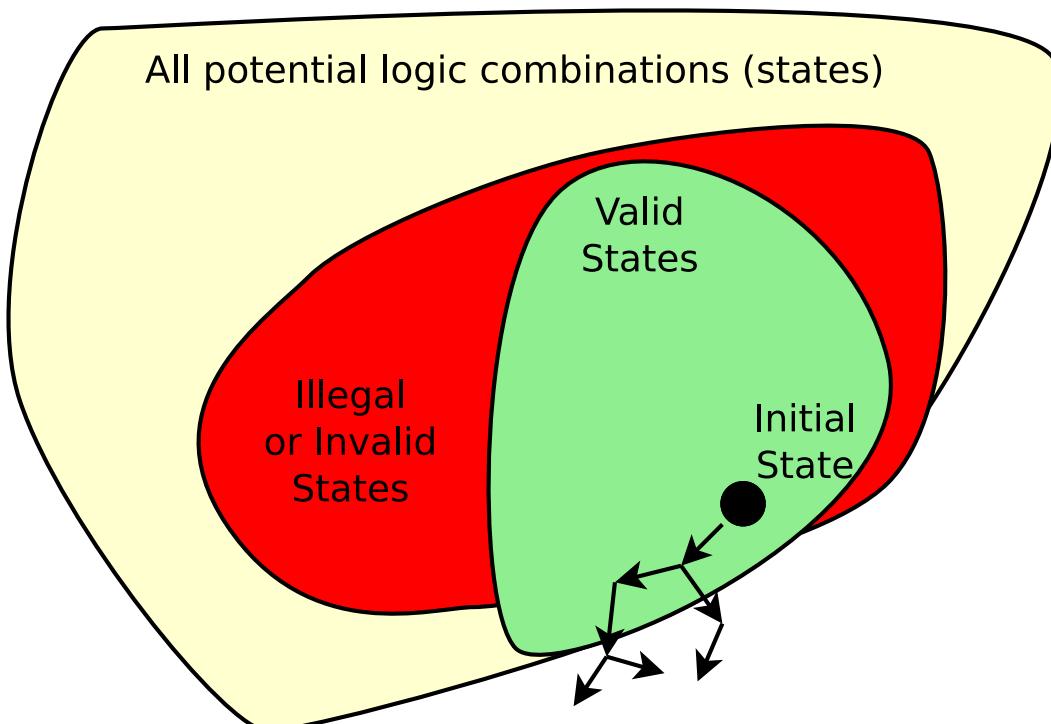
[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[▷ BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

For bounded model checking,

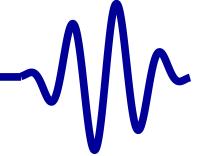
1. Start at the initial state
2. Examine *all* possible states for N clocks
3. Try to find a way to make an **assert**(); fail
4. If it's not possible in N clocks, then *pass*

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[▷ BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Problem: **initial assume(!initial_state);**
Model fails, *no line number given.*

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[▷ BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Problem: **assume(!reachable_state);**
Model fails, *no line number given.*



Welcome

Motivation

Basics

Basics

General Rule

Assert

Assume

▷ BMC

Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

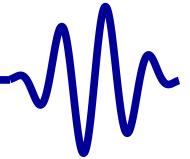
Sequences

Quizzes

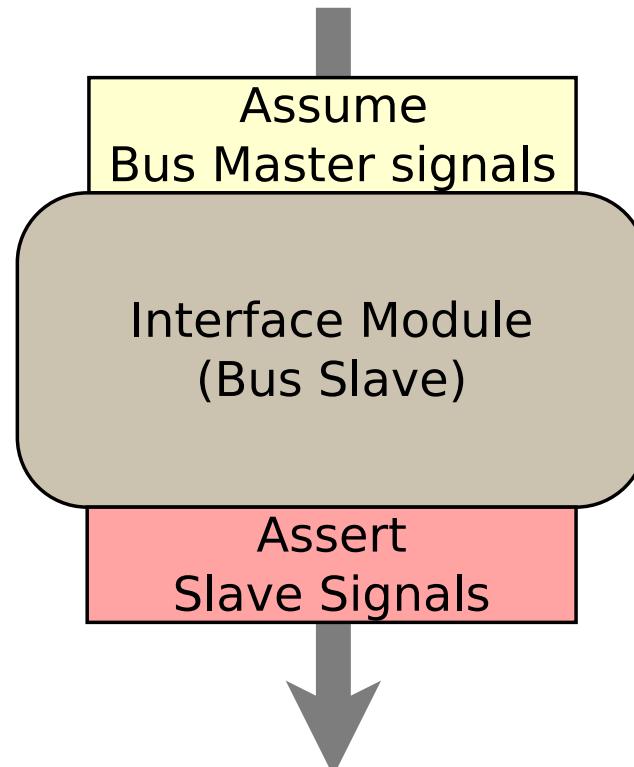
Unlike the rest of your digital design, formal properties . . .

- don't need to meet timing
- don't need to meet a minimum logic requirement

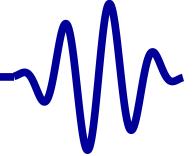
We'll discuss this more as we go along.

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[▷ BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Here's an example of a bus slave



- Inputs are assumed
- Outputs are asserted



Welcome

Motivation

Basics

Basics

General Rule

Assert

Assume

▷ BMC

Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

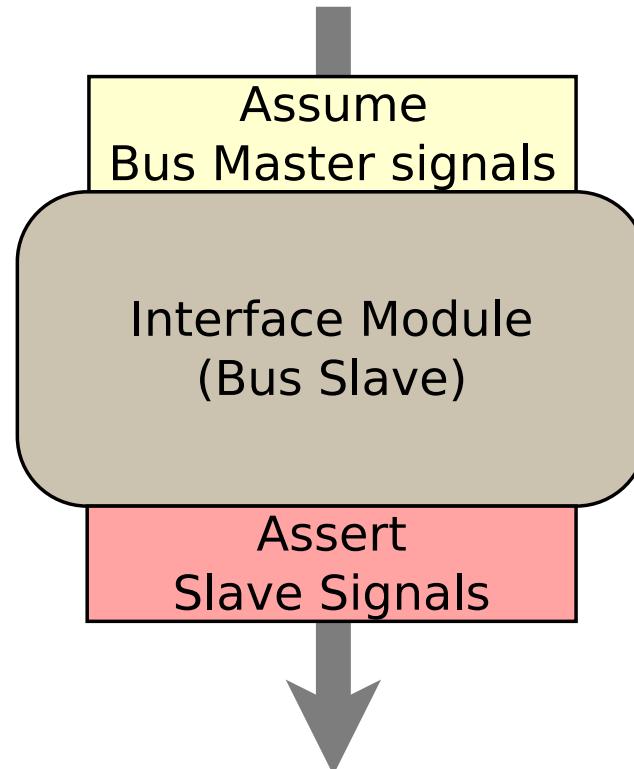
Multiple-Clocks

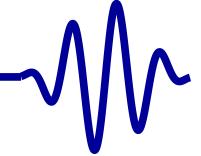
Cover

Sequences

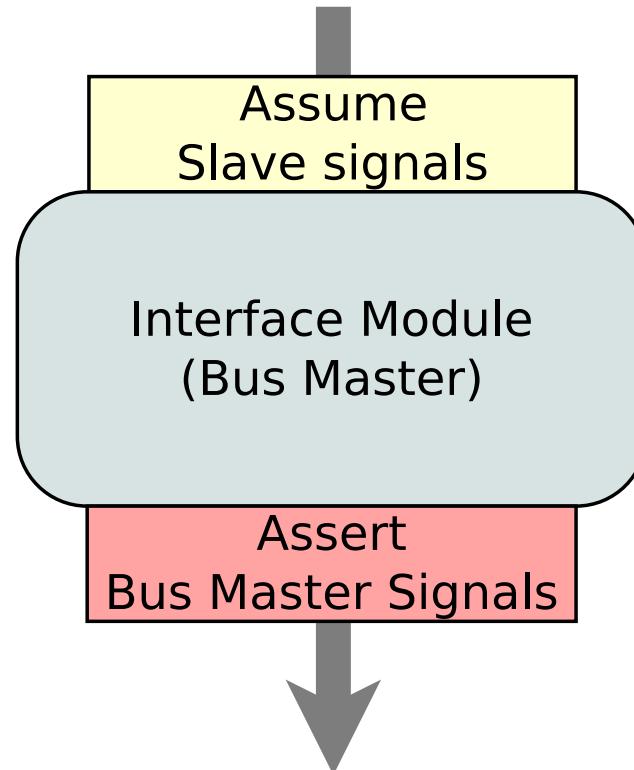
Quizzes

Question: How would a bus master be different?



[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[▷ BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Question: How would a bus master be different?

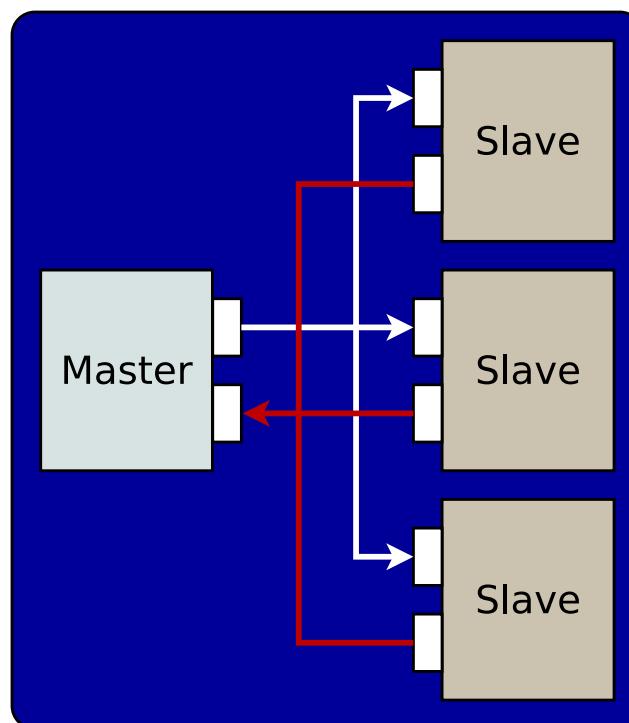


The slave's outputs are the master's inputs

- **assume()** the inputs from the slave
- **assert()** the outputs from the master

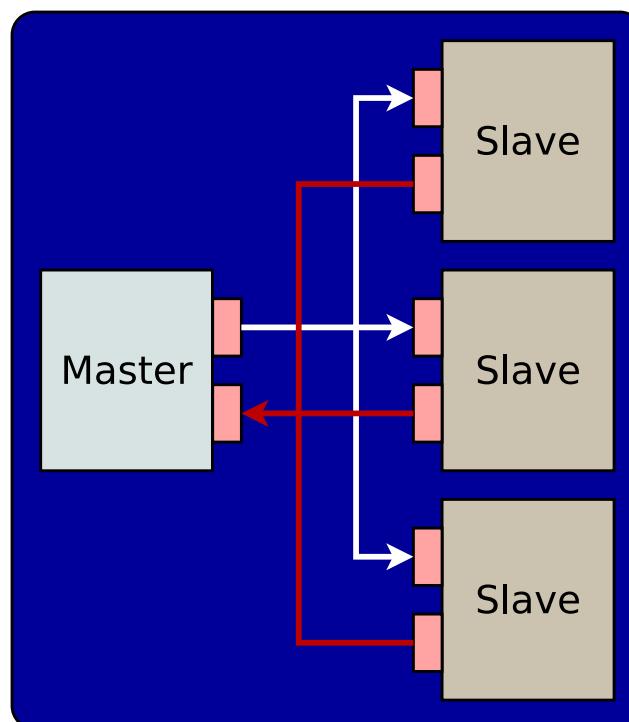
[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[▷ BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Question: What if both slave and master signals were part of the same design?



[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[▷ BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Question: What if both slave and master signals were part of the same design?



- All of the wires are now internal
- They should therefore be **assert()**ed

Serial Port Transmitter



Welcome

Motivation

Basics

Basics

General Rule

Assert

Assume

\triangleright BMC

Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

- Whenever the serial port is idle, the output line should be high

```
if (state == IDLE)
    assert(o_uart_tx);
```

- Whenever the serial port is not idle, busy should be high

```
if (state != IDLE)
    assert(o_busy);
else
    assert(!o_busy);
```

- The design can only ever be in a valid state

```
assert((state <= TXUL_STOP)
    ||(state == TXUL_IDLE));
```

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[▷ BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- Arbiter cannot grant both A and B access

```
always @(*)  
    assert ((!grant_A) || (!grant_B));
```

- While one has access, the other must be stalled

```
always @(*)  
    if (grant_A)  
        assert(stall_B);  
  
always @(*)  
    if (grant_B)  
        assert(stall_A);
```

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[▷ BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- While one is stalled, its outstanding requests must be zero

```
always @(*)  
  if (grant_A)  
    begin  
      assert(f_nreqs_B == 0);  
      assert(f_nacks_B == 0);  
      assert(f_outstanding_B == 0);  
    end
```

I use the prefix f_ to indicate a variable that is

- Not part of the design
- But only used for Formal Verification

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[▷ BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- Avalon bus: will never issue a read and write request at the same time

always @(*)**assume**((!i_av_read)||(!i_av_write));

- The bus is initially idle

initial assume(!i_av_read);**initial assume**(!i_av_write);**initial assume**(!i_av_lock);**initial assert**(!o_av_readdatavalid);**initial assert**(!o_av_writeresponsevalid);

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[▷ BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- Cannot respond to both read and write in the same clock

```
always @(*)  
    assume((!i_av_readdatavalid)  
           ||(!i_av_writeresponsevalid));
```

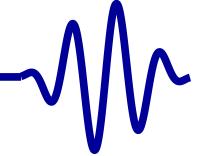
Remember ! (A&&B) is equivalent to (!A)||(! B)

- Cannot respond if no request is outstanding

```
always @(*)  
begin  
    if (f_wr_outstanding == 0)  
        assert(!o_av_writeresponsevalid);  
    if (f_rd_outstanding == 0)  
        assert(!o_av_readdatavalid);  
end
```



Wishbone



Welcome

Motivation

Basics

Basics

General Rule

Assert

Assume

▷ BMC

Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

- o_STB can only be high if o_CYC is also high

```
always @(*)  
  if (o_STB) assert(o_CYC);
```

- Count the number of outstanding requests:

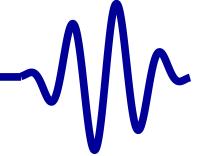
```
assign f_outstanding = (i_reset) ? 0  
      : f_nreqs - f_nacks;
```

- Acks can only respond to valid requests

```
if (f_outstanding == 0)  
  assume (!i_wb_ack);
```



Wishbone



Welcome

Motivation

Basics

Basics

General Rule

Assert

Assume

> BMC

Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

- Well, what if a request is being made now?

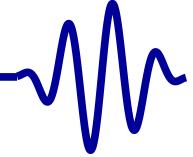
```
if ((f_outstanding == 0)
    &&(!o_wb_stb) || (i_wb_stall))
assume (!i_wb_ack);
```

- If not within a bus request, the ACK and ERR lines must be low

```
if (!o_CYC)
begin
    assume (!i_ACK);
    assume (!i_ERR);
end
```

- Following any reset, the bus will be idle
- Requests remain unchanged until accepted

GT Cache



Welcome

Motivation

Basics

Basics

General Rule

Assert

Assume

▷ BMC

Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Want a guarantee that the cache response is consistent?

- A valid cache entry must ...

```
always @(posedge i_clk)
if (o_valid)
begin
    // Be marked valid in the cache
    assert(cache_valid[f_addr[CW-1:LW]]);
    // Have the same cache tag as address
    assert(f_addr[AW-1:LW] ==
           cache_tag[f_addr[CW-1:LW]]);
    // Match the value in the cache
    assert(o_data ==
           cache_data[f_addr[CW-1:0]]);
    // Must be in response to a valid
    // request
    assert(waiting_requests != 0);
end
```



Multiply



Welcome

Motivation

Basics

Basics

General Rule

Assert

Assume

▷ BMC

Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

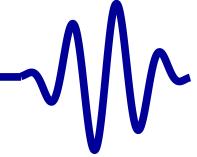
Cover

Sequences

Quizzes

Consider a multiply

- Just because an algorithm doesn't meet timing



Welcome

Motivation

Basics

Basics

General Rule

Assert

Assume

▷ BMC

Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Consider a multiply

- Just because an algorithm doesn't meet timing, or
- Just because it take up logic your FPGA doesn't have

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[▷ BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

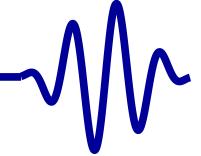
Consider a multiply

- Just because an algorithm doesn't meet timing, or
- Just because it take up logic your FPGA doesn't have, doesn't mean you can't use it now

```
always @ (posedge i_clk)
begin
    f_answer = 0;
    for (k=0; k<NA; k=k+1)
        begin
            if (i_a[k])
                f_answer = f_answer + (i_b<<k);
        end
    assert(o_result == f_answer);
end
```



Multiply



Welcome

Motivation

Basics

Basics

General Rule

Assert

Assume

▷ BMC

Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Let's talk about that multiply some more . . .

- The one thing formal solver's don't handle well is multiplies



Welcome

Motivation

Basics

Basics

General Rule

Assert

Assume

▷ BMC

Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

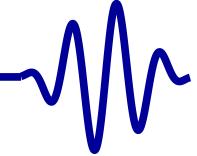
Sequences

Quizzes

Let's talk about that multiply some more . . .

- The one thing formal solver's don't handle well is multiplies

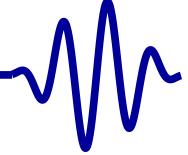
Abstraction offers alternatives

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[▷ BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- For a page result to be valid, it must match the TLB

```
always @(*)
  if (last_page_valid)
    begin
      assert(tlb_valid[f_last_page]);
      assert(last_ppage ==
             tlb_pdata[f_last_page]);
      assert(last_vpage ==
             tlb_vdata[f_last_page]);
      assert(last_ro ==
             tlb_flags[f_last_page][ROFLAG]);
      assert(last_exe ==
             tlb_flags[f_last_page][EXEFLG]);
      assert(r_context_word[LGCTXT-1:1]
             == tlb_cdata[f_last_page]);
    end
```

GT SDRAM



Welcome

Motivation

Basics

Basics

General Rule

Assert

Assume

▷ BMC

Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

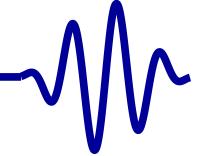
Quizzes

- Writing requires the right row of the right bank to be activated

```
always @(posedge i_clk)
  if ((f_past_valid)&&(!maintenance_mode))
    case(f_cmd)
      // ...
      F_WRITE: begin
        // Response to a write request
        assert(f_we);
        // Bank in question must be active
        assert(bank_active[o_ram_bs] == 3'b111);
        // Active row must be for this address
        assert(bank_row[o_ram_bs]
              == f_addr[22:10]);
        // Must be selecting the right bank
        assert(o_ram_bs == f_addr[9:8]);
      end
    // ...
  
```



Ex: Counter



Welcome

Motivation

Basics

Basics

General Rule

Assert

Assume

BMC

▷ Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

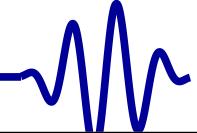
Quizzes

Let's work through a counter as an example.

- | | |
|---------------------------|--|
| <code>exercise-01/</code> | Contains two files |
| <code>counter.v</code> | This will be the source code for our demo. |
| <code>counter.sby</code> | This is the SymbiYosys script for the demo |

Our Objectives:

- Walk through the steps in the tool-flow
- Hands on experience with SymbiYosys
- Ensure everyone has a working version of SymbiYosys
- Find and fix a design bug

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

```
parameter [15:0] MAX_AMOUNT = 22;
reg [15:0] counter;

always @ (posedge i_clk)
    if ((i_start_signal)&&(counter == 0))
        counter <= MAX_AMOUNT - 1'b1;
    else if (counter != 0)
        counter <= counter - 1'b1;

always @ (*)
    o_busy = (counter != 0);

`ifdef FORMAL
    always @ (*)
        assert(counter < MAX_AMOUNT);
`endif
```

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

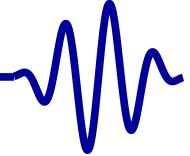
In the file, exercise-01/counter.sby, you'll find:

```
[options]
mode bmc

[engines]
smtbmc

[script]
read -formal counter.v
# ... other files would go here
prep -top counter

[files]
counter.v
```

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

In the file, exercise-01/counter.sby, you'll find:

[**options**]

mode bmc ← Bounded model checking mode

[**engines**]

smtbmc

[**script**]

read -formal counter.v

... other files would go here

prep -top counter

[**files**]

counter.v

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

In the file, exercise-01/counter.sby, you'll find:

[**options**]

mode bmc

[**engines**]

smtbmc ← Run, using yosys-smtbmc

[**script**]

read -formal counter.v

... other files would go here

prep -top counter

[**files**]

counter.v

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

In the file, exercise-01/counter.sby, you'll find:

[**options**]

mode bmc

[**engines**]

smtbmc

[**script**] ← Yosys commands

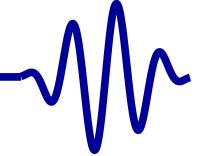
read -formal counter.v

... other files would go here

prep -top counter

[**files**]

counter.v

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

In the file, exercise-01/counter.sby, you'll find:

[**options**]

mode bmc

[**engines**]

smtbmc

[**script**]

read -formal counter.v ← Read file

... other files would go here

prep -top counter

[**files**]

counter.v

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

In the file, exercise-01/counter.sby, you'll find:

```
[ options ]
```

```
mode bmc
```

```
[ engines ]
```

```
smtbmc
```

```
[ script ]
```

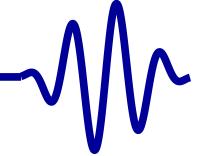
```
read -formal counter.v
```

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

```
prep -top counter ← Prepare the file for formal
```

```
[ files ]
```

```
counter.v
```

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

In the file, exercise-01/counter.sby, you'll find:

[**options**]

mode bmc

[**engines**]

smtbmc

[**script**]

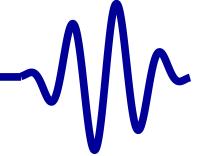
read -formal counter.v

... other files would go here

prep -top counter

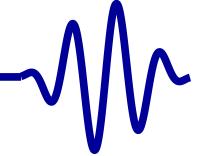
[**files**] ← List of files to be used

counter.v

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Other usefull yosys commands

```
[options]
mode bmc
depth 20
[engines]
smtbmc yices
# smtbmc boolector
# smtbmc z3
[script]
read -formal counter.v
# ... other files would go here
prep -top counter
opt_merge -share_all
[files]
counter.v
```

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Other usefull yosys commands

[options]

```
mode bmc ← Other modes: prove, cover, live
```

```
depth 20
```

[engines]

```
smtbmc yices
```

```
# smtbmc boolector
```

```
# smtbmc z3
```

[script]

```
read -formal counter.v
```

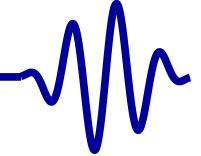
```
# ... other files would go here
```

```
prep -top counter
```

```
opt_merge -share_all
```

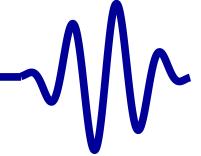
[files]

```
counter.v
```

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

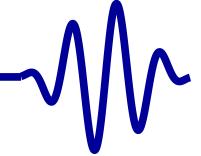
Other usefull yosys commands

```
[options]
mode bmc
depth 20 ← # of Steps to examine
[engines]
smtbmc yices
# smtbmc boolector
# smtbmc z3
[script]
read -formal counter.v
# ... other files would go here
prep -top counter
opt_merge -share_all
[files]
counter.v
```

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

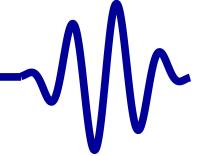
Other usefull yosys commands

```
[options]
mode bmc
depth 20
[engines]
smtbmc yices ← Yices theorem prover (default)
# smtbmc boolector
# smtbmc z3
[script]
read -formal counter.v
# ... other files would go here
prep -top counter
opt_merge -share_all
[files]
counter.v
```

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

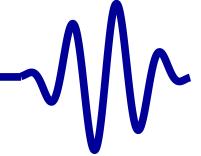
Other usefull yosys commands

```
[options]
mode bmc
depth 20
[engines]
smtbmc yices
# smtbmc boolector ← Other potential solvers
# smtbmc z3
[script]
read -formal counter.v
# ... other files would go here
prep -top counter
opt_merge -share_all
[files]
counter.v
```

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Other usefull yosys commands

```
[options]
mode bmc
depth 20
[engines]
smtbmc yices
# smtbmc boolector
# smtbmc z3
[script]
read -formal counter.v
# ... other files would go here
prep -top counter
opt_merge -share_all ← We'll discuss this later
[files]
counter.v
```

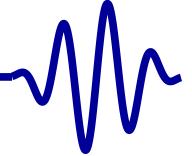
[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Other usefull yosys commands

```
[options]
mode bmc
depth 20
[engines]
smtbmc yices
# smtbmc boolector
# smtbmc z3
[script]
read -formal counter.v
# ... other files would go here
prep -top counter
opt_merge -share_all
[files]
counter.v ← Full or relative pathnames go here
```



Running SymbiYosys



Welcome

Motivation

Basics

Basics

General Rule

Assert

Assume

BMC

▷ Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

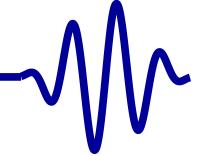
Sequences

Quizzes

Run: % sby -f counter.sby



Running SymbiYosys



Welcome

Motivation

Basics

Basics

General Rule

Assert

Assume

BMC

▷ Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

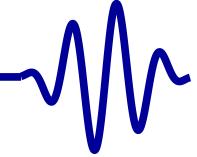
Sequences

Quizzes

Run: % sbt -f counter.sbt

```
dan@jericho:~/work/rnd/opencores/formal/ysfvclass/demo-bench$ sbt -f counter.sbt
SBY [counter] Removing direcory 'counter'.
SBY [counter] Copy '../demo-rtl/counter.v' to 'counter/src/counter.v'.
SBY [counter] engine_0: smtbmc
SBY [counter] script: starting process "cd counter/src; yosys -ql ../model/design.log ../model/design.yo"
SBY [counter] script: finished (returncode=0)
SBY [counter] smt2: starting process "cd counter/model; yosys -ql design_smt2.log design_smt2.ys"
SBY [counter] smt2: finished (returncode=0)
SBY [counter] engine_0: starting process "cd counter; yosys-smtbmc --noprogress --presat --unroll -t 20 --append 0 --dump-vcd engine_0/trace.vcd --dump-vlogtb engine_0/trace_tb.v --dump-smtc engine_0/trace.smtc model/design_smt2.smt2"
SBY [counter] engine_0: ##      0  0:00:00  Solver: yices
SBY [counter] engine_0: ##      0  0:00:00  Checking assumptions in step 0..
SBY [counter] engine_0: ##      0  0:00:00  Checking assertions in step 0..
SBY [counter] engine_0: ##      0  0:00:00  BMC failed!
SBY [counter] engine_0: ##      0  0:00:00  Assert failed in counter: counter.v:63
SBY [counter] engine_0: ##      0  0:00:00  Writing trace to VCD file: engine_0/trace.vcd
SBY [counter] engine_0: ##      0  0:00:00  Writing trace to Verilog testbench: engine_0/trace_tb.v
SBY [counter] engine_0: ##      0  0:00:00  Writing trace to constraints file: engine_0/trace.smvc
SBY [counter] engine_0: ##      0  0:00:00  Status: FAILED (!)
SBY [counter] engine_0: finished (returncode=1)
SBY [counter] engine_0: Status returned by engine: FAIL
SBY [counter] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)
SBY [counter] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00 (0)
SBY [counter] summary: engine_0 (smtbmc) returned FAIL
SBY [counter] summary: counterexample trace: counter/engine_0/trace.vcd
SBY [counter] DONE (FAIL, rc=2)
dan@jericho:~/work/rnd/opencores/formal/ysfvclass/demo-bench$
```

BMC Failed



Welcome

Motivation

Basics

Basics

General Rule

Assert

Assume

BMC

▷ Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

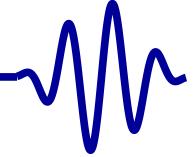
Sequences

Quizzes

Run: % sbt -f counter.sbt

```
dan@jericho:~/work/rnd/opencores/formal/ysfvclass/demo-bench$ sbt -f counter.sbt
SBY [counter] Removing directory 'counter'.
SBY [counter] Copy '../demo-rtl/counter.v' to 'counter/src/counter.v'.
SBY [counter] engine_0: smtbmc
SBY [counter] script: starting process "cd counter/src; yosys -ql ../model/design.log ../model/design.y"
SBY [counter] script: finished (returncode=0)
SBY [counter] smt2: starting process "cd counter/model; yosys -ql design_smt2.log design_smt2.ys"
SBY [counter] smt2: finished (returncode=0)
SBY [counter] engine_0: starting process "cd counter; yosys-smtbmc --noprogress --presat --unroll -t 20 --append 0 --dump-vcd engine_0/trace.vcd --dump-vlogtb engine_0/trace_tb.v --dump-smtc engine_0/trace.smvc model/design_smt2.smt2"
SBY [counter] engine_0: ##      0  0:00:00  Solver: yices
SBY [counter] engine_0: ##      0  0:00:00  Checking assumptions in step 0...
SBY [counter] engine_0: ##      0  0:00:00  Checking assertions in step 0...
SBY [counter] engine_0: ##      0  0:00:00  BMC failed!
SBY [counter] engine_0: ##      0  0:00:00  Assert failed in counter: counter.v:63
SBY [counter] engine_0: ##      0  0:00:00  Writing trace to VCD file: engine_0/trace.vcd
SBY [counter] engine_0: ##      0  0:00:00  Writing trace to Verilog testbench: engine_0/trace_tb.v
SBY [counter] engine_0: ##      0  0:00:00  Writing trace to constraints file: engine_0/trace.smvc
SBY [counter] engine_0: ##      0  0:00:00  Status: FAILED (!)
SBY [counter] engine_0: finished (returncode=1)
SBY [counter] engine_0: Status returned by engine: FAIL
SBY [counter] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)
SBY [counter] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00 (0)
SBY [counter] summary: engine_0 (smtbmc) returned FAIL
SBY [counter] summary: counter example trace: counter/engine_0/trace.vcd
SBY [counter] DONE (FAIL, rc=2)
dan@jericho:~/work/rnd/opencores/formal/ysfvclass/demo-bench$
```

Where Next



Welcome

Motivation

Basics

Basics

General Rule

Assert

Assume

BMC

▷ Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

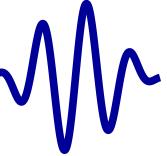
Cover

Sequences

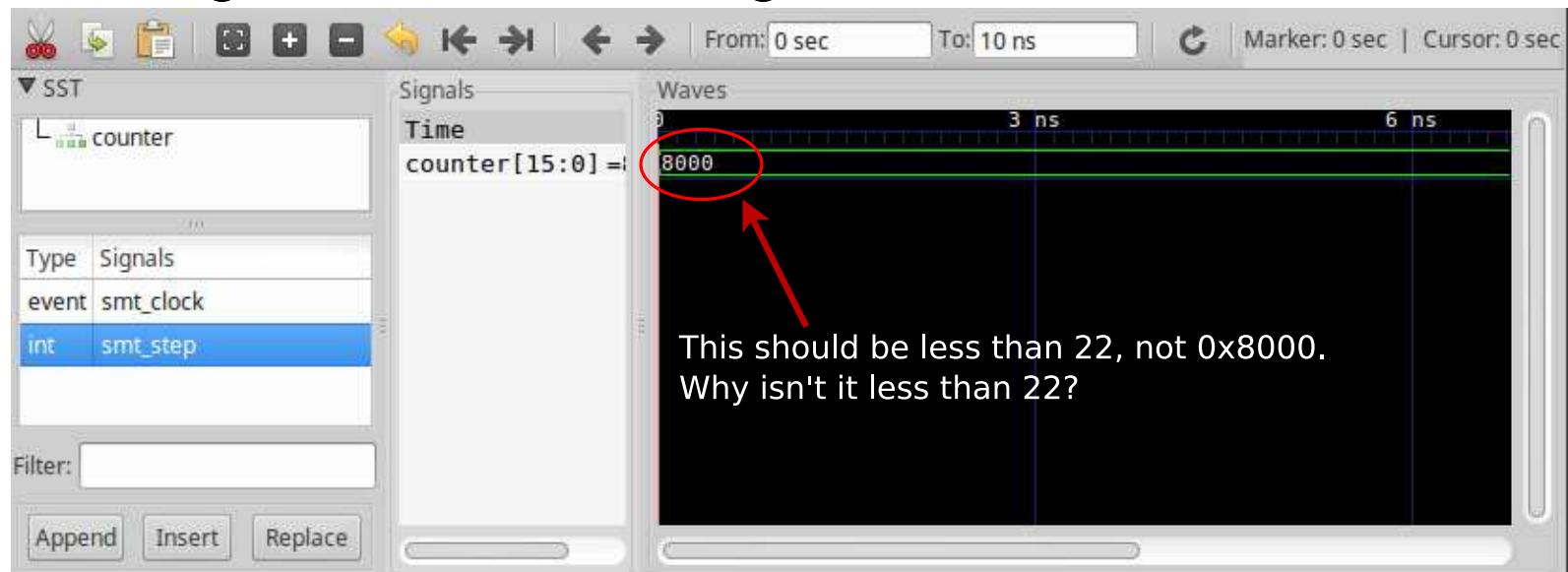
Quizzes

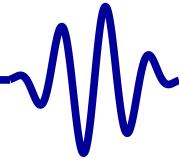
Look at source line 63, and fire up gtkwave

```
dan@jericho:~/work/rnd/opencores/formal/ysfvclass/demo-bench$ sby -f counter.sby
SBY [counter] Removing direcory 'counter'.
SBY [counter] Copy '../demo-rtl/counter.v' to 'counter/src/counter.v'.
SBY [counter] engine_0: smtbmc
SBY [counter] script: starting process "cd counter/src; yosys -ql ../model/design.log ../model/design.y"
SBY [counter] script: finished (returncode=0)
SBY [counter] smt2: starting process "cd counter/model; yosys -ql design_smt2.log design_smt2.ys"
SBY [counter] smt2: finished (returncode=0)
SBY [counter] engine_0: starting process "cd counter; yosys-smtbmc --noprocess --presat --unroll -t 20 --append 0 --dump-vcd engine_0/trace.vcd --dump-vlogtb engine_0/trace_tb.v --dump-smtc engine_0/trace.smvc model/design_smt2.smt2"
SBY [counter] engine_0: ##      0  0:00:00  Solver: yices
SBY [counter] engine_0: ##      0  0:00:00  Checking assumptions in step 0..
SBY [counter] engine_0: ##      0  0:00:00  Checking assertions in step 0..
SBY [counter] engine_0: ##      0  0:00:00  BMC failed!
SBY [counter] engine_0: ##      0  0:00:00  Assert failed in counter: counter.v:63
SBY [counter] engine_0: ##      0  0:00:00  Writing trace to VCD file: engine_0/trace.vcd
SBY [counter] engine_0: ##      0  0:00:00  Writing trace to Verilog testbench: engine_0/trace_tb.v
SBY [counter] engine_0: ##      0  0:00:00  Writing trace to constraints file: engine_0/trace.smvc
SBY [counter] engine_0: ##      0  0:00:00  Status: FAILED (!)
SBY [counter] engine_0: finished (returncode=1)
SBY [counter] engine_0: Status returned by engine: FAIL
SBY [counter] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)
SBY [counter] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00 (0)
SBY [counter] summary: engine_0 (smtbmc) returned FAIL
SBY [counter] summary: counterexample trace: counter/engine_0/trace.vcd
SBY [counter] DONE (FAIL, rc=2)
dan@jericho:~/work/rnd/opencores/formal/ysfvclass/demo-bench$
```

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Run: % gtkwave counter/engine_0/trace.vcd



[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

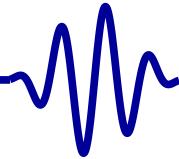
Run: % demo-rtl/counter.v

What did we do wrong?

```
File Edit Tools Syntax Buffers Window Help
39 // 
40 `default_nettype none
41 //
42 module counter(i_clk, i_start_signal, o_busy);
43     parameter [15:0] MAX_AMOUNT = 22;
44     //
45     input wire i_clk;
46     //
47     input wire i_start_signal;
48     output reg o_busy;
49
50     reg [15:0] counter;
51
52     always @(posedge i_clk)
53         if ((i_start_signal)&&(counter == 0))
54             counter <= MAX_AMOUNT-1'b1;
55         else if (counter != 0)
56             counter <= counter - 1'b1;
57
58     always @(*)
59         o_busy <= (counter != 0);
60
61 `ifdef FORMAL
62     always @(*)
63         assert(counter < MAX_AMOUNT);
64 `endif
65 endmodule
```

Line 63, Here's the assertion that failed

53,37-51 Bot

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Run: % demo-rtl/counter.v

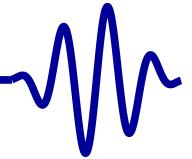
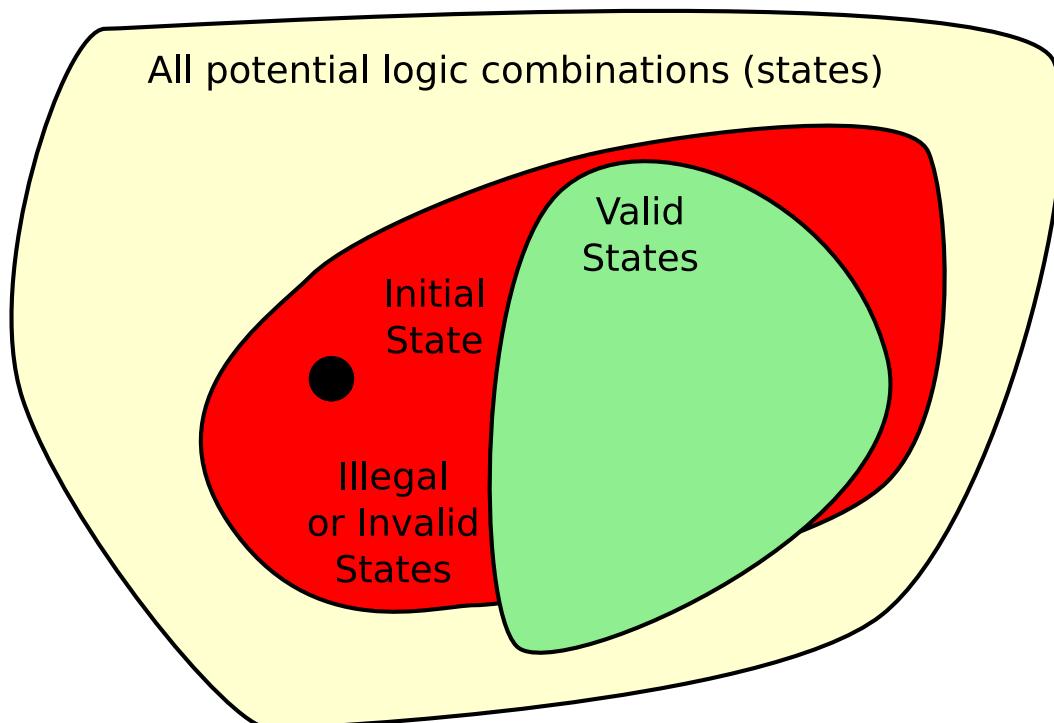
What did we do wrong?

```
File Edit Tools Syntax Buffers Window Help
39 // 
40 `default_nettype none
41 //
42 module counter(i_clk, i_start_signal, o_busy);
43     parameter [15:0] MAX_AMOUNT = 22;
44     //
45     input wire i_clk;
46     //
47     input wire i_start_signal;
48     output reg o_busy;
49
50     reg [15:0] counter;
51
52     always @(posedge i_clk)
53         if ((i_start_signal)&&(counter == 0))
54             counter <= MAX_AMOUNT-1'b1;
55         else if (counter != 0)
56             counter <= counter - 1'b1;
57
58     always @(*)
59         o_busy <= (counter != 0);
60
61 `ifdef FORMAL
62     always @(*)
63         assert(counter < MAX_AMOUNT);
64 `endif
65 endmodule
```

Line 63, Here's the assertion that failed

53,37-51 Bot

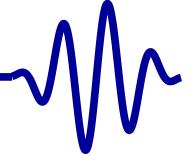
Did you notice the missing initial statement?

[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[Ex: Counter](#) [\$\triangleright\$ Sol'n](#)[Clocked and \\$past](#) [\$k\$ Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- Problem: No initial statement
- Solver finds an invalid initial state
- Model fails



Exercise



Welcome

Motivation

Basics

Basics

General Rule

Assert

Assume

BMC

Ex: Counter

▷ Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

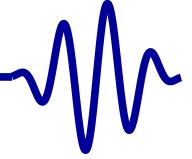
Multiple-Clocks

Cover

Sequences

Quizzes

Try adding in the initial statement, will it work?



Welcome

Motivation

Basics

Clocked and

▷ \$past

Past

\$past Rule

Past Assertions

Past Valid

Examples

Ex: Busy Counter

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

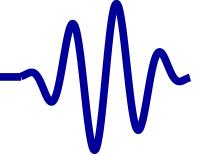
Sequences

Quizzes

Clocked and \$past



Lesson Overview



Welcome

Motivation

Basics

Clocked and \$past

▷ Past

\$past Rule

Past Assertions

Past Valid

Examples

Ex: Busy Counter

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

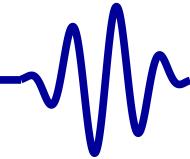
Cover

Sequences

Quizzes

Our Objective:

- To learn how to make assertions crossing time intervals
 - **\$past()**
- Before the beginning of time
 - Assumptions always hold
 - Assertions rarely hold
- How to get around this with f_past_valid

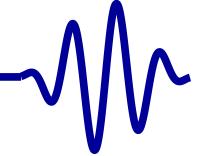
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[▷ Past](#)[\\$past Rule](#)[Past Assertions](#)[Past Valid](#)[Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- **\$past(X)** Returns the value of X one clock ago.
- **\$past(X,N)** Returns the value of X N clocks ago.
- Depends upon a clock
 - This is illegal

```
always @(*)  
if (x)  
    assert(y == $past(y));
```

- No clock is associated with the **\$past** operator.
- But you can do this

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

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[▷ \\$past Rule](#)[Past Assertions](#)[Past Valid](#)[Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

\$past FV Rule

Only use \$past as a precondition

```
always @(posedge clk)
  if ((f_past_valid)&&($past(value)))
    assert(something);
```

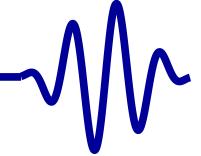
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[▷ Past Assertions](#)[Past Valid](#)[Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Let's modify our counter, by creating some additional properties:

```
always @(*)  
    assume (! i_start_signal );  
  
always @(posedge clk)  
    assert ($past(counter == 0));
```

- `i_start_signal` is now never true, so the counter should always be zero.
- `assert(counter == 0);`
This should always be true, since counter starts at zero, and is never changed from zero.
- Will `assert($past(counter == 0));` succeed?

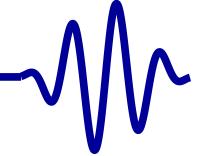
You can find this file in `exercise-02/pastassert.v`

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[▷ Past Assertions](#)[Past Valid](#)[Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- This fails

```
always @(*)  
    assume (!i_start_signal);
```

```
always @(*)  
    assert ($past(counter == 0));
```

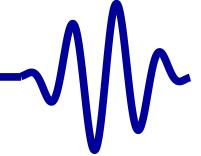
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[▷ Past Assertions](#)[Past Valid](#)[Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- This succeeds

```
always @(*)  
    assume (!i_start_signal);
```

```
always @(*)  
    assert (counter == 0);
```

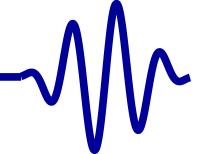
- Before time, counter is unconstrained.
- The solver can make it take on any value it wants in order to make things fail
- This will not show in the VCD file

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[▷ Past Assertions](#)[Past Valid](#)[Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Let's try again:

```
always @(posedge clk)
if ($past(i_start_signal))
    assert(counter == MAX_AMOUNT-1'b1);
```

This should work, right?

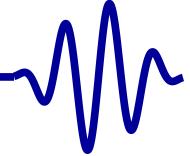
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[▷ Past Assertions](#)[Past Valid](#)[Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Let's try again:

```
always @(posedge clk)
if ($past(i_start_signal))
    assert(counter == MAX_AMOUNT - 1'b1);
```

This should work, right? No, it fails.

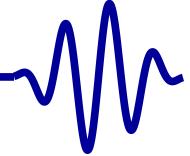
- `i_start_signal` is unconstrained before time
- `counter` is initially constrained to zero
- If `i_start_signal` is one before time,
`counter` will still be zero when time begins

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[Past Assertions](#)[▷ Past Valid](#)[Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

We can fix this with a register I call, f_past_valid:

```
reg f_past_valid;  
  
initial f_past_valid = 1'b0;  
always @(posedge clk)  
    f_past_valid <= 1'b1;  
  
always @(posedge clk)  
if ((f_past_valid)&&($past(i_start_signal)))  
    assert(counter == MAX_AMOUNT-1'b1);
```

Will this work?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[Past Assertions](#)[▷ Past Valid](#)[Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

We can fix this with a register I call, f_past_valid:

```
reg f_past_valid;  
  
initial f_past_valid = 1'b0;  
always @(posedge clk)  
    f_past_valid <= 1'b1;  
  
always @(posedge clk)  
if ((f_past_valid)&&($past(i_start_signal)))  
    assert(counter == MAX_AMOUNT-1'b1);
```

Will this work? Almost, but not yet.

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[Past Assertions](#)[▷ Past Valid](#)[Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- What about the case where `i_start_signal` is raised while the counter isn't zero?

```
reg f_past_valid;  
  
initial f_past_valid = 1'b0;  
always @ (posedge clk)  
    f_past_valid <= 1'b1;  
  
always @ (posedge clk)  
if ((f_past_valid)&&($past(i_start_signal))  
    &&($past(counter == 0)))  
    assert(counter == MAX_AMOUNT - 1'b1);
```

- Will this work?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[Past Assertions](#)[▷ Past Valid](#)[Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- What about the case where `i_start_signal` is raised while the counter isn't zero?

```
reg f_past_valid;

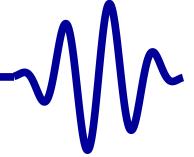
initial f_past_valid = 1'b0;
always @ (posedge clk)
    f_past_valid <= 1'b1;

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

- Will this work? Yes, now it will work
- You'll find lots of references to `f_past_valid` in my own code.



Examples



Welcome

Motivation

Basics

Clocked and \$past

Past

\$past Rule

Past Assertions

Past Valid

▷ Examples

Ex: Busy Counter

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

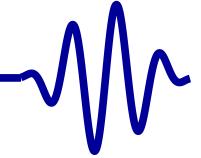
Multiple-Clocks

Cover

Sequences

Quizzes

Let's look at some practical examples

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[Past Assertions](#)[Past Valid](#)[▷ Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

The rule: Every design should start in the reset state.

```
initial assume(i_RESET);
```

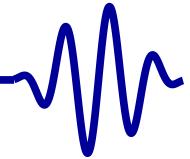
```
always @(*)
  if (!f_past_valid)
    assume(i_RESET);
```

What would be the difference between these two properties?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[Past Assertions](#)[Past Valid](#)[▷ Examples](#)[Ex: Busy Counter](#)[*k* Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

The rule: On the clock following a reset, there should be no outstanding bus requests.

```
always @(posedge clk)
if ((f_past_valid)&&($past(i_RESET)))
    assert (!o_CYC);
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[Past Assertions](#)[Past Valid](#)[▷ Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Two times registers must have their reset value

- Initially
- Following a reset

```
always @(posedge clk)
if ((!f_past_valid)||($past(i_reset)))
begin
    assert (!o_CYC);
    assert (!o_STB);
    // etc.
end
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[Past Assertions](#)[Past Valid](#)[▷ Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

The rule: while a request is being made, the request cannot change until it is accepted.

```
always @(posedge clk)
  if ((f_past_valid)
      &&($past(o_STB))&&($past(i_STALL)))
    begin
      assert(o_STB);
      assert(o_REQ == $past(o_REQ));
    end
```

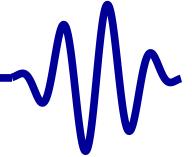
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[Past Assertions](#)[Past Valid](#)[Examples](#)[Ex: Busy](#)[▷ Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Many of my projects include some type of “busy counter”

- Serial port logic must wait for a baud clock
Transmit characters must wait for the port to be idle
- I2C logic needs to slow the clock down
- SPI logic may also need to slow the clock down

Objectives:

- Gain some confidence using formal methods to prove that alternative designs are equivalent

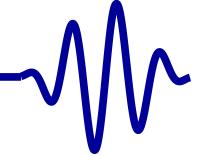
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[Past Assertions](#)[Past Valid](#)[Examples](#)

Ex: Busy
▷ Counter

[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Here's the basic code. It should look familiar.

```
parameter [15:0] MAX_AMOUNT = 22;  
  
reg [15:0] counter;  
  
initial counter = 0;  
always @(posedge i_clk)  
if (i_reset)  
    counter <= 0;  
else if ((i_start_signal)&&(counter == 0))  
    counter <= MAX_AMOUNT-1'b1;  
else if (counter != 0)  
    counter <= counter - 1;  
  
always @(*)  
o_busy = (counter != 0);
```



Welcome

Motivation

Basics

Clocked and \$past

Past

\$past Rule

Past Assertions

Past Valid

Examples

Ex: Busy

▷ Counter

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

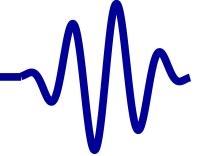
Sequences

Quizzes

You can find the code in `exercise-03/busyctr.v`.

Exercise: Create the following properties:

1. `i_start_signal` may be raised at any time
No property needed here
2. Once raised, *assume* `i_start_signal` will remain high until it is high and the counter is no longer busy.
3. `o_busy` will always be true while the counter is non-zero
This is an assertion
4. If the counter is non-zero, it should always be counting down

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[Past Assertions](#)[Past Valid](#)[Examples](#)[Ex: Busy](#)[▷ Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

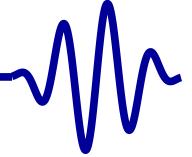
Exercise:

1. Make o_busy a clocked register

```
always @(posedge i_clk)
    o_busy <= /* your logic goes here */;
```

2. Prove that o_busy is true if and only if the counter is non-zero

- You can use this approach to adjust your design to meet timing
 - Shuffle logic from one clock to another, then
 - Prove the new design remains valid



Welcome

Motivation

Basics

Clocked and \$past

$\triangleright k$ Induction

Lesson Overview

vs BMC

General Rule

The Trap

Examples

Bus Properties

Free Variables

Abstraction

Invariants

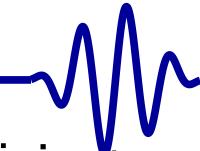
Multiple-Clocks

Cover

Sequences

Quizzes

k Induction

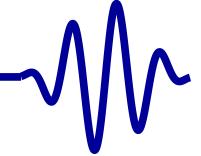
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[▷ Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

If you want to formally verify your design, BMC is insufficient

- Bounded Model Checking (BMC) will only prove that your design is correct for the first N clocks.
- It cannot prove that the design won't fail on the next clock, clock $N + 1$
- This is the purpose of the *induction* step: proving correctness for all time

Our Goals

- Be able to explain what induction is
- Be able to explain why induction is valuable
- Know how to run induction
- What are the unique problems associated with induction

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[▷ Lesson Overview
vs BMC](#)[General Rule](#)[The Trap](#)[Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Proof by induction has two steps:

1. **Base case:** Prove for $N = 0$ (or one)
2. **Inductive step:** Assume true for N , prove true for $N + 1$.

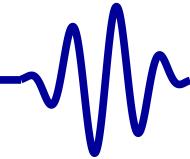
Example: Prove $\sum_{n=0}^{N-1} x^n = \frac{1-x^N}{1-x}$

- For $N = 1$, the sum is x^0 or one

$$\sum_{n=0}^{N-1} x^n = x^0 = \frac{1-x}{1-x}$$

So this is true (for $x \neq 1$).

- For the inductive step, we'll
 - Assume true for N , then prove for $N + 1$



Welcome

Motivation

Basics

Clocked and \$past

k Induction

▷ Lesson Overview

vs BMC

General Rule

The Trap

Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Prove $\sum_{n=0}^{N-1} x^n = \frac{1-x^N}{1-x}$ for all N

- Assume true for N , prove for $N + 1$

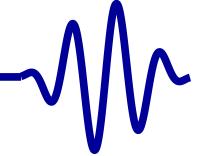
$$\sum_{n=0}^N x^n = x^N + \sum_{n=0}^{N-1} x^n = x^N + \frac{1-x^N}{1-x}$$

- Prove for $N + 1$

$$\begin{aligned} \sum_{n=0}^N x^n &= \frac{1-x}{1-x} x^N + \frac{1-x^N}{1-x} \\ &= \frac{x^N - x^{N+1} + 1 - x^N}{1-x} = \frac{1 - x^{N+1}}{1-x} \end{aligned}$$

This proves the inductive case.

- Hence this is true for all N (where $N > 0$ and $x \neq 1$)

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[▷ Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

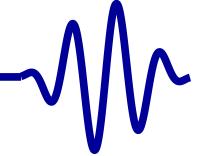
Suppose $\forall n : P[n]$ is what we wish to prove

- Traditional induction

- Base case: show $P[0]$
 - Inductive case: show $P[n] \rightarrow P[n + 1]$

- k induction

- Base case: show $\bigwedge_{k=0}^{N-1} P[k]$
 - k -induction step: $\left(\bigwedge_{k=n-N+1}^n P[k] \right) \rightarrow P[n + 1]$

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[▷ Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Suppose $\forall n : P[n]$ is what we wish to prove

- Traditional induction

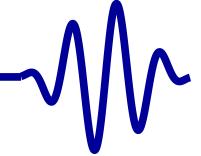
- Base case: show $P[0]$
 - Inductive case: show $P[n] \rightarrow P[n + 1]$

- k induction

- Base case: show $\bigwedge_{k=0}^{N-1} P[k]$

This is what we did with BMC

- k -induction step: $\left(\bigwedge_{k=n-N+1}^n P[k] \right) \rightarrow P[n + 1]$

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[▷ Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Suppose $\forall n : P[n]$ is what we wish to prove

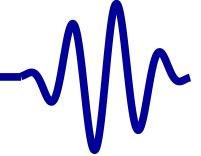
- Traditional induction

- Base case: show $P[0]$
 - Inductive case: show $P[n] \rightarrow P[n + 1]$

- k induction

- Base case: show $\bigwedge_{k=0}^{N-1} P[k]$
 - k -induction step: $\left(\bigwedge_{k=n-N+1}^n P[k] \right) \rightarrow P[n + 1]$

This is our next step

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[▷ Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Suppose $\forall n : P[n]$ is what we wish to prove

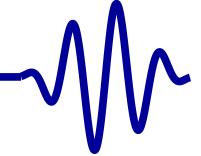
- Traditional induction

- Base case: show $P[0]$
 - Inductive case: show $P[n] \rightarrow P[n + 1]$

- k induction

- Base case: show $\bigwedge_{k=0}^{N-1} P[k]$
 - k -induction step: $\left(\bigwedge_{k=n-N+1}^n P[k] \right) \rightarrow P[n + 1]$

Why use k induction?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[▷ Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Formal verification uses k induction

- **Base case:**

Assume the first N steps do not violate any assumptions, . . .

Prove that the first N steps do not violate any assertions.

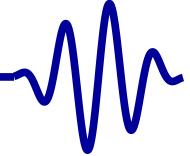
This is the BMC pass we've already done.

- **Inductive Step:**

Assume N steps exist that neither violate any assumptions nor any assertions, and

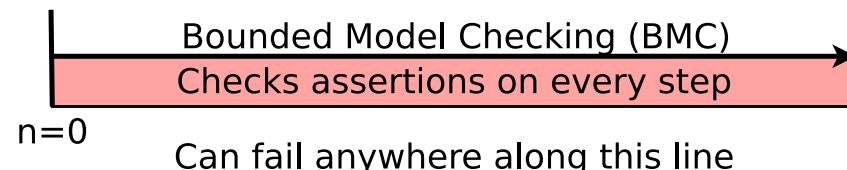
Assume the $N + 1$ step violates no assumptions, . . .

Prove that the $N + 1$ step does not violate any assertions.

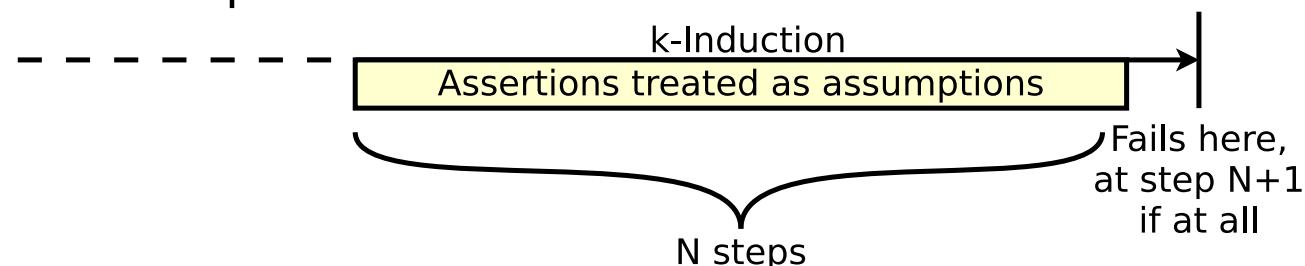
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[▷ vs BMC](#)[General Rule](#)[The Trap](#)[Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

BMC and induction are very different.

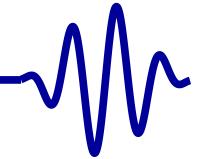
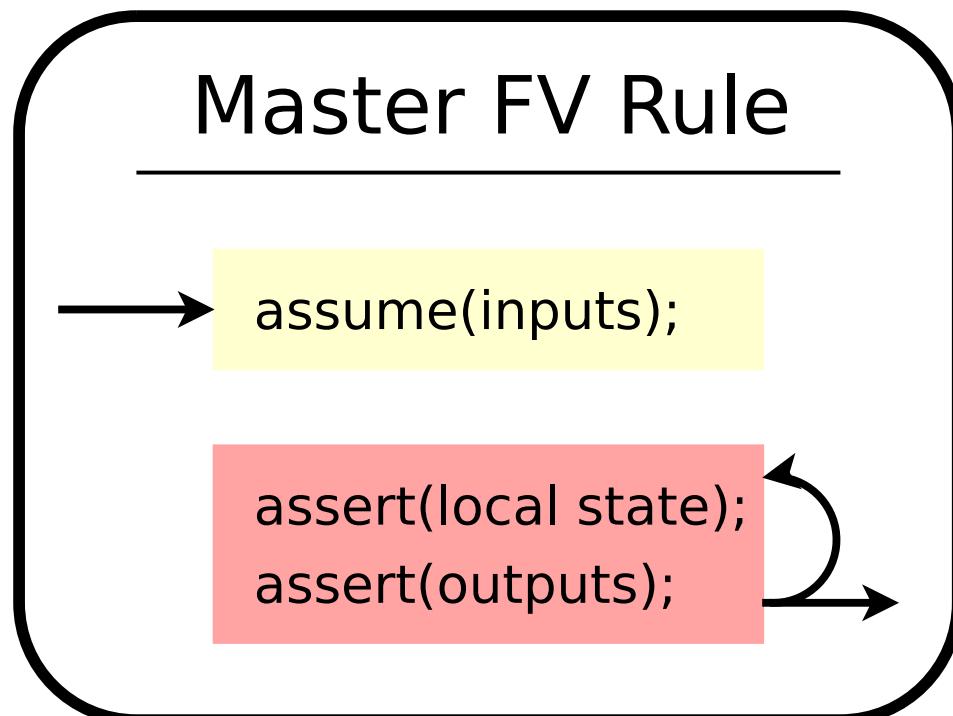
- BMC, the base case



- Induction step



- The number of BMC time-steps must be more than the number of inductive time-steps
- Register values at the beginning of the inductive step can be *anything* allowed by your assertions and assumptions
- This is where the work takes place.

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[▷ General Rule](#)[The Trap](#)[Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

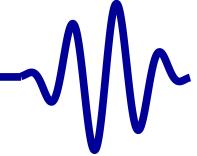
The general rule hasn't changed:

- assume inputs,
- assert internal states and any outputs.

If you assume too much, your design will pass formal verification and still not work.



Checkers



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Lesson Overview

vs BMC

▷ General Rule

The Trap

Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

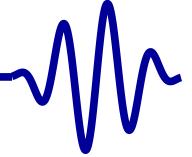
Sequences

Quizzes

Some assertions:

- Games are played on black squares
- Players will never have more than 12 pieces
- Only legal moves are possible
- Game is over when one side can no longer move

Where might the induction engine start?



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Lesson Overview

vs BMC

▷ General Rule

The Trap

Examples

Bus Properties

Free Variables

Abstraction

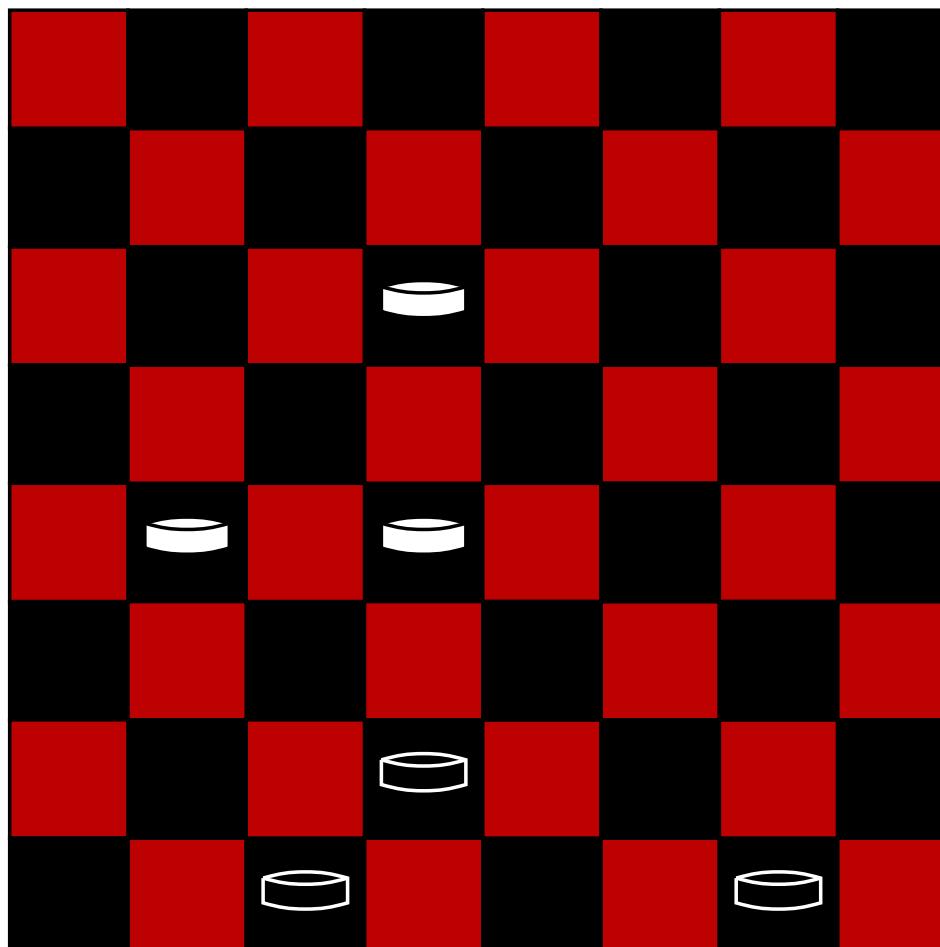
Invariants

Multiple-Clocks

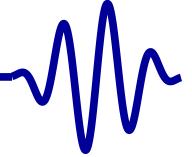
Cover

Sequences

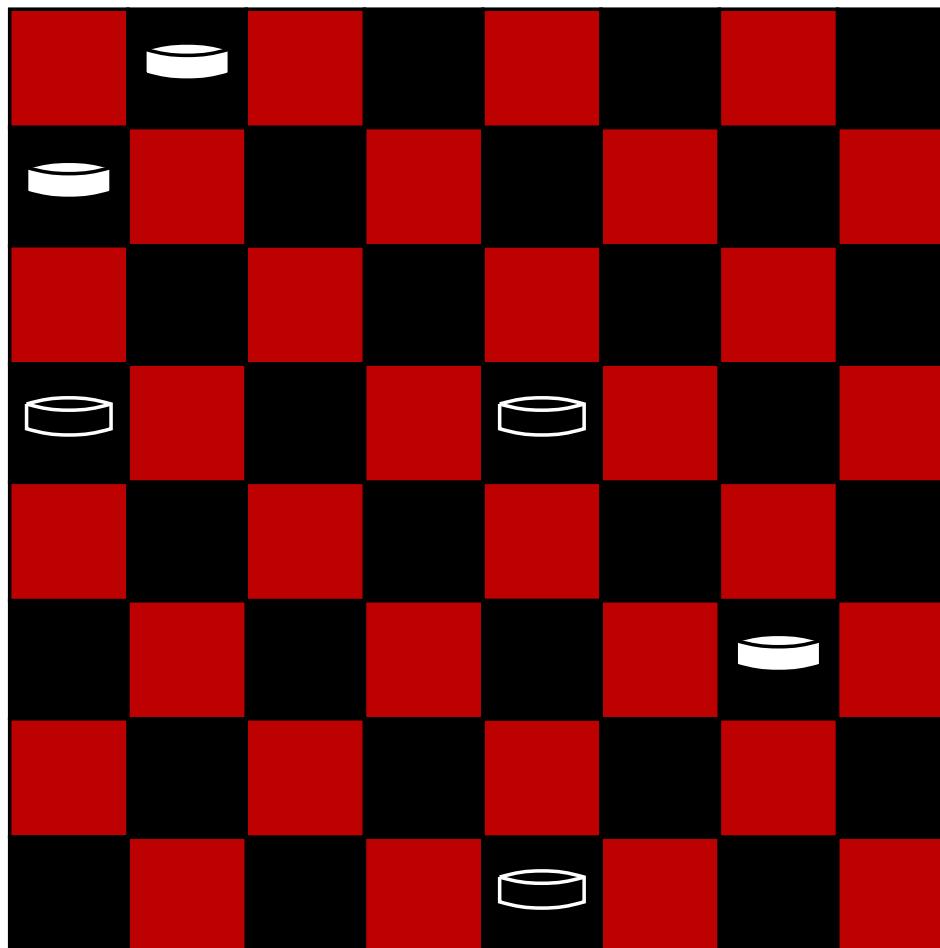
Quizzes



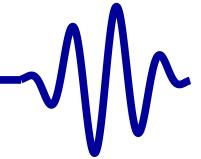
Black's going to move and win



- Welcome
- Motivation
- Basics
- Clocked and \$past
- k Induction
- Lesson Overview
- vs BMC
- ▷ General Rule
- The Trap
- Examples
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Cover
- Sequences
- Quizzes



White's going to move and win



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Lesson Overview

vs BMC

▷ General Rule

The Trap

Examples

Bus Properties

Free Variables

Abstraction

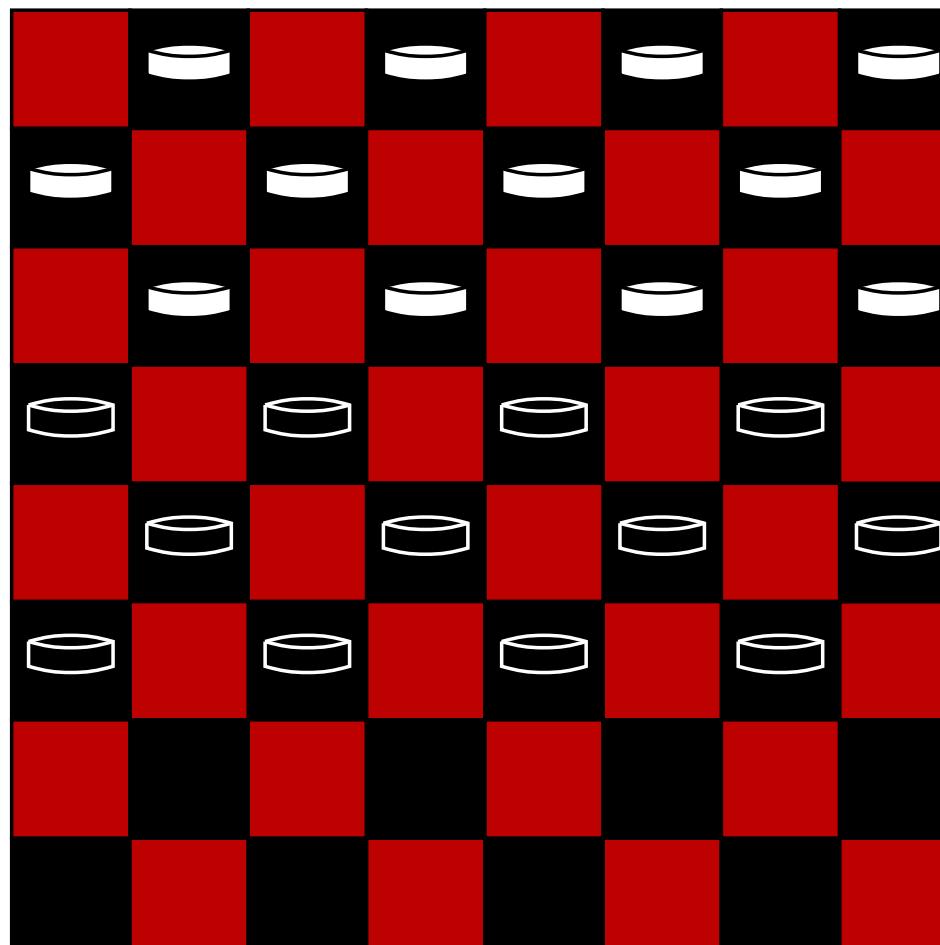
Invariants

Multiple-Clocks

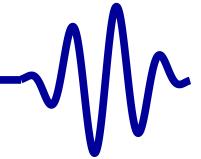
Cover

Sequences

Quizzes



Black's going to . . . , huh?



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Lesson Overview

vs BMC

▷ General Rule

The Trap

Examples

Bus Properties

Free Variables

Abstraction

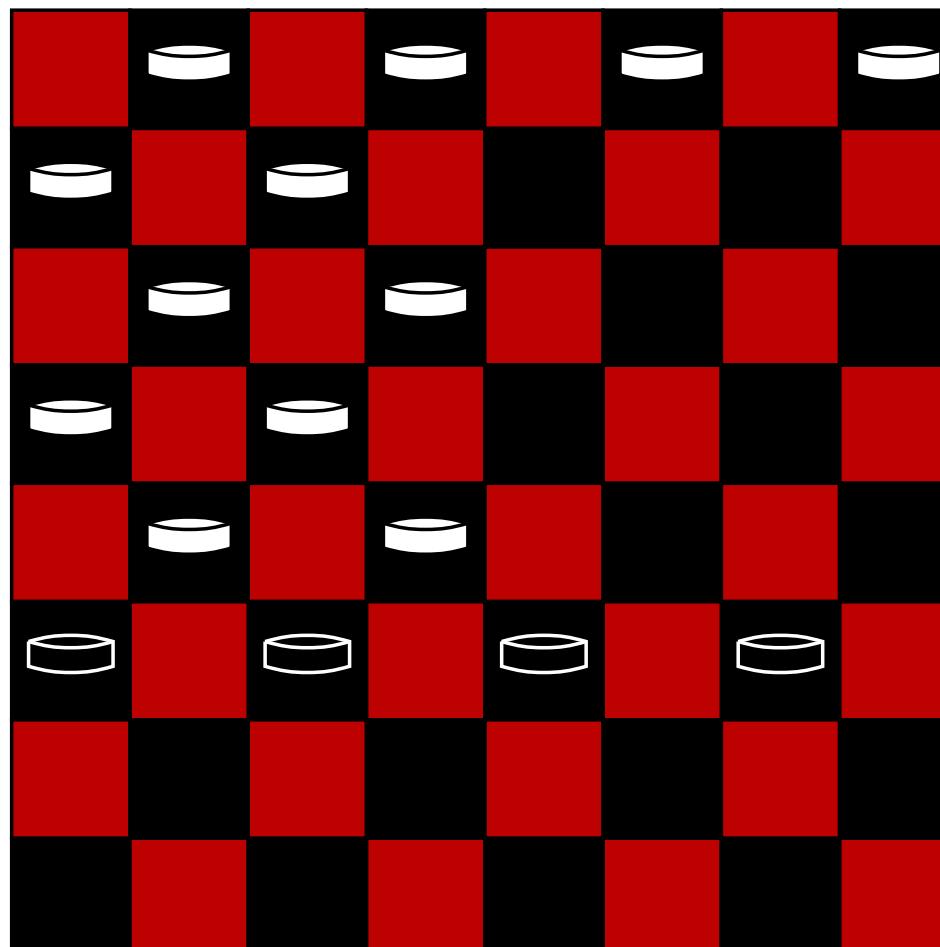
Invariants

Multiple-Clocks

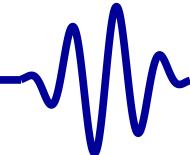
Cover

Sequences

Quizzes



Would this pass our criteria?



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Lesson Overview

vs BMC

▷ General Rule

The Trap

Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

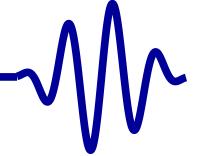
Cover

Sequences

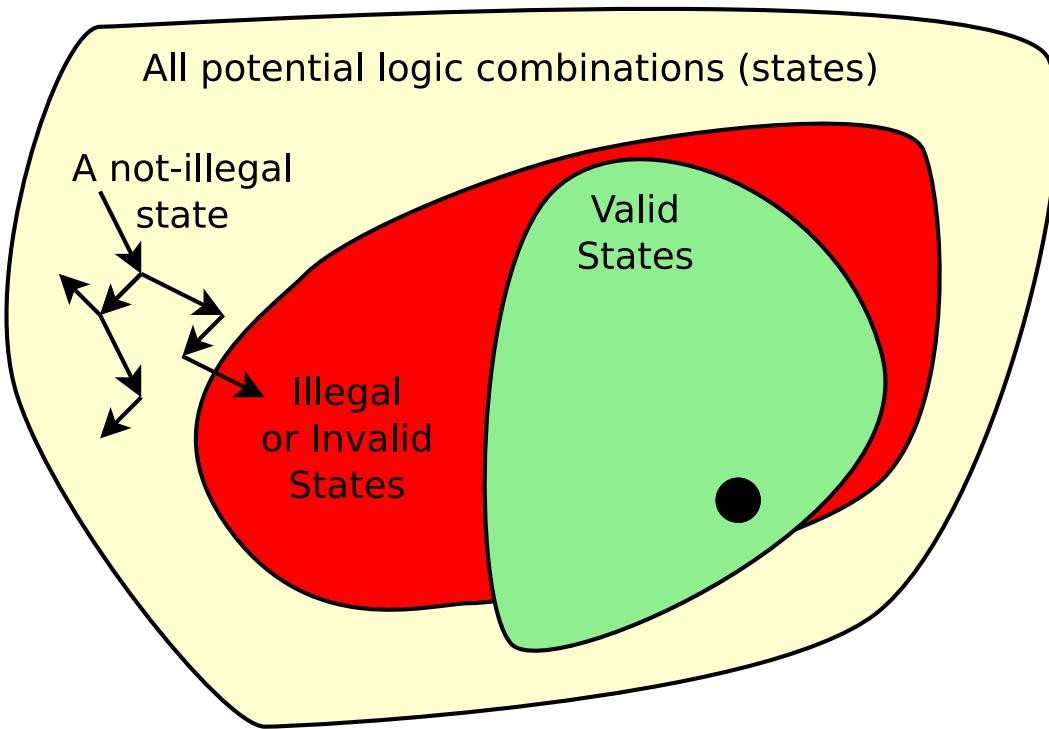
Quizzes

What can we learn from Checkers?

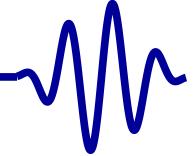
- Inductive step starts in the *middle of the game*
Only the assumptions and asserts are used to validate the game
- All of the FF's (variables) start in arbitrary states
These states are *only* constrained by your assumptions and assertions.
- Your formal constraints are required to limit the allowable states



- Welcome
- Motivation
- Basics
- Clocked and \$past
- k Induction
- Lesson Overview
- vs BMC
- General Rule
- ▷ The Trap
- Examples
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Cover
- Sequences
- Quizzes



- If your formal properties are not strict enough,
Induction may start in an illegal state
- *This is a common problem!*



Welcome

MotivationBasicsClocked and \$pastk Induction

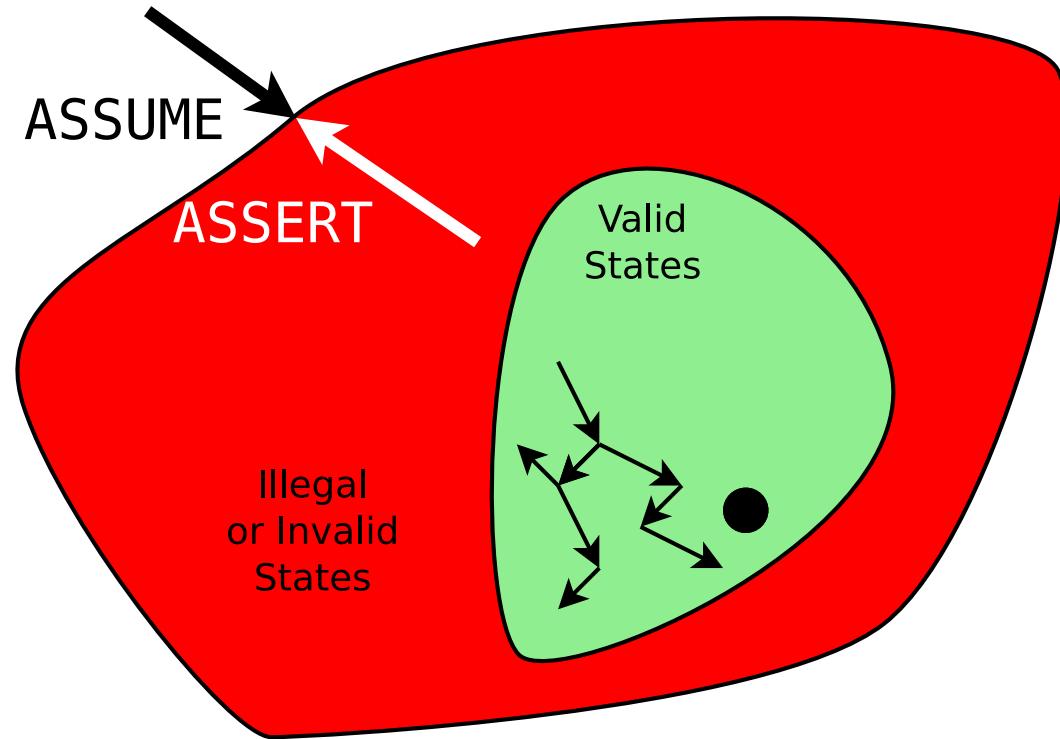
Lesson Overview

vs BMC

General Rule

▷ The Trap

Examples

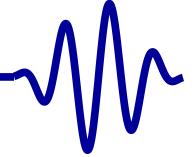
Bus PropertiesFree VariablesAbstractionInvariantsMultiple-ClocksCoverSequencesQuizzes

To make induction work, you must . . .

- **assume** unrealistic inputs will never happen
- **assert** any remaining unreachable states are illegal
- Induction often requires more properties than BMC alone



Examples



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Lesson Overview

vs BMC

General Rule

The Trap

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

- Let's look at some examples

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[▷ Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

This design would pass *many* steps of BMC

```
reg [15:0] counter;  
  
initial counter = 0;  
always @ (posedge clk)  
    counter <= counter + 1'b1;  
  
always @ (*)  
    assert(counter < 16'd65000);
```

It will not pass induction.

Can you explain why not?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[▷ Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Here's another counter that will pass BMC, but not induction

```
reg [15:0] counter;  
  
initial counter = 0;  
always @(posedge clk)  
if (counter == 16'd22)  
    counter <= 0;  
else  
    counter <= counter + 1'b1;  
  
always @(*)  
    assert(counter != 16'd500);
```

Can you explain why not?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[▷ Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

These shift registers will be equal during BMC, but require at least sixteen steps to pass induction

```
reg      [15:0]  sa , sb ;
initial sa = 0;
initial sb = 0;
always @ (posedge clk)
          sa <= { sa[14:0] , i_bit } ;

always @ (posedge clk)
          sb <= { sb[14:0] , i_bit } ;

always @ (*)
          assert (sa[15] == sb[15]);
```

Can you explain why it would take so long?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[▷ Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

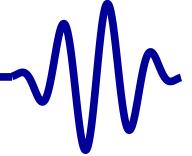
This design is almost identical to the last one, yet fails induction. The key difference is the **if (i_ce)**.

```
reg      [15:0]  sa, sb;
initial sa = 0;
initial sb = 0;
always @ (posedge clk)
    if (i_ce)
        sa <= { sa[14:0], i_bit };
always @ (posedge clk)
    if (i_ce)
        sb <= { sb[14:0], i_bit };
always @ (*)
    assert (sa[15] == sb[15]);
```

Can you explain why this wouldn't pass?



Fixing Shift Reg



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Lesson Overview

vs BMC

General Rule

The Trap

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

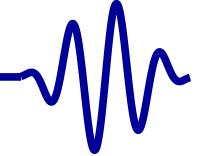
Cover

Sequences

Quizzes

Several approaches to fixing this:

1. **assume(i_ce);**



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Lesson Overview

vs BMC

General Rule

The Trap

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

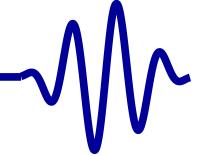
Sequences

Quizzes

Several approaches to fixing this:

1. **assume(i_ce);**
Doesn't really test the design
2. opt_merge –share_all, yosys option

Fixing Shift Reg



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Lesson Overview

vs BMC

General Rule

The Trap

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

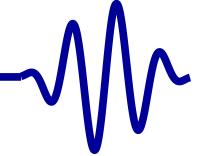
Quizzes

Several approaches to fixing this:

1. **assume(i_ce);**
Doesn't really test the design
2. opt_merge –share_all, yosys option
Works for some designs
3. **assert(sa == sb);**



Fixing Shift Reg



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Lesson Overview

vs BMC

General Rule

The Trap

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

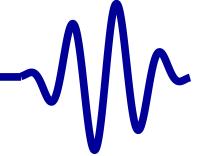
Sequences

Quizzes

Several approaches to fixing this:

1. **assume(i_ce);**
Doesn't really test the design
2. opt_merge –share_all, yosys option
Works for some designs
3. **assert(sa == sb);**
Best, but only works when sa and sb are visible
4. Insist on no more than M clocks between i_ce's

Fixing Shift Reg



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Lesson Overview

vs BMC

General Rule

The Trap

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

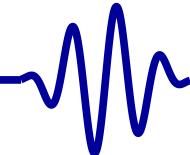
Cover

Sequences

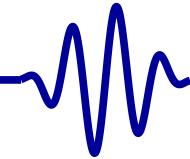
Quizzes

Several approaches to fixing this:

1. **assume(i_ce);**
Doesn't really test the design
2. opt_merge –share_all, yosys option
Works for some designs
3. **assert(sa == sb);**
Best, but only works when sa and sb are visible
4. Insist on no more than M clocks between i_ce's
5. Use a different prover, under the [**engines**] option
 - smtbmc
 - abc pdr
 - aiger avy
 - aiger suprove

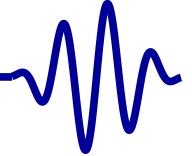


Several approaches to fixing this:



Several approaches to fixing this:

Most of these options work for *some* designs only



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Lesson Overview

vs BMC

General Rule

The Trap

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Here's how we'll change our sby file:

[**options**]

mode prove

[**engines**]

smtbmc

[**script**]

read -formal module.v

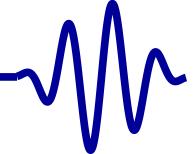
... other files would go here

prep -top module

opt_merge -share_all

[**files**]

.. / path-to/module.v

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[▷ Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Here's how we'll change our sby file:

[**options**]

mode prove ← Use BMC and *k*-induction

[**engines**]

smtbmc

[**script**]

read -formal module.v

... other files would go here

prep -top module

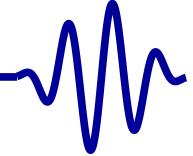
opt_merge -share_all

[**files**]

../path-to/module.v



SymbiYosys



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Lesson Overview

vs BMC

General Rule

The Trap

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Here's how we'll change our sby file:

```
[ options ]
```

```
mode prove
```

```
[ engines ]
```

```
smtbmc ← Other potential engines would go here
```

```
[ script ]
```

```
read -formal module.v
```

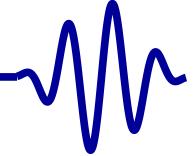
```
# ... other files would go here
```

```
prep -top module
```

```
opt_merge -share_all
```

```
[ files ]
```

```
../path-to/module.v
```



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Lesson Overview

vs BMC

General Rule

The Trap

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Here's how we'll change our sby file:

```
[ options ]
```

```
mode prove
```

```
[ engines ]
```

```
smtbmc
```

```
[ script ]
```

```
read -formal module.v
```

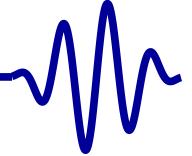
```
# ... other files would go here
```

```
prep -top module
```

```
opt_merge -share_all ← Here's where opt_merge would go
```

```
[ files ]
```

```
../path-to/module.v
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[▷ Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Exercise #4: dblpipe.v

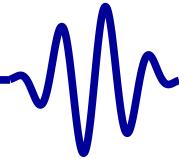
```
module dblpipe(i_clk,
               i_ce, i_data, o_data);
  // ...

  wire a_data, b_data;

  lfsr_fib one(i_clk, 1'b0, i_ce,
                i_data, a_data);
  lfsr_fib two(i_clk, 1'b0, i_ce,
                i_data, b_data);

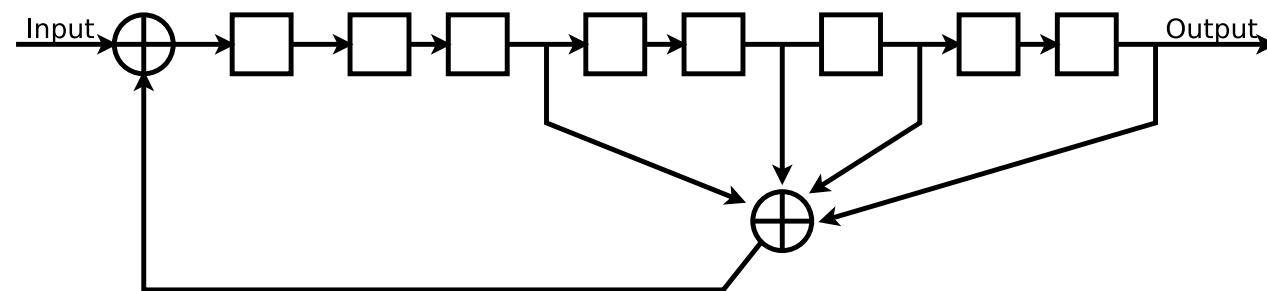
  initial o_data = 1'b0;
  always @ (posedge i_clk)
    o_data <= a_data ^ b_data;

endmodule
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[▷ Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Exercise #4: dblpipe.v

- `lfsr_fib` just implements a Fibonacci linear feedback shift register,



```
sreg[(LN-2):0] <= sreg[(LN-1):1];
sreg[(LN-1)] <= (^ (sreg & TAPS)) ^ i_in;
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[▷ Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

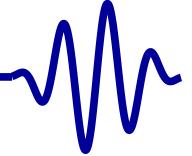
Exercise #4: dblpipe.v, lfsr_fib.v

```
reg      [(LN-1):0]      sreg;  
  
initial sreg = INITIAL_FILL;  
always @(posedge i_clk)  
if (i_reset)  
    sreg <= INITIAL_FILL;  
else if (i_ce)  
begin // Basic shift register update operation  
    sreg[(LN-2):0] <= sreg[(LN-1):1];  
    sreg[(LN-1)] <= (^ (sreg & TAPS)) ^ i_in;  
end  
  
assign o_bit = sreg[0];
```

- Both registers one and two use *the exact same logic*



Ex: DblPipe



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Lesson Overview

vs BMC

General Rule

The Trap

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

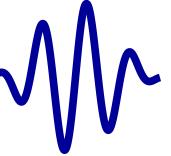
Cover

Sequences

Quizzes

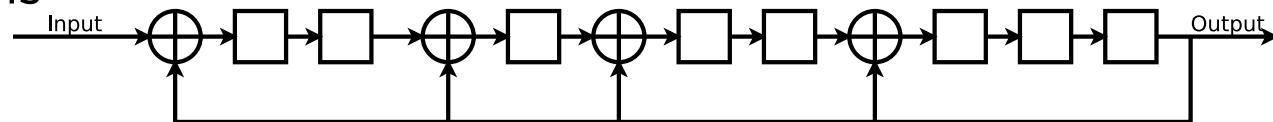
Exercise #4:

- Using dblpipe.v
 - Prove that the output, o_data, is zero

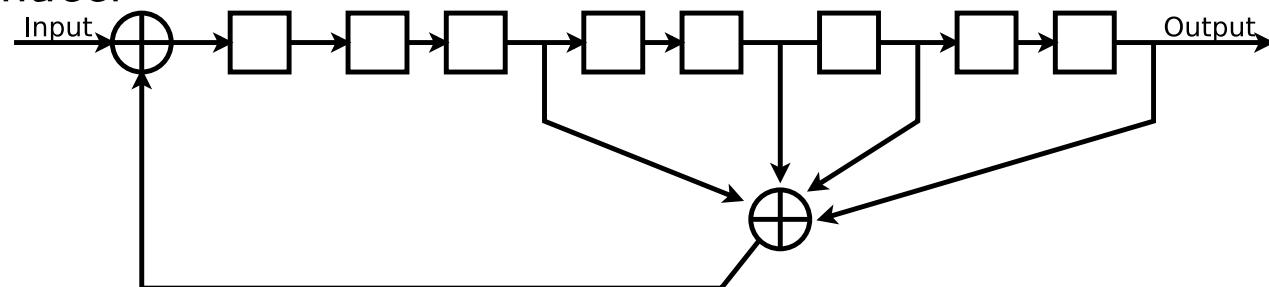
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[▷ Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Galois and Fibonacci are supposedly identical

- Galois



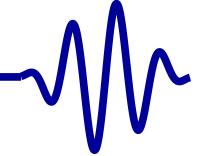
- Fibonacci



- Exercise #5 will be to prove these two implementations are identical



Ex: LFSRs



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Lesson Overview

vs BMC

General Rule

The Trap

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

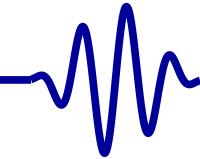
Quizzes

Exercise #5:

- exercise-05/ contains files `lfsr_equiv.v`, `lfsr_gal.v`, and `lfsr_fib.v`.
- `lfsr_gal.v` contains a Galois version of an LFSR
- `lfsr_fib.v` contains a Fibonacci version of the same LFSR
- `lfsr_equiv.v` contains an assertion that these are equivalent

Prove that these are truly equivalent shift registers.

Where is the bug?



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Lesson Overview

vs BMC

General Rule

The Trap

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

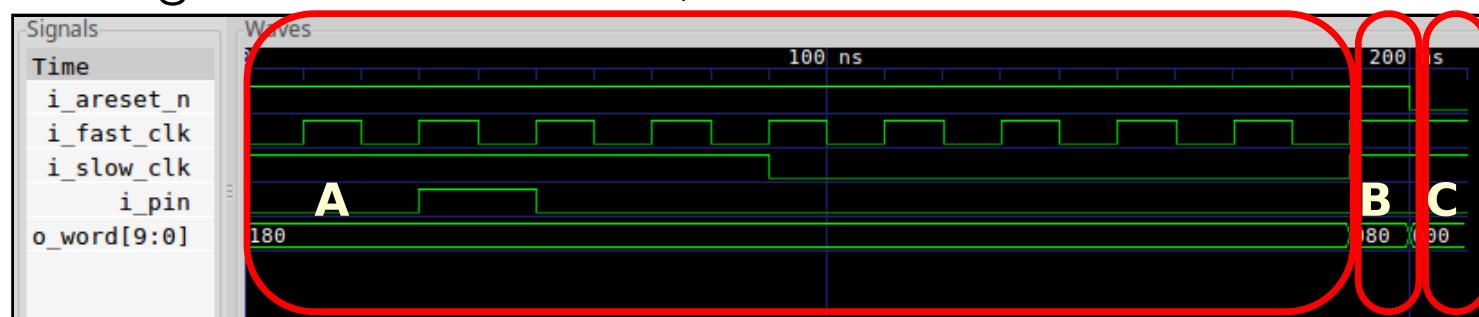
Multiple-Clocks

Cover

Sequences

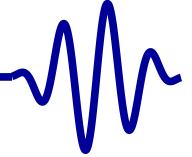
Quizzes

Following an induction failure, look over the trace



If you see a problem in section ...

- A You have a missing one or more assertions
You'll only have this problem with induction.
- B You have a failing **assert @(posedge clk)**
- C You have a failing **assert @(*)**
These latter two indicate a potential logic failure, but they could still be caused by property failures.



Welcome

Motivation

Basics

Clocked and \$past

k Induction

▷ Bus Properties

Ex: WB Bus

AXI

Avalon

Wishbone

WB Basics

WB Basics

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Bus Properties

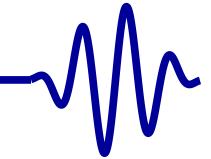
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[▷ Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

We have everything we need now to write formal properties for a bus

- This lesson walks through an example the Wishbone Bus

Our Objectives:

- Learn to apply formal methods to something imminently practical
- Learn to build the formal description of a bus component
- Help lead up to a bus arbiter component



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Ex: WB Bus

▷ AXI

Avalon

Wishbone

WB Basics

WB Basics

Free Variables

Abstraction

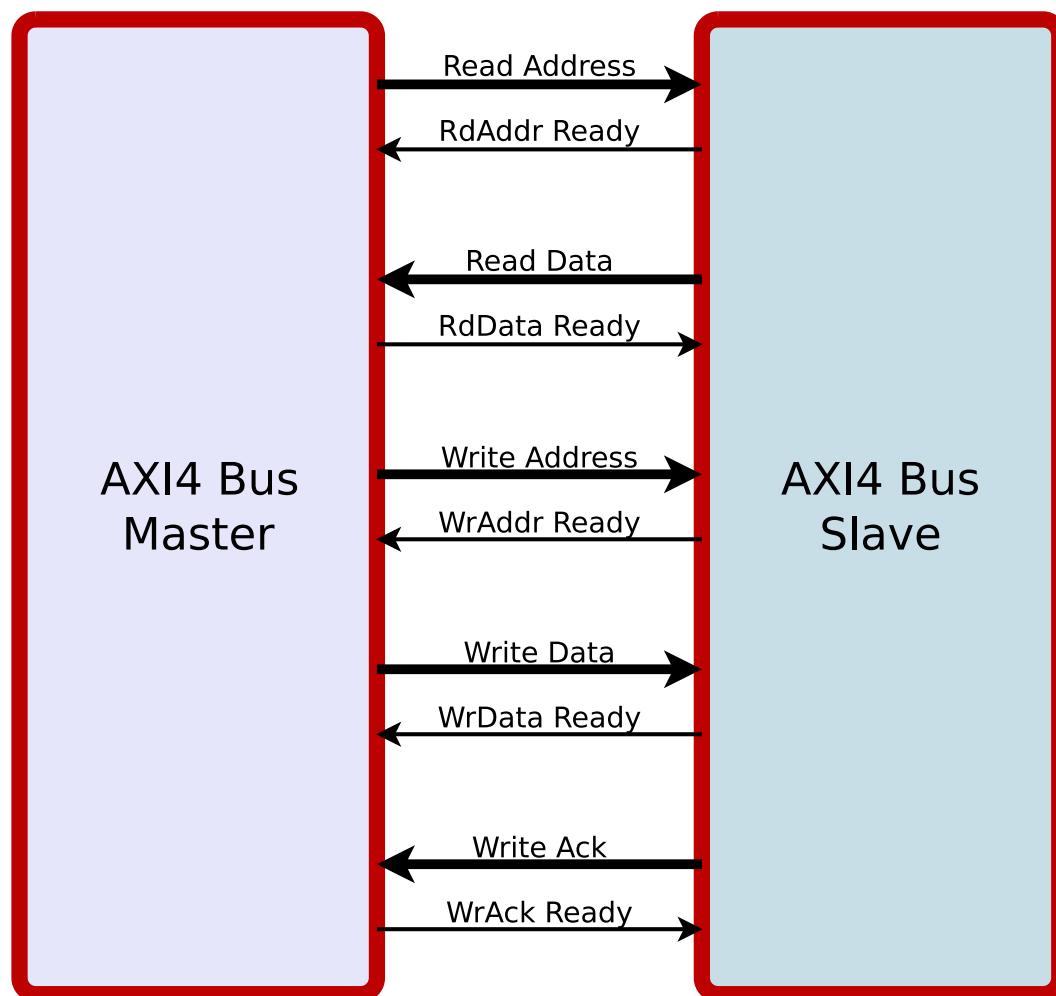
Invariants

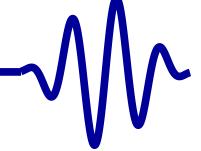
Multiple-Clocks

Cover

Sequences

Quizzes





Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Ex: WB Bus

AXI

▷ Avalon

Wishbone

WB Basics

WB Basics

Free Variables

Abstraction

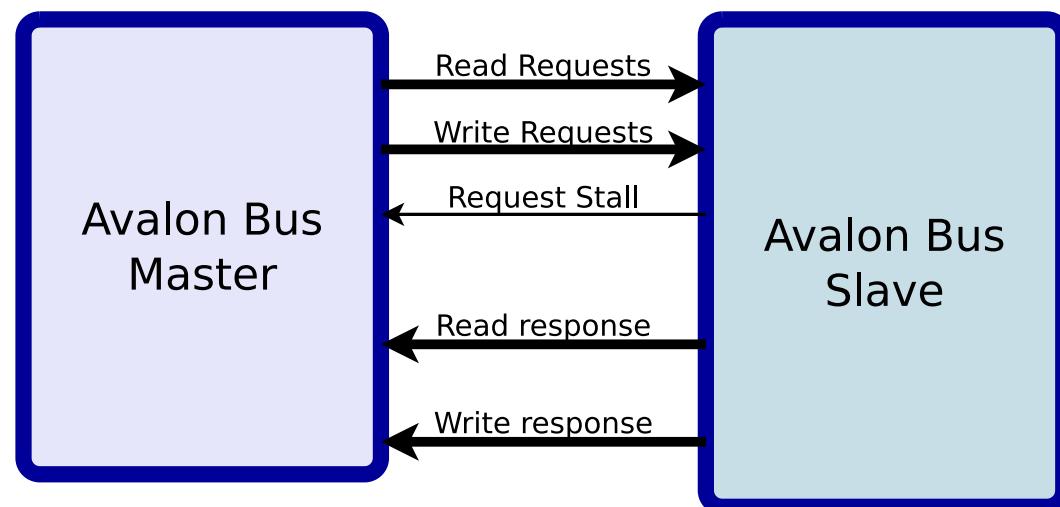
Invariants

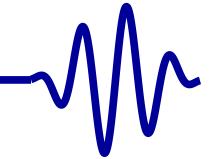
Multiple-Clocks

Cover

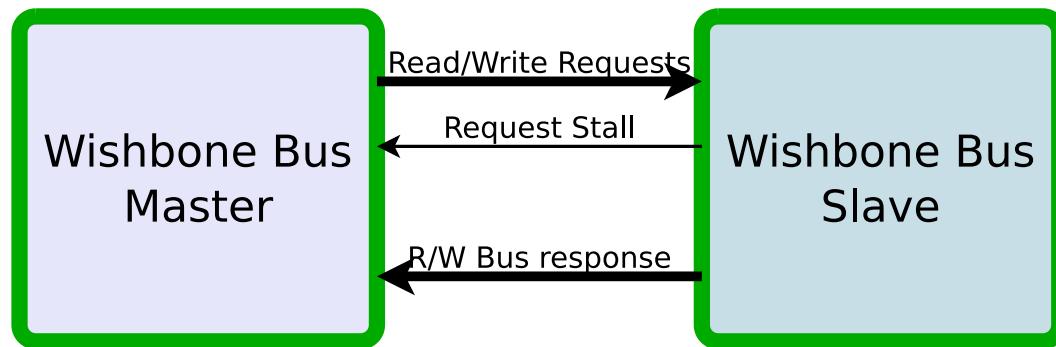
Sequences

Quizzes

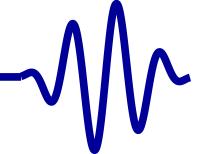




- Welcome
- Motivation
- Basics
- Clocked and \$past
- k Induction
- Bus Properties
 - Ex: WB Bus
 - AXI
 - Avalon
 - ▷ Wishbone
 - WB Basics
 - WB Basics
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Cover
- Sequences
- Quizzes



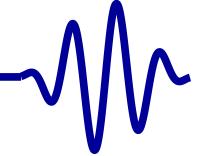
- Why use the Wishbone? *It's simpler!*

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[▷ WB Basics](#)[WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

From the master's perspective:

Specification name	My name
CYC_O	o_wb_cyc
STB_O	o_wb_stb
WE_O	o_wb_we
ADDR_O	o_wb_addr
DATA_O	o_wb_data
SEL_O	o_wb_sel
STALL_I	i_wb_stall
ACK_I	i_wb_ack
DATA_I	i_wb_data
ERR_I	i_wb_err

WB Signals



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Ex: WB Bus

AXI

Avalon

Wishbone

▷ WB Basics

WB Basics

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

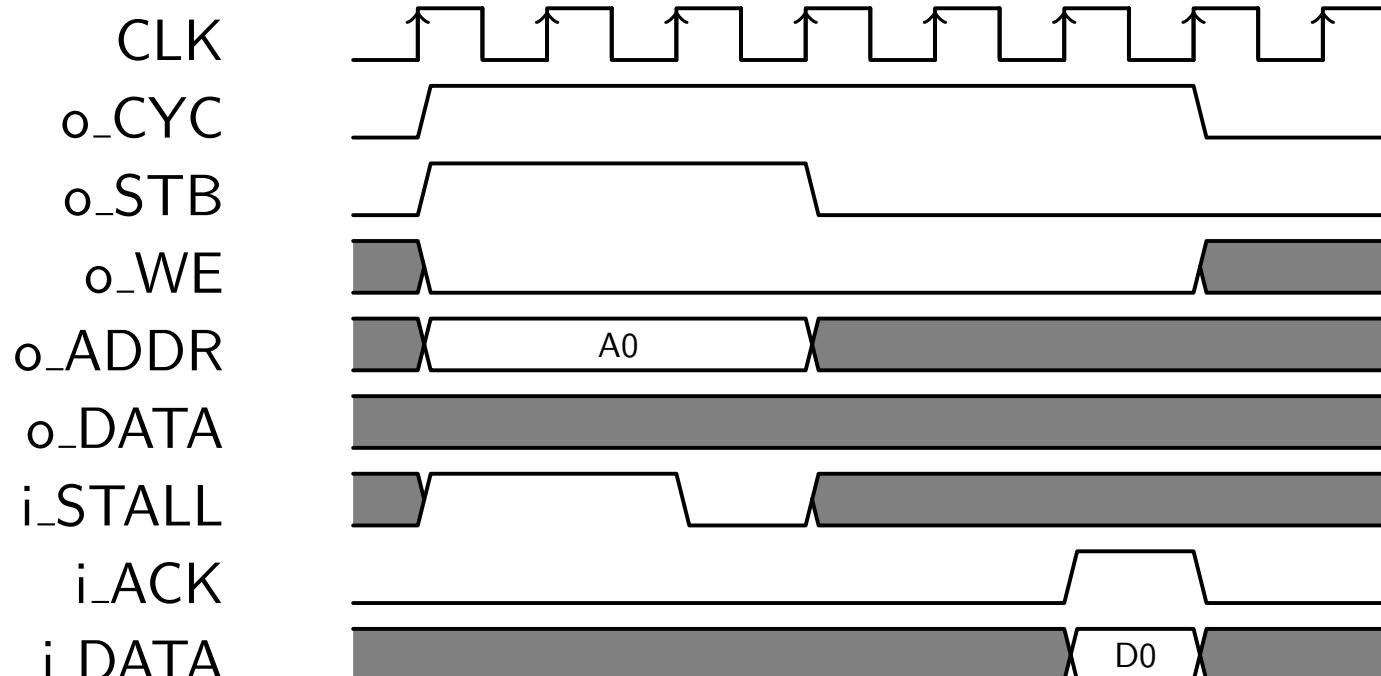
Quizzes

From the slave's perspective:

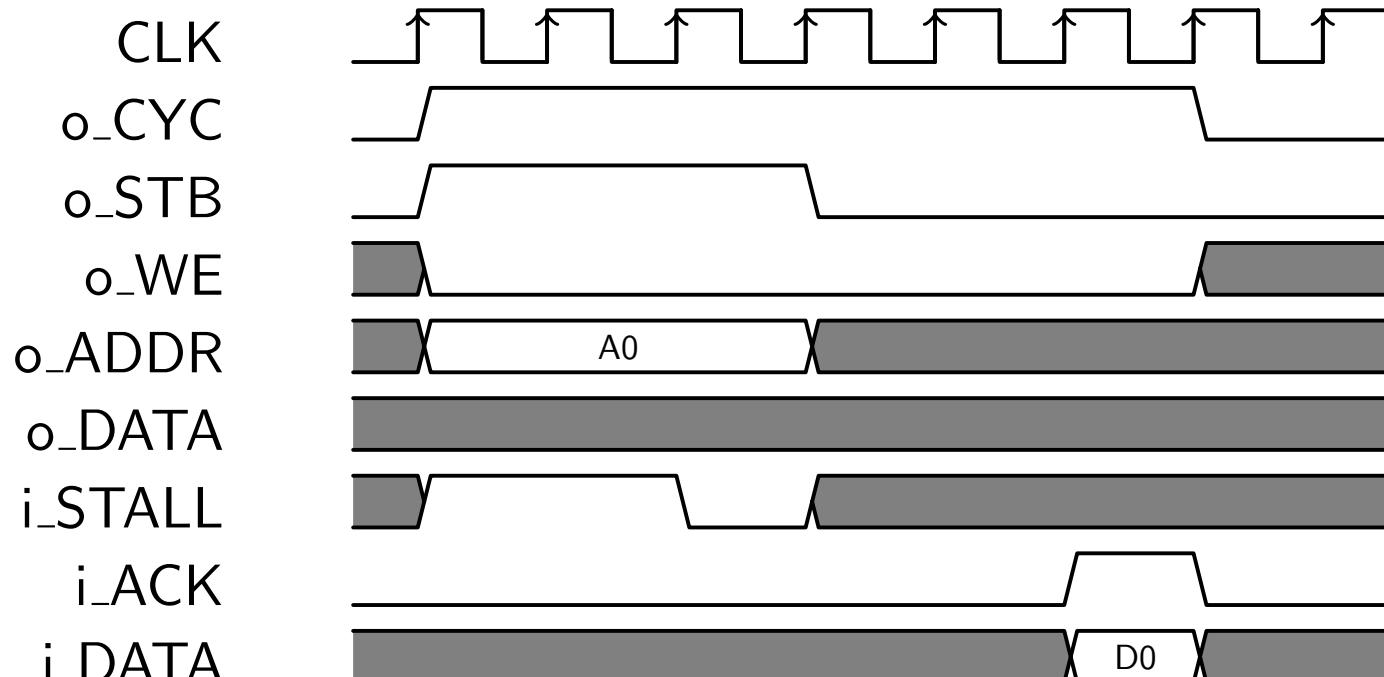
Specification name	My name
CYC_I	i_wb_cyc
STB_I	i_wb_stb
WE_I	i_wb_we
ADDR_I	i_wb_addr
DATA_I	i_wb_data
SEL_I	i_wb_sel
STALL_O	o_wb_stall
ACK_O	o_wb_ack
DATA_O	o_wb_data
ERR_O	o_wb_err

To swap perspectives from master to slave ...

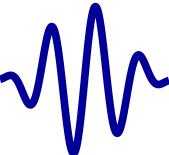
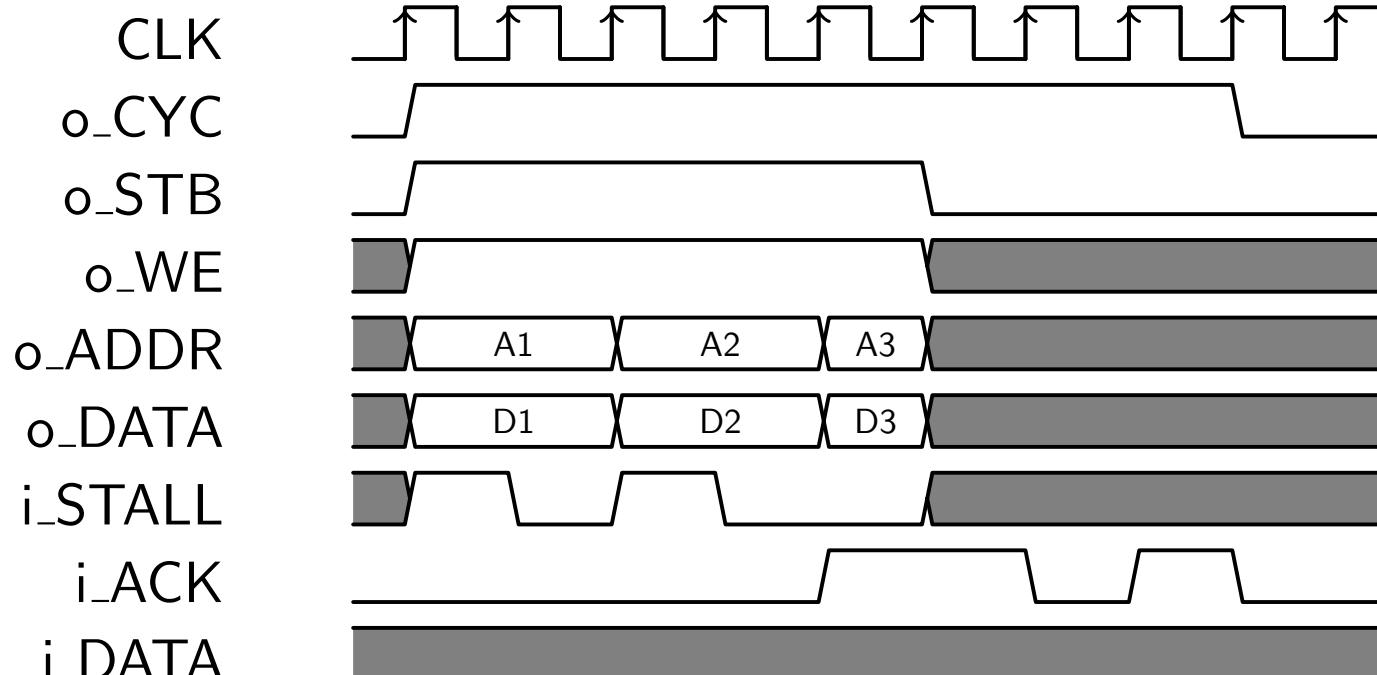
- Swap the port direction
- Swap the **assume()** statements for **assert()**s

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[▷ WB Basics](#)[WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- STB must be low when CYC is low
- If CYC goes low mid-transaction, the transaction is aborted
- While STB and STALL are active, the request cannot change
- One request is made for every clock with STB and !STALL

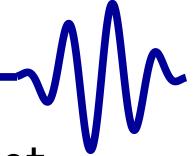
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[▷ WB Basics](#)[WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- One ACK response per request
- No ACKs allowed when the bus is idle
- No way to stall the ACK line
- The bus result is in i_DATA when i_ACK is true

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[▷ WB Basics](#)[WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Let's start building some formal properties

GT CYC and STB



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Ex: WB Bus

AXI

Avalon

Wishbone

▷ WB Basics

WB Basics

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

- The bus starts out idle, and returns to idle after a reset

```
always @(posedge i_clk)
  if ((!f_past_valid)||($past(i_reset)))
    begin
      assume (!i_wb_ack);
      assume (!i_wb_err);
      //
      assert (!o_wb_cyc);
      assert (!o_wb_stb);
    end
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[▷ WB Basics](#)[WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- The bus starts out idle, and returns to idle after a reset

```
always @(posedge i_clk)
  if ((!f_past_valid)||($past(i_reset)))
    begin
      assume (!i_wb_ack);
      assume (!i_wb_err);
      //
      assert (!o_wb_cyc);
      assert (!o_wb_stb);
    end
```

- STB is low whenever CYC is low

```
always @(*)
  if (!o_wb_cyc)
    assert (!o_wb_stb);
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[▷ WB Basics](#)[WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- While STB and STALL are active, the request doesn't change

```
assign f_request = { o_stb, o_we, o_addr,
                     o_data };
always @(posedge clk)
if ($past(o_wb_stb)&&($past(i_wb_stall)))
    assert(f_request == $past(f_request));
```

- Did we get it?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[▷ WB Basics](#)[WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- While STB and STALL are active, the request doesn't change

```
assign f_request = { o_stb, o_we, o_addr,  
                     o_data };  
always @(posedge clk)  
if ($past(o_wb_stb)&&($past(i_wb_stall)))  
    assert(f_request == $past(f_request));
```

- Did we get it? Well, not quite
o_data is a don't care for any read request

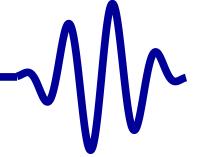
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[▷ WB Basics](#)[WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- While STB and STALL are active, the request doesn't change

```
assign f_rd_request = { o_stb, o_we, o_addr };
assign f_wr_request = { f_rd_request, o_data };

always @(posedge clk)
if ((f_past_valid)
  &&($past(o_wb_stb))&&($past(i_wb_stall)))
begin
  // First, for reads—o_data is a don't care
  if ($past(!i_wb_we))
    assert(f_rd_request == $past(f_rd_request));
  // Second, for writes—o_data must not change
  if ($past(i_wb_we))
    assert(f_wr_request == $past(f_wr_request));
end
```

GT CYC and STB



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Ex: WB Bus

AXI

Avalon

Wishbone

WB Basics

▷ WB Basics

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

- No acknowledgements without a request
- No errors without a request
- Following any error, the bus cycle ends
- A bus cycle can be terminated early

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

The rule: the slave (external) cannot stall the master more than F_OPT_MAXSTALL counts:

```
initial f_stall_count = 0;
always @(posedge i_clk)
if ((i_reset)||(!o_CYC)|| ((o_STB)&&(!i_STALL)))
    f_stall_count <= 0;
else if (o_STB)
    f_stall_count <= f_stall_count + 1'b1;

always @(posedge i_clk)
if (o_CYC)
    assume(f_stall_count < F_OPT_MAXSTALL);
```

This solves the i_ce problem, this time with the i_STALL signal

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)

Ex: WB Bus

AXI

Avalon

Wishbone

WB Basics

▷ WB Basics

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

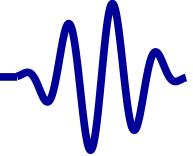
The rule: the slave can only respond to requests

```
initial f_nreqs = 0;
always @(posedge clk)
if ((i_reset)||(!i_CYC))
    f_nreqs <= 1'b0;
else if ((i_STB)&&(!o_STALL))
    f_nreqs <= f_nreqs + 1'b1;
// Similar counter for acknowledgements
always @(*)
if (f_nreqs == f_nacks)
    assert (!o_ACK);
```

The logic above *almost* works. Can any one spot the problems?



Two Exercises



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Ex: WB Bus

AXI

Avalon

Wishbone

WB Basics

▷ WB Basics

Free Variables

Abstraction

Invariants

Multiple-Clocks

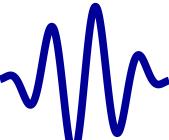
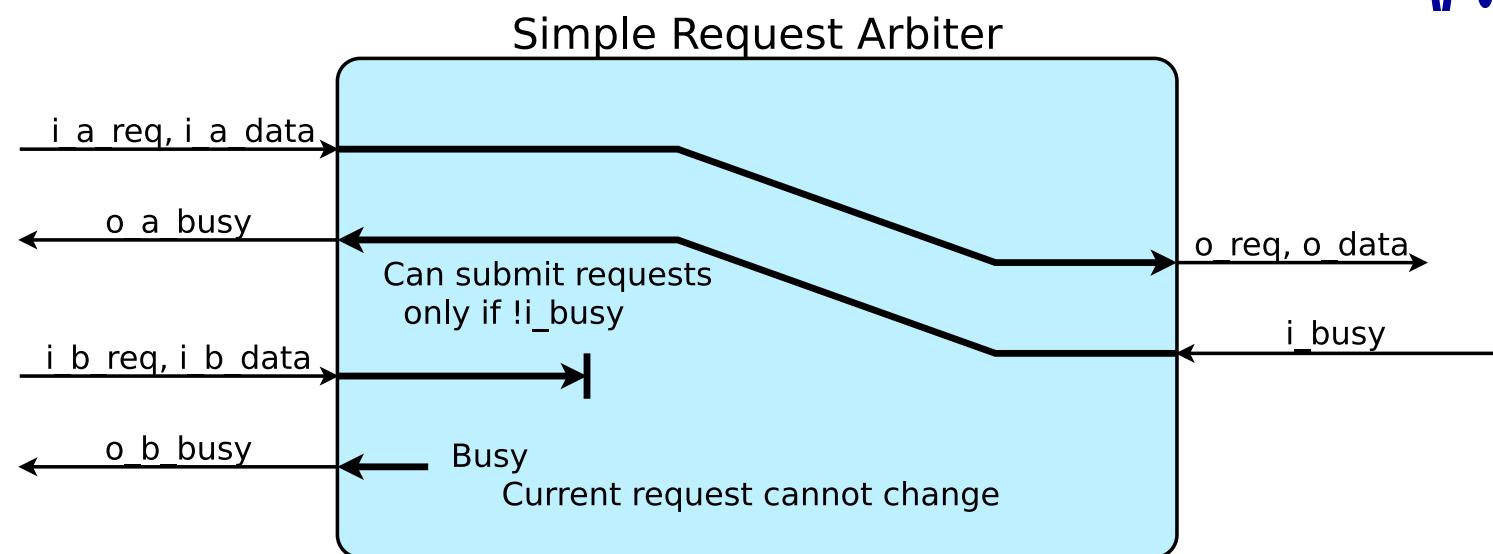
Cover

Sequences

Quizzes

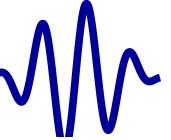
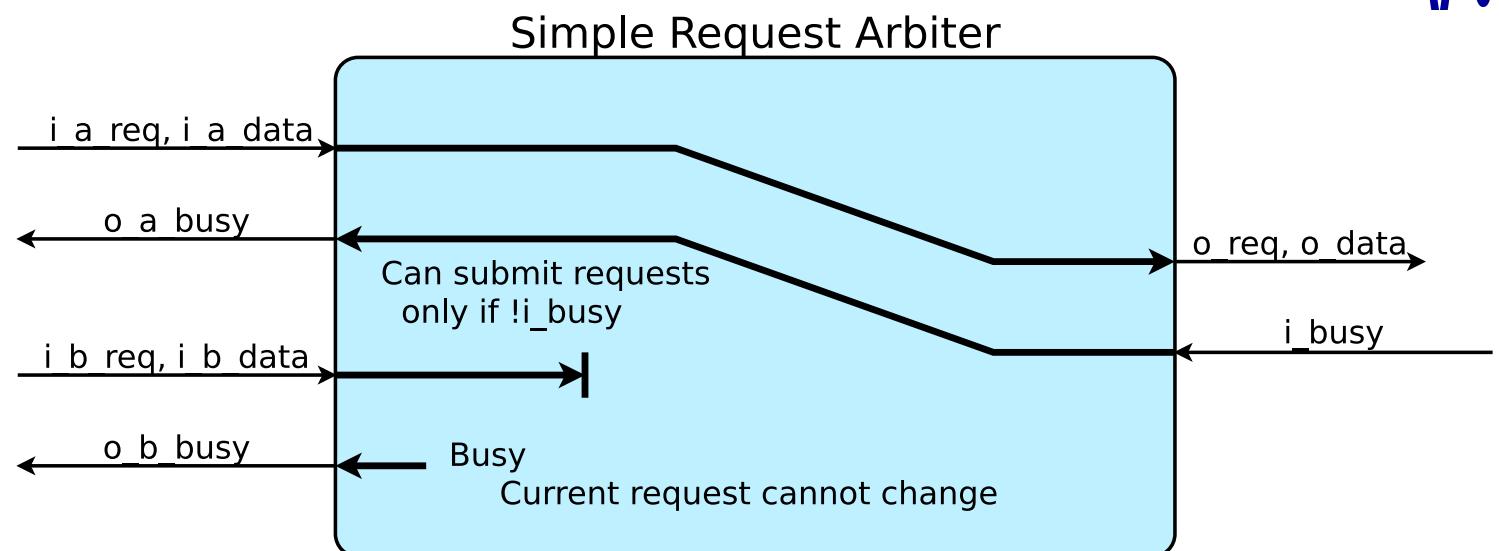
Let's build up to proving a WB arbiter

- Let's prove (BMC + k -Induction) . . .
 1. Exercise #6: A simple arbiter
`exercise-06/reqlarb.v`
 2. Exercise #7: Then a Wishbone bus arbiter
`exercise-07/wbpriarbiter.v`
- Given a set of bus properties: `fwb_slave.v`

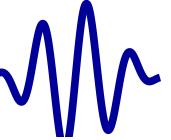
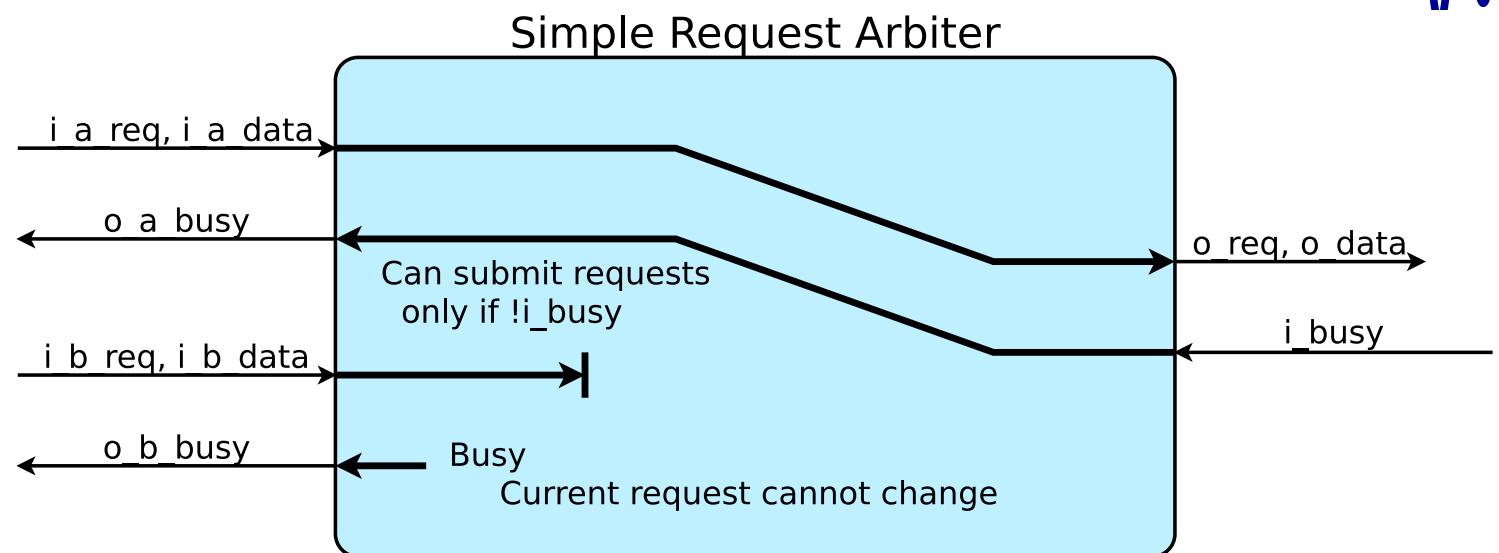
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

The basics

- *_req requests a transaction
- *_data, the contents of the transaction
- *_busy, true if the source must wait

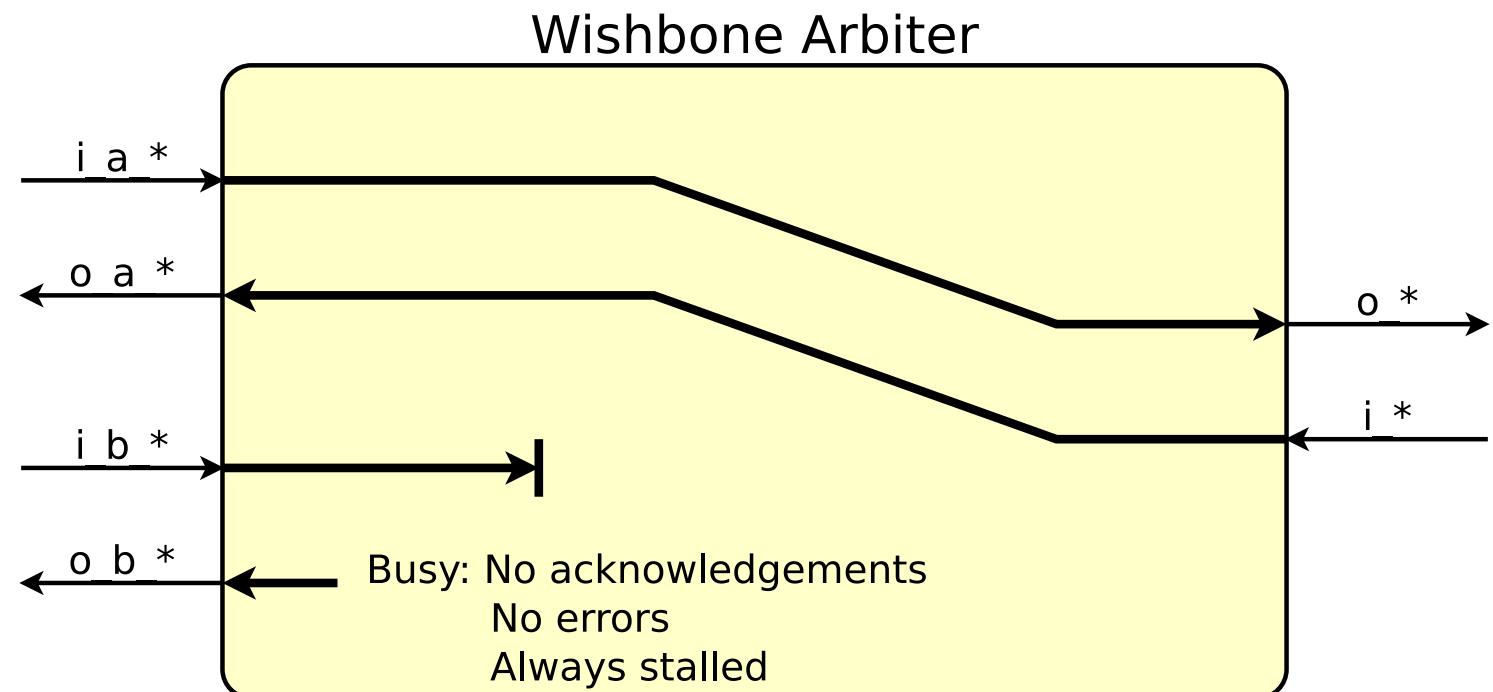
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- If $(*_\text{req}) \&\& (\neg *_\text{busy})$,
the request is accepted
- If $(*_\text{req}) \&\& (*_\text{busy})$,
the request may not change, except on reset

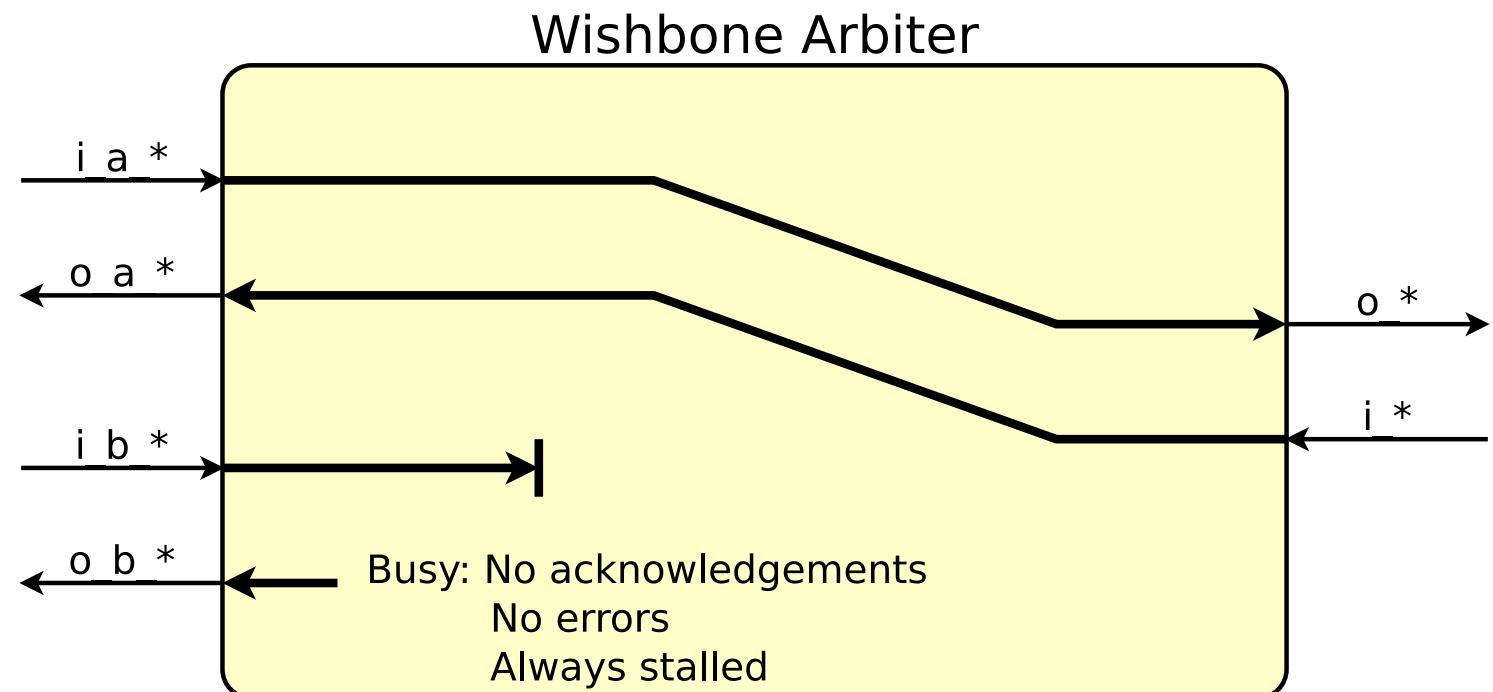
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

To prove:

- No data will be lost, no requests will be dropped
Assume all requests remain stable until accepted
- Only one source ever gets access at a time
Assert one busy line is always high
- Therefore, all requests go through . . . eventually

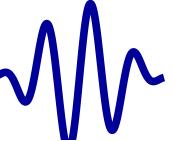
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Shall we try this with Wishbone?

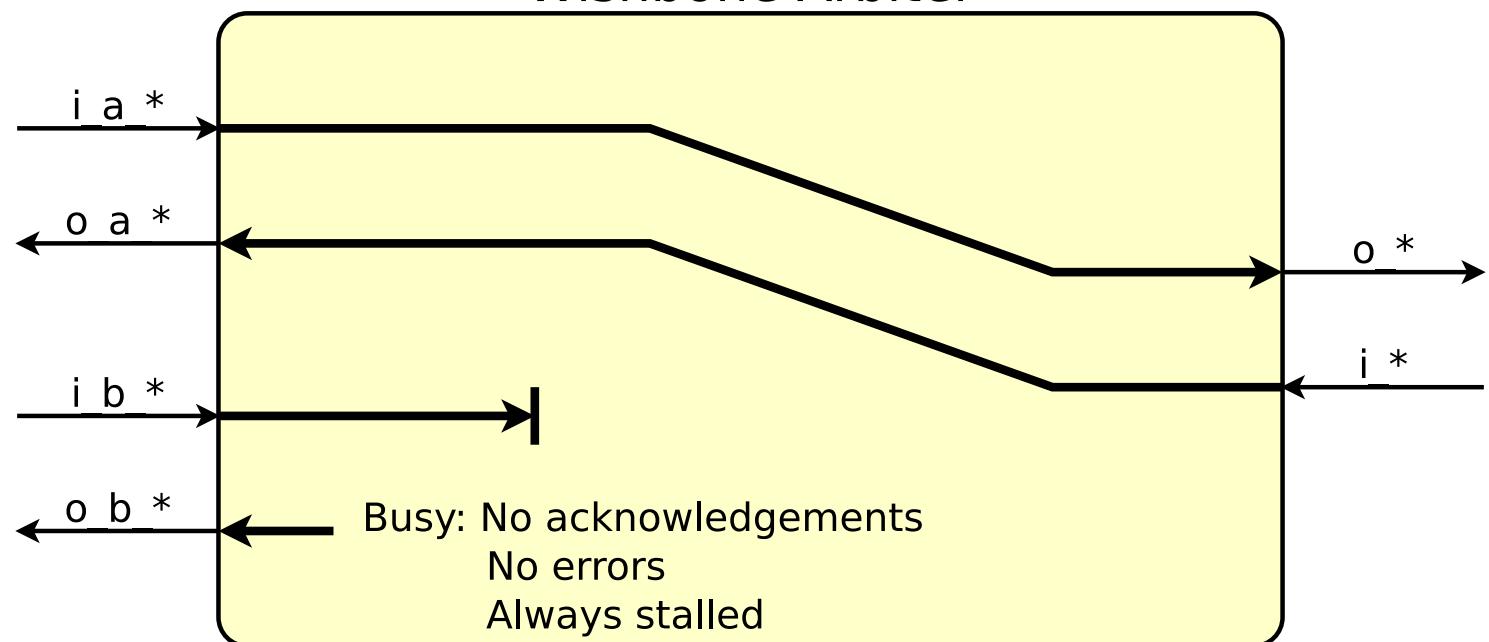
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

This request side is almost identical

- If $(STB) \&\& (!STALL)$
the request is accepted
- If $(STB) \&\& (STALL)$
the request must not change

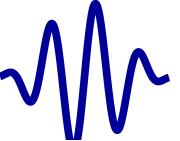
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Wishbone Arbiter

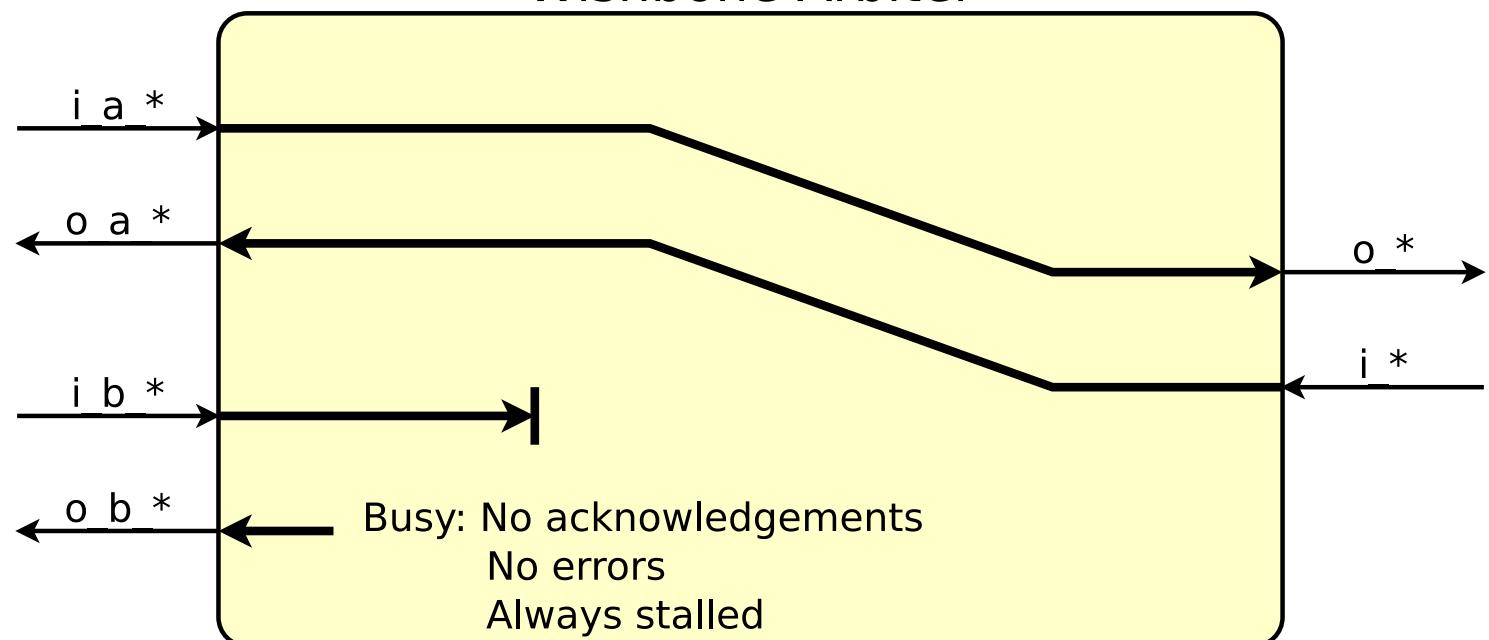


The difference is the acknowledgements

- The arbiter cannot change during an active transaction
- All requests get responses
- No response can be returned without a request

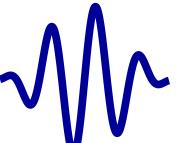
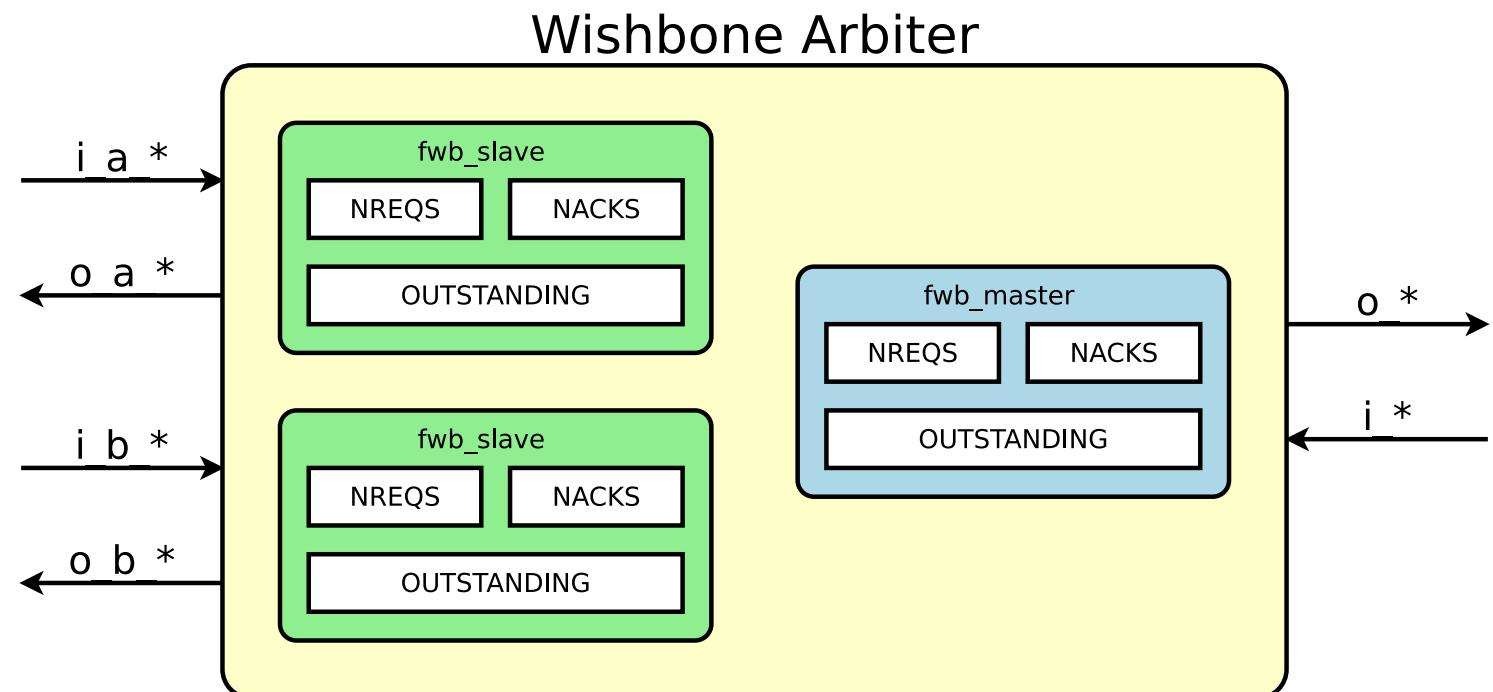
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Wishbone Arbiter



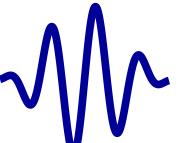
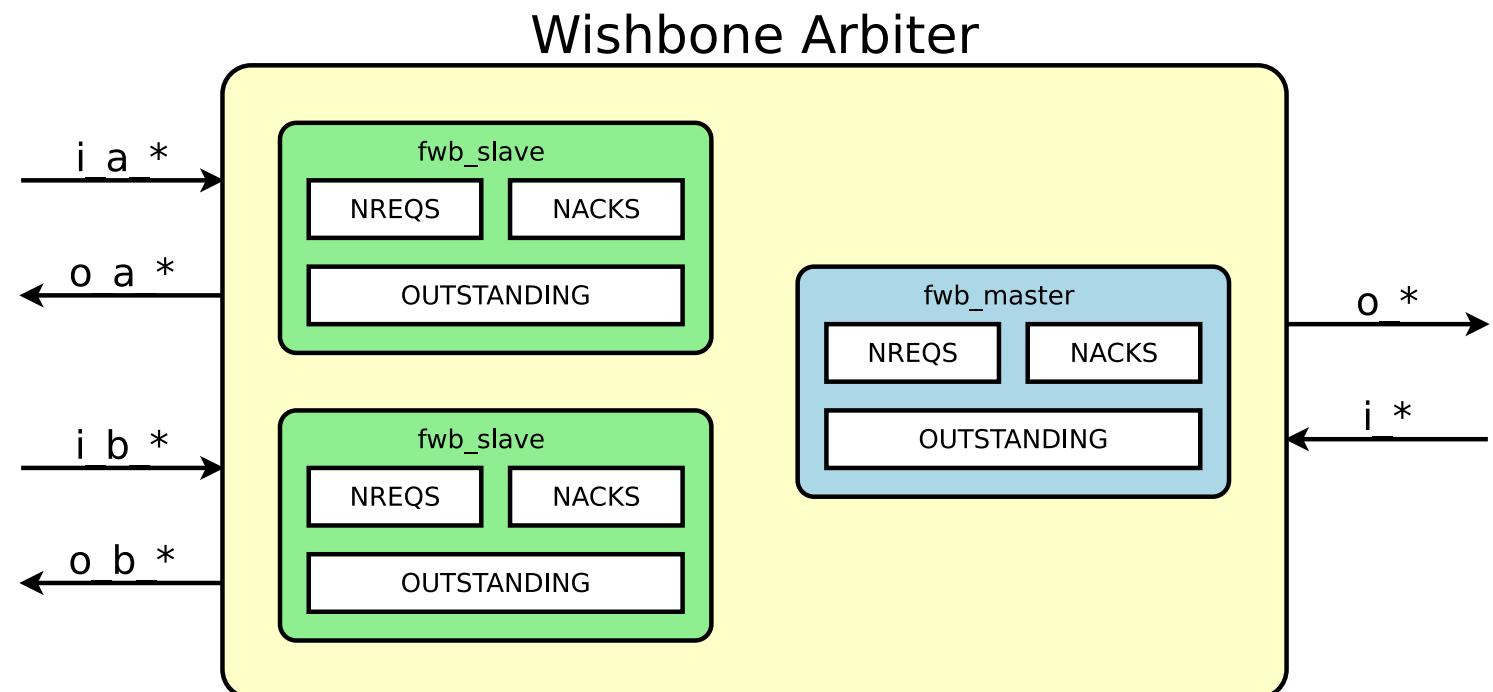
Now, prove that `exercise-07/wbpriarbiter.v` works.

- Use both BMC and k -induction (move prove)
- You'll need to build `fwb_master.v` properties

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

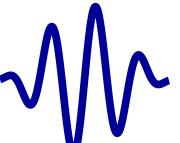
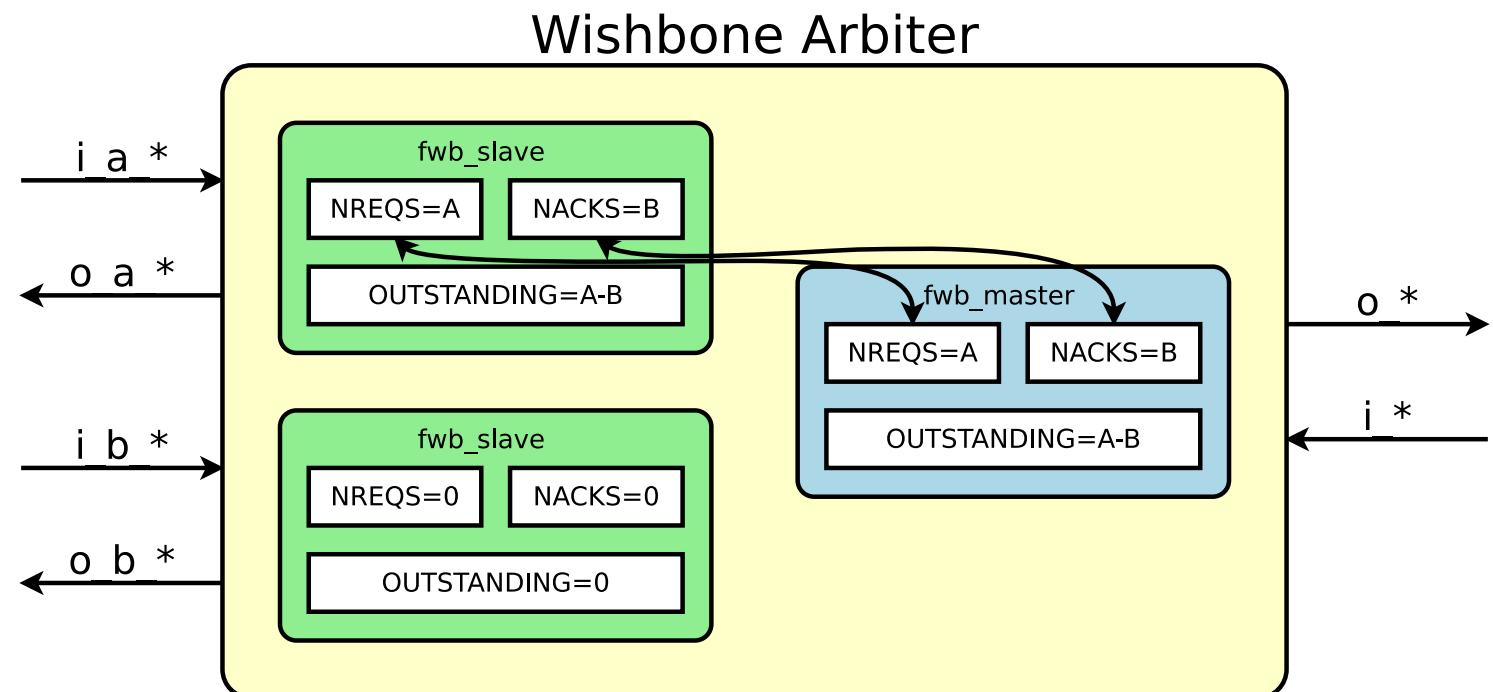
The `fwb_slave.v` properties will

- Assume a behaving master
- Assert a behaving slave

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

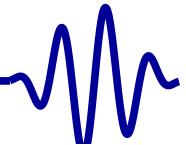
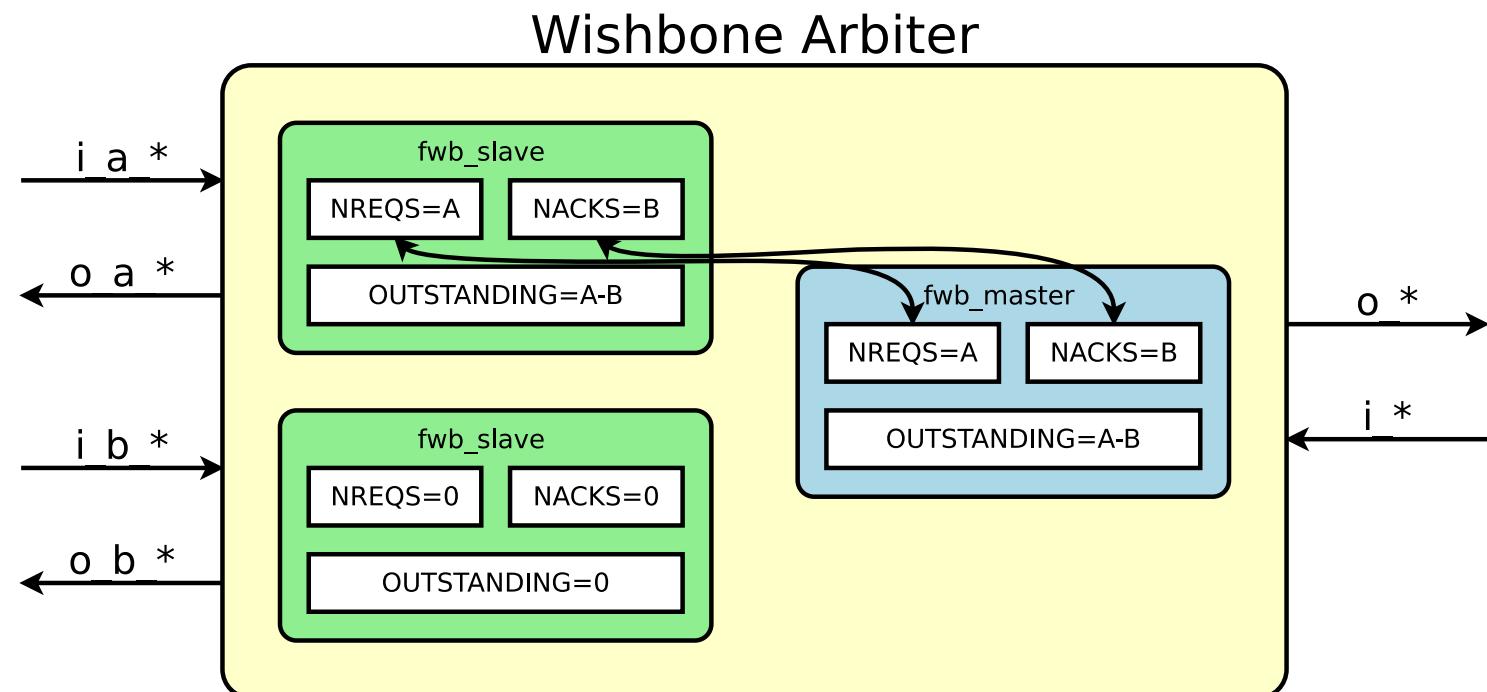
You'll write the `fwb_master.v` properties

- Swapping inputs with outputs
 - Port names need not change
- Swapping assumptions with assertions

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

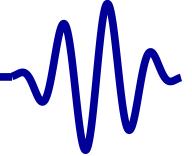
The magic is in how the files are connected

- If one interface is connected, both master and slave...
 - Should see the same number of requests
 - Should see the same number of acknowledgements

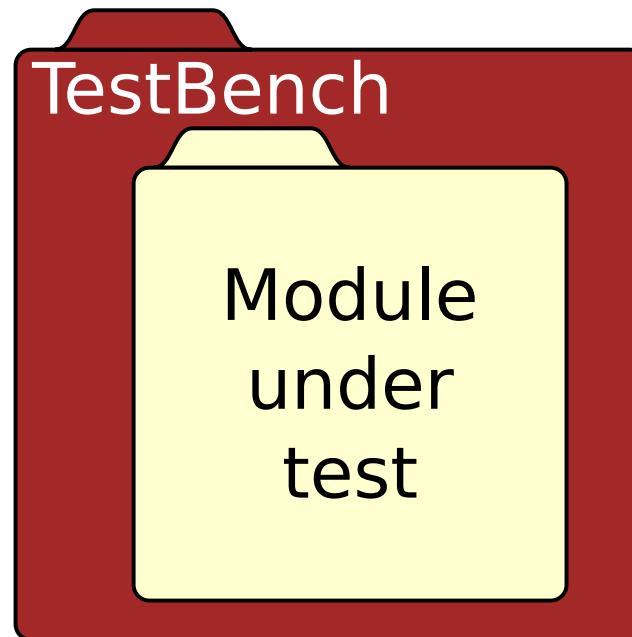
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

The magic is in how the files are connected

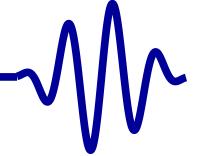
- If one interface is connected, the other ...
 - Should not have made any successful requests
 - Should not have received any acknowledgements



- Welcome
- Motivation
- Basics
- Clocked and \$past
- k* Induction
- Bus Properties
- Ex: WB Bus
- AXI
- Avalon
- Wishbone
- WB Basics
 - ▷ WB Basics
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Cover
- Sequences
- Quizzes



- Traditional test-bench file structure
- Doesn't work with yosys formal
- Why not?



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Ex: WB Bus

AXI

Avalon

Wishbone

WB Basics

▷ WB Basics

Free Variables

Abstraction

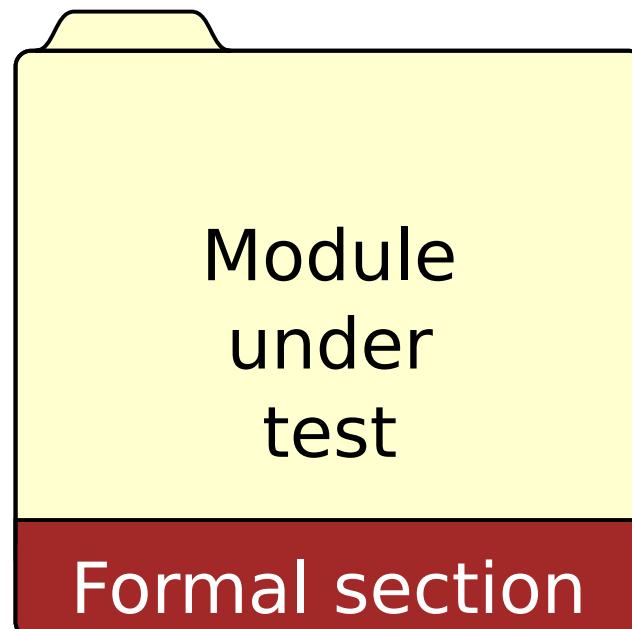
Invariants

Multiple-Clocks

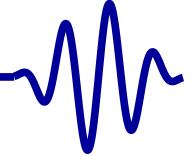
Cover

Sequences

Quizzes



- Formal Properties can be placed at the bottom
- This works well for testing some modules
- What's the limitation?



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Ex: WB Bus

AXI

Avalon

Wishbone

WB Basics

▷ WB Basics

Free Variables

Abstraction

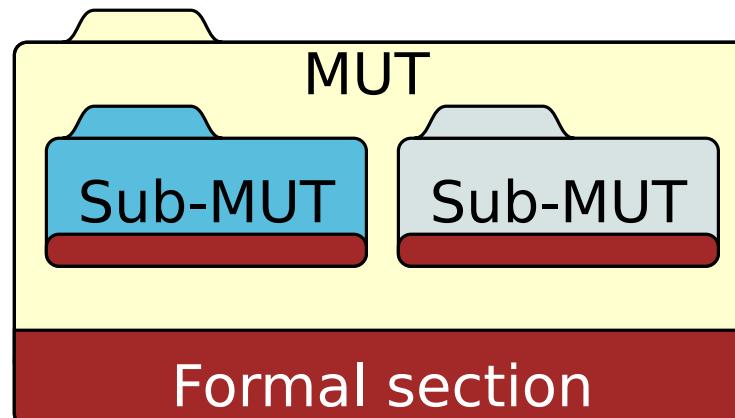
Invariants

Multiple-Clocks

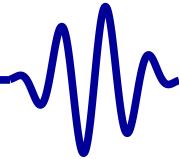
Cover

Sequences

Quizzes



- Design with multiple files
- They were each formally correct
- Problems?



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Ex: WB Bus

AXI

Avalon

Wishbone

WB Basics

▷ WB Basics

Free Variables

Abstraction

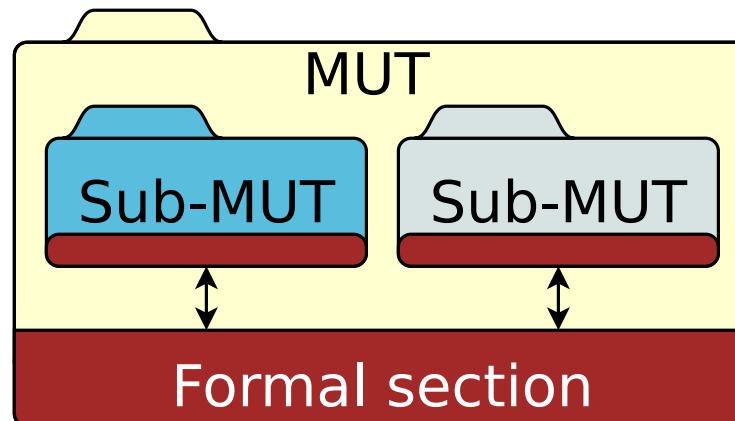
Invariants

Multiple-Clocks

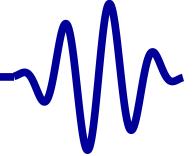
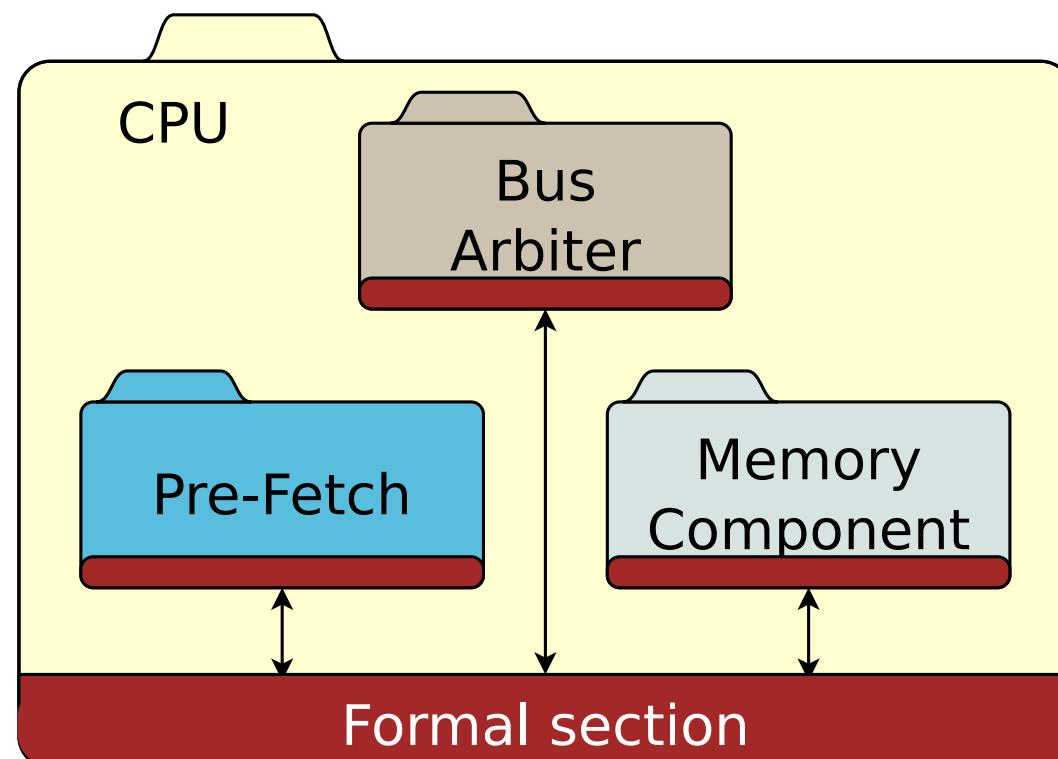
Cover

Sequences

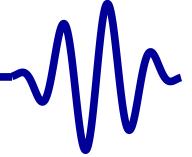
Quizzes



- Design with multiple files
- They were each formally correct
- Problems? Yes! In induction
- State variables needed to be formally synchronized (**assert()**)

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Proving properties for many components together can quickly get out of hand!



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

▷ Free Variables

Lesson Overview

Formal

Memory

So what?

Rule

Discussion

Abstraction

Invariants

Multiple-Clocks

Cover

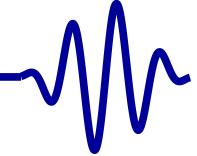
Sequences

Quizzes

Free Variables



Lesson Overview



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

▷ Lesson Overview

Formal
Memory

So what?

Rule
Discussion

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

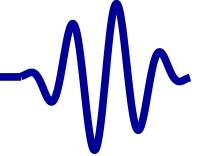
When dealing with memory, ...

- Testing the entire memory is not required
- Testing an arbitrary value is

It's time to discuss (* `anyconst` *) and (* `anyseq` *)
Objectives

- Understand what a free variable is
- Understand how (* `anyconst` *) and (* `anyseq` *) can be used to create free variables
- Learn how you can use free variables to validate memory and memory interfaces

any*



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Lesson Overview

▷ Formal

Memory

So what?

Rule

Discussion

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

- (* anyconst *)

```
(* anyconst *) wire [N-1:0] cval;
```

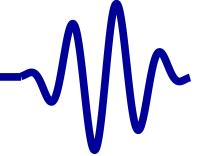
- Can be anything
- Defined at the beginning of time
- Never changed

- (* anyseq *)

```
(* anyseq *) wire [N-1:0] sval;
```

- Can change from one timestep to the next

Both can still be constrained via **assume()** statements

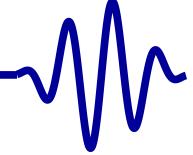
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Lesson Overview](#)[Formal](#)[▷ Memory](#)[So what?](#)[Rule](#)[Discussion](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

How might you build a memory with this?

```
(* anyconst *) wire [AW-1:0] f_const_addr;  
                    reg [AW-1:0] f_mem_value;
```

```
// Handle writes  
always @(posedge i_clk)  
if ((i_stb)&&(i_we)&&(i_addr == f_const_addr))  
    f_mem_value <= i_data;  
  
// Handle reads  
always @(posedge i_clk)  
if ((f_past_valid)&&($past(i_stb))&&(!$past(i_we))  
    &&($past(i_addr == f_const_addr)))  
    assert(o_data == f_mem_value);
```

GT So what?



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Lesson Overview

Formal

Memory

▷ So what?

Rule

Discussion

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Consider the specification of a prefetch

- The contract

```
(* anyconst *) wire [31:0] f_const_data;  
  
always @ (posedge i_clk)  
if ((o_valid)&&(o_pc == f_const_addr))  
    assert(o_insn == f_const_data);
```

- You'll also need to assume a bus input

```
always @ (posedge i_clk)  
if ((i_ack)&&(ackd_address == f_const_addr))  
    assume(i_data == f_const_data);
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Lesson Overview](#)[Formal](#)[Memory](#)[So what?](#)

▷ Rule
Discussion

[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

How would our general rule apply here?

- Assume inputs, assert internal state and outputs
- You could have written

```
input    wire  i_value;  
  
always @ (posedge i_clk)  
      assume(i_value == $past(i_value));
```

for the same effect as (* `anyconst` *)

- Both (* `anyconst` *) and (* `anyseq` *) act like inputs
- **assume()** them therefore, and not **assert()**

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Lesson Overview](#)[Formal](#)[Memory](#)[So what?](#)[▷ Rule](#)[Discussion](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

This works for a flash (or other ROM) controller too:

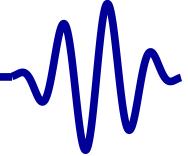
```
(* anyconst *) wire [AW-1:0] f_addr;
(* anyconst *) wire [31:0] f_data;

always @(*)
if ((o_wb_ack)&&(f_request_addr == f_addr))
    assert(o_wb_data == f_data);
```

Don't forget the corollary assumptions!

```
always @(*)
if (f_request_addr == f_addr)
    assume(i_spi_data
          == f_data[controller_state]);
```

... or something similar

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Lesson Overview](#)[Formal](#)[Memory](#)[So what?](#)[▷ Rule](#)[Discussion](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

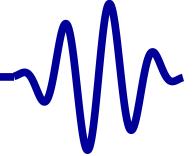
You can use this to build a serial port transmitter

```
(* anyseq *) wire f_tx_start;
(* anyseq *) wire [7:0] f_tx_data;
always @(*)
if (f_tx_busy)
    assume (!f_tx_start);

always @(posedge f_txclk)
if (f_tx_busy)
    assume(f_tx_data == $past(f_tx_data));
```

You can then

- Tie assertions to partially received data
- ... and pass induction



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Lesson Overview

Formal

Memory

So what?

Rule

▷ Discussion

Abstraction

Invariants

Multiple-Clocks

Cover

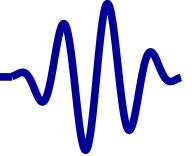
Sequences

Quizzes

How would you use free variables to verify a cache implementation?



Discussion



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Lesson Overview

Formal

Memory

So what?

Rule

▷ Discussion

Abstraction

Invariants

Multiple-Clocks

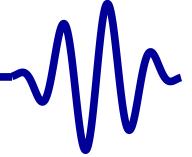
Cover

Sequences

Quizzes

How would you use free variables to verify a cache implementation?

Hint: you only need *three properties* for the cache contract



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

▷ Abstraction

Lesson Overview

Formal

Proof

Pictures

Examples

Exercise

Invariants

Multiple-Clocks

Cover

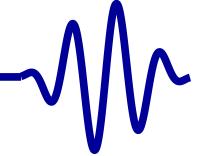
Sequences

Quizzes

Abstraction



Lesson Overview



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

▷ Lesson Overview

Formal

Proof

Pictures

Examples

Exercise

Invariants

Multiple-Clocks

Cover

Sequences

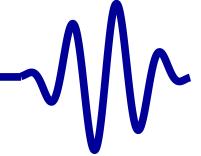
Quizzes

- Proving simple modules is easy.
- What about large and complex ones?

It's time to discuss *abstraction*.

Objectives

- Understand what abstraction is
- Gain confidence in the idea of abstraction
- Understand how to reduce a design via abstraction

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#) [\$k\$ Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[▷ Formal](#)[Proof](#)[Pictures](#)[Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Formally, if

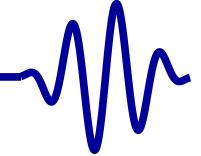
$$A \rightarrow C$$

then we can also say that

$$(AB) \rightarrow C$$



Formal Proof



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Lesson Overview

Formal

\triangleright Proof

Pictures

Examples

Exercise

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Shall we go over the proof?

$$A \rightarrow C \Rightarrow \neg A \vee C = \text{True}$$

True or anything is still true, so

$$(\neg A \vee C) \vee \neg B$$

Rearranging terms

$$\neg A \vee \neg B \vee C$$

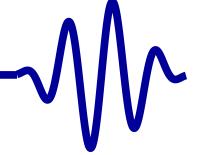
$$\neg (AB) \vee C$$

Expressing as an implication

$$(AB) \rightarrow C$$

Q.E.D.!

GT So what?



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Lesson Overview

Formal

▷ Proof

Pictures

Examples

Exercise

Invariants

Multiple-Clocks

Cover

Sequences

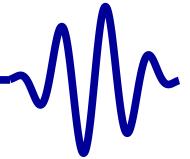
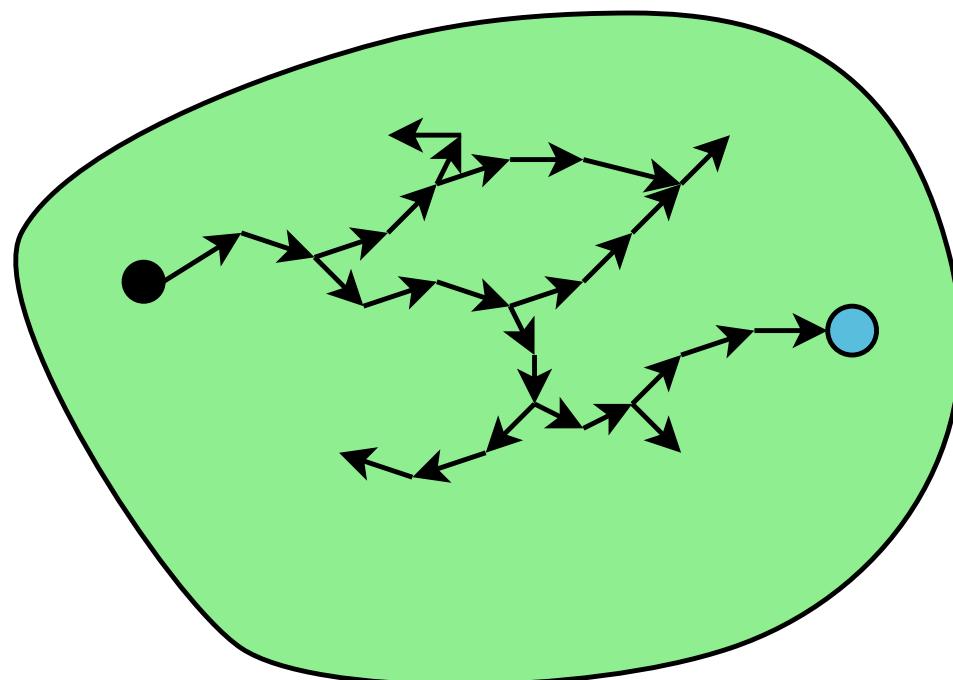
Quizzes

With every additional module,

- Formal verification becomes more difficult
- Complexity increases exponentially
- You only have so many hours and dollars

On the other hand,

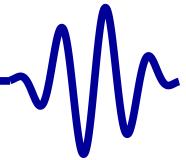
- Anything you can simplify by abstraction . . .
- is one less thing you need to prove

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[▷ Pictures](#)[Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Suppose your state space looked like this

- It takes many transitions required to get to interesting states

GT In Pictures



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Lesson Overview

Formal

Proof

▷ Pictures

Examples

Exercise

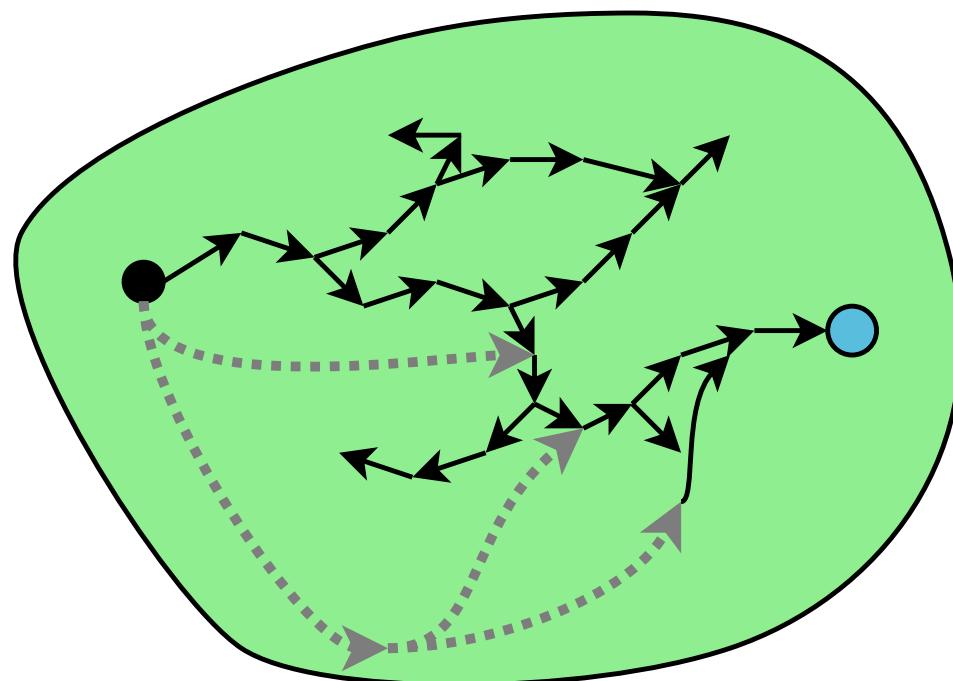
Invariants

Multiple-Clocks

Cover

Sequences

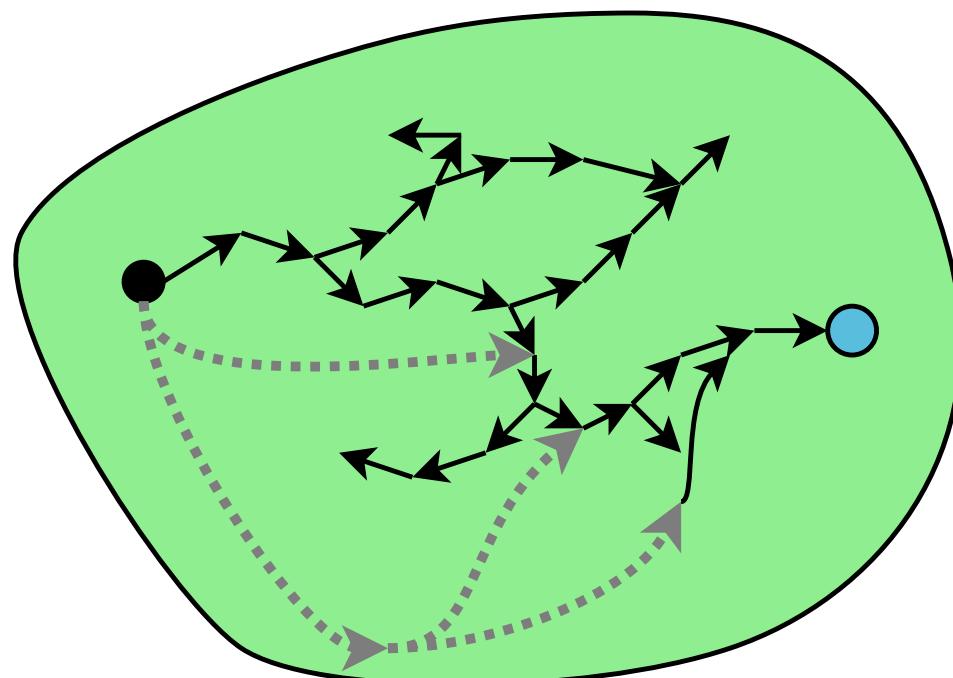
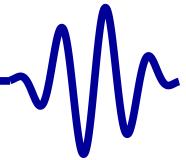
Quizzes



Suppose we added to this design ...

- Some additional states, and
- Additional transitions

The *real* states and transitions must still remain

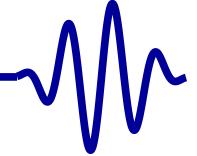


If this new design still passes, then . . .

- Since the original design is a subset . . .
 - The original design must also still pass

If done well, the new design will require less effort to prove

GT A CPU



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Lesson Overview

Formal

Proof

▷ Pictures

Examples

Exercise

Invariants

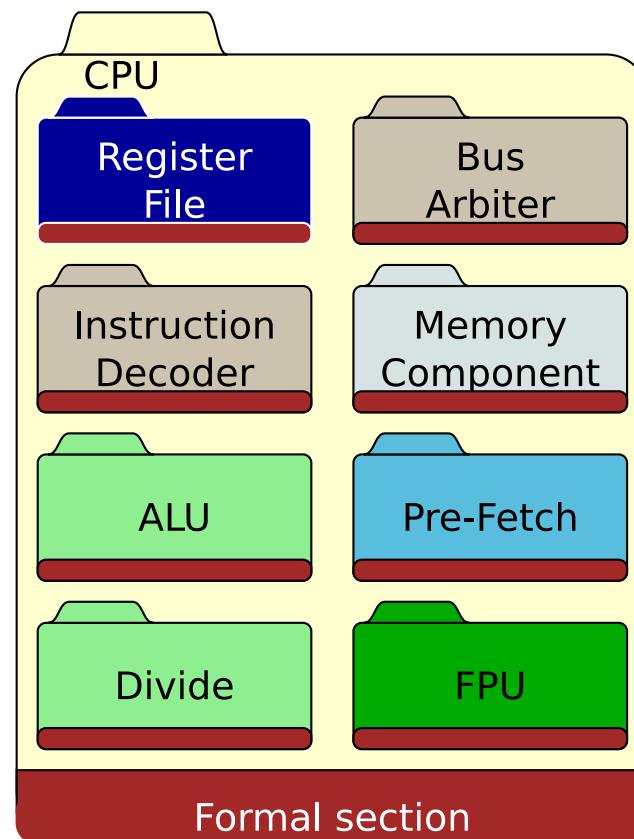
Multiple-Clocks

Cover

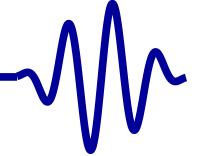
Sequences

Quizzes

Where would you start?



GT A CPU



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Lesson Overview

Formal

Proof

▷ Pictures

Examples

Exercise

Invariants

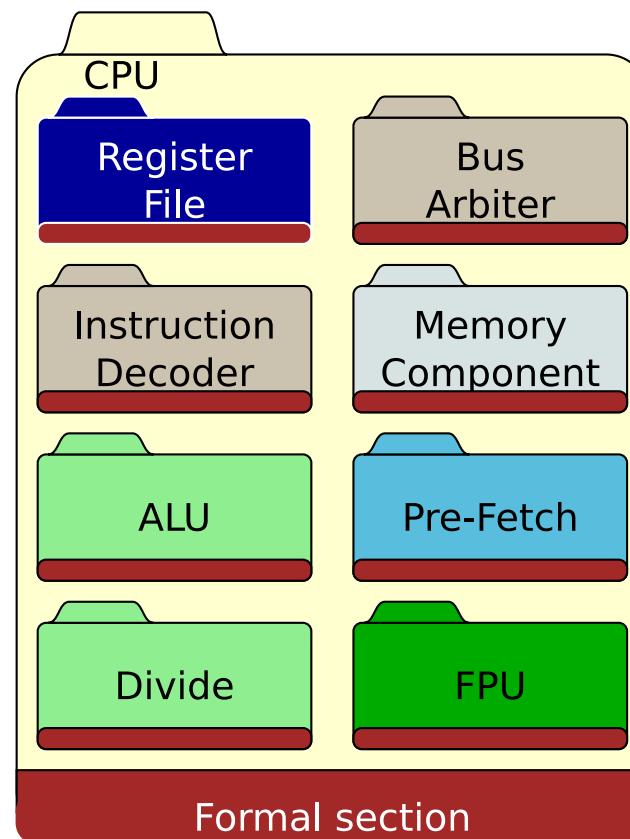
Multiple-Clocks

Cover

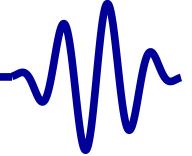
Sequences

Quizzes

Where would you start?



At the interfaces!



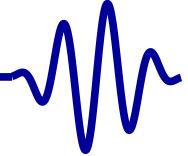
Let's consider a prefetch module as an example.



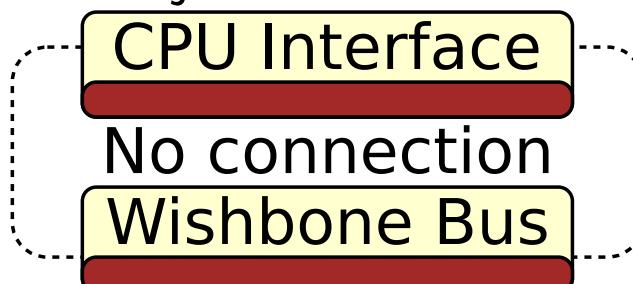
If you do this right,

- Any internally consistent Prefetch,
- that properly responds to the CPU, *and*
- interacts properly with the bus,
- must work!

Care to try a different prefetch approach?

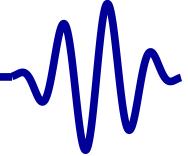
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[▷ Pictures](#)[Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Suppose the prefetch was just a shell

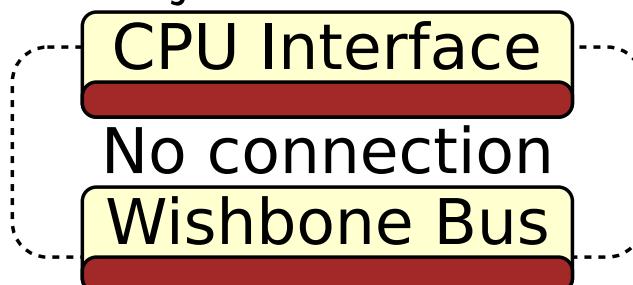


It would still interact properly with

- The bus, and
- The CPU
- It just might not return values from the bus to the CPU

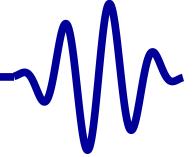
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[▷ Pictures](#)[Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Suppose the prefetch was just a shell



If the CPU still acted “correctly”

- With either the right, or the wrong instructions, then
- The CPU *must act correctly with the right instructions*



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Lesson Overview

Formal

Proof

Pictures

▷ Examples

Exercise

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

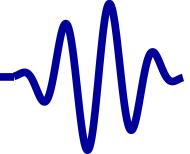
Consider these statements:

□

If
And
Then



Examples



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Lesson Overview

Formal

Proof

Pictures

▷ Examples

Exercise

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Consider these statements:

- Prefetch is bus master, interfaces w/CPU

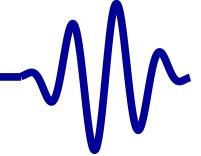
If (Prefetch responds to CPU insn requests)

And (Prefetch produces the right instructions)

Then (The prefetch works within the design)



Examples



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Lesson Overview

Formal

Proof

Pictures

▷ Examples

Exercise

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

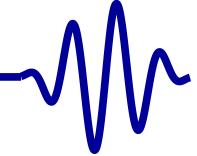
Consider these statements:

- The CPU is just a wishbone master within a design

If (The CPU is valid bus master)

And (CPU properly executes instructions)

Then (CPU works within a design)

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[Pictures](#)[▷ Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

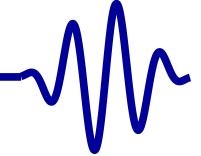
Consider these statements:

- The ALU must return a calculated number

If (ALU returns a value when requested)

And (It is the right value)

Then (The ALU works within the design)

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[Pictures](#)[▷ Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

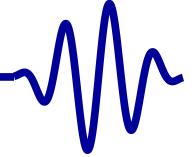
Consider these statements:

- A flash device responds in 8-80 clocks

If (Bus master reads/responds to a request)

And (The response comes back in 8-80 clocks)

Then (The CPU can interact with a flash memory)

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[Pictures](#)[▷ Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Consider these statements:

- The divide must return a calculated number

If (Divide returns a value when requested)

And (It is the right value)

Then (The divide works within the design)

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[Pictures](#)[▷ Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

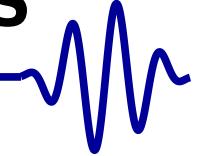
Consider these statements:

- Formal solvers break down when applied to multiplies

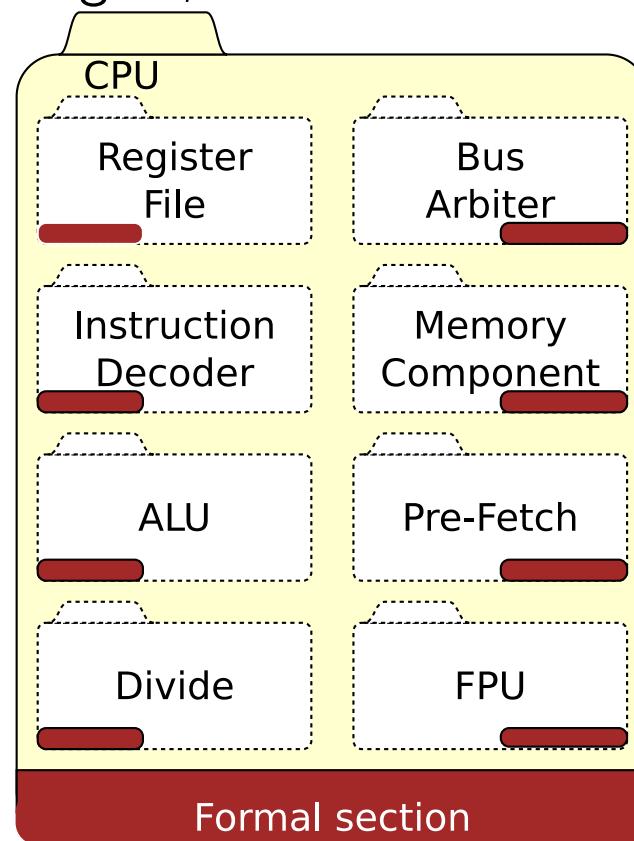
If (Multiply unit returns an answer N clocks later)

And (It is the right value)

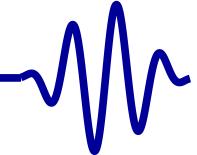
Then (The multiply works within the design)

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[Pictures](#)[▷ Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Looking at the CPU again,



- Replace all the components with abstract shells
- ... shells that *might* produce the same answers

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[Pictures](#)[▷ Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

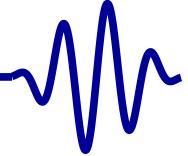
Let's consider a fractional counter:

```
reg      [31:0]  r_count;
initial r_count = 0;
initial o_pps = 0;
always @(posedge i_clk)
    { o_pps, r_count } <= r_count + 32'd43;
```

The problem with this counter

- It will take 100×10^6 clocks to roll over and set o_pps
- Formally checking 100×10^6 clocks is prohibitive

We'll need a better way, or we'll never deal with this

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[Pictures](#)[▷ Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

How might we build an abstract counter?

- First, create an arbitrary counter increment

```
(* anyseq *) wire [31:0]           increment;
assign rollover = - r_count;
always @(*)
begin
    assume(increment > 0);
    assume(increment < { 2'h1, 30'h0 });
    if (rollover < 32'd43)
        assume(increment == 32'd43);
    else
        assume(increment < rollover);
end
```

The correct increment, 32'd43, must be a possibility

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[Pictures](#)[▷ Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

We can now increment our counter by this arbitrary increment

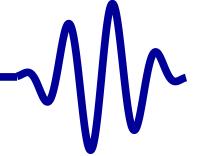
```
always @ (posedge i_clk)
    { o_pps, r_count } <= r_count + increment;
```

Will this work?

- Let's try this to see!

```
always @ (posedge i_clk)
if (f_past_valid)
    assert (r_count != $past(r_count));
```

```
always @ (posedge i_clk)
if ((f_past_valid)&&(r_count < $past(r_count)))
    assert (o_pps);
```



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Lesson Overview

Formal

Proof

Pictures

▷ Examples

Exercise

Invariants

Multiple-Clocks

Cover

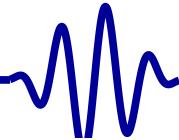
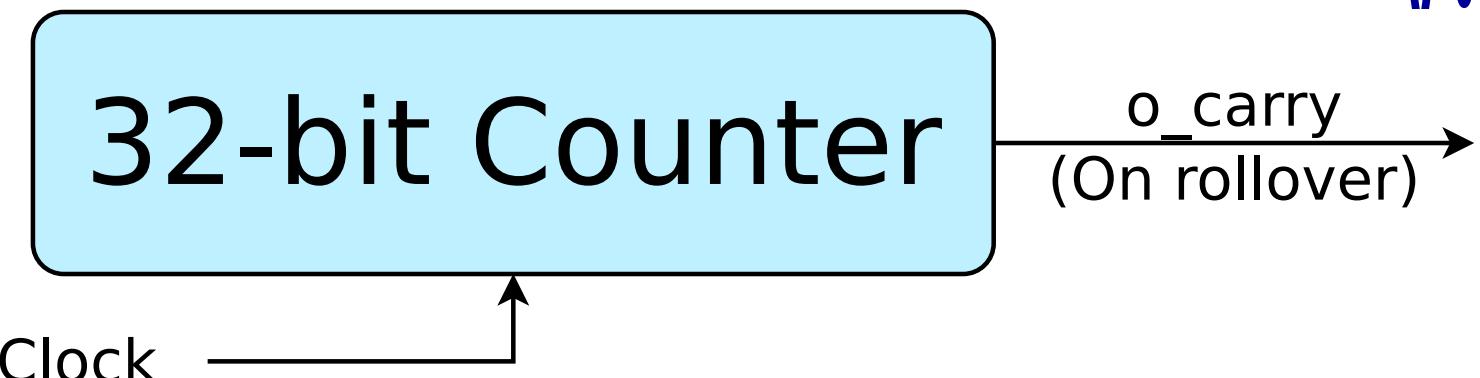
Sequences

Quizzes

How else might you use this?

- Bypassing the runup for an external peripheral
- Testing a real-time clock or date

Or . . . how about that CPU?

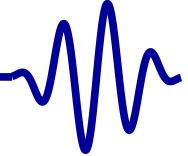
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[Pictures](#)[Examples](#)[▷ Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Let's modify this abstract counter

- Increment by one, rather than fractionally

Exercise Objectives:

- Prove a design works both with and without abstraction
- Gain some confidence using abstraction

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[Pictures](#)[Examples](#)[▷ Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Your task:

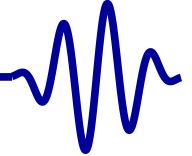
- Rebuild the counter
- Make it increment by one
- Build it so that ...

```
always @(*)  
    assert(o_carry == (r_count == 0));
```

// and

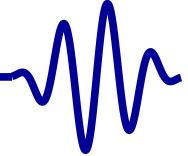
```
always @(posedge i_clk)  
    if ((f_past_valid)&&(!$past(&r_count)))  
        assert(!o_carry);
```

- Prove that this abstracted counter works

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[Pictures](#)[Examples](#)[▷ Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Your task:

- Rebuild the counter
- Make it increment by one
- *Prove that this abstracted counter works*

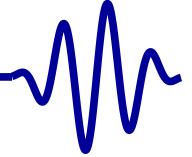
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal Proof](#)[Pictures](#)[Examples](#)[▷ Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Your task:

- Rebuild the counter
- Make it increment by one
- *Prove that this abstracted counter works*

Hints:

- `&r_count` must take place before `r_count==0`
- You cannot skip `&r_count`
- Neither can you skip `r_count == 0`



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

▷ Invariants

Lesson Removed

Multiple-Clocks

Cover

Sequences

Quizzes

Invariants



Lesson Removed



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

▷ Lesson Removed

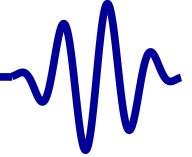
Multiple-Clocks

Cover

Sequences

Quizzes

This lesson is currently being revised, and will be released again shortly



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

▷ Multiple-Clocks

Basics

SBY File

\$global_clock

\$rose

\$stable

Examples

Exercises

Cover

Sequences

Quizzes

Multiple-Clocks

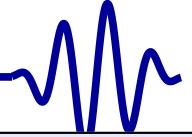
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[▷ Basics](#)[SBY File](#)[\\$global_clock](#)[\\$rose](#)[\\$stable](#)[Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

The SymbiYosys option `multiclock` . . .

- Used to process systems with dissimilar clocks
- Examples
 - A serial port, with a formally generated transmitter coming from a different clock domain
 - A SPI controller that needs both high speed and low speed logic

Our Objective:

- To learn how to handle multiple clocks within a design
 - **\$global_clock**
 - **\$stable, \$changed**
 - **\$rose, \$fell**

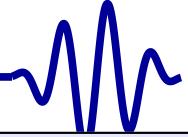
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[▷ SBY File](#)[\\$global_clock](#)[\\$rose](#)[\\$stable](#)[Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

```
[options]
mode prove
multiclock on

[engines]
smtbmc

[script]
read -formal module.v
prep -top module

[files]
# file list
```



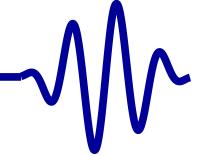
Welcome
Motivation
Basics
Clocked and \$past
 k Induction
Bus Properties
Free Variables
Abstraction
Invariants
Multiple-Clocks
Basics
▷ SBY File
\$global_clock
\$rose
\$stable
Examples
Exercises
Cover
Sequences
Quizzes

```
[options]
mode prove
multiclock on ← Multiple clocks require this line

[engines]
smtbmc

[script]
read -formal module.v
prep -top module

[files]
# file list
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[▷ SBY File](#)[\\$global_clock](#)[\\$rose](#)[\\$stable](#)[Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

- **\$global_clock**

A global simulation clock, updated on each time-step

- **\$stable**

True if a signal is stable (i.e. doesn't change) with this clock.

Equivalent to $A == \$past(A)$

- **\$changed**

True if a signal has changed since the last clock tick.

Equivalent to $A != \$past(A)$

- **\$rose**

True if the signal rises on this simulation step

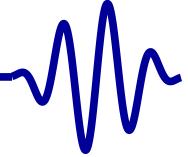
This is very useful for positive edged clocks transitions

\$rose(A) is equivalent to $(A[0]) \&\& (!\$past(A[0]))$

- **\$fell**

True if a signal falls on this simulation step, creating a negative edge

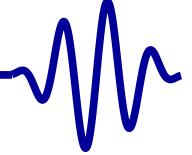
\$fell(A) is equivalent to $(!A[0]) \&\& (\$past(A[0]))$

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[▷ \\$global_clock](#)[\\$rose](#)[\\$stable](#)[Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

- A global simulation clock, updated on each time-step
- You can use this to describe clock properties

```
// Assume a single clock signal
//
reg      f_last_clk;

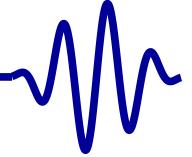
initial f_last_clk = 0;
always @($global_clock)
begin
    f_last_clk <= !f_last_clk;
    assume(i_clk == f_last_clk);
end
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[▷ \\$global_clock](#)[\\$rose](#)[\\$stable](#)[Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

- A global simulation clock, updated on each time-step
- You can use this to describe clock properties

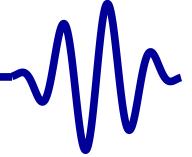
```
// Assume two related clock signals
//
reg [3:0] f_clk_counter;

initial f_clk_counter = 0;
always @($global_clock)
begin
    f_clk_counter <= f_clk_counter + 1'b1;
    assume(i_clk_fast == f_clk_counter[0]);
    assume(i_clk_slow == f_clk_counter[3]);
end
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[▷ \\$global_clock](#)[\\$rose](#)[\\$stable](#)[Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

- A global simulation clock, updated on each time-step
- You can use this to describe clock properties

```
// Assume two clocks, same speed,  
// unknown constant phase offset  
// ...  
(* anyconst *) wire [3:0] f_clk_offset;  
  
initial f_clk_counter= 0;  
always @($global_clock)  
begin  
    f_clk_counter <= f_clk_counter + 1'b1;  
    f_clk_two <= f_clk_counter  
        + f_clk_offset;  
    assume(i_clk_one == f_clk_counter[3]);  
    assume(i_clk_two == f_clk_two[3]);  
end
```



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

▷ \$global_clock

\$rose

\$stable

Examples

Exercises

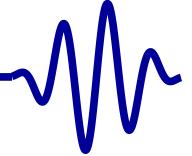
Cover

Sequences

Quizzes

How might you describe two unrelated clocks?

\$global_clock



How might you describe two unrelated clocks?

```
(* anyconst *) wire [7:0] f_a_step;  
always @(*)  
assume((f_a_step > 0)  
      &&(f_a_step[7] == 1'b0));  
  
always @($global_clock)  
begin  
    f_a_counter <= f_a_counter + f_a_step;  
  
    assume(i_clk_a == f_a_counter[7]);  
end
```

- The `(* anyconst *)` will take on any arbitrary, but constant value
- You can repeat this logic for the second clock.

Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

▷ \$global_clock

\$rose

\$stable

Examples

Exercises

Cover

Sequences

Quizzes



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

\$global_clock

▷ \$rose

\$stable

Examples

Exercises

Cover

Sequences

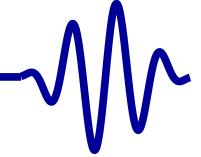
Quizzes

Synchronous logic has some requirements

- Inputs should *only* change on a clock edge
They should be stable otherwise
- **\$rose(i_clk)** can be used to express this

Here's an example using **\$rose(i_clk)** . . .

```
always @($global_clock)
if (! $rose(i_clk))
    assume(i_input == $past(i_input));
```



Would this work?

```
always @($global_clock)
if (! $rose(i_clk))
    assert(i_input == $past(i_input));
```

Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

\$global_clock

▷ \$rose

\$stable

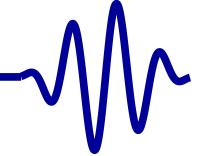
Examples

Exercises

Cover

Sequences

Quizzes



Would this work?

```
always @($global_clock)
if (! $rose(i_clk))
    assert(i_input == $past(i_input));
```

- No. The *general rule* hasn't changed

Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

\$global_clock

▷ \$rose

\$stable

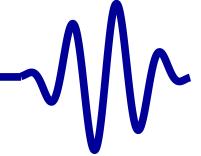
Examples

Exercises

Cover

Sequences

Quizzes



Could we do it this way?

```
always @($global_clock)
if ($fell(i_clk))
    assert(state == $past(state));
```

Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

\$global_clock

▷ \$rose

\$stable

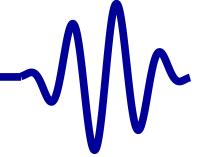
Examples

Exercises

Cover

Sequences

Quizzes



Could we do it this way?

```
always @($global_clock)
if ($fell(i_clk))
    assert(state == $past(state));
```

- No, this doesn't work either

Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

\$global_clock

▷ \$rose

\$stable

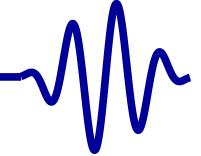
Examples

Exercises

Cover

Sequences

Quizzes



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

\$global_clock

▷ \$rose

\$stable

Examples

Exercises

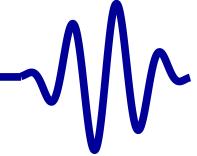
Cover

Sequences

Quizzes

Is this equivalent?

```
always @($global_clock)
if (! $past(i_clk))
    assert(state == $past(state));
```



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

\$global_clock

▷ \$rose

\$stable

Examples

Exercises

Cover

Sequences

Quizzes

Is this equivalent?

```
always @($global_clock)
if (! $past(i_clk))
    assert(state == $past(state));
```

- Why not?

\$rose



This fixes our problems. Will this work?

```
always @($global_clock)
if (! $rose(i_clk))
    assert(state == $past(state));
```

Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

\$global_clock

▷ \$rose

\$stable

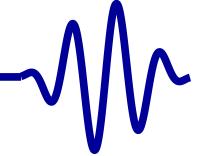
Examples

Exercises

Cover

Sequences

Quizzes



This fixes our problems. Will this work?

```
always @($global_clock)
if (! $rose(i_clk))
    assert(state == $past(state));
```

- Not quite. Can you see the problem?

Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

\$global_clock

▷ \$rose

\$stable

Examples

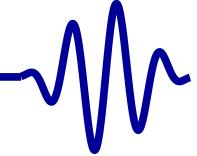
Exercises

Cover

Sequences

Quizzes

\$rose



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

\$global_clock

▷ \$rose

\$stable

Examples

Exercises

Cover

Sequences

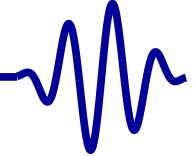
Quizzes

- State/outputs should be clock synchronous

```
always @($global_clock)
  if ((f_past_valid)&&(!$rose(i_clk))
      assert(state == $past(state));
```

- With f_past_valid this works
- \$rose requires a clock, such as
always @(\$global_clock)

\$stable



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

\$global_clock

\$rose

▷ \$stable

Examples

Exercises

Cover

Sequences

Quizzes

Describes a signal which has not changed

- Requires a clock edge

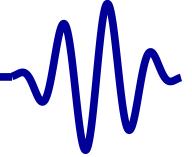
```
always @($global_clock)
```

```
always @(posedge i_clk)
```

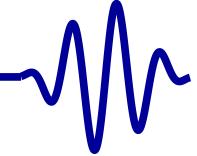
- *Caution:* The value might still change between clock edges

```
always @($global_clock)
if ((f_past_valid)&&(!$rose(i_clk)))
    assert ($stable(state));
```

- This is basically the same as state == \$past(state)

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\\$global_clock](#)[\\$rose](#)[▷ \\$stable](#)[Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

\$fell is like **\$rose**, only it describes a negative edge

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\\$global_clock](#)[\\$rose](#)[\\$stable](#)[▷ Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

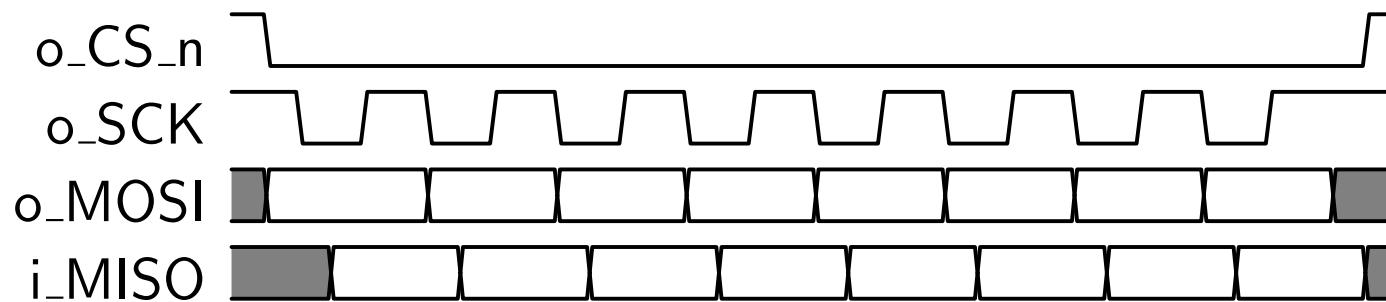
- Most logic doesn't need the multiclock option
- To help with logic that might need it, I use a parameter

```
parameter [0:0] F_OPT_CLK2FFLOGIC = 1'b0;

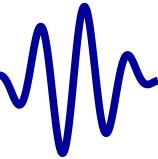
generate if (F_OPT_CLK2FFLOGIC)
begin
    always @($global_clock)
        if ((f_past_valid)&&(!$rose(i_clk)))
            begin
                assume($stable(i_axi_awready));
                assume($stable(i_axi_wready));
                //
                assert($stable(o_axi_bid));
                //
                ...
            end
end endgenerate
```



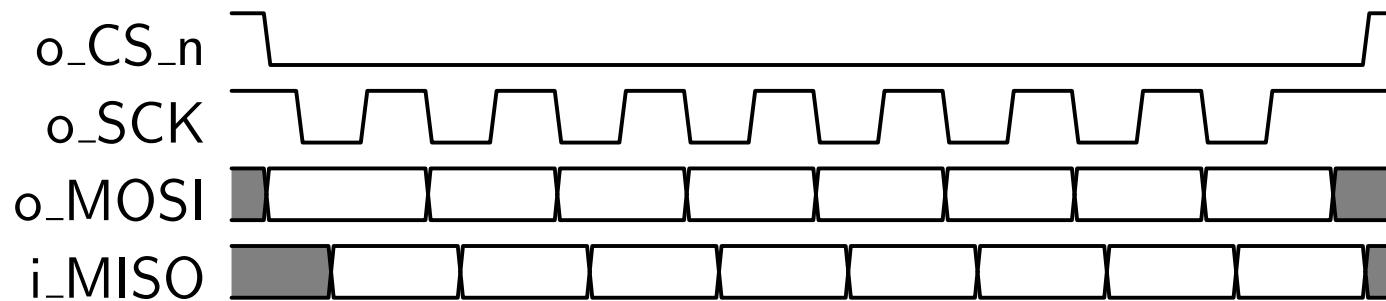
- [Welcome](#)
- [Motivation](#)
- [Basics](#)
- [Clocked and \\$past](#)
- [k Induction](#)
- [Bus Properties](#)
- [Free Variables](#)
- [Abstraction](#)
- [Invariants](#)
- [Multiple-Clocks](#)
- [Basics](#)
- [SBY File](#)
- [\\$global_clock](#)
- [\\$rose](#)
- [\\$stable](#)
- ▷ Examples
- [Exercises](#)
- [Cover](#)
- [Sequences](#)
- [Quizzes](#)



- How would you formally describe the o_SCK and o_CS_n relationship?



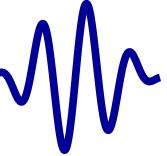
- Welcome
- Motivation
- Basics
- Clocked and \$past
- k Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Basics
- SBY File
- \$global_clock
- \$rose
- \$stable
- ▷ Examples
- Exercises
- Cover
- Sequences
- Quizzes



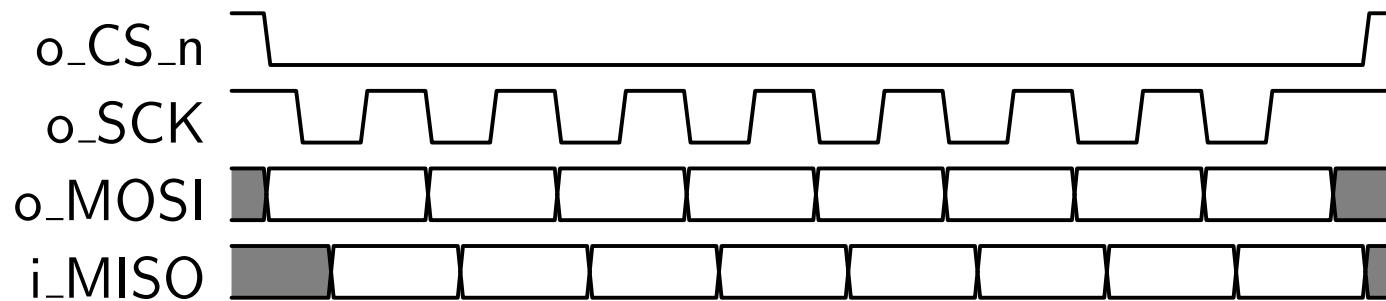
- How would you formally describe the o_SCK and o_CS_n relationship?

```
initial assert(o_CS_n);
initial assert(o_SCK);

always @(*)
if (!o_SCK)
    assert(!o_CS_n);
```



- Welcome
- Motivation
- Basics
- Clocked and \$past
- k Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Basics
- SBY File
- \$global_clock
- \$rose
- \$stable
- ▷ Examples
- Exercises
- Cover
- Sequences
- Quizzes

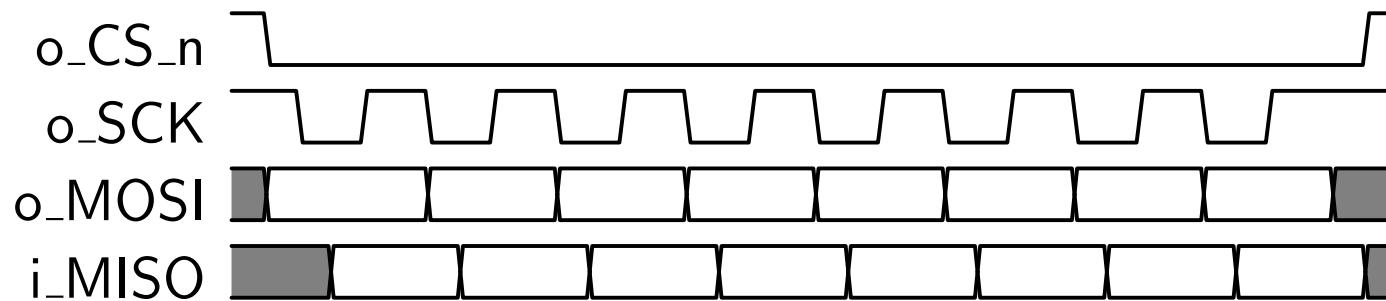


- How would you formally describe the o_SCK and o_CS_n relationship?

```
always @($global_clock)
if ((f_past_valid)
    &&($rose(o_CS_n))||($fell(o_CS_n))))
    assert ((o_SCK)&&($stable(o_SCK)));
```

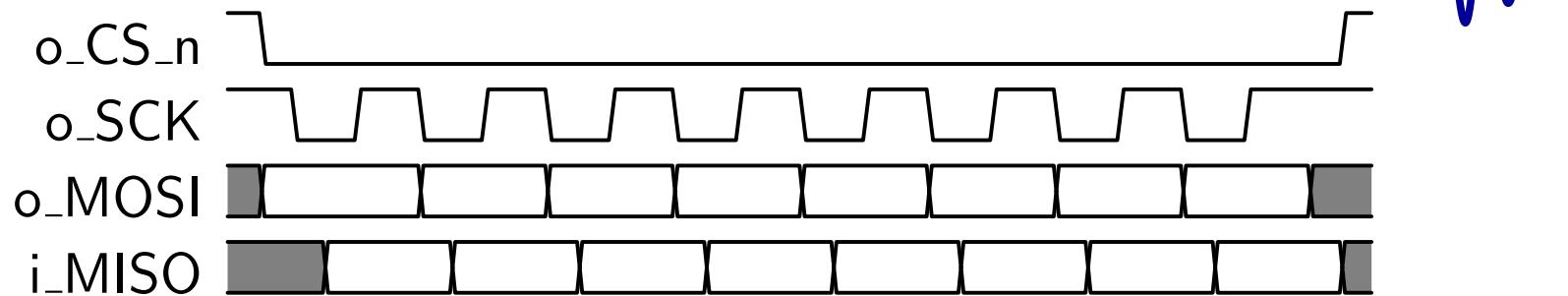


- Welcome
- Motivation
- Basics
- Clocked and \$past
- k Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Basics
- SBY File
- \$global_clock
- \$rose
- \$stable
- ▷ Examples
- Exercises
- Cover
- Sequences
- Quizzes



- How would you describe o_MOSI?

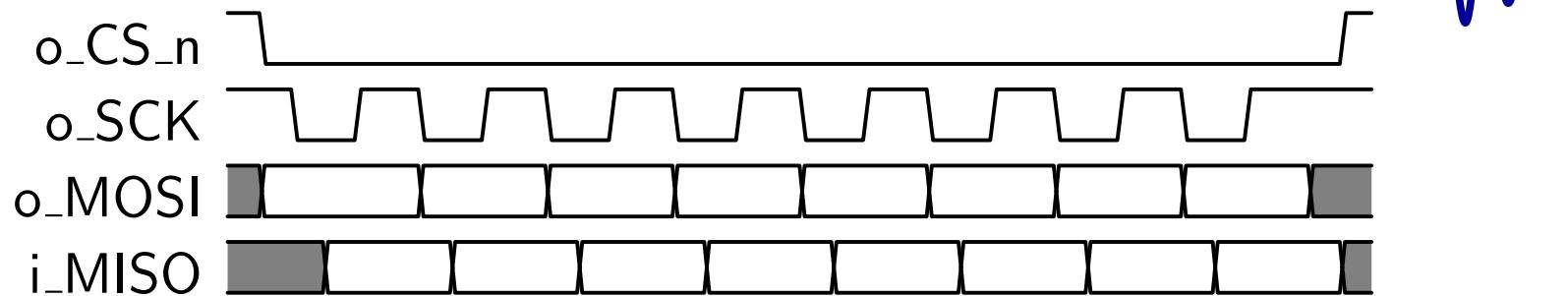
- Welcome
- Motivation
- Basics
- Clocked and \$past
- k Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Basics
- SBY File
- \$global_clock
- \$rose
- \$stable
- ▷ Examples
- Exercises
- Cover
- Sequences
- Quizzes



- How would you describe o_MOSI?

```
always @($global_clock)
if ((f_past_valid)&&(!o_CS_n)&&(!$fell(o_SCK)))
    assert($stable(o_MOSI));
```

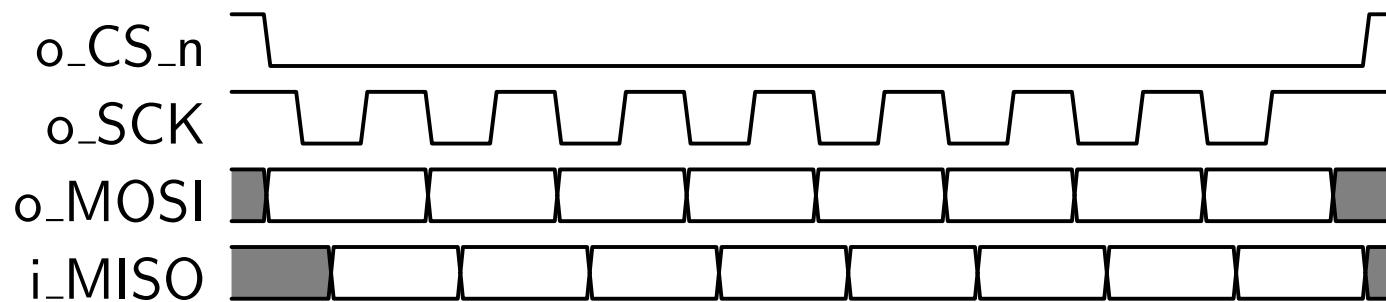
- Welcome
- Motivation
- Basics
- Clocked and \$past
- k Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Basics
- SBY File
- \$global_clock
- \$rose
- \$stable
- ▷ Examples
- Exercises
- Cover
- Sequences
- Quizzes



- How would you describe `i_MISO`?



- Welcome
- Motivation
- Basics
- Clocked and \$past
- k Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Basics
- SBY File
- \$global_clock
- \$rose
- \$stable
- ▷ Examples
- Exercises
- Cover
- Sequences
- Quizzes

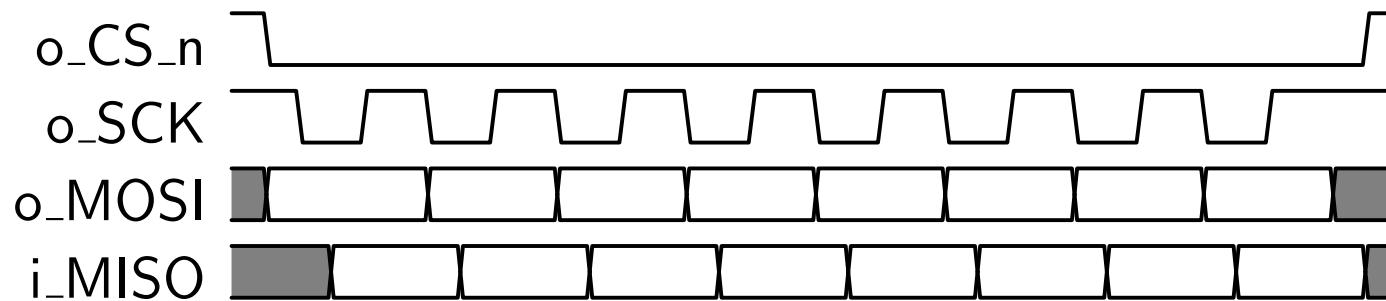


- How would you describe i_MISO?

```
always @($global_clock)
if ((!o_CS_n)&&(o_SCK))
    assume ($stable(i_MISO));
```



- Welcome
- Motivation
- Basics
- Clocked and \$past
- k Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
 - Basics
 - SBY File
 - \$global_clock
 - \$rose
 - \$stable
 - ▷ Examples
 - Exercises
- Cover
- Sequences
- Quizzes



- Should the i_MISO be able to change more than once per clock?

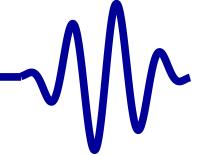
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\\$global_clock](#)[\\$rose](#)[\\$stable](#)[▷ Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

- A little logic will force `i_MISO` to have only one transition per clock

```
always @($global_clock)
  if ((o_CS_n) || (o_SCK))
    f_chgd <= 1'b0;
  else if (i_MISO != $past(i_MISO))
    f_chgd <= 1'b1;
```

```
always @($global_clock)
  if ((f_past_valid)&&(f_chgd))
    assume ($stable(i_MISO));
```

- How would we force exactly 8 `o_SCK` clocks?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\\$global_clock](#)[\\$rose](#)[\\$stable](#)[▷ Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

- Forcing exactly 8 clocks

```
always @($global_clock)
if (o_CS_n)
    f_spi_bits <= 0;
else if ($rose(o_SCK))
    f_spi_bits <= f_spi_bits + 1'b1;
```

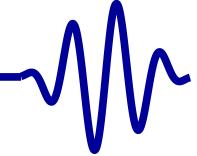
```
always @($global_clock)
if ((f_past_valid)&&($rose(o_CS_n)))
    assert(f_spi_bits == 8);
```

- Don't forget the induction requirement

```
always @(*)
assert(f_spi_bits <= 8);
```



Exercises



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

\$global_clock

\$rose

\$stable

Examples

▷ Exercises

Cover

Sequences

Quizzes

Three exercises, chose one to verify:

1. Input serdes

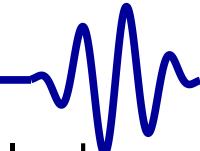
`exercises-09/iserdes.v`

2. Clock gate

`exercises-10/clkgate.v`

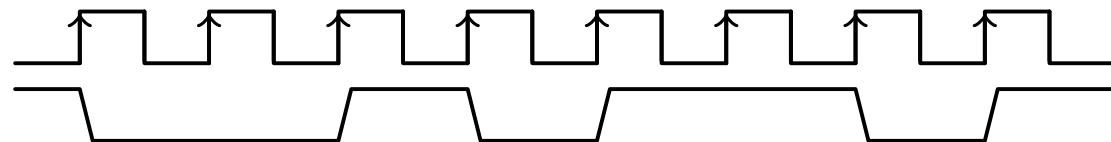
3. Clock Switch

`exercises-11/clkswitch.v`

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\\$global_clock](#)[\\$rose](#)[\\$stable](#)[Examples](#)[▷ Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

Getting a SERDES right is a good example of multiple clocks

i_fast_clk



i_pin

i_slow_clk

o_word

0x0b

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\\$global_clock](#)[\\$rose](#)[\\$stable](#)[Examples](#)[▷ Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

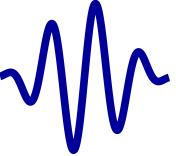
Getting a SERDES right is a good example of multiple clocks

- Two clocks, one fast and one slow

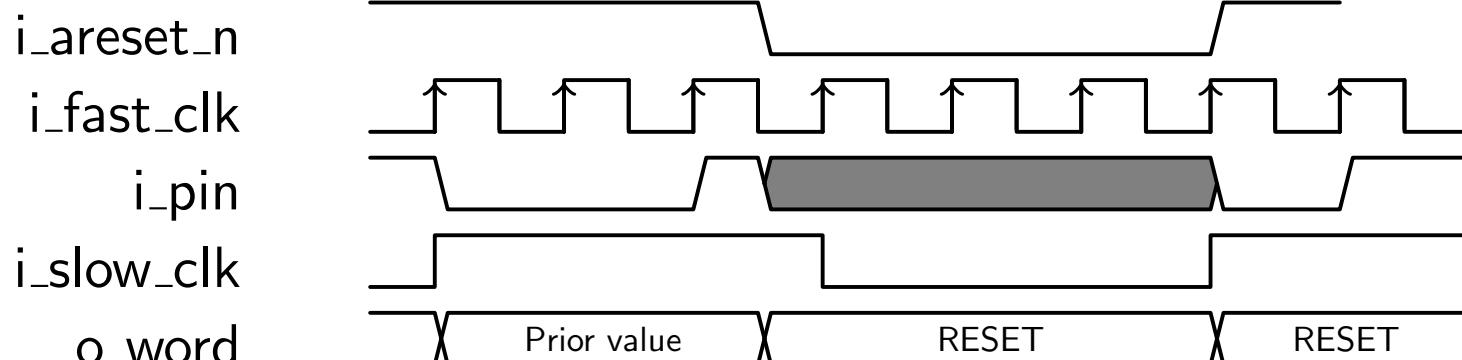
Clocks must be synchronous

\$rose(slow_clk) implies **\$rose**(fast_clk)

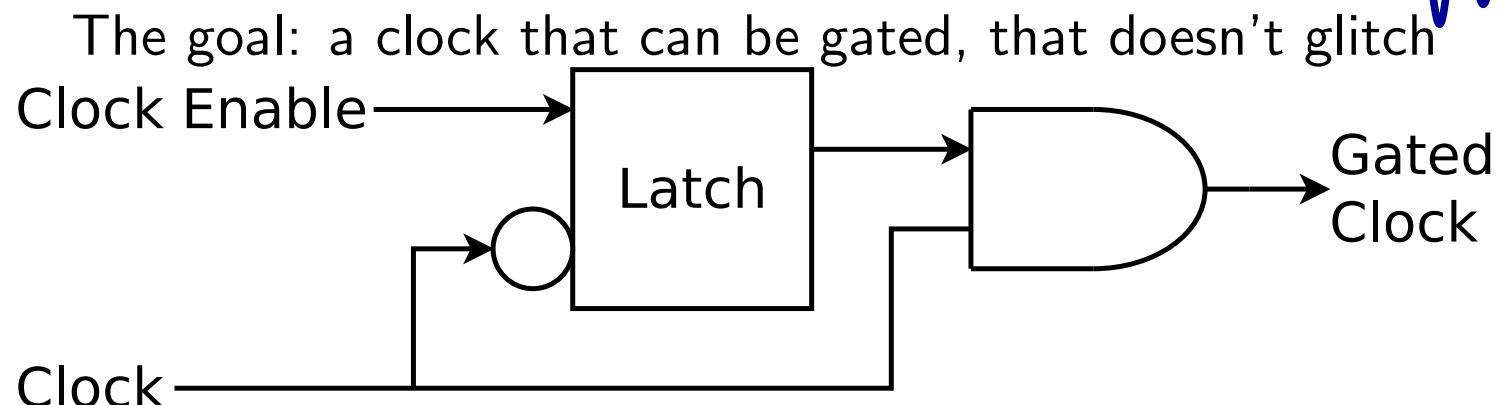
- exercise-09/ Contains the file `iserdes.v`
- Can you formally verify that it works?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\\$global_clock](#)[\\$rose](#)[\\$stable](#)[Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

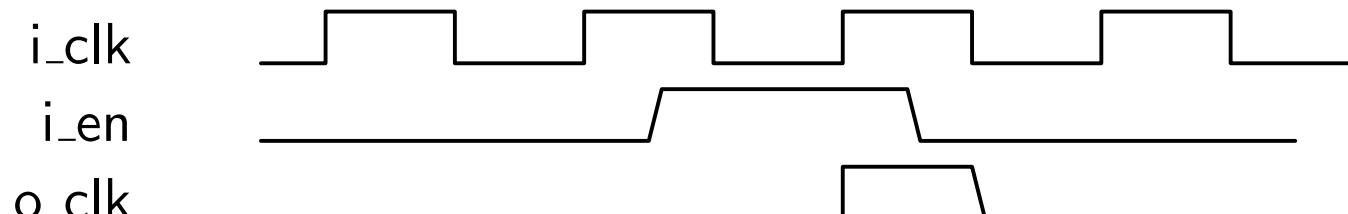
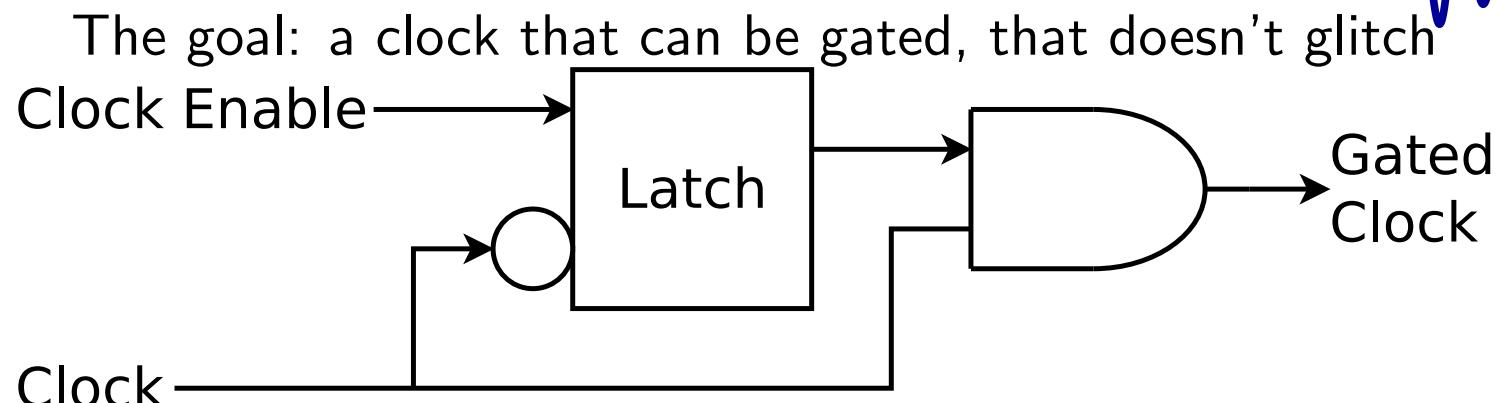
Be aware of the asynchronous reset signal!



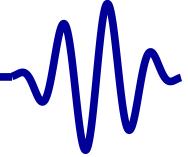
- Can be asserted at any time
- Can only be de-asserted on **\$rose(i_slow_clk)**
- **assume()** these properties, since the reset is an input

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\\$global_clock](#)[\\$rose](#)[\\$stable](#)[Examples](#)[▷ Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

- exercise-10/ Contains the file clkgate.v

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\\$global_clock](#)[\\$rose](#)[\\$stable](#)[Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

Ex: Clock Gate



The goal: a clock that can be gated, that doesn't glitch

- One clock, one unrelated enable
- Prove that the output clock
 - is always high for the full width, but
 - . . . never longer.
 - For any clock rate

See `exercise-10/clkgate.v`

Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

\$global_clock

\$rose

\$stable

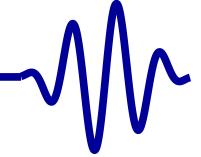
Examples

▷ Exercises

Cover

Sequences

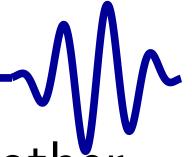
Quizzes

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\\$global_clock](#)[\\$rose](#)[\\$stable](#)[Examples](#)[▷ Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

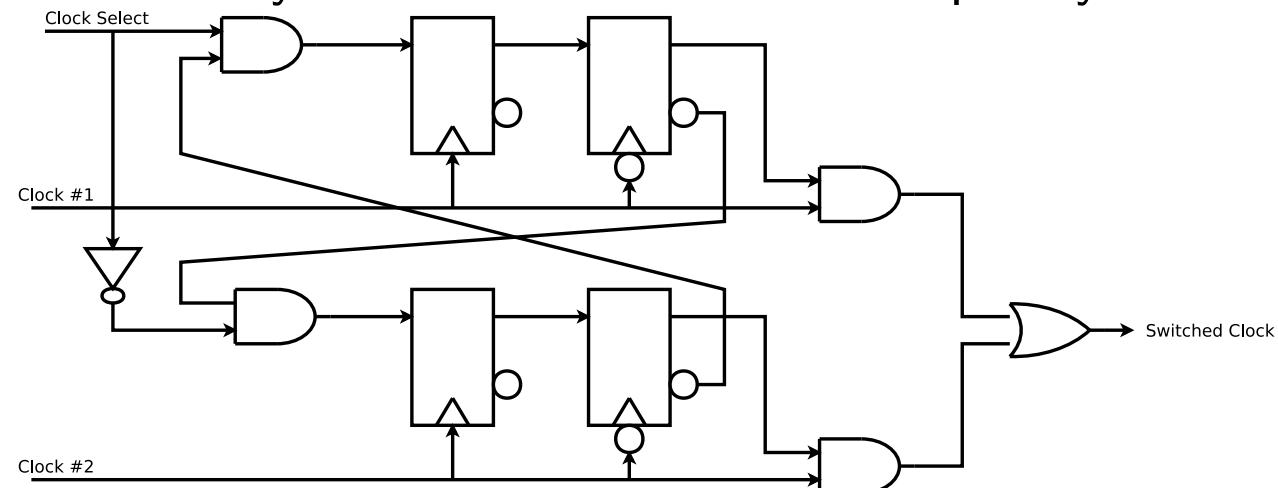
Hints:

- The output clock should only rise if the incoming clock rises
- The output clock should only fall if the incoming clock fall
- If the output clock is ever high, it should always fall with the incoming clock

Be aware of the reset! The output clock might fall mid-clock period due to the asynchronous reset.

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\\$global_clock](#)[\\$rose](#)[\\$stable](#)[Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

Goal: To safely switch from one clock frequency to another



[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\\$global_clock](#)[\\$rose](#)[\\$stable](#)[Examples](#)[▷ Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

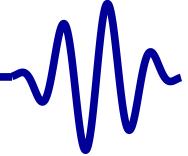
Goal: To safely switch from one clock frequency to another

- Inputs
 - Two arbitrary clocks
 - One select line

Prove that the output clock

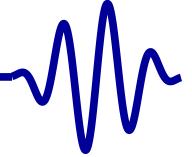
- Is always high (or low) for at least the duration of one of the clocks
- Doesn't stop

You may need to constrain the select line.

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\\$global_clock](#)[\\$rose](#)[\\$stable](#)[Examples](#)[▷ Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

Hints:

- You may assume the reset is only ever initially true
- Only one set of FF's should ever change at any time



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

▷ Cover

Lesson Overview

BMC vs Cover

Cover in Verilog

State Space

SymbiYosys

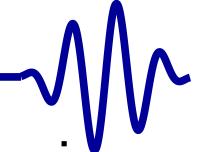
Examples

Counter

Sequences

Quizzes

Cover

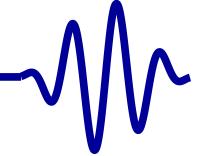
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[▷ Lesson Overview](#)[BMC vs Cover](#)[Cover in Verilog](#)[State Space](#)[SymbiYosys](#)[Examples](#)[Counter](#)[Sequences](#)[Quizzes](#)

The cover element is used to make certain something remains possible

- BMC and induction test *safety* properties
They prove that something *will not* happen
- Cover tests a *liveness* property
It proves that something *may* happen

Objectives

- Understand why cover is important
- Understand how to use cover

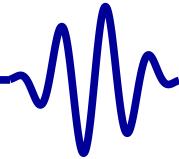
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[▷ Lesson Overview](#)[BMC vs Cover](#)[Cover in Verilog](#)[State Space](#)[SymbiYosys](#)[Examples](#)[Counter](#)[Sequences](#)[Quizzes](#)

Personal examples:

- Forgot to set f_past_valid to one
Many assertions were ignored
- Av to WB bridge, passed FV, but couldn't handle writes
- Error analysis
The simulation trace doesn't make sense. Can it be reproduced?
- As an anti-assertion
Can this situation actually happen?

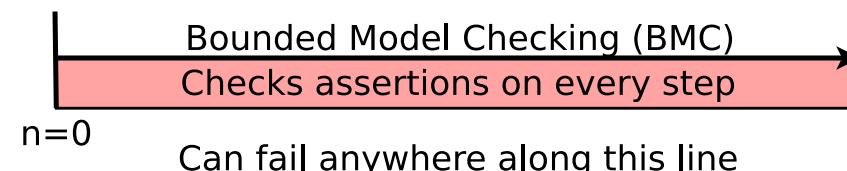
What is cover good for? Catching the *careless assumption!*

What else? Ad hoc simulation traces!

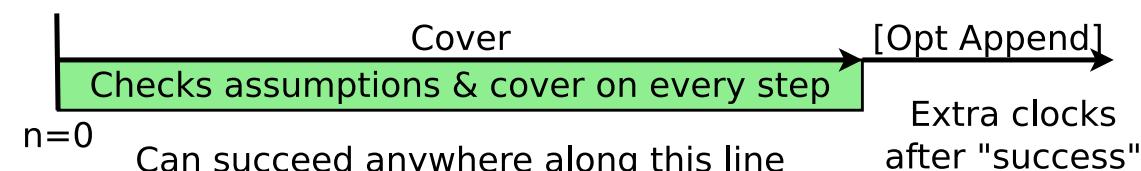
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[▷ BMC vs Cover](#)[Cover in Verilog](#)[State Space](#)[SymbiYosys](#)[Examples](#)[Counter](#)[Sequences](#)[Quizzes](#)

Cover is more like BMC than Induction is

- BMC



- Cover



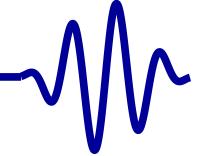
- BMC searches for failures
- Cover searches for a success

Formally, we might say . . .

- BMC + k-Induction: proof for all
 $\forall \text{assume}() \Rightarrow \forall \text{assert}()$
- Cover: there exists one
 $\forall \text{assume}() \Rightarrow \exists \text{cover}()$



Cover in Verilog



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Lesson Overview

BMC vs Cover

▷ Cover in Verilog

State Space

SymbiYosys

Examples

Counter

Sequences

Quizzes

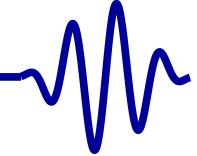
Just like an assumption or an assertion

```
// Make sure a write is possible
always @(posedge i_clk)
cover((o_wb_stb)&&(!i_wb_stall)&&(o_wb_we));

// Or

// What happens when a bus cycle is aborted?
always @(posedge i_clk)
if (i_reset)
    cover((o_wb_cyc)&&(f_wb_outstanding>0));
```

Well, almost but not quite.

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[BMC vs Cover](#)[▷ Cover in Verilog](#)[State Space](#)[SymbiYosys](#)[Examples](#)[Counter](#)[Sequences](#)[Quizzes](#)

Assert and cover handle surrounding logic differently

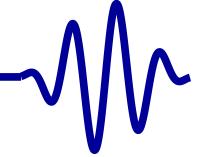
- Assert logic

```
always @(posedge i_clk)
  if (A)
    assert (B);
```

is equivalent to,

```
always @(posedge i_clk)
  assert( (!A) || (B) );
```

This is not true of cover.

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[BMC vs Cover](#)[▷ Cover in Verilog](#)[State Space](#)[SymbiYosys](#)[Examples](#)[Counter](#)[Sequences](#)[Quizzes](#)

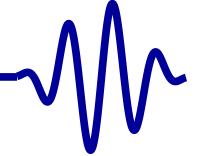
Assert and cover handle surrounding logic differently

- Assert logic
- Cover logic

```
always @(posedge i_clk)
  if (A)
    cover(B);
```

is equivalent to,

```
always @(posedge i_clk)
  cover( (A) && (B) );
// NOT the same as
//      assert( (!A) || (B) );
```



Welcome

Motivation

Basics

Clocked and \$past

 k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Lesson Overview

BMC vs Cover

Cover in Verilog

▷ State Space

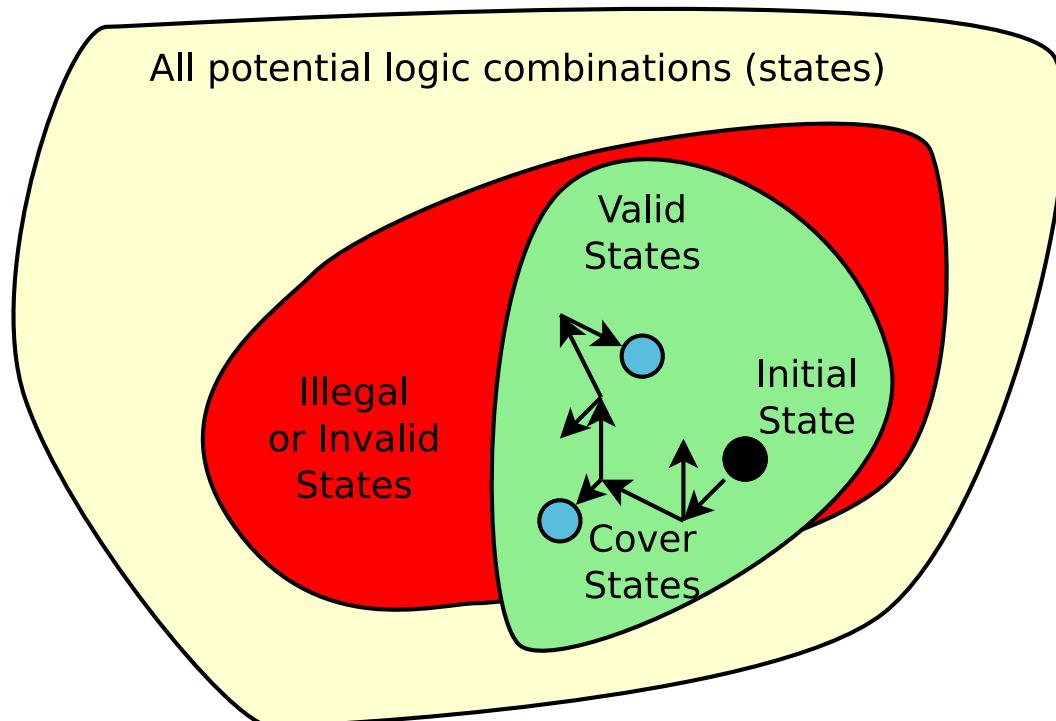
SymbiYosys

Examples

Counter

Sequences

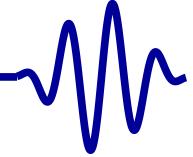
Quizzes



- Goal is to *prove* certain state's are reachable
- Prover solves for example traces



SymbiYosys



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Lesson Overview

BMC vs Cover

Cover in Verilog

State Space

▷ SymbiYosys

Examples

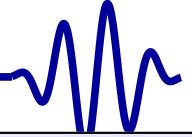
Counter

Sequences

Quizzes

The SymbiYosys script for cover needs to change as well

- SymbiYosys needs the option: **mode cover**
- Produces one trace per cover statement
... or fail



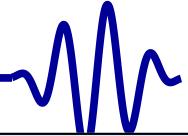
Welcome
Motivation
Basics
Clocked and \$past
 k Induction
Bus Properties
Free Variables
Abstraction
Invariants
Multiple-Clocks
Cover
Lesson Overview
BMC vs Cover
Cover in Verilog
State Space
▷ SymbiYosys
Examples
Counter
Sequences
Quizzes

```
[options]
mode cover
depth 40
append 20

[engines]
smtbmc

[script]
read -formal module.v
prep -top module

[files]
# file list
```



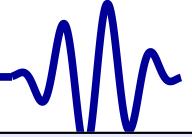
Welcome
Motivation
Basics
Clocked and \$past
k Induction
Bus Properties
Free Variables
Abstraction
Invariants
Multiple-Clocks
Cover
Lesson Overview
BMC vs Cover
Cover in Verilog
State Space
▷ SymbiYosys
Examples
Counter
Sequences
Quizzes

```
[options]
mode cover ← Run a coverage analysis
depth 40
append 20

[engines]
smtbmc

[script]
read -formal module.v
prep -top module

[files]
# file list
```



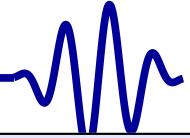
Welcome
Motivation
Basics
Clocked and \$past
 k Induction
Bus Properties
Free Variables
Abstraction
Invariants
Multiple-Clocks
Cover
Lesson Overview
BMC vs Cover
Cover in Verilog
State Space
▷ SymbiYosys
Examples
Counter
Sequences
Quizzes

```
[options]
mode cover
depth 40 ← How far to look for a covered state
append 20

[engines]
smtbmc

[script]
read -formal module.v
prep -top module

[files]
# file list
```



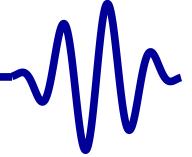
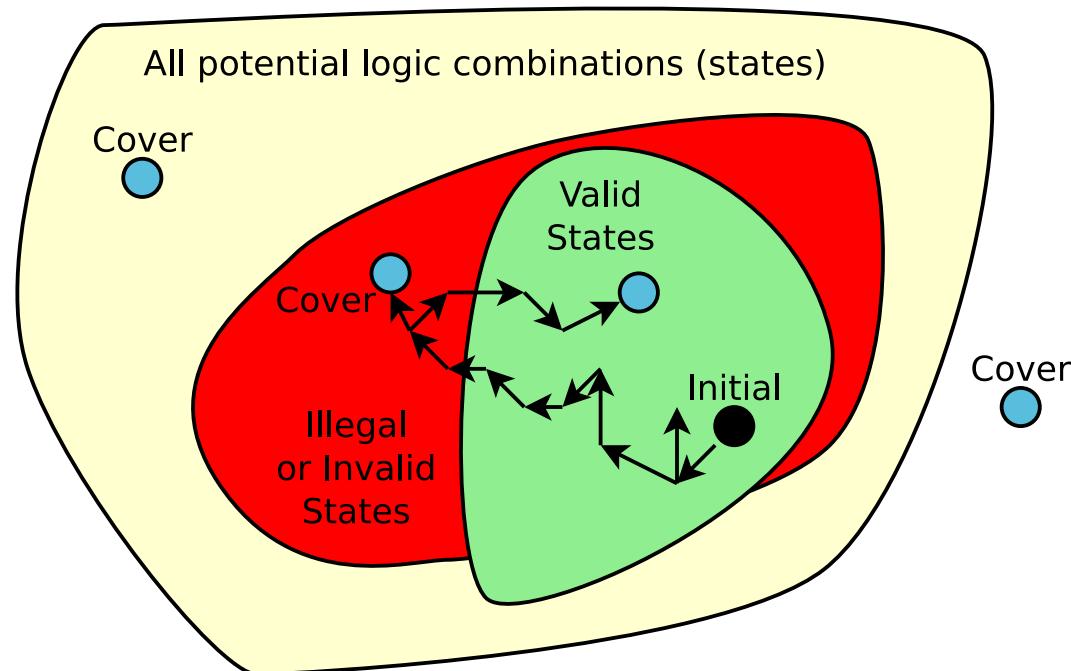
Welcome
Motivation
Basics
Clocked and \$past
 k Induction
Bus Properties
Free Variables
Abstraction
Invariants
Multiple-Clocks
Cover
Lesson Overview
BMC vs Cover
Cover in Verilog
State Space
▷ SymbiYosys
Examples
Counter
Sequences
Quizzes

```
[options]
mode cover
depth 40
append 20 ← Follow each trace with 20 extra clocks

[engines]
smtbmc

[script]
read -formal module.v
prep -top module

[files]
# file list
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[BMC vs Cover](#)[Cover in Verilog](#)[State Space](#)[▷ SymbiYosys](#)[Examples](#)[Counter](#)[Sequences](#)[Quizzes](#)

Two basic types of cover failures

1. Covered state is unreachable
No VCD file will be generated upon failure
2. Covered state is reachable, but only by breaking assertions
VCD file will be generated

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[BMC vs Cover](#)[Cover in Verilog](#)[State Space](#)[SymbiYosys](#)[▷ Examples](#)[Counter](#)[Sequences](#)[Quizzes](#)

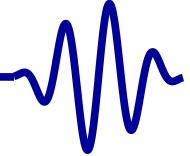
Consider a CPU I-cache:

```
always @(posedge i_clk)
    cover(o_valid);
```

With no other formal logic, what will this trace look like?

- CPU must provide a PC address
- Design must fill the appropriate cache line
- Design returns an item from that cache line

That's a lot of trace for two lines of added code!

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[BMC vs Cover](#)[Cover in Verilog](#)[State Space](#)[SymbiYosys](#)[▷ Examples](#)[Counter](#)[Sequences](#)[Quizzes](#)

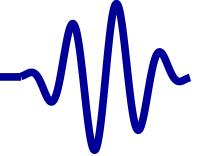
Consider a Flash controller:

```
always @(posedge i_clk)
    cover(o_wb_ack);
```

With no other formal logic, what will this trace look like?

The controller must,

- Initialize the flash device
- Accept a bus request
- Request a read from the flash
- Accumulate the result to return on the bus

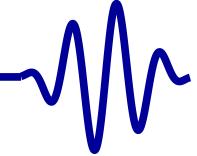
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[BMC vs Cover](#)[Cover in Verilog](#)[State Space](#)[SymbiYosys](#)[▷ Examples](#)[Counter](#)[Sequences](#)[Quizzes](#)

Consider a Memory Management Unit (MMU):

```
always @(posedge i_clk)
    cover(o_wb_ack);
```

The MMU must,

- Be told a TLB entry
- Accept a bus request
- Look the request up in the TLB
- Forward the modified request downstream
- Wait for a return
- Forward the value returned upstream

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[BMC vs Cover](#)[Cover in Verilog](#)[State Space](#)[SymbiYosys](#)[▷ Examples](#)[Counter](#)[Sequences](#)[Quizzes](#)

How about an SDRAM controller?

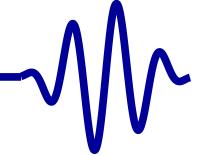
```
always @(posedge i_clk)
    cover(o_wb_ack);
```

The controller must,

- Initialize the SDRAM
- Accept a bus request
- Activate a row on a bank
- Issue a read (or write) command from that row
- Wait for a return value
- Return the result



Counter



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Lesson Overview

BMC vs Cover

Cover in Verilog

State Space

SymbiYosys

Examples

▷ Counter

Sequences

Quizzes

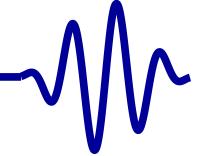
Remember our counter?

```
initial counter = 0;
always @ (posedge i_clk)
    if ((i_start_signal)&&(counter == 0))
        counter <= MAX_AMOUNT-1'b1;
    else if (counter != 0)
        counter <= counter - 1'b1;

always @ (*)
    o_busy = (counter != 0);
```



Examples



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Lesson Overview

BMC vs Cover

Cover in Verilog

State Space

SymbiYosys

Examples

▷ Counter

Sequences

Quizzes

Let's add some cover statements...

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

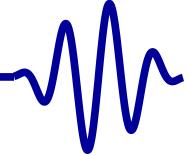
// Transition back to idle
always @(posedge i_clk)
if ((f_past_valid)&&($past(o_busy)))
    cover(!o_busy);

// Mid-cycle
always @(posedge i_clk)
    cover(counter == 3);
```

Will SymbiYosys find traces?



Examples



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Lesson Overview

BMC vs Cover

Cover in Verilog

State Space

SymbiYosys

Examples

▷ Counter

Sequences

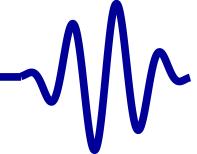
Quizzes

How about now?

```
always @(posedge i_clk)
    cover((o_busy)&&(counter == 0));
```



Examples



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Lesson Overview

BMC vs Cover

Cover in Verilog

State Space

SymbiYosys

Examples

▷ Counter

Sequences

Quizzes

How about now?

```
always @(posedge i_clk)
    cover((o_busy)&&(counter == 0));
```

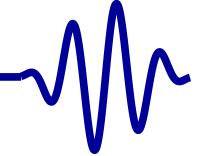
Or this one,

```
always @(posedge i_clk)
    cover(counter == MAX_AMOUNT);
```

Will these succeed?



Examples



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Lesson Overview

BMC vs Cover

Cover in Verilog

State Space

SymbiYosys

Examples

▷ Counter

Sequences

Quizzes

How about now?

```
always @(posedge i_clk)
    cover((o_busy)&&(counter == 0));
```

Or this one,

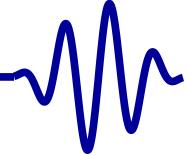
```
always @(posedge i_clk)
    cover(counter == MAX_AMOUNT);
```

Will these succeed? No. Both will fail

- These are outside the reachable state space



Examples



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Lesson Overview

BMC vs Cover

Cover in Verilog

State Space

SymbiYosys

Examples

▷ Counter

Sequences

Quizzes

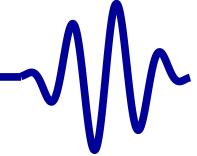
What if the state is unreachable?

```
// Keep the counter from ever starting
always @(*)  
    assume (!i_start_signal);  
  
always @(posedge i_clk)  
    cover(counter != 0);
```

Will this succeed?



Examples



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Lesson Overview

BMC vs Cover

Cover in Verilog

State Space

SymbiYosys

Examples

▷ Counter

Sequences

Quizzes

What if the state is unreachable?

```
// Keep the counter from ever starting
always @(*)
    assume (!i_start_signal);

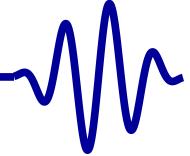
always @(posedge i_clk)
    cover(counter != 0);
```

Will this succeed? No. This will fail with no trace.

- If `i_start_signal` is never true, the cover cannot be reached



Examples



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Lesson Overview

BMC vs Cover

Cover in Verilog

State Space

SymbiYosys

Examples

▷ Counter

Sequences

Quizzes

What if an assertion needs to be violated?

```
always @(*)  
    assert(counter != 10);
```

```
always @(posedge i_clk)  
    cover(counter == 4);
```

What will happen here?



Examples



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Lesson Overview

BMC vs Cover

Cover in Verilog

State Space

SymbiYosys

Examples

▷ Counter

Sequences

Quizzes

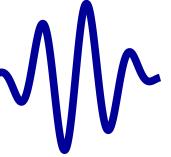
What if an assertion needs to be violated?

```
always @(*)  
    assert(counter != 10);
```

```
always @(posedge i_clk)  
    cover(counter == 4);
```

What will happen here?

- Cover statement is reachable
- But requires an assertion failure, so a trace is generated

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[BMC vs Cover](#)[Cover in Verilog](#)[State Space](#)[SymbiYosys](#)[Examples](#)[▷ Counter](#)[Sequences](#)[Quizzes](#)

Covering the clock switch



- Shows the clock switching from fast to slow,
- and again from slow to fast

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[BMC vs Cover](#)[Cover in Verilog](#)[State Space](#)[SymbiYosys](#)[Examples](#)[▷ Counter](#)[Sequences](#)[Quizzes](#)

Return to your Wishbone arbiter. Let's cover two cases:

1. Cover both A and B receiving the bus
2. Cover how B will get the bus after A gets an acknowledgement
3. Cover how A will get the bus after B gets an acknowledgement
4. Add to the last cover
 - B must request while A still holds the bus

Plot and examine traces for both cases. Do they look right?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[BMC vs Cover](#)[Cover in Verilog](#)[State Space](#)[SymbiYosys](#)[Examples](#)[▷ Counter](#)[Sequences](#)[Quizzes](#)

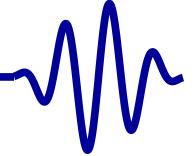
Notice what we just proved:

1. The arbiter will allow both sources to master the bus
2. The arbiter will transition from one source to another
3. The arbiter won't starve A or B

This wasn't possible with just the safety properties (assert statements)



Discussion



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Lesson Overview

BMC vs Cover

Cover in Verilog

State Space

SymbiYosys

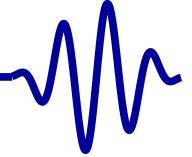
Examples

▷ Counter

Sequences

Quizzes

When should you use cover?



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

▷ Sequences

Overview

Clocking

Bind

Sequences

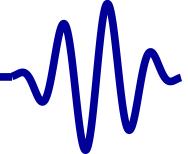
Questions?

Quizzes

Sequences



Lesson Overview



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

▷ Overview

Clocking

Bind

Sequences

Questions?

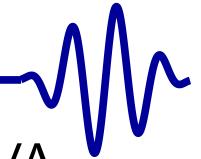
Quizzes

SystemVerilog has some amazing formal properties

- **property** can be assumed or asserted
By rewriting our assert's and assume's as properties, we can then control when they are asserted or assumed better.
- **bind** formal properties to a subset of your design
Allows us to (finally) separate the properties from the module they support
- **sequence** – A standard property description language

Objectives

- Learn the basics of SystemVerilog Assertions
- Gain confidence with yosys+verific

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[▷ Overview](#)[Clocking](#)[Bind](#)[Sequences](#)[Questions?](#)[Quizzes](#)

Much of what we've written can easily be rewritten in SVA

```
always @(*)  
if (A)  
    assert(B);
```

can be rewritten as,

```
always @(*)  
    assert(A |-> B);
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[▷ Overview](#)[Clocking](#)[Bind](#)[Sequences](#)[Questions?](#)[Quizzes](#)

Much of what we've written can easily be rewritten in SVA

```
always @(posedge i_clk)
if ((f_past_valid)&&($past(A)))
    assert(B);
```

Can be rewritten as,

```
always @(posedge i_clk)
    assert(A  $\Rightarrow$  B);
```

```
assert property( @(posedge i_clk) A  $\Rightarrow$  B);
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[▷ Overview](#)[Clocking](#)[Bind](#)[Sequences](#)[Questions?](#)[Quizzes](#)

Much of what we've written can easily be rewritten in SVA

```
always @(posedge i_clk)
if ((f_past_valid)&&($past(A)))
    assert(B);
```

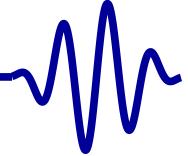
Can be rewritten as,

```
always @(posedge i_clk)
    assert(A  $\Rightarrow$  B);
```

```
assert property( @(posedge i_clk) A  $\Rightarrow$  B);
```

- Read this as A implies B on the next clock tick.
- No f_past_valid required anymore. This is a statement about the next clock tick, not the last one.

These equivalencies apply to **assume()** as well

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[▷ Overview](#)[Clocking](#)[Bind](#)[Sequences](#)[Questions?](#)[Quizzes](#)

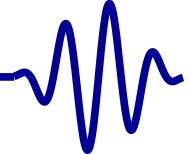
You can also declare properties:

```
property SIMPLE_PROPERTY;  
    @(posedge i_clk) a |=> b;  
endproperty
```

```
assert property(SIMPLE_PROPERTY);
```

This would be the same as

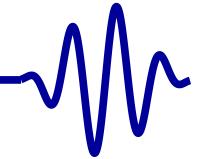
```
always @(posedge i_clk)  
if ((f_past_valid)&&($past(a)))  
    assert(b);
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[▷ Overview](#)[Clocking](#)[Bind](#)[Sequences](#)[Questions?](#)[Quizzes](#)

You could also do something like:

```
parameter [0:0] SUBMODULE = 1'b0;  
  
generate if (SUBMODULE)  
begin  
    assume property(INPUT_PROP);  
    assert property(LOCAL_PROP);  
    assert property(OUTPUT_PROP);  
end else begin  
    assert property(INPUT_PROP);  
    assume property(LOCAL_PROP);  
    assume property(OUTPUT_PROP);  
end endgenerate
```

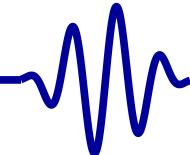
Applications: Invariants, bus properties, etc.

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[▷ Overview](#)[Clocking](#)[Bind](#)[Sequences](#)[Questions?](#)[Quizzes](#)

Properties can also accept parameters

```
property IMPLIES(a, b);  
    a |-> b;  
endproperty
```

```
assert property( IMPLIES(x, y));
```

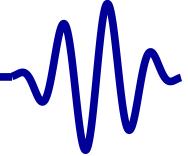
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[▷ Overview](#)[Clocking](#)[Bind](#)[Sequences](#)[Questions?](#)[Quizzes](#)

Properties can also accept parameters

```
property IMPLIES_NEXT(a, b);  
    @ (posedge i_clk) a |=> b;  
endproperty
```

```
assert property (IMPLIES_NEXT(x, y));
```

Remember, if you want to use $|=>$, **\$past**, etc., you need to define a clock.

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[▷ Clocking](#)[Bind](#)[Sequences](#)[Questions?](#)[Quizzes](#)

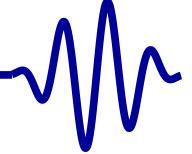
Getting tired of writing `@(posedge i_clk)`?

- You can set a default clock

```
default clocking @(posedge i_clk);  
endclocking
```

Assumes `i_clk` if no clock is given.

Clocking



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Overview

▷ Clocking

Bind

Sequences

Questions?

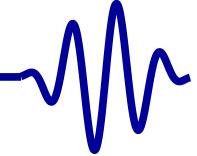
Quizzes

Getting tired of writing @(**posedge** i_clk)?

- You can set a default clock
- You can set a default clock within a given block

```
clocking @(posedge i_clk);  
    // Your properties can go here  
    // As with assert, assume,  
    // sequence, etc.  
endclocking
```

Assumes i_clk for all of the properties within the clocking block.

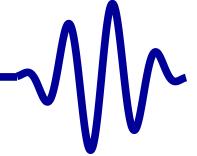
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[▷ Clocking](#)[Bind](#)[Sequences](#)[Questions?](#)[Quizzes](#)

When using verific, **\$global_clock** must first be defined

```
(* gclk *) wire gbl_clk;  
global clocking @(posedge gbl_clk); endclocking
```

This defines the **\$global_clock** ...

- as a positive edge transition of gbl_clk.
- The (* gclk *) attribute turns it into a formal timestep



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Overview

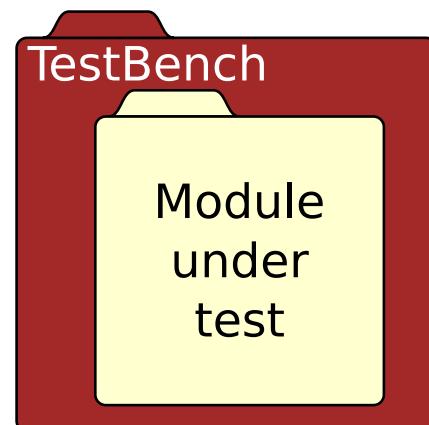
Clocking

▷ Bind

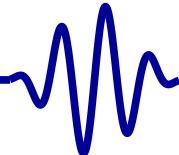
Sequences

Questions?

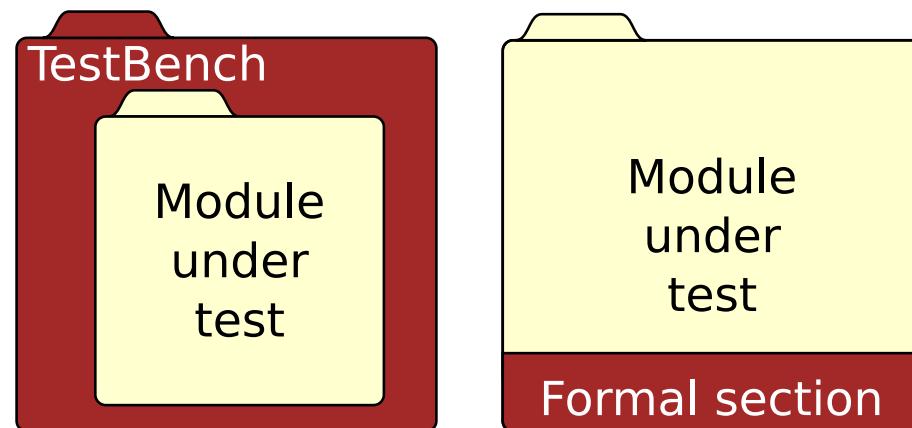
Quizzes



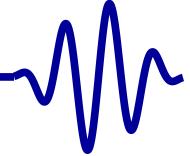
- Common bench testing works on black boxes
- This doesn't work well with formal methods



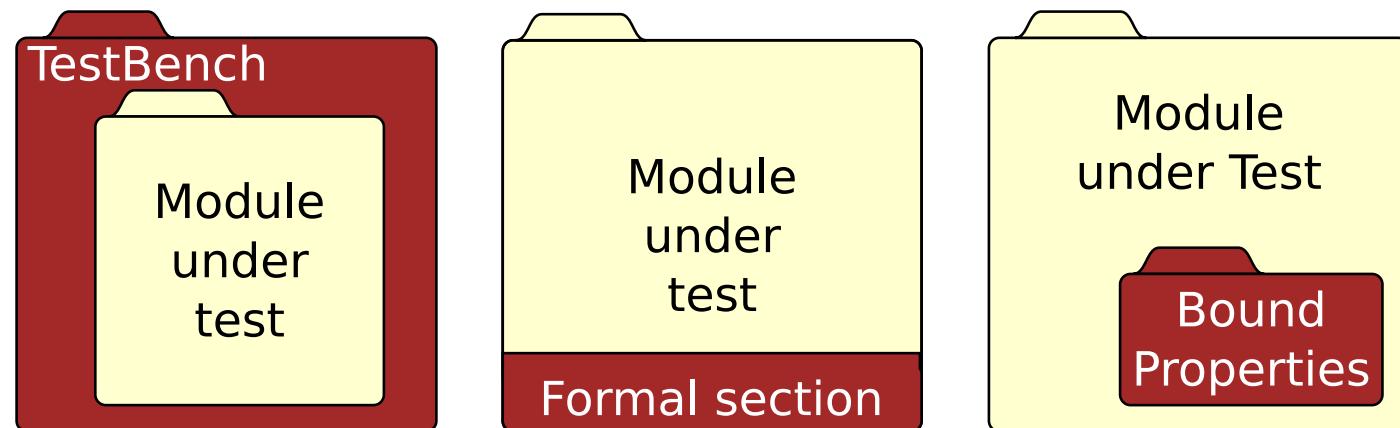
- Welcome
- Motivation
- Basics
- Clocked and \$past
- k Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Cover
- Sequences
- Overview
- Clocking
- ▷ Bind
- Sequences
- Questions?
- Quizzes



- Common bench testing works on black boxes
- This doesn't work well with formal methods
- Placing properties within a module doesn't separate the two



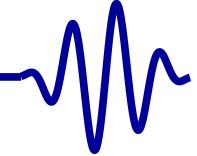
- Welcome
- Motivation
- Basics
- Clocked and \$past
- k Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Cover
- Sequences
- Overview
- Clocking
- ▷ Bind
- Sequences
- Questions?
- Quizzes



- Common bench testing works on black boxes
- This doesn't work well with formal methods
- Placing properties within a module doesn't separate the two

Using the SVA *bind* command, we can

- Separate properties from a design
- Maintains the necessary “white box” perspective

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[▷ Bind](#)[Sequences](#)[Questions?](#)[Quizzes](#)

- Can bind to specific named variables

```
module mut(input i, output o);
    reg r;
    // Your logic here
endmodule
```

```
module mut_formal(input a, input b, input r);
    // Your formal properties go here
endmodule
```

```
bind mut mut_formal mut_instance (
    // Bind inputs together
    .a(i), .b(o), .r(r)
    // The general format is
    .mut_formal_name(mut_name));
```

- Note all mut_formal ports must be inputs

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[▷ Bind](#)[Sequences](#)[Questions?](#)[Quizzes](#)

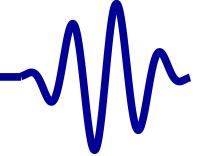
- Can bind to specific named variables
- Can also make *all* variables available to your properties

```
module mut(input i, output o);
    reg      r;
    // Your logic here
endmodule

module mut_formal(input i, input o, input r);
    // Your formal properties go here
endmodule

// Make every mut variable available in
// mut_formal with a variable of the same
// name
bind mut mut_formal mut_instance (*.);
```

- In order to use `.*`, names must match

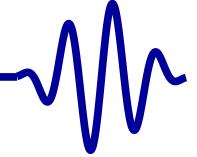
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[Sequences](#)[Questions?](#)[Quizzes](#)

- Can bind to specific named variables
- Can also make *all* variables available to your properties
- Can pass parameters through as well

```
module mut( input i, output o );
    parameter ONE = 5;
    // Your logic here
endmodule
```

```
module mut_formal( input i, input o, input r );
    parameter TWO = 14;
    // Your formal properties go here
endmodule
```

```
bind mut mut_formal #(TWO(ONE))
    mut_instance (.*);
```



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Overview

Clocking

Bind

▷ Sequences

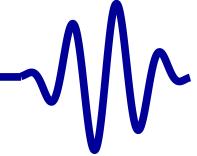
Questions?

Quizzes

So far with properties,

- We haven't done anything really all that new.
- We've just rewritten what we've done before in a new form.

Sequences are something new

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

With sequences, you can

- Specify a series of actions

sequence EXAMPLE ;

```
@(posedge i_clk) a ##1 b ##1 c ##1 d;
```

endsequence

In this example, b always follows a by one clock, c follows b, and d follows c

Sequence



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Overview

Clocking

Bind

▷ Sequences

Questions?

Quizzes

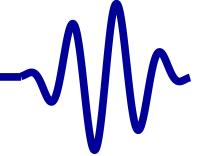
With sequences, you can

- Specify a series of actions, separated by some number of clocks

```
sequence EXAMPLE;  
  @(posedge i_clk) a ##2 b ##5 c;  
endsequence
```

In this example, b always follows a two clocks later, and c follows five clocks after b

Sequence



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Overview

Clocking

Bind

▷ Sequences

Questions?

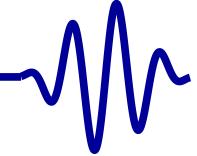
Quizzes

With sequences, you can

- Specify a series of predicates, separated in time
- Can express range(s) of repeated values

```
sequence EXAMPLE;
    @(posedge i_clk) b[*2:3] ##1 c;
endsequence
// is equivalent to ...
sequence EXAMPLE_A_2x; // 2x
    @(posedge i_clk) b ##1 b ##1 c;
endsequence
// or
sequence EXAMPLE_A_3x; // 3x
    @(posedge i_clk) b ##1 b ##1 b ##1 c;
endsequence
```

Sequence



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Overview

Clocking

Bind

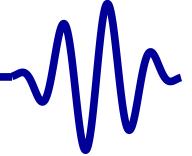
▷ Sequences

Questions?

Quizzes

With sequences, you can

- Specify a series of predicates, separated in time
- Can express range(s) of repeated values
 - $[*0:M]$ Predicate may be skipped
 - $[*N:M]$ specifies from N to M repeats
 - $[*N:$]$ Repeats at least N times, with no maximum
- Ranges can include empty sequences, such as $\#\#[*0:4]$
- Compose multiple sequences together
 - AND, seq_1 **and** seq_2
 - OR, seq_1 **or** seq_2
 - NOT, **not** seq

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

The **and** and **intersect** operators are very similar

- **and** is only true if both sequences are true
- **intersect** is only true if both sequences are true *and* have the same length

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

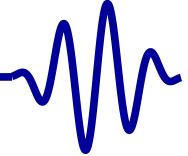
- Throughout

```
sequence A;  
  @(posedge i_clk)  
  (EXP) [*0:$] intersect SEQ;  
endsequence
```

is equivalent to

```
sequence B;  
  @(posedge i_clk)  
  (EXP) throughout SEQ;  
endsequence
```

The EXP expression must be true from now until SEQ ends

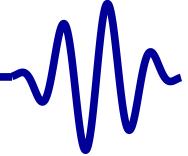
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

- Throughout
- Until

```
sequence A ;  
    @(posedge i_clk)  
        (E1) [*0:$] ##1 (E2);  
endsequence
```

is equivalent to

```
sequence B ;  
    @(posedge i_clk)  
        (E1) until E2;  
endsequence
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

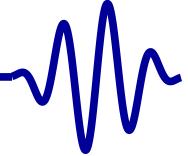
- Throughout
- Until

```
sequence A ;  
    @(posedge i_clk)  
        (E1) [*0:$] ##1 (E2);  
endsequence
```

is equivalent to

```
sequence B ;  
    @(posedge i_clk)  
        (E1) until E2;  
endsequence
```

- There is an ugly subtlety here

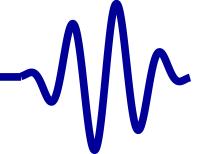
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

- Throughout
- Until
- Within

```
sequence A ;  
  @(posedge i_clk)  
  (1[*0:$] ##1 S1 ##1 1[*0:$])  
    intersect S2 ;  
endsequence
```

is equivalent to

```
sequence B ;  
  @(posedge i_clk)  
  (S1) within S2 ;  
endsequence
```



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Overview

Clocking

Bind

▷ Sequences

Questions?

Quizzes

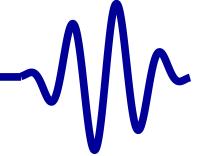
Properties can reference sequences

- Directly

```
assert property (seq);  
assert property (expr |-> seq);
```

- Implication: sequences can imply properties

```
assert property (seq |-> some_other_property);  
assert property (seq |=> another_property);
```



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Overview

Clocking

Bind

▷ Sequences

Questions?

Quizzes

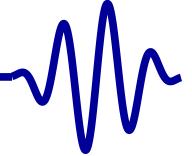
Properties can include . . .

- **if** statements

```
assert property ( if ( A ) P1 else P2 );
```

- **not**, **and**, or even **or** statements

```
assert property ( not P1 );
assert property ( P1 and P2 );
assert property ( P1 or P2 );
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

A bus request will not change until it is accepted

```
property BUS_REQUEST_HOLD;
  @(posedge i_clk)
    (STB)&&(STALL)
    |=> (STB)&&($stable(REQUEST));
endproperty

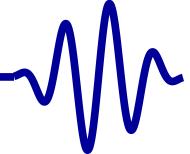
assert property (BUS_REQUEST_HOLD);
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

A request persists until it is accepted

```
sequence BUS_REQUEST;  
  @( posedge i_clk )  
    // Repeat up to MAX_STALL clks  
    ( STB ) && ( STALL ) [ *0 : MAX_STALL ]  
    ##1 ( STB ) && ( !STALL );  
endsequence  
  
assert property ( STB |-> BUS_REQUEST );
```

You no longer need to count stalls yourself.

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

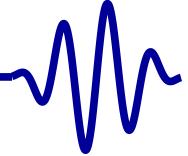
A request persists until it is accepted

```
sequence BUS_REQUEST;
    @(posedge i_clk)
        // Repeat up to MAX_STALL clks
        (STB)&&(STALL) [*0:MAX_STALL]
        ##1 (STB)&&(!STALL);
endsequence

assert property (STB |-> BUS_REQUEST);
```

You no longer need to count stalls yourself.

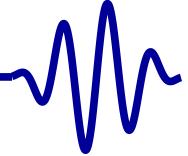
Could we do this with an **until** statement?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

A request persists until it is accepted

```
sequence BUS_REQUEST;  
  @( posedge i_clk )  
    ( STB )&&(STALL) until ( STB )&&(!STALL);  
endsequence  
  
assert property ( STB |→ BUS_REQUEST );
```

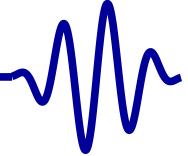
What is the difference?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

A request persists until it is accepted

```
sequence BUS_REQUEST;  
  @(posedge i_clk)  
  (STB)&&(STALL) until (STB)&&(!STALL);  
endsequence  
  
assert property (STB |> BUS_REQUEST);
```

What is the difference? The **until** statement goes forever, our prior example was limited to MAX_STALL clock cycles.

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

A request persists until it is accepted

```
sequence BUS_REQUEST;  
  @( posedge i_clk)  
  ( STB)&&(STALL) until ( STB)&&(!STALL);  
endsequence  
  
assert property ( STB |→ BUS_REQUEST );
```

What is the difference?

But . . . what happens if RESET is asserted?

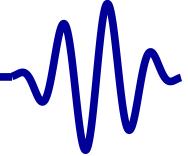
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

A property can be conditionally disabled

```
sequence BUS_REQUEST;
  @(posedge i_clk)
    // Repeat up to MAX_STALL clks
    (STB)&&(STALL) [*0:MAX_STALL]
    ##1 (STB)&&(!STALL);
endsequence

assert property (
  @(posedge i_clk)
  disable iff (i_reset)
  STB |-> BUS_REQUEST);
```

The assertion will no longer fail if `i_reset` clears the request
What if the request is aborted?

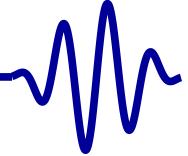
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

A property can be conditionally disabled

```
sequence BUS_REQUEST;
  @(posedge i_clk)
    // Repeat up to MAX_STALL clks
    (STB)&&(STALL) [*0:MAX_STALL]
    ##1 (STB)&&(!STALL);
endsequence

assert property (
  @(posedge i_clk)
  disable iff ((i_reset)||(!CYC))
  STB |-> BUS_REQUEST);
```

Will this work?

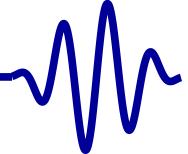
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

A property can be conditionally disabled

```
sequence BUS_REQUEST;
  @(posedge i_clk)
    // Repeat up to MAX_STALL clks
    (STB)&&(STALL) [*0:MAX_STALL]
    ##1 (STB)&&(!STALL);
endsequence

assert property (
  @(posedge i_clk)
  disable iff ((i_reset)||(!CYC))
  STB |-> BUS_REQUEST);
```

Will this work? Yes!

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

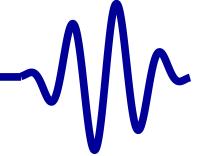
Some peripherals will only ever accept one request

```
sequence SINGLE_ACK(MAX_DELAY);
  @(posedge i_clk)
    (!ACK)&&(STALL) [*0:MAX_DELAY]
    ##1 (ACK)&&(!STALL);
endsequence

assert property (
  disable iff ((i_reset)||(!CYC))
  (STB)&&(!STALL) |=> SINGLE_ACK(32);
);
```

This peripheral will

- Stall up to 32 clocks following any accepted request, until it
- Acknowledges the request, and
- Releases the bus on the same cycle

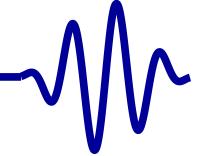
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

Some peripherals will

- Never stall the bus, and
- Acknowledge every request after a fixed number of clock ticks

```
property NEVER_STALL(DELAY);  
  @(posedge i_clk)  
  disable iff ((i_reset) || (!CYC))  
    (STB) |-> ##[*DELAY] (ACK);  
endproperty  
  
assert property (NEVER_STALL(DELAY)  
  and (!STALL));
```

This is illegal. Can you spot the bug?

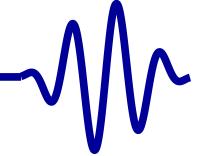
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

Some peripherals will

- Never stall the bus, and
- Acknowledge every request after a fixed number of clock ticks

```
property NEVER_STALL(DELAY);  
  @(posedge i_clk)  
  disable iff ((i_reset) || (!CYC))  
    (STB) |-> ##[*DELAY] (ACK);  
endproperty  
  
assert property (NEVER_STALL(DELAY)  
  and (!STALL));
```

This is illegal. Can you spot the bug? What logic does the **disable iff** apply to?

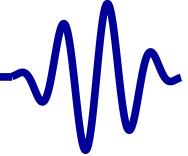
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

Some peripherals will

- Never stall the bus, and
- Acknowledge every request after a fixed number of clock ticks

```
property NEVER_STALL(DELAY);  
  @(posedge i_clk)  
  disable iff ((i_reset) || (!CYC))  
    (STB) |-> ##[*DELAY] (ACK);  
endproperty  
  
assert property (NEVER_STALL(DELAY));  
assert property (!STALL);
```

This is valid

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

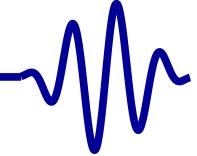
Cannot ACK or ERR when no request is pending

```
assert property (@(posedge i_clk)
    ((!i_CYC)||| (i_reset))
    ###1 ((!i_CYC)||| (i_reset))
    |-> ((!o_ACK)&&(!o_ERR));
```

Or as we did it before

```
always @(posedge i_clk)
if ((f_past_valid)
    &&(!$past(i_reset))||| (!$past(i_CYC)))
    &&((i_reset)||| (!i_CYC))
    assert ((!o_ACK)&&(!o_ERR));
```

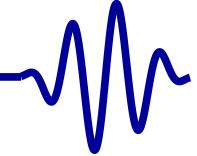
Which is simpler to understand?



Let's look at an serial port transmitter example.
A baud interval is CKS clocks . . .

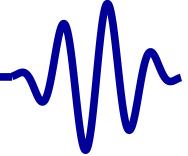
- Output data is constant
- Logic doesn't change state
- Internal shift register value is known
- Ends with zero_baud_counter

```
sequence BAUD_INTERVAL(CKS, DAT, SR, ST);
    ((o_uart_tx == DAT)&&(state == ST)
     &&(lcl_data == SR)
     &&(!zero_baud_counter))[* (CKS - 1)]
    ##1 ((o_uart_tx == DAT)&&(state == ST)
         &&(lcl_data == SR)
         &&(zero_baud_counter))
endsequence
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

A byte consists of 10 Baud intervals

```
sequence SEND(CKS, DATA);  
    BAUD_INTERVAL(CKS, 1'b0, DATA, 4'h0)  
##1 BAUD_INTERVAL(CKS, DATA[0],  
                  {{(1){1'b1}},DATA[7:1], 4'h1})  
##1 BAUD_INTERVAL(CKS, DATA[1],  
                  {{(2){1'b1}},DATA[7:2], 4'h2})  
//  
##1 BAUD_INTERVAL(CKS, DATA[6],  
                  {{(7){1'b1}},DATA[7], 4'h7})  
##1 BAUD_INTERVAL(CKS, DATA[7],  
                  7'hff,DATA[7], 4'h8)  
##1 BAUD_INTERVAL(CKS, 1'b1, 8'hff, 4'h9);  
endsequence
```

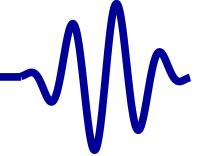
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

Transmitting a byte requires

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

assert property (@(posedge i_clk)
    (i_wr)&&(!o_busy)
    => ((o_busy) throughout
          SEND(CLOCKS_PER_BAUD, fsv_data))
    ##1 ((!o_busy)&&(o_uart_tx)
        &&(zero_baud_counter)));
```

- A transmit request is received
- The data is sent
- The controller returns to idle

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

Transmitting a byte requires

```
assert property (@(posedge i_clk)
    (i_wr)&&(!o_busy)
    |=> ((o_busy) throughout
          SEND(CLOCKS_PER_BAUD, fsv_data))
    ##1 ((!o_busy)&&(o_uart_tx)
        &&(zero_baud_counter));
```

Make sure . . .

- The sequence has a defined beginning
Only ever triggered once at a time
- Doesn't reference changing data
- **throughout** is within parenthesis
- You tie all relevant state information together



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Overview

Clocking

Bind

▷ Sequences

Questions?

Quizzes

Using SystemVerilog Assertions with Yosys requires Verific

```
[ options ]
mode prove
[ engines ]
smtbmc
[ script ]
#
#
read -formal module.v
# ... other files would go here
prep -top module
opt_merge -share_all

[ files ]
../demo-rtl/module.v
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

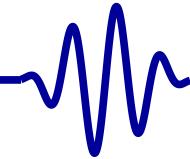
Using SystemVerilog Assertions with Yosys requires Verific

```
[options]
mode prove
[engines]
smtbmc
[script]
# The read command works both with and without Verific
# SymbiYosys script doesn't change therefore
read -formal module.v ←
# ... other files would go here
prep -top module
opt_merge -share_all

[files]
../demo-rtl/module.v
```



SysVerilog Conclusions



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Overview

Clocking

Bind

▷ Sequences

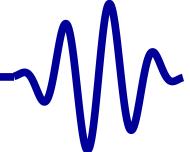
Questions?

Quizzes

SystemVerilog Concurrent Assertions . . .

- can be very powerful
- can be very confusing
- can be used with immediate assertions

You can keep using the simpler property form we've been using

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

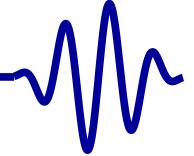
Let's formally verify a synchronous FIFO

```
module sfifo(i_clk, i_reset,
             i_wr, i_data, o_full,
             i_rd, o_data, o_empty,
             o_err);

    // ...
    'ifdef FORMAL
    // Properties understood by either
    // Yosys or Verific
    // .....

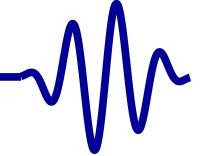
    'endif
    'ifdef VERIFIC_SVA
    // Verific-only properties
    // .....

    'endif
endmodule
```



Welcome
Motivation
Basics
Clocked and \$past
 k Induction
Bus Properties
Free Variables
Abstraction
Invariants
Multiple-Clocks
Cover
Sequences
Overview
Clocking
Bind
▷ Sequences
Questions?
Quizzes

Let's formally verify a synchronous FIFO
What properties do you think would be appropriate?



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Overview

Clocking

Bind

▷ Sequences

Questions?

Quizzes

Let's formally verify a synchronous FIFO

What properties do you think would be appropriate?

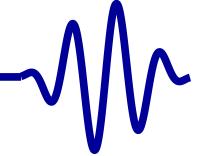
- Should never go from full to empty

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

Let's formally verify a synchronous FIFO

What properties do you think would be appropriate?

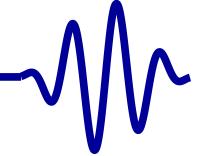
- Should never go from full to empty except on a reset

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

Let's formally verify a synchronous FIFO

What properties do you think would be appropriate?

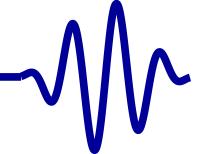
- Should never go from full to empty except on a reset
- Should never go from empty to full

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

Let's formally verify a synchronous FIFO

What properties do you think would be appropriate?

- Should never go from full to empty except on a reset
- Should never go from empty to full
- The two outputs, o_empty and o_full, should properly reflect the size of the FIFO
 - o_empty means the FIFO is currently empty
 - o_full means the FIFO has 2^N elements within it

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

Let's formally verify a synchronous FIFO

What properties do you think would be appropriate?

- Should never go from full to empty except on a reset
- Should never go from empty to full
- The two outputs, `o_empty` and `o_full`, should properly reflect the size of the FIFO
 - `o_empty` means the FIFO is currently empty
 - `o_full` means the FIFO has 2^N elements within it
- **Challenge:** Use sequences to prove that
 - Given any two values written successfully
 - Verify that those two values can (some time later) be read successfully, and in the right order
(Unless a reset takes place in the meantime)

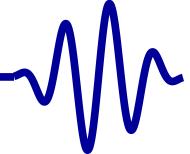
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

When using sequences, . . .

- It can be very difficult to figure out what part of the sequence failed.
The assertion that fails will reference the entire failing sequence.

Suggestions:

- Sequences must be triggered
Be aware of what triggers a sequence
- Use combinational logic to define wires that will then represent steps in the sequence
- Build the sequences out of these wires

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

Here's an example:

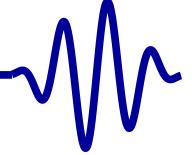
```
wire f_a, f_b, f_c;  
//  
assign f_a = // your logic  
assign f_b = // your logic  
assign f_c = // your logic  
//  
sequence ARBITRARY_EXAMPLE_SEQUENCE  
    f_a [*0:4] ##1 f_b ##1 f_c [*12:16];  
endsequence
```

If you use this approach

- Interpreting the wave file will be much easier
- The f_a, etc., lines will be in the trace



Questions?



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Overview

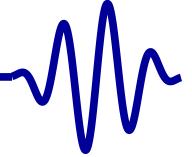
Clocking

Bind

Sequences

▷ Questions?

Quizzes



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

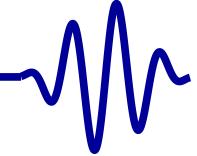
Sequences

▷ Quizzes

Quizzes



Quiz #1



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

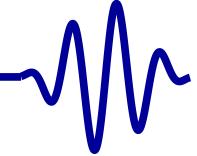
Cover

Sequences

Quizzes

Will the assertion below ever fail?

```
reg [15:0] counter;  
  
initial counter = 0;  
always @(posedge clk)  
    counter <= counter + 1'b1;  
  
always @(*)  
begin  
    assert(counter <= 100);  
    assume(counter <= 90);  
end
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

No, it will never fail.

The assumption will prohibit the assertion from being evaluated.

```
always @(*)  
begin  
    assert(counter <= 100);  
    assume(counter <= 90);  
end
```

This is an example of what I call a *careless assumption*.



Quiz #2



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

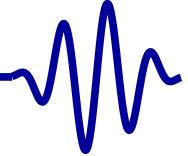
Will this simple counter ever pass formal verification?

```
parameter [15:0] MAX_AMOUNT = 22;
reg [15:0] counter;

always @ (posedge i_clk)
if ((i_start_signal)&&(counter == 0))
    counter <= MAX_AMOUNT - 1'b1;
else if (counter != 0)
    counter <= counter - 1;

always @ (*)
    o_busy = (counter != 0);

`ifdef FORMAL
    always @ (*)
        assert(counter < MAX_AMOUNT);
`endif
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

This design just needs an initial counter value to pass

```
parameter [15:0] MAX_AMOUNT = 22;
reg [15:0] counter = 0;

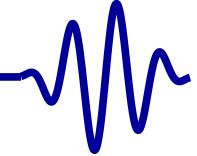
always @ (posedge i_clk)
if ((i_start_signal)&&(counter == 0))
    counter <= MAX_AMOUNT - 1'b1;
else if (counter != 0)
    counter <= counter - 1;

always @ (*)
    o_busy = (counter != 0);

`ifdef FORMAL
    always @ (*)
        assert(counter < MAX_AMOUNT);
`endif
```



Quiz #3



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Will the following design pass formal verification?

```
reg [15:0] counter;  
  
initial counter = 0;  
always @(posedge clk)  
if (counter == 16'd22)  
    counter <= 0;  
else  
    counter <= counter + 1'b1;  
  
always @(*)  
    assert(counter != 16'd500);
```

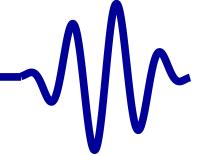
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

The following approach will pass both BMC and induction.

```
reg [15:0] counter;  
  
initial counter = 0;  
always @(posedge i_clk)  
if (i_reset) // Keep ASIC designers happy  
    counter <= 0;  
else if (counter == 16'd22)  
    counter <= 0;  
else  
    counter <= counter + 1'b1;  
  
// The correct assertion should reference  
// all of the unreachable counter values  
always @(*)  
    assert(counter <= 16'd22);
```



Quiz #4



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

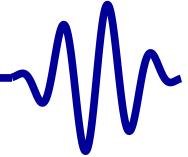
Quizzes

Will the following design pass formal verification?

```
initial counter = 0;
always @(posedge i_clk)
if ((i_start_signal)&&(counter == 0))
    counter <= 23;
else if (counter != 0)
    counter <= counter - 1'b1;

always @(*)
    assert(counter < 24);
always @(*)
    assume(!i_start_signal);

always @(posedge i_clk)
    assert($past(counter == 0));
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

If you replace **assert(\$past(counter==0));** with
assert(counter==0);, then this design passes.

```
initial counter = 0;
always @(posedge i_clk)
if ((i_start_signal)&&(counter == 0))
    counter <= 23;
else if (counter != 0)
    counter <= counter - 1'b1;

always @(*)
    assert(counter < 24);
always @(*)
    assume(!i_start_signal);

always @(posedge i_clk)
    assert(counter == 0);
```



Quiz #5



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

How are the following two assertions different?

```
initial f_past_valid = 1'b0;
always @(posedge i_clk)
    f_past_valid <= 1'b1;

always @(posedge i_clk)
if ((f_past_valid)&&($past(o_wb_stb))
    &&($past(i_wb_stall)))
    assert((o_wb_stb)
        &&($stable({i_wb_addr, i_wb_we})));
```

```
assert property (@(posedge i_clk)
    (o_wb_stb)&&(i_wb_stall)
    |=> o_wb_stb
        &&($stable({i_wb_addr, i_wb_we})));
```



Answer #5



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

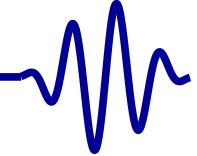
- The first assertion was an “immediate” assertion, the second a “concurrent assertion”.
- The free version of Yosys does not support concurrent assertions.
- The second assertion is easier to read

```
assert property (@(posedge i_clk)
                 (o_wb_stb)&&(i_wb_stall)
                 |=> o_wb_stb
                   &&($stable({i_wb_addr, i_wb_we})));
```

Functionally, the two assertions are *identical!*



Quiz #6



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

When using multiclock techniques, which of the below descriptions describes a signal that only changes on the positive edge of a clock?

```
always @($global_clock)
if ($fell(i_clk))
    assert($stable(signal));
```

```
always @($global_clock)
if (!$rose(i_clk))
    assert($stable(signal));
```

```
always @($global_clock)
if (!$past(i_clk))
    assert($stable(signal));
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

The correct way to assert that a signal will only change on a positive clock edge requires asserting that the signal will be stable in all other cases.

```
always @($global_clock)
if ((f_past_valid_gbl)&&(!$rose(i_clk)))
    assert($stable(signal));
```

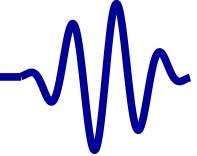
Be aware, **\$rose()** depends upon the **\$past()**, so don't forget an **f_past_valid** signal!

With **\$global_clock**, I like to call it **f_past_valid_gbl**, and define it as,

```
reg f_past_valid_gbl = 1'b0;
always @($global_clock)
    f_past_valid_gbl <= 1'b1;
```



Quiz #7



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Will this simple counter ever pass formal verification?

```
reg [15:0] counter = 0;  
  
always @ (posedge i_clk)  
if ((i_start_signal)&&(counter == 0))  
    counter <= 21;  
else if (counter != 0)  
    counter <= counter - 1;  
  
always @ (*)  
o_busy = (counter != 0);  
  
always @ (posedge i_clk)  
if ($past(i_start_signal))  
    assert(counter == 21);
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

No, the assertion would not pass: it neither checked for the past counter == 0, nor did it make sure **\$past()** was valid.

The modified assertion, below, will pass.

```
always @(posedge i_clk)
if ((f_past_valid)
    &&($past(i_start_signal))
    &&($past(counter) == 0))
    assert(counter == 21);
```

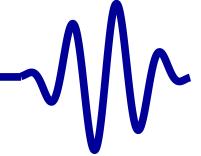
Alternatively, the following concurrent assertion would also work:

```
assert property @(posedge i_clk)
    (i_start_signal)&&(counter == 0)
    |=> (counter == 21);
```

This exercise is a good example of how formal methods force you to look just a little harder at a problem.



Quiz #8



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

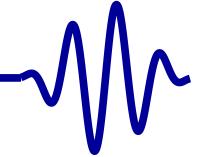
Cover

Sequences

Quizzes

Will this design pass a Bounded Model Check (BMC)?

```
reg [15:0] counter;  
  
initial counter = 0;  
always @ (posedge clk)  
    counter <= counter + 1'b1;  
  
always @ (*)  
    assert(counter < 16'd65000);
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Will this design pass a Bounded Model Check (BMC)?

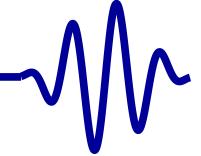
```
reg [15:0] counter;  
  
initial counter = 0;  
always @ (posedge clk)  
    counter <= counter + 1'b1;  
  
always @ (*)  
    assert(counter < 16'd65000);
```

Not unless you prove it with a depth of over 65,000!

This is a classic example of a proof that is easier to do with induction. Less than five steps of induction would find this problem.



Quiz #9



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

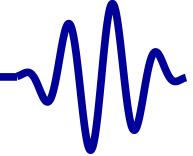
Cover

Sequences

Quizzes

Will the following design pass formal verification?

```
reg [15:0] counter;  
  
always @(*)  
begin  
    counter = 2;  
    assert(counter == 5);  
    counter = counter + 3;  
end
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Will the following design pass formal verification?

```
always @(*)  
begin  
    counter = 2;  
    assert(counter == 5);  
    counter = counter + 3;  
end
```

No, it will not pass.

- counter = 2 is a blocking statement. It is completed before the **assert()**.
- counter==2 when the **assert** is applied
- Only after the **assert** is counter set to 5.
- Were the **assert** the last line of the block, it would've passed
- This is one reason why I separate my assertions from my logic



Quiz #10



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

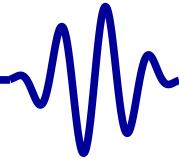
Sequences

Quizzes

Goal: to prove that whenever a request is being made, the request will stay stable until it is accepted.

Will this assertion capture what we want?

```
if (( $past( o_REQUEST ))&&( $past( i_STALL )))  
begin  
    assert( o_REQUEST );  
    assert( $stable( o_REQUEST_DETAILS ));  
end
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Not quite, there's a couple of things missing

Two examples would be `i_reset` and `f_past_valid`

Here's an updated assertion that should fix those lacks

```
if ((f_past_valid)&&(!$past(i_reset))  
    &&($past(o_REQUEST))&&($past(i_STALL)))  
begin  
    assert(o_REQUEST);  
    assert($stable(o_REQUEST_DETAILS));  
end
```

Alternatively, we could have written,

```
assert property @ (posedge i_clk)  
    disable iff (i_reset)  
    (o_REQUEST)&&(i_STALL)  
    |=> (o_REQUEST)  
        &&($stable(o_REQUEST_DETAILS));
```



Quiz #11



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

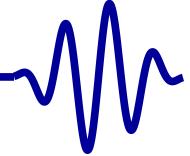
Cover

Sequences

Quizzes

The following design fails induction. How would you adjust it so that it would pass?

```
reg [15:0] sa = 0, sb = 0;  
  
always @ (posedge i_clk)  
if (i_ce)  
begin  
    sa <= { sa[14:0], i_bit };  
    sb <= { i_bit, sb[15:1] };  
end  
  
always @ (*)  
    assert (sa[15] == sb[0]);
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

There are many solutions to this problem

1. Use a non-smtbmc engine, such as abc pdr
2. Force i_ce

```
always @(posedge i_clk)
if (! $past(i_ce))
    assume(i_ce)
```

3. Assert all bits

```
always @(*)
begin
    assert(sa[14] == sb[1]);
    assert(sa[13] == sb[2]);
    assert(sa[12] == sb[3]);
    assert(sa[11] == sb[4]);
    // ... through all combinations
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

The logic below is designed to ensure that the design will only acknowledge requests and nothing more: one acknowledgment per request. It almost works. Can you spot any problem(s)?

```
initial f_nreqs = 0;
always @(posedge i_clk)
  if ((i_reset)||(!i_wb_cyc))
    f_nreqs <= 1'b0;
  else if ((i_wb_stb)&&(!o_wb_stall))
    f_nreqs <= f_nreqs + 1'b1;
  // f_nack is a similarly defined counter,
  // only one that counts acknowledgments
  always @(*)
    if (f_nreqs == f_nacks)
      assert (!o_wb_ack);
```

Assume a sufficient number of bits in f_nreqs and f_nacks.

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

No, it will not pass. The problem is that it may be possible to ACK a request on the same clock it is received. The following updated assertion will fix this.

```
always @(*)
if ((f_nreqs == f_nacks)
    &&((!i_wb_stb)||(o_wb_stall)))
    assert (!o_wb_ack);
```

Originally, I disallowed ACK's on the same clock as the STB. Then I tried formally verifying someone else's code. When it didn't pass, I went back and re-read the WB-spec only to discover the error in my ways.



Quiz #13



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Given that X is defined somehow, which of the following assertions will fail?

```
always @(posedge i_clk)
if (f_past_valid)
begin
    assert ($stable(X)
            == (X == $past(X)));
    assert ($changed(X)
            == (X != $past(X)));
    assert ($rose(X)
            == ((X)&&(!$past(X))));
    assert ($fell(X)
            == ((!X)&&($past(X))));

end
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Two of these assertions will fail if x is wider than one bit

```
assert($rose(x) == ((x)&&(!$past(x))));  
assert($fell(x) == ((!x)&&($past(x))));
```

From the 2012 SystemVerilog standard,

- `$rose` returns true if the LSB of the expression changed to 1. Otherwise, it returns false.
- `$fell` returns true if the LSB of the expression changed to 0. Otherwise, it returns false.
- `$stable` returns true if the value of the expression did not change. Otherwise, it returns false.
- `$changed` returns true if the value of the expression changed. Otherwise, it returns false.

These updated assertions will succeed,

```
assert($rose(x) == ((x[0])&&(!$past(x[0]))));  
assert($fell(x) == ((!x[0])&&($past(x[0]))));
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

The following logic creates two clocks with nearly identical frequencies. Can you spot any missing assumptions?

```
(* anyconst *) reg [7:0] f_step_one, f_step_two;
always @(*)
if (f_step_one > f_step_two)
    assume(f_step_one - f_step_two < 8'h2)
else
    assume(f_step_two - f_step_one < 8'h2)
always @($global_clock)
begin
    f_counter_one <= f_counter_one + f_step_one;
    f_counter_two <= f_counter_two + f_step_two;
    //
    assume(i_clk_one == f_counter_one[7]);
    assume(i_clk_two == f_counter_two[7]);
end
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

The step sizes cannot ever be zero, and steps greater than $8'h80$ will alias.

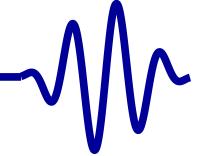
```
always @(*)
begin
    assume(f_step_one != 0);
    assume(f_step_two != 0);
    assume(f_step_one <= 8'h80);
    assume(f_step_two <= 8'h80);
end
```

For performance reasons, you may choose to assume the speed of the fastest clock.

```
always @(*)
    assume((f_step_one == 8'h80)
        ||( f_step_two == 8'h80));
```



Quiz #15



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Will the following assertion pass?

```
always @(posedge i_clk)
begin
    if (i_write)
        mem[i_waddr] <= i_data;
    if (i_read)
        o_data <= mem[i_raddr];
end

always @(posedge i_clk)
if ((f_past_valid)
    &&($past(i_write))&&($past(i_read))
    &&($past(i_waddr)==$past(i_raddr)))
    assert(o_data == $past(i_data));
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Will the following assertion pass?

```
always @(posedge i_clk)
begin
    if (i_write)
        mem[i_waddr] <= i_data;
    if (i_read)
        o_data <= mem[i_raddr];
end

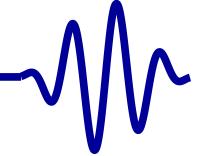
always @(posedge i_clk)
if ((f_past_valid)
    &&($past(i_write))&&($past(i_read))
    &&($past(i_waddr)==$past(i_raddr)))
    assert(o_data == $past(i_data));
```

No.

How would you describe a write-through block RAM?



Quiz #16



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

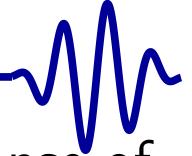
Sequences

Quizzes

The formal property below was written for the case of a synchronous reset. How would you adjust it so that it accurately reflects the behavior of the flip-flop under an asynchronous reset?

```
always @(posedge i_clk, negedge i_areset_n)
  if (!i_areset_n)
    a <= 0;
  else
    a <= something;

always @(posedge i_clk)
  if ((f_past_valid)&&($past(i_areset_n))
    assert(a == $past(something));
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

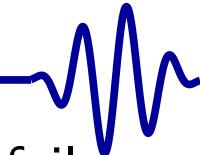
The following assertion can be used to describe the response of logic to a negative logic asynchronous reset.

```
always @(posedge i_clk, negedge i_areset_n)
  if (!i_areset_n)
    a <= 0;
  else
    a <= something;

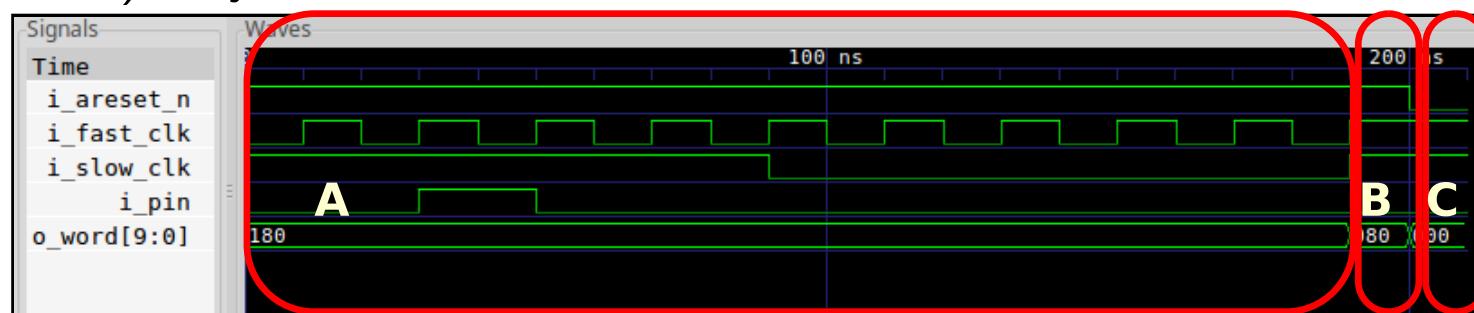
always @(posedge i_clk)
  if (!i_areset_n)
    assert(a == 0);
  else if ((f_past_valid)&&($past(i_areset_n))
    assert(a == $past(something));
```

Don't forget to assume an initial reset!

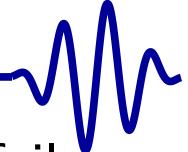
```
initial assume(!i_areset_n);
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

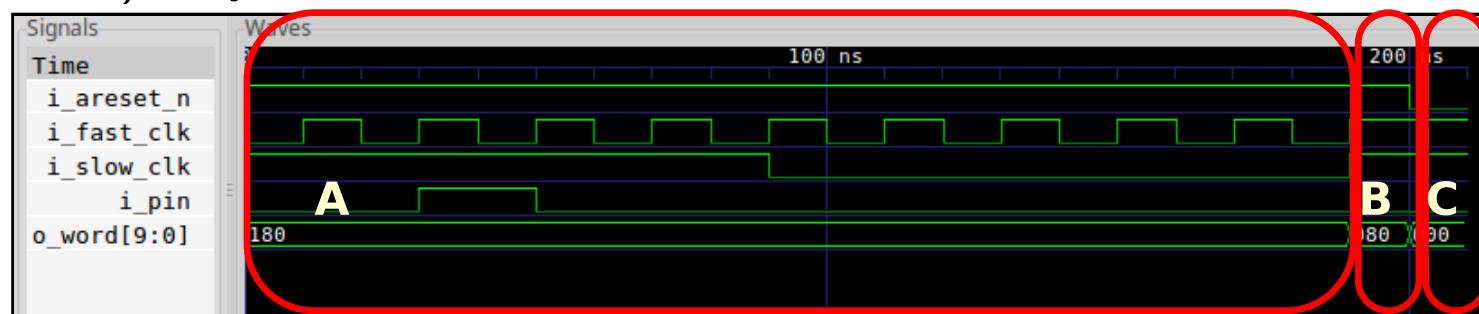
Your design passes a bounded model check (BMC), but fails during induction. Upon inspection, you find a failure in section A (below) of your trace.



How should you address this problem?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Your design passes a bounded model check (BMC), but fails during induction. Upon inspection, you find a failure in section A (below) of your trace.

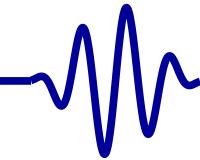


How should you address this problem?

This is not a problem with your logic. Rather, the formal properties that are constraining your logic are insufficient

- You need more properties to keep the design from failing
- If an input is out of bounds, **assume** it will be within bounds
- If your design starts in an invalid state, **assert** such invalid states will never happen
- **initial** statements will not help during induction

GT Quiz #18



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

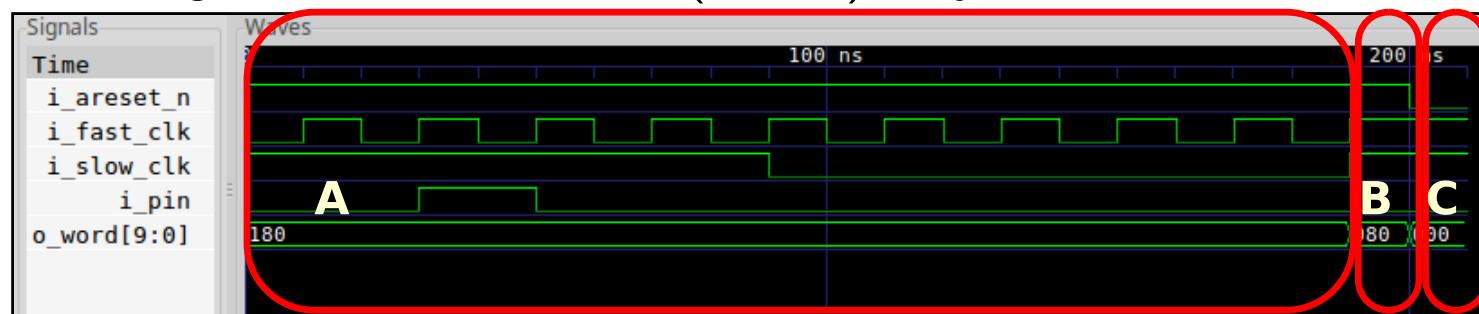
Multiple-Clocks

Cover

Sequences

Quizzes

Your design fails in section C (below) of your trace.



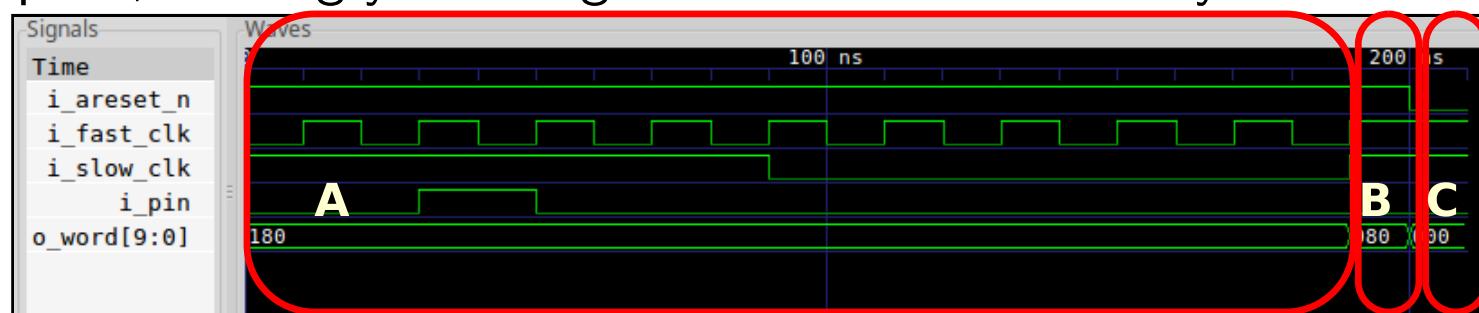
Upon inspection, you discover an

always @(posedge i_clk) assume(x); property is not getting applied.

How would you fix this situation?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

An **always @(posedge i_clk) assume(X);** property is not getting applied, causing your design to fail in section C of your trace



The problem is that **always @(posedge i_clk)** properties are not applied until the next clock edge (i.e. section B of the trace)

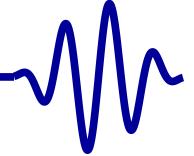
- This can cause an **always @(*) assert(Y);** to fail in section C

How would you fix this situation?

- You can make the **always @(*)** property a clocked property
- You can evaluate the **always @(posedge i_clk)** assumption as an **always @(*)** assumption instead
 - You might need to create your own **\$past** value to do this



Quiz #19



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Will the following design pass formal verification?

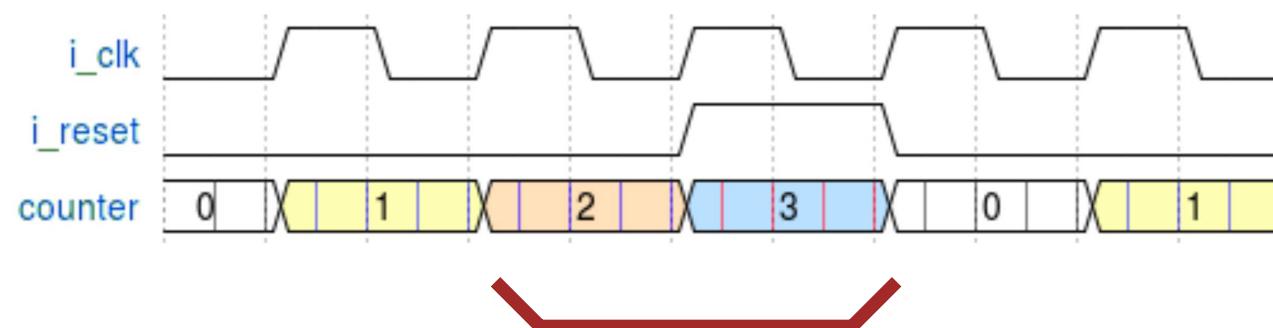
```
reg [15:0] counter = 0;
always @(posedge i_clk)
if (i_reset)
    counter <= 0;
else
    counter <= counter + 1;

always @(*)
if (counter > 2)
    assume(i_reset);

assert property (@(posedge i_clk)
    disable iff (i_reset)
    (counter < 2));
endproperty
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Much to my own surprise, this design will *pass* a formal check.



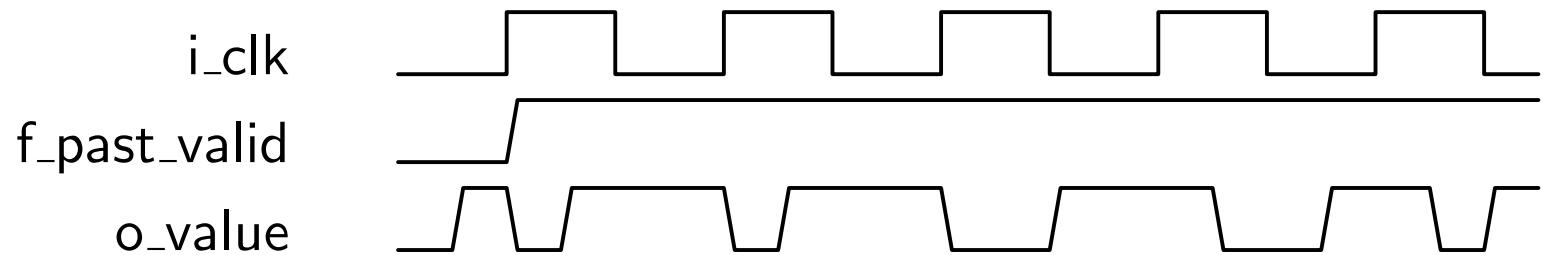
disable iff (*i_reset*) disables the check across both of these cycles

This is roughly equivalent to:

```
reg      check = 1;
always @(*posedge i_clk)
    check <= (counter < 2)||(i_reset);
always @(*)
    if (!i_reset) assert(check);
```

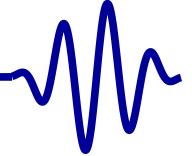
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Consider the following trace from an asynchronous context:

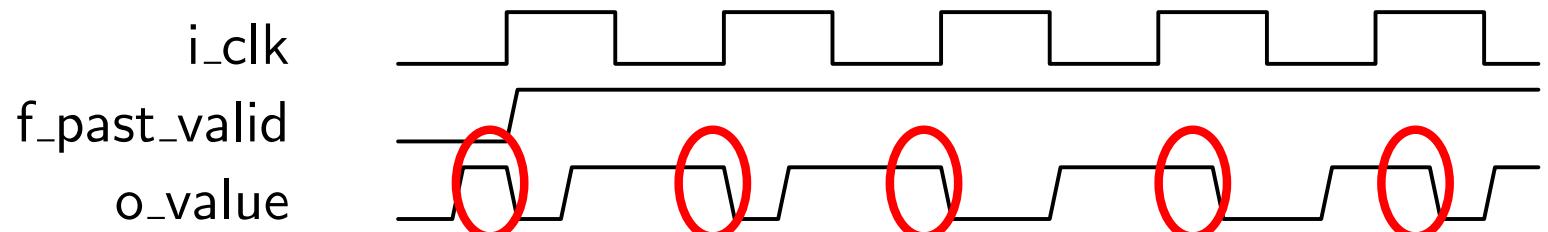


Will this formal stability assertion pass or fail?

```
always @(posedge i_clk)
  if (f_past_valid)
    assert($stable(o_value));
```

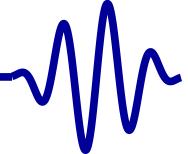
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Yes, this stability assertion will hold.



- Note that everytime **\$rose(i_clk)** is true, **\$past(o_value)** is also true.
- Since the check is only accomplished on the positive edge of **i_clk**, **o_value** is *only* checked at this time.
- Since **\$past(o_value)** is always true just prior to **@(posedge i_clk)**, the assertion passes

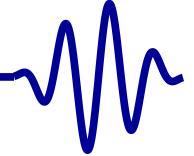
```
always @(posedge i_clk)
if (f_past_valid)
    assert ($stable(o_value));
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Your design contains the following generate block:

```
parameter [0:0] A = 1;
parameter [0:0] B = 1;
// ...
generate if (A)
begin : A_BLOCK
    // Some logic
end else if (B)
begin : B_BLOCK
    // Some other logic
end else begin : ELSE_BLOCK
    // Some final set of logic
end endgenerate
```

How should this impact the design of your SymbiYosys configuration file?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

How should conditional generate blocks be handled?

- By creating a separate task for each parameter set
- Each set of parameters can then be verified independently

[**tasks**]

A

B

Other

[**script**]

...

read -formal toplvl.v

A: chparam -set A 1 toplvl

~A: chparam -set A 0 toplvl

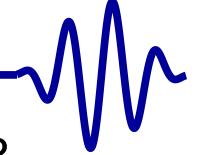
B: chparam -set B 1 toplvl

~B: chparam -set B 0 toplvl

prep -top toplvl

...

GT Quiz #22



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

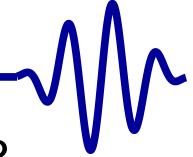
Sequences

Quizzes

When working with **cover()**, how do you handle a failure?

- On a **cover()** success a trace is generated.
No trace is generated on a **cover()** failure.
- At first glance, you have nothing to go with

How do you debug your design in this situation?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

When working with **cover()**, how do you handle a failure?

- Suppose your design needs to accomplish a sequence of steps, and then cover the last one.

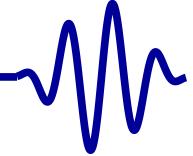
```
always @(*)  
    cover(step_24);
```

- How shall you debug this failure?

Solution: cover the intermediate steps

```
always @(*)  
begin  
    cover(step_01);  
    // ...  
    cover(step_23);  
end
```

This will lead you to the failing clock cycle

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Consider the following design:

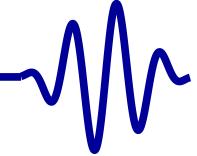
```
input wire [31:0] i_v;
output wire o_v;

assign o_v = (i_v == 32'hdeadbeef);

always @(*)
    assert(i_v != 32'hdeadbeef);

always @(*)
    assume (!o_v);
```

Given that the solver can pick any value for `i_v`, will the assertion ever fail?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

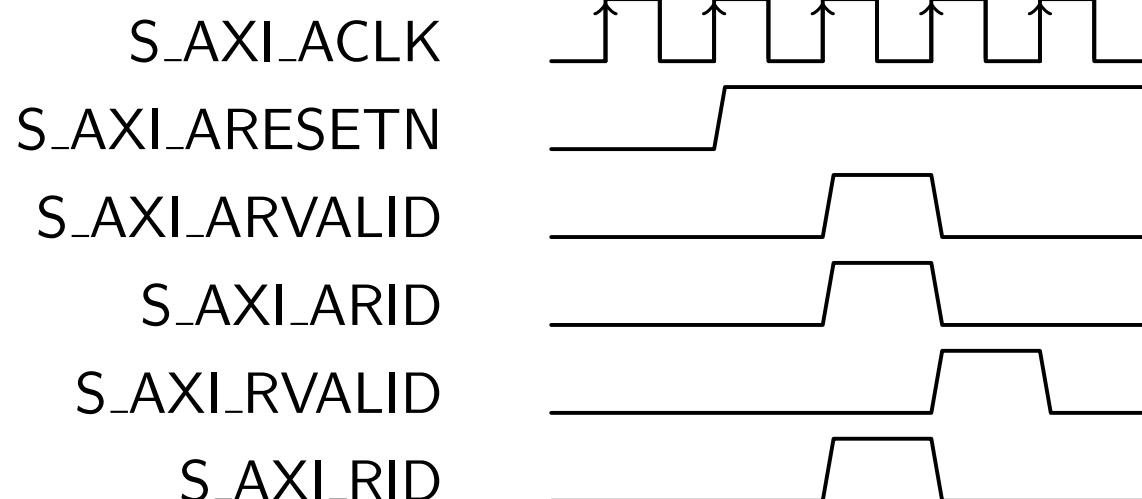
Consider the following design:

```
assign o_v = (i_v == 32'hdeadbeef);
always @(*)
    assert(i_v != 32'hdeadbeef);
always @(*)
    assume (!o_v);
```

- The assumption is forced to be true before evaluating any assertions
- $\neg o_v$ will only ever be true if $i_v \neq 32'hdeadbeef$
- Therefore, the solver will never even consider the case where $i_v == 32'hdeadbeef$
- The assertion can *never* fail

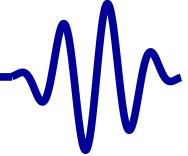
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Consider the following trace from an AXI read interaction:

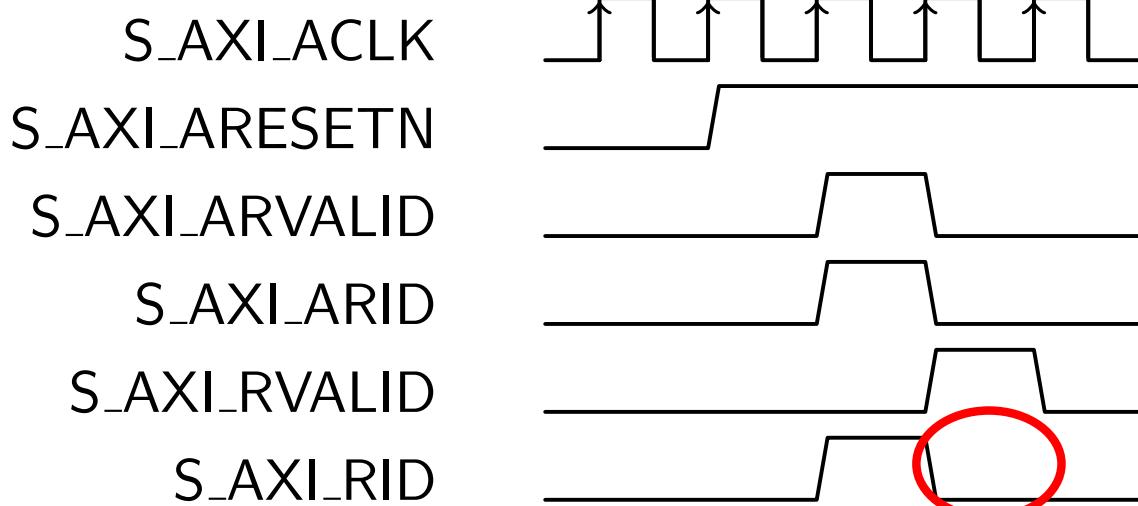


- Assume all of the relevant xREADY lines are high

Can you spot the bug?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

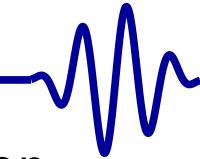
Can you spot the bug?



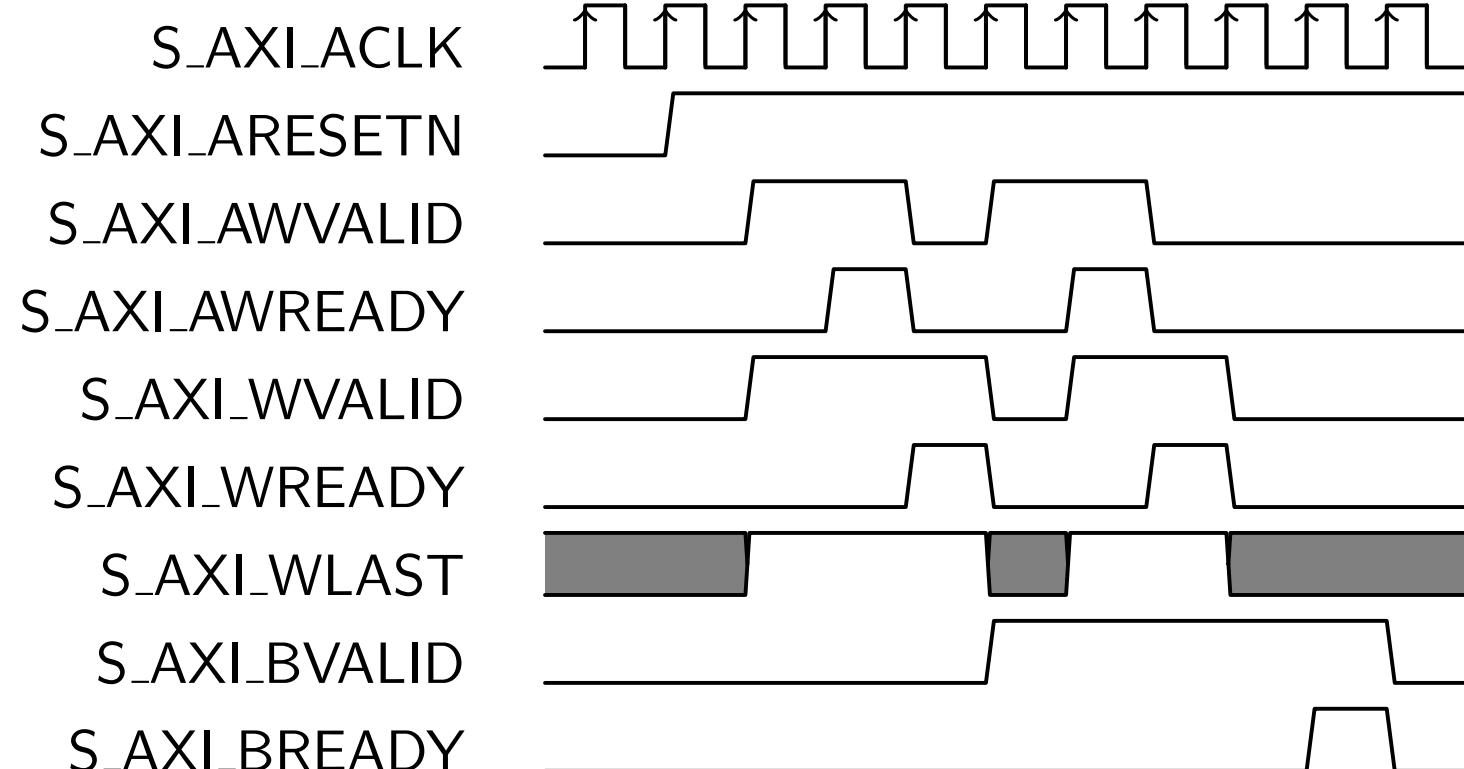
The request response has the wrong ID

- Request was made for ID=1, response has ID=0
- The cause? Xilinx's example core doesn't register the ID

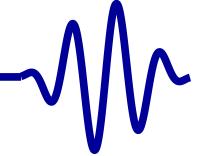
The trace above was found by applying the Symbiotic EDA Suite to Xilinx's example AXI4 core

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Consider the following trace from an AXI write interaction, ending in a steady state



What sort of formal property would catch this bug?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

A transaction timeout can find this bug

```
always @(posedge i_clk)
if ((!i_axi_reset_n)||(!i_axi_awvalid)
    ||(i_axi_awready)
    ||(f_axi_wr_pending > 0))
    f_axi_awstall <= 0;
else if ((!i_axi_bvalid)||(i_axi_bready))
    f_axi_awstall <= f_axi_awstall + 1'b1;

always @(*)
    assert(f_axi_awstall < F_AXI_MAXWAIT);
```

where `f_axi_wr_pending` is a reference to the number of remaining write data transactions in this burst

The bug in this question was found by applying the Symbiotic EDA Suite to Xilinx's example AXI4 core

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Oops, the last timeout logic captured when the incoming write address channel was *stalled*, not the *delay* on the write response channel.

- Here's the timeout logic that actually found this bug.

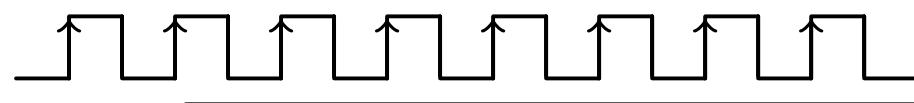
```
always @ (posedge i_clk)
if ((!i_reset_n) || (i_bvalid) || (i_wvalid)
    || ((f_awr_nbursts == 1)
        && (f_wr_pending > 0))
    || (f_awr_nbursts == 0))
    f_awr_ack_delay <= 0;
else
    f_awr_ack_delay <= f_awr_ack_delay + 1'b1;

always @ (posedge i_clk)
assert (f_awr_ack_delay < F_AXI_MAXDELAY);
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Consider the following trace drawn from an AXI interconnect I had the opportunity to verify. It had never seen a formal check before.

S_AXI_ACLK



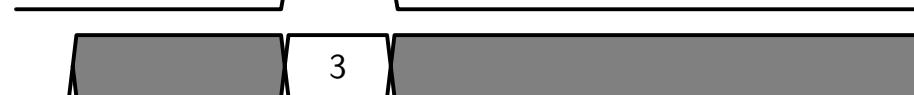
S_AXI_ARESETN



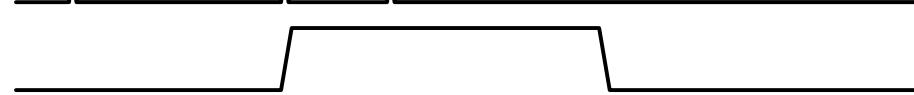
S_AXI_AWVALID



S_AXI_AWLEN



S_AXI_WVALID



S_AXI_WLAST

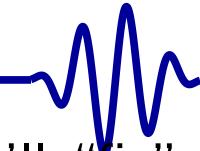


S_AXI_BVALID



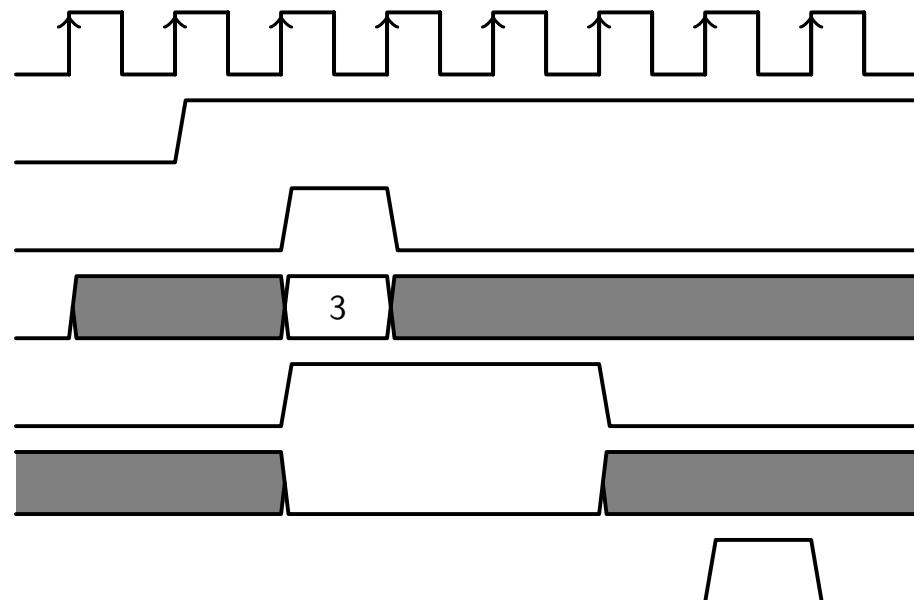
Assume all *READY signals are true

Can anyone see the bug? What formal property would catch this bug?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Correctly identifying the bug is important, otherwise you'll "fix" the wrong "bug"

S_AXI_ACLK
S_AXI_ARESETN
S_AXI_AWVALID
S_AXI_AWLEN
S_AXI_WVALID
S_AXI_WLAST
S_AXI_BVALID



In this case, there is no missing S_AXI_WLAST signal. According to spec, the burst is S_AXI_AWLEN+1 beats long, so there's still a missing write beat. The bus master just hasn't sent the final beat yet.

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

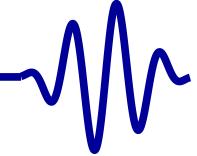
The bug? You can't return a BVALID response until the first write burst has completed.

To verify this, you need to count items remaining in the burst, I use f_wr_pending, as well as the number of bursts outstanding, something I call f_awr_nbursts. You can then check,

```
always @(*)
  if (f_awr_nbursts == 0)
    // If there are no bursts outstanding
    // then no BVALID can be returned
    assert(!S_AXI_BVALID);
  else if (f_awr_nbursts == 1)
    // If the write channel is still sending
    // data, then the BVALID cannot (yet) be
    // returned.
    assert((f_wr_pending == 0)
           || !S_AXI_BVALID);
```



Quiz #27



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Can you explain why the following cover statement fails?

```
reg      read_counter;
initial read_counter = 0;
always @(*posedge i_clk)
if (i_reset)
    read_counter <= 0;
else if (some_event)
    read_counter <= read_counter + 1;

always @(*)
    cover(read_counter > 4);
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Can you explain why the following cover statement fails?

```
reg      read_counter;
initial read_counter = 0;
always @(*posedge i_clk)
if (i_reset)
    read_counter <= 0;
else if (some_event)
    read_counter <= read_counter + 1;

always @(*)
    cover(read_counter > 4);
```

Did you notice the number of bits in the `read_counter`? At only one bit, `read_counter` can never be more than one.

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Let NM be the number of masters, and NS the number of slaves.
You want to cover a full set of write grants.

```
reg      cvr_property;
always @(*)
begin
    cvr_property = 1;
    for(iN=0; iN < (NM > NS) ? NS:NM; iN=iN+1)
        if (!write_grant[iN])
            cvr_property = 0;
end
always @(*)
    cover(cvr_property);
```

Much to my surprise, yosys ran out of memory while elaborating this code.

Can anyone see why?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

This is an order of operations issue. The example design is equivalent to

```
always @(*)
begin
    cvr_property = 1;
    for(iN=0; (iN < (NM > NS)) ? NS : NM;
        iN=iN+1)
        if (!write_grant[iN])
            cvr_property = 0;
end
```

The end condition will therefore elaborate to either NM or NS, both of which are non-zero and therefore “true”.

As for the out-of-memory error, remember this is hardware. Yosys is elaborating new hardware circuits every time through the loop, and the loop doesn't have an end.

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

There are three steps required to verify an AXI-lite interface:

1. First, attach the [formal interface property file](#)

```
'ifdef FORMAL
    faxil_slave #(
        .C_AXI_ADDR_WIDTH(C_S_AXI_ADDR_WIDTH))
    properties (
        .i_clk(S_AXI_ACLK),
        .i_axi_reset_n(S_AXI_ARESETN),
        // ...
```

2. If using SymbiYosys, you'll also need to create [an SBY file](#)

What's the missing step that's required to formally verify an AXI-lite slave interface matches bus requirements for all time?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

3. Reference the state information from the property file,

```
'ifdef FORMAL
    faxil_slave #(/* ... */)
    properties (
        .f_axi_rd_outstanding(rd_inproc),
        // ...
    )
```

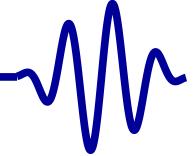
and use it to **assert()** that the state matches your logic

```
always @(*)
    assert(rd_inproc == (axi_rvalid ? 1:0)
          +(axi_already ? 0:1));
    // ...
```

The example above is from [one of my own designs](#), as this step can be very design dependent.



Quiz #30



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

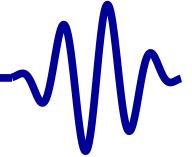
Sequences

Quizzes

The following illustrates a common FIFO mistake

```
always @ (posedge i_clk)
  if (i_reset)
    { rd_addr, wr_addr } <= 0;
  else if (i_rd)
    rd_addr <= rd_addr + 1;
  else if (i_wr)
    wr_addr <= wr_addr + 1;
```

Can you identify the bug, and suggest a way of fixing it?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

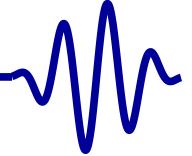
The first bug is not setting the pointers initially

```
initial {rd_addr, wr_addr } = 0;
```

The next bug is not checking for underflow or overflow

```
always @(posedge i_clk)
if (i_reset)
    { rd_addr, wr_addr } <= 0;
else if (i_rd && !o_empty)
    rd_addr <= rd_addr + 1;
else if (i_wr && !o_full)
    wr_addr <= wr_addr + 1;
```

That leaves at least one more bug

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

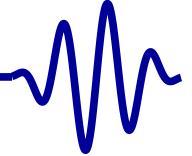
The real problem is that the whole structure is wrong.

- This really needs ot be handled in either two logic blocks, or
- Using a case statement, as shown below

```
initial {rd_addr, wr_addr } = 0;
always @(posedge i_clk)
if (i_reset)
    { rd_addr, wr_addr } <= 0;
else case({i_rd & !o_empty, i_wr && !o_full})
2'b10: rd_addr <= rd_addr + 1;
2'b01: wr_addr <= wr_addr + 1;
2'b11: begin
        rd_addr <= rd_addr + 1;
        wr_addr <= wr_addr + 1;
    end
endcase
```



Quiz #31



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

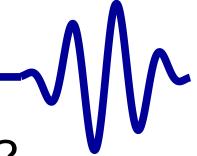
Sequences

Quizzes

The following proof passes.

```
reg      f_past_valid = 0;  
always @(posedge i_clk)  
    f_past_valid <= 1;  
  
always @(*)  
if (f_past_valid)  
    assume(i_reset);  
  
always @(posedge i_clk)  
    counter <= really_complex_logic;  
  
always @(*)  
if (f_past_valid && !i_reset)  
    assert(counter == counter + 1);
```

Can you spot the bug?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Did you notice the assumption that `i_reset` is held high?

```
always @(*)
if (f_past_valid)
    assume(i_reset);
```

The assertion never got checked!

```
always @(*)
if (f_past_valid && !i_reset)
    assert(counter == counter + 1);
```

A basic cover test would find this problem

```
always @(*)
    cover(f_past_valid && !i_reset);
// or even
always @(*)
    cover(counter == counter + 1);
```



Quiz #32



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

How would you verify the o_empty and o_full properties of a FIFO, given the read and write addresses?

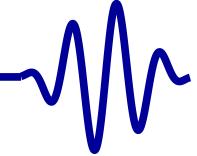
- The o_empty flag

```
assign fill = wr_addr - rd_addr;  
always @(*)  
begin  
    assert(o_empty == (fill == 0));
```

- The o_full flag, given a FIFO with FIFO_SIZE elements

```
assert(o_full == (fill >= FIFO_SIZE));  
// ...  
end
```

What property is missing?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

The missing property?

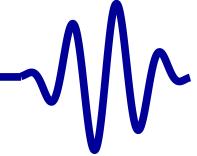
- We checked the `o_empty` flag
- We checked the `o_full` flag
- Don't forget to check that the fill never exceeds the capacity of the FIFO

```
assert(fill <= FIFO_SIZE);
```

Checking the data content of the FIFO still requires the twin write followed by twin read test. You can read more about that in [my on-line tutorial](#).



Quiz #33



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Formally verifying a cache requires three properties

First, let the solver to pick an arbitrary address and value

```
(* anyconst *) reg [AW-1:0] f_const_addr;  
(* anyconst *) reg [DW-1:0] f_const_data;
```

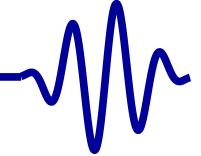
1. Then when the bus returns a value for the given address, **assume** the known value.

```
if (i_wb_ack && ackd_address == f_const_addr)  
    assume(i_wb_data == f_const_data);
```

2. Whenever the cache returns the value for the special address, **assert** that the known value is returned

```
if (o_valid && o_address == f_const_addr)  
    assert(o_value == f_const_data);
```

3. What's missing?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Formally verifying a cache requires three properties

First, allow the solver to pick an arbitrary address, and an arbitrary data word at that address.

1. **assume** a known bus response from the given address
2. **assert** that same response from the cache when that same address is requested

The missing property?

3. Assert that, if the known address is validly within the cache, that the value associated with that address matches the solver chosen value

```
always @(*)
  if (cache_valid[f_const_addr])
    assert(cache[f_const_addr [CW - 1:0]]
          == f_const_data);
```



Quiz #34



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

The following code illustrates a common AXI coding mistake:

```
always @(posedge S_AXI_ACLK)
if (!S_AXI_ARESETN)
    // Do something
else if (S_AXI_AWVALID && S_AXI_AWREADY
            && something_else)
    // Write logic
else if (S_AXI_BREADY)
    // Last condition
    // ....
```

Can you identify the bug, and suggest one or two fixes?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

The following code illustrates a common AXI coding mistake:

```
always @(posedge S_AXI_ACLK)
// ...
if (S_AXI_AWVALID && S_AXI_AWREADY
    && something_else)
    // ...
```

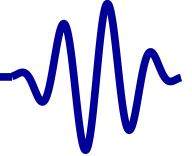
The mistake? Checking for `something_else` when processing information from the bus. To fix it,

1. Adjust the logic for `S_AXI_AWREADY`
2. Prove that every time `something_else` is false, then `S_AXI_AWREADY` is will also be false

```
assert property (@(posedge S_AXI_ACLK)
    !something_else |-> !S_AXI_AWREADY);
```



Quiz #35



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

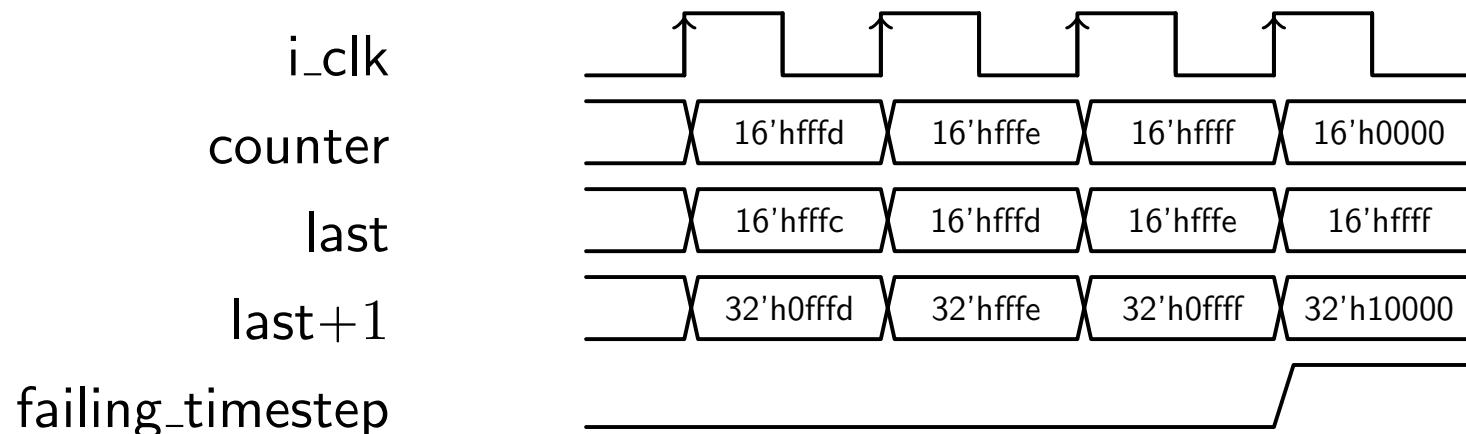
Quizzes

Will the following logic pass formal verification?

```
reg [15:0] counter, last;  
  
initial counter = 1;  
initial last = 0;  
  
always @ (posedge i_clk)  
begin  
    counter <= counter + 1;  
    last <= counter;  
end  
  
always @ (*)  
    assert(last + 1 == counter);
```

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

The problem is that `last+1` is a 32-bit value, whereas `counter` is a 16-bit unsigned value. This assertion will always fail when counter rolls over.

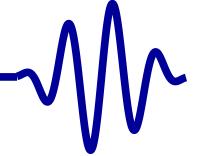


If you map `last+1` to a 16-bit value, the assertion will pass

```
wire [15:0] last_plus_one = last + 1;  
always @(*)  
    assert(last_plus_one == counter);
```



Quiz #36



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

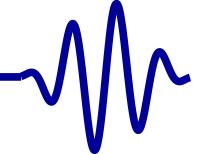
Sequences

Quizzes

The following code generates a warmup failure.

```
input    wire    [31:0]  i_a, i_b, i_c;  
  
always @(*)  
begin  
    assume(i_a+ i_b == 32'h4);  
    assume(      i_b +i_c == 32'h8);  
    assume(i_a+{ i_b, 1'b0}+i_c == 32'h7);  
end
```

Which assumption is at fault?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Which assumption is at fault?

```
input    wire    [31:0]  i_a, i_b, i_c;  
  
always @(*)  
begin  
    assume(i_a+ i_b == 32'h4);  
    assume(      i_b +i_c == 32'h8);  
    assume(i_a+{ i_b, 1'b0}+i_c == 32'h7);  
end
```

Removing any one of these assumptions will resolve the warmup failure.

- This illustrates one of the fundamental problems of warmup failures: Since any one of several assumptions might cause the design to fail, there's no way for the solver to tell which assumption was truly at fault.