



Gisselquist  
Technology, LLC

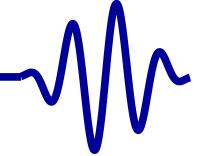
# An Introduction to Formal Methods

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

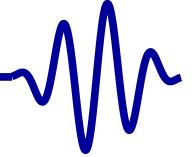
Multiple-Clocks

Cover

Sequences

Quizzes

- We'll be primarily using the *immediate assertion* subset of the full SystemVerilog assertion language
  - It's easier to understand
  - Concurrent assertions are built on top of immediate assertions under the hood
- 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

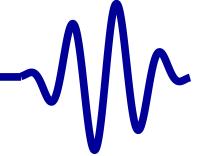
Sequences

Quizzes

# Motivation



# Lesson Overview



Welcome

Motivation

▷ Intro

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

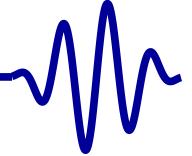
1. Why are you here?
2. What can I provide?
3. What have I learned from formal methods?

## 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.”

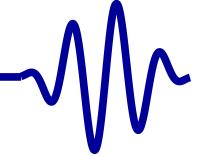
[Welcome](#)[Motivation](#)[▷ Intro](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

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

– Shahar Ariel, now the former Head of VLSI design at Mellanox



# Pentium FDIV



Welcome

Motivation

▷ Intro

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

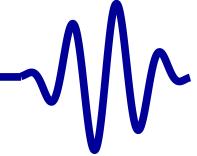
One little mistake . . .

. . . \$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,

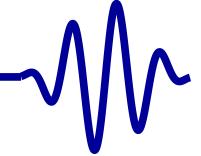
- Formal bus properties (Wishbone, Avalon, AXI, etc.)
- Bus bridges (WB-AXI, Avalon-WB)
- AXI DMA's, firewalls, crossbars
- Prefetches, cache controllers, memory controllers, MMU
- SPI slaves and masters
- UART, both TX and RX
- FIFO's, signal processing flows, FFT
- Display (VGA) Controller
- Flash 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](#)

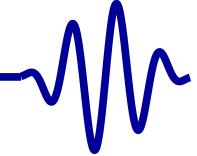
I've found bugs in things I thought were working.

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](#)

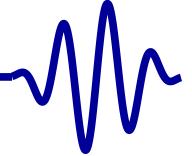
- *It worked in my test bench*
- Failed when reading and writing on the same clock while empty
  - Write first then read worked
  - R+W on full FIFO is okay
  - R+W on an empty FIFO

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

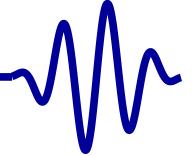
- *It worked in my test bench*
- Failed when reading and writing on the same clock while empty
  - Write first then read worked
  - R+W on full FIFO is okay
  - R+W on an empty FIFO . . . **not so much**
- My test bench didn't check that, formal did

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

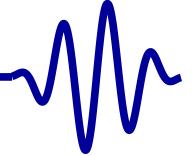
- *It worked in my test bench*
- Ugliest bug I ever came across was in the prefetch cache  
It passed test-bench muster, but failed in the hardware with a  
strange set of symptoms
- When I learned formal, it was easy to prove that this would  
never happen again.
- Low logic has always been one of my 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

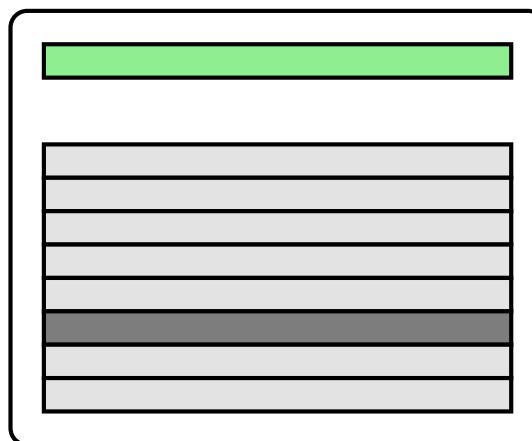
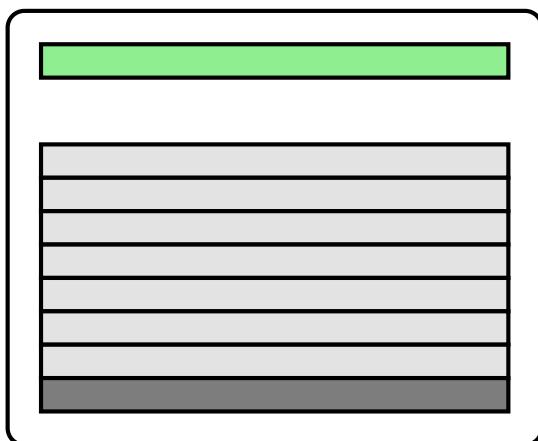
Multiple-Clocks

Cover

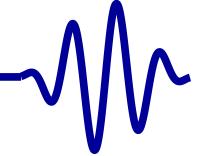
Sequences

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 design for years. Formal methods caught it.



Welcome

Motivation

▷ Intro

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

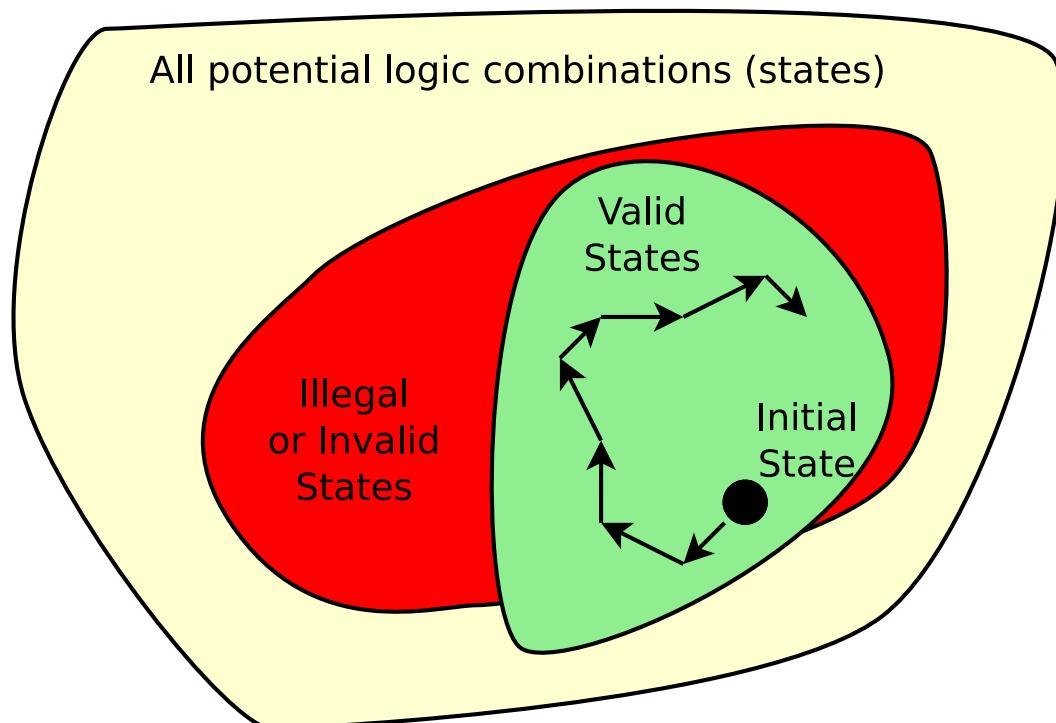
Invariants

Multiple-Clocks

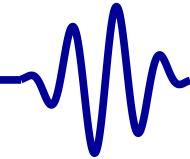
Cover

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 design 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,

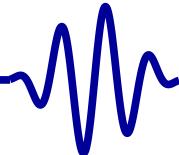
- I only read when I knew it wasn't empty

For the Prefetch,

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

For the SDRAM,

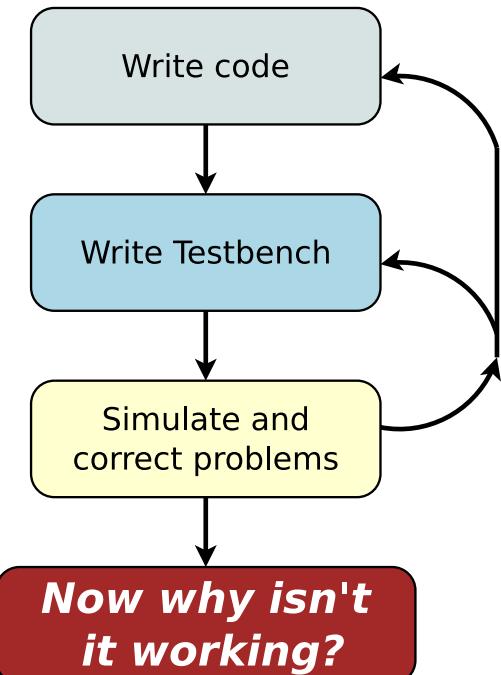
- The error was so obscure, it would be hard to trigger

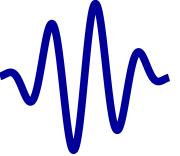
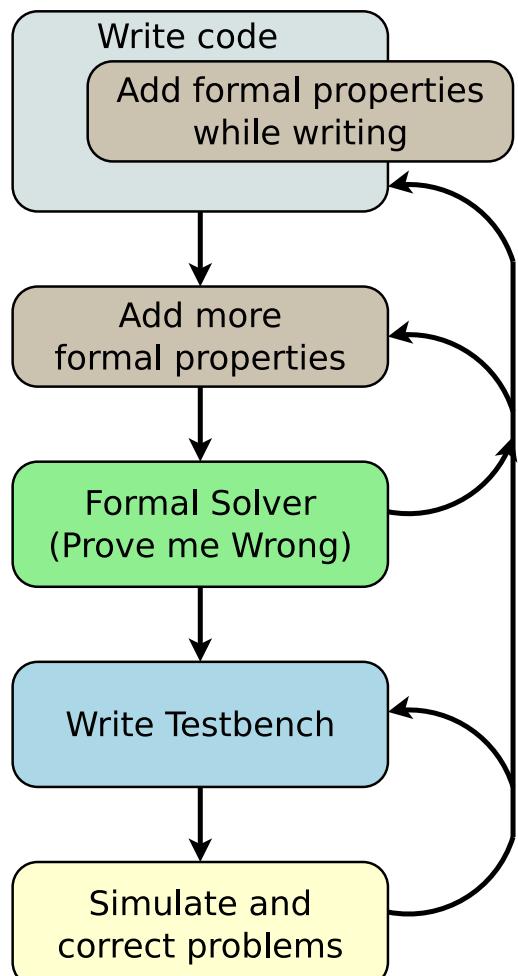
[Welcome](#)[Motivation](#)[Intro](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

This was my method before starting to work with formal.

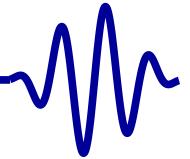
- After . . .
  - Proving my design with test benches
  - Directed simulation
- I was still chasing bugs in hardware

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

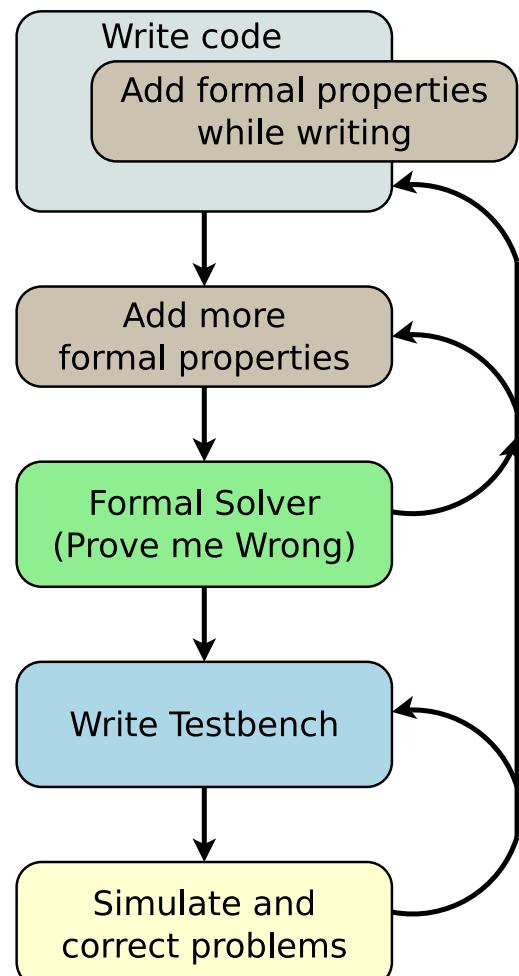
Invariants

Multiple-Clocks

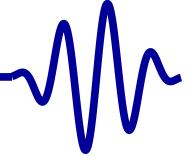
Cover

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.

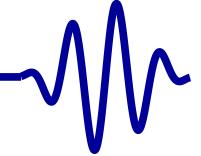
Our Objective:

- 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 design 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](#)

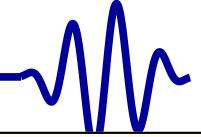
Let's start with the two basic operators

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](#)

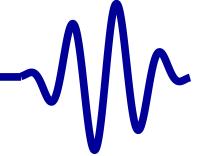
```
always @(*)  
    assert(x);
```

// 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

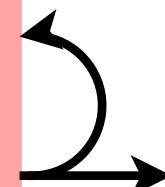
Sequences

Quizzes

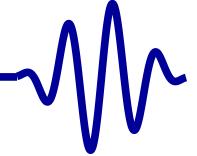
## Master FV Rule

→ assume(inputs);

assert(local state);  
assert(outputs);



# GT Assert



Welcome

Motivation

Basics

Basics

General Rule

▷ Assert

Assume

BMC

Ex: Counter

Sol'n

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

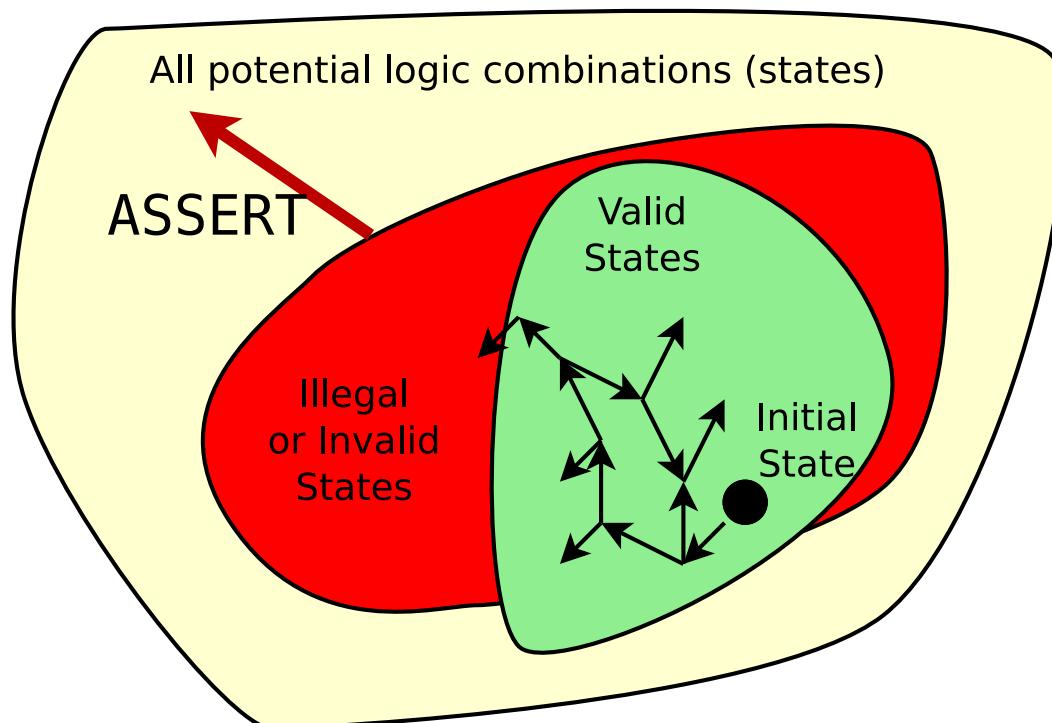
Invariants

Multiple-Clocks

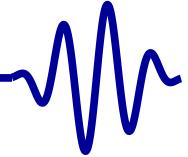
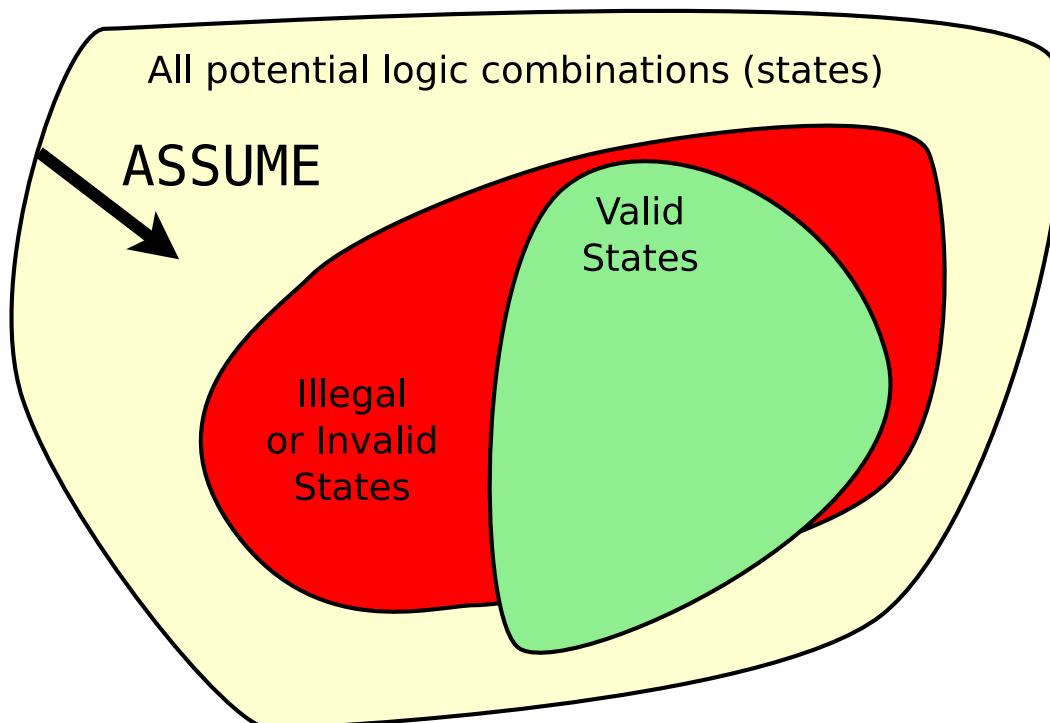
Cover

Sequences

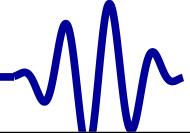
Quizzes



- 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](#)

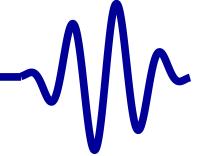
```
reg      [15:0]  counter;  
  
initial counter = 0;  
always @ (posedge clk)  
    counter <= counter + 1'b1;  
  
always @ (*)  
begin  
    assert(counter <= 100);  
    assume(counter <= 90);  
end
```

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()**

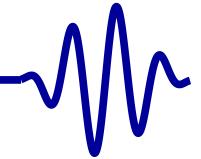
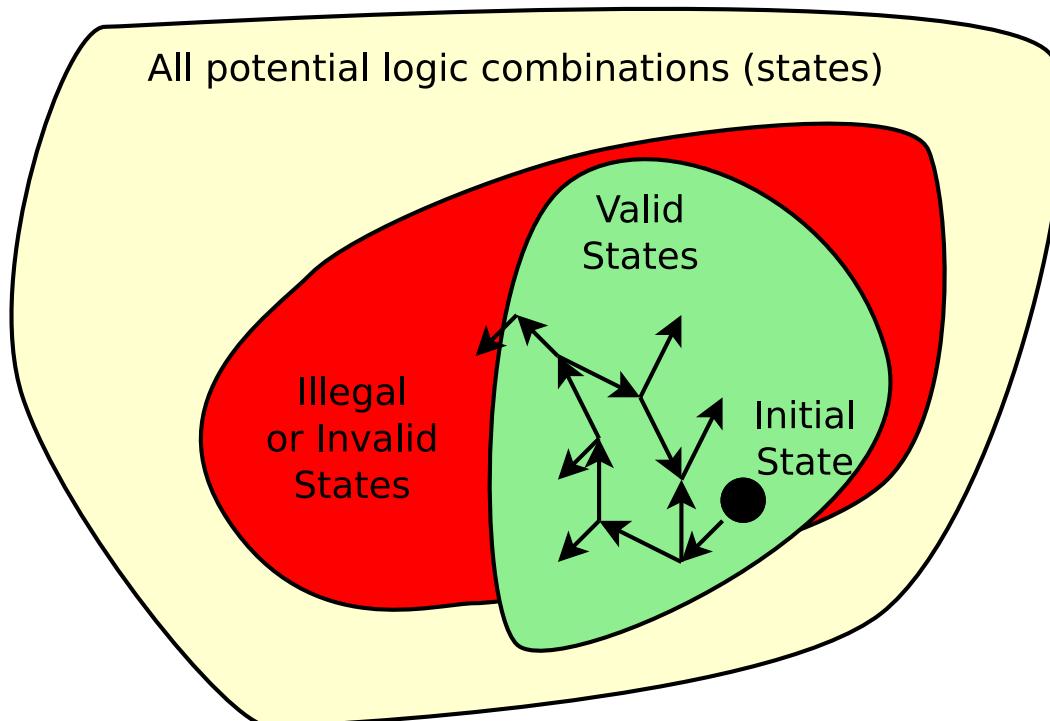
Operator	Formal Verification	Traditional Simulation
<b>restrict ()</b>	Restricts search space	Ignored
<b>assume()</b>		Halts simulation with an error
<b>assert()</b>	Illegal state	

[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()**

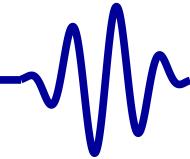
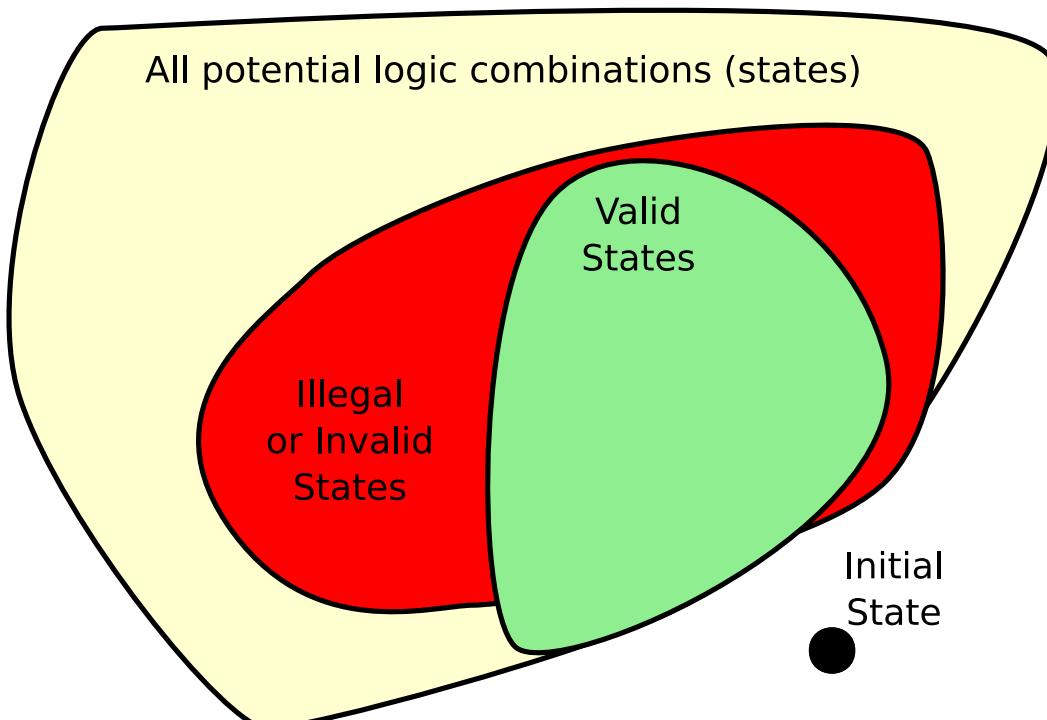
Operator	Formal Verification	Traditional Simulation
<b>restrict ()</b>	Restricts search space	Ignored
<b>assume()</b>		Halts simulation with an error
<b>assert()</b>	Illegal state	

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

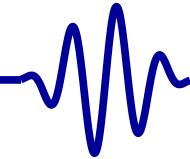
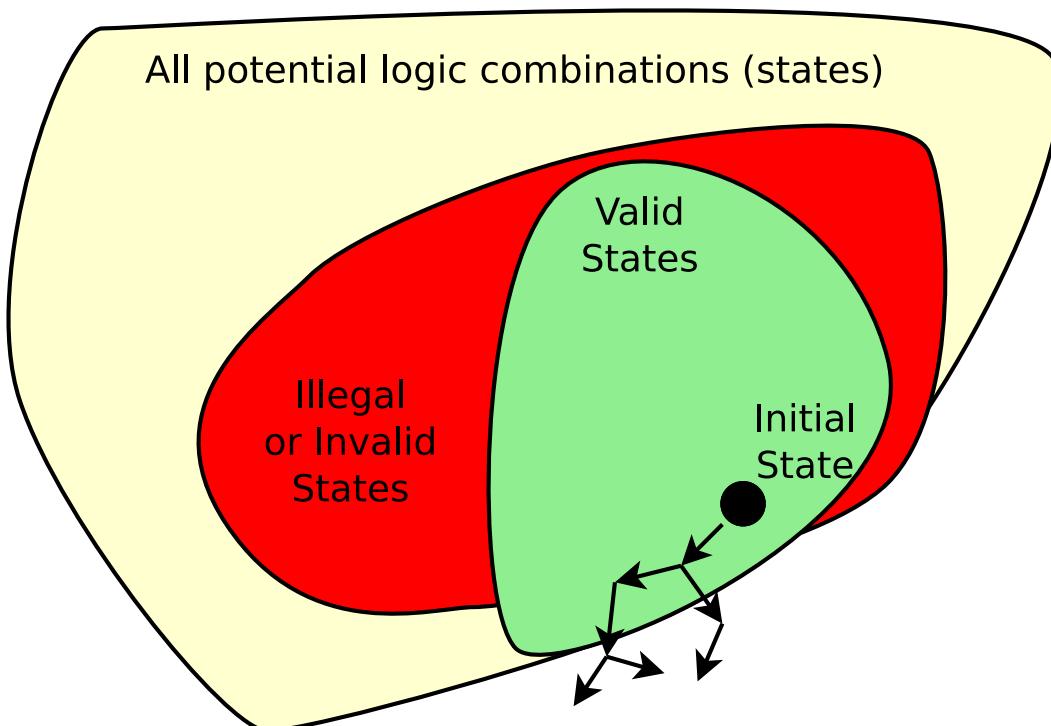
[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[▷ BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

For bounded model checking,

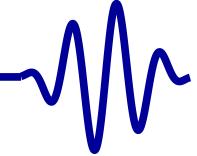
1. Start at the initial state
2. Examine *all* possible states for  $N$  clocks
3. Try to find a way to make an **assert**(); fail
4. If it's not possible in  $N$  clocks, then *pass*

[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

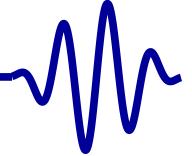
Sequences

Quizzes

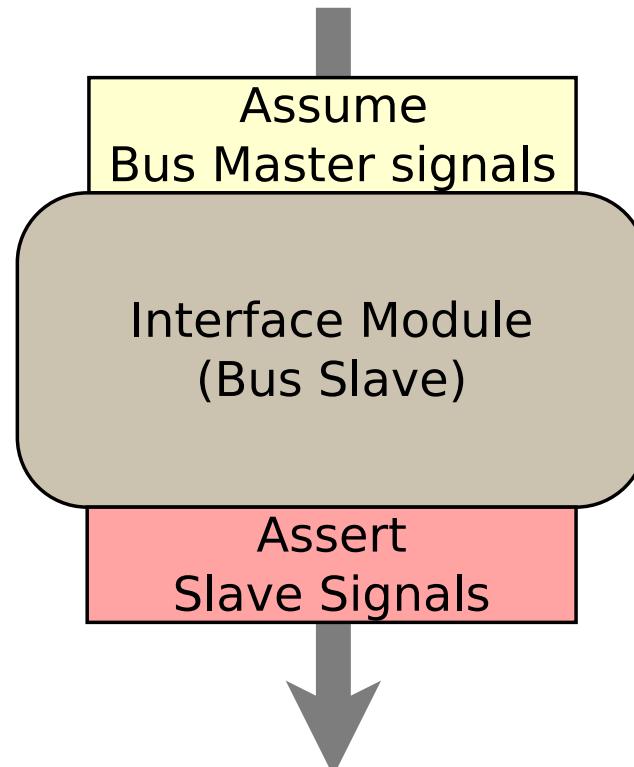
Unlike the rest of your digital design, formal properties . . .

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

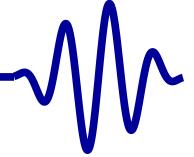
We'll discuss this more as we go along.

[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

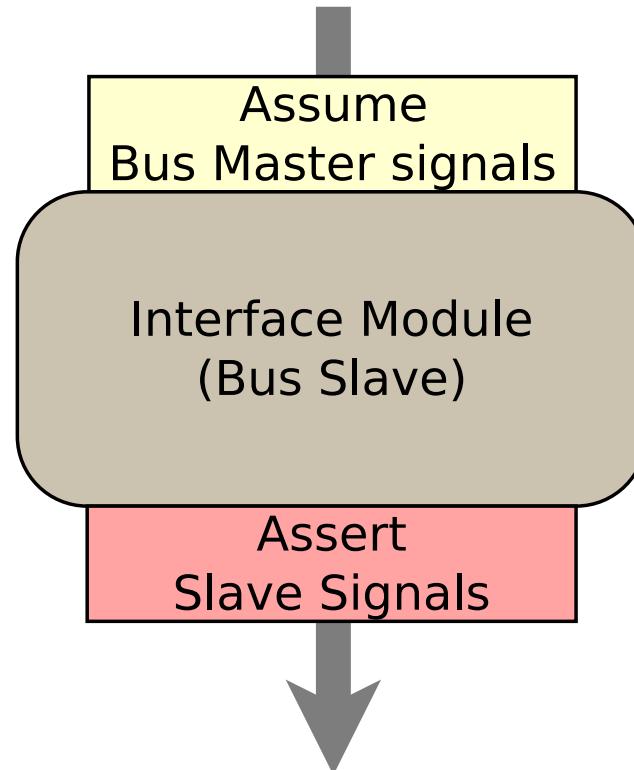
Multiple-Clocks

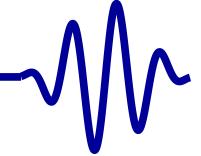
Cover

Sequences

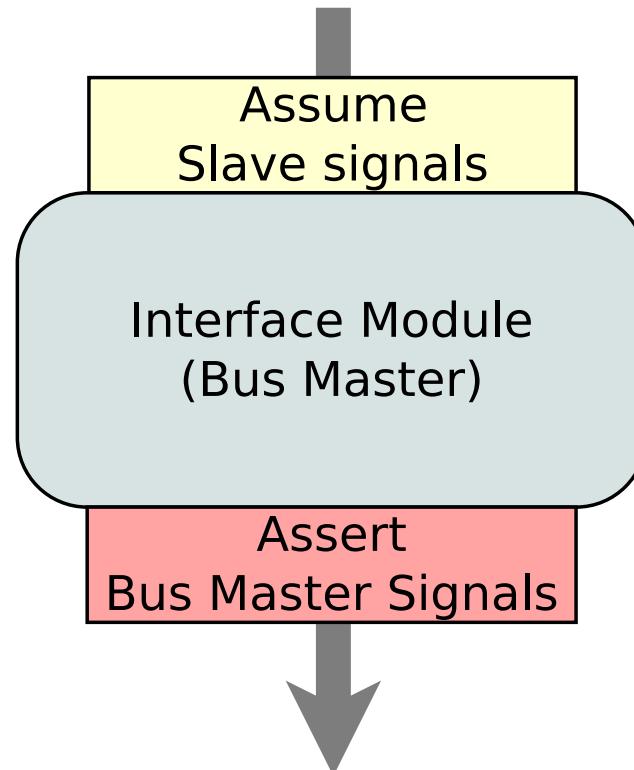
Quizzes

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?

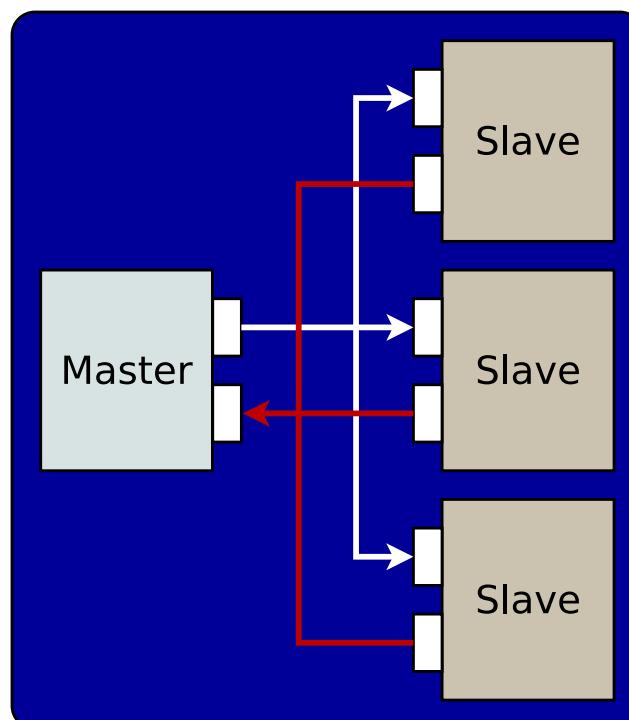


The slave's outputs are the master's inputs

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

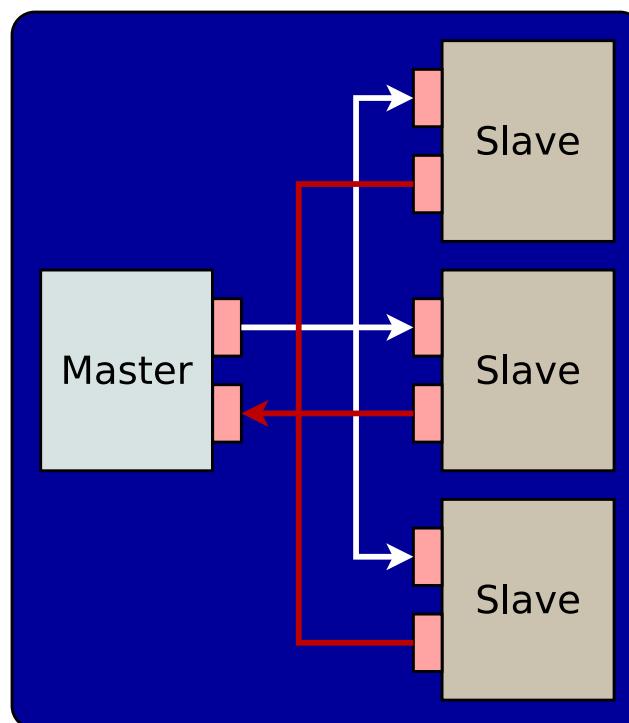
[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[▷ BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

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));
```

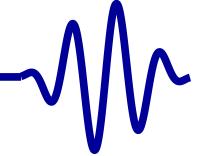
Remember ! (A&&B) is equivalent to (!A )||(! B)

- 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:

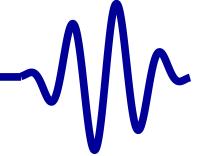
```
assign f_outstanding = (i_reset) ? 0  
      : f_nreqs - f_nacks;
```

- 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

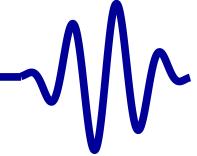
- Well, what if a request is being made now?

```
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

[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](#)

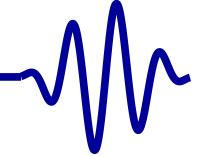
Want a guarantee that the cache response is consistent?

- 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

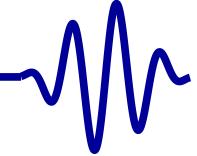
Quizzes

## Consider a multiply

- Just because an algorithm doesn't meet timing



# 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, 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](#)

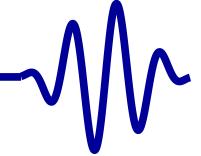
## Consider a multiply

- 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

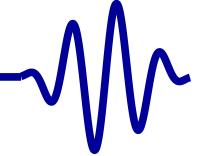
Cover

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

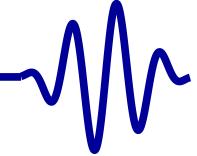
Sequences

Quizzes

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

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

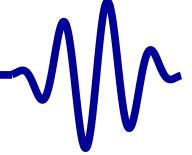
Abstraction offers alternatives

[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

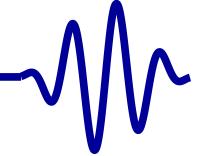
Cover

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
    // ...
  
```

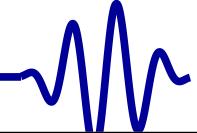
[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Let's work through a counter as an example.

- |                           |  |
|---------------------------|--|
| <code>exercise-01/</code> | Contains two files                         |
| <code>counter.v</code>    | This will be the HDL source 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

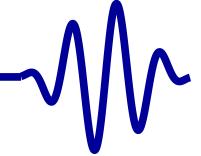
[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

```
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** ]

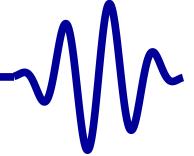
**read** –formal counter.v

# ... 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** ]

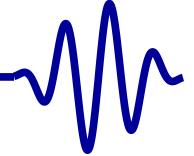
**read -formal counter.v**

# ... 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** ]

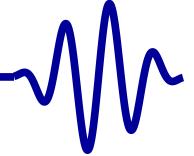
**read** –formal counter.v ← Read file

# ... 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 ]
```

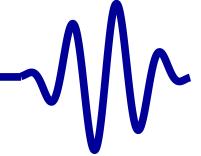
```
read -formal counter.v
```

```
# ... 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** ]

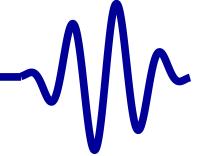
**read** –formal counter.v

# ... other files would go here

**prep** –top counter

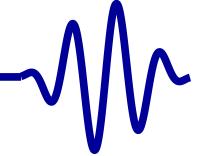
[ **files** ] ← List of files to be used

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

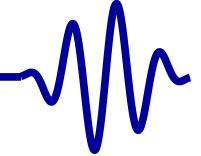
# ... other files would go here

**prep** -top counter

opt\_merge -share\_all

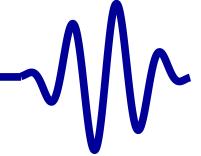
**[ files ]**

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](#)

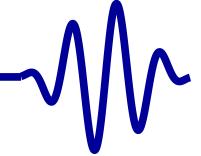
## Other usefull yosys commands

```
[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](#)

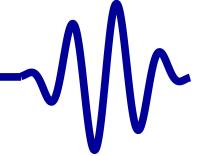
## Other usefull yosys commands

```
[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](#)

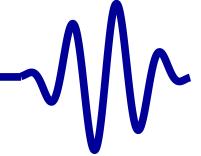
## Other usefull yosys commands

```
[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
```

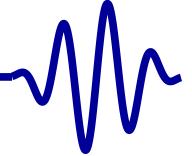
[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

## 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

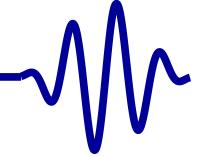
Sequences

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

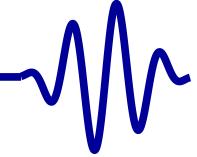
Sequences

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.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$
```

# 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

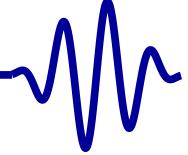
Quizzes

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

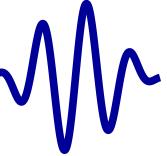
Cover

Sequences

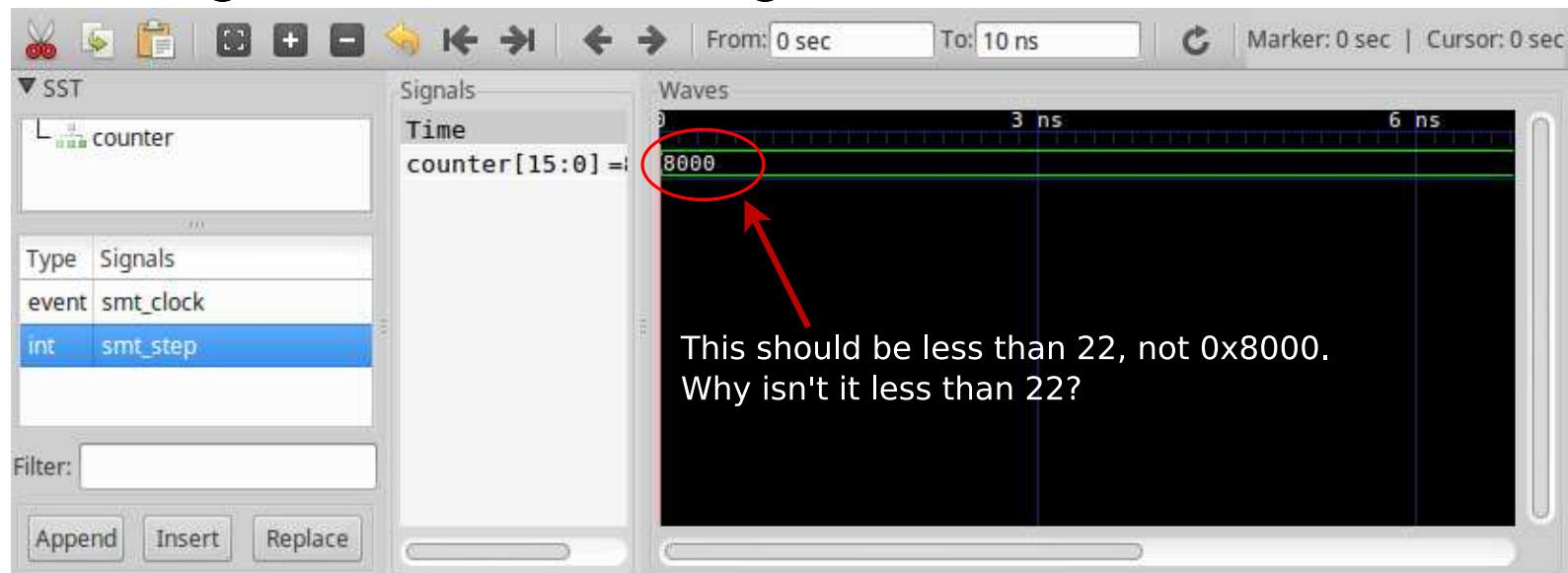
Quizzes

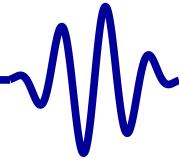
Look at source line 63, and fire up gtkwave

```
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/tr
ace.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](#)

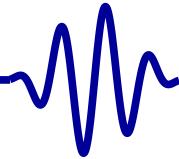
Run: % demo-rtl/counter.v

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

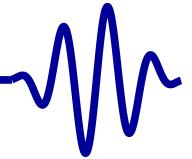
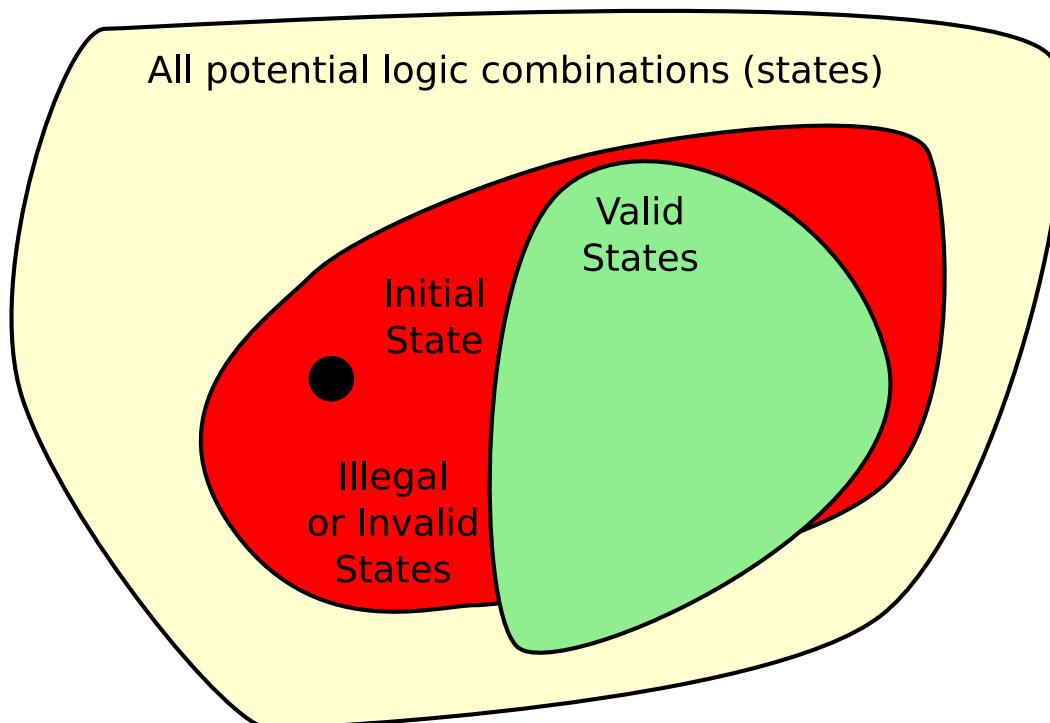
What did we do wrong?

```
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

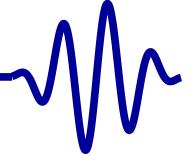
Did you notice the missing initial statement?

[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

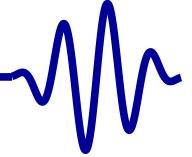
Multiple-Clocks

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

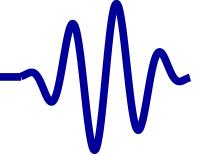
Sequences

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

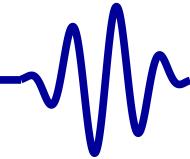
Cover

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

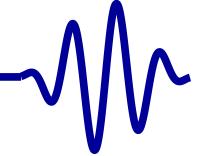
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[▷ Past](#)[\\$past Rule](#)[Past Assertions](#)[Past Valid](#)[Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- **\$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);
```

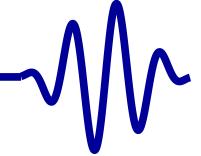
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[▷ Past Assertions](#)[Past Valid](#)[Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

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](#)

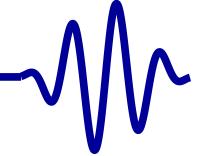
- This fails

```
always @(*)
```

```
assume (!i_start_signal);
```

```
always @ (posedge clk)
```

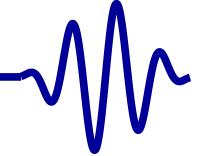
```
assert ($past(counter == 0));
```

[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);
```

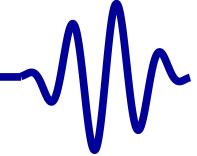
- Before time, counter is unconstrained.
- The solver can make it take on any value it wants in order to make things fail
- This will not show in the VCD file

[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?

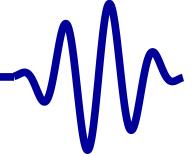
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[▷ Past Assertions](#)[Past Valid](#)[Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

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.

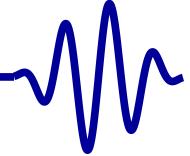
- `i_start_signal` is unconstrained before time
- `counter` is initially constrained to zero
- If `i_start_signal` is one before time,  
`counter` will still be zero when time begins

[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](#)

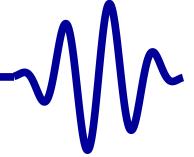
- What about the case where `i_start_signal` is raised while the counter isn't zero?

```
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 designs



# 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

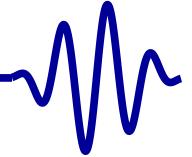
Multiple-Clocks

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);
```

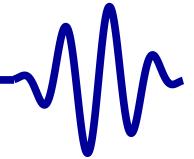
```
always @(*)
  if (!f_past_valid)
    assume(i_RESET);
```

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
```

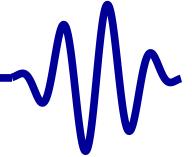
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[Past Assertions](#)[Past Valid](#)[Examples](#)[Ex: Busy](#)[▷ Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

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

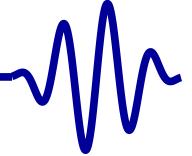
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[Past Assertions](#)[Past Valid](#)[Examples](#)

Ex: Busy  
▷ Counter

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

Here's the basic design. 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 design 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

Make sure you check `o_busy` both when `counter == 0` and `counter != 0`

*This requires an assertion*

4. If the counter is non-zero, it should always be counting down

Beware of the reset!

*This requires another assertion*

[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](#)

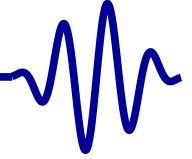
### Exercise:

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

▷  $k$  Induction

Lesson Overview

vs BMC

General Rule

The Trap

Results

Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

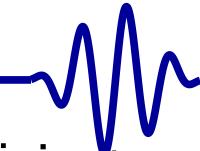
Sequences

Quizzes

# $k$ Induction



# Lesson Overview



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

▷ Lesson Overview

vs BMC

General Rule

The Trap

Results

Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

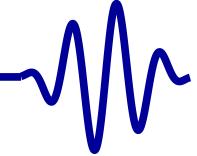
Quizzes

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](#)[Results](#)[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

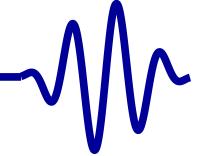
$$\sum_{n=0}^{N-1} x^n = x^0 = \frac{1-x}{1-x}$$

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

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



# Proof, continued



Welcome

Motivation

Basics

Clocked and \$past

k Induction

▷ Lesson Overview

vs BMC

General Rule

The Trap

Results

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  $N + 1$

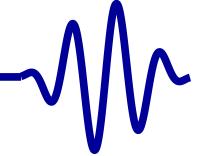
$$\sum_{n=0}^N x^n = x^N + \sum_{n=0}^{N-1} x^n = x^N + \frac{1-x^N}{1-x}$$

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

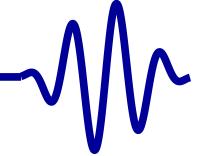
Suppose  $\forall n : P[n]$  is what we wish to prove

- 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](#)[Results](#)[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

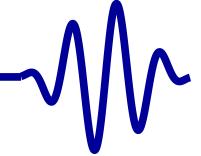
- Base case: show  $P[0]$
  - Inductive case: show  $P[n] \rightarrow P[n + 1]$

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

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

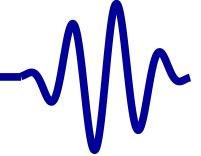
- Traditional induction

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

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

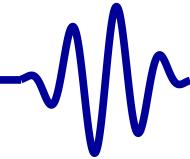
- Traditional induction

- 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](#)[Results](#)[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.*

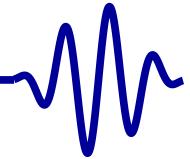
This is the BMC pass we've already done.

- **Inductive Step:**

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

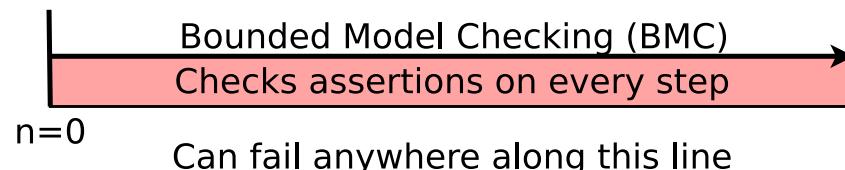
*Assume the  $N + 1$  step violates no assumptions, . . .*

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

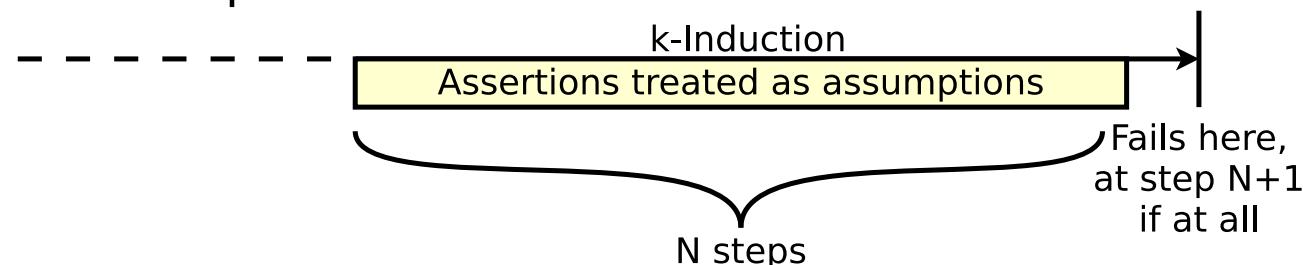
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[▷ vs BMC](#)[General Rule](#)[The Trap](#)[Results](#)[Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

BMC and induction are very different.

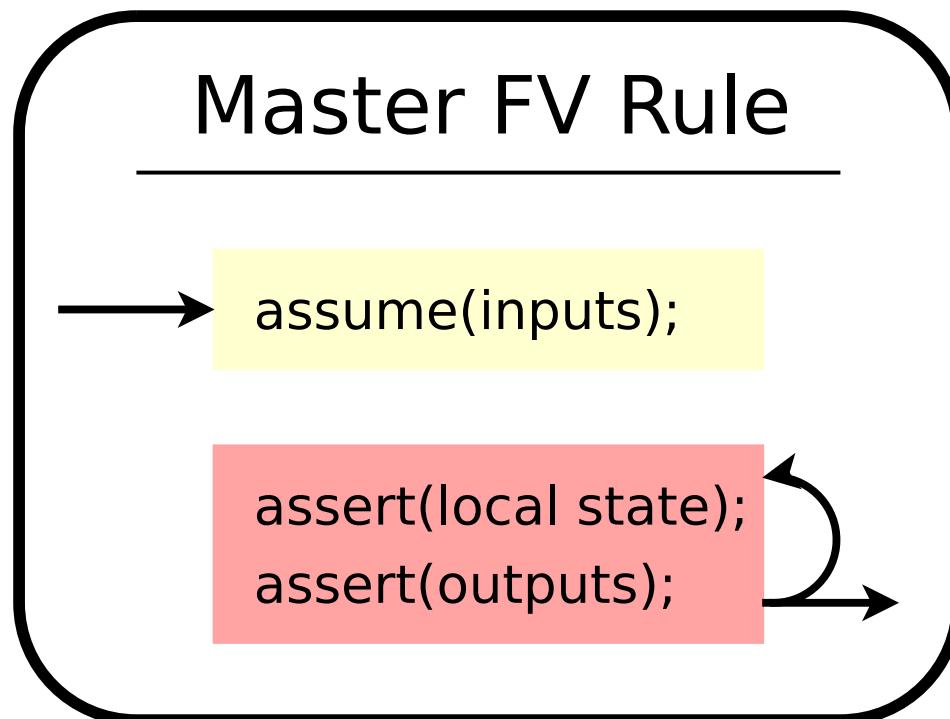
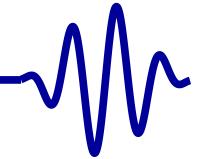
- BMC, the base case



- Induction step



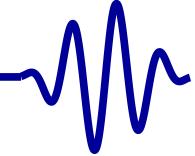
- The number of BMC time-steps must be more than the number of inductive time-steps
- Register values at the beginning of the inductive step can be *anything* allowed by your assertions and assumptions
- This is where the work takes place.



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.



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Lesson Overview

vs BMC

▷ General Rule

The Trap

Results

Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

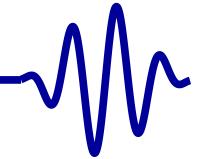
Sequences

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

Results

Examples

Bus Properties

Free Variables

Abstraction

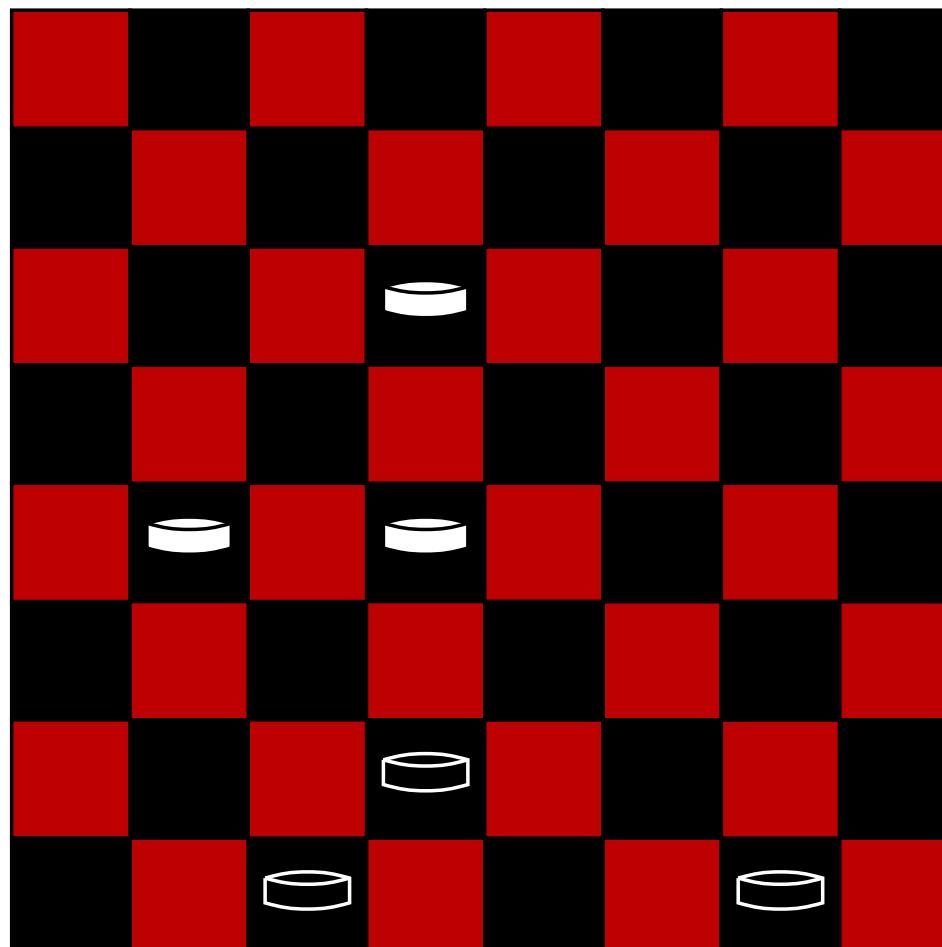
Invariants

Multiple-Clocks

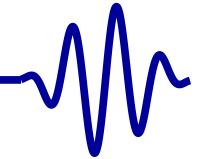
Cover

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

Results

Examples

Bus Properties

Free Variables

Abstraction

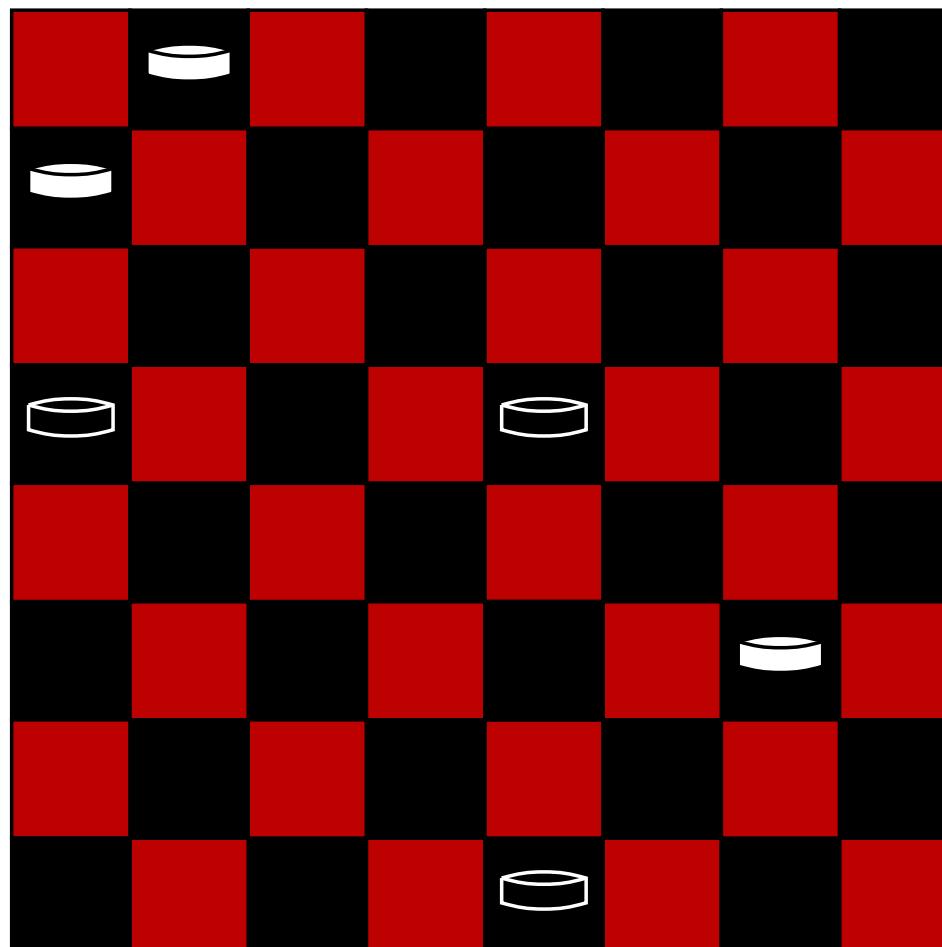
Invariants

Multiple-Clocks

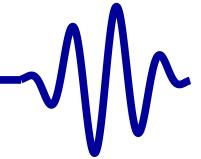
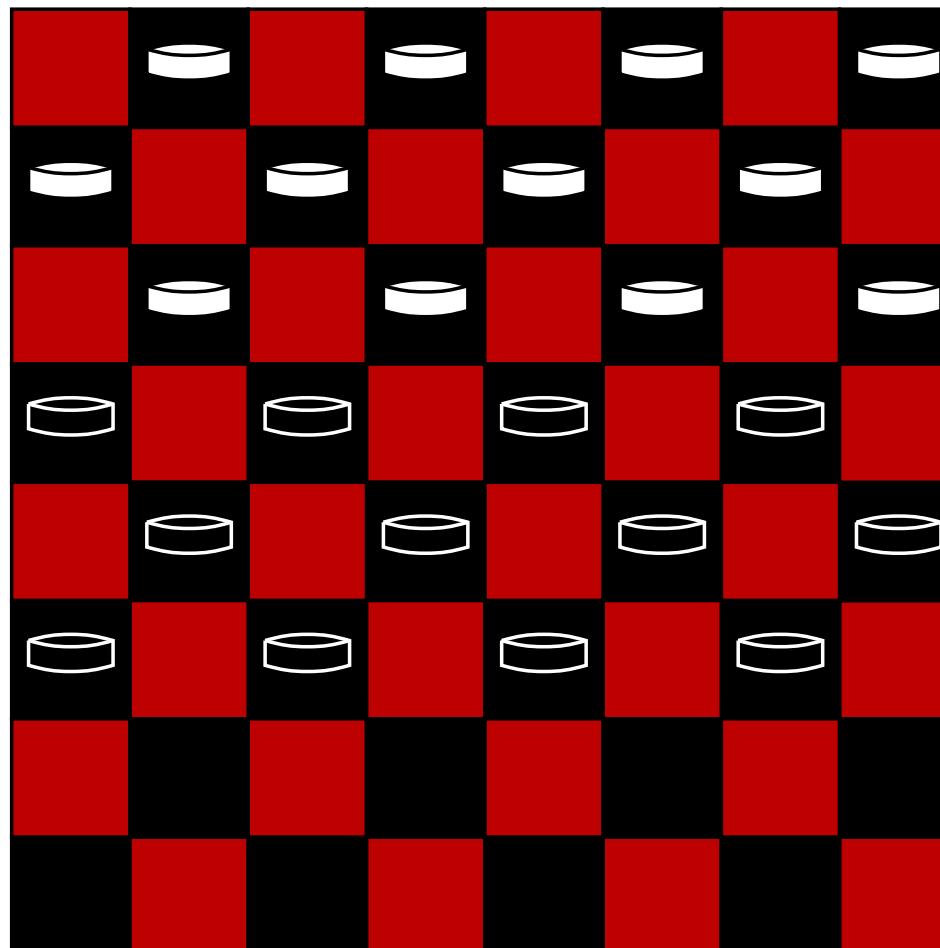
Cover

Sequences

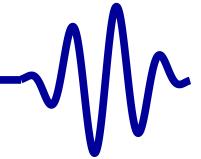
Quizzes



White's going to move and win

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[▷ General Rule](#)[The Trap](#)[Results](#)[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

Results

Examples

Bus Properties

Free Variables

Abstraction

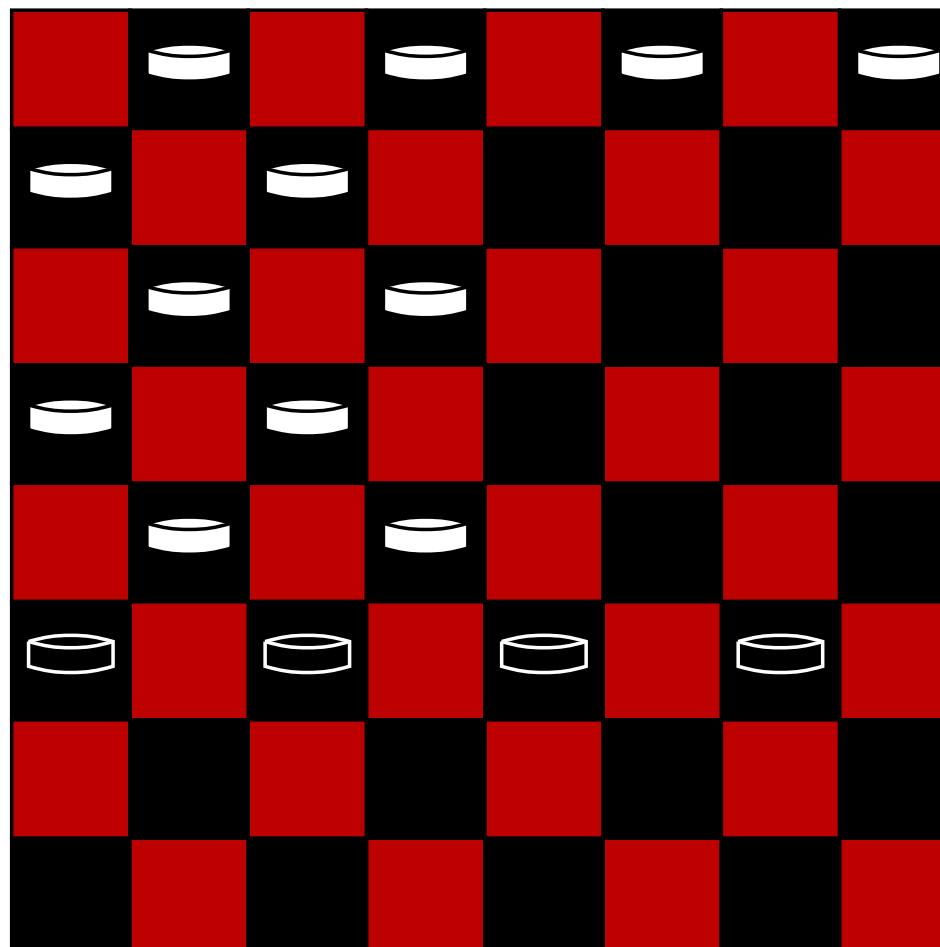
Invariants

Multiple-Clocks

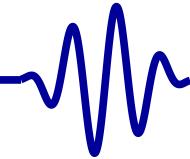
Cover

Sequences

Quizzes

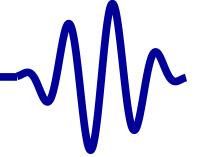


Would this pass our criteria?

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[▷ General Rule](#)[The Trap](#)[Results](#)[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



1

Welcome

MotivationBasicsClocked and \$pastk Induction

Lesson Overview

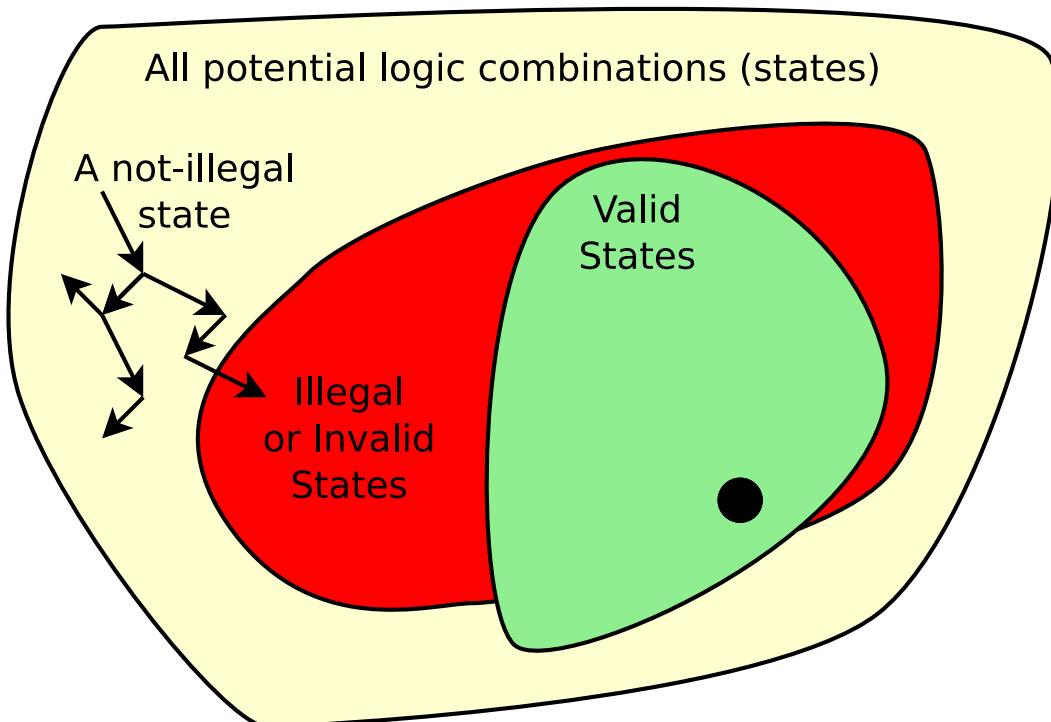
vs BMC

General Rule

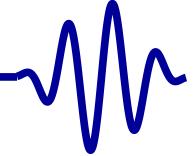
▷ The Trap

Results

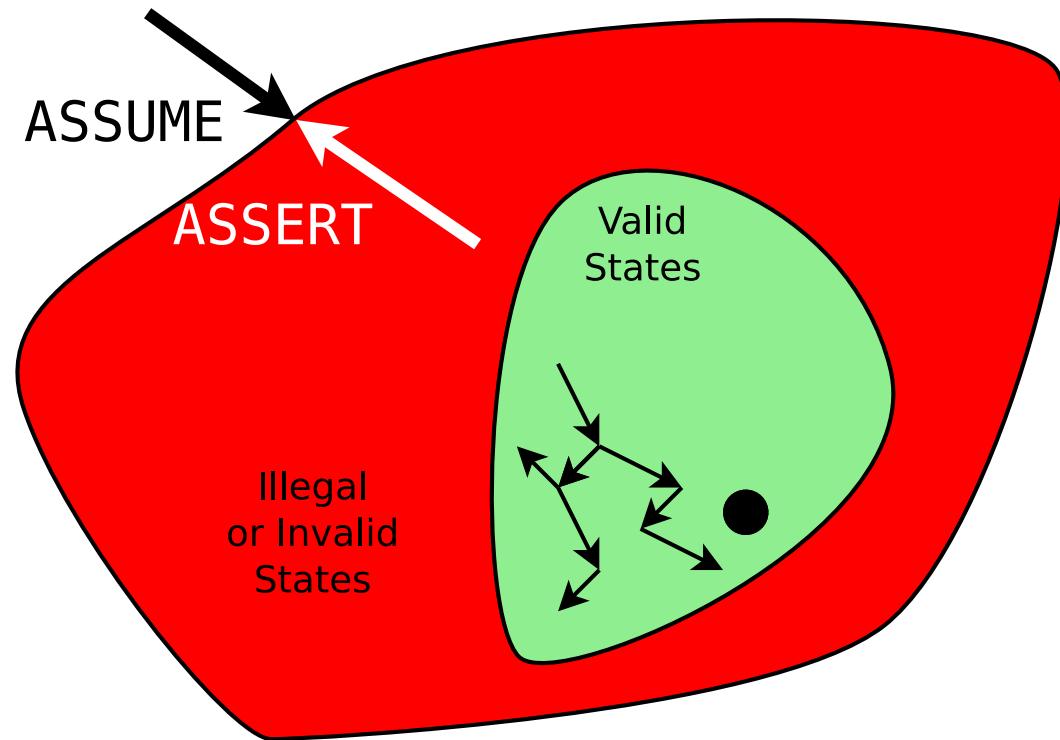
Examples

Bus PropertiesFree VariablesAbstractionInvariantsMultiple-ClocksCoverSequencesQuizzes

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



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



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

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

Unlike BMC, the results of induction might be inconclusive

$k$ Induction	BMC	
	FAIL	PASS
	FAIL	Design Fails
	UNKNOWN	SUCCESS!

The  $k$  induction pass will fail if your design doesn't have enough assertions.

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

There's also a difference in when BMC and induction finish

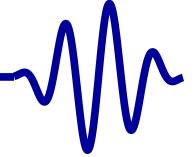
- BMC will finish early if the design FAILs
- Induction will finish early if the design PASSes
- In all other cases, they will take a full depth steps

You can use this fact to trim the depth of your proof

- Once induction succeeds, trim your proof depth to that length
- This will immediately make your proof run that much faster



# Examples



Welcome

Motivation

Basics

Clocked and \$past

*k* Induction

Lesson Overview

vs BMC

General Rule

The Trap

Results

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

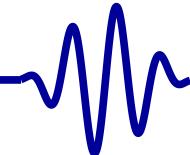
Multiple-Clocks

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](#)[Results](#)[▷ 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](#)[Results](#)[▷ 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](#)[Results](#)[▷ Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

With one simple change, this design will now pass 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'd22);
```

See the difference?

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

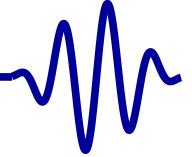
This design is almost identical to the last one, yet fails induction. The key difference is the **if** (`i_ce`).

```
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

Results

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

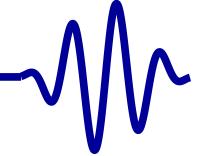
Quizzes

Several approaches to fixing this:

1. **assume(i\_ce);**



# Fixing Shift Reg



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Lesson Overview

vs BMC

General Rule

The Trap

Results

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

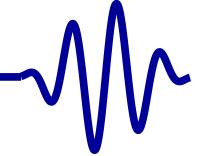
Quizzes

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

Results

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

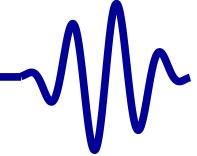
Cover

Sequences

Quizzes

Several approaches to fixing this:

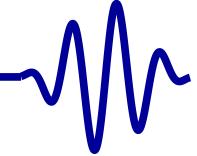
1. **assume(i\_ce);**  
*Doesn't really test the design*
2. opt\_merge –share\_all, yosys option  
*Works for some designs*
3. **assert(sa == sb);**

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[Results](#)[▷ 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

Results

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

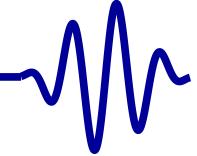
Several approaches to fixing this:

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 suprove

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

Several approaches to fixing this:

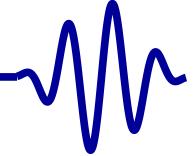
1. **assume(i\_ce);**  
*Doesn't really test the design*
2. opt\_merge –share\_all, yosys option  
*Works for some designs*
3. **assert(sa == sb);**  
*Best, but only works when sa and sb are visible*
4. Insist on no more than  $M$  clocks between i\_ce's
5. Use a different prover, under the [**engines**] option
  - smtbmc                    **Inconclusive Proof (Induction fails)**
  - abc pdr                **Pass**
  - aiger suprove        **Pass**

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[Results](#)[▷ 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                    **Inconclusive Proof (Induction fails)**
  - abc pdr                **Pass**
  - aiger suprove        **Pass**

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](#)[Results](#)[▷ 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

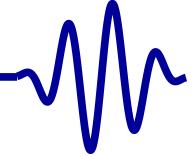
# ... *other files would go here*

**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](#)[Results](#)[▷ 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

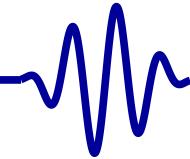
# ... other files would go here

**prep** –top module

**opt\_merge** –share\_all

[ **files** ]

.. / path-to/module.v

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[Results](#)[▷ 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
```

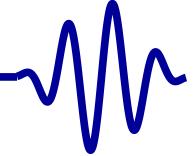
```
# ... other files would go here
```

```
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](#)[Results](#)[▷ 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
```

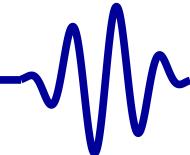
```
# ... other files would go here
```

```
prep -top module
```

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

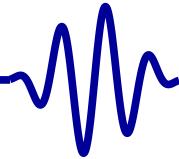
```
[ files ]
```

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

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

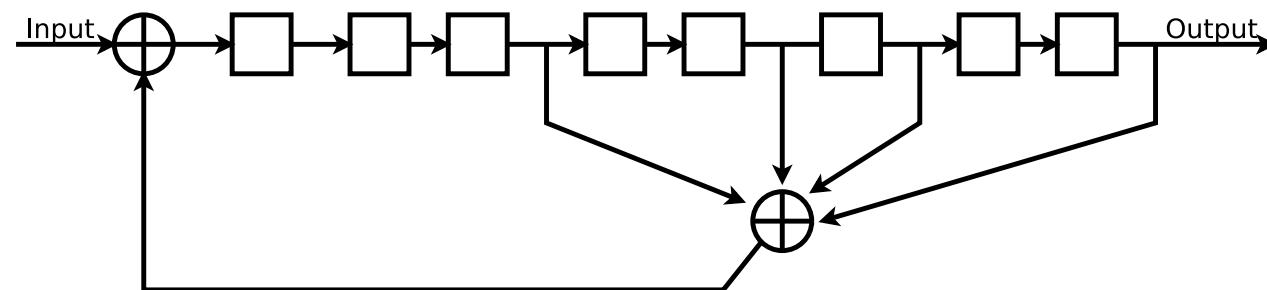
## Exercise #4: dblpipe.v

```
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
```

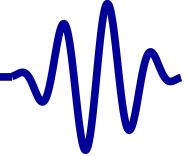
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[Results](#)[▷ Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

### 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](#)[Results](#)[▷ Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

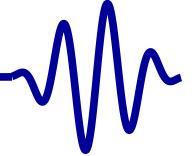
## Exercise #4: dblpipe.v, lfsr\_fib.v

```
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

Results

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

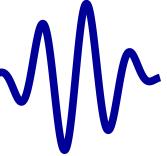
Cover

Sequences

Quizzes

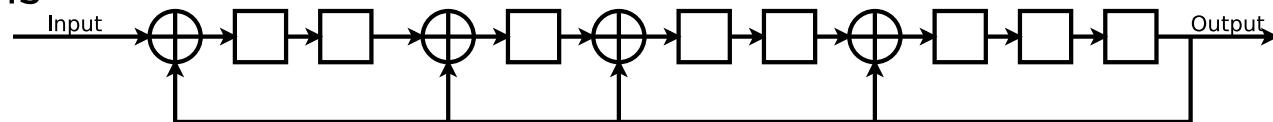
## Exercise #4:

- Using dblpipe.v
  - Prove that the output, o\_data, is zero

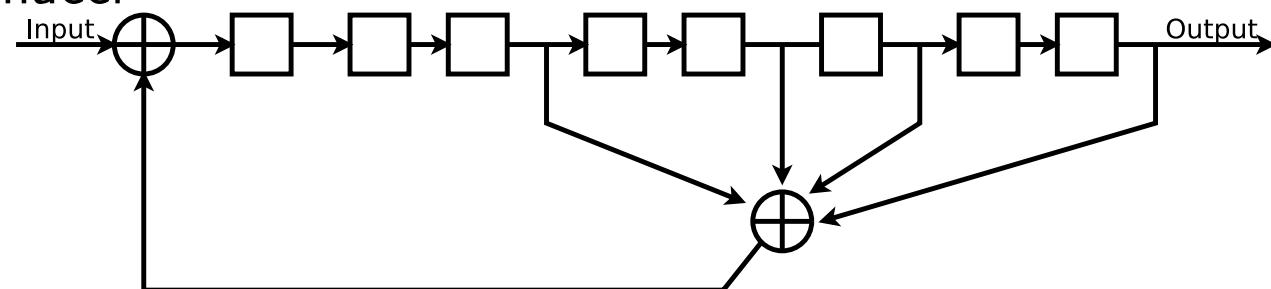
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[Results](#)[▷ Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Galois and Fibonacci are supposedly identical

- Galois



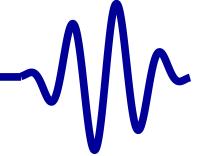
- Fibonacci



- 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

Results

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

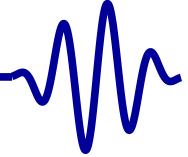
Quizzes

## 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

Results

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

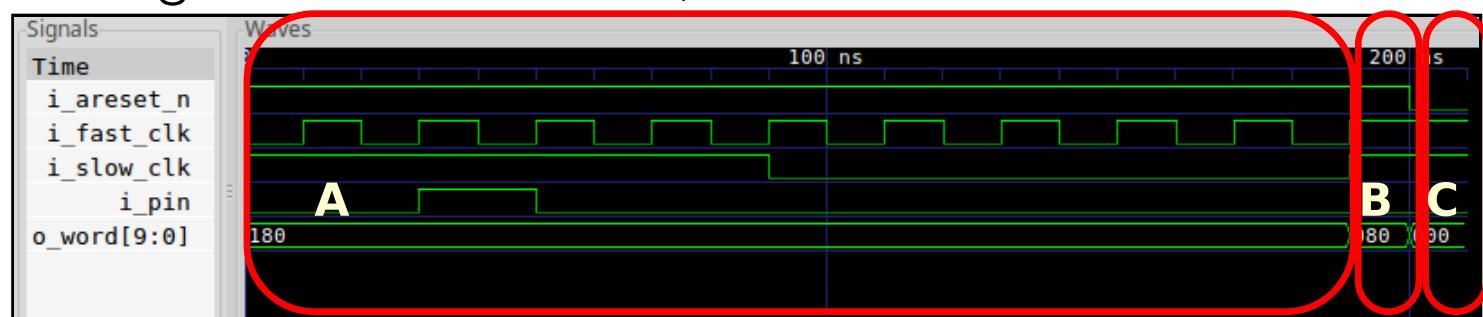
Multiple-Clocks

Cover

Sequences

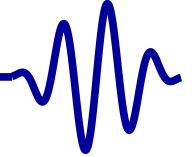
Quizzes

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

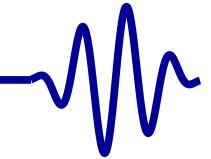
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[▷ Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

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

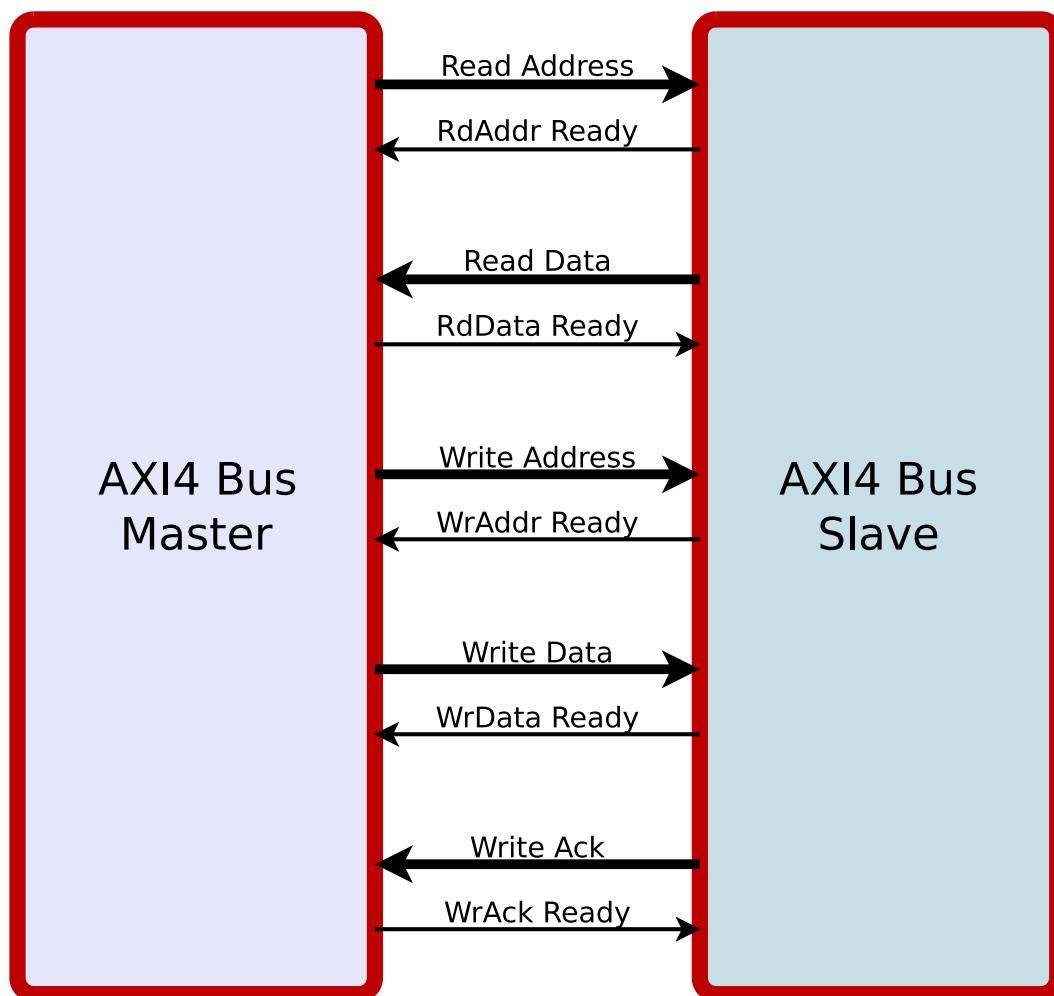
▷ AXI

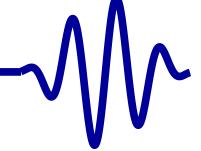
Avalon

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

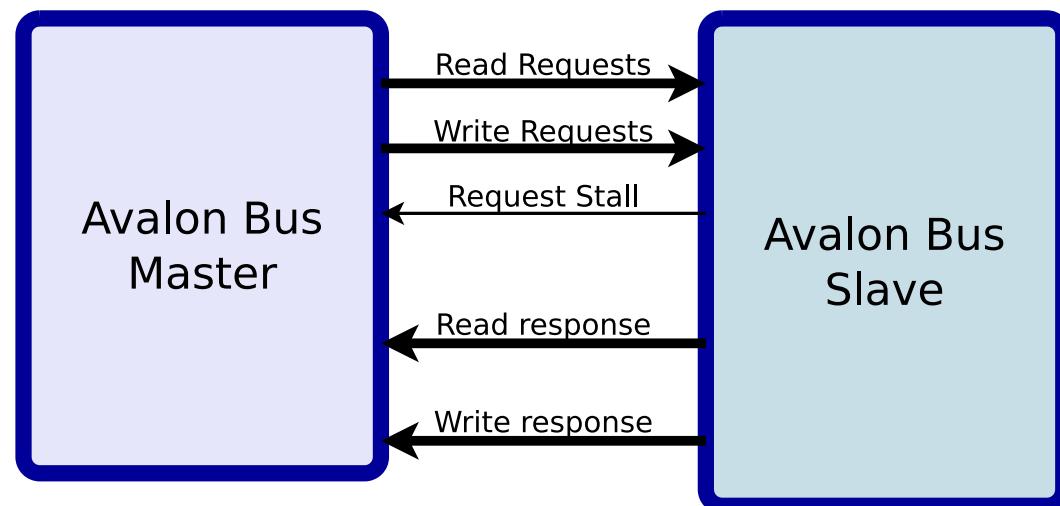
Invariants

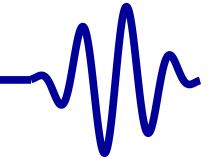
Multiple-Clocks

Cover

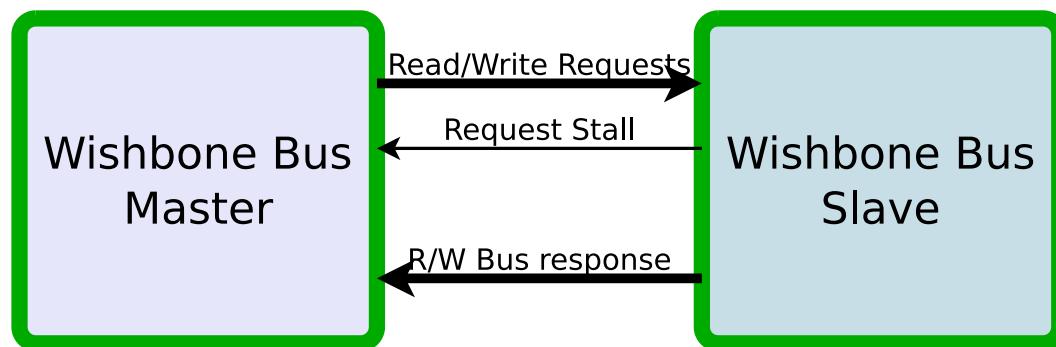
Sequences

Quizzes

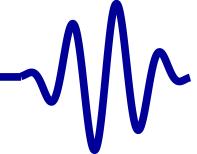




- Welcome
- Motivation
- Basics
- Clocked and \$past
- $k$  Induction
- Bus Properties
  - Ex: WB Bus
  - AXI
  - Avalon
  - ▷ Wishbone
  - WB Basics
  - WB Basics
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Cover
- Sequences
- Quizzes

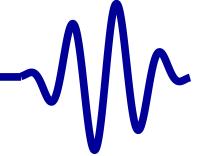


- Why use the Wishbone? *It's simpler!*

[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

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

Ex: WB Bus

AXI

Avalon

Wishbone

▷ WB Basics

WB Basics

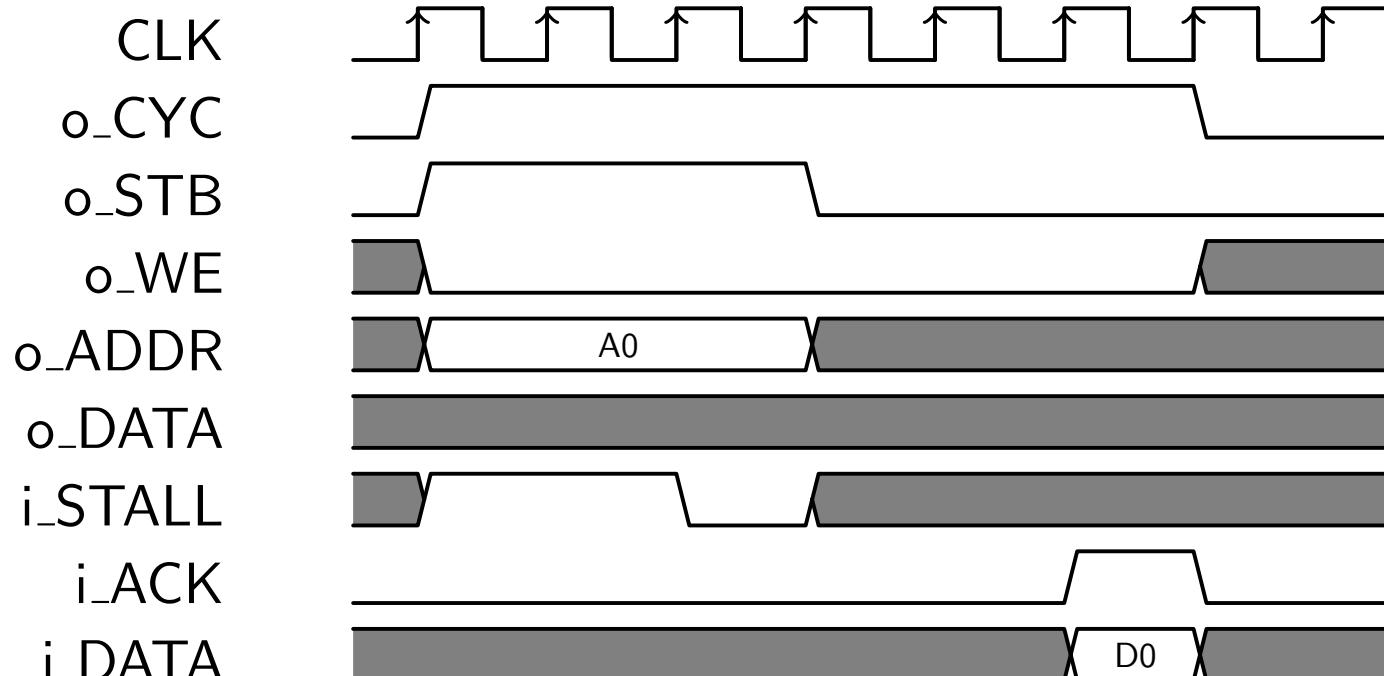
[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

From the slave's perspective:

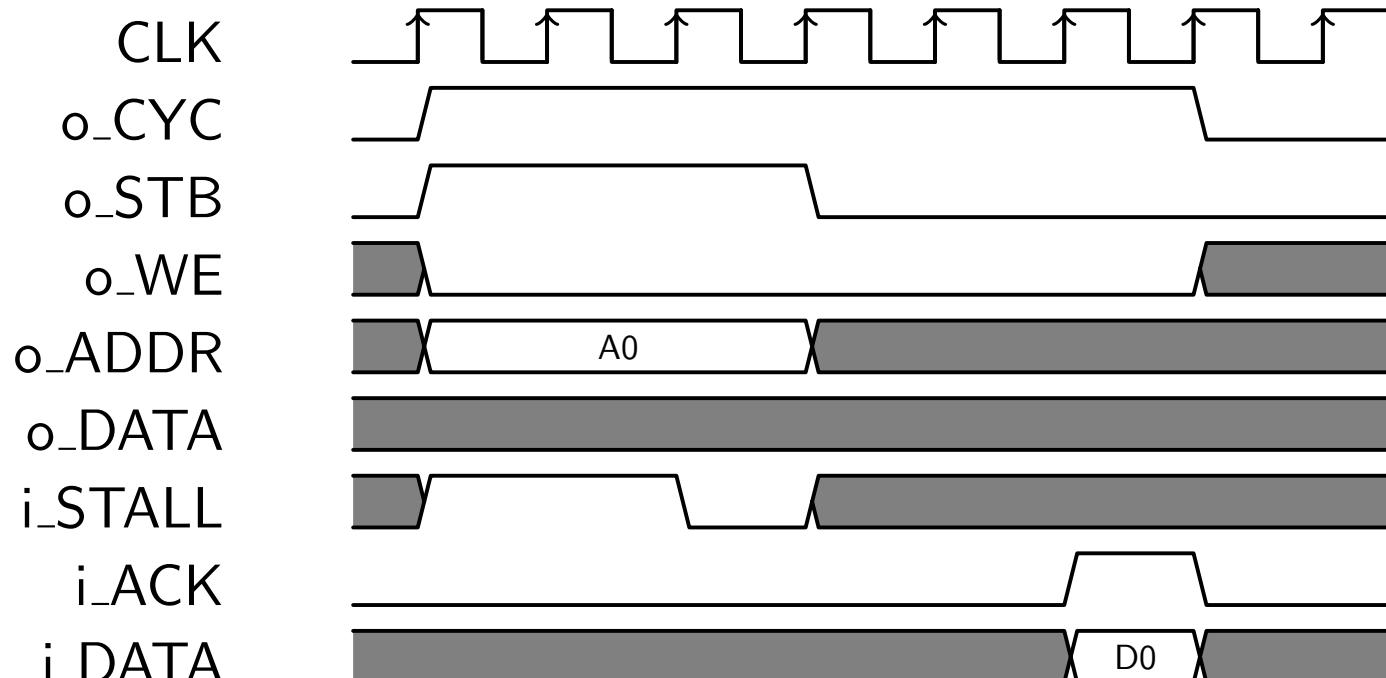
Specification name	My name
CYC_I	i_wb_cyc
STB_I	i_wb_stb
WE_I	i_wb_we
ADDR_I	i_wb_addr
DATA_I	i_wb_data
SEL_I	i_wb_sel
STALL_O	o_wb_stall
ACK_O	o_wb_ack
DATA_O	o_wb_data
ERR_O	o_wb_err

To swap perspectives from master to slave ...

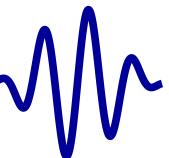
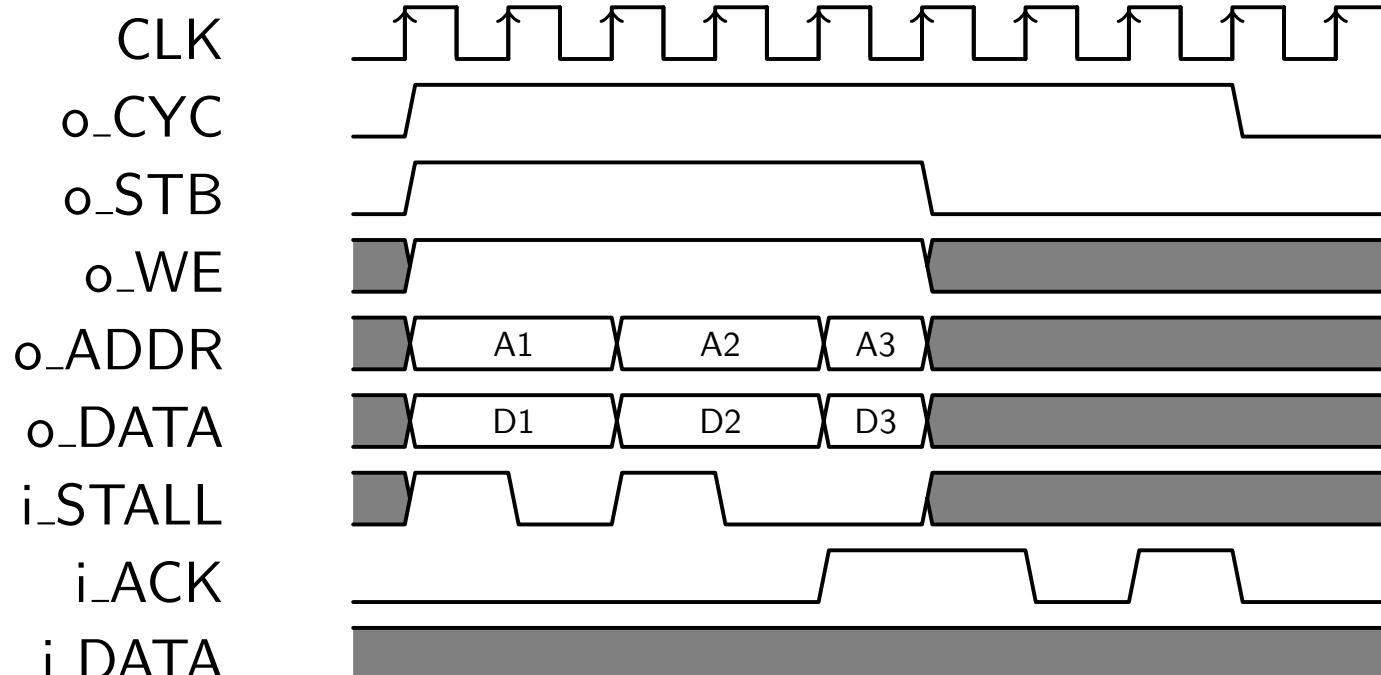
- Swap the port direction
- Swap the **assume()** statements for **assert()**s

[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

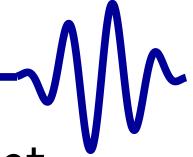
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[▷ WB Basics](#)[WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

- 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

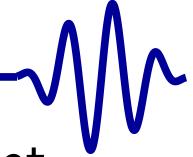
Sequences

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
```

# 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
```

- 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));
```

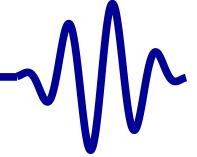
- Did we get it? Well, not quite  
o\_data is a don't care for any read request

[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
```



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

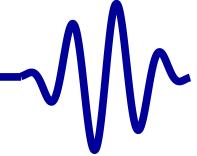
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

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

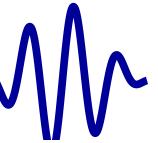
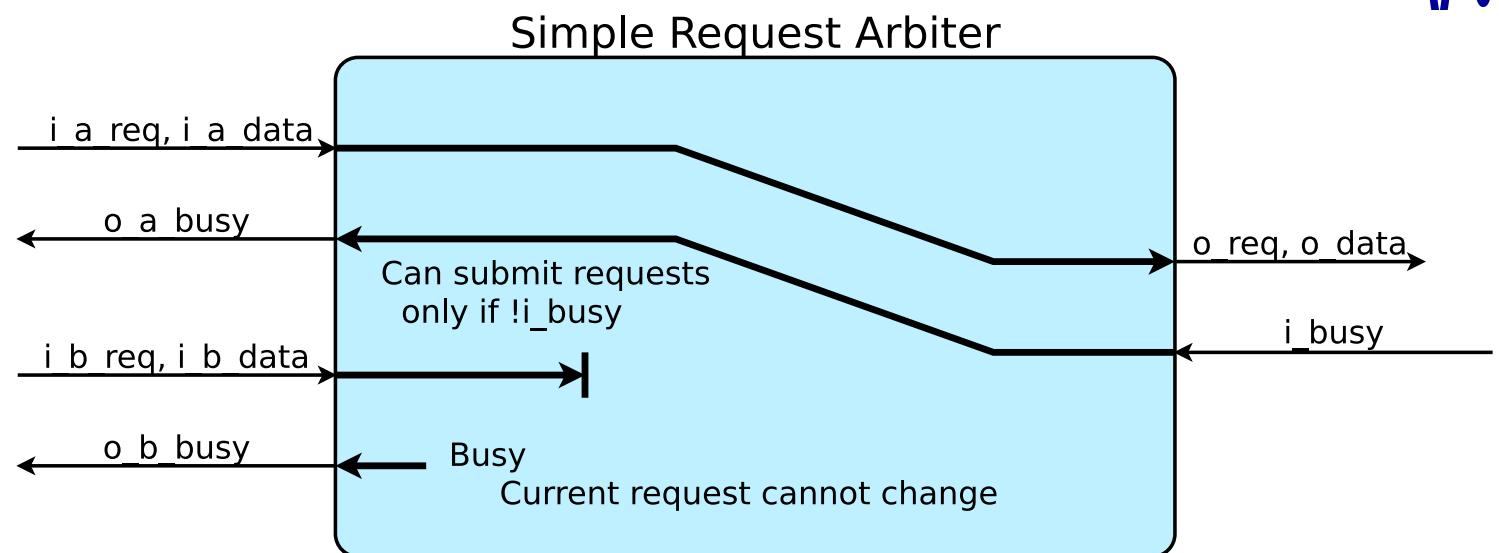
```
initial f_nreqs = 0;
always @(posedge clk)
if ((i_reset)||(!i_CYC))
    f_nreqs <= 1'b0;
else if ((i_STB)&&(!o_STALL))
    f_nreqs <= f_nreqs + 1'b1;
// Similar counter for acknowledgements
always @(*)
if (f_nreqs == f_nacks)
    assert (!o_ACK);
```

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

[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](#)

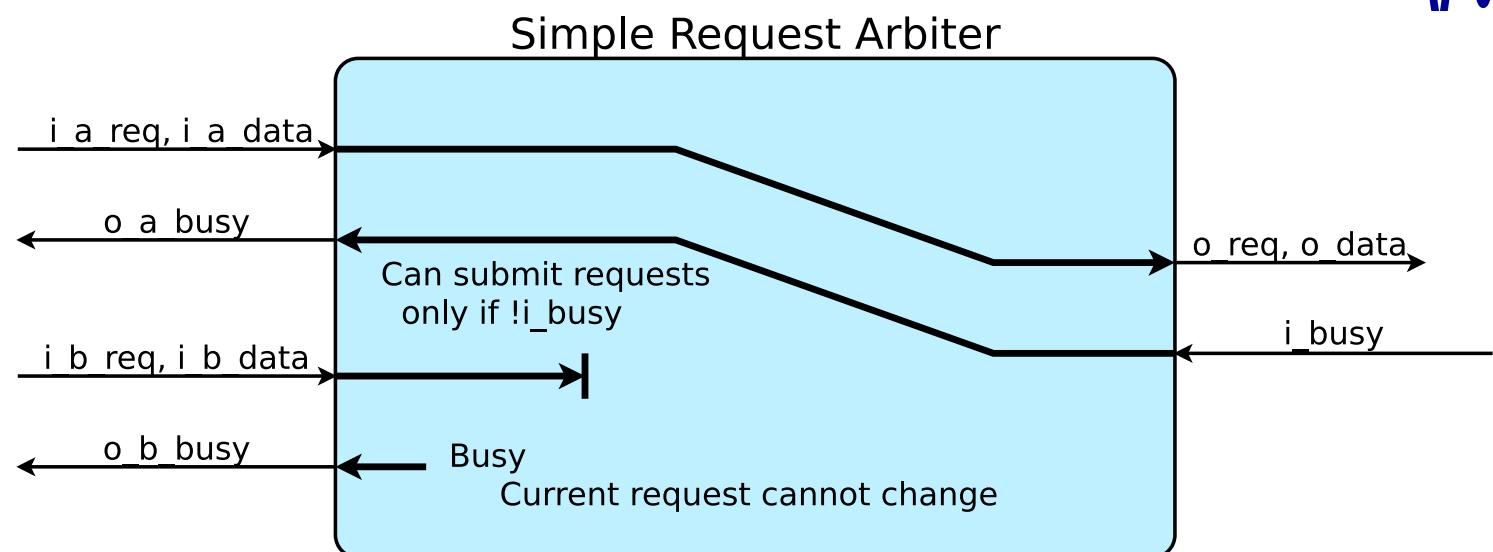
Let's build up to proving a WB arbiter

- 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`

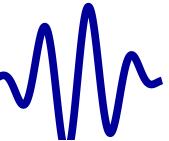
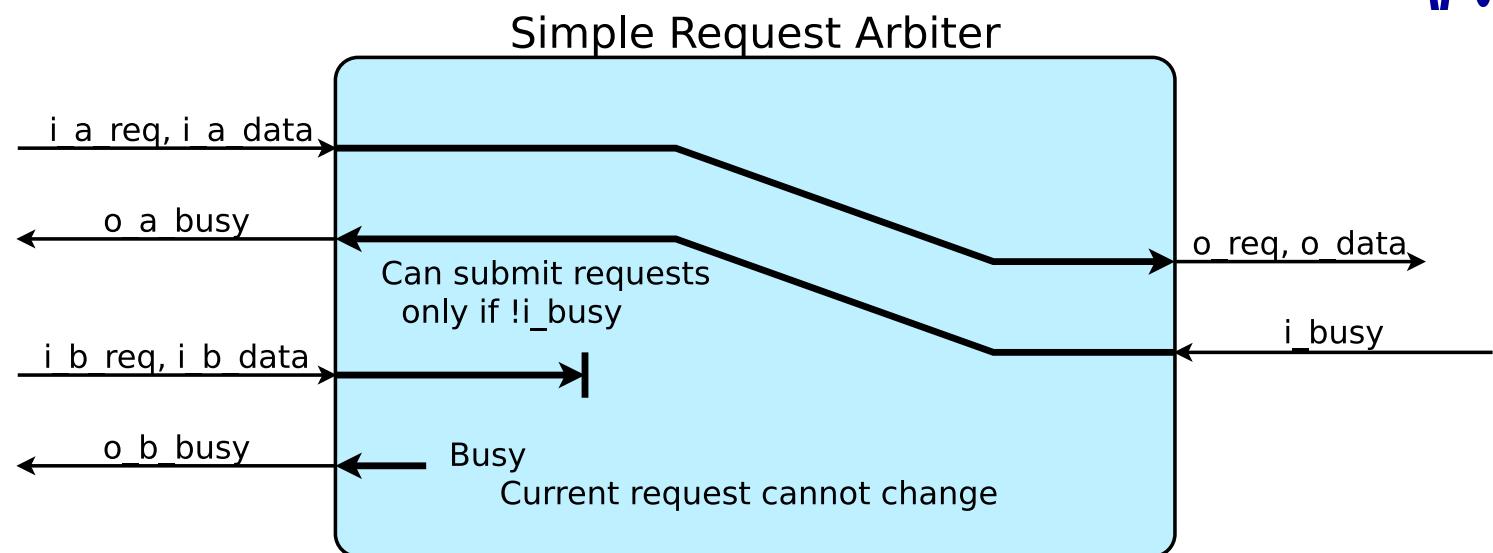
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

## The basics

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

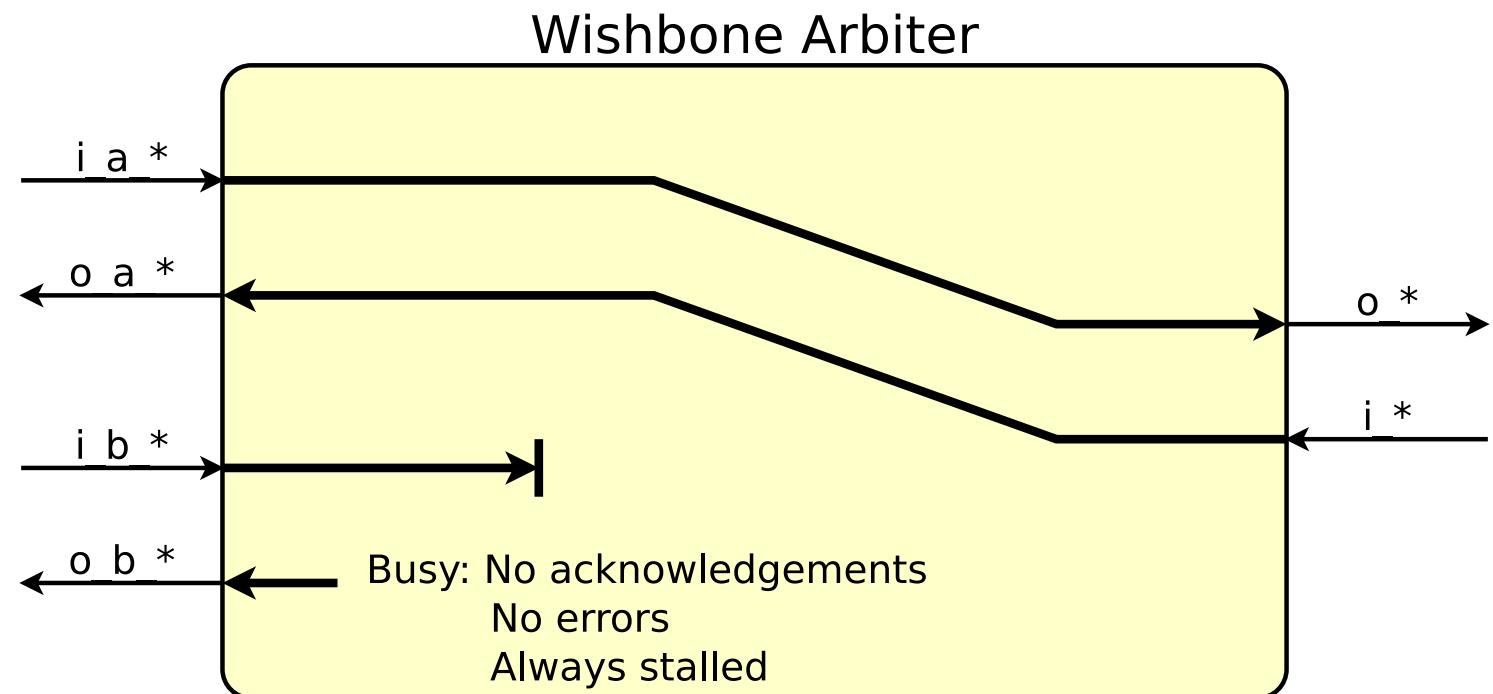
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

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

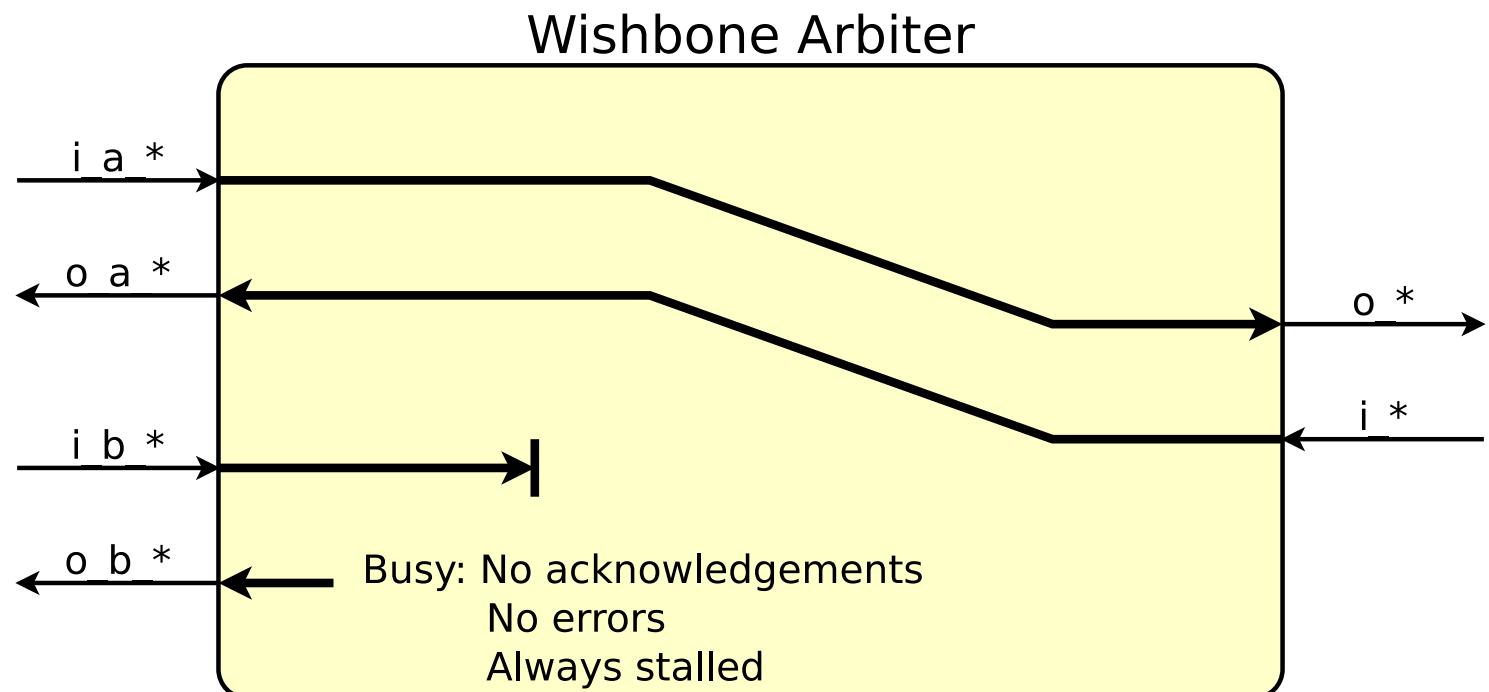
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

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

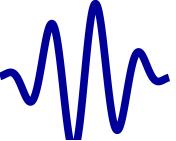
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Shall we try this with Wishbone?

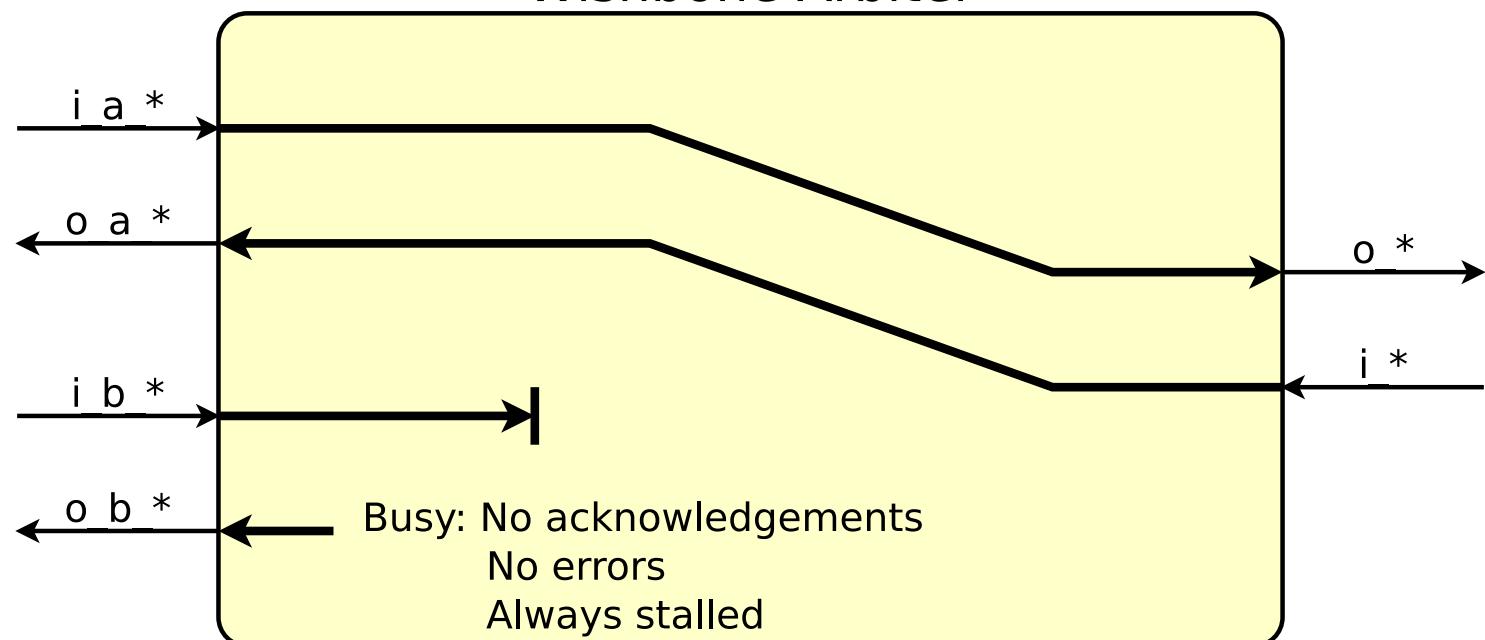
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

This request side is almost identical

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

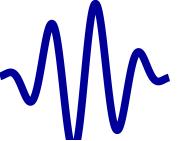
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

## Wishbone Arbiter

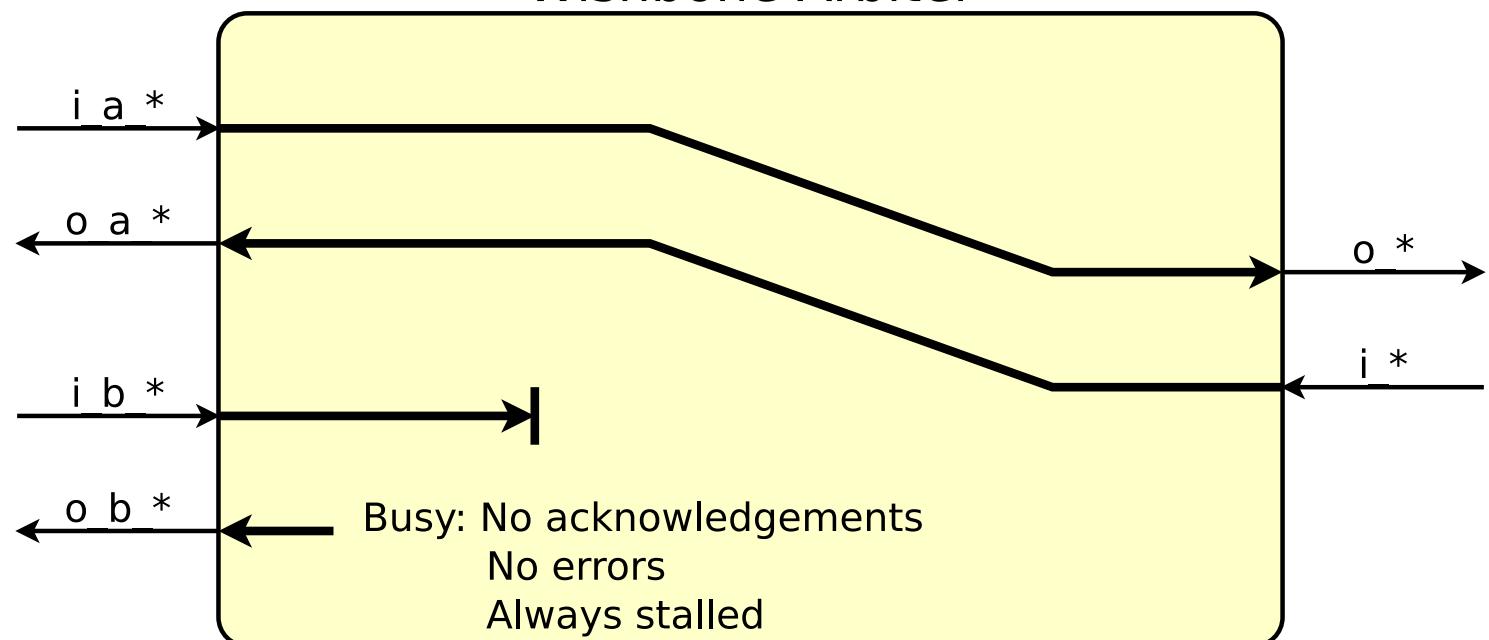


The difference is the acknowledgements

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

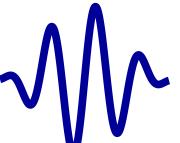
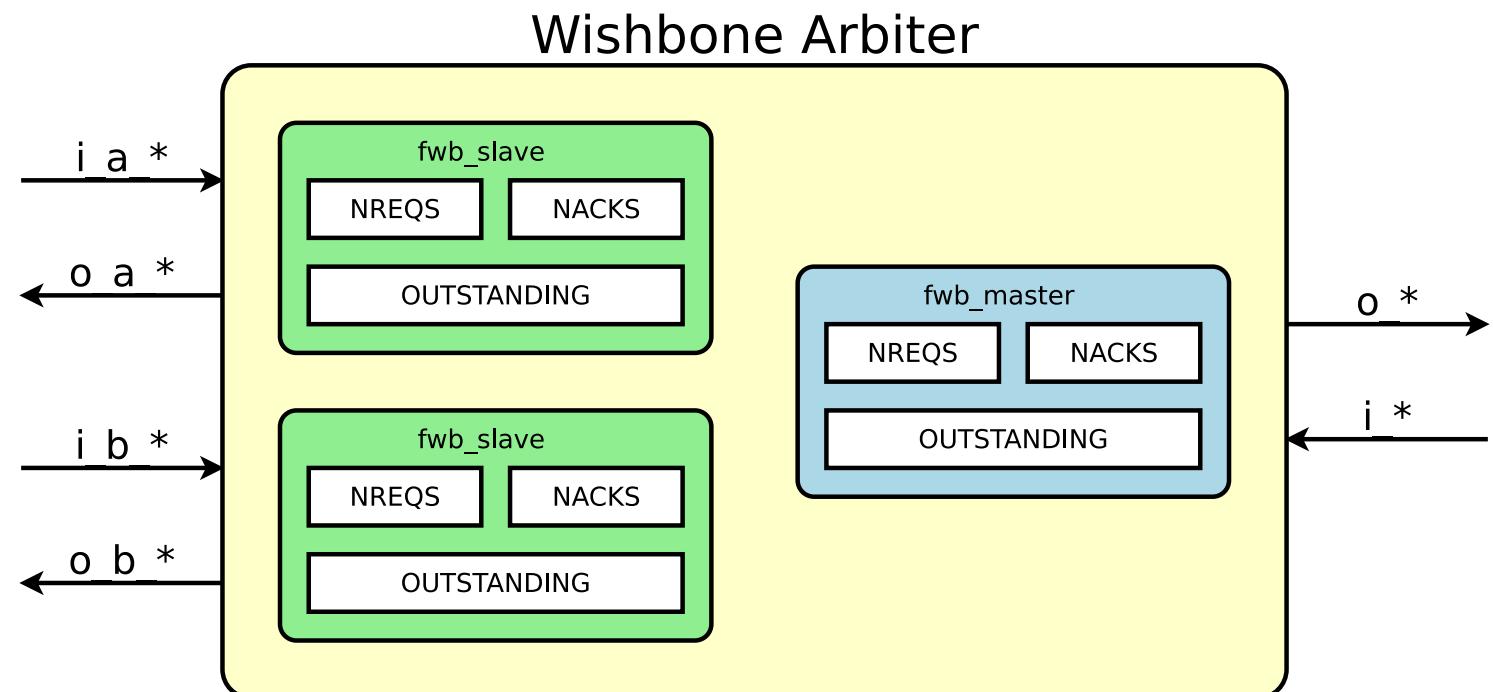
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

## Wishbone Arbiter



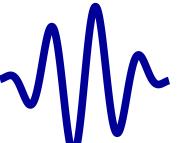
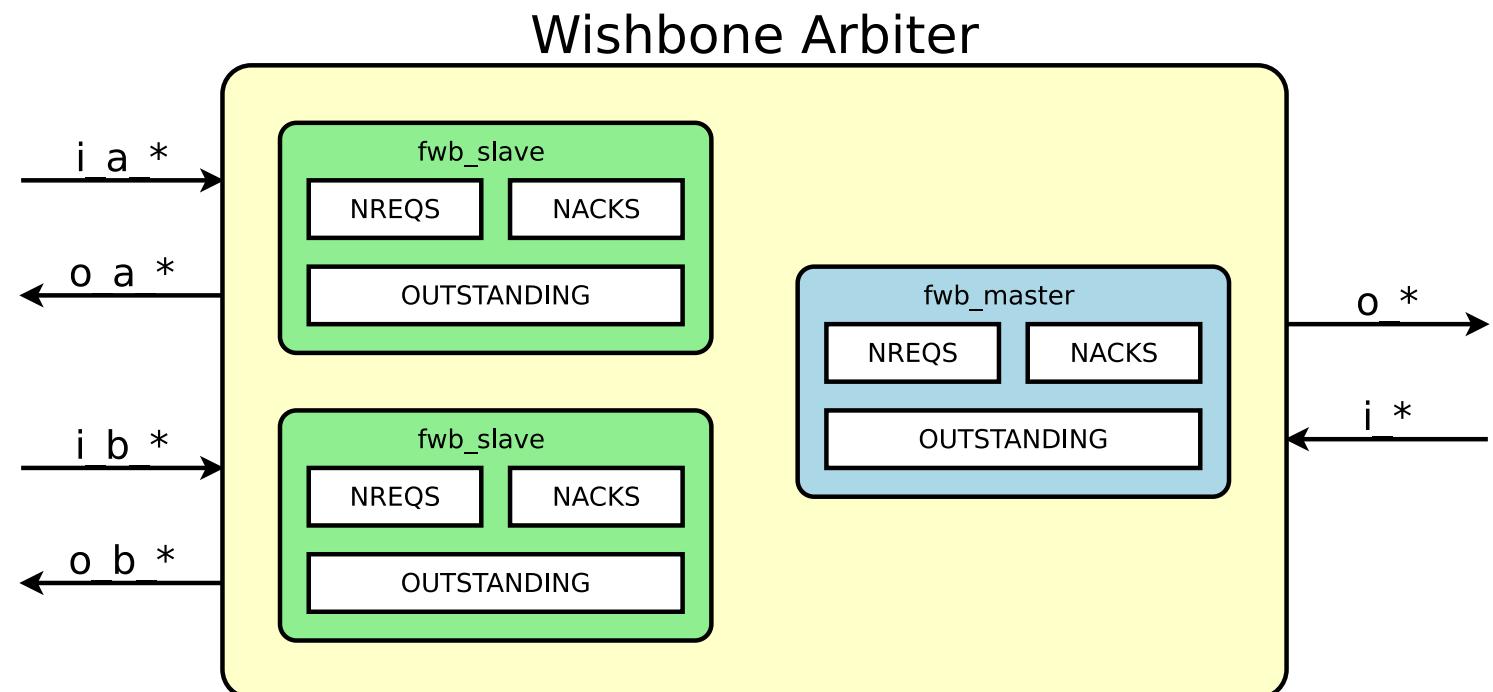
Now, prove that `exercise-07/wbpriarbiter.v` works.

- Use both BMC and  $k$ -induction (mode 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](#)

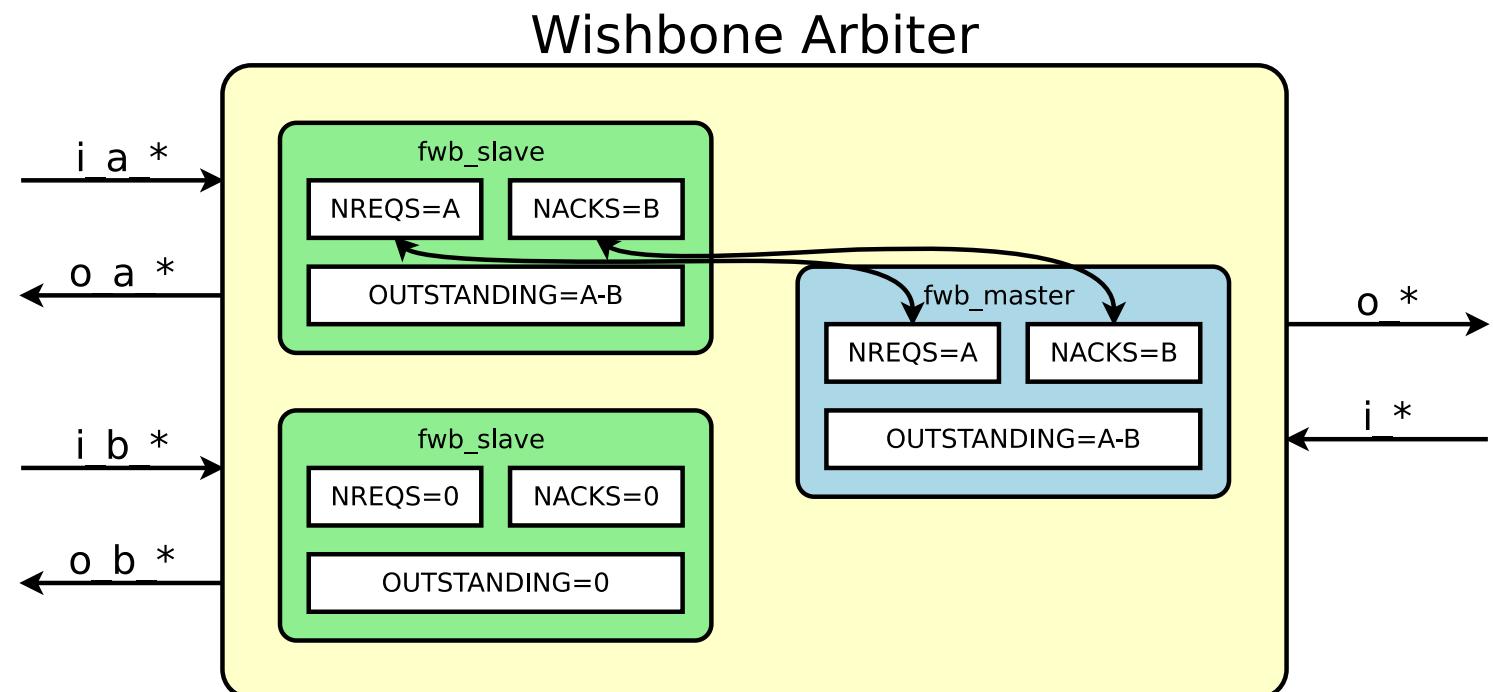
The `fwb_slave.v` properties will

- 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](#)

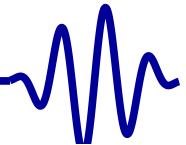
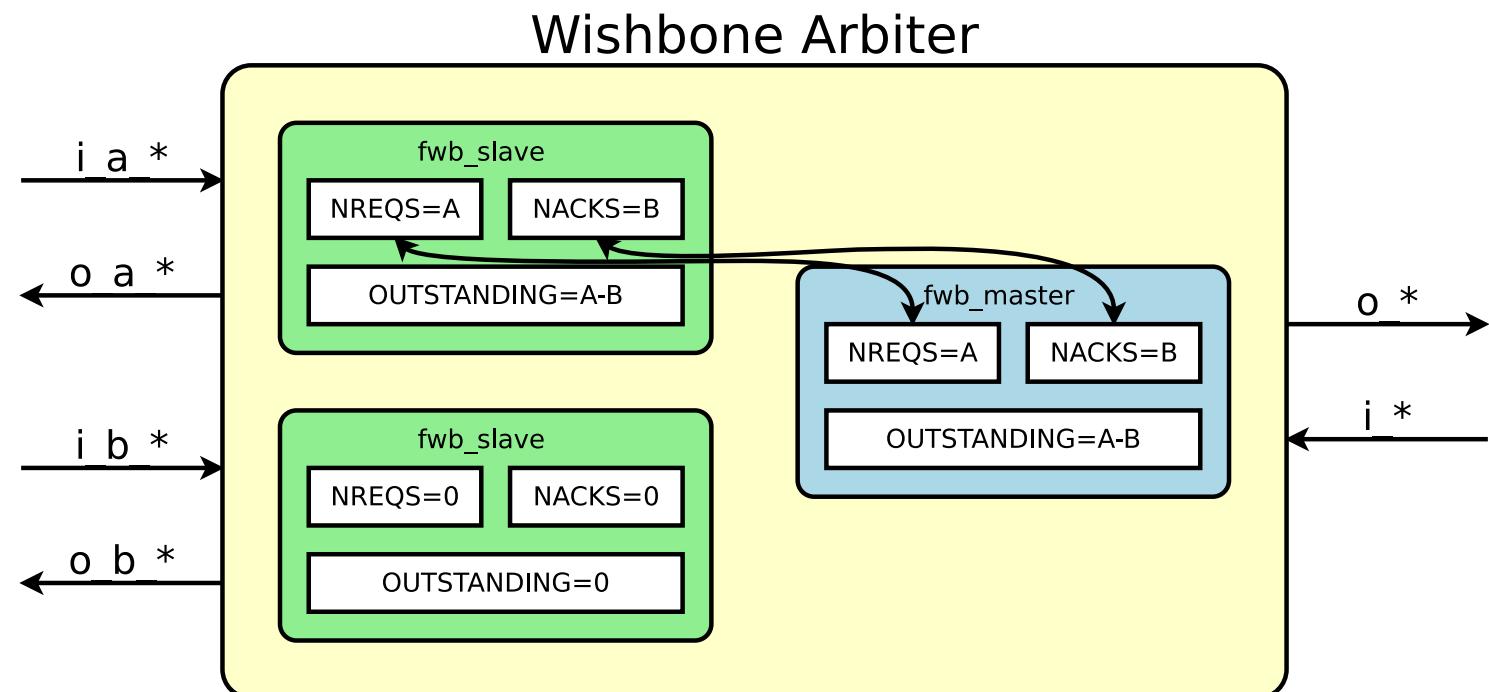
You'll write the `fwb_master.v` properties

- 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](#)

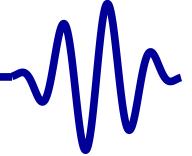
The magic is in how the files are connected

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

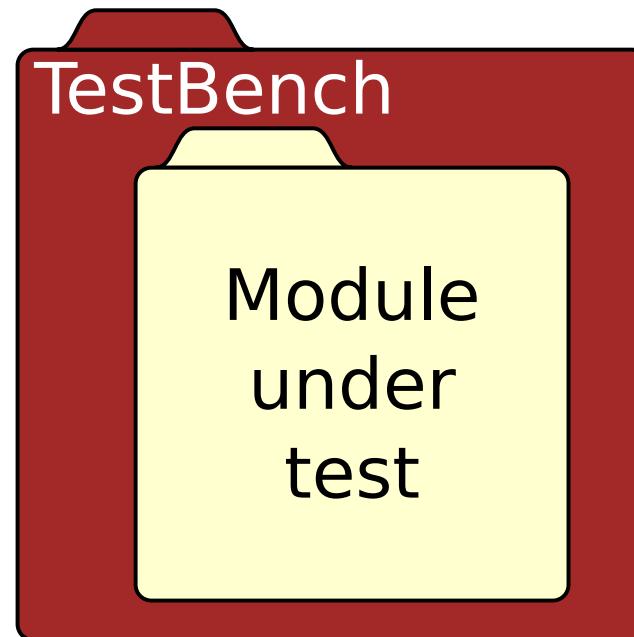
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

The magic is in how the files are connected

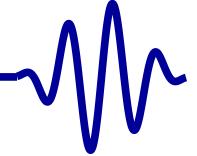
- If one interface is connected, the other ...
  - Should not have made any successful requests
  - Should not have received any acknowledgements



- 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

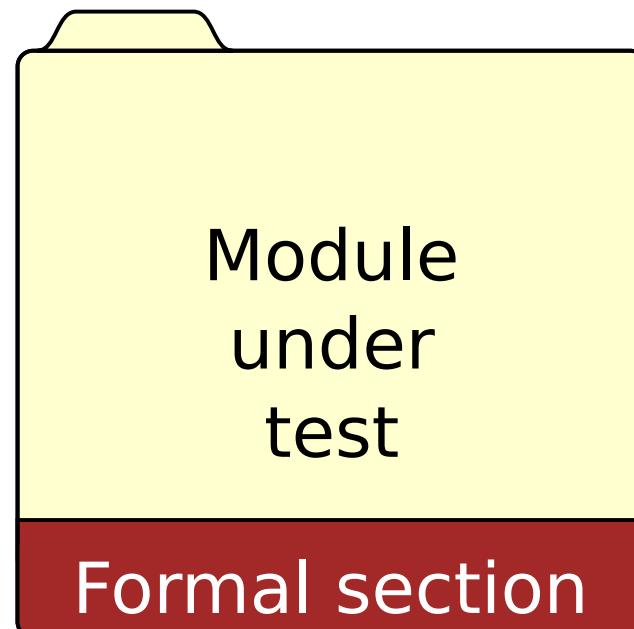
Invariants

Multiple-Clocks

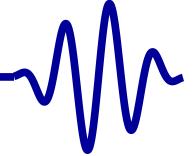
Cover

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

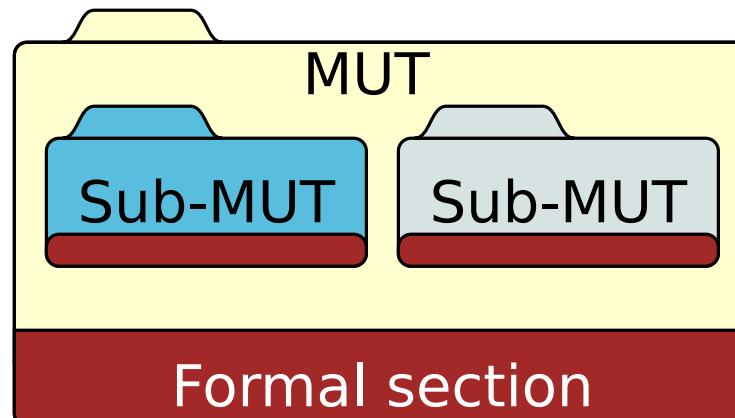
Invariants

Multiple-Clocks

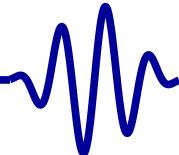
Cover

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

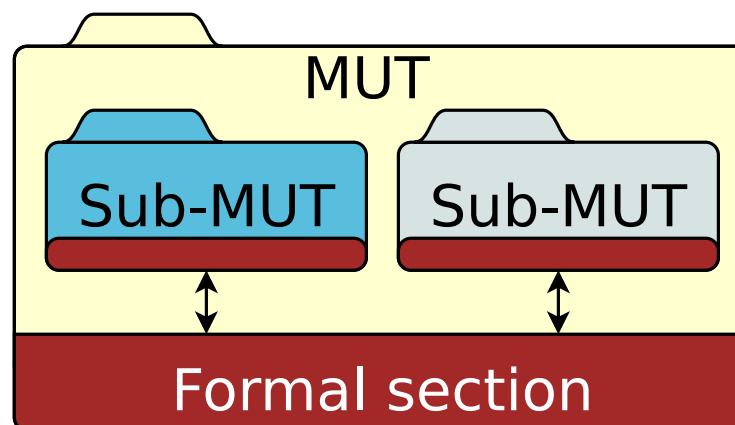
Invariants

Multiple-Clocks

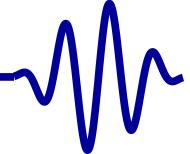
Cover

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

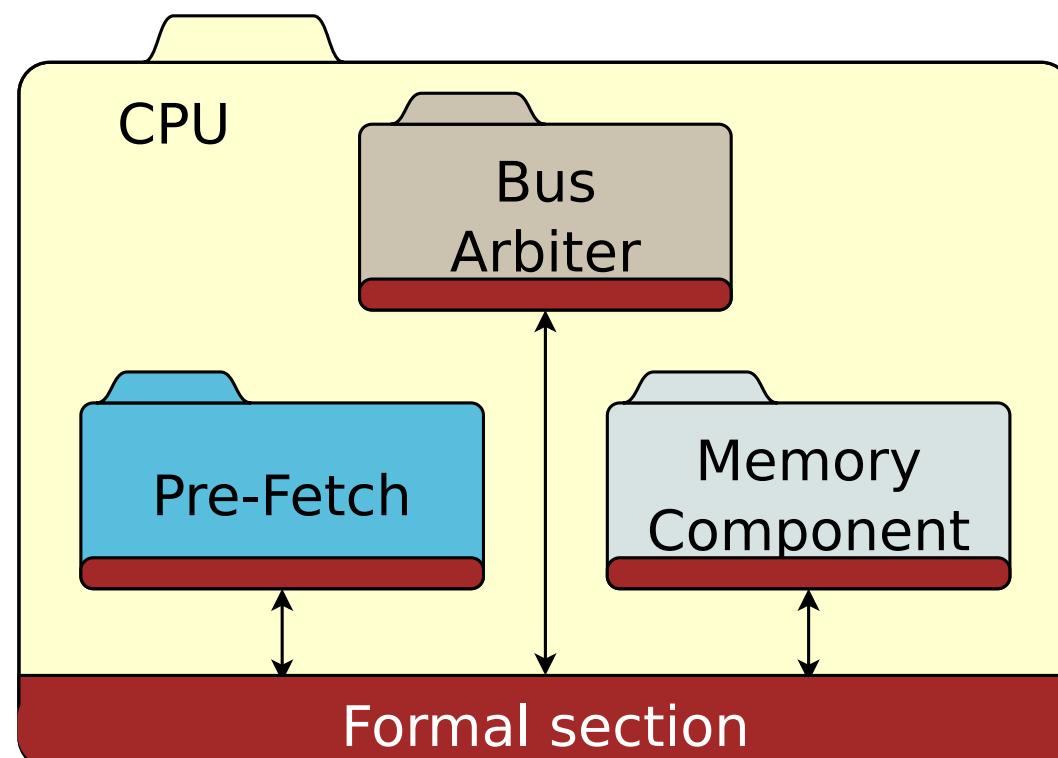
AXI

Avalon

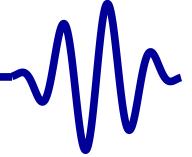
Wishbone

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

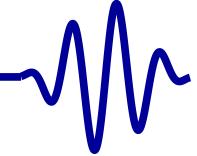
When dealing with memory, ...

- 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

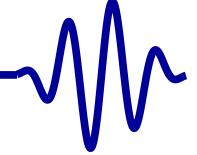
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Lesson Overview](#)[Formal](#)[▷ Memory](#)[So what?](#)[Rule](#)[Discussion](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

How might you verify 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

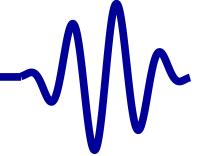
Consider the specification of a prefetch

- 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
- Both (\* `anyconst` \*) and (\* `anyseq` \*) act like inputs
- You could have written

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

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

- **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:

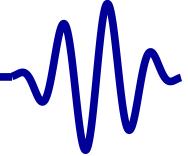
```
(* anyconst *) wire [AW-1:0] f_addr;
(* anyconst *) wire [31:0] f_data;

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_miso
        == 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](#)

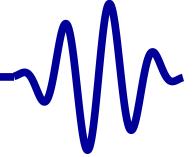
You can use this to build a serial port transmitter

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

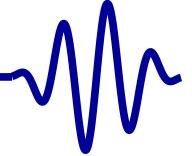
always @(posedge f_txclk)
if (f_tx_busy)
    assume(f_tx_data == $past(f_tx_data));
```

You can then

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

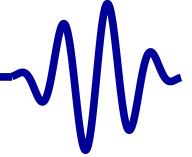
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Lesson Overview](#)[Formal](#)[Memory](#)[So what?](#)[Rule](#)[▷ Discussion](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

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

[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

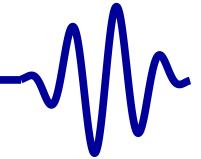
Quizzes

- 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

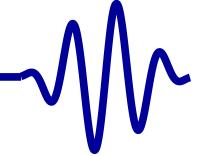
$$A \rightarrow C$$

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

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

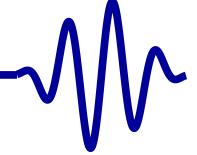
$$\neg(AB) \vee C$$

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

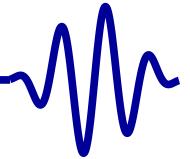
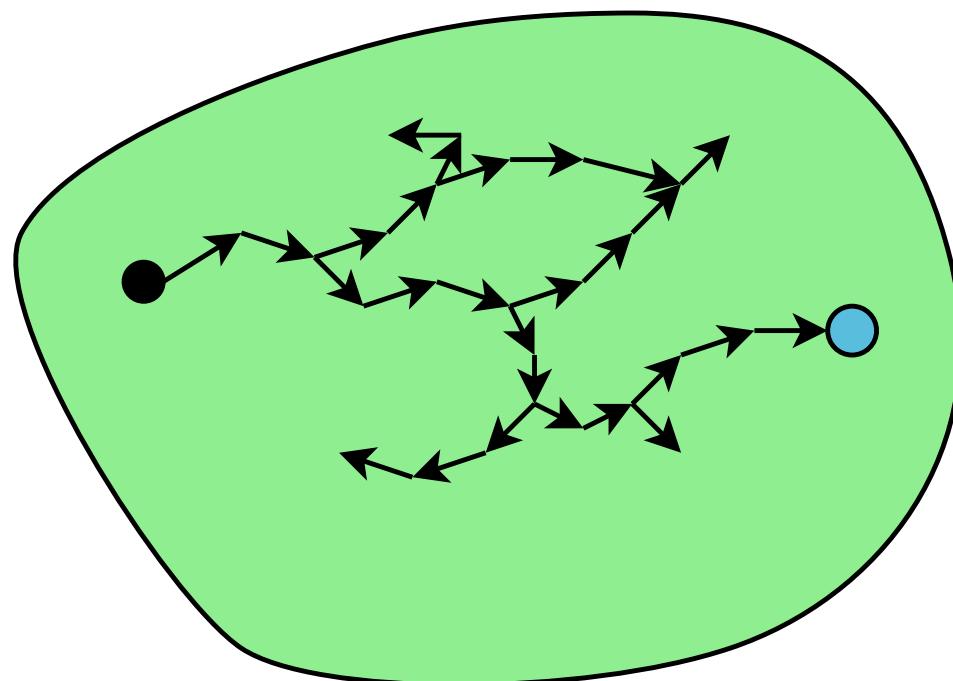
Quizzes

With every additional module,

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

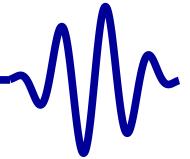
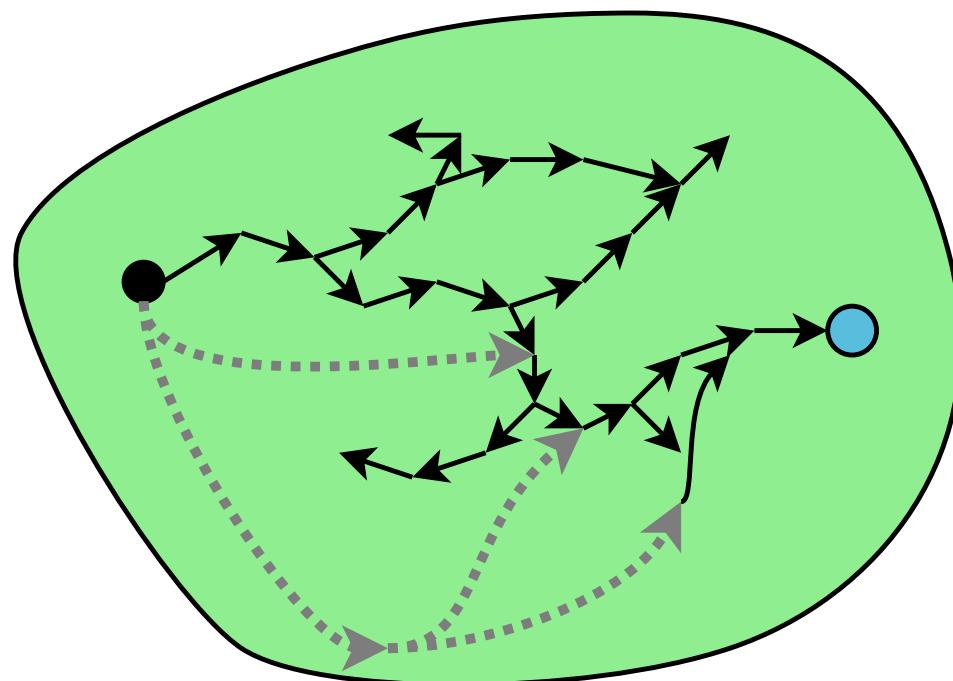
On the other hand,

- 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

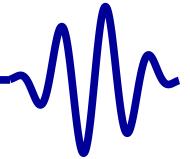
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[▷ Pictures](#)[Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Suppose we added to this design ...

- Some additional states, and
- Additional transitions

The *real* states and transitions must still remain

# GT In Pictures



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Lesson Overview

Formal

Proof

▷ Pictures

Examples

Exercise

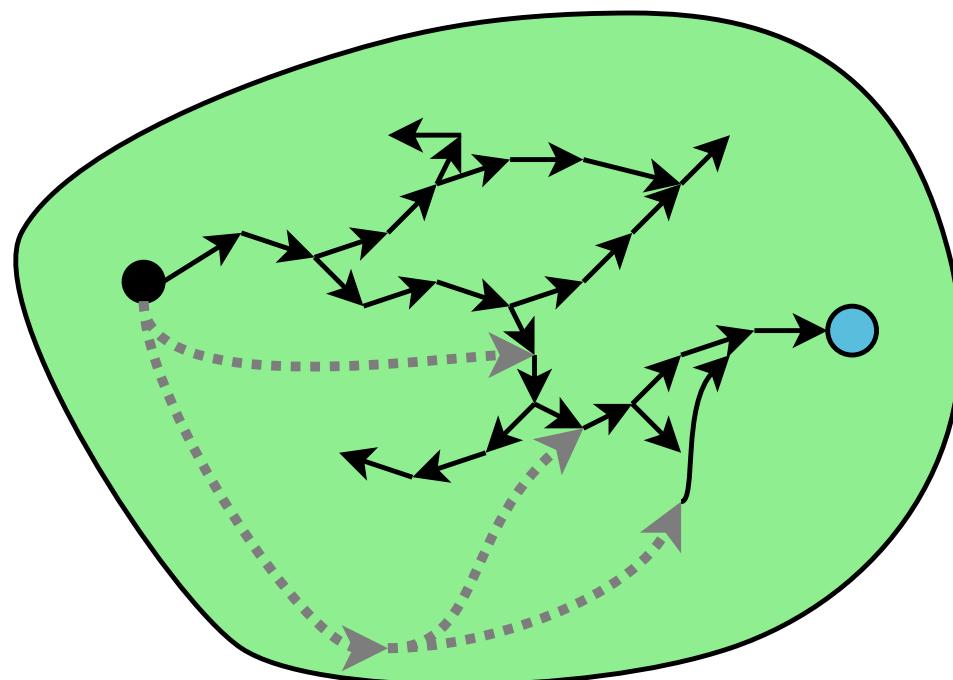
Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

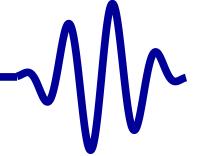


If this new design still passes, then ...

- 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

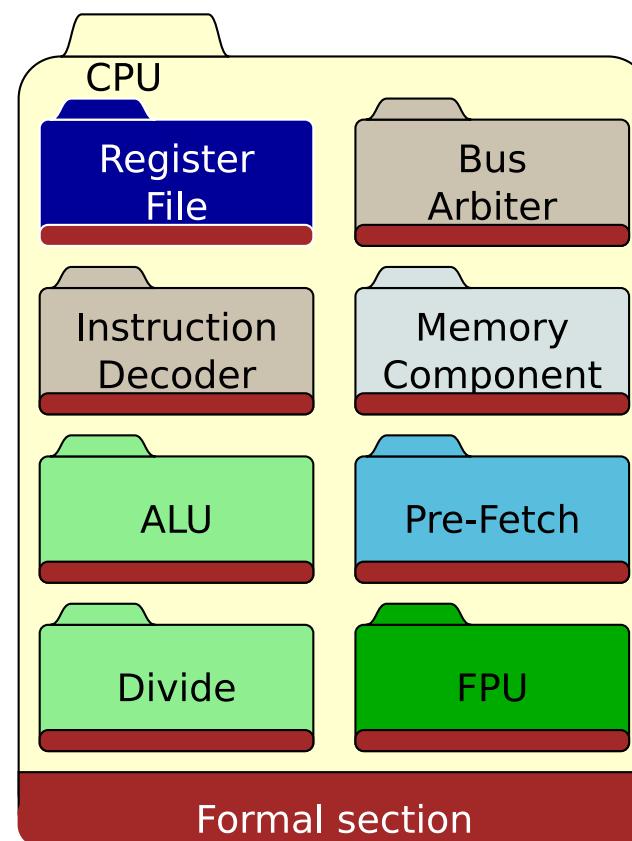
Multiple-Clocks

Cover

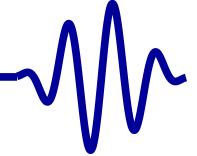
Sequences

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

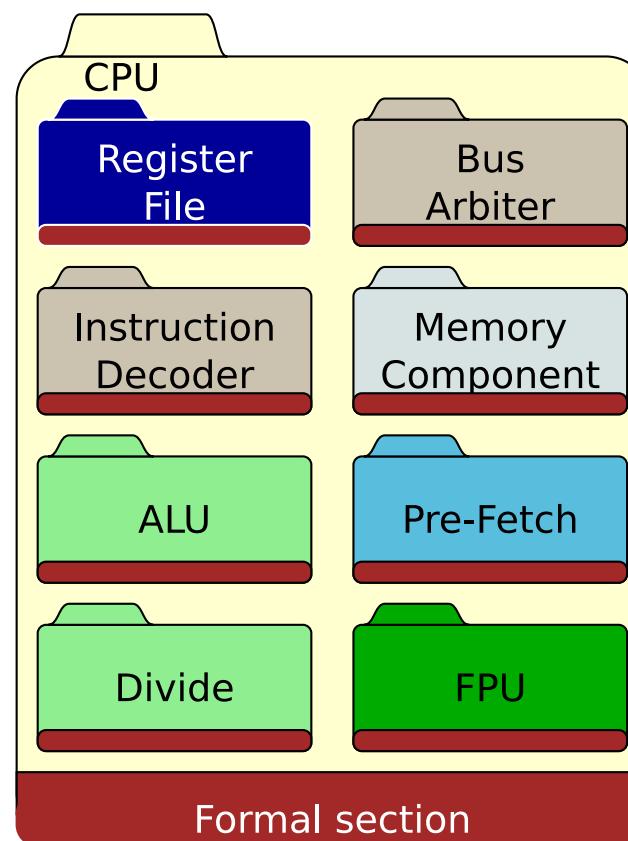
Multiple-Clocks

Cover

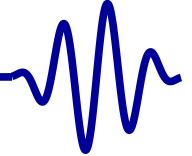
Sequences

Quizzes

Where would you start?



At the interfaces!

[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](#)

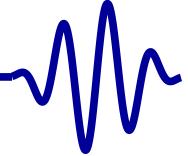
Let's consider a prefetch module as an example.



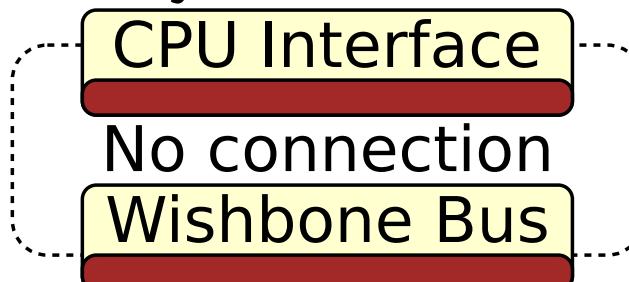
If you do this right,

- 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?

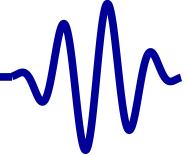
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[▷ Pictures](#)[Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Suppose the prefetch was just a shell

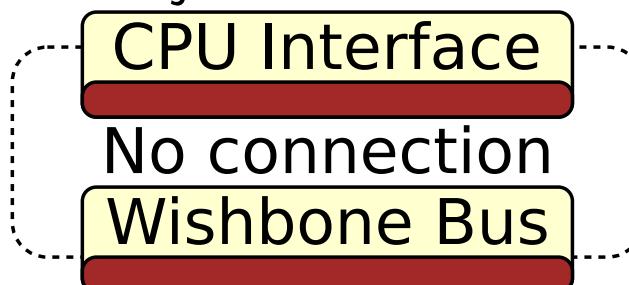


It would still interact properly with

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

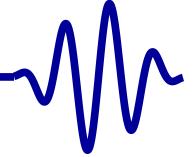
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[▷ Pictures](#)[Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

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

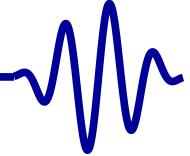
Consider these statements:

□

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

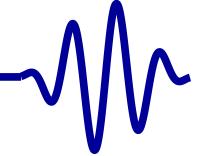
Consider these statements:

- 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](#)

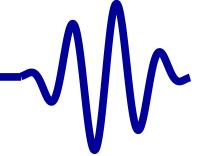
Consider these statements:

- 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](#)

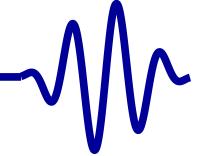
Consider these statements:

- 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

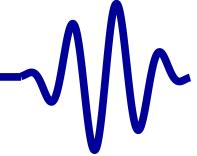
If (Divide returns a value when requested)

And (It is the right value)

Then (The divide works within the design)



# 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

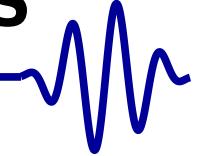
Consider these statements:

- Formal solvers break down when applied to multiplies

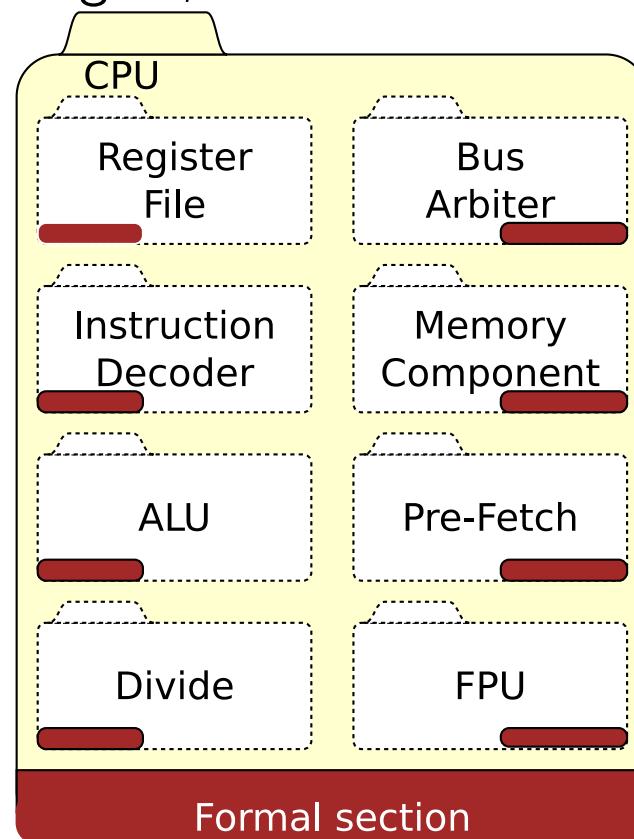
If (Multiply unit returns an answer  $N$  clocks later)

And (It is the right value)

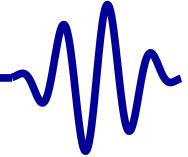
Then (The multiply works within the design)

[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](#)

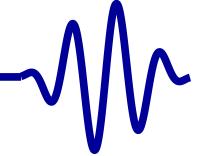
Let's consider a fractional counter:

```
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  $100 \times 10^6$  clocks to roll over and set o\_pps
- Formally checking  $100 \times 10^6$  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

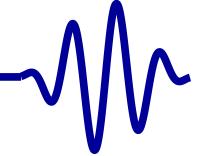
```
always @ (posedge i_clk)
    { o_pps, r_count } <= r_count + increment;
```

Will this work?

- Let's try this to see!

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

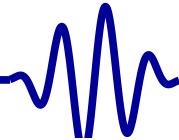
```
always @ (posedge i_clk)
if ((f_past_valid)&&(r_count < $past(r_count)))
    assert (o_pps);
```

[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?

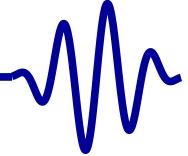
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[Pictures](#)[Examples](#)[▷ Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

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:

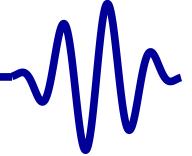
- Rebuild the counter
- Make it increment by one
- Build it so that ...

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

// and

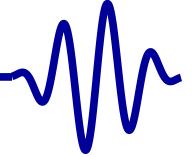
```
always @(posedge i_clk)  
    if ((f_past_valid)&&(!$past(&r_count)))  
        assert(!o_carry);
```

- 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*

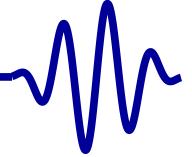
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal Proof](#)[Pictures](#)[Examples](#)[▷ Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

### 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



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

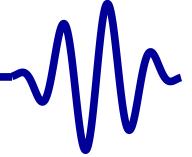
▷ Lesson Removed

Multiple-Clocks

Cover

Sequences

Quizzes



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

▷ Multiple-Clocks

Basics

SBY File

(\* gclk \*)

\$rose

\$stable

Examples

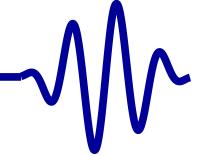
Exercises

Cover

Sequences

Quizzes

# Multiple-Clocks

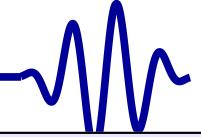
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[▷ Basics](#)[SBY File](#)[\(\\* gclk \\*\)](#)[\\$rose](#)[\\$stable](#)[Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

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
  - `(* gclk *)`
  - **\$stable, \$changed**
  - **\$rose, \$fell**

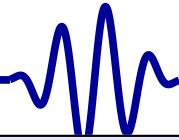
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[▷ SBY File](#)[\(\\* gclk \\*\)](#)[\\$rose](#)[\\$stable](#)[Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

```
[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  
(\* gclk \*)  
\$rose  
\$stable  
Examples  
Exercises  
Cover  
Sequences  
Quizzes

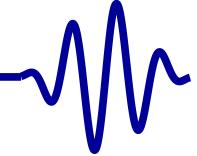
```
[options]
mode prove
multiclock on ← Multiple clocks require this line

[engines]
smtbmc

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

[files]
# file list
```

# Five Tools



Welcome

Motivation

Basics

Clocked and  $\$past$

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

▷ SBY File

(\* gclk \*)

$\$rose$

$\$stable$

Examples

Exercises

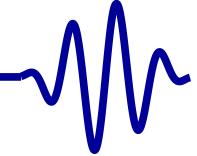
Cover

Sequences

Quizzes

- `(* gclk *)`  
Provides access to the global formal 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 formal time-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 time-step, creating a negative edge  
 $\$fell(A)$  is equivalent to  $(!A[0]) \&\& (\$past(A[0]))$

# (\* gclk \*)



- A global formal time step

```
(* gclk *) wire gbl_clk;
```

- You can use this to describe clock properties

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

initial f_last_clk = 0;
always @ (posedge gbl_clk)
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

▷ (\* gclk \*)

\$rose

\$stable

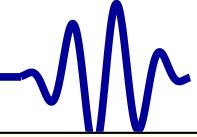
Examples

Exercises

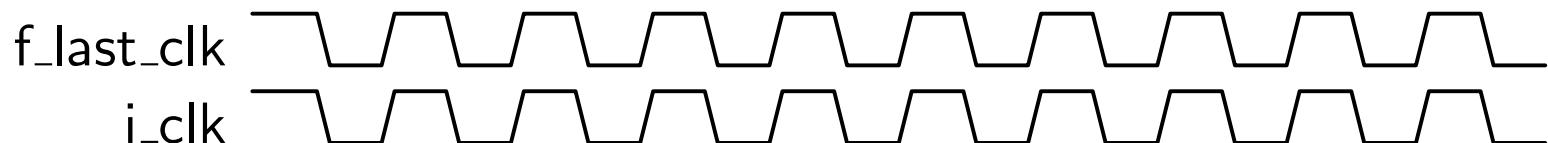
Cover

Sequences

Quizzes



```
always @(*posedge gbl_clk)
begin
    f_last_clk <= !f_last_clk;
    assume(i_clk == f_last_clk);
end
```



# (\* gclk \*)



- Used to gain access to the formal time-step

```
(* gclk *) wire gbl_clk;
```

- You can use this to describe clock properties

```
// Assume two related clock signals
//
reg [2:0] f_clk_counter;

initial f_clk_counter = 0;
always @ (posedge gbl_clk)
begin
    f_clk_counter <= f_clk_counter + 1'b1;
    assume(i_clk_fast == f_clk_counter[0]);
    assume(i_clk_slow == f_clk_counter[2]);
end
```

Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

▷ (\* gclk \*)

\$rose

\$stable

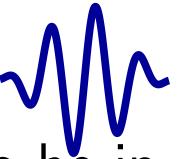
Examples

Exercises

Cover

Sequences

Quizzes

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

The clock logic on the last slide forces these two clocks to be in sync

f\_clk\_counter



i\_clk\_fast



i\_clk\_slow



# (\* gclk \*)



- Used to gain access to the formal time-step
- You can use this to describe clock properties

```
// Assume two clocks, same speed,  
// unknown constant phase offset  
(* gclk *)      wire      gbl_clk;  
(* anyconst *)   wire [2:0] f_clk_offset;  
  
initial f_clk_counter= 0;  
always @ (posedge gbl_clk)  
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[2]);  
    assume(i_clk_two == f_clk_two[2]);  
end
```

Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

▷ (\* gclk \*)

\$rose

\$stable

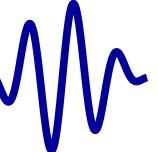
Examples

Exercises

Cover

Sequences

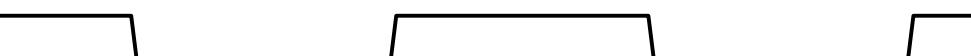
Quizzes

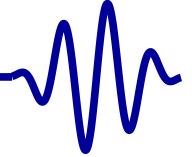
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[▷ \(\\* gclk \\*\)](#)[\\$rose](#)[\\$stable](#)[Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

The formal tool will pick the phase offset between these two generated clock waveforms

f\_clk\_counter 

i\_clk\_one 

i\_clk\_two 



How might you describe two unrelated clocks?

Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

▷ (\* gclk \*)

\$rose

\$stable

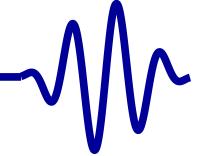
Examples

Exercises

Cover

Sequences

Quizzes

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

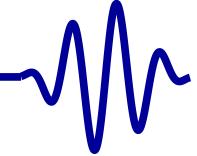
## How might you describe two unrelated clocks?

```
(* gclk *)      wire      gbl_clk;
(* anyconst *)  wire [7:0] f_a_step;
always @(*)
assume((f_a_step > 0) &&(f_a_step[7] == 1'b0));

always @ (posedge gbl_clk)
begin
    f_a_counter <= f_a_counter + f_a_step;

    assume(i_clk_a == f_a_counter[7]);
end
```

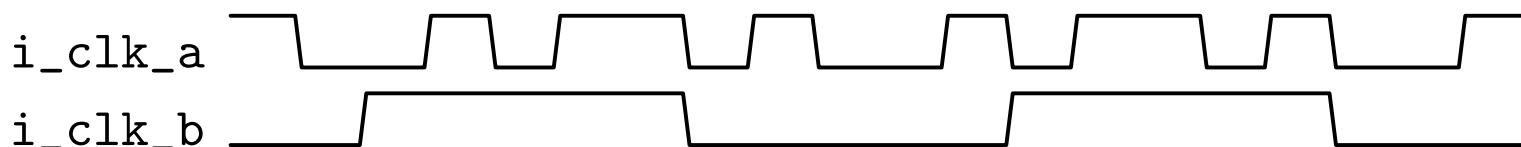
- The (\* anyconst \*) register may take on any constant value
- You can repeat this logic for the second clock.

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

The timing relationship between these two clocks can be anything

- Each clock can have an arbitrary frequency
- Each clock can have an arbitrary phase

Here's a theoretical example trace



Don't be surprised by the appearance of phase noise

**Bonus:** The trace above isn't realistic. Why not?



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

(\* gclk \*)

▷ \$rose

\$stable

Examples

Exercises

Cover

Sequences

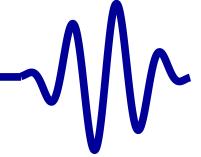
Quizzes

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 @(posedge gbl_clk)
if (! $rose(i_clk))
    assume(i_input == $past(i_input));
```



Would this work?

```
always @(posedge gbl_clk)
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

(\* gclk \*)

▷ \$rose

\$stable

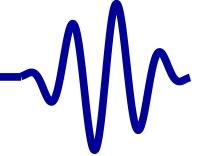
Examples

Exercises

Cover

Sequences

Quizzes



Would this work?

```
always @(*posedge gbl_clk)
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

(\* gclk \*)

▷ \$rose

\$stable

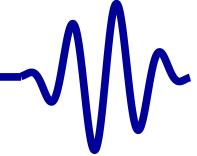
Examples

Exercises

Cover

Sequences

Quizzes



Could we do it this way?

```
always @(posedge gbl_clk)
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

(\* gclk \*)

▷ \$rose

\$stable

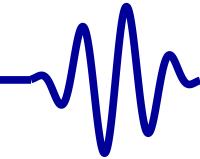
Examples

Exercises

Cover

Sequences

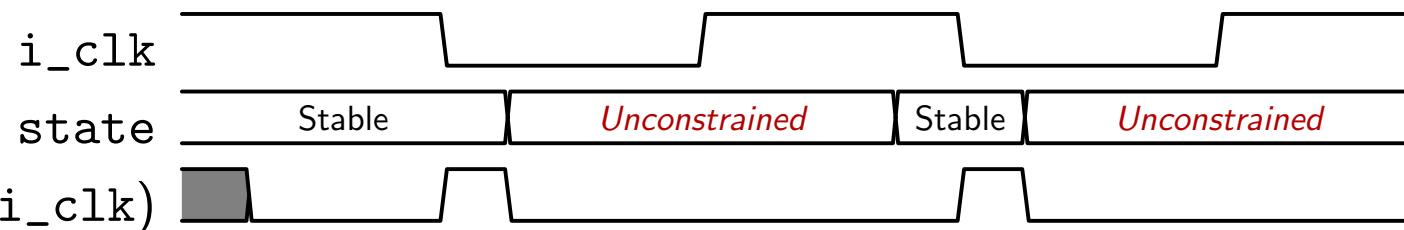
Quizzes

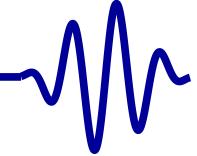
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\(\\* gclk \\*\)](#)[▷ \\$rose](#)[\\$stable](#)[Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

Could we do it this way?

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

- No, this doesn't work either





Is this equivalent?

```
always @(posedge gbl_clk)
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

(\* gclk \*)

▷ \$rose

\$stable

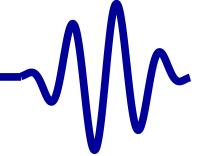
Examples

Exercises

Cover

Sequences

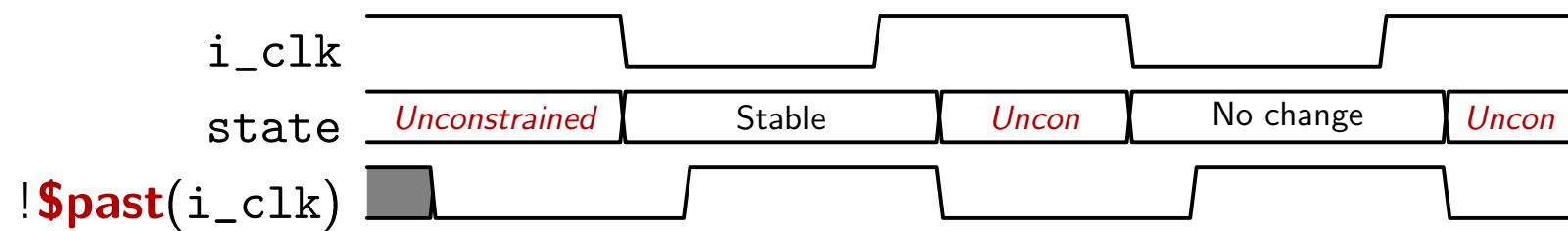
Quizzes

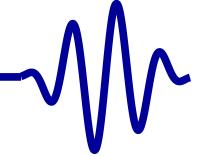
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\(\\* gclk \\*\)](#)[▷ \\$rose](#)[\\$stable](#)[Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

Is this equivalent?

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

- Why not?





This fixes our problems. Will this work?

```
always @(posedge gbl_clk)
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

(\* gclk \*)

▷ \$rose

\$stable

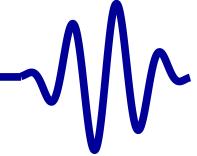
Examples

Exercises

Cover

Sequences

Quizzes



This fixes our problems. Will this work?

```
always @(posedge gbl_clk)
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

(\* gclk \*)

▷ \$rose

\$stable

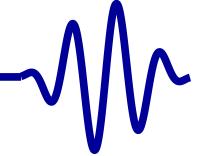
Examples

Exercises

Cover

Sequences

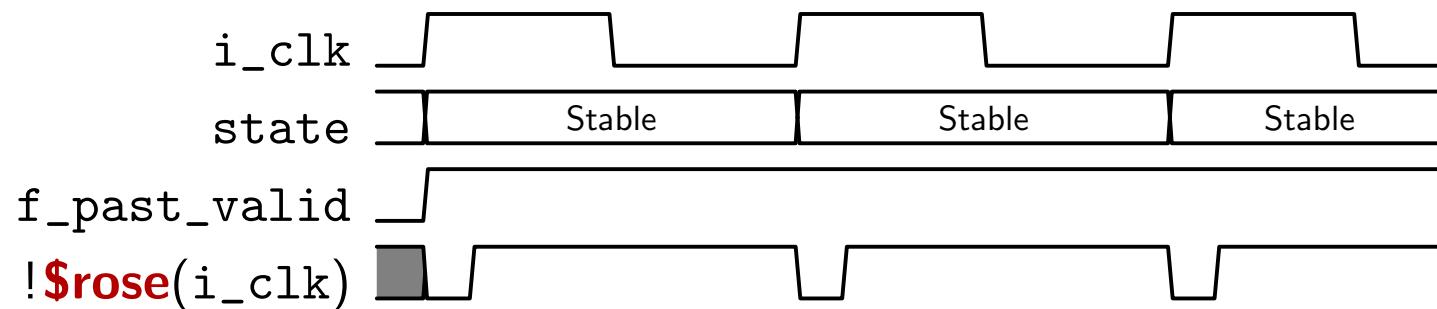
Quizzes

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

- State/outputs should be clock synchronous

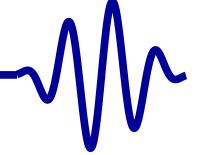
```
always @ (posedge gbl_clk)
  if ((f_past_valid)&&(!$rose(i_clk))
      assert(state == $past(state));
```

- With f\_past\_valid this works



- **\$rose** requires a clock, such as  
**always @(posedge gbl\_clk)**

# \$stable



Describes a signal which has not changed

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

- Requires a clock edge
  - always @(**posedge** gbl\_clk)**
  - always @(**posedge** i\_clk)**
- 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

(\* gclk \*)

\$rose

▷ \$stable

Examples

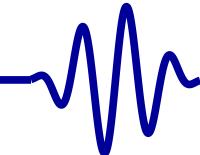
Exercises

Cover

Sequences

Quizzes

# \$stable



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File  
(\* gclk \*)

\$rose

> \$stable

Examples

Exercises

Cover

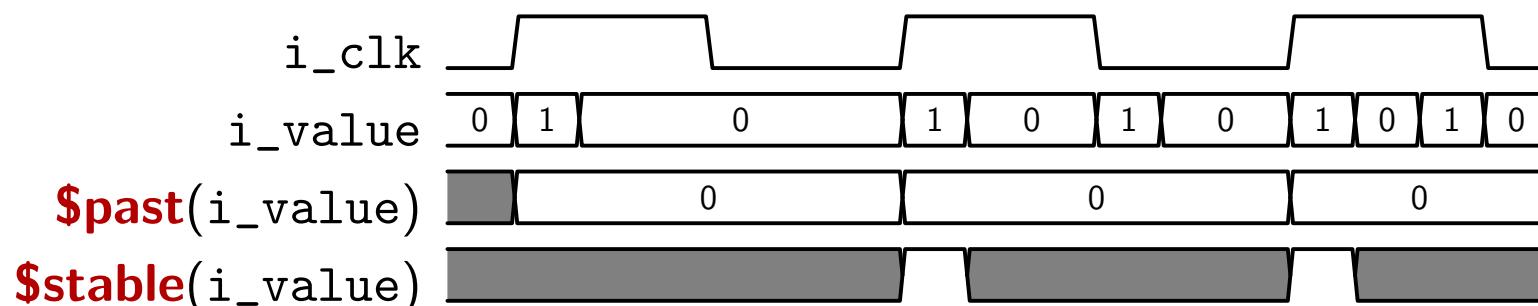
Sequences

Quizzes

*Caution: \$stable(x) might still change between clock edges*

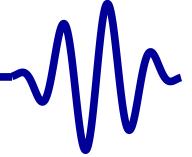
```
always @(posedge i_clk)
    assume($stable(i_value));
```

The waveform below would satisfy the assumption above

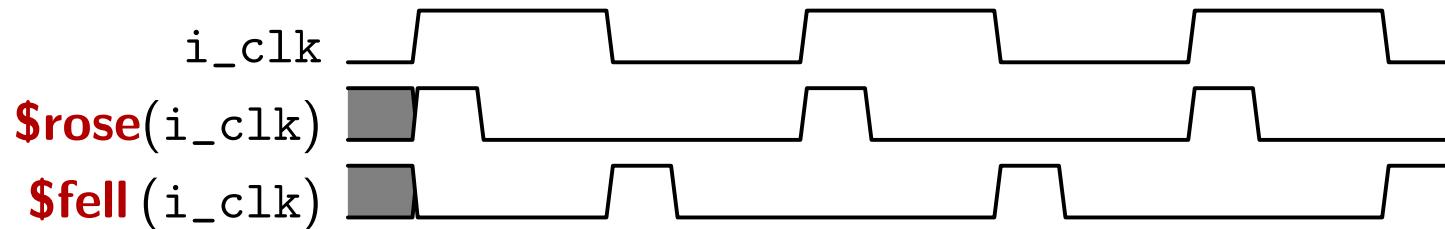


The key to understanding what's going on is to realize . . .

- The assumption is only evaluated on @(**posedge** i\_clk)
- **\$past(i\_value)** is only sampled @(**posedge** i\_clk)
- . . . and not on the formal (\* **gclk** \*) time step.

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

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





# Examples



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

(\* gclk \*)

\$rose

\$stable

▷ Examples

Exercises

Cover

Sequences

Quizzes

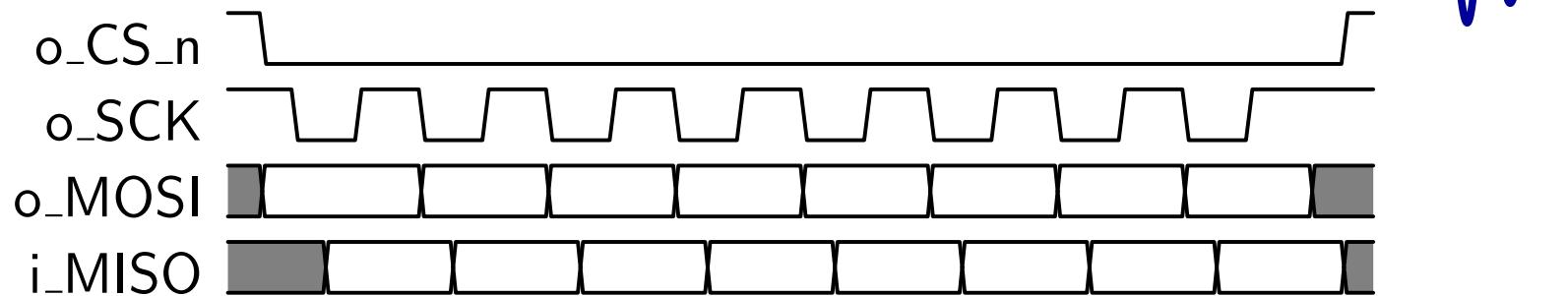
- Most logic doesn't need the multiclock option
- To help with logic that might need it, I use a parameter

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

generate if (F_OPT_CLK2FFLOGIC)
begin
    (* gclk *) wire gbl_clk;

    always @(*(posedge gbl_clk))
        if ((f_past_valid)&&(!$rose(i_clk)))
            begin
                assume($stable(i_axi_awready));
                assume($stable(i_axi_wready));
                // ...
            end
    end
end generate
```

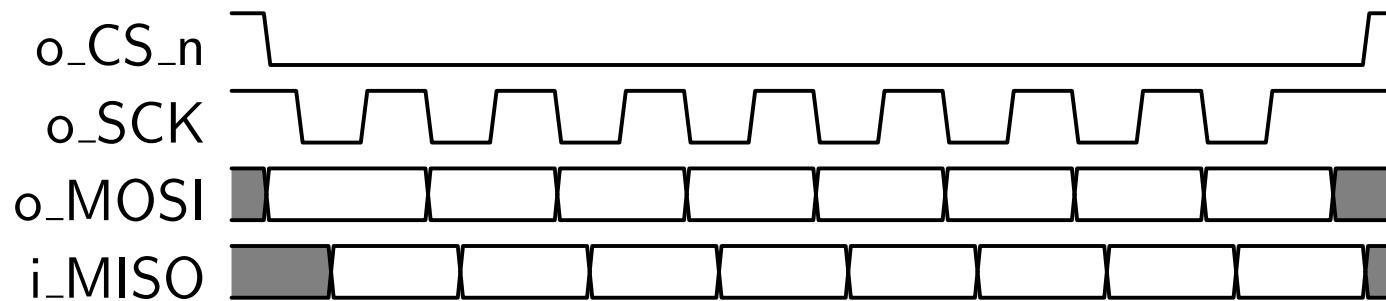
- [Welcome](#)
- [Motivation](#)
- [Basics](#)
- [Clocked and \\$past](#)
- [k Induction](#)
- [Bus Properties](#)
- [Free Variables](#)
- [Abstraction](#)
- [Invariants](#)
- [Multiple-Clocks](#)
- [Basics](#)
- [SBY File](#)
- [\(\\* gclk \\*\)](#)
- [\\$rose](#)
- [\\$stable](#)
- [▷ Examples](#)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



- How would you formally describe the o\_SCK and o\_CS\_n relationship?



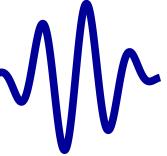
- Welcome
- Motivation
- Basics
- Clocked and \$past
- $k$  Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Basics
- SBY File
- (\* gclk \*)
- \$rose
- \$stable
- ▷ Examples
- Exercises
- Cover
- Sequences
- Quizzes



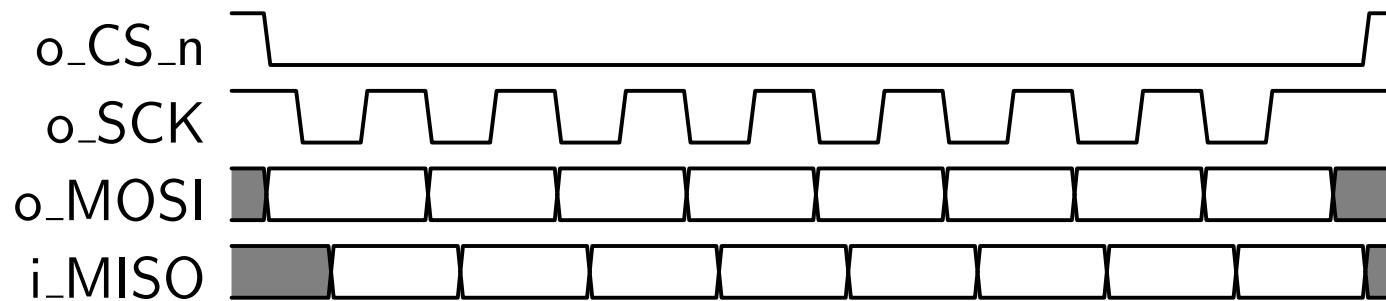
- How would you formally describe the `o_SCK` and `o_CS_n` relationship?

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

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



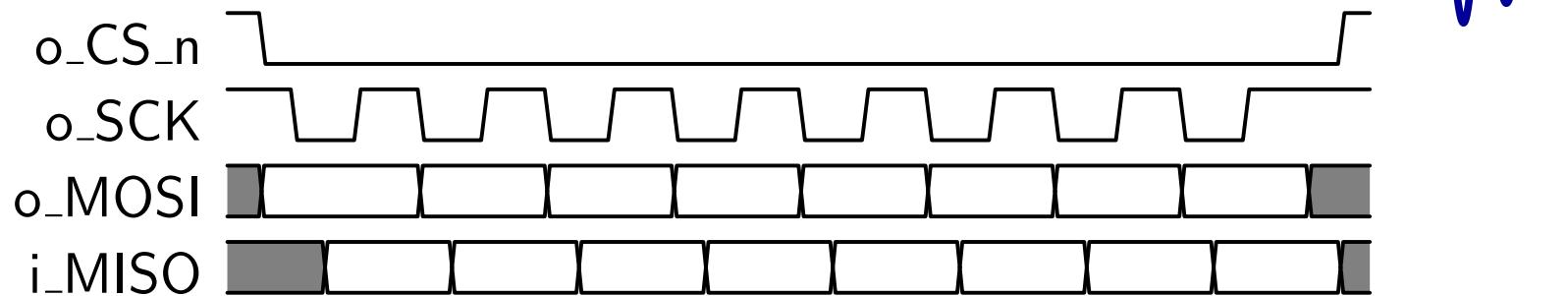
- Welcome
- Motivation
- Basics
- Clocked and \$past
- $k$  Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Basics
- SBY File
- (\* gclk \*)
- \$rose
- \$stable
- ▷ Examples
- Exercises
- Cover
- Sequences
- Quizzes



- How would you formally describe the o\_SCK and o\_CS\_n relationship?

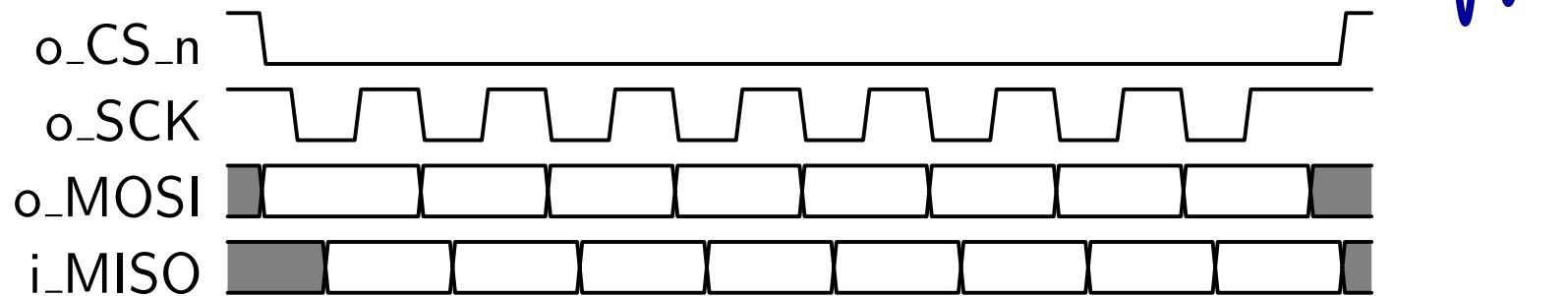
```
always @(posedge gbl_clk)
if ((f_past_valid)
      &&(($rose(o_CS_n))||($fell(o_CS_n))))
      assert((o_SCK)&&($stable(o_SCK)));
```

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



- How would you describe o\_MOSI?

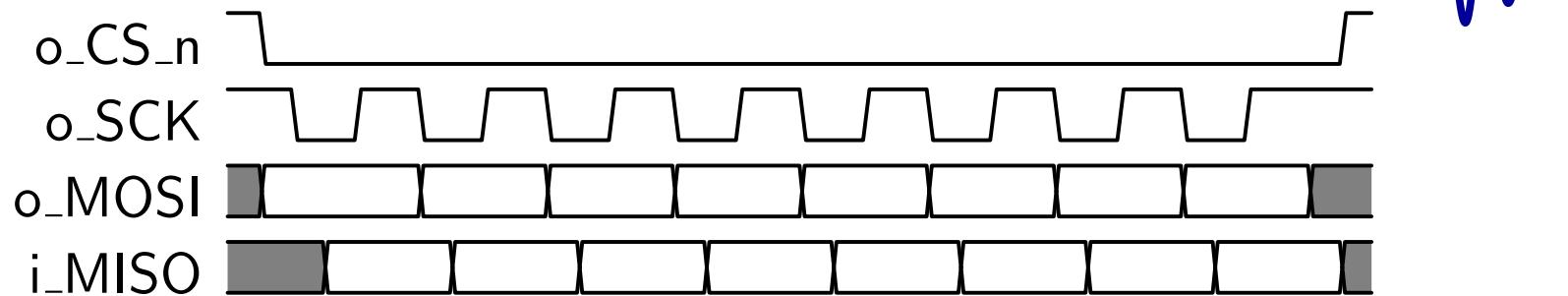
- Welcome
- Motivation
- Basics
- Clocked and \$past
- $k$  Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Basics
- SBY File
- (\* gclk \*)
- \$rose
- \$stable
- ▷ Examples
- Exercises
- Cover
- Sequences
- Quizzes



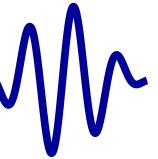
- How would you describe o\_MOSI?

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

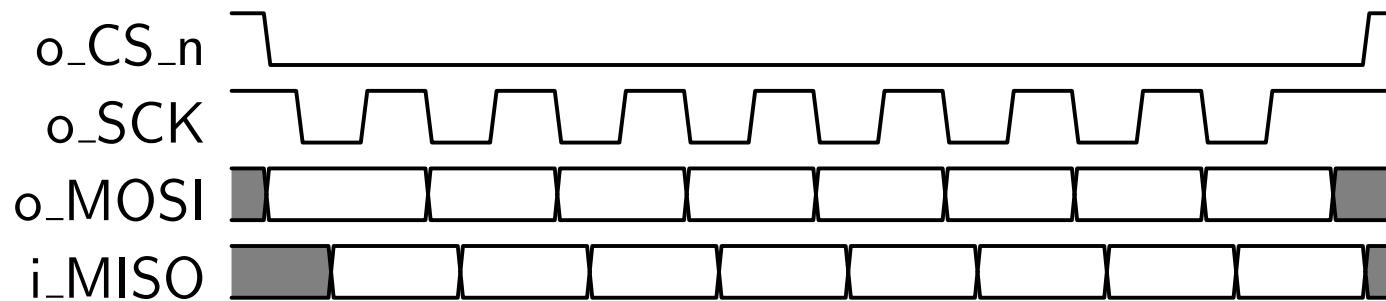
- Welcome
- Motivation
- Basics
- Clocked and \$past
- $k$  Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Basics
- SBY File
- (\* gclk \*)
- \$rose
- \$stable
- ▷ Examples
- Exercises
- Cover
- Sequences
- Quizzes



- How would you describe i\_MISO?



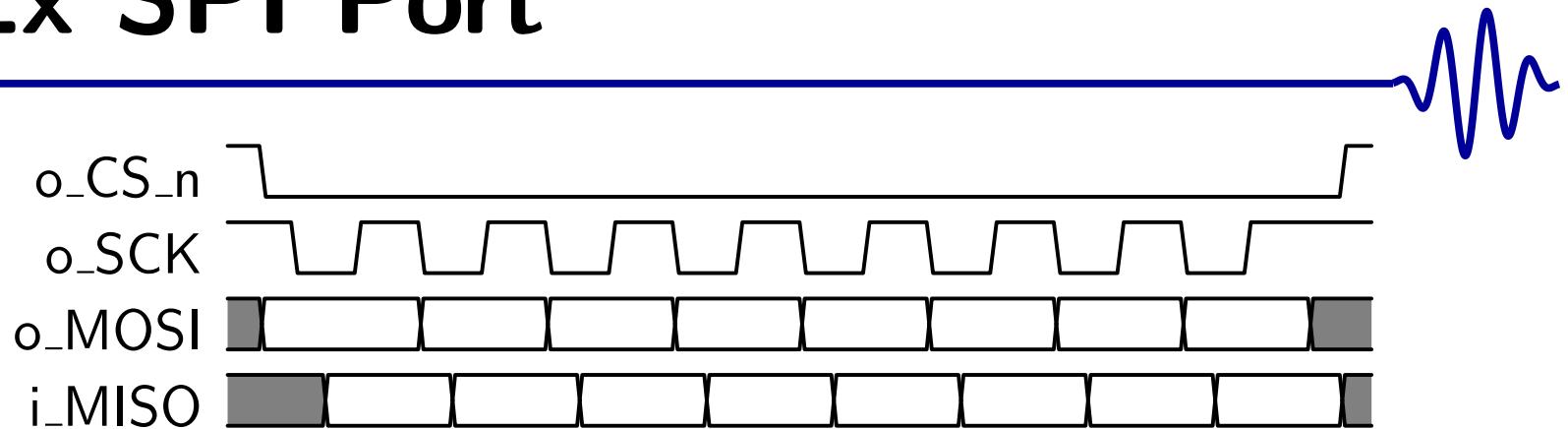
- Welcome
- Motivation
- Basics
- Clocked and \$past
- $k$  Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Basics
- SBY File
- (\* gclk \*)
- \$rose
- \$stable
- ▷ Examples
- Exercises
- Cover
- Sequences
- Quizzes



- How would you describe i\_MISO?

```
always @(posedge gbl_clk)
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](#)
  - [\(\\* gclk \\*\)](#)
  - [\\$rose](#)
  - [\\$stable](#)
  - [▷ Examples](#)
  - [Exercises](#)
- [Cover](#)
- [Sequences](#)
- [Quizzes](#)



- Should the **i\_MISO** be able to change more than once per clock?

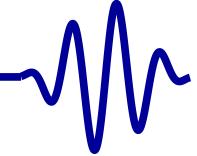
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\(\\* gclk \\*\)](#)[\\$rose](#)[\\$stable](#)[▷ Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

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

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

```
always @ (posedge gbl_clk)
  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](#)[\(\\* gclk \\*\)](#)[\\$rose](#)[\\$stable](#)[▷ Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

- Forcing exactly 8 clocks

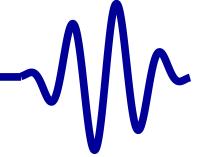
```
always @ (posedge gbl_clk)
  if (o_CS_n)
    f_spi_bits <= 0;
  else if ($rose(o_SCK))
    f_spi_bits <= f_spi_bits + 1'b1;
```

```
always @ (posedge gbl_clk)
  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

(\* gclk \*)

\$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`

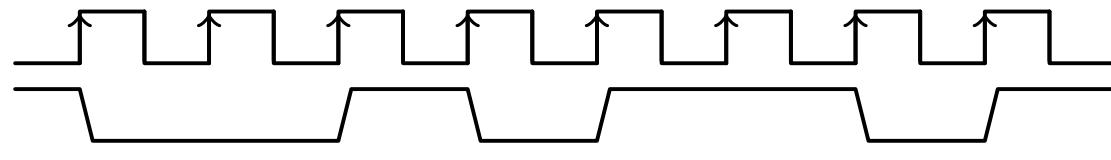
3. Clock Switch

`exercises-11/clkswitch.v`

[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\(\\* gclk \\*\)](#)[\\$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

# Ex: Input Serdes



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

(\* gclk \*)

\$rose

\$stable

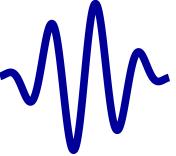
Examples

▷ Exercises

Cover

Sequences

Quizzes

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

Be aware of the asynchronous reset signal!

i\_areset\_n



i\_fast\_clk



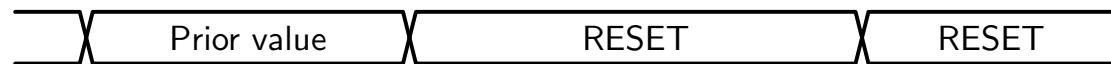
i\_pin



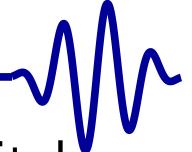
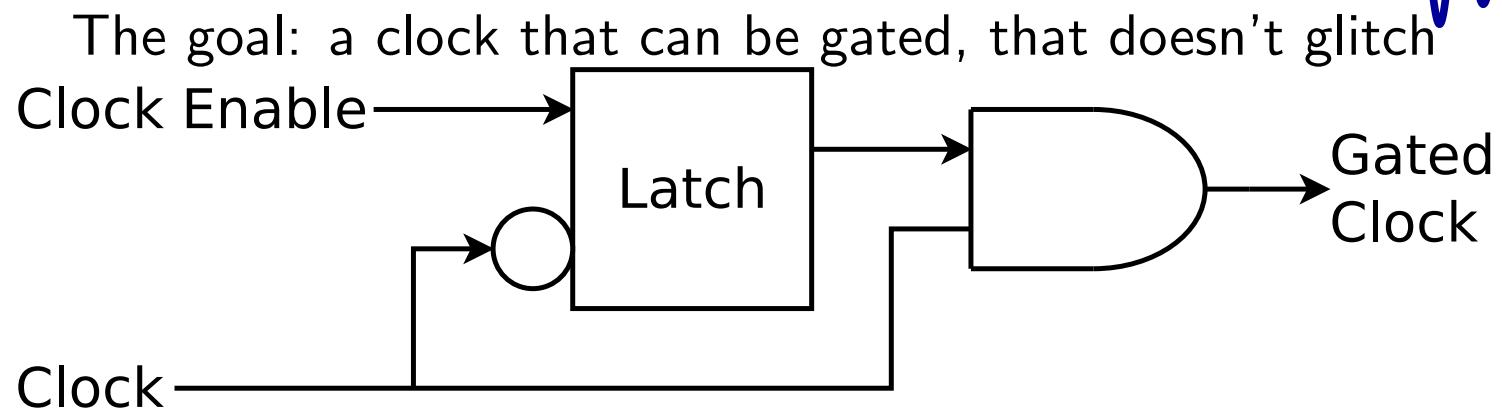
i\_slow\_clk



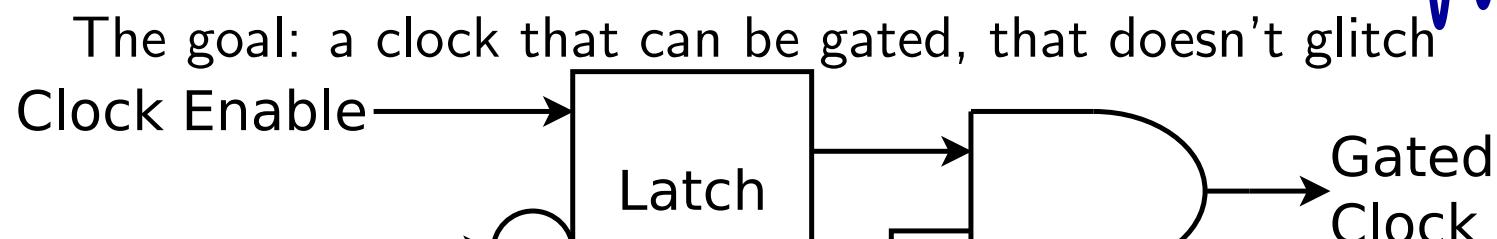
o\_word



- 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](#)[\(\\* gclk \\*\)](#)[\\$rose](#)[\\$stable](#)[Examples](#)[▷ Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

- exercise-10/ Contains the file clkgate.v

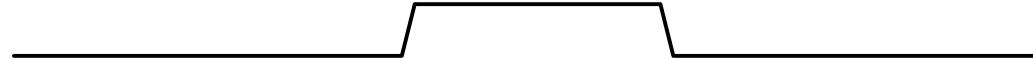
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\(\\* gclk \\*\)](#)[\\$rose](#)[\\$stable](#)[Examples](#)[▷ Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

Clock

i\_clk



i\_en



o\_clk

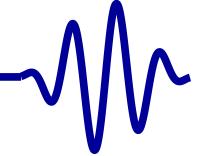


[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\(\\* gclk \\*\)](#)[\\$rose](#)[\\$stable](#)[Examples](#)[▷ Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

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

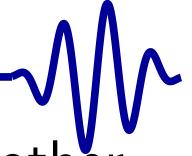
See `exercise-10/clkgate.v`

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

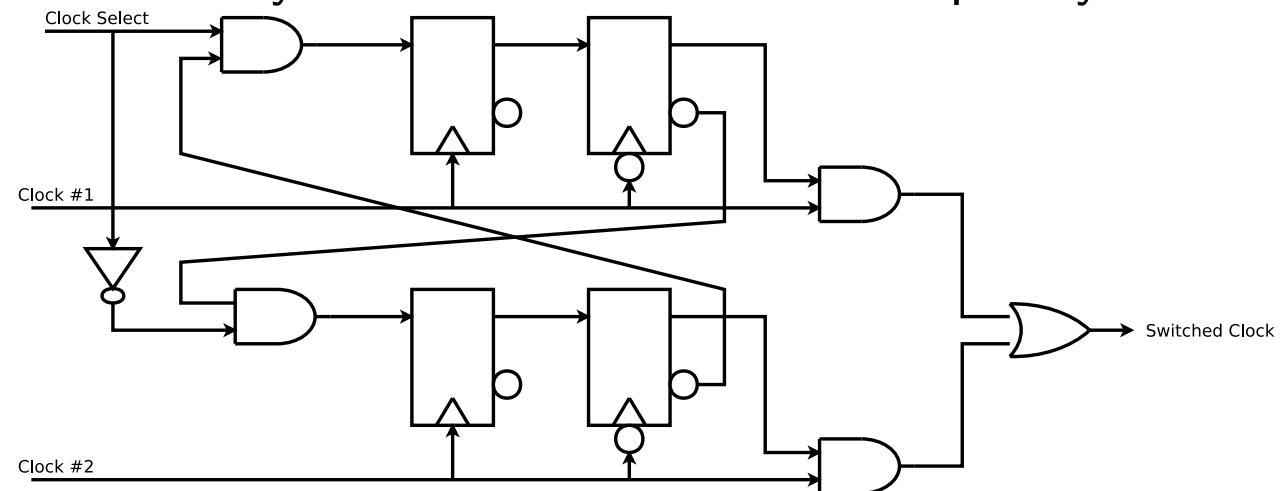
### Hints:

- 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](#)[\(\\* gclk \\*\)](#)[\\$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](#)[\(\\* gclk \\*\)](#)[\\$rose](#)[\\$stable](#)[Examples](#)[▷ Exercises](#)[Cover](#)[Sequences](#)[Quizzes](#)

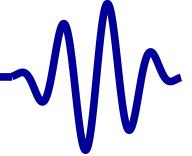
Goal: To safely switch from one clock frequency to another

- Inputs
  - Two arbitrary clocks
  - One select line

Prove that the output clock

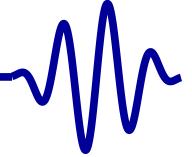
- Is always high (or low) for at least the duration of one of the clocks
- Doesn't stop

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](#)[\(\\* gclk \\*\)](#)[\\$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

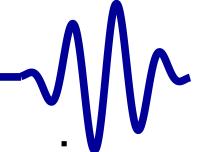
Sequences

Quizzes

# Cover



# Lesson Overview



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

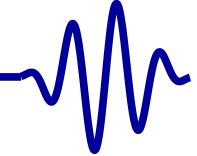
- BMC and induction test *safety* properties  
They prove that something *will not* happen
- Cover tests a *liveness* property  
It proves that something *may* happen

## Objectives

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



# Why 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

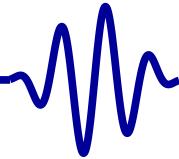
Quizzes

## 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?

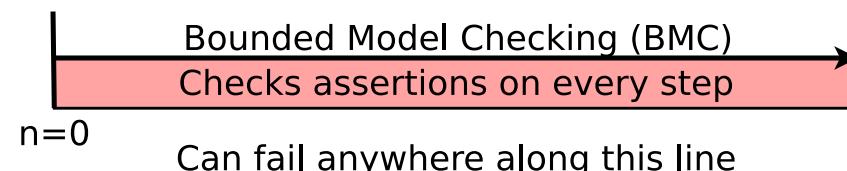
What is cover good for? Catching the *careless assumption!*

What else? Ad hoc simulation traces!

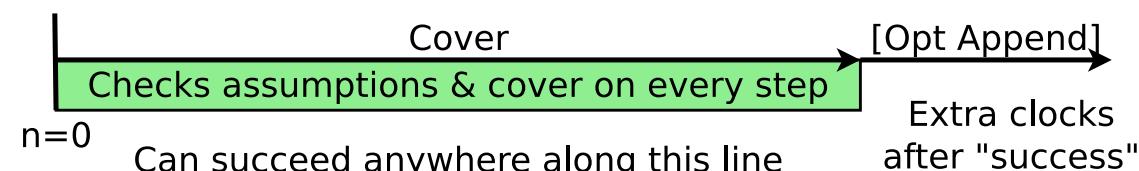
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[▷ BMC vs Cover](#)[Cover in Verilog](#)[State Space](#)[SymbiYosys](#)[Examples](#)[Counter](#)[Sequences](#)[Quizzes](#)

Cover is more like BMC than Induction is

- BMC



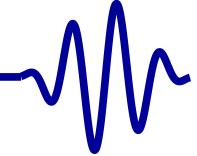
- Cover



- 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  
 $\forall \text{assume}() \Rightarrow \exists \text{cover}()$

[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](#)

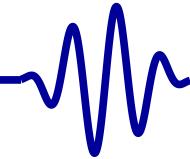
Just like an assumption or an assertion

```
// 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](#)

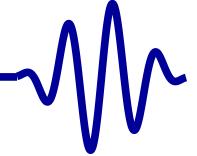
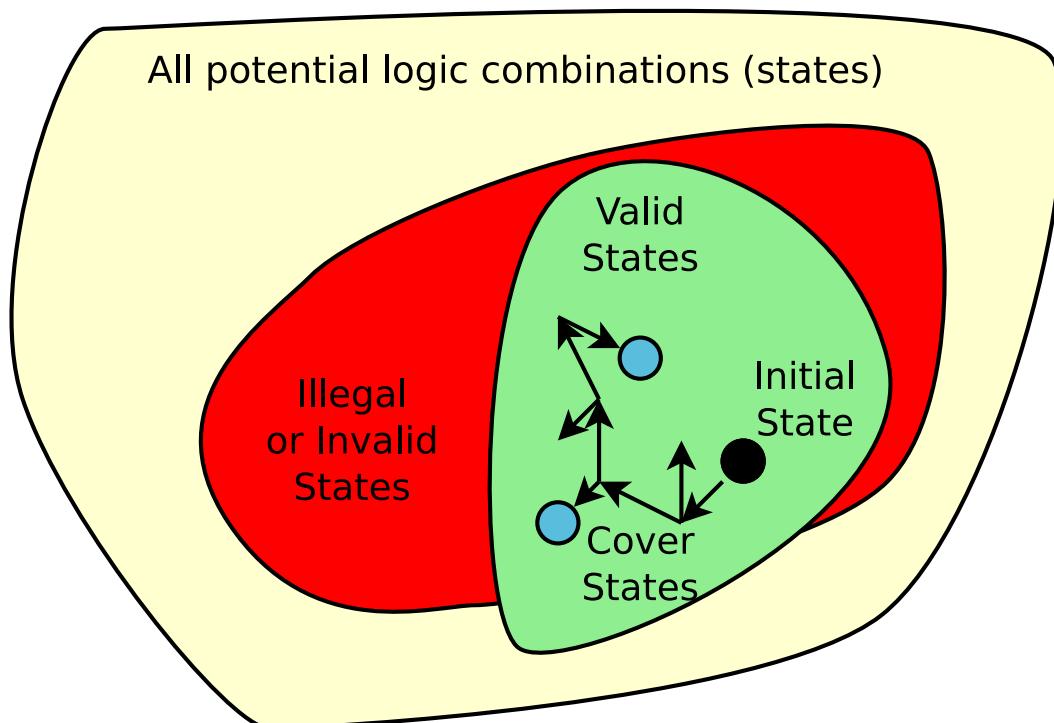
Assert and cover handle surrounding logic differently

- Assert logic
- Cover logic

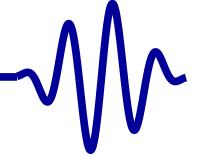
```
always @(posedge i_clk)
  if (A)
    cover(B);
```

is equivalent to,

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

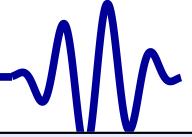
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[BMC vs Cover](#)[Cover in Verilog](#)[▷ State Space](#)[SymbiYosys](#)[Examples](#)[Counter](#)[Sequences](#)[Quizzes](#)

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

[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



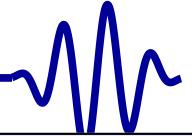
Welcome  
Motivation  
Basics  
Clocked and \$past  
 $k$  Induction  
Bus Properties  
Free Variables  
Abstraction  
Invariants  
Multiple-Clocks  
Cover  
Lesson Overview  
BMC vs Cover  
Cover in Verilog  
State Space  
▷ SymbiYosys  
Examples  
Counter  
Sequences  
Quizzes

```
[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
```

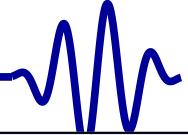


```
[options]
mode cover
depth 40 ← How far to look for a covered state
append 20

[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

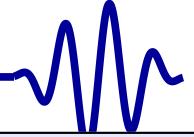
[engines]
smtbmc

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

[files]
# file list
```



# SymbiYosys tasks



- [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](#)

[ **tasks** ]

prf

cvr

[ **options** ]

prf : **mode** prove

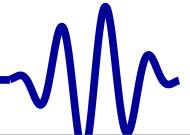
cvr : **mode** cover

**depth** 40

# . . .



# SymbiYosys tasks



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

[ **tasks** ]

prf ← Run two tasks: prf and cvr

cvr

[ **options** ]

prf : **mode** prove

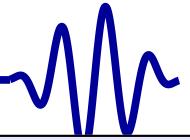
cvr : **mode** cover

**depth** 40

# . . .



# SymbiYosys tasks



- >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

[ **tasks** ]

prf

cvr

[ **options** ]

prf : **mode** prove ← The prf tasks runs induction

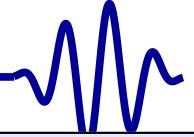
cvr : **mode** cover

**depth** 40

# . . .



# SymbiYosys tasks



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

[ **tasks** ]

prf

cvr

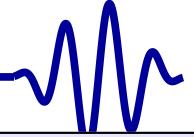
[ **options** ]

prf : **mode** prove

cvr : **mode** cover ← The cvr tasks runs in cover mode

**depth** 40

# . . .



- 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

[ **tasks** ]

prf

cvr

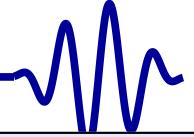
[ **options** ]

prf : **mode** prove

cvr : **mode** cover

**depth** 40 ← The same depth can apply to both

# . . .



[ **tasks** ]

prf

cvr

[ **options** ]

prf: **mode** prove

cvr: **mode** cover

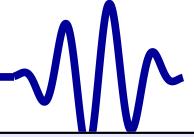
**depth** 40

# ...

% sby -f sbyfil.sby now runs both modes



# SymbiYosys tasks



- >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

[ **tasks** ]

prf

cvr

[ **options** ]

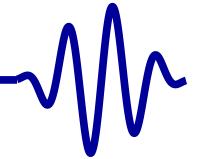
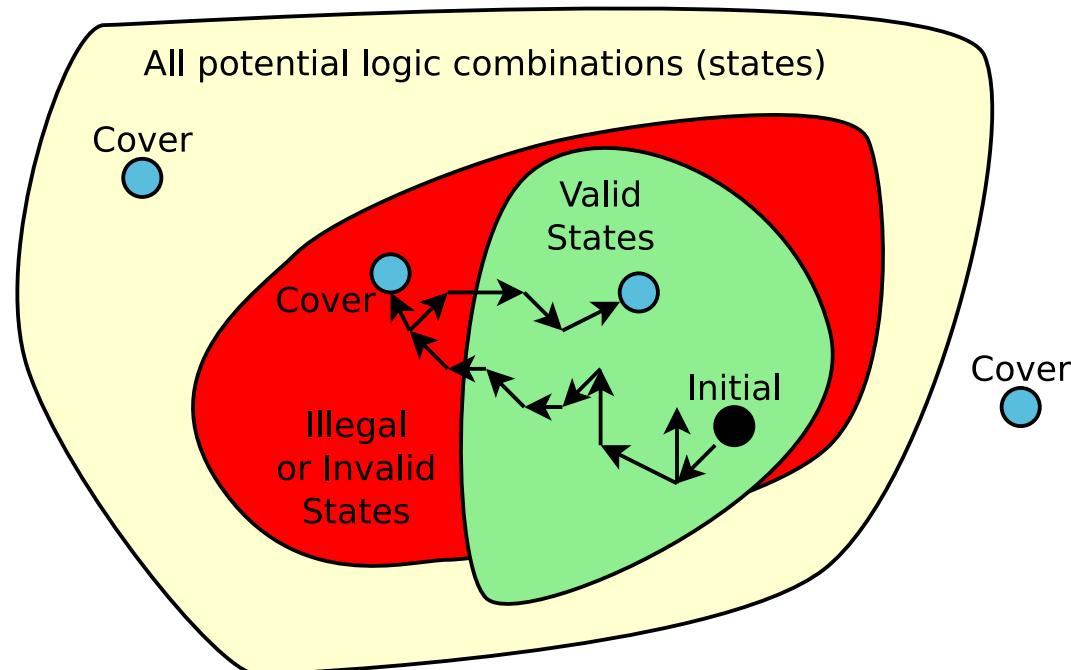
prf : **mode** prove

cvr : **mode** cover

**depth** 40

# . . .

% sby -f sbyfil.sby cvr will run the cover mode alone

[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



# Ex: I-Cache



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

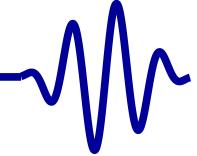
Consider a CPU I-cache:

```
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 HDL!

[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](#)

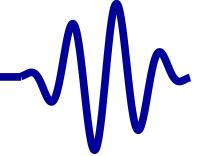
Consider a Flash controller:

```
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

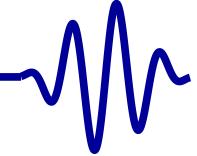
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[BMC vs Cover](#)[Cover in Verilog](#)[State Space](#)[SymbiYosys](#)[▷ Examples](#)[Counter](#)[Sequences](#)[Quizzes](#)

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

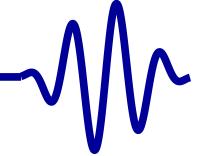
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[BMC vs Cover](#)[Cover in Verilog](#)[State Space](#)[SymbiYosys](#)[▷ Examples](#)[Counter](#)[Sequences](#)[Quizzes](#)

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

[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](#)

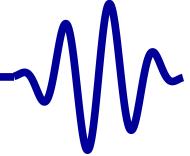
Remember our counter?

```
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);

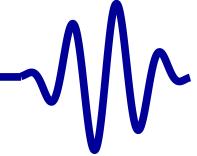
// Transition back to idle
always @(posedge i_clk)
if ((f_past_valid)&&($past(o_busy)))
    cover(!o_busy);

// 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

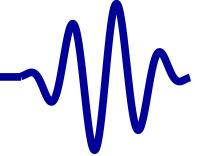
Sequences

Quizzes

How about now?

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

# GT 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));
```

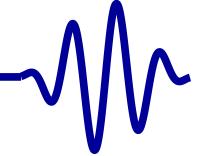
Or this one,

```
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,

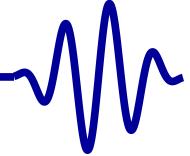
```
always @(posedge i_clk)
    cover(counter == MAX_AMOUNT);
```

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

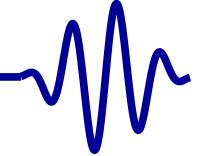
What if the state is unreachable?

```
// 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

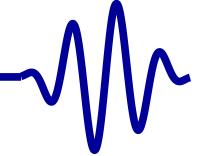
Quizzes

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

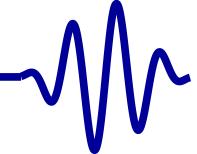
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[BMC vs Cover](#)[Cover in Verilog](#)[State Space](#)[SymbiYosys](#)[Examples](#)[▷ Counter](#)[Sequences](#)[Quizzes](#)

What if an assertion needs to be violated?

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

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

What will happen here?

[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](#)

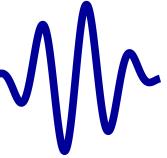
What if an assertion needs to be violated?

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

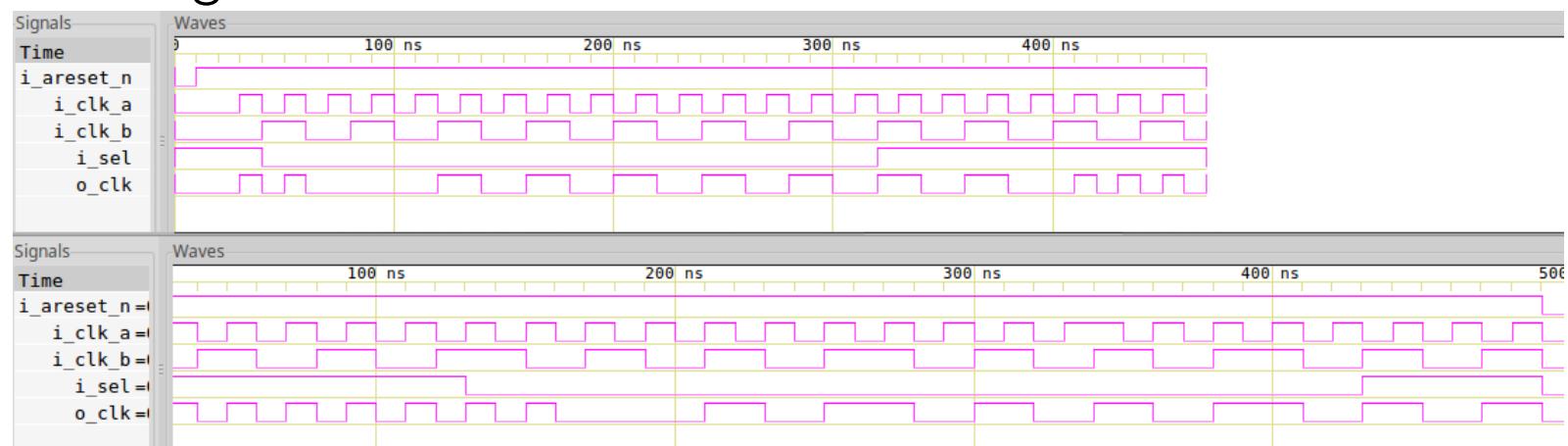
```
always @(posedge i_clk)  
    cover(counter == 4);
```

What will happen here?

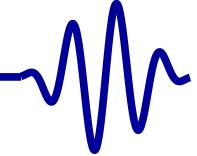
- Cover statement is reachable
- But requires an assertion failure, so a trace is generated

[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

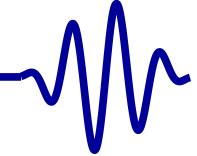
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[BMC vs Cover](#)[Cover in Verilog](#)[State Space](#)[SymbiYosys](#)[Examples](#)[▷ Counter](#)[Sequences](#)[Quizzes](#)

Return to your Wishbone arbiter. Let's cover four 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 each cases. Do they look right?

- If everything works, the first case showing both A and B receiving the bus will FAIL
- No trace is needed from that case
- After getting this failure, you may want to remove it from your cover checks

[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](#)

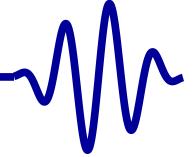
Notice what we just proved:

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

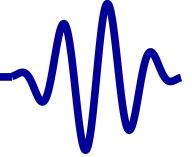
Examples

▷ 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

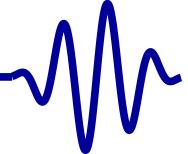
Bind

Sequences

Questions?

Quizzes

# Sequences

[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,

```
assert property (@(posedge i_clk)  
                  A |-> B);
```

Note that this is now a *clocked* assertion, but otherwise it's equivalent

[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,

```
assert property (@(posedge i_clk)
                  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

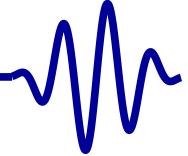
```
always @(posedge i_clk)
if ((f_past_valid)&&($past(A)))
    assert(B);
```

Can be rewritten as,

```
assert property (@(posedge i_clk)
                  A |=> 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](#)

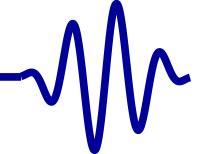
You can also declare properties:

```
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](#)

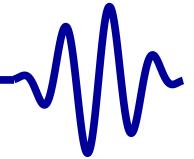
You could also do something like:

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

generate if (F_SUBMODULE)
begin
    assume property(INPUT_PROP);
end else begin
    assert property(INPUT_PROP);
end endgenerate

assert property(LOCAL_PROP);
assert property(OUTPUT_PROP);
```

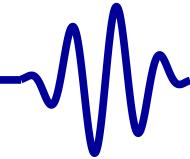
This would work quite nicely for a bus property file

[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);
  @(posedge i_clk)
  a |-> b;
endproperty

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

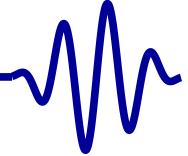
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[▷ Overview](#)[Clocking](#)[Bind](#)[Sequences](#)[Questions?](#)[Quizzes](#)

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](#)

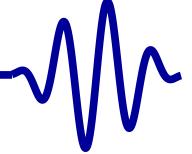
Getting tired of writing @(**posedge i\_clk**)?

- 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?

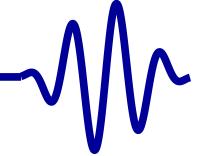
Quizzes

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.

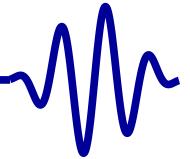
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[▷ Clocking](#)[Bind](#)[Sequences](#)[Questions?](#)[Quizzes](#)

When using verific, **\$global\_clock** must first be defined

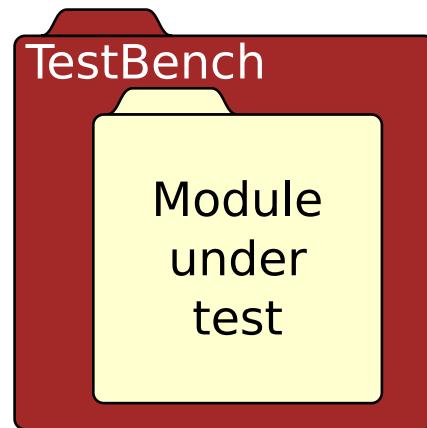
```
(* gclk *) wire gbl_clk;  
global clocking @(posedge gbl_clk); endclocking
```

This defines the **\$global\_clock** ...

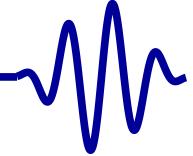
- as a positive edge transition of gbl\_clk.
- The (\* gclk \*) attribute turns it into a formal timestep



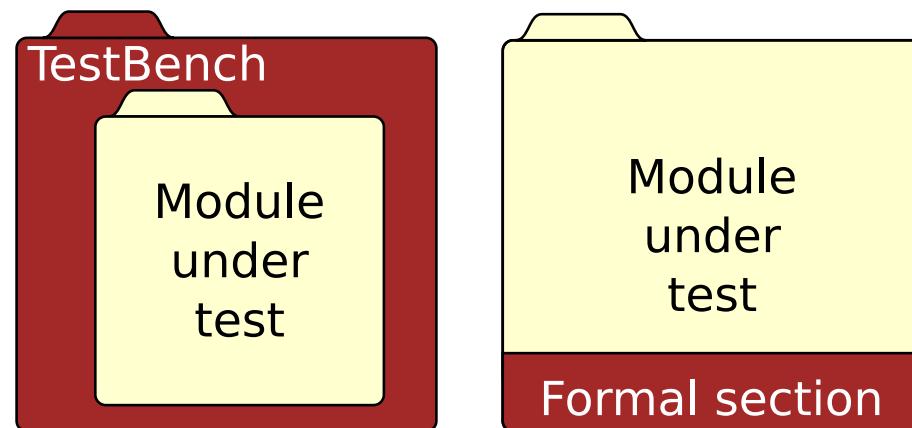
- Welcome
- Motivation
- Basics
- Clocked and \$past
- $k$  Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Cover
- Sequences
- Overview
- Clocking
- ▷ Bind
- Sequences
- Questions?
- Quizzes



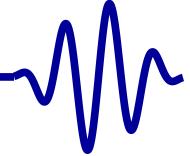
- Common bench testing works on black boxes
- This doesn't work well with formal methods



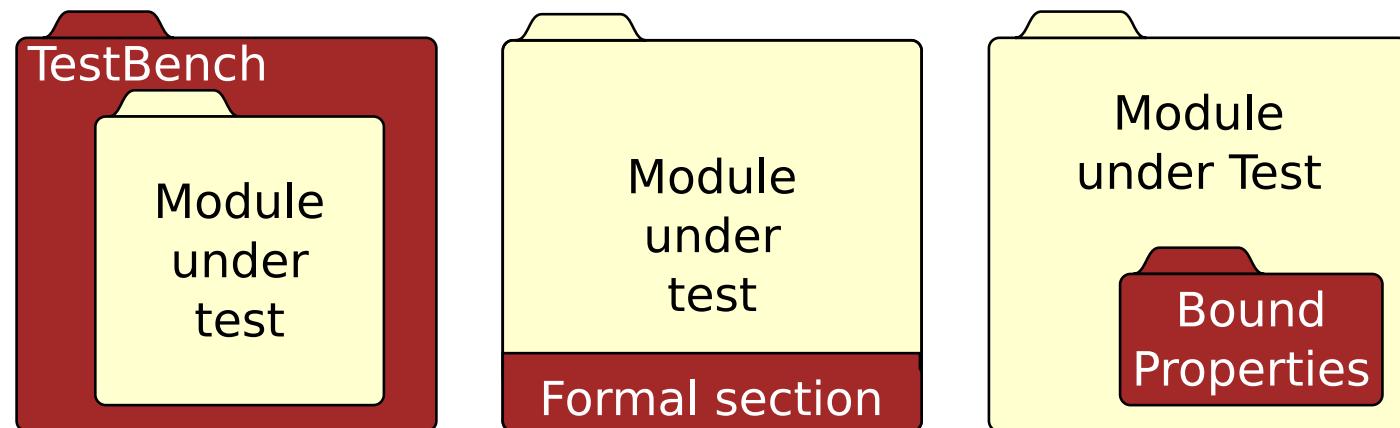
- Welcome
- Motivation
- Basics
- Clocked and \$past
- $k$  Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Cover
- Sequences
- Overview
- Clocking
- ▷ Bind
- Sequences
- Questions?
- Quizzes



- 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



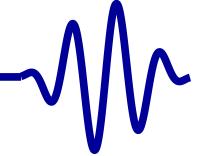
- Welcome
- Motivation
- Basics
- Clocked and \$past
- k Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Cover
- Sequences
- Overview
- Clocking
- ▷ Bind
- Sequences
- Questions?
- Quizzes



- 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](#)

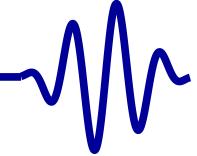
- Can bind to specific named variables
- Can also make *all* variables available to your properties

```
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

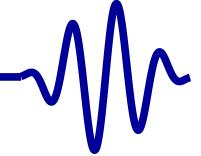
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[Sequences](#)[Questions?](#)[Quizzes](#)

- 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 (.*);
```

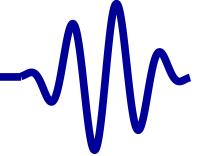
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

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

# 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

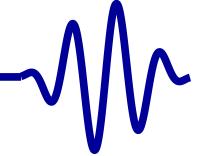
With sequences, you can

- Specify a series of actions

```
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

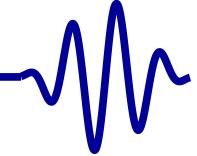
With sequences, you can

- 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?

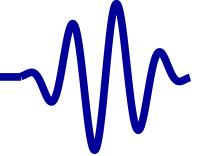
Quizzes

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

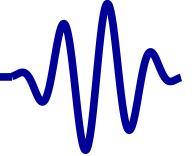
▷ Sequences

Questions?

Quizzes

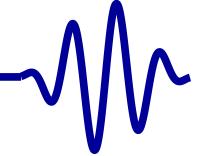
With sequences, you can

- 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](#)

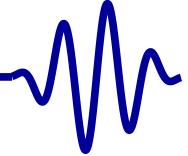
- Throughout

```
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

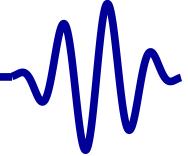
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

- 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](#)

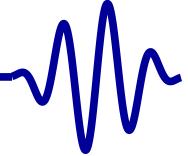
- Throughout
- Until

```
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

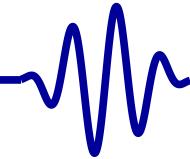
Properties can reference sequences

- 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

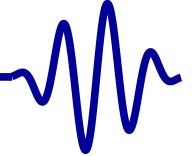
Properties can include . . .

- **if** statements

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

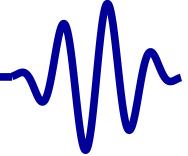
- **not**, **and**, or even **or** statements

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



A bus request will not change until it is accepted

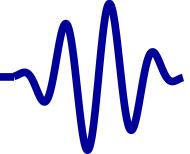
```
property BUS_REQUEST_HOLD;  
  @(posedge i_clk)  
  ( STB)&&(STALL)  
  |=> ( STB)&&($stable(REQUEST));  
endproperty  
  
assert property ( BUS_REQUEST_HOLD);
```

[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](#)

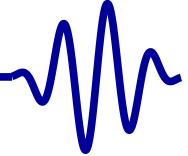
A request persists until it is accepted

```
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.

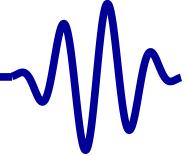
Could we do this with an **until** statement?

[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 );
```

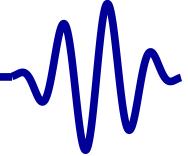
What is the difference?

[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.

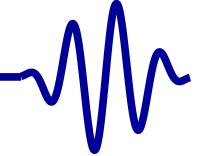
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

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](#)

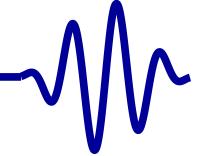
A property can be conditionally disabled

```
sequence BUS_REQUEST;
    // 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?

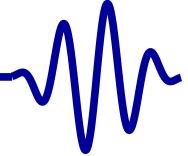
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

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?

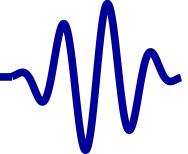
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

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](#)

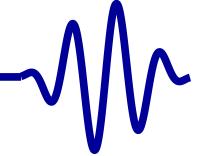
Some peripherals will only ever accept one request

```
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](#)

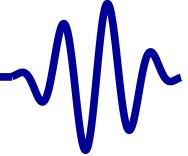
Some peripherals will

- 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?

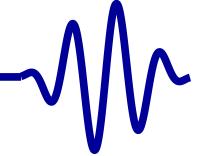
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

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?

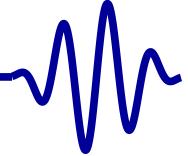
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

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](#)

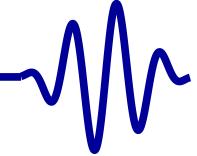
Cannot ACK or ERR when no request is pending

```
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 . . .

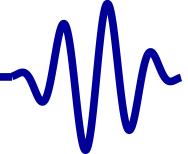
- Output data is constant
- Logic doesn't change state
- Internal shift register value is known
- Ends with zero\_baud\_counter

```
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
```

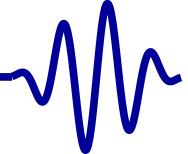
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

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](#)

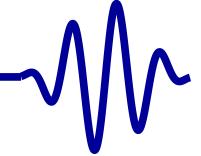
Using SystemVerilog Assertions with Yosys requires Verific

```
[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

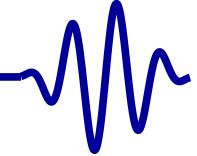
Questions?

Quizzes

## SystemVerilog Concurrent Assertions . . .

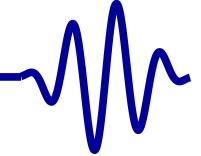
- can be very powerful
- can be very confusing
- can be used with immediate assertions

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](#)

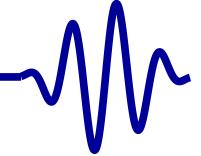
Let's formally verify a synchronous FIFO

```
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

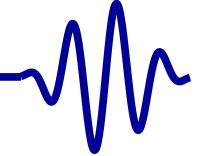
Let's formally verify a synchronous FIFO  
What properties do you think would be appropriate?

[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?

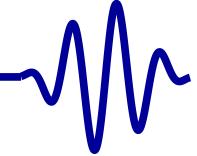
- Should never go from full to empty

[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?

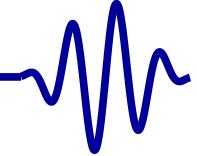
- Should never go from full to empty except on a reset

[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?

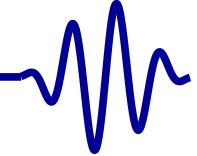
- Should never go from full to empty except on a reset
- Should never go from empty to full

[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?

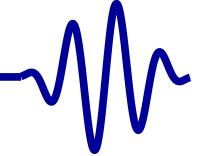
- Should never go from full to empty except on a reset
- Should never go from empty to full
- The two outputs, `o_empty` and `o_full`, should properly reflect the size of the FIFO
  - `o_empty` means the FIFO is currently empty
  - `o_full` means the FIFO has  $2^N$  elements within it

[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)

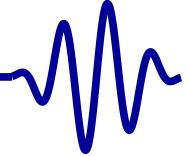
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Bind](#)[▷ Sequences](#)[Questions?](#)[Quizzes](#)

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:

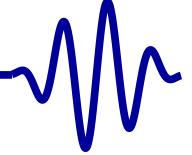
```
wire f_a, f_b, f_c;  
//  
assign f_a = // your logic  
assign f_b = // your logic  
assign f_c = // your logic  
//  
sequence ARBITRARY_EXAMPLE_SEQUENCE  
    f_a [*0:4] ##1 f_b ##1 f_c [*12:16];  
endsequence
```

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

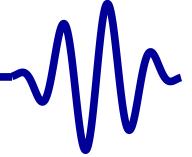
Clocking

Bind

Sequences

▷ Questions?

Quizzes



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

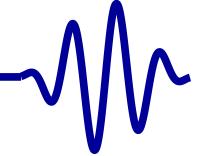
Sequences

▷ 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

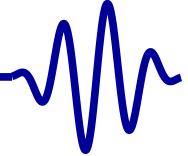
Will this simple counter ever pass formal verification?

```
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;

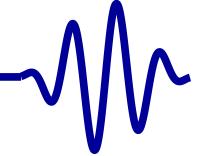
always @ (posedge i_clk)
if ((i_start_signal)&&(counter == 0))
    counter <= MAX_AMOUNT - 1'b1;
else if (counter != 0)
    counter <= counter - 1;

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);
```

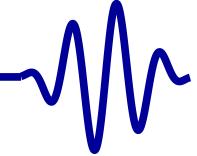
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

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

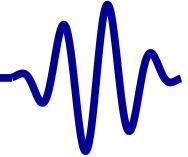
Quizzes

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.

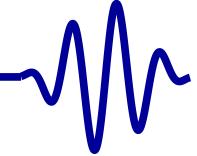
```
initial counter = 0;
always @(posedge i_clk)
if ((i_start_signal)&&(counter == 0))
    counter <= 23;
else if (counter != 0)
    counter <= counter - 1'b1;

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})));
```

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

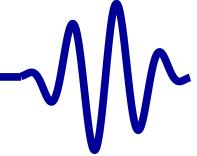
- The first assertion was an “immediate” assertion, the second a “concurrent assertion”.
- While the Symbiotic EDA Suite supports both assertions, the free version of Yosys only supports immediate assertions
- The second assertion is more compact, and perhaps even easier to read

```
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?

```
(* gclk *) reg gbl_clk;  
always @ (posedge gbl_clk)  
if ($fell(i_clk))  
    assert ($stable(signal));
```

```
always @ (posedge gbl_clk)  
if (! $rose(i_clk))  
    assert ($stable(signal));
```

```
always @ (posedge gbl_clk)  
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 @(posedge gbl_clk)
if ((f_past_valid_gbl)&&(!$rose(i_clk)))
    assert($stable(signal));
```

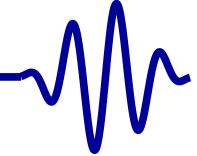
Be aware, **\$rose()** depends upon the **\$past()**, so don't forget an **f\_past\_valid** signal!

With (\* **gclk** \*), I like to call it **f\_past\_valid\_gbl**, and define it as,

```
reg f_past_valid_gbl = 1'b0;
always @(posedge gclk)
    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);
```

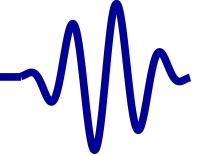
Alternatively, the following concurrent assertion would also work:

```
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

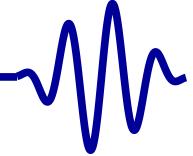
Cover

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)?

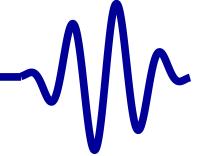
```
reg [15:0] counter;  
  
initial counter = 0;  
always @ (posedge clk)  
    counter <= counter + 1'b1;  
  
always @ (*)  
    assert(counter < 16'd65000);
```

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

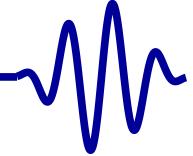
Cover

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

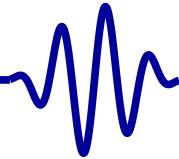
Sequences

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

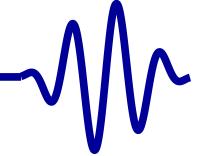
Cover

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 design. 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

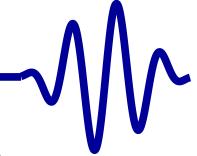
Sequences

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]))));
```



# Quiz #14



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?

```
(* gclk *) reg gbl_clk;
(* 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 @ (posedge gbl_clk) 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.

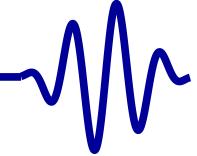
```
always @(*)
begin
    assume(f_step_one != 0);
    assume(f_step_two != 0);
    assume(f_step_one <= 8'h80);
    assume(f_step_two <= 8'h80);
end
```

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

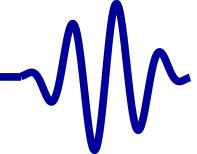
Sequences

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

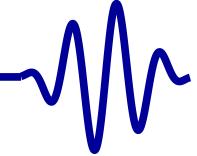
always @(posedge i_clk)
if ((f_past_valid)
    &&($past(i_write))&&($past(i_read))
    &&($past(i_waddr)==$past(i_raddr)))
    assert(o_data == $past(i_data));
```

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](#)

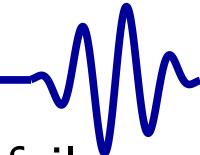
The following assertion can be used to describe the response of logic to a negative logic asynchronous reset.

```
always @(posedge i_clk, negedge i_areset_n)
  if (!i_areset_n)
    a <= 0;
  else
    a <= something;

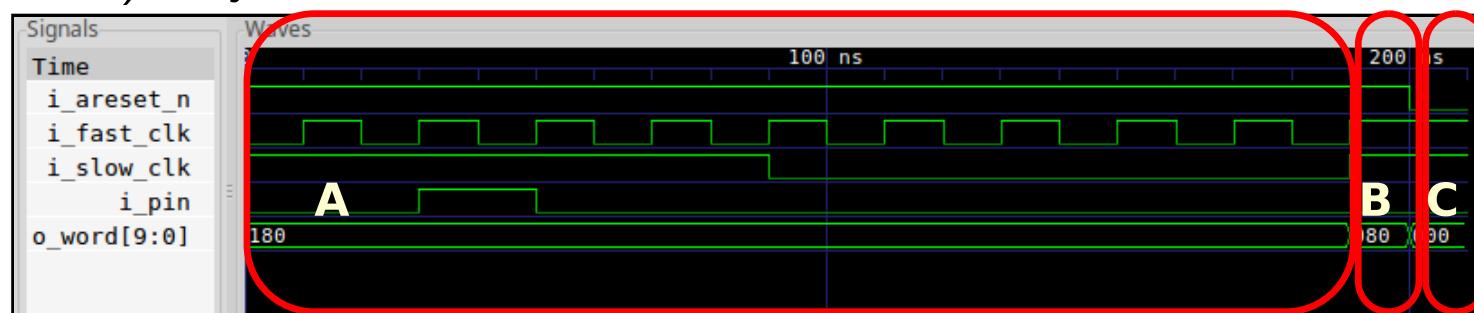
always @(posedge i_clk)
  if (!i_areset_n)
    assert(a == 0);
  else if ((f_past_valid)&&($past(i_areset_n))
    assert(a == $past(something));
```

Don't forget to assume an initial reset!

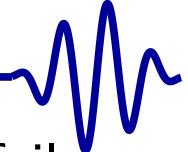
```
initial assume(!i_areset_n);
```

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

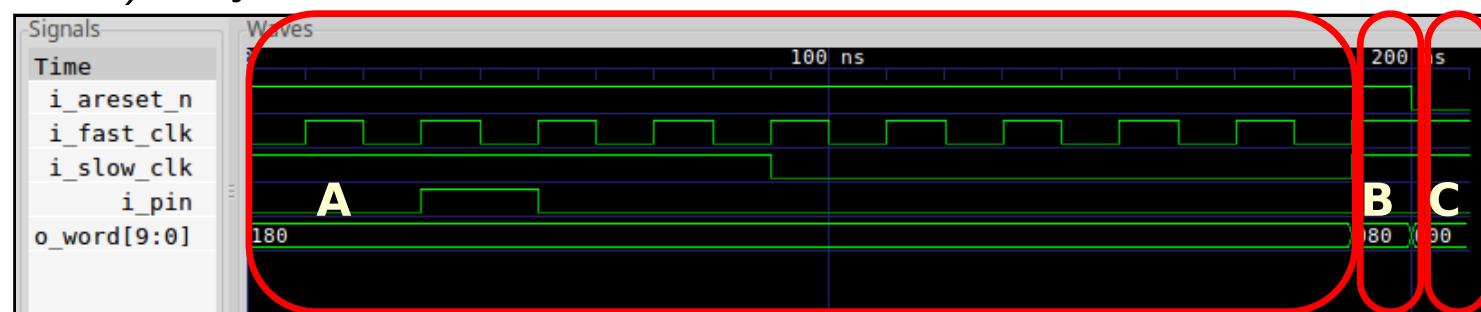
Your design passes a bounded model check (BMC), but fails during induction. Upon inspection, you find a failure in section A (below) of your trace.



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](#)

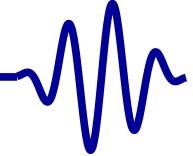
Your design passes a bounded model check (BMC), but fails during induction. Upon inspection, you find a failure in section A (below) of your trace.



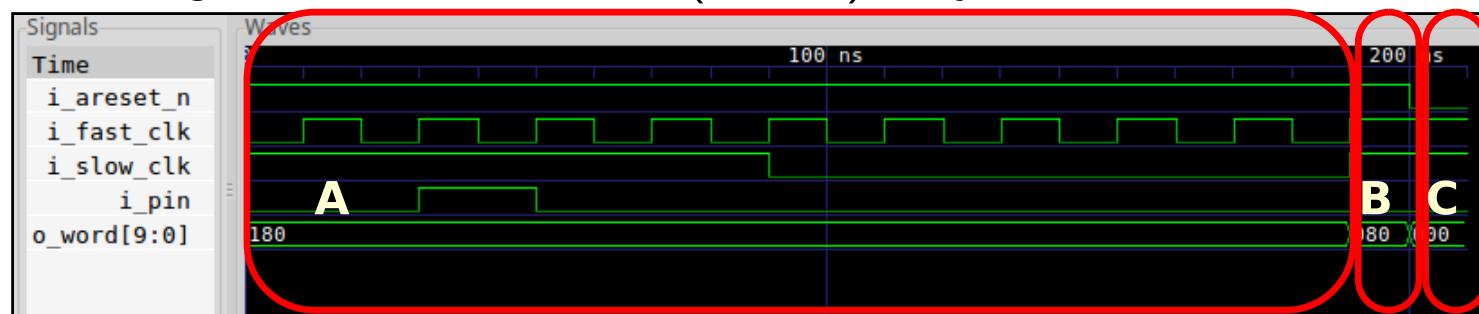
How should you address this problem?

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

[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.



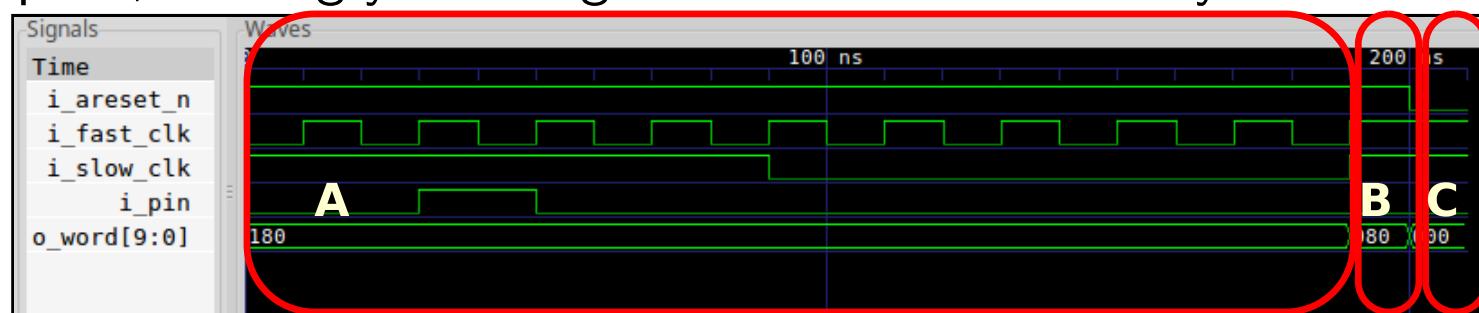
Upon inspection, you discover an

**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)

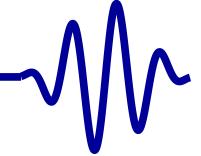
- This can cause an **always @(\*) assert(Y);** to fail in section C

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

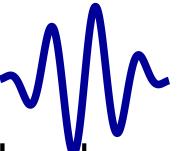
Quizzes

Will the following design pass formal verification?

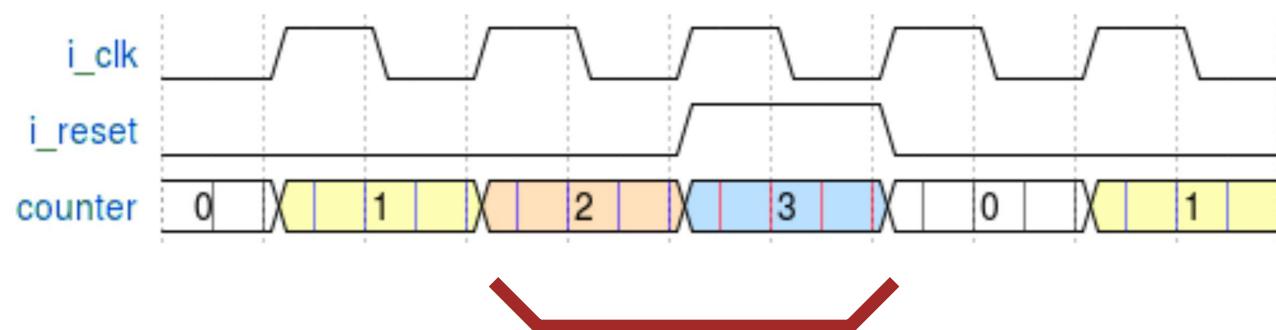
```
reg [15:0] counter = 0;
always @(posedge i_clk)
if (i_reset)
    counter <= 0;
else
    counter <= counter + 1;

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.



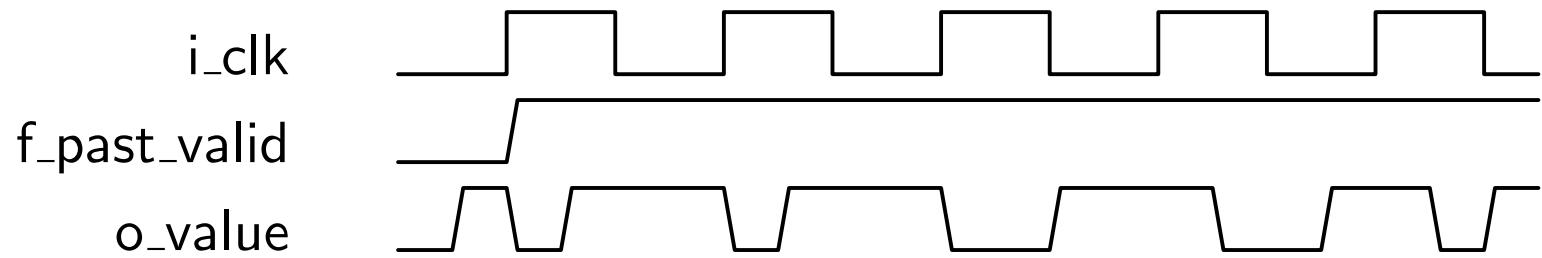
disable iff (*i\_reset*) disables the check across both of these cycles

This is roughly equivalent to:

```
reg      check = 1;
always @(*posedge i_clk)
    check <= (counter < 2)||(i_reset);
always @(*)
    if (!i_reset) assert(check);
```

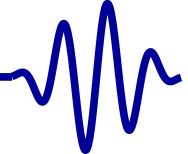
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Consider the following trace from an asynchronous context:

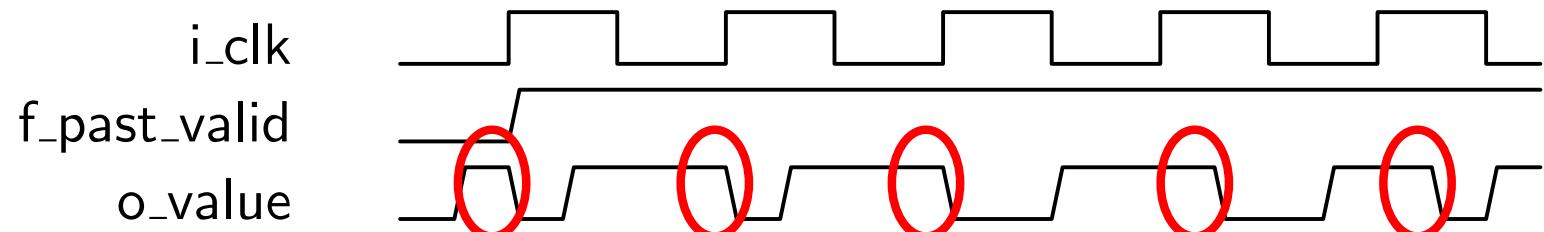


Will this formal stability assertion pass or fail?

```
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));
```



# Quiz #21



Welcome

Motivation

Basics

Clocked and \$past

*k* Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

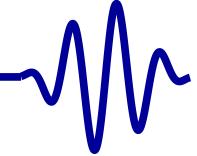
Sequences

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

—pycode—begin—

cmd=" hierarchy --top\_toplvl "

cmd+=" -chparam\_A %d " % (1 if "A" in tags else 0)

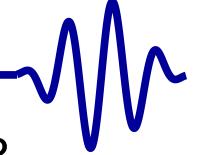
cmd+=" -chparam\_B %d " % (1 if "B" in tags else 0)

output(cmd)

—pycode—end—

**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.

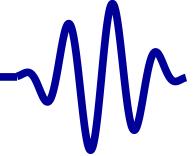
```
always @(*)  
    cover(step_24);
```

- 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:

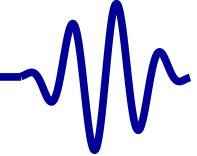
```
input wire [31:0] i_v;
output wire o_v;

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](#)

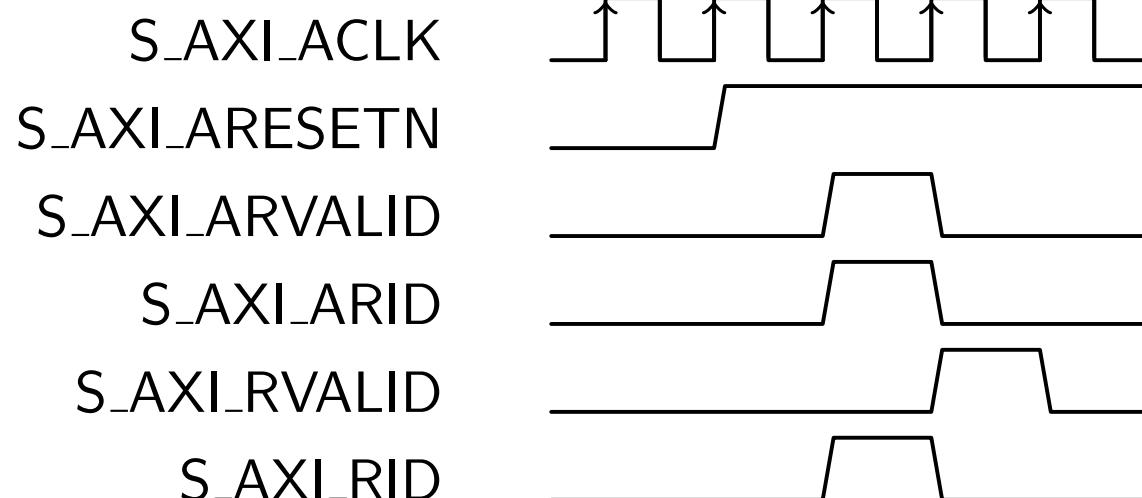
Consider the following design:

```
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

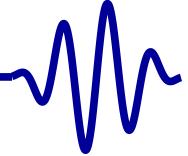
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Consider the following trace from an AXI read interaction:

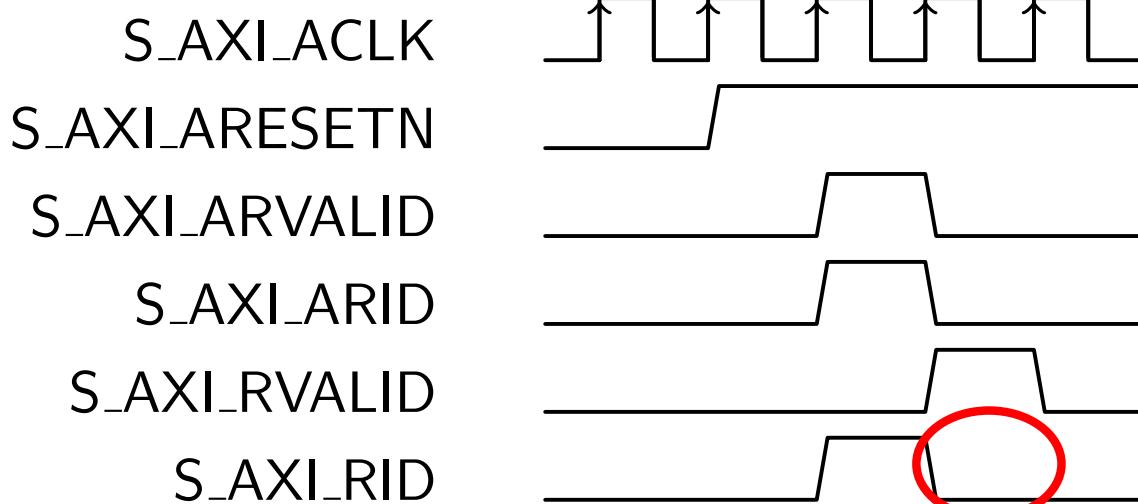


- Assume all of the relevant xREADY lines are high

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?



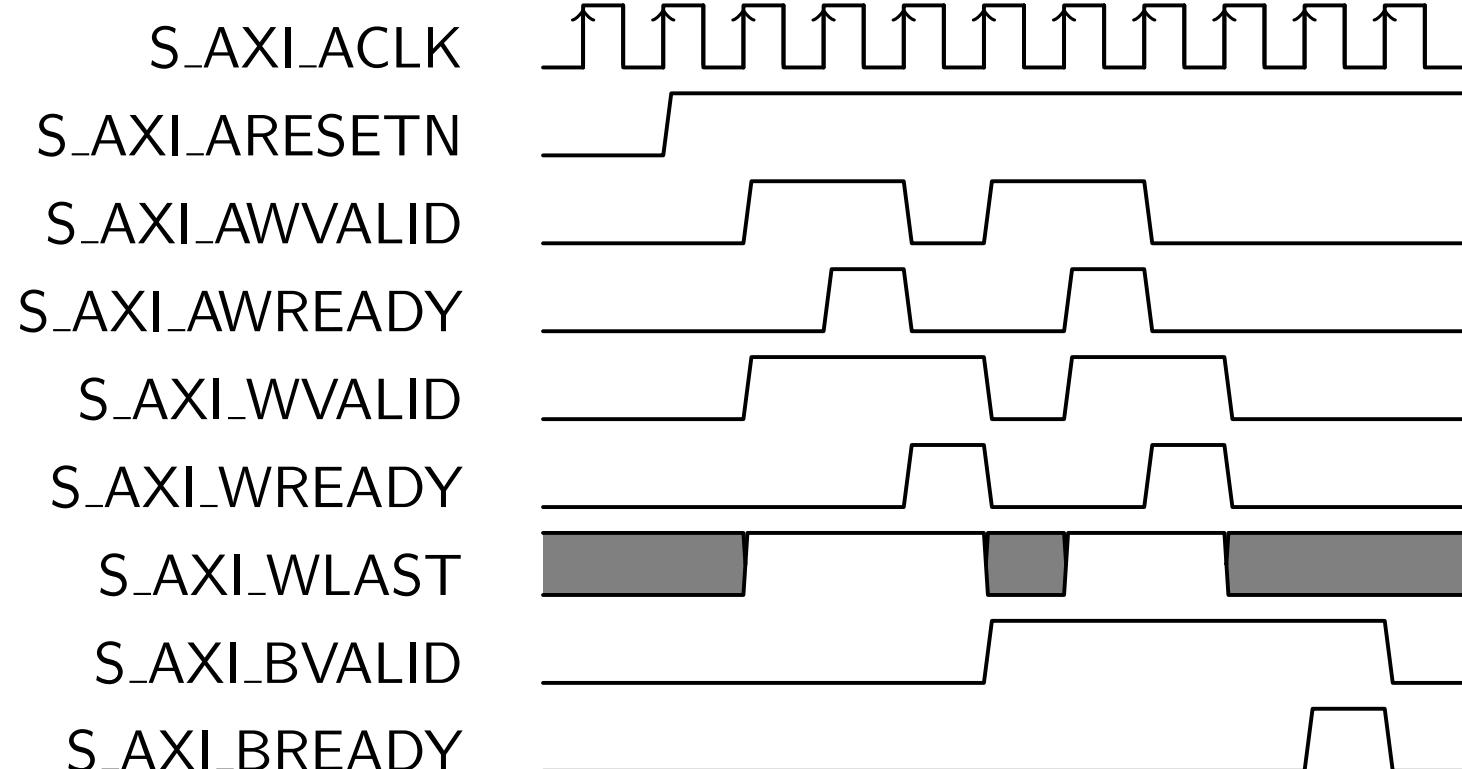
The request response has the wrong ID

- Request was made for ID=1, response has ID=0
- The cause? Xilinx's example core doesn't register the ID

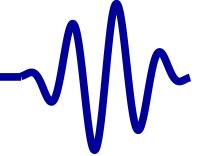
The trace above was found by applying the Symbiotic EDA Suite to Xilinx's example AXI4 core

[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](#)

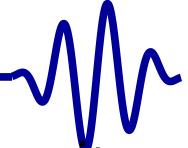
A transaction timeout can find this bug

```
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.

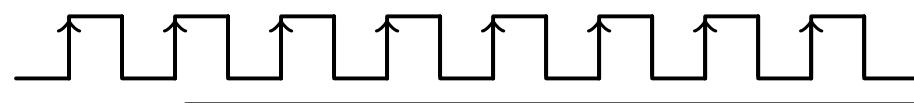
```
always @ (posedge i_clk)
if ((!i_reset_n) || (i_bvalid) || (i_wvalid)
    || ((f_awr_nbursts == 1)
        && (f_wr_pending > 0))
    || (f_awr_nbursts == 0))
    f_awr_ack_delay <= 0;
else
    f_awr_ack_delay <= f_awr_ack_delay + 1'b1;

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

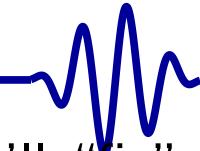


S\_AXI\_BVALID



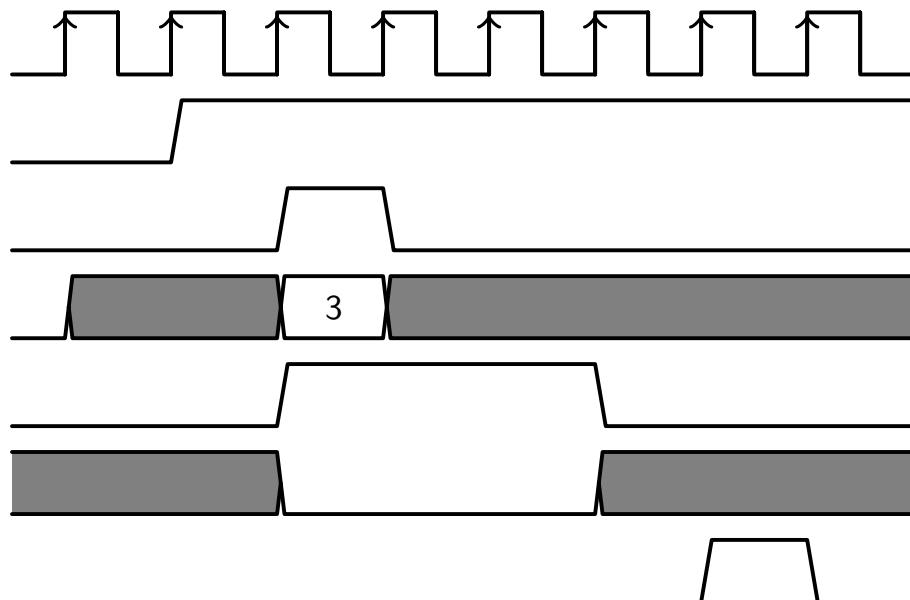
Assume all \*READY signals are true

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 design.

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),
        // ...
    )
```

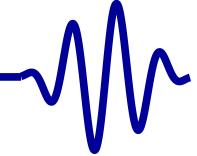
and use it to **assert()** that the state matches your logic

```
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

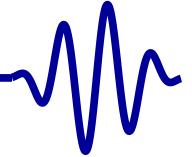
Sequences

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](#)

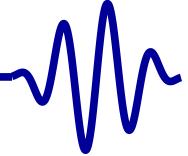
The first bug is not setting the pointers initially

```
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](#)

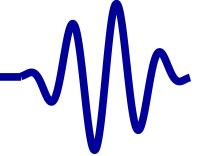
The real problem is that the whole structure is wrong.

- 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.

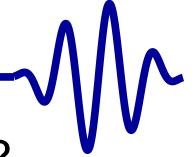
```
reg      f_past_valid = 0;
always @(posedge i_clk)
    f_past_valid <= 1;

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);
```

[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?

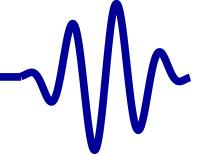
- The `o_empty` flag

```
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?

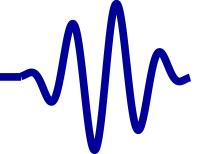
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

## 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](#).

[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;
```

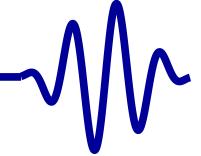
- Then when the bus returns a value for the given address, **assume** the known value.

```
if (i_wb_ack && ackd_address == f_const_addr)  
    assume(i_wb_data == f_const_data);
```

- 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);
```

- 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 design 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 design illustrates a common AXI coding mistake:

```
always @ (posedge S_AXI_ACLK)
// ...
if (S_AXI_AWVALID && S_AXI_AWREADY
    && something_else)
    // ...
```

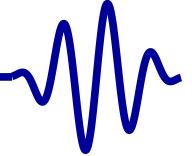
The mistake? Checking for `something_else` when processing information from the bus. To fix it,

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

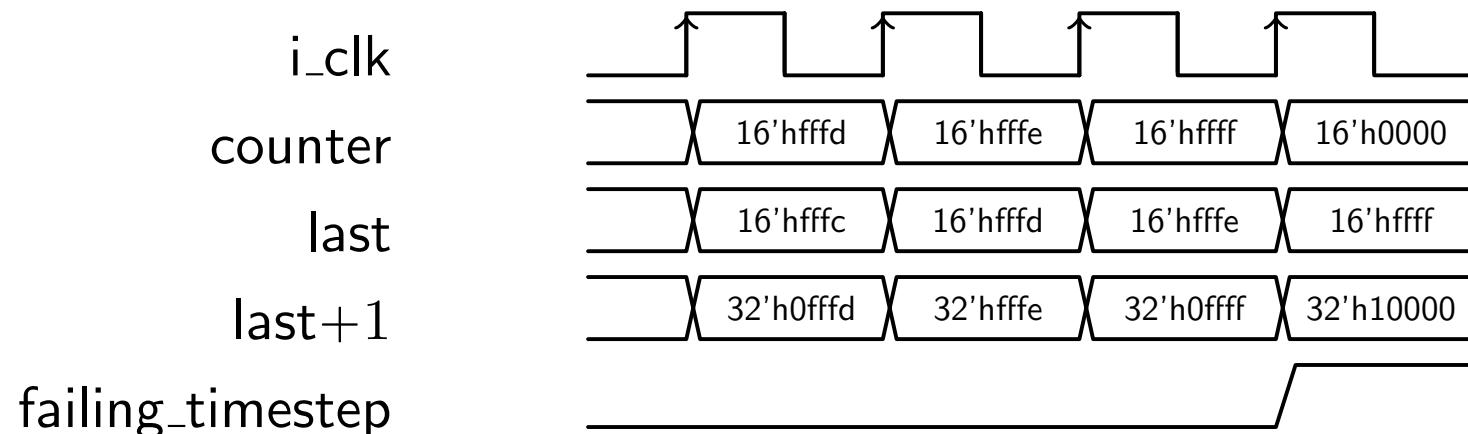
Quizzes

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.

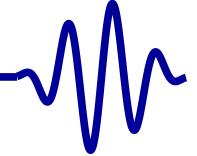


If you map `last+1` to a 16-bit value, the assertion will pass

```
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

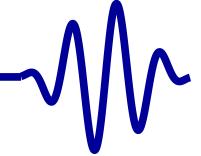
Sequences

Quizzes

The following design 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?

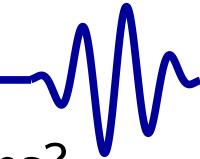
```
input    wire    [31:0]  i_a, i_b, i_c;  
  
always @(*)  
begin  
    assume(i_a+ i_b == 32'h4);  
    assume(      i_b +i_c == 32'h8);  
    assume(i_a+{ i_b, 1'b0}+i_c == 32'h7);  
end
```

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.



# Quiz #37



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

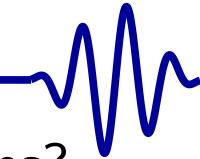
Sequences

Quizzes

What are the three most common bus interface properties?

1. Following a reset, the bus should return to an idle state and any pending requests should be dropped
2. If the bus is stalled, the request must not change
3. . . .

There's one other basic, yet common, bus interface property that's missing. What is it?

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

What are the three most common bus interface properties?

1. Following a reset, the bus should return to an idle state and any pending requests should be dropped
2. If the bus is stalled, the request must not change
3. *There should be one and only one response for every bus request*

I'll ask about the "contract" property to insure that the bus actually works next week



# Quiz #38



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

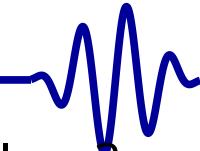
Sequences

Quizzes

None of the properties we examined last week truly expresses the “contract” associated with bus transactions. How should that contract be expressed for a generic bus component?

1. Let the solver pick an arbitrary address, and a value to be at that address
2. ...
3. Prove that reads from that address return the value from within the slave found at that address

What's the missing step?

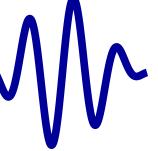
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

How should the formal contract be expressed for a bus slave?

1. Let the solver pick an arbitrary address, and a value to be at that address
2. *Adjust the value at that address following any write request*
3. Prove that reads from that address return the value from within the slave found at that address

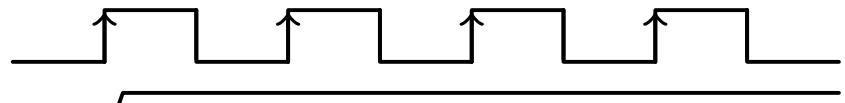
You should find these basic property steps common across many bus components

1. Not-so-generic bus slaves may need to use a slightly different approach, verifying instead that the result matches the value within the bus slave
2. Sequence is important, especially with AXI: the return value might be waiting for a RREADY longer than that return value accurately expresses the register's value within the core

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

Can you spot the AXI bug below?

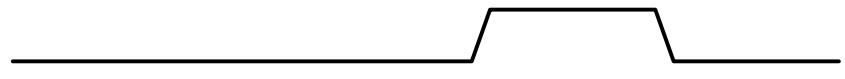
S\_AXI\_ACLK



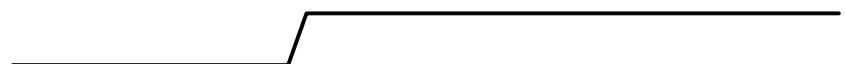
S\_AXI\_ARESETN



S\_AXI\_AVALID



S\_AXI\_AREADY



S\_AXI\_AWADDR



S\_AXI\_AWLEN



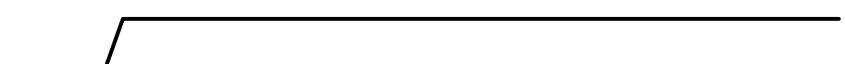
S\_AXI\_AWSIZE



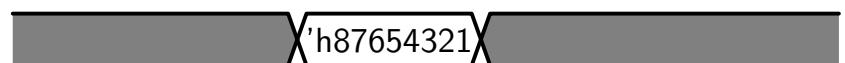
S\_AXI\_WVALID



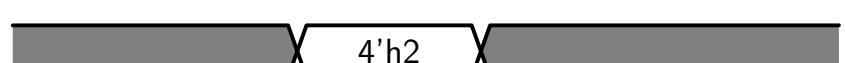
S\_AXI\_WREADY

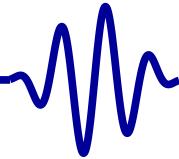


S\_AXI\_WDATA[31:0]



S\_AXI\_WSTRB[3:0]



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

Take a closer look at AWADDR, AWSIZE, and WSTRB

S\_AXI\_ACLK



S\_AXI\_AWVALID



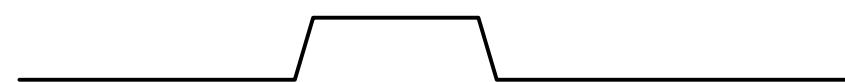
S\_AXI\_AWADDR



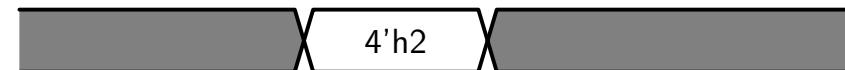
S\_AXI\_AWSIZE



S\_AXI\_WVALID



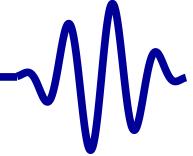
S\_AXI\_WSTRB[3:0]



If AWADDR ends in 4'h0, for an 8-bit transfer (AWSIZE=0),  
WSTRB can only be 4'h0 or 4'h1



# Quiz #40



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Consider the design below

```
reg      A, B, C, D, E, Z;  
always @(posedge clk)  
begin  
    // Assign to A, B, C, D, E, and Z somehow  
end  
  
assert property (@(posedge clk)  
    Z |=> (A && B && C && D && E));
```

Would you consider this to be a good or a bad assertion?

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

While the assertion below is *legal*,

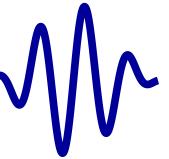
```
assert property (@(posedge clk))
    Z |=> (A && B && C && D && E);
```

because the assertion tests for the *and* of many conditions, it can be difficult to tell from a trace which condition caused the assertion failure. You might find that splitting it up makes it easier to work with.

```
assert property (@(posedge clk)) Z |=> A;
assert property (@(posedge clk)) Z |=> B;
assert property (@(posedge clk)) Z |=> C;
assert property (@(posedge clk)) Z |=> D;
assert property (@(posedge clk)) Z |=> E;
```



# Quiz #41



Welcome

Motivation

Basics

Clocked and \$past

*k* Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

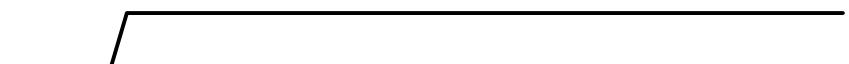
Quizzes

Can you spot the AXI bug below?

S\_AXI\_ACLK



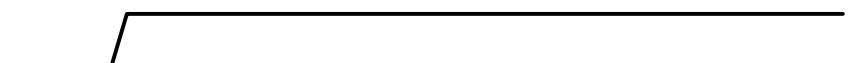
S\_AXI\_ARESETN



S\_AXI\_AWVALID



S\_AXI\_AWREADY



S\_AXI\_AWADDR



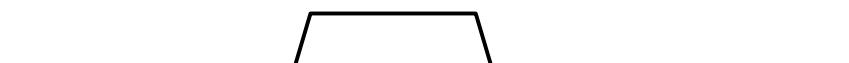
S\_AXI\_AWLEN



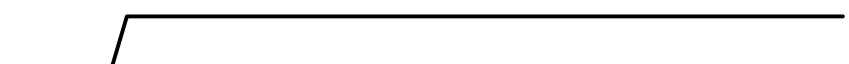
S\_AXI\_AWSIZE



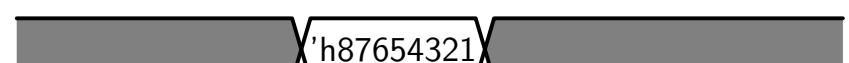
S\_AXI\_WVALID



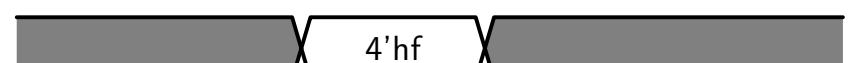
S\_AXI\_WREADY

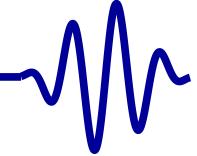


S\_AXI\_WDATA[31:0]



S\_AXI\_WSTRB[3:0]



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

Can you spot the AXI bug below?

S\_AXI\_AWVALID



S\_AXI\_AWADDR



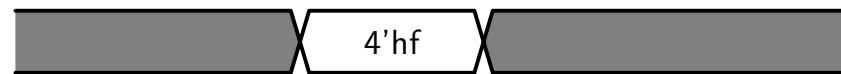
S\_AXI\_AWSIZE



S\_AXI\_WVALID

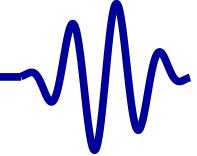


S\_AXI\_WSTRB[3:0]



1. If AWSIZE==1, then only two bits of WSTRB may ever be set on any given beat. These can either be 4'h3 or 4'hc for a 32-bit bus
2. If AWADDR[1:0]==2'b01, then only bit WSTRB[1] may be set

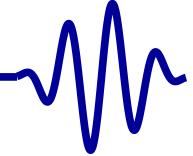
Note that AXI explicitly allows WVALID before AWVALID

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

Consider the design below

```
reg      A, B, C, D, Z;  
always @(posedge clk)  
begin  
    // Assign to A, B, C, D, and Z somehow  
end  
  
assert property (@(posedge clk)  
    Z |=> A  
    ##1 B [*0:$]  
    ##1 C  
    ##1 B [*0:$]  
    ##1 D);
```

Would you consider this to be a good or a bad assertion?

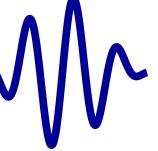
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

This assertion will never pass induction

```
assert property (@(posedge clk)
                  Z |=> A ##1 B [*0:$] ##1 C
                                ##1 B [*0:$] ##1 D);
```

Why?

- Because the induction engine doesn't start at  $t = 0$ 
  - There's no way to tell if the design is in the first B state or the second B state
- Worse, if B & C might ever hold, then the induction engine doesn't know how many times B was ever entered
  - The design might start with B true, and then set B & C for any number of clock ticks
  - The same applies to D

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

Is this a valid AXI read request?

S\_AXI\_ACLK

S\_AXI\_ARVALID

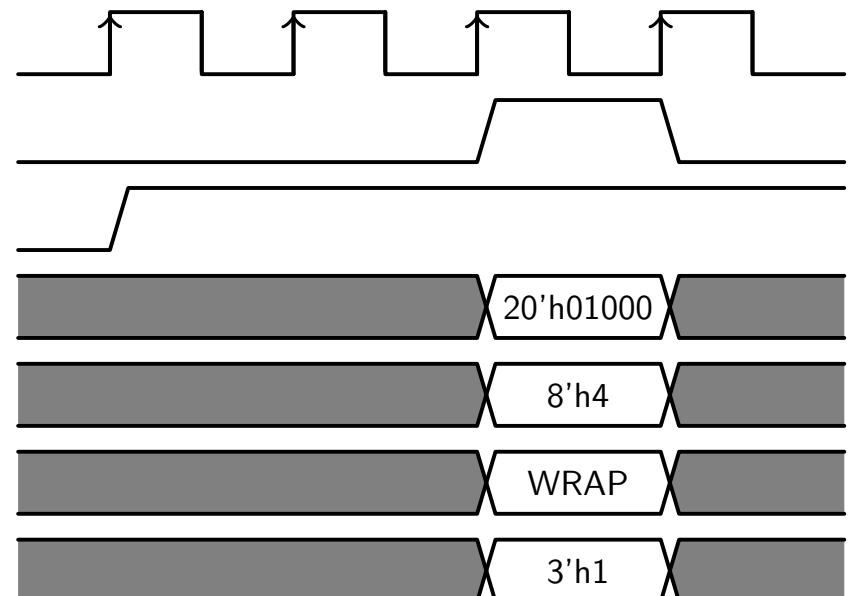
S\_AXI\_ARREADY

S\_AXI\_ARADDR

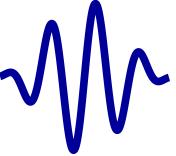
S\_AXI\_ARLEN

S\_AXI\_ARBURST

S\_AXI\_ARSIZE



You may assume the reset is inactive.

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

Is this a valid AXI read request?

S\_AXI\_ACLK



S\_AXI\_ARVALID



S\_AXI\_ARREADY



S\_AXI\_ARADDR



S\_AXI\_ARLEN

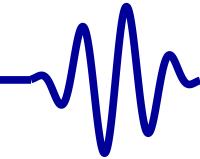


S\_AXI\_ARBURST



No.

- When using wrapped addressing, the burst length must be either 2, 4, 8 or 16.  
AxLEN must be one less than that length
- In this case, ARLEN = 4, indicating a burst length of 5.

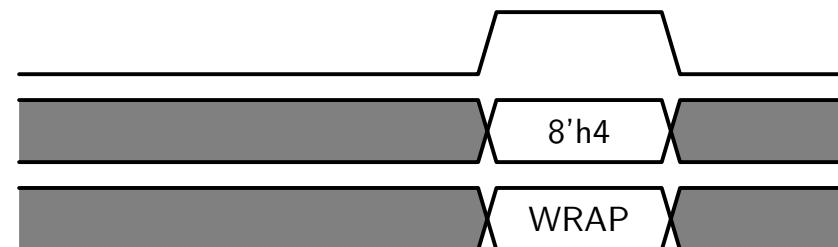
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

How would you detect this problem?

S\_AXI\_ARVALID

S\_AXI\_ARLEN

S\_AXI\_ARBURST



The following property would capture this check

```
always @(*)
  if ((S_AXI_ARVALID)&&(S_AXI_ARBURST == WRAP))
    assert((S_AXI_ARLEN == 8'h1)
           ||(S_AXI_ARLEN == 8'h3)
           ||(S_AXI_ARLEN == 8'h7)
           ||(S_AXI_ARLEN == 8'h15));
```

Be aware: Passing induction would take a bit more work

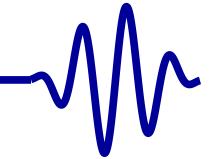
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

Consider the following FIFO design that passed its testbench

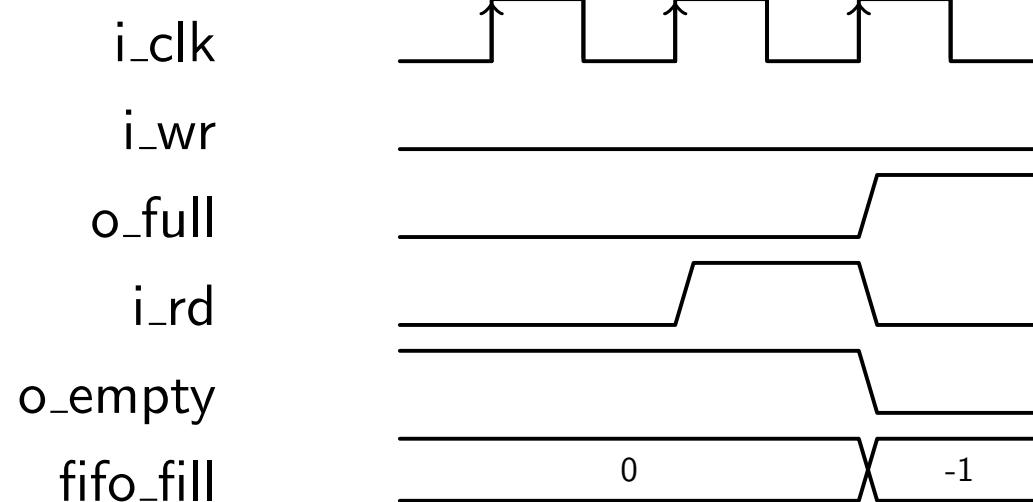
```
always @(posedge i_clk)
begin
    if (i_rd && !o_empty)
        rd_addr <= rd_addr + 1;
    if (i_wr && !o_full)
        wr_addr <= wr_addr + 1;
end

always @(posedge i_clk)
if (i_rd && !i_wr)
    fifo_fill <= fifo_fill - 1;
else if (i_wr && !i_rd)
    fifo_fill <= fifo_fill + 1;
```

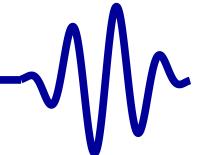
Ignoring the missing reset and initial states, and assuming o\_empty and o\_full are suitably defined, do you see any bugs?

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

Bugs in the FIFO? What about the following sequence?



Did you see any others? (There were more ...)

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

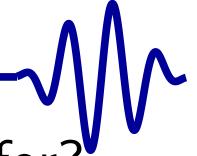
What formal properties might have found these bugs?

```
reg [LGFIFO:0] f_fifo_fill;  
  
always @(*)  
    f_fifo_fill = wr_addr - rd_addr;  
always @(*)  
    assert(f_fifo_fill == fifo_fill);
```

This one assertion would've caught these bugs. You could easily pivot from here and catch any o\_empty or o\_full errors as well,

```
always @(*)  
    assert(o_empty == (f_fifo_fill == 0));  
always @(*)  
    assert(o_full ==  
        (f_fifo_fill == (1<<LGFIFO))));
```

But this goes beyond what was in the quiz question.

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

What addresses and in what order is this request asking for?

S\_AXI\_ACLK

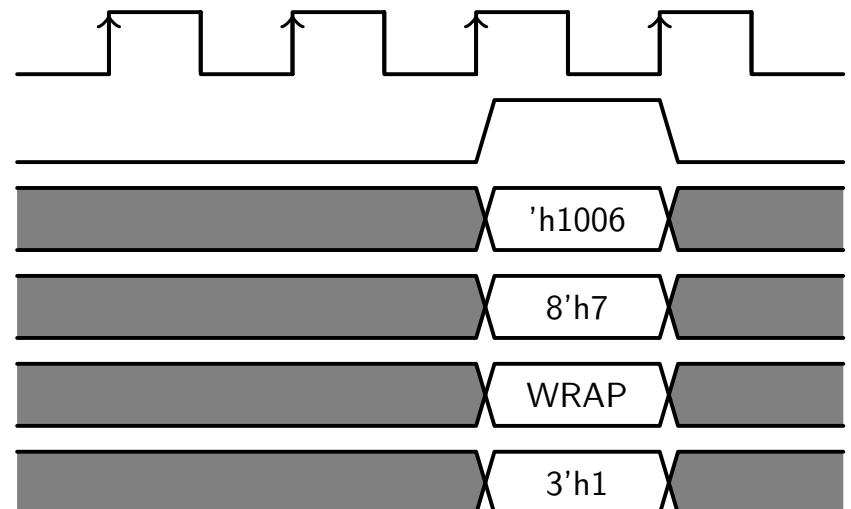
S\_AXI\_ARVALID

S\_AXI\_ARADDR

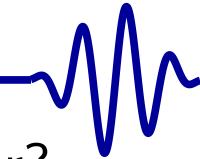
S\_AXI\_ARLEN

S\_AXI\_ARBURST

S\_AXI\_ARSIZE



Assume a 32'bit bus width

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

What address and in what order is this request asking for?

S\_AXI\_ACLK

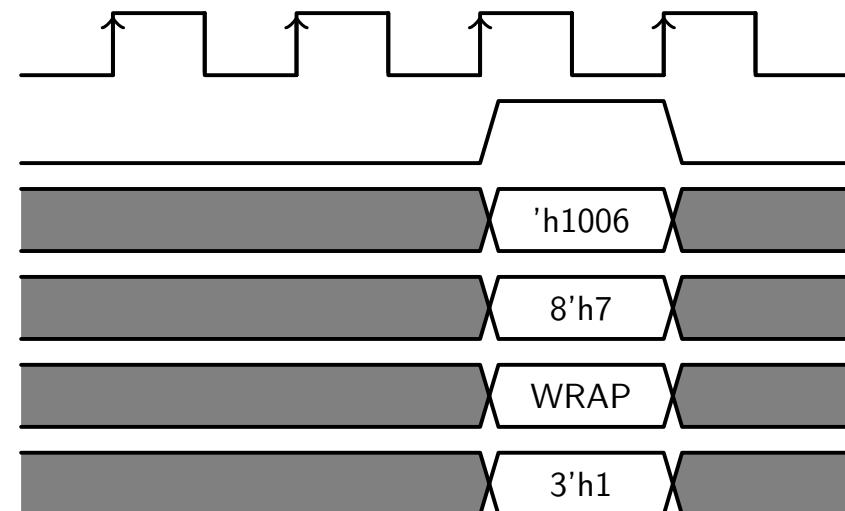
S\_AXI\_ARVALID

S\_AXI\_ARADDR

S\_AXI\_ARLEN

S\_AXI\_ARBURST

S\_AXI\_ARSIZE



The addresses read and returned will be 1006h, 1008h, 100Ah, 100Ch, 100Eh, 1000h, 1002h, 1004h in that order

# GT Quiz #46

---



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

You've just built a new peripheral. You'd like to formally verify it.  
What properties would you start with?

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

This is a very open ended question, so there are many answers to this question.

Here are some of my own:

1. Start with any bus interface formal property files  
This will immediately include a set of assumptions and assertions, which will then validate your bus interface
2. Consider assuming an initial reset
3. **cover()** the end of every type of bus request you expect to respond to  
Don't forget to **cover()** the design returning back to idle!
4. Create sequences (SVA or poor man's) describing the actions associated with each operation you expect to perform, and ending with the bus response  
Don't forget the return to idle!

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

There are three basic methods to include formal properties into a design

1. Placing the formal properties within the design itself

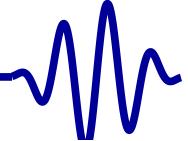
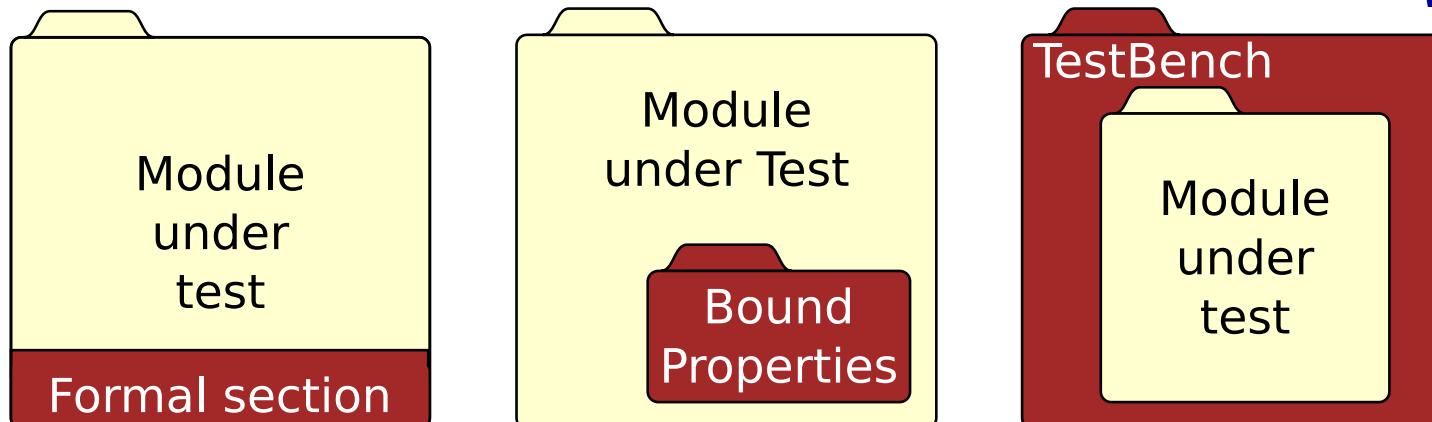
```
module modulename(* ... *);  
    // Design logic  
    #ifdef FORMAL  
        // Properties  
    #endif // FORMAL  
endmodule
```

This works nicely with the open version of SymbiYosys.

2. Binding the properties from one file into the logic of another

```
bind designmodule propertymodule instance (*.);
```

Can anyone think of a third method?

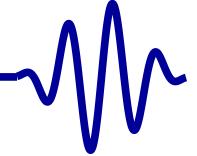
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

A third method of adding properties into a design is to wrap the design with the properties like you would with a test bench.

- Without access to internal state values, passing induction can be a challenge  
Remember, induction is a form of *white-box* verification
- State registers within the design may still be referenced using dot notation  
Dot notation support is currently only available when using commercial formal tools, such as the SymbioticEDA Suite



# Quiz #48



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

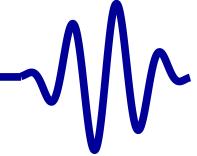
Cover

Sequences

Quizzes

You are trying to verify a CPU.

- How would you go about verifying that your *instruction fetch* works?
- What formal properties would be appropriate to describe the “contract” between the instruction fetch and the CPU?

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

1. Include a formal bus property file, to verify the bus interaction
2. Pick an address in memory, pick a piece of data at that address, decide if the address will return a bus error or not

```
(* anyconst *) reg [AW-1:0] f_fetch_addr;  
(* anyconst *) reg [DW-1:0] f_fetch_data;  
(* anyconst *) reg f_fetch_err;
```

3. **assume()** on the bus interface ...
  - That any request for f\_fetch\_addr returns f\_fetch\_data
  - That it also returns a bus error if and only if f\_fetch\_err
4. **assert()** within your CPU, that any time the instruction address matches f\_fetch\_addr
  - That the instruction matches f\_fetch\_data
  - That an error condition exists if f\_fetch\_err is ever true



# Quiz #49



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

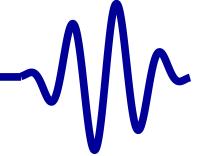
Sequences

Quizzes

The following design is used to read from either a control register, or sequential elements from a block RAM.

```
always @(posedge i_clk) begin
    if (i_wb_stb && i_wb_we
        i_wb_addr == CONTROL)
        addr <= 0;
    else if (i_wb_stb && !i_wb_we
        && i_wb_addr == DATA)
        addr <= addr + 1;
    memv <= mem[addr];
    case(i_wb_addr)
        CONTROL: o_wb_data <= control_reg;
        DATA:     o_wb_data <= memv;
    endcase
    o_wb_ack <= i_wb_stb; // ...
```

See 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 time it takes to read a value?

- Reads take two clocks: one to read the value from memory, and a second to select the value read.
- By setting `o_wb_ack` immediately after `o_wb_stb`, the memory value doesn't make it into `o_wb_data` in time.
- Delaying `o_wb_ack` by one clock would fix this.

This bug was **living in one of my cores** for years.

- Reading all ones or all zeros values never caught it
- Neither did slower serial port commanded reads.
- I only caught this bug recently when reading from a DMA returned elements 0, 0, 1, 2, 3, etc.

What formal properties would you recommend adding to this design in order to catch these bugs?

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

Chances are the process of formal verification would catch this

- Just putting the property together is likely to force you to think through what you want your logic to do
- ... and catch the bug

Once thought out, the following property would double-check the two clock read.

```
assert property (@(posedge i_clk)
    disable iff (i_reset || !i_wb_cyc)
    (i_wb_stb && !o_wb_stall
     && !i_wb_we && i_wb_addr == DATA )
    |=> (addr == $past(addr + 1))
    ##1 o_wb_ack
    && (o_wb_data == $past(mem[addr], 2)));
```

Watch out for overflow in that addition!



# Quiz #50



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

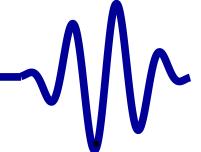
Sequences

Quizzes

The following construct works well to make certain that initial values and reset values match

```
reg      f_past_valid = 0;  
always @ (posedge i_clk)  
    f_past_valid <= 1;  
  
always @ (posedge i_clk)  
if (!f_past_valid || $past(i_reset))  
begin  
    // Check for reset properties  
    // For example ...  
    assert(counter == 0);  
end
```

How would you go about verifying the reset works on a design with no initial values or for hardware that doesn't support them?

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

The key to not having any initial value support lies in assuming an initial reset

```
initial assume(i_reset);  
  
always @ (posedge i_clk)  
if (!i_reset && $past(i_reset))  
begin  
    // Check reset properties  
    // For example ...  
    assert(counter == 0);  
end
```

Bonus: How would you verify a design with an asynchronous reset?



# Quiz #51



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

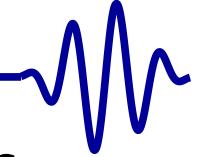
Cover

Sequences

Quizzes

Your design contains a FIFO. You want to assert a property of its output. How do you go about it?

```
sfifo fifo(i_clk, i_reset, i_wr, i_wval,  
          i_rd, i_rval);  
  
always @(*)  
    assert (something_about_i_rval);
```

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

FIFO's are typically verified by following one or two items through the FIFO process. These special values can be used to prove the assertion below.

```
sfifo fifo(i_clk, i_reset, i_wr, i_wval,  
           i_rd, i_rval);  
  
always @(*)  
  if (rval_is_special_value)  
    assert(something_about_i_rval);  
  else // if (!rval_is_special_value)  
    assume(something_about_i_rval);  
always @(*)  
  if (special_value_in_fifo)  
  begin  
    // Assert something about the special  
    // value while it is in the FIFO
```

# GT Quiz #52



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

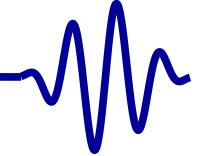
Sequences

Quizzes

You are trying to formally verify a CPU. How would you go about verifying that your load/store unit works?



# Answer #52



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

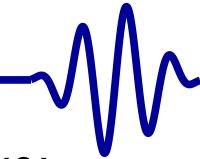
Multiple-Clocks

Cover

Sequences

Quizzes

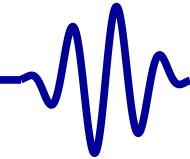
1. Start by including the formal bus property file
2. As with the instruction fetch, let the solver pick a ...
  - Special address, `f_lsu_addr`,
  - Special data value, `f_lsu_data`, and
  - Whether the bus should return an error, `f_lsu_err`.
3. Track writes to `f_lsu_addr` using the data values
  - Any time a store instruction is issued for `f_lsu_addr`, adjust the value of `f_lsu_data`
  - Any time a write is issued over the bus for `f_lsu_addr`, **assert()** the value written is `f_lsu_data`
4. **assume()** reads from the address return `f_lsu_data`, and return errors if and only if `f_lsu_err`
5. **assert()** within your CPU, that any time `f_lsu_addr` is read, `f_lsu_data` is written to the register file

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

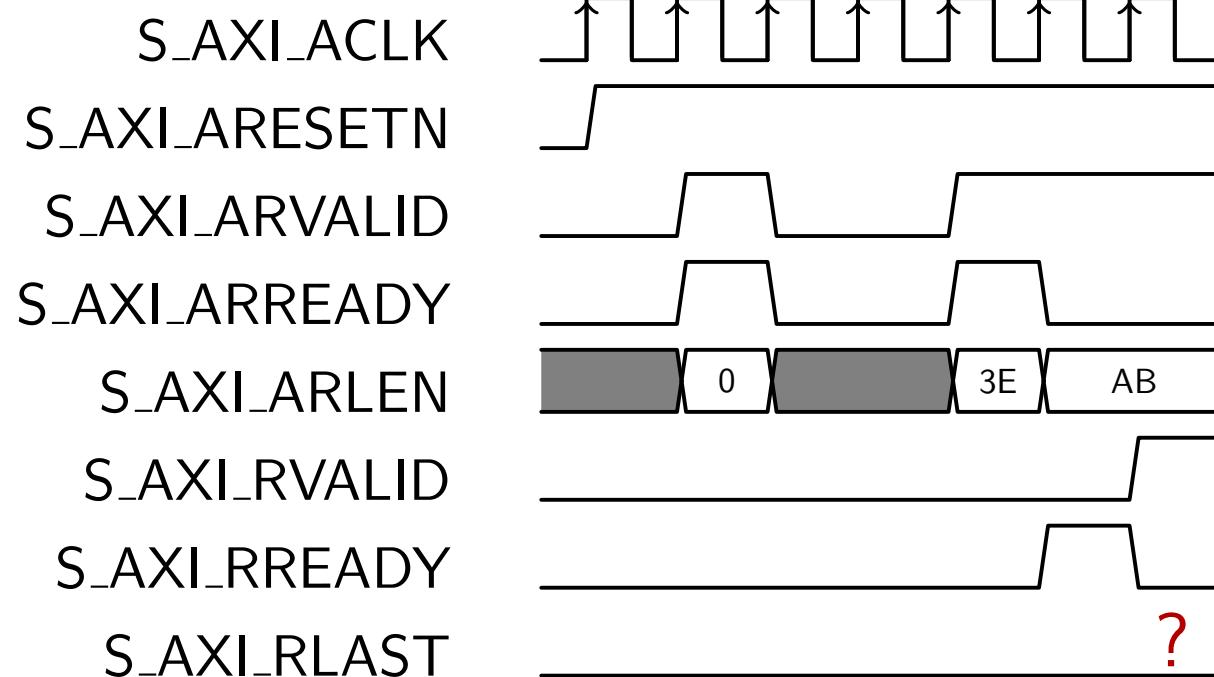
Consider the VHDL design below controlling an AXI slave:

```
AXI_READ_RLAST_P : process (S_AXI_ACLK) is
begin
    if (S_AXI_ACLK'event and S_AXI_ACLK='1') then
        if (S_AXI_ARESETN = '0') then
            S_AXI_RLAST <= '0';
        elsif S_AXI_RREADY = '1' then
            S_AXI_RLAST <= s_axi_rlast_i and rvalid;
        end if;
    end if;
end process AXI_READ_RLAST_P;
```

Can you spot any bugs in this snippet alone?

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

SymbiYosys found the following trace,



This bug lived for years in a piece of commercial IP that was regularly checked by a “best in class” property checker. A first ever formal AXI property check turned it up immediately.

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

The correct check would include not only S\_AXI\_RREADY, but also the possibility that !S\_AXI\_RVALID.

```
AXI_READ_RLAST_P : process (S_AXI_ACLK) is
begin
    if (S_AXI_ACLK'event and S_AXI_ACLK='1') then
        if (S_AXI_ARESETN = '0') then
            S_AXI_RLAST <= '0';
        elsif (S_AXI_RVALID = '0' -- extra check!
               or S_AXI_RREADY = '1') then
            S_AXI_RLAST <= s_axi_rlast_i and rvalid;
        end if;
    end if;
end process AXI_READ_RLAST_P;
```



# Quiz #54



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

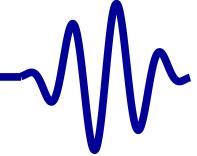
Multiple-Clocks

Cover

Sequences

Quizzes

You are trying to verify a CPU. How can you go about verifying that a single ALU instruction works? Let's consider an ADD instruction for this example.

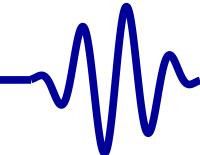
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Quizzes](#)

How shall you verify an ADD instruction within a CPU?

1. Generate a packet as the ADD instruction gets processed
  - Capture the instruction word, current/next program counter, register inputs, ALU output, etc.
2. **cover()** an ADD instruction getting retired
3. When the instruction is retired, use assertions to check . . .
  - Is the output equal to the register inputs summed together?
  - Pick a register. If the input to the instruction is that register, does it match the value of the last time the register was written?
  - Is the current program counter equal to the next program counter from the previous instruction?
  - Is the next program counter the next location in memory?



# Quiz #55



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

You are working on a bus component, and you want to know how much throughput you can achieve per clock using that component

How might you use formal tools to solve this problem?

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

**cover()** makes a great way of measuring best case throughput. The following formal logic will generate a trace demonstrating the maximum AXI write throughput within a design

```
reg [3:0] cvr_writes;
initial cvr_writes = 0;
always @(*(posedge i_clk))
if (!S_AXI_ARESETN)
    cvr_writes <= 0;
else if (S_AXI_BVALID && S_AXI_BREADY)
    cvr_writes <= cvr_writes + 1;

always @(*)
    cover(cvr_writes > 4);
```

This logic will generate *the earliest possible* trace showing a response to five separate write requests (each w/ AWLEN=0)



# Quiz #56



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

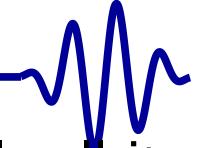
Cover

Sequences

Quizzes

You are working on an AXI bus slave, and you want to know how much throughput you can achieve per clock. Moreover, your core is able to handle multiple burst sizes.

How might you determine how fast your core can handle burst writes?

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

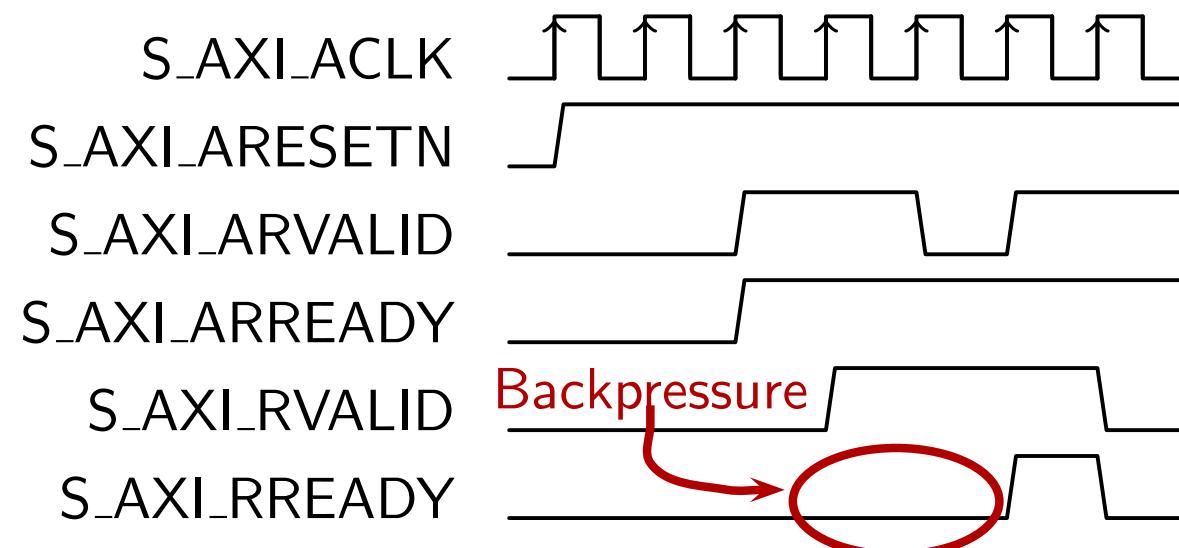
You can use **cover()** again! This time, create a flag, we'll call it `cvr_wr_bursts`, that will only be true if all write requests are of length four or greater.

```
reg                      cvr_wr_bursts = 1;  
always @(*(posedge i_clk))  
if (!S_AXI_ARESETN)  
    cvr_wr_bursts <= 1;  
else if (S_AXI_AWVALID && S_AXI_AWLEN < 3)  
    cvr_wr_bursts <= 0;  
  
// cvr_writes counts BVALID & BREADY as before  
always @(*)  
    cover(cvr_wr_bursts && cvr_writes > 2);
```

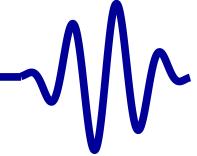
The above example will generate a trace showing a response to three separate write bursts, each with AWLEN=3.

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

Many of the AXI bugs I've found have centered around the inability of a slave design to handle backpressure.



What simulation or **cover()** goals might you use to guarantee your design doesn't suffer from an inability to handle backpressure?

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

A useful simulation or **cover()** goal might be to hold S\_AXI\_ARVALID high while holding S\_AXI\_RREADY low, creating a maximum forward and backpressure. You could then examine the trace to see if it looks right.

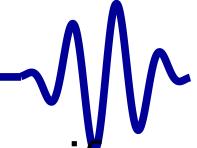
- This still requires examining the trace to know if the core handled the backpressure correctly
- A formal property checker, given a bus property file, would automatically check this setup by nature
- Such a checker would also examine the signals for you, to find exactly where a request wasn't properly given a response.

Of course, this is *only one* of the many possible simulation goals

- With simulation, you'll never know if you've done enough



# Quiz #58



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

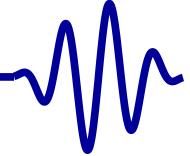
Cover

Sequences

Quizzes

You've built a complex state machine, and now want to verify that without a start signal the state machine will remain idle. Worse, you want to verify several other consequences of remaining idle as well.

How might you go about building such a proof using Yosys?

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

Here's an approach that I've used on several projects

- First, let the solver pick whether to do this check or not

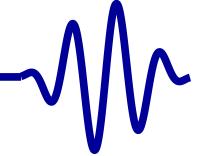
```
( * anyconst *) reg f_idle_check;
```

- Then, if set, assume no start signal

```
always @(*)
  if (f_idle_check)
    begin
      assume (!i_start_signal);
```

- Finally, assert your special case conditions

```
assert(state == IDLE);
assert(consequence_one);
// ... etc.
end
```

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

You are trying to verify a hardware DMA

- A DMA is essentially a hardware memory copy
  1. It receives a source address, destination address, and copy length from the bus
  2. Then copies (length) bytes of memory from source to destination address
- Ignoring the obvious undefined behavior associated with overlap between source and destination ...

What formal properties would be appropriate to describe the “contract” that such a DMA is required to fulfill?

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

What formal properties would be appropriate to describe the “contract” that a DMA is required to fulfill?

- The first step is easy: connect your bus properties to both control port and the data port.

That might just find most of your bugs, but for completeness you'll want to do one more:

- Pick a value in memory, at some offset within the source region
- **assume** this value is returned by a read of that address
- **assert** this value is written by a write to the same offset, but within the destination region
- If the solver can pick the value and offset arbitrarily, and the resulting proof passes, then the entire DMA will therefore work.



# Quiz #60



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

You are trying to verify a CPU. How can you go about verifying that a multiplication instruction works?

```
always @(* posedge i_clk)
    mpy_out <= i_a * i_b;

always @(* posedge i_clk)
    case(insn_type)
        ALU_INSN: result <= alu_out;
        MPY_INSN: result <= mpy_out;
        DIV_INSN: result <= alu_out;
        LOD_INSN: result <= lsu_out; // Load/Store Insn
    endcase
```

```
always @(*) // What assertion(s) might you use?
if (insn_type == MPY_INSN)
    assert(mpy_out == ?);
```

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

This issue is complicated by the fact that formally verifying the result of a multiplication tends to be beyond the capability of the state of the art of formal verification. Given that, here are some things you can do:

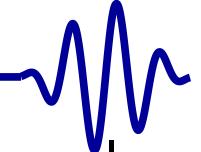
- Replace the output of the multiply with a (constrained) arbitrary value
  - Possible constraints include assuming the correct value in the case of multiplication by zero, one, or negative one
  - Alternatively, you might XOR'ing the inputs together with another value

Although these solutions don't check the result of the instruction, they can still catch bugs associated with the pipeline timing, forwarding, etc.

- The actual multiply result can then be checked via simulation



# Quiz #61



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

Just as formal tools struggle with multiplies, they also struggle with divides. Worse, many divide instructions take many clocks to complete

- How can you go about verifying a divide using either BMC or cover, but without processing all 32 (or more) steps of the divide?

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

Verifying that the divide pipeline works is still valuable

- Consider using the approaches we used for a multiply to verify that the divide is properly handled by its context
- You can capture the duration of the divide using a (\* `anyseq` \*) “free variable.” Let this value range from only a couple of clocks in duration all the way to the correct length of the divide. This will keep things within the range of both BMC and **cover()**

Verifying that the pipeline works for all durations of the divide effectively verifies that it works for the correct duration

- You can use simulation to actually verify the *result* of the divide
- Alternatively, you can use formal to verify the *individual internal steps* of the divide

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

You have a counter that is supposed to count down from some programmable value to zero. How can you assert that this counter will never be higher than the programmable value, given that the value might change mid count?

```
always @(posedge i_clk)
begin
    if (set_value) max_value <= new_value;

    if (counter == 0)
        counter <= max_value;
    else
        counter <= counter - 1;

    // This fails if the max_value ever
    // changes mid countdown!
    assert(counter <= max_value);

end
```

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

Q: How can you assert that a counter will never be higher than the programmable value, given that the value might change mid count?

Answer: Capture a copy of the maximum value at the time the counter is set

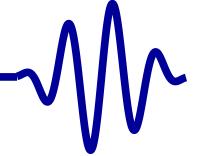
```
always @(posedge i_clk)
  if (counter == 0)
    f_max_value <= max_value;

always @(*)
  assert(counter <= f_max_value);
```

Remember: you can use Verilog to your advantage!



# Quiz #63



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Quizzes

You have a CPU component of a larger design.

```
cpu mycpu(i_clk, i_reset,  
          bus_master_outputs, // ...  
          bus_master_inputs, // ...  
          interrupt_line); // or lines
```

Your CPU passes formal verification.

How would you go about formally verifying the rest of the design?



# Answer #63



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

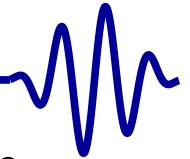
Quizzes

How would you go about formally verifying the rest of the design?

Replace the CPU with a set of bus interface properties!

- Assume the CPU is a generic bus master
- This will disconnect any bus transactions from the CPU operation that would cause them  
On the other hand, you just proved the CPU would properly execute its instructions
- You will want to do the same thing with your bus slaves as well as the interconnect

This will then allow you to verify the top level of your design

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

How would you go about formally verifying the rest of the design?

Replace the bus components with bus interface properties!

