



Gisselquist  
Technology, LLC

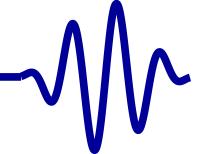
# An Introduction to Formal Methods

Daniel E. Gisselquist, Ph.D.





# Lessons



▷ Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Parting Thoughts

## Day one

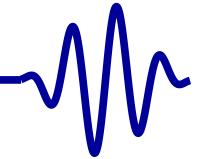
1. Motivation
2. Basic Operators
3. Clocked Operators
4. Induction
5. Bus Properties

## 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

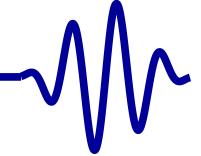
Sequences

Parting Thoughts

- VHDL logic examples



# Course Structure



▷ Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

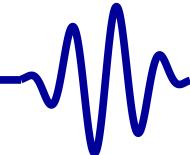
Sequences

Parting Thoughts

- VHDL logic examples
- System Verilog assertion wrappers



# Course Structure



▷ Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

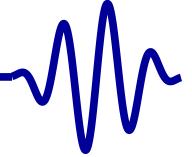
Multiple-Clocks

Cover

Sequences

Parting Thoughts

- VHDL logic examples
- System Verilog assertion wrappers
- 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

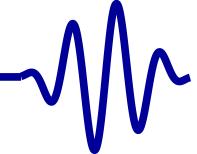
Sequences

Parting Thoughts

# Motivation



# Lesson Overview



Welcome

Motivation

▷ Intro

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Parting Thoughts

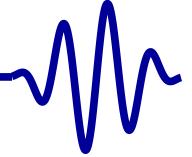
1. Why are you here?
2. What can I provide?
3. What have I learned from formal methods?

## 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

Parting Thoughts

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

Parting Thoughts

- “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

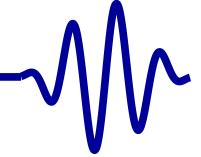
Cover

Sequences

Parting Thoughts

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

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



Welcome

Motivation

▷ Intro

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

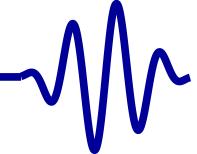
Cover

Sequences

Parting Thoughts

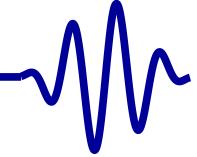
One little mistake . . .

. . . \$475M later.

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

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](#)[Parting Thoughts](#)

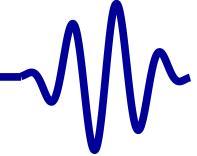
I've found bugs in things I thought were working.

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](#)[Parting Thoughts](#)

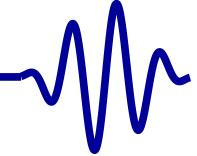
- *It worked in my test bench*
- Failed when reading and writing on the same clock while empty
  - Write first then read worked
  - R+W on full FIFO is okay
  - R+W on an empty FIFO

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

- *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



# Ex: Prefetch



Welcome

Motivation

▷ Intro

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

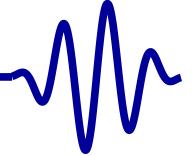
Sequences

Parting Thoughts

- *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.



# Ex: SDRAM



Welcome

Motivation

▷ Intro

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

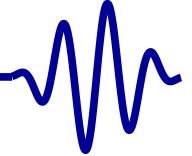
Multiple-Clocks

Cover

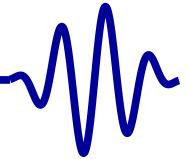
Sequences

Parting Thoughts

- *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](#)[Parting Thoughts](#)

- *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

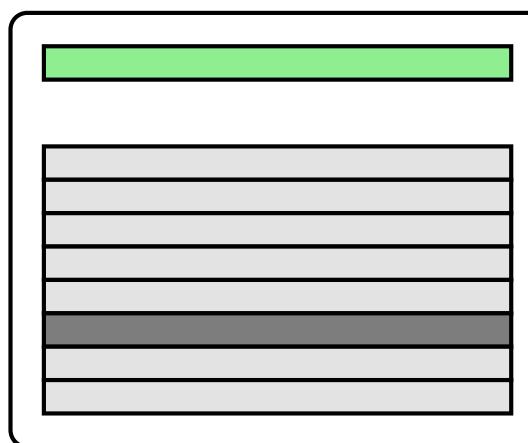
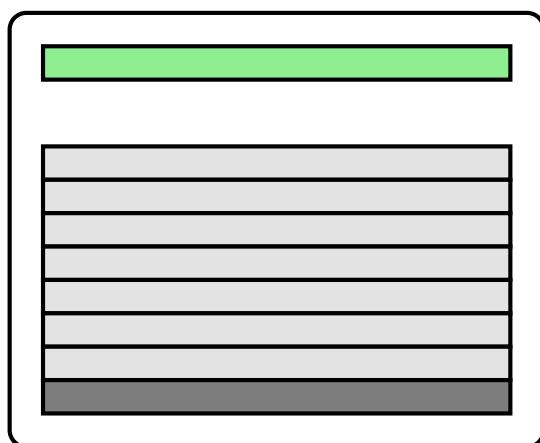
Multiple-Clocks

Cover

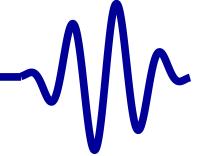
Sequences

Parting Thoughts

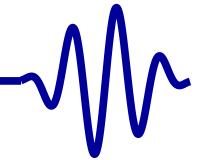
- *It worked in my test bench*
- It passed my hardware testing
- Background



• • •

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

- *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

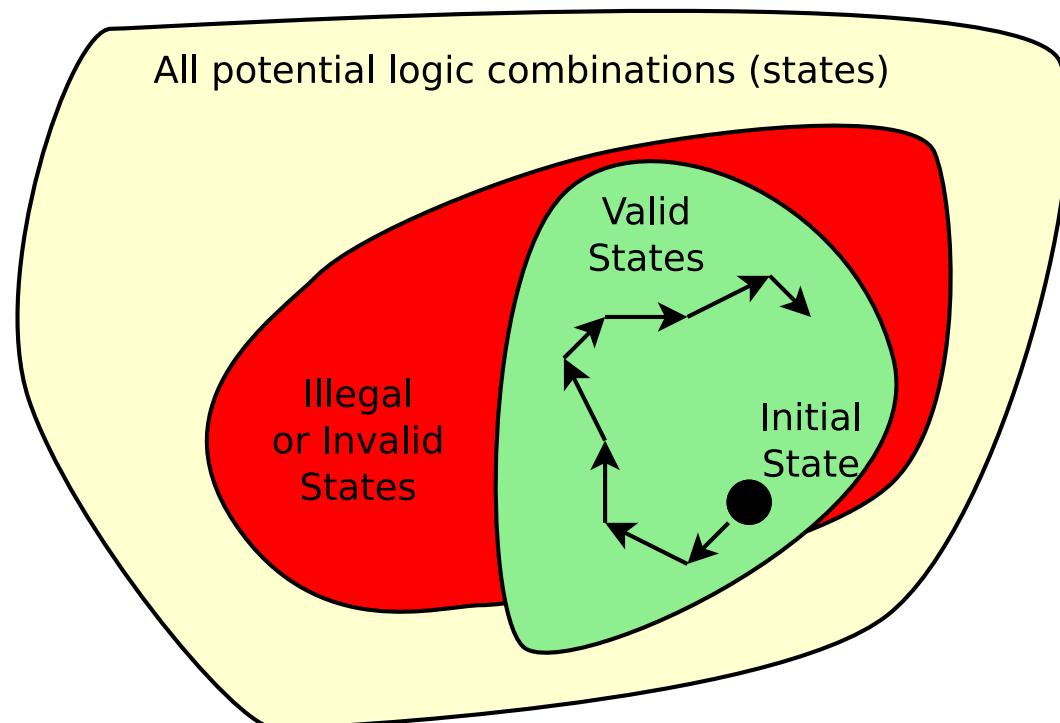
Invariants

Multiple-Clocks

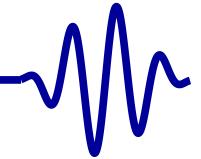
Cover

Sequences

Parting Thoughts



- 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](#)[Parting Thoughts](#)

- 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,

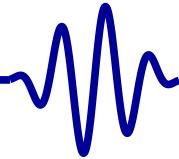
- I only read when I knew it wasn't empty

For the Prefetch,

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

For the SDRAM,

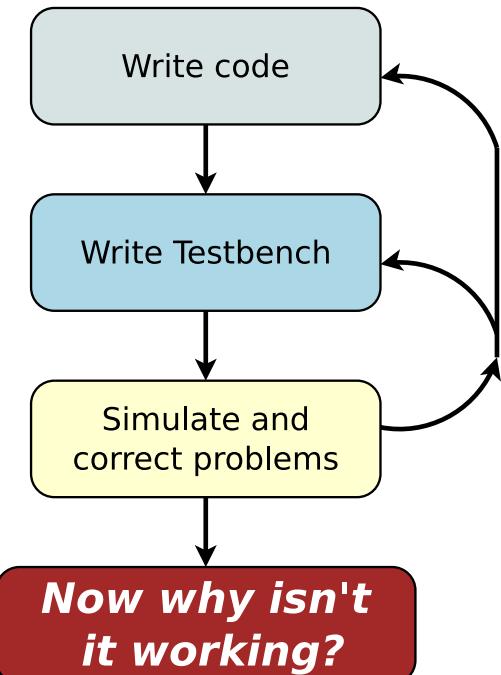
- The error was so obscure, it would be hard to trigger

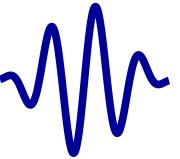
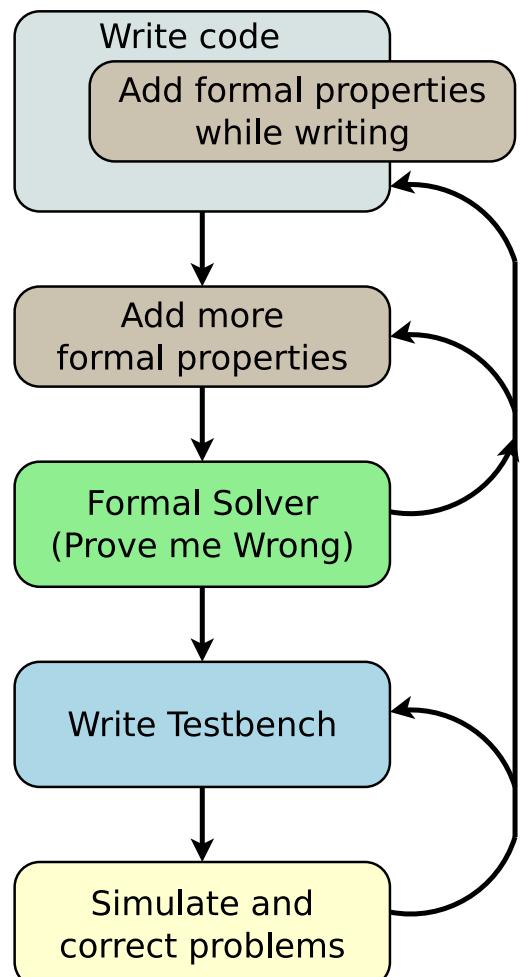
[Welcome](#)[Motivation](#)[Intro](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

This was my method before starting to work with formal.

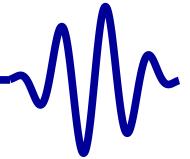
- After . . .
  - Proving my design with test benches
  - Directed simulation
- I was still chasing bugs in hardware

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](#)[Parting Thoughts](#)

- 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

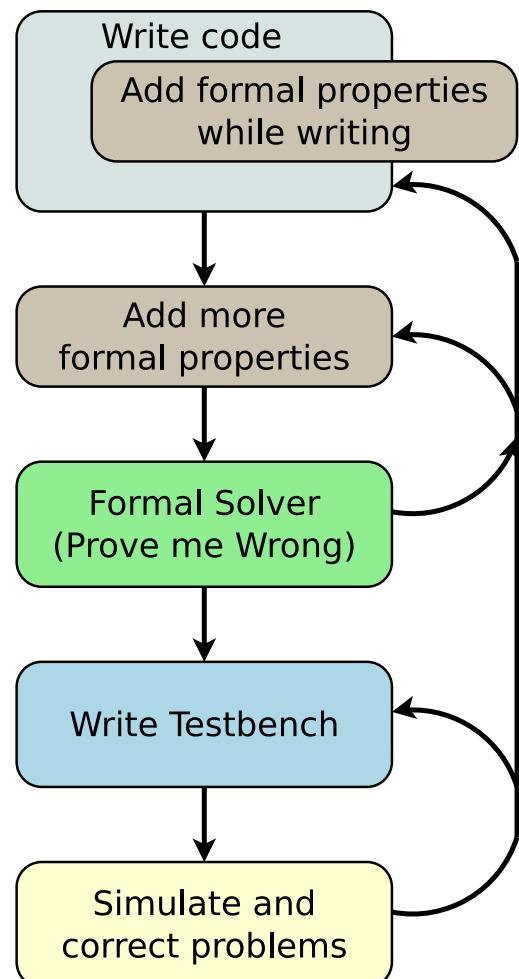
Invariants

Multiple-Clocks

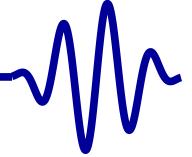
Cover

Sequences

Parting Thoughts



- 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

Parting Thoughts

# 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

Parting Thoughts

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

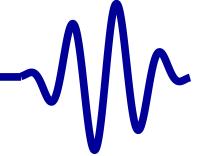
Our Objective:

- 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](#)[Parting Thoughts](#)

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](#)[Parting Thoughts](#)

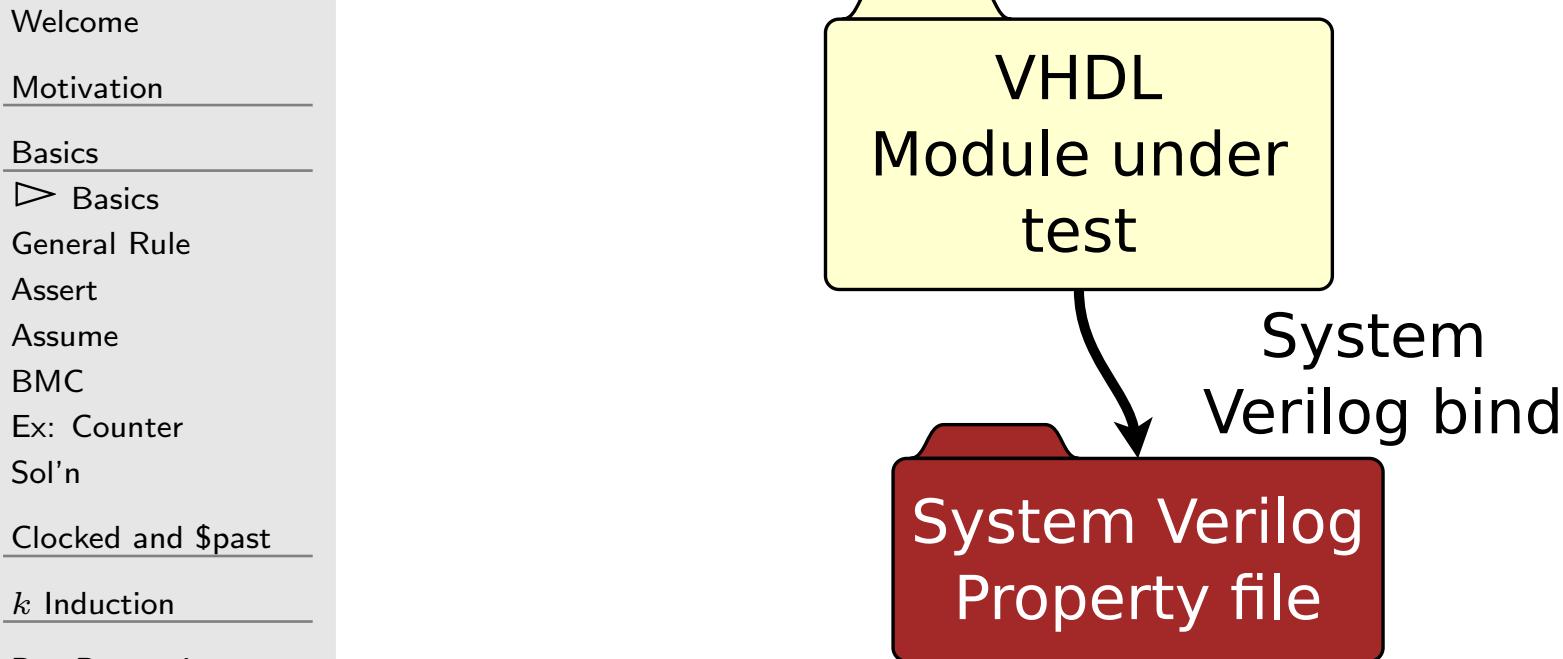
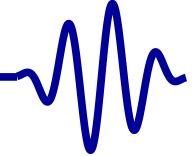
Let's start with the two basic operators

1. **assume()**

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

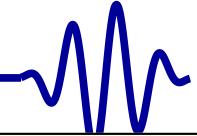
2. **assert()**

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



We'll be using VHDL logic, System Verilog Assertions

- Proprietary Verific library gives Yosys access to VHDL
- Formal properties will be written in System Verilog
- System Verilog **bind** operator will connect the two



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

Parting Thoughts

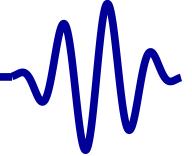
```
always @(*)  
    assert(x);
```

// Use when your property has clock dependencies,  
// such as referencing an item's 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

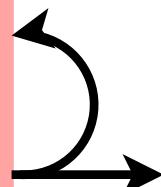
Sequences

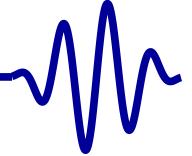
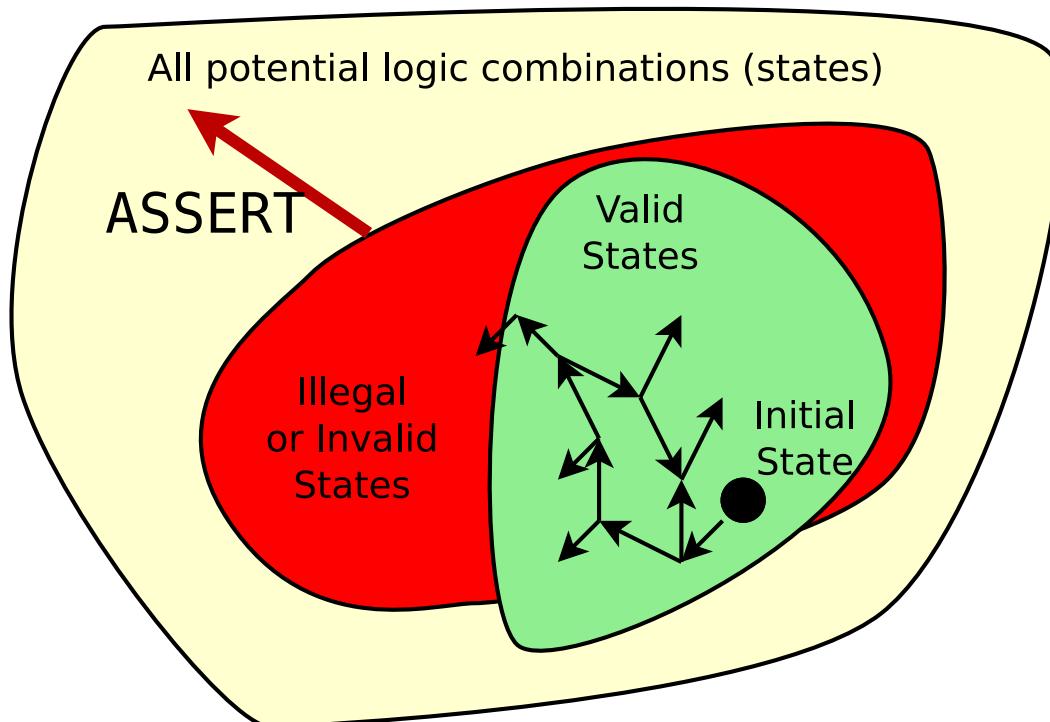
Parting Thoughts

## Master FV Rule

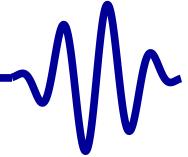
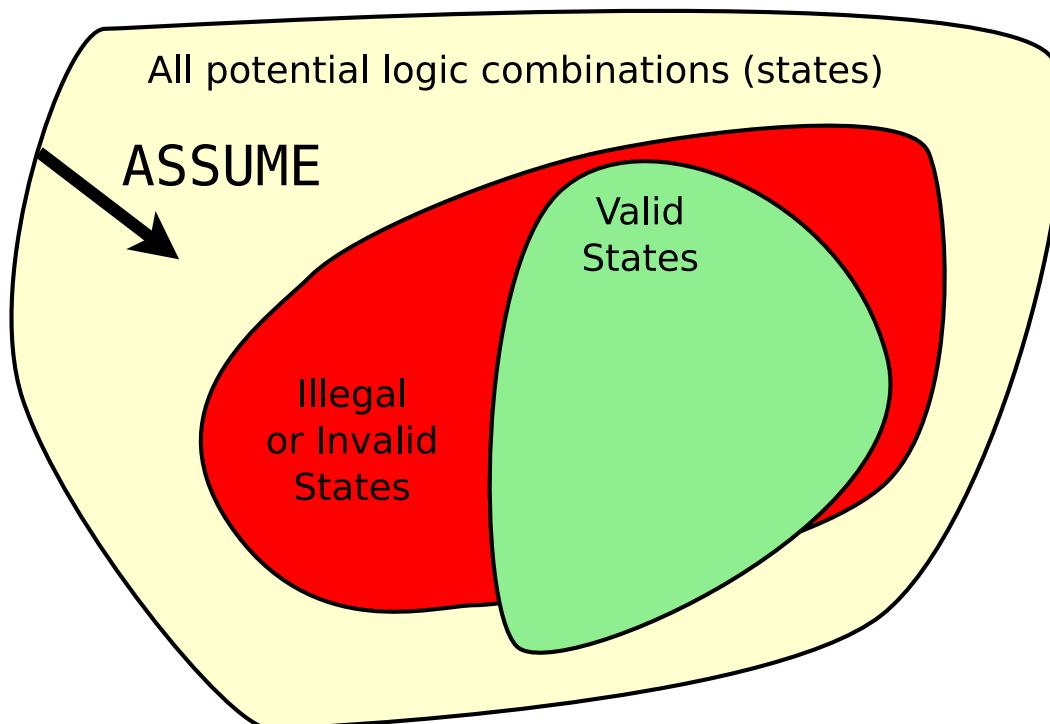
→ assume(inputs);

assert(local state);  
assert(outputs);

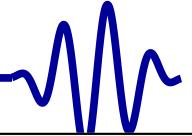


[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[▷ Assert](#)[Assume](#)[BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

- 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](#)[Parting Thoughts](#)

- 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

Parting Thoughts

```
signal : unsigned(15 downto 0) := 0;  
  
process(clk)  
begin  
    if (rising_edge(clk)) then  
        counter <= counter + 1;  
    end if;  
end process;
```

```
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](#)[Parting Thoughts](#)

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

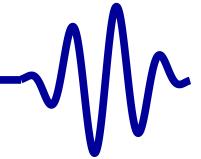
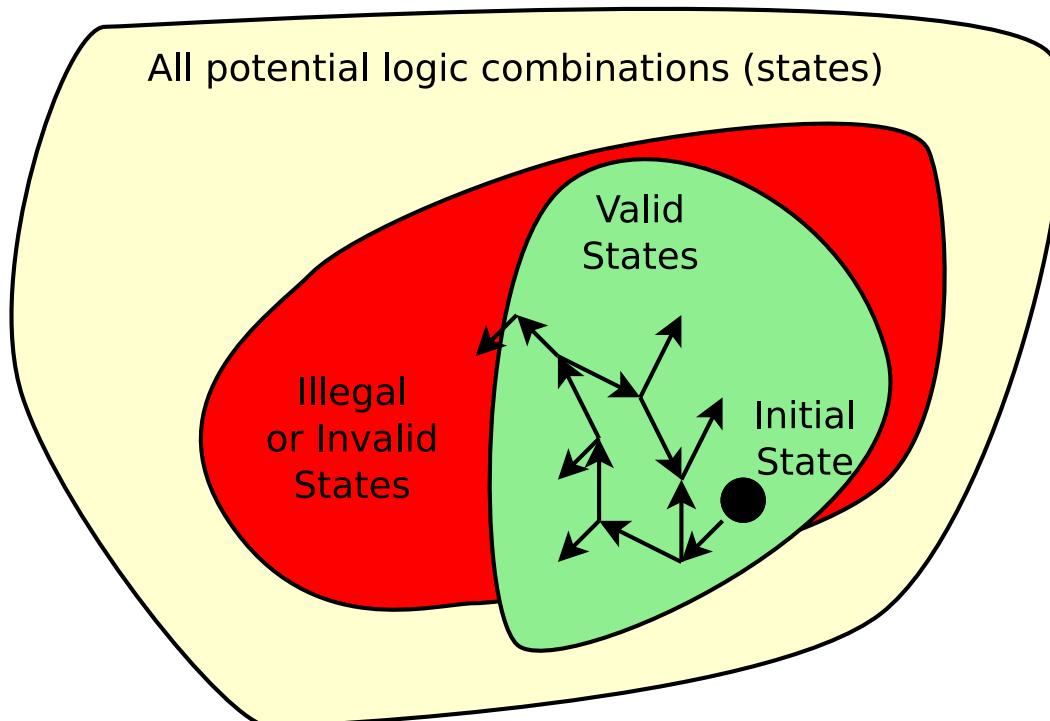
Operator	Formal Verification	Traditional Simulation
<b>restrict ()</b>	Restricts search space	Ignored
<b>assume()</b>		Halts simulation with an error
<b>assert()</b>	Illegal state	

[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](#)[Parting Thoughts](#)

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

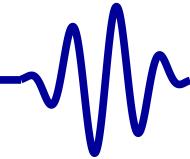
Operator	Formal Verification	Traditional Simulation
<b>restrict ()</b>	Restricts search space	Ignored
<b>assume()</b>		Halts simulation with an error
<b>assert()</b>	Illegal state	

- **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](#)[Parting Thoughts](#)

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

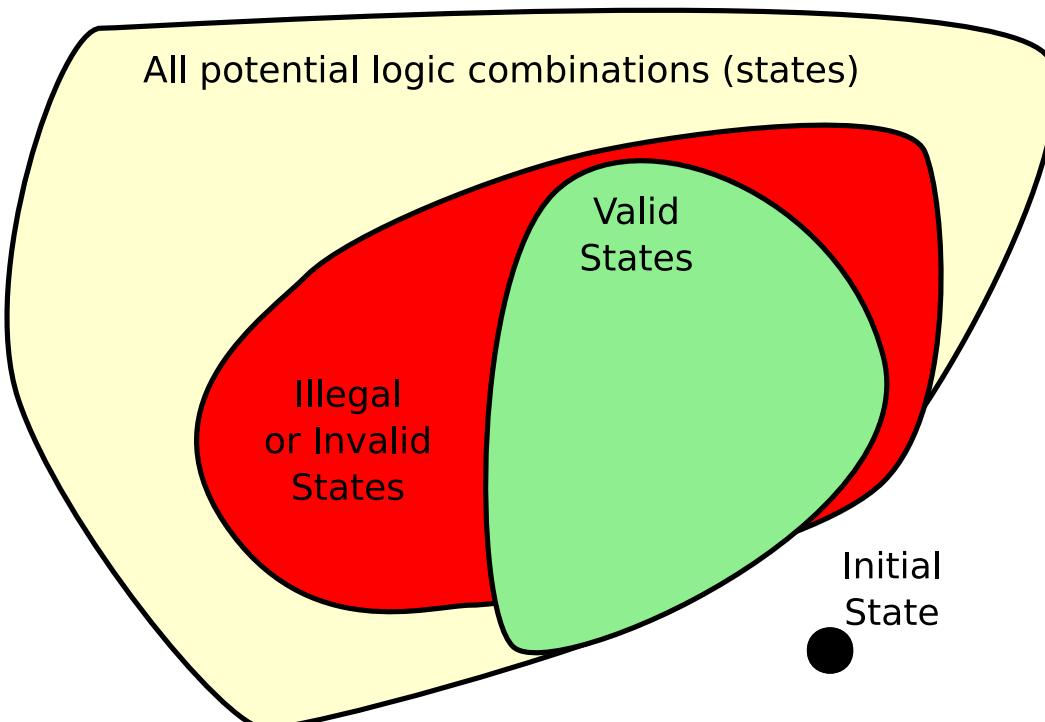
Invariants

Multiple-Clocks

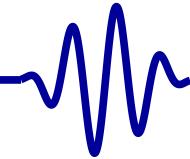
Cover

Sequences

Parting Thoughts



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

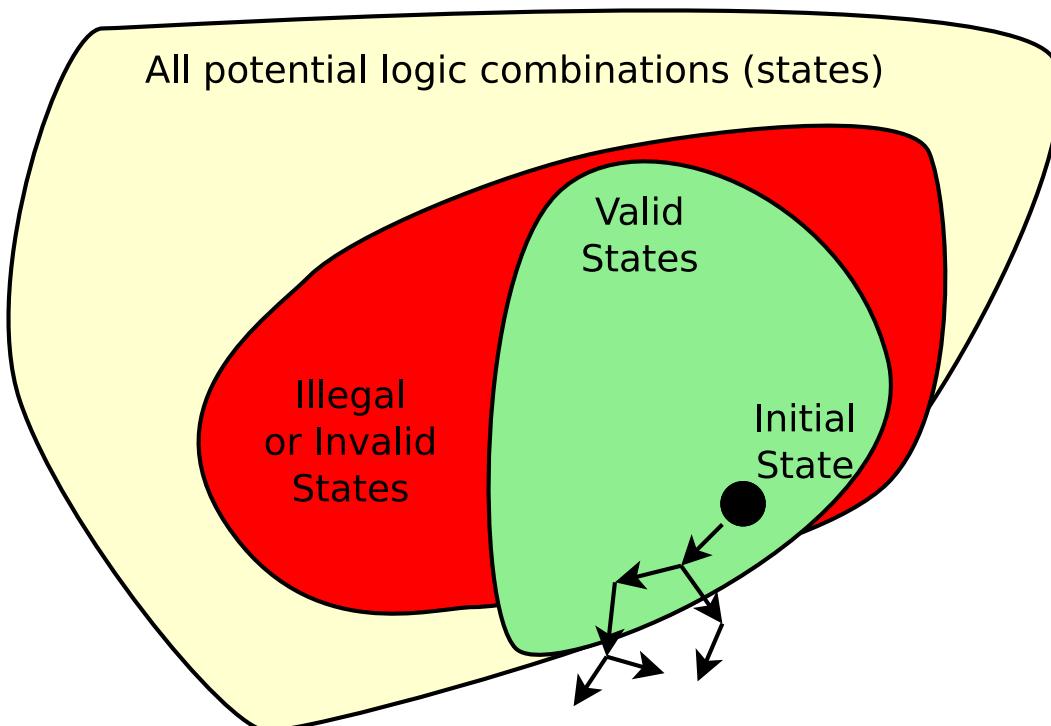
Invariants

Multiple-Clocks

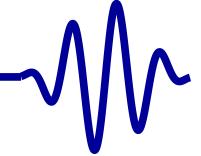
Cover

Sequences

Parting Thoughts



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

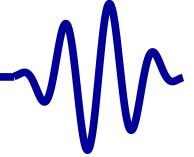
Sequences

Parting Thoughts

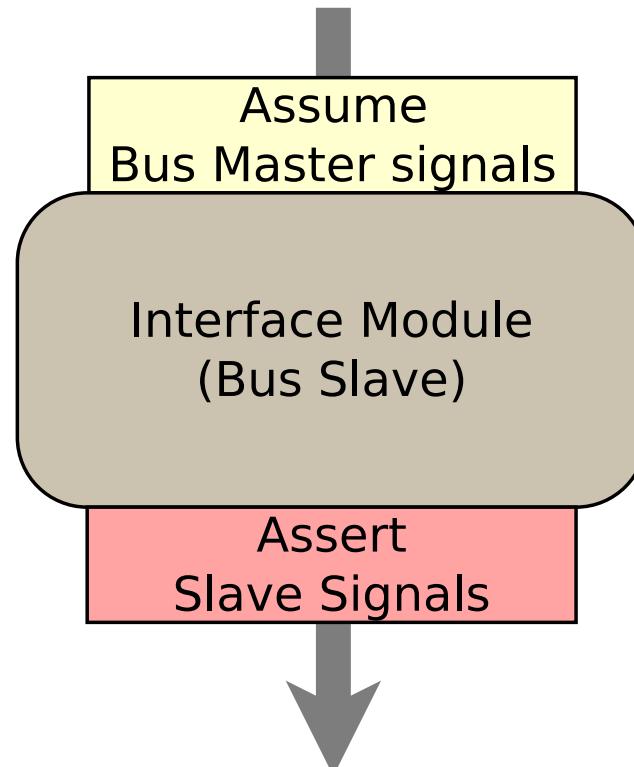
Unlike the rest of your digital design, formal properties . . .

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

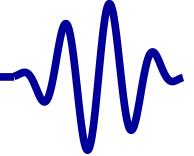
We'll discuss this more as we go along.

[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](#)[Parting Thoughts](#)

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

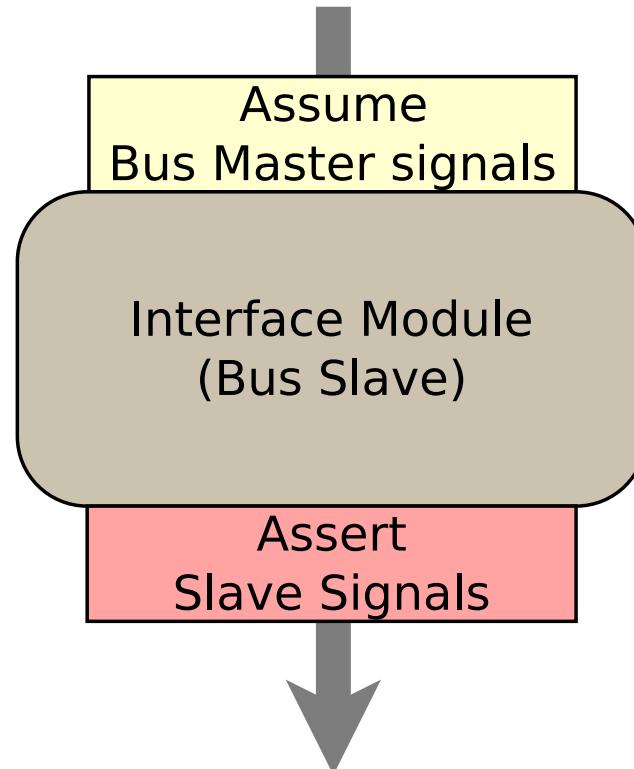
Multiple-Clocks

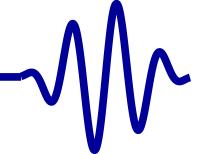
Cover

Sequences

Parting Thoughts

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

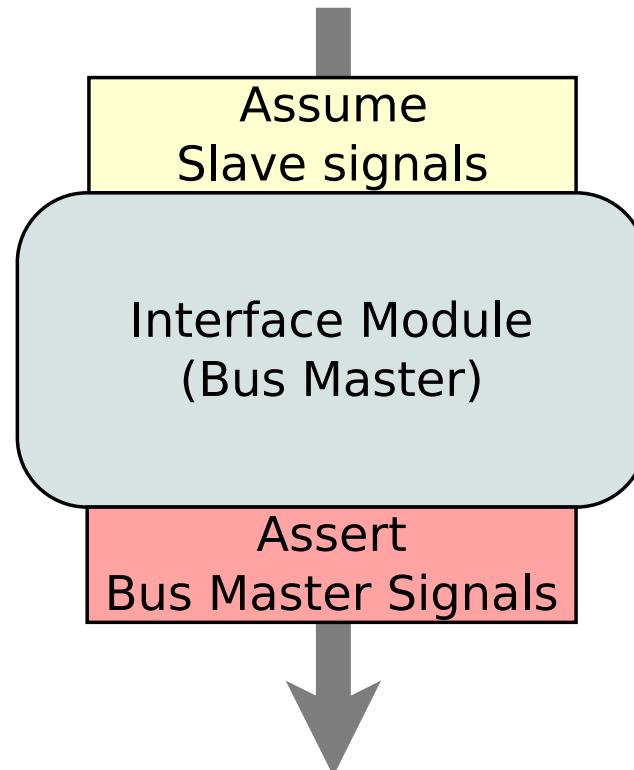
Multiple-Clocks

Cover

Sequences

Parting Thoughts

Question: How would a bus master be different?

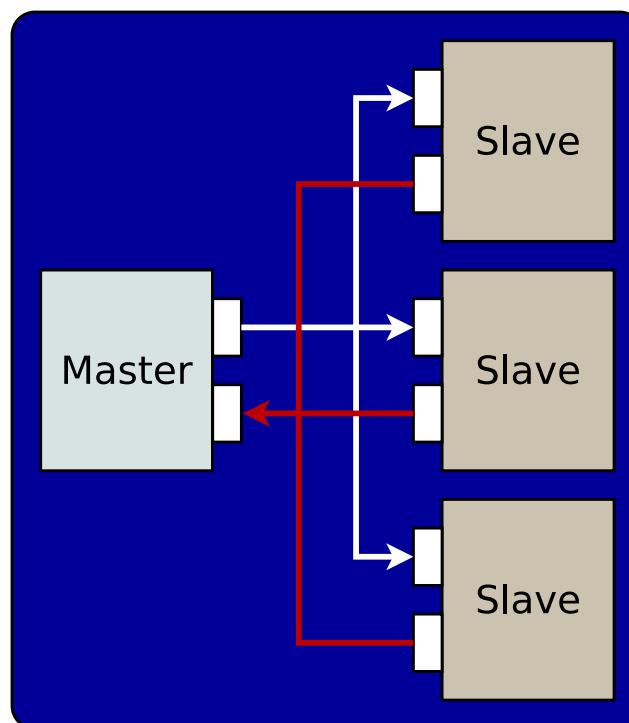


The slave's outputs are the master's inputs

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

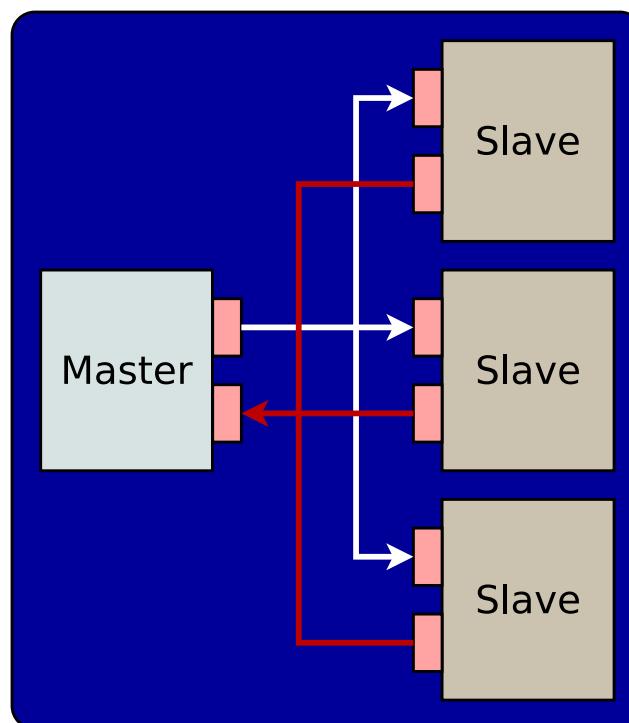
[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[▷ BMC](#)[Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

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](#)[Parting Thoughts](#)

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

Parting Thoughts

- 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](#)[Parting Thoughts](#)

- 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](#)[Parting Thoughts](#)

- 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](#)[Parting Thoughts](#)

- 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](#)[Parting Thoughts](#)

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

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

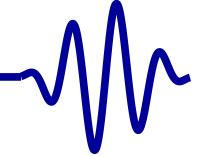
Remember ! (A&&B) is equivalent to (!A )||(! B)

- 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

Parting Thoughts

- 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:

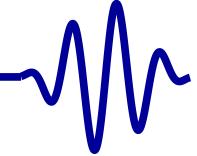
```
f_outstanding <= "0" when (i_reset)  
  else f_nreqs - f_nacks;
```

- 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

Parting Thoughts

- 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
```

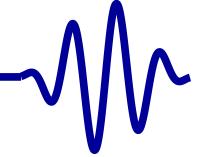
- Following any reset, the bus will be idle
- Requests remain unchanged until accepted

[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](#)[Parting Thoughts](#)

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
```



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

Parting Thoughts

## Consider a multiply

- Just because an algorithm doesn't meet timing



Welcome

Motivation

Basics

Basics

General Rule

Assert

Assume

▷ BMC

Ex: Counter

Sol'n

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Parting Thoughts

## 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](#)[Parting Thoughts](#)

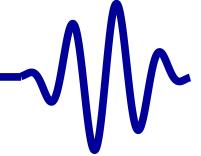
## Consider a multiply

- 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

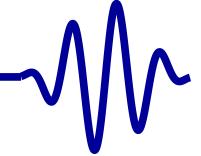
Cover

Sequences

Parting Thoughts

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

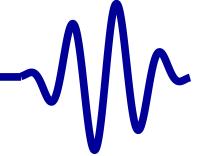
- The one thing formal solver's don't handle well is multiplies

[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](#)[Parting Thoughts](#)

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

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

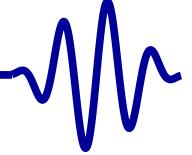
Abstraction offers alternatives

[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](#)[Parting Thoughts](#)

- 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

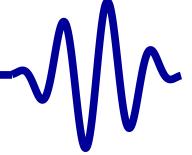
Sequences

Parting Thoughts

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

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

# Ex: Counter



Welcome

Motivation

Basics

Basics

General Rule

Assert

Assume

BMC

▷ Ex: Counter

Sol'n

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

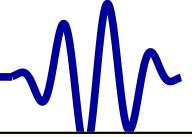
Parting Thoughts

Let's work through a counter as an example.

exercise-01/	Contains three files
counter.vhd	This will be the HDL source for our demo
counter_vhd.sv	This contains the formal properties
counter_vhd.sby	This is the SymbiYosys script for the demo

Our Objectives:

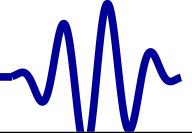
- Walk through the steps in the tool-flow
- Hands on experience with SymbiYosys
- Ensure everyone has a working version of SymbiYosys
- Find and fix a design bug

[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](#)[Parting Thoughts](#)

```
entity counter is
    generic ( MAX_AMOUNT: natural := 22 );
        ...
    signal counts : unsigned(15 downto 0);

    process(i_clk)
    begin
        if (rising_edge(i_clk)) then
            if ((i_start_signal = '1')
                and (0 = counts)) then
                counts <= to_unsigned(MAX_AMOUNT-1, 16);
            else
                counts <= counts - 1;
            end if;
        end if;
    end process;

    o_busy <= '1' when (0 = counts) else '0';
```

[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](#)[Parting Thoughts](#)

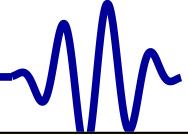
```
module counter_vhd(i_clk, i_start_signal,
                     counts, o_busy);

    parameter      [15:0] MAX_AMOUNT = 22;

    input   wire      i_clk, i_start_signal;
    input   wire [15:0] counts;
    input   wire      o_busy;

    always @(*)
        assert(counts < MAX_AMOUNT);
endmodule

bind counter counter_vhd
#(.MAX_AMOUNT(MAX_AMOUNT)) copy (. *);
```



---

[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](#)

---

[Parting Thoughts](#)

```
// VHDL Ports and internal signals
module counter_vhd(i_clk, i_start_signal,
                     counts, o_busy);

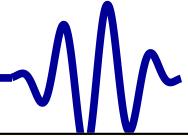
    parameter      [15:0] MAX_AMOUNT = 22;

    input   wire          i_clk, i_start_signal;
    input   wire [15:0] counts;
    input   wire          o_busy;

    always @(*)
        assert(counts < MAX_AMOUNT);
endmodule

bind counter counter_vhd
        #(.MAX_AMOUNT(MAX_AMOUNT)) copy (. *);


```

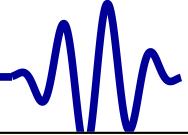
[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

```
module counter_vhd(i_clk, i_start_signal,
                     counts, o_busy);
    // Generic declaration
    parameter      [15:0] MAX_AMOUNT = 22;

    input   wire      i_clk, i_start_signal;
    input   wire [15:0] counts;
    input   wire      o_busy;

    always @(*)
        assert(counts < MAX_AMOUNT);
endmodule

bind counter counter_vhd
#(.MAX_AMOUNT(MAX_AMOUNT)) copy (. *);
```

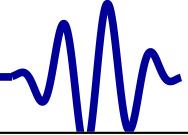
[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

```
module counter_vhd(i_clk, i_start_signal,
                     counts, o_busy);

    parameter      [15:0] MAX_AMOUNT = 22;
    // All wrapper ports are inputs
    input   wire          i_clk, i_start_signal;
    input   wire [15:0]    counts;
    input   wire          o_busy;

    always @(*)
        assert(counts < MAX_AMOUNT);
    endmodule

bind counter counter_vhd
    #( .MAX_AMOUNT(MAX_AMOUNT) ) copy (. * );
```

[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](#)[Parting Thoughts](#)

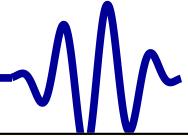
```
module counter_vhd(i_clk, i_start_signal,
                     counts, o_busy);

    parameter      [15:0] MAX_AMOUNT = 22;

    input   wire      i_clk, i_start_signal;
    input   wire [15:0] counts;
    input   wire      o_busy;

    // Formal properties start here
    always @(*)
        assert(counts < MAX_AMOUNT);
endmodule

bind counter counter_vhd
#(.MAX_AMOUNT(MAX_AMOUNT)) copy (. *);
```

[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](#)[Parting Thoughts](#)

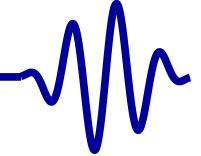
```
module counter_vhd(i_clk, i_start_signal,
                     counts, o_busy);

    parameter      [15:0] MAX_AMOUNT = 22;

    input   wire      i_clk, i_start_signal;
    input   wire [15:0] counts;
    input   wire      o_busy;

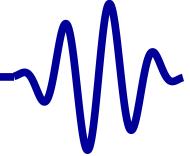
    always @(*)
        assert(counts < MAX_AMOUNT);
endmodule

// Connect the two modules together
bind counter counter_vhd
    #( .MAX_AMOUNT(MAX_AMOUNT) ) copy (. * );
```

[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](#)[Parting Thoughts](#)

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

```
[options]
mode bmc
[engines]
smtbmc
[script]
read -vhdl counter.vhd
read -formal counter_vhd.sv
# ... other files would go here
prep -top counter
[files]
counter.vhd
counter_vhd.sv
```

[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](#)[Parting Thoughts](#)

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

```
[options]
mode bmc ← Bounded model checking mode
[engines]
smtbmc
[script]
read -vhdl counter.vhd
read -formal counter_vhd.sv
# ... other files would go here
prep -top counter
[files]
counter.vhd
counter_vhd.sv
```

[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](#)[Parting Thoughts](#)

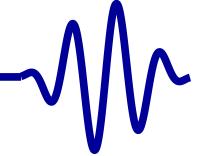
In the file, `exercise-01/counter_vhd.sby`, you'll find:

```
[options]
mode bmc
[engines]
smtbmc ← Run, using yosys-smtbmc
[script]
read -vhdl counter.vhd
read -formal counter_vhd.sv
# ... other files would go here
prep -top counter
[files]
counter.vhd
counter_vhd.sv
```

[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](#)[Parting Thoughts](#)

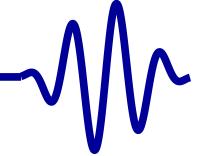
In the file, `exercise-01/counter_vhd.sby`, you'll find:

```
[options]
mode bmc
[engines]
smtbmc
[script] ← Yosys commands
read -vhdl counter.vhd
read -formal counter_vhd.sv
# ... other files would go here
prep -top counter
[files]
counter.vhd
counter_vhd.sv
```

[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](#)[Parting Thoughts](#)

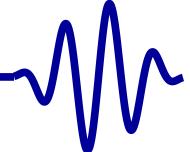
In the file, `exercise-01/counter_vhd.sby`, you'll find:

```
[options]
mode bmc
[engines]
smtbmc
[script]
read -vhdl counter.vhd ← Read VHDL file
read -formal counter_vhd.sv
# ... other files would go here
prep -top counter
[files]
counter.vhd
counter_vhd.sv
```

[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](#)[Parting Thoughts](#)

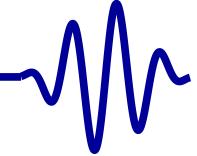
In the file, exercise-01/counter\_vhd.sby, you'll find:

```
[options]
mode bmc
[engines]
smtbmc
[script]
read -vhdl counter.vhd
read -formal counter_vhd.sv ← Read SV file w/SVA
# ... other files would go here
prep -top counter
[files]
counter.vhd
counter_vhd.sv
```

[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](#)[Parting Thoughts](#)

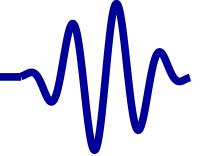
In the file, exercise-01/counter\_vhd.sby, you'll find:

```
[options]
mode bmc
[engines]
smtbmc
[script]
read -vhdl counter.vhd
read -formal counter_vhd.sv
# ... other files would go here
prep -top counter ← Prepare the file for formal
[files]
counter.vhd
counter_vhd.sv
```

[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](#)[Parting Thoughts](#)

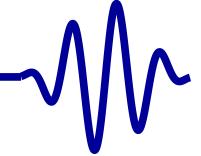
In the file, exercise-01/counter\_vhd.sby, you'll find:

```
[options]
mode bmc
[engines]
smtbmc
[script]
read -vhdl counter.vhd
read -formal counter_vhd.sv
# ... other files would go here
prep -top counter
[files] ← List of files to be used
counter.vhd
counter_vhd.sv
```

[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](#)[Parting Thoughts](#)

## 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](#)[Parting Thoughts](#)

## 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

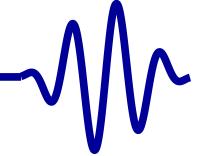
# ... other files would go here

**prep** -top counter

opt\_merge -share\_all

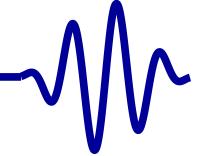
**[ files ]**

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](#)[Parting Thoughts](#)

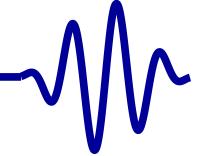
## Other usefull yosys commands

```
[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](#)[Parting Thoughts](#)

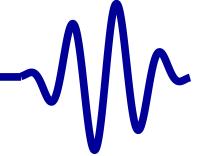
## Other usefull yosys commands

```
[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](#)[Parting Thoughts](#)

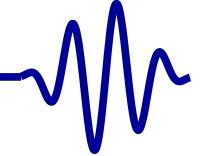
## Other usefull yosys commands

```
[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](#)[Parting Thoughts](#)

## 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
```

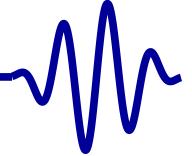
[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)[▷ Ex: Counter](#)[Sol'n](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

## 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

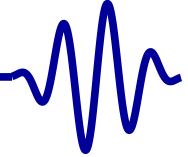
Cover

Sequences

Parting Thoughts

Run: % sby -f counter\_vhd.sby

# GT 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

Parting Thoughts

Run: % sbt -f counter\_vhd.sbt

```
dan@beta:~/vhdl$ sbt -f counter_vhd.sbt
SBY 19:44:22 [counter_vhd] Copy 'counter.vhd' to 'counter_vhd/src/counter.vhd'.
SBY 19:44:22 [counter_vhd] Copy 'counter.vhd.sv' to 'counter_vhd/src/counter_vhd.sv'.
SBY 19:44:22 [counter_vhd] engine_0: smtbmc
SBY 19:44:22 [counter_vhd] base: starting process "cd counter_vhd/src; yosys -ql ../model/design.log ../model/design.yos"
SBY 19:44:22 [counter_vhd] base: -- (c) Copyright 1999 - 2018 Verific Design Automation Inc. All rights reserved
SBY 19:44:22 [counter_vhd] base: -- Compilation time was Wed Mar 7 20:49:25 2018
SBY 19:44:22 [counter_vhd] base: -- This program will expire on Fri Jun 15 20:49:25 2018
SBY 19:44:24 [counter_vhd] base: finished (returncode=0)
SBY 19:44:24 [counter_vhd] smt2: starting process "cd counter_vhd/model; yosys -ql design_smt2.log design_smt2.yos"
SBY 19:44:24 [counter_vhd] smt2: -- (c) Copyright 1999 - 2018 Verific Design Automation Inc. All rights reserved
SBY 19:44:24 [counter_vhd] smt2: -- Compilation time was Wed Mar 7 20:49:25 2018
SBY 19:44:24 [counter_vhd] smt2: -- This program will expire on Fri Jun 15 20:49:25 2018
SBY 19:44:24 [counter_vhd] smt2: finished (returncode=0)
SBY 19:44:24 [counter_vhd] engine_0: starting process "cd counter_vhd; yosys-smtbmc --presat --unroll --noprogress -t 20 --append 0 --dump-vcd engine_0/trace.vcd --dump-vlogtb engine_0/trace_tb.v --dump-smtc engine_0/trace.smvc mode l/design_smt2.smt2"
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Solver: yices
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Checking assumptions in step 0..
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Checking assertions in step 0..
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 BMC failed!
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Assert failed in counter.copy: counter.vhd.sv:55
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Writing trace to VCD file: engine_0/trace.vcd
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Writing trace to Verilog testbench: engine_0/trace_tb.v
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Writing trace to constraints file: engine_0/trace.smvc
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Status: FAILED (!)
SBY 19:44:25 [counter_vhd] engine_0: finished (returncode=1)
SBY 19:44:25 [counter_vhd] engine_0: Status returned by engine: FAIL
SBY 19:44:25 [counter_vhd] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:03 (3)
SBY 19:44:25 [counter_vhd] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:01 (1)
SBY 19:44:25 [counter_vhd] summary: engine_0 (smtbmc) returned FAIL
SBY 19:44:25 [counter_vhd] summary: counterexample trace: counter_vhd/engine_0/trace.vcd
SBY 19:44:25 [counter_vhd] DONE (FAIL, rc=2)
dan@beta:~/vhdl$
```

# 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

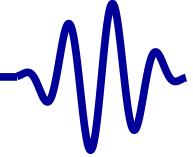
Sequences

Parting Thoughts

Run: % sbt -f counter\_vhd.sbt

```
dan@beta:~/vhdl$ sbt -f counter_vhd.sbt
SBY 19:44:22 [counter_vhd] Copy 'counter.vhd' to 'counter_vhd/src/counter.vhd'.
SBY 19:44:22 [counter_vhd] Copy 'counter.vhd' to 'counter_vhd/src/counter.vhd'.
SBY 19:44:22 [counter_vhd] Copy 'counter_vhd.sv' to 'counter_vhd/src/counter_vhd.sv'.
SBY 19:44:22 [counter_vhd] engine_0: smtbmc
SBY 19:44:22 [counter_vhd] base: starting process "cd counter_vhd/src; yosys -ql ../model/design.log ../model/design.yos"
SBY 19:44:22 [counter_vhd] base: -- (c) Copyright 1999 - 2018 Verific Design Automation Inc. All rights reserved
SBY 19:44:22 [counter_vhd] base: -- Compilation time was Wed Mar 7 20:49:25 2018
SBY 19:44:22 [counter_vhd] base: -- This program will expire on Fri Jun 15 20:49:25 2018
SBY 19:44:24 [counter_vhd] base: finished (returncode=0)
SBY 19:44:24 [counter_vhd] smt2: starting process "cd counter_vhd/model; yosys -ql design_smt2.log design_smt2.yos"
SBY 19:44:24 [counter_vhd] smt2: -- (c) Copyright 1999 - 2018 Verific Design Automation Inc. All rights reserved
SBY 19:44:24 [counter_vhd] smt2: -- Compilation time was Wed Mar 7 20:49:25 2018
SBY 19:44:24 [counter_vhd] smt2: -- This program will expire on Fri Jun 15 20:49:25 2018
SBY 19:44:24 [counter_vhd] smt2: finished (returncode=0)
SBY 19:44:24 [counter_vhd] engine_0: starting process "cd counter_vhd; yosys-smtbmc --presat --unroll --noprogress -t 20 --append 0 --dump-vcd engine_0/trace.vcd --dump-vlogtb engine_0/trace_tb.v --dump-smtc engine_0/trace.smvc mode l/design_smt2.smt2"
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Solver: yices
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Checking assumptions in step 0..
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Checking assertions in step 0..
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 BMC failed!
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 assert failed in counter.copy: counter_vhd.sv:55
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Writing trace to VCD file: engine_0/trace.vcd
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Writing trace to Verilog testbench: engine_0/trace_tb.v
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Writing trace to constraints file: engine_0/trace.smvc
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Status: FAILED (!)
SBY 19:44:25 [counter_vhd] engine_0: finished (returncode=2)
SBY 19:44:25 [counter_vhd] engine_0: Status returned by engine: FAIL
SBY 19:44:25 [counter_vhd] summary: Elapsed clock time [H:M:S.F]: 0:00:03 (3)
SBY 19:44:25 [counter_vhd] summary: Elapsed process time [H:M:S.F]: 0:00:01 (1)
SBY 19:44:25 [counter_vhd] summary: engine_0 (smtbmc) returned FAIL
SBY 19:44:25 [counter_vhd] summary: counterexample trace: counter_vhd/engine_0/trace.vcd
SBY 19:44:25 [counter_vhd] DONE (FAIL, rc=2)
dan@beta:~/vhdl$
```

# 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

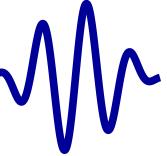
Cover

Sequences

Parting Thoughts

Look at source line 55, and fire up gtkwave

```
dan@beta:~/vhdl$ sby -f counter_vhd.sby
SBY 19:44:22 [counter_vhd] Copy 'counter.vhd' to 'counter_vhd/src/counter.vhd'.
SBY 19:44:22 [counter_vhd] Copy 'counter_vhd.sv' to 'counter_vhd/src/counter_vhd.sv'.
SBY 19:44:22 [counter_vhd] engine_0: smtbmc
SBY 19:44:22 [counter_vhd] base: starting process "cd counter_vhd/src; yosys -ql ./model/design.log ./model/design.yo"
SBY 19:44:22 [counter_vhd] base: -- (c) Copyright 1999 - 2018 Verific Design Automation Inc. All rights reserved
SBY 19:44:22 [counter_vhd] base: -- Compilation time was Wed Mar 7 20:49:25 2018
SBY 19:44:22 [counter_vhd] base: -- This program will expire on Fri Jun 15 20:49:25 2018
SBY 19:44:24 [counter_vhd] base: finished (returncode=0)
SBY 19:44:24 [counter_vhd] smt2: starting process "cd counter_vhd/model; yosys -ql design_smt2.log design_smt2.yo"
SBY 19:44:24 [counter_vhd] smt2: -- (c) Copyright 1999 - 2018 Verific Design Automation Inc. All rights reserved
SBY 19:44:24 [counter_vhd] smt2: -- Compilation time was Wed Mar 7 20:49:25 2018
SBY 19:44:24 [counter_vhd] smt2: -- This program will expire on Fri Jun 15 20:49:25 2018
SBY 19:44:24 [counter_vhd] smt2: finished (returncode=0)
SBY 19:44:24 [counter_vhd] engine_0: starting process "cd counter_vhd; yosys-smtbmc --presat --unroll --noprogress -t 20 --append 0 --dump-vcd engine_0/trace.vcd --dump-vlogtb engine_0/trace_tb.v --dump-smtc engine_0/trace.smvc mode_l/design_smt2.smt2"
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Solver: yices
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Checking assumptions in step 0..
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Checking assertions in step 0..
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 BMC failed!
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Assert failed in counter.copy: counter_vhd.sv:55
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Writing trace to VCD file: engine_0/trace.vcd
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Writing trace to Verilog testbench: engine_0/trace_tb.v
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Writing trace to constraints file: engine_0/trace.smvc
SBY 19:44:25 [counter_vhd] engine_0: ## 0:00:00 Status: FAILED (!)
SBY 19:44:25 [counter_vhd] engine_0: finished (returncode=1)
SBY 19:44:25 [counter_vhd] engine_0: Status returned by engine: FAIL
SBY 19:44:25 [counter_vhd] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:03 (3)
SBY 19:44:25 [counter_vhd] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:01 (1)
SBY 19:44:25 [counter_vhd] summary: engine_0 (smtbmc) returned FAIL
SBY 19:44:25 [counter_vhd] summary: counterexample trace: counter_vhd/engine_0/trace.vcd
SBY 19:44:25 [counter_vhd] DONE (FAIL, rc=2)
dan@beta:~/vhdl$
```

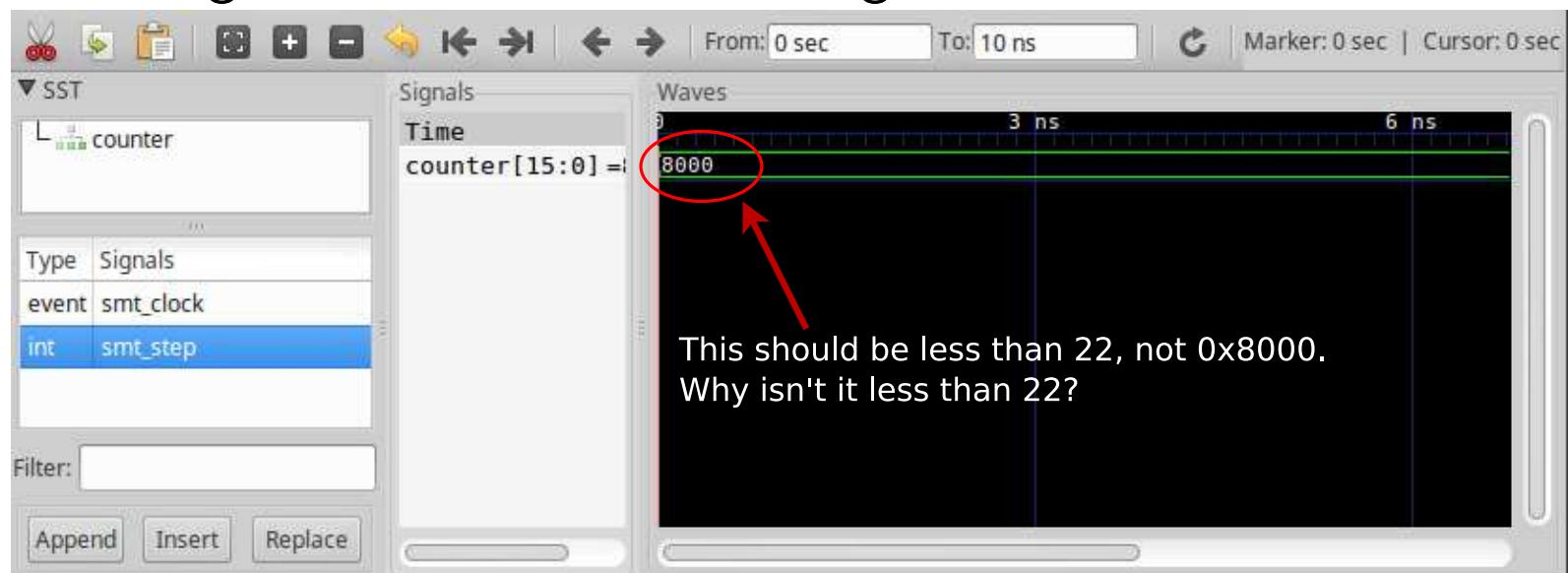
[Welcome](#)[Motivation](#)[Basics](#)[Basics](#)[General Rule](#)[Assert](#)[Assume](#)[BMC](#)

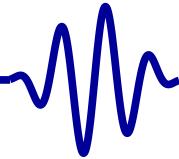
▷ Ex: Counter

Sol'n

[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

Run: % gtkwave counter\_vhd/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](#)[Parting Thoughts](#)

Run: % demo-rtl/counter\_vhd.v

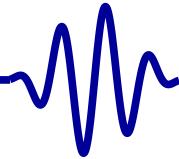
What did we do wrong?

```
44 //  
45 `default_nettype none  
46 //  
47 module counter_vhd(i_clk, i_start_signal, counts, o_busy);  
48   parameter [15:0] MAX_AMOUNT = 22;  
49   input wire [15:0] i_clk, i_start_signal;  
50   input wire [15:0] counts;  
51   output wire o_busy;  
52  
53 `ifdef FORMAL  
54   always @(*)  
55     assert(counts < MAX_AMOUNT);  
56 `endif  
57 endmodule  
58  
59 bind counter counter_vhd  
60   #(MAX_AMOUNT(MAX_AMOUNT)) copy (.);
```

Line 55, Here's the assertion that failed

55,1-8

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](#)[Parting Thoughts](#)

Run: % demo-rtl/counter\_vhd.v

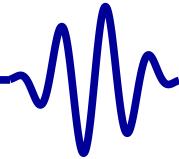
Notice anything wrong?

```
51 architecture behavior of counter is
52     signal counts : unsigned(15 downto 0);
53 begin
54
55     process(i_clk)
56     begin
57         if (rising_edge(i_clk)) then
58             if ((i_start_signal = '1') and (0 = counts)) then
59                 counts <= to_unsigned(MAX_AMOUNT-1, 16);
60             else
61                 counts <= counts - 1;
62             end if;
63         end if;
64     end process;
65
66     process(all)
67     begin
68         if (0 = counts) then
69             o_busy <= '1';
70         else
71             o_busy <= '0';
72         end if;
73     end process;

```

51,1

96%

[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](#)[Parting Thoughts](#)

Run: % demo-rtl/counter\_vhd.v

Notice anything wrong?

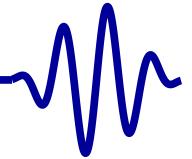
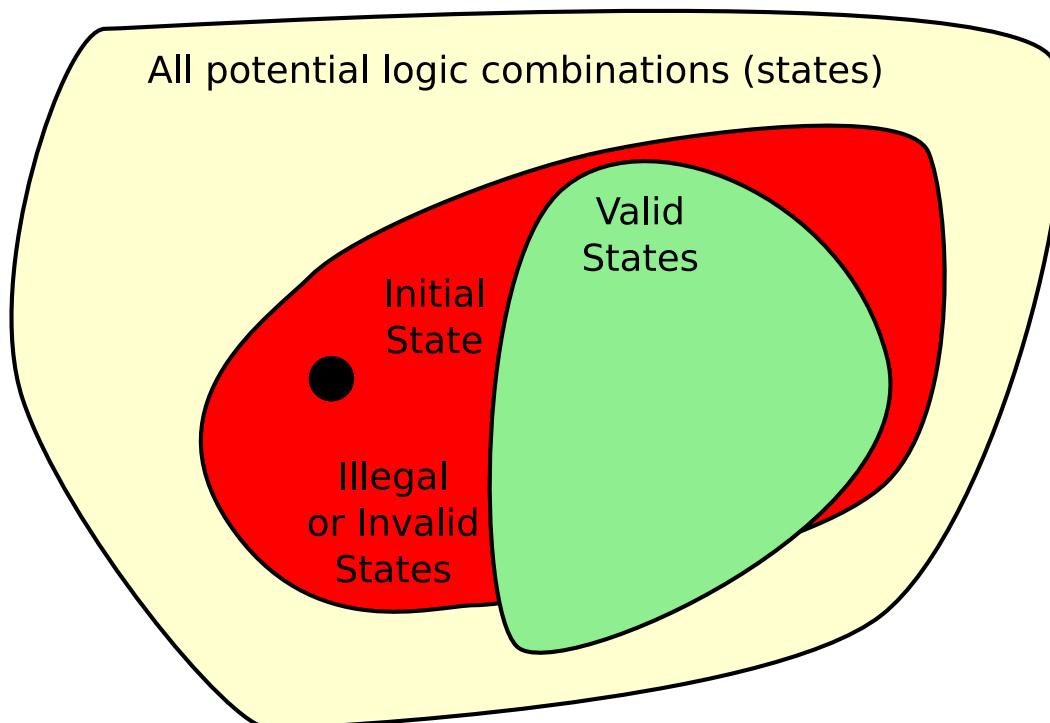
```
51 architecture behavior of counter is
52     signal counts : unsigned(15 downto 0);
53 begin
54
55     process(i_clk)
56     begin
57         if (rising_edge(i_clk)) then
58             if ((i_start_signal = '1') and (0 = counts)) then
59                 counts <= to_unsigned(MAX_AMOUNT-1, 16);
60             else
61                 counts <= counts - 1;
62             end if;
63         end if;
64     end process;
65
66     process(all)
67     begin
68         if (0 = counts) then
69             o_busy <= '1';
70         else
71             o_busy <= '0';
72         end if;
73     end process;

```

51,1

96%

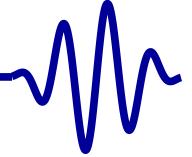
How about the missing initial value?

[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](#)[Parting Thoughts](#)

- 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

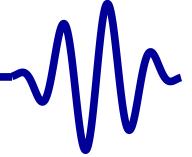
Multiple-Clocks

Cover

Sequences

Parting Thoughts

Try adding in the initial value, 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

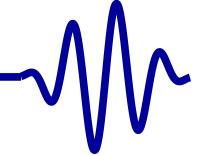
Sequences

Parting Thoughts

# 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

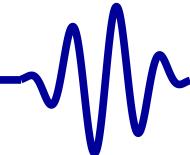
Cover

Sequences

Parting Thoughts

## 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

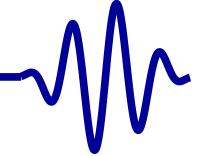
Parting Thoughts

- **\$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](#)[Parting Thoughts](#)

## \$past FV Rule

Only use \$past as a precondition

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

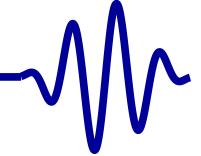
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[▷ Past Assertions](#)[Past Valid](#)[Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

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.vhd`

[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](#)[Parting Thoughts](#)

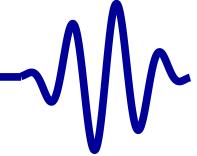
- This fails

```
always @(*)
```

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

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

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

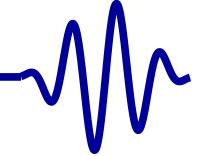
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[▷ Past Assertions](#)[Past Valid](#)[Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

- This succeeds

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

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

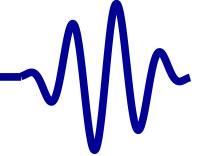
- Before time, counter is unconstrained.
- The solver can make it take on any value it wants in order to make things fail
- This will not show in the VCD file

[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](#)[Parting Thoughts](#)

Let's try again:

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

This should work, right?

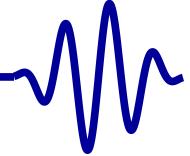
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[▷ Past Assertions](#)[Past Valid](#)[Examples](#)[Ex: Busy Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

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.

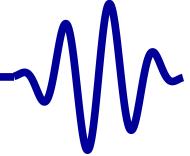
- `i_start_signal` is unconstrained before time
- `counter` is initially constrained to zero
- If `i_start_signal` is one before time,  
`counter` will still be zero when time begins

[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](#)[Parting Thoughts](#)

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](#)[Parting Thoughts](#)

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](#)[Parting Thoughts](#)

- 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);
```

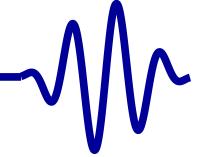
- Will this work?

[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](#)[Parting Thoughts](#)

- 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



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

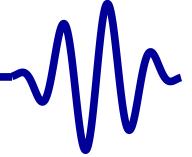
Multiple-Clocks

Cover

Sequences

Parting Thoughts

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](#)[Parting Thoughts](#)

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

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

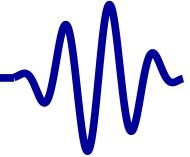
```
always @(*)
  if (!f_past_valid)
    assume(i_RESET);
```

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](#)[Parting Thoughts](#)

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](#)[Parting Thoughts](#)

Two times registers must have their reset value

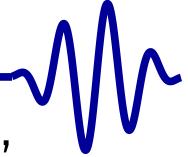
- Initially
- Following a reset

```
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](#)[Parting Thoughts](#)

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
```

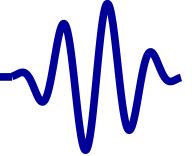
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[Past](#)[\\$past Rule](#)[Past Assertions](#)[Past Valid](#)[Examples](#)[Ex: Busy](#)[▷ Counter](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting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

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:

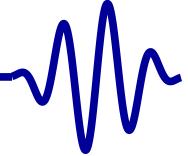
- Gain some confidence using formal methods to prove that alternative designs are equivalent

[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](#)[Parting Thoughts](#)

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

```
process(i_clk)
begin
    if (rising_edge(i_clk)) then
        if (i_reset) then
            counter <= to_unsigned(0, 16);
        elsif ((i_start_signal = '1')
                and (0 = counter)) then
            counter <= to_unsigned(MAX_AMOUNT - 1, 16);
        elsif (0 /= counter) then
            counter <= counter - 1;
        end if;
    end if;
end process;

o_busy <= '0' when (0 = counter) else '1';
```



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

Parting Thoughts

You can find the design in `exercise-03/busyctr.vhd`.

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

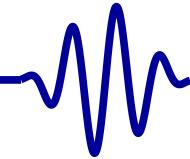
Make sure you check `o_busy` both when `counter == 0` and `counter != 0`

*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](#)[Parting Thoughts](#)

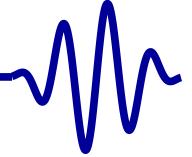
### Exercise:

1. Make o\_busy a clocked register

```
process(i_clk)
begin
    if (rising_edge(i_clk)) then
        o_busy <= — Your logic goes here
    end if;
end process;
```

2. Prove that o\_busy is true if and only if the counter is non-zero

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



Welcome

Motivation

Basics

Clocked and \$past

$\triangleright k$  Induction

Lesson Overview

vs BMC

General Rule

The Trap

Results

Examples

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

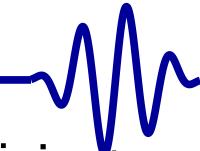
Sequences

Parting Thoughts

# $k$ 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

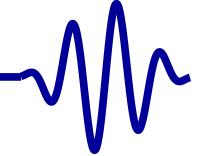
Parting Thoughts

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](#)[Parting Thoughts](#)

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

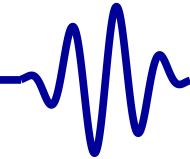
$$\sum_{n=0}^{N-1} x^n = x^0 = \frac{1-x}{1-x}$$

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

Parting Thoughts

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

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

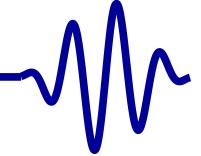
$$\sum_{n=0}^N x^n = x^N + \sum_{n=0}^{N-1} x^n = x^N + \frac{1-x^N}{1-x}$$

- 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](#)[Parting Thoughts](#)

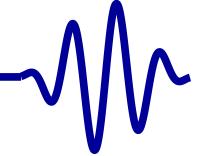
Suppose  $\forall n : P[n]$  is what we wish to prove

- 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](#)[Parting Thoughts](#)

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

- Traditional induction

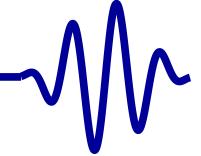
- Base case: show  $P[0]$
  - Inductive case: show  $P[n] \rightarrow P[n + 1]$

- $k$  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](#)[Parting Thoughts](#)

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

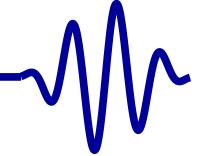
- Traditional induction

- 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](#)[Parting Thoughts](#)

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

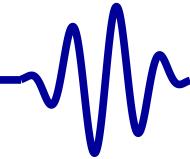
- Traditional induction

- 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](#)[Parting Thoughts](#)

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.*

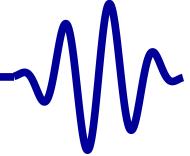
This is the BMC pass we've already done.

- **Inductive Step:**

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

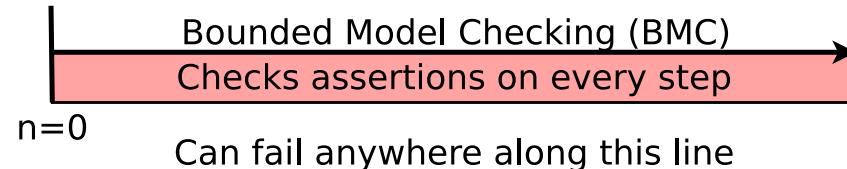
*Assume the  $N + 1$  step violates no assumptions, ...*

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

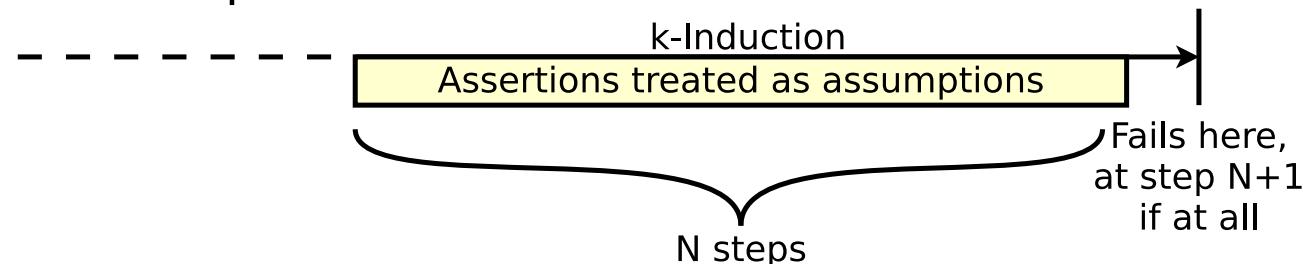
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[▷ vs BMC](#)[General Rule](#)[The Trap](#)[Results](#)[Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

BMC and induction are very different.

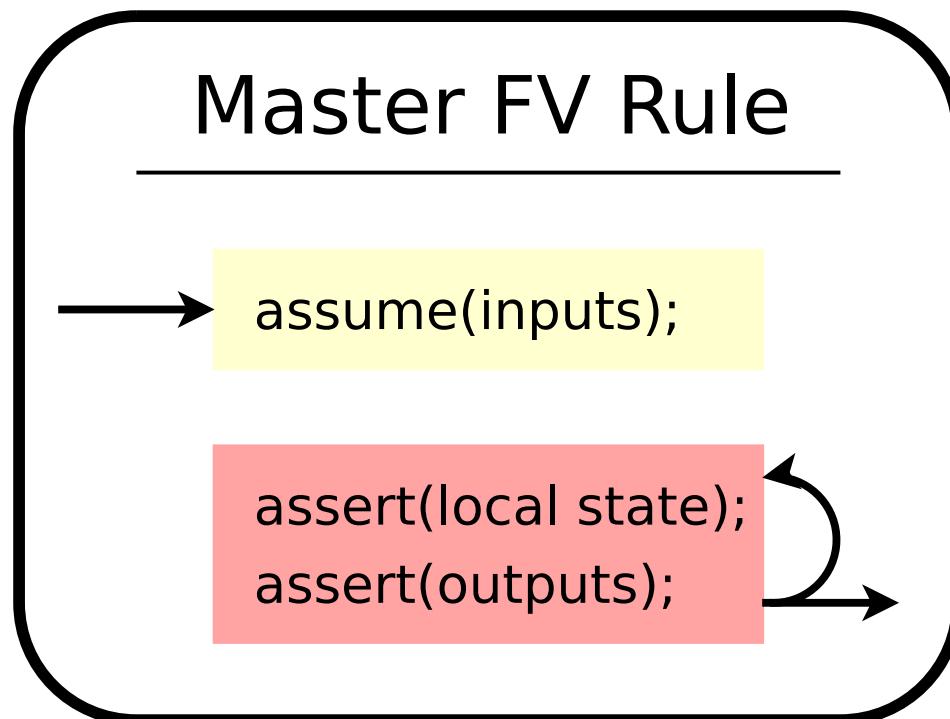
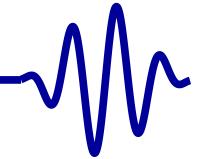
- BMC, the base case



- 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.



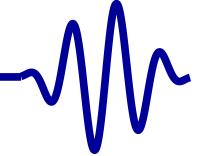
The general rule hasn't changed:

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

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



# Checkers



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

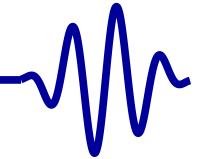
Sequences

Parting Thoughts

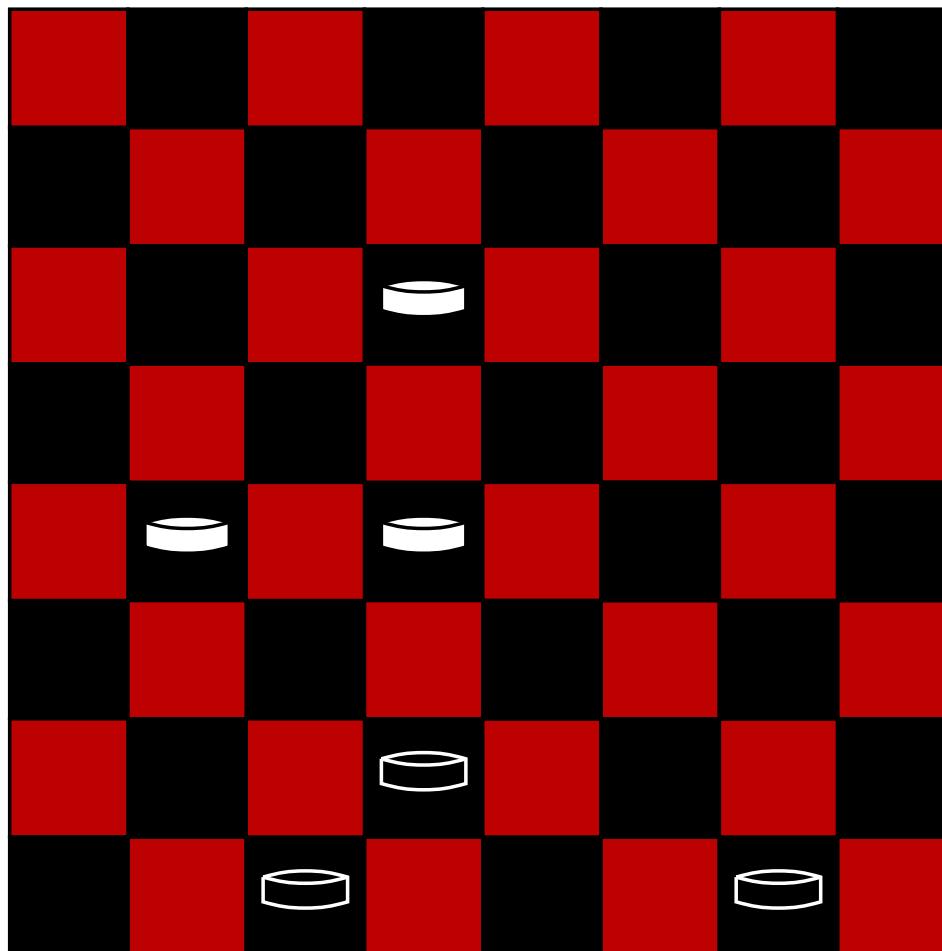
Some assertions:

- 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

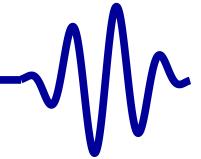
Where might the induction engine start?



- 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
- Parting Thoughts



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

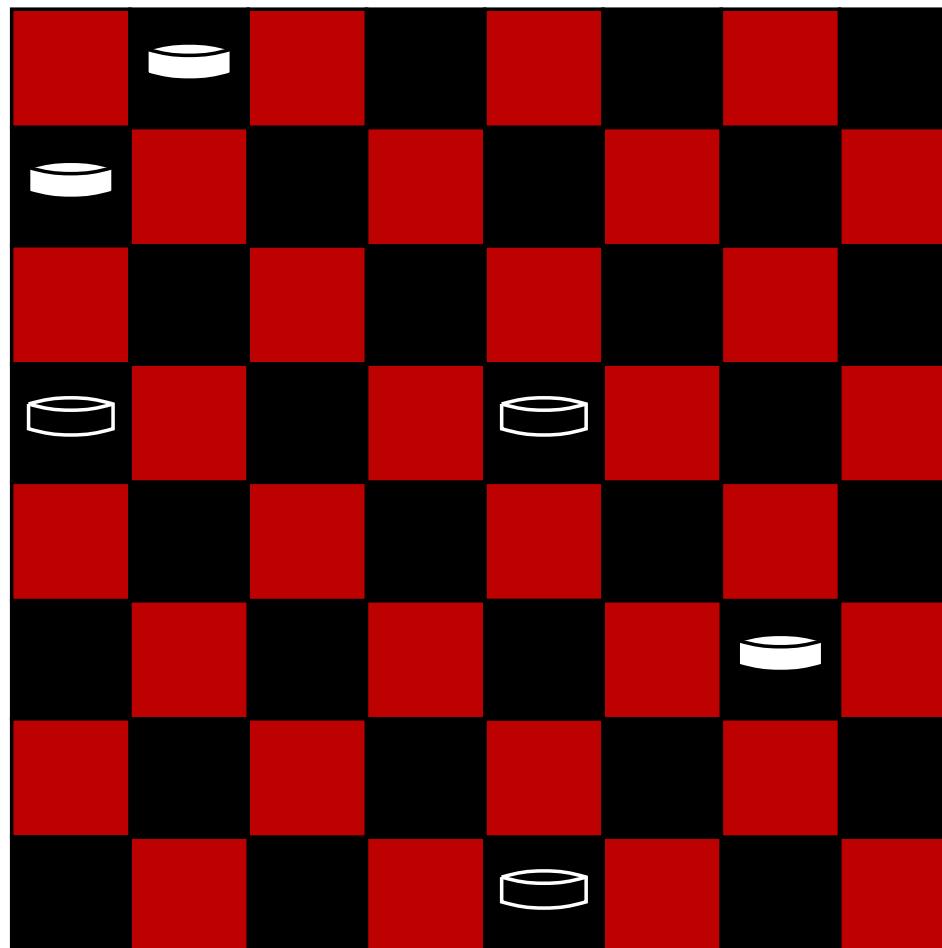
Invariants

Multiple-Clocks

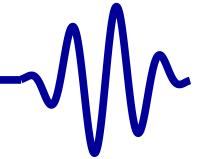
Cover

Sequences

Parting Thoughts



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

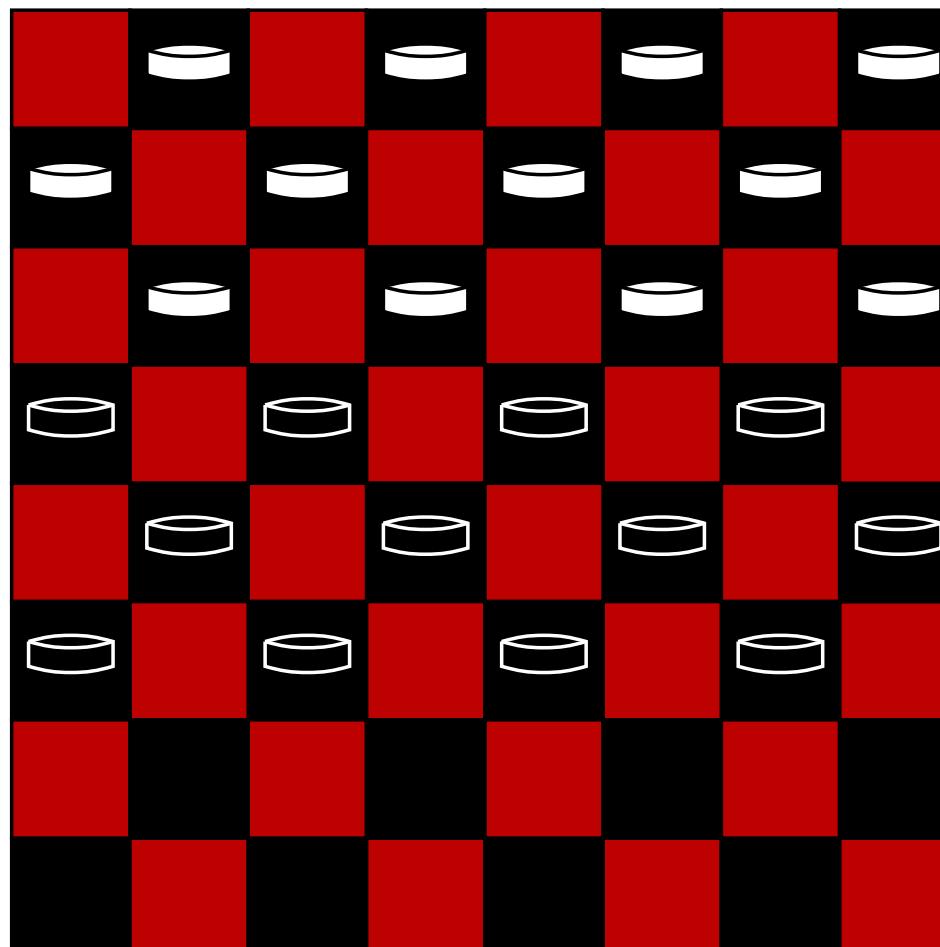
Invariants

Multiple-Clocks

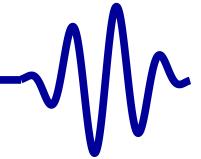
Cover

Sequences

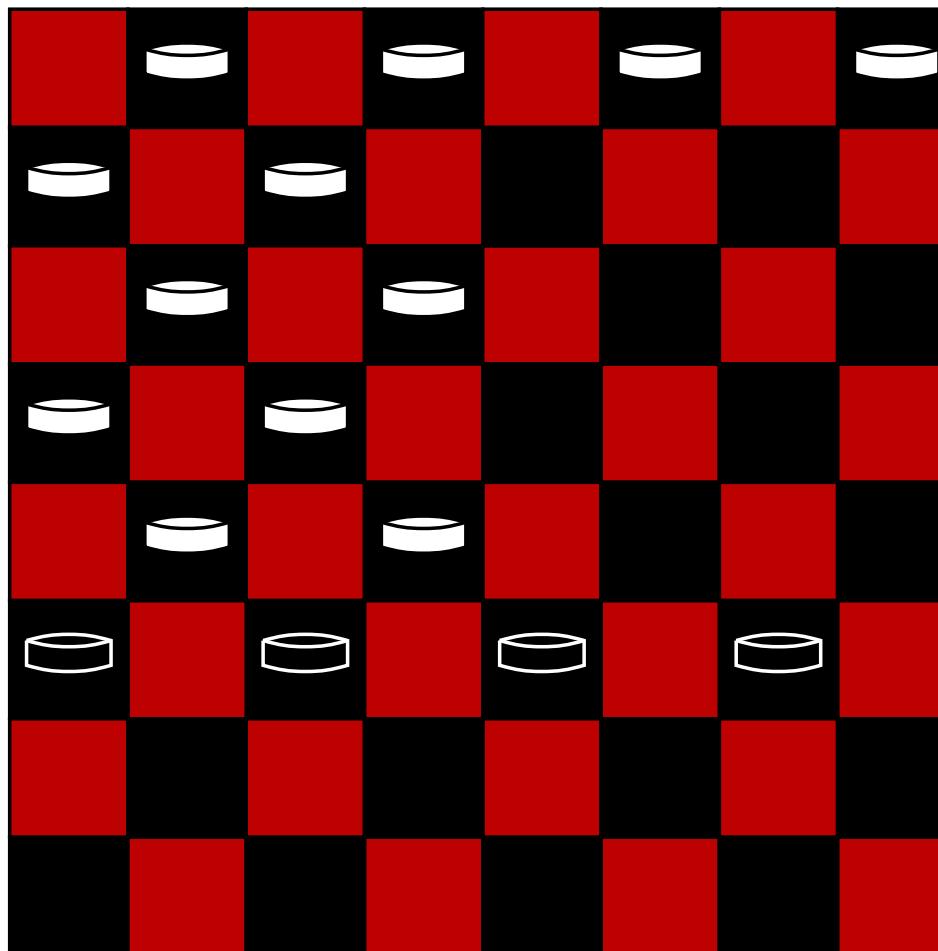
Parting Thoughts



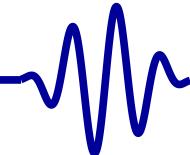
Black's going to . . . , huh?



- 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
- Parting Thoughts



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

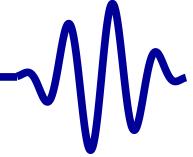
Cover

Sequences

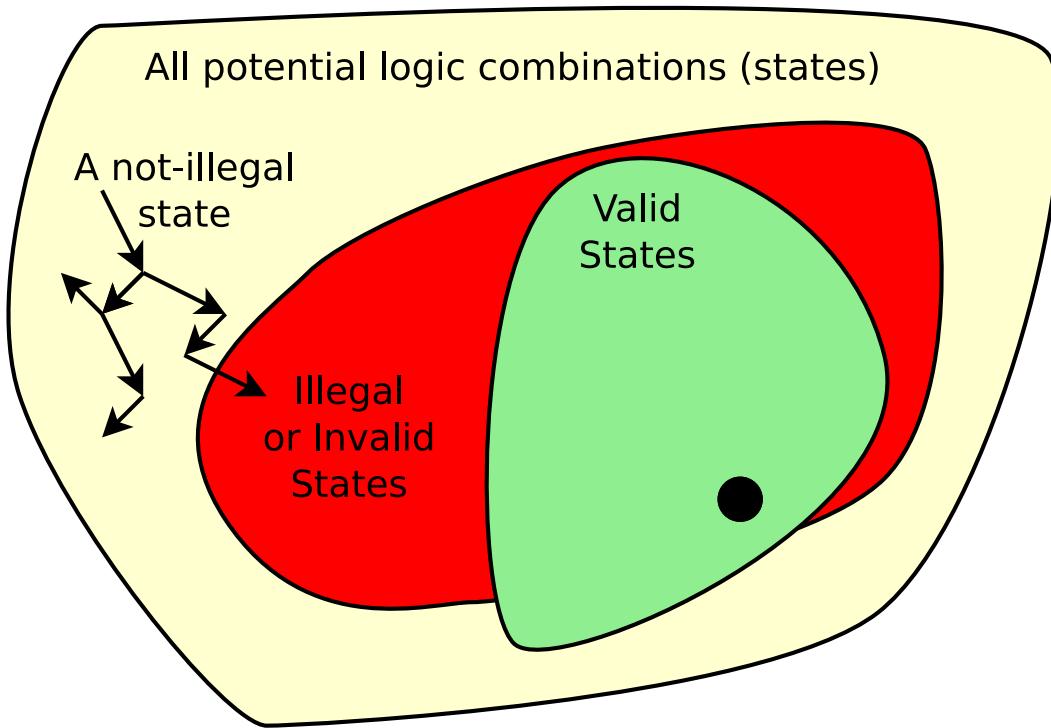
Parting Thoughts

## What can we learn from Checkers?

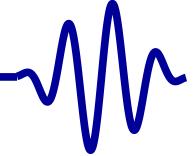
- Inductive step starts in the *middle of the game*  
Only the assumptions and asserts are used to validate the game
- All of the FF's (variables) start in arbitrary states  
These states are *only* constrained by your assumptions and assertions.
- Your formal constraints are required to limit the allowable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



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



- 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

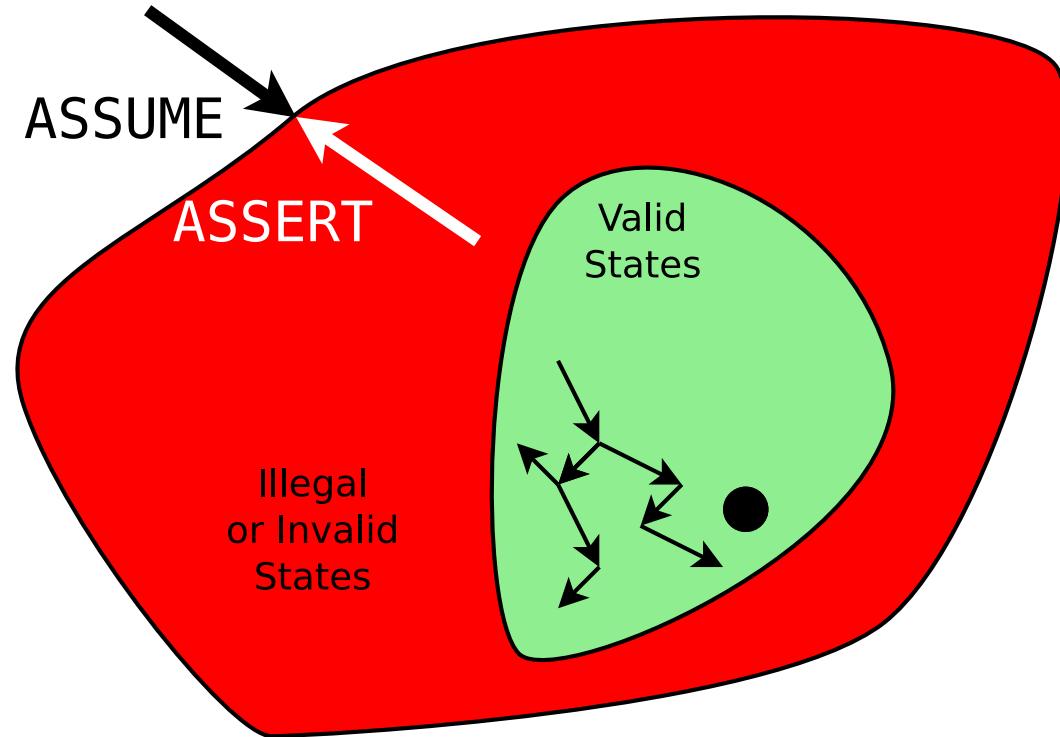
---

Cover

---

Sequences

---

Parting Thoughts

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](#)[Parting Thoughts](#)

Unlike BMC, the results of induction might be inconclusive

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

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

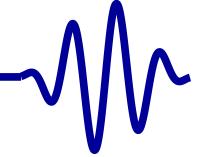
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview  
vs BMC](#)[General Rule](#)[The Trap](#)[▷ Results](#)[Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

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



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Lesson Overview

vs BMC

General Rule

The Trap

Results

▷ Examples

Bus Properties

Free Variables

Abstraction

Invariants

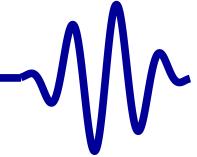
Multiple-Clocks

Cover

Sequences

Parting Thoughts

- 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](#)[Parting Thoughts](#)

This design would pass *many* steps of BMC

```
signal counter : unsigned(15 downto 0) := 0;  
  
process(clk)  
begin  
    if (rising_edge(clk)) then  
        counter <= counter + 1;  
    end if;  
end process;
```

```
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](#)[Parting Thoughts](#)

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

```
signal : counter : unsigned(15 downto 0) := 0;  
  
if (rising_edge(clk)) then  
  if (counter = to_unsigned(22, 16)) then  
    counter <= 0;  
  else  
    counter <= counter + 1;  
  end if;  
end if;
```

```
always @(*)  
  assert(counter != 16'd500);
```

Can you explain why not?

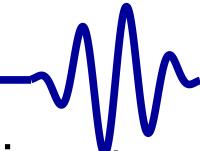
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[Results](#)[▷ Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

With one simple change, this design will now pass induction

```
signal : counter : unsigned(15 downto 0) := 0;  
  
if (rising_edge(clk)) then  
  if (counter = to_unsigned(22, 16)) then  
    counter <= 0;  
  else  
    counter <= counter + 1;  
  end if;  
end if;
```

```
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](#)[Parting Thoughts](#)

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

```
signal sa : unsigned(15 downto 0) := 0;  
signal sb : unsigned(15 downto 0) := 0;  
  
if (rising_edge(clk)) then  
begin  
  
    sa <= sa(14 downto 0) & i_bit;  
    sb <= sb(14 downto 0) & i_bit;  
  
end if;
```

```
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](#)[Parting Thoughts](#)

This design is almost identical to the last one, yet fails induction.  
The key difference is the **if** (*i\_ce* = '1').

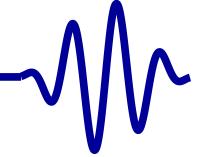
```
signal sa : unsigned(15 downto 0) := 0;  
signal sb : unsigned(15 downto 0) := 0;  
  
if (rising_edge(clk)) then  
begin  
    if (i_ce = '1') then  
        sa <= sa(14 downto 0) & i_bit;  
        sb <= sb(14 downto 0) & i_bit;  
    end if;  
end if;
```

```
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

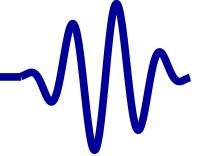
Parting Thoughts

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

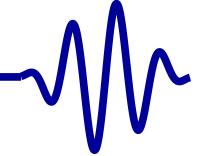
Parting Thoughts

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

Parting Thoughts

Several approaches to fixing this:

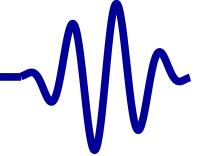
1. **assume(i\_ce);**  
*Doesn't really test the design*
2. opt\_merge –share\_all, yosys option  
*Works for some designs*
3. **assert(sa == sb);**

[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](#)[Parting Thoughts](#)

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

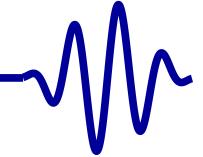
Cover

Sequences

Parting Thoughts

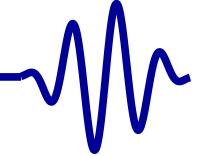
Several approaches to fixing this:

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](#)[Parting Thoughts](#)

Several approaches to fixing this:

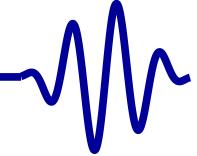
1. **assume(i\_ce);**  
*Doesn't really test the design*
2. opt\_merge –share\_all, yosys option  
*Works for some designs*
3. **assert(sa == sb);**  
*Best, but only works when sa and sb are visible*
4. Insist on no more than  $M$  clocks between i\_ce's
5. Use a different prover, under the [**engines**]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

Several approaches to fixing this:

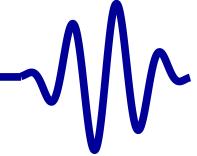
1. **assume(i\_ce);**  
*Doesn't really test the design*
2. opt\_merge –share\_all, yosys option  
*Works for some designs*
3. **assert(sa == sb);**  
*Best, but only works when sa and sb are visible*
4. Insist on no more than  $M$  clocks between i\_ce's
5. Use a different prover, under the [**engines**]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

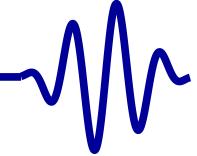
Here's how we'll change our sby file:

```
[options]
mode prove
[engines]
smtbmc
[script]
read -vhdl lfsr_fib.vhd
read -vhdl dblpipe.vhd
read -formal dblpipe_vhd.sv
prep -top dblpipe
opt_merge -share_all
[files]
lfsr_fib.vhd
dblpipe.vhd
dblpipe_vhd.sv
```

[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](#)[Parting Thoughts](#)

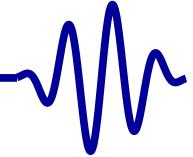
Here's how we'll change our sby file:

```
[options]
mode prove ← Use BMC and k-induction
[engines]
smtbmc
[script]
read -vhdl lfsr_fib.vhd
read -vhdl dblpipe.vhd
read -formal dblpipe_vhd.sv
prep -top dblpipe
opt_merge -share_all
[files]
lfsr_fib.vhd
dblpipe.vhd
dblpipe_vhd.sv
```

[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](#)[Parting Thoughts](#)

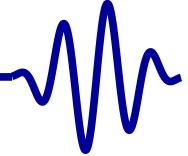
Here's how we'll change our sby file:

```
[options]
mode prove
[engines]
smtbmc ← Other potential engines would go here
[script]
read -vhdl lfsr_fib.vhd
read -vhdl dblpipe.vhd
read -formal dblpipe_vhd.sv
prep -top dblpipe
opt_merge -share_all
[files]
lfsr_fib.vhd
dblpipe.vhd
dblpipe_vhd.sv
```

[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](#)[Parting Thoughts](#)

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

```
[options]
mode prove
[engines]
smtbmc
[script]
read -vhdl lfsr_fib.vhd
read -vhdl dblpipe.vhd
read -formal dblpipe_vhd.sv
prep -top dblpipe
opt_merge -share_all ← Here's where opt_merge would go
[files]
lfsr_fib.vhd
dblpipe.vhd
dblpipe_vhd.sv
```

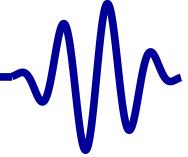
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[Results](#)[▷ Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

### Exercise #4: dblpipe.vhd

```
one: lfsr_fib port map (
    i_clk => i_clk, i_reset => '0',
    i_ce  => i_ce,  i_in  => i_data,
    o_bit => a_data);

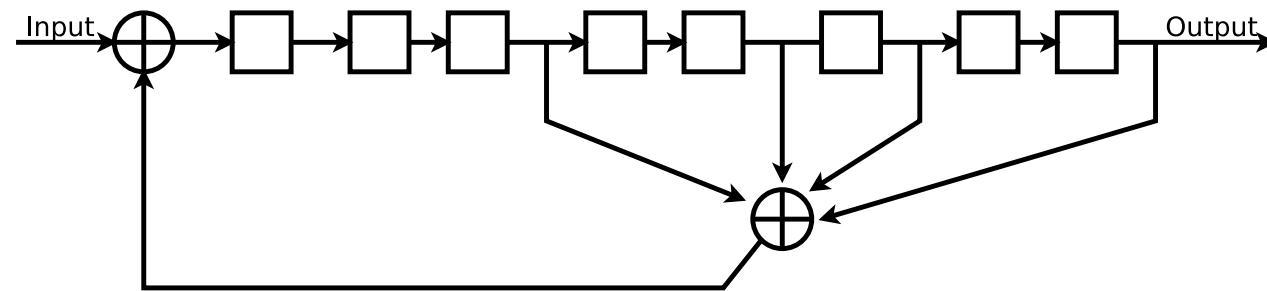
two: lfsr_fib port map (
    i_clk => i_clk, i_reset => '0',
    i_ce  => i_ce,  i_in  => i_data,
    o_bit => b_data);

process(a_data, b_data)
begin
    o_data <= a_data xor b_data;
end process;
```

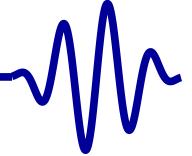
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[Results](#)[▷ Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

### Exercise #4: dblpipe.vhd

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



```
sreg(LN-2 downto 0) <= sreg(LN-1 downto 1);  
sreg(LN-1) <= (xor (sreg and TAPS)) xor 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](#)[Parting Thoughts](#)

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

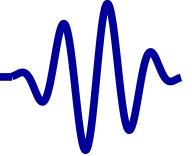
```
process(i_clk)
begin
    if (rising_edge(i_clk)) then
        if (i_reset = '1') then
            sreg <= INITIAL_FILL;
        elsif (i_ce = '1') then
            sreg(LN-2 downto 0) <= sreg(LN-1 downto 1);
            sreg(LN-1) <= (xor (sreg and TAPS))
                xor i_in;
        end if;
    end if;
end process;

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

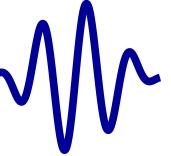
Cover

Sequences

Parting Thoughts

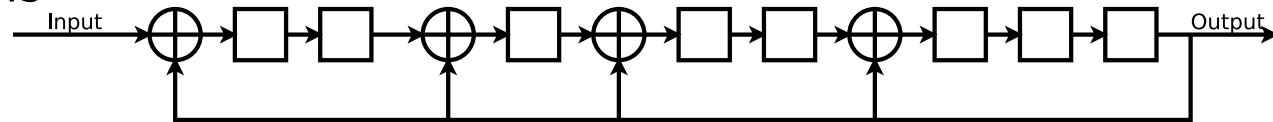
## Exercise #4:

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

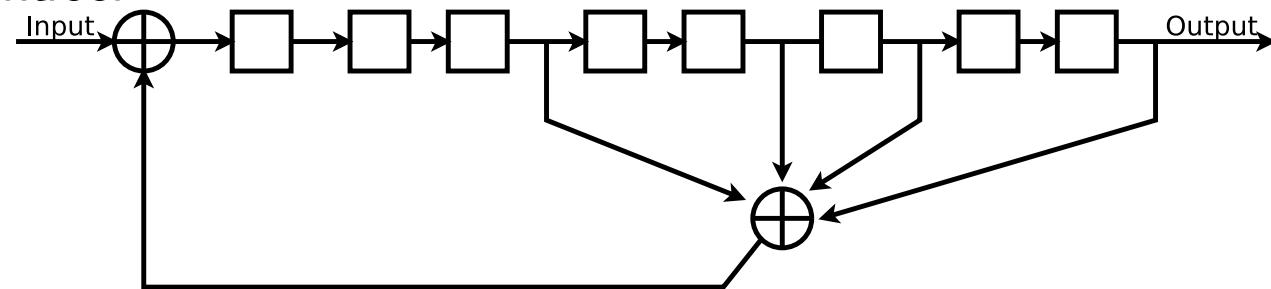
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Lesson Overview](#)[vs BMC](#)[General Rule](#)[The Trap](#)[Results](#)[▷ Examples](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

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

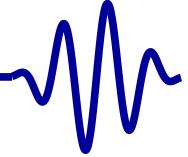
Parting Thoughts

## Exercise #5:

- exercise-05/ contains files lfsr\_equiv.vhd, lfsr\_gal.vhd, and lfsr\_fib.vhd.
- lfsr\_gal.vhd contains a Galois version of an LFSR
- lfsr\_fib.vhd contains a Fibonacci version of the same LFSR
- lfsr\_equiv.vhd 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

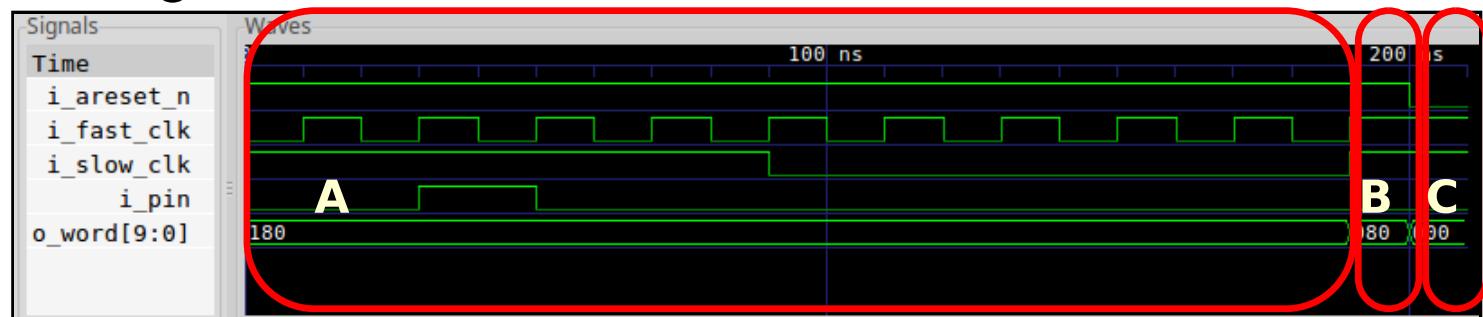
Multiple-Clocks

Cover

Sequences

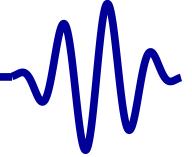
Parting Thoughts

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

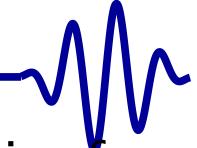
Multiple-Clocks

Cover

Sequences

Parting Thoughts

# Bus Properties

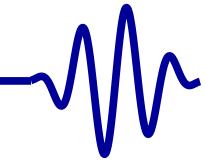
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[▷ Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

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

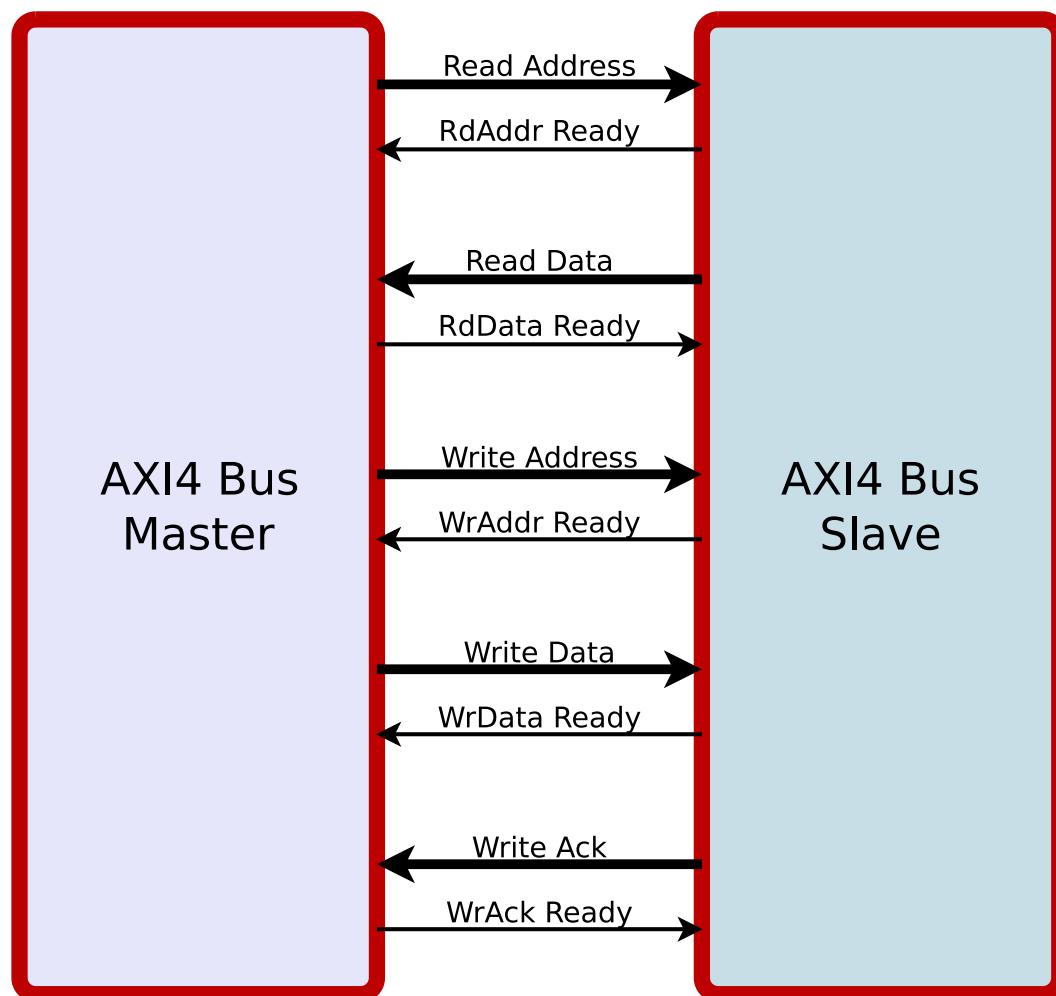
Invariants

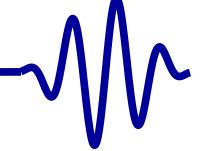
Multiple-Clocks

Cover

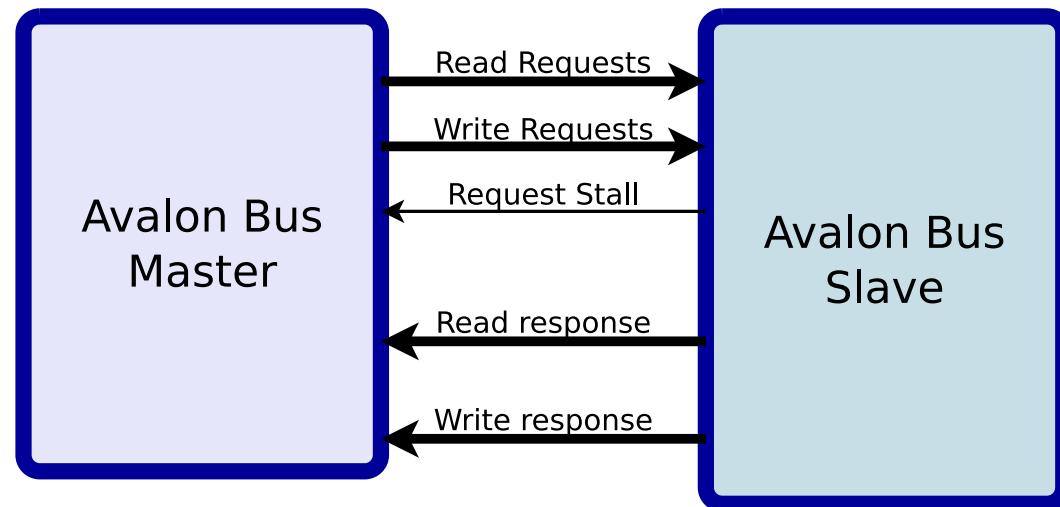
Sequences

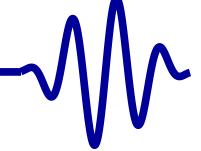
Parting Thoughts



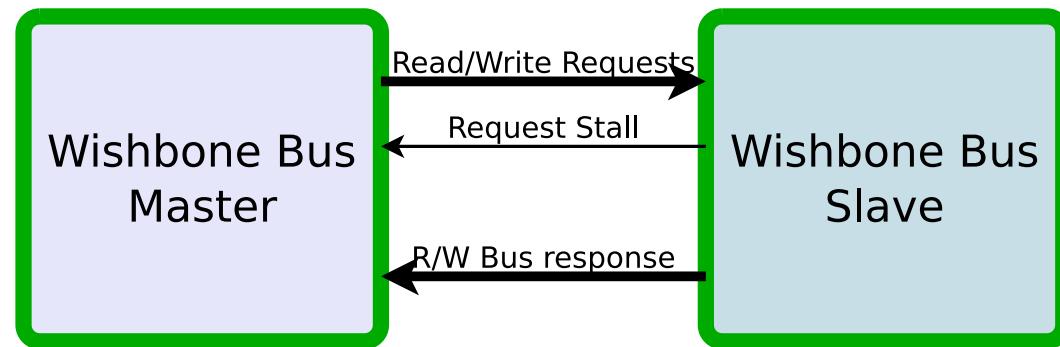


- Welcome
- Motivation
- Basics
- Clocked and \$past
- $k$  Induction
- Bus Properties
  - Ex: WB Bus
  - AXI
  - ▷ Avalon
  - Wishbone
  - WB Basics
  - WB Basics
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Cover
- Sequences
- Parting Thoughts





- 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
- Parting Thoughts



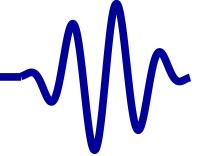
- Why use the Wishbone? *It's simpler!*

[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](#)[Parting Thoughts](#)

From the master's perspective:

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

# WB Signals



Welcome

Motivation

Basics

Clocked and \$past

*k* Induction

Bus Properties

Ex: WB Bus

AXI

Avalon

Wishbone

▷ WB Basics

WB Basics

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

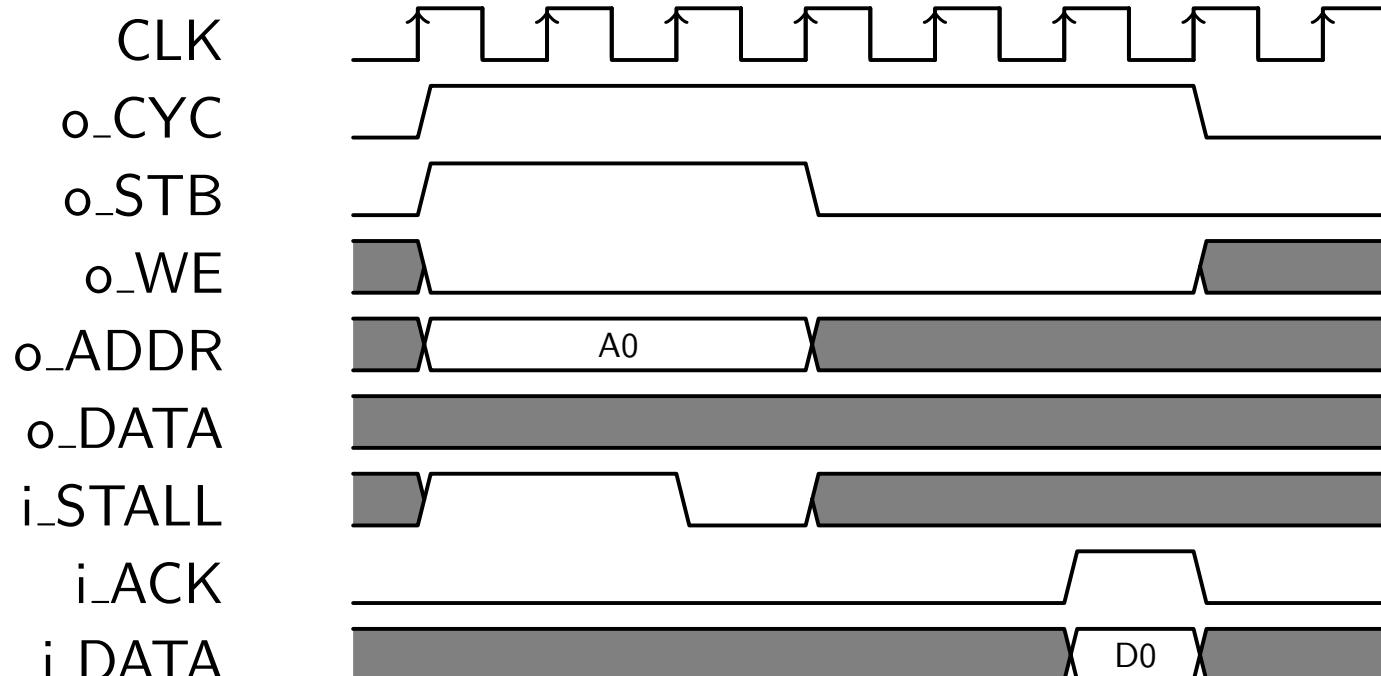
Parting Thoughts

From the slave's perspective:

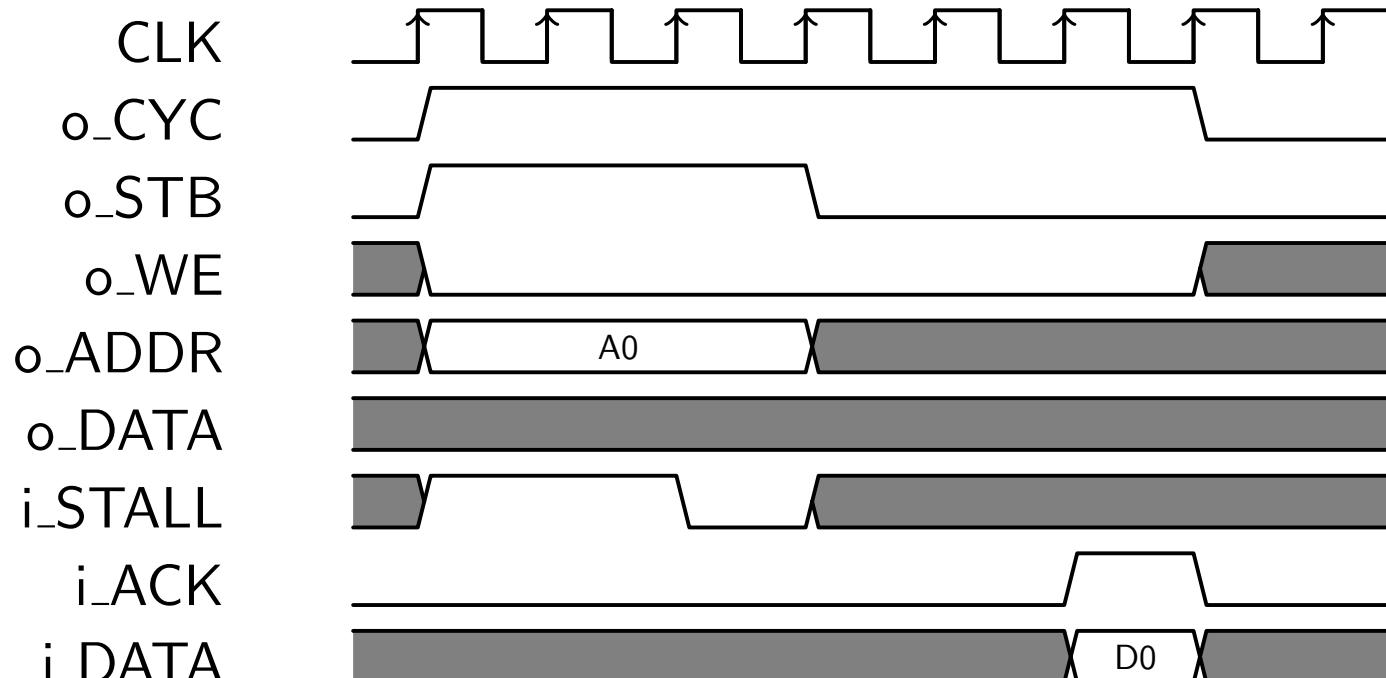
Specification name	My name
CYC_I	i_wb_cyc
STB_I	i_wb_stb
WE_I	i_wb_we
ADDR_I	i_wb_addr
DATA_I	i_wb_data
SEL_I	i_wb_sel
STALL_O	o_wb_stall
ACK_O	o_wb_ack
DATA_O	o_wb_data
ERR_O	o_wb_err

To swap perspectives from master to slave ...

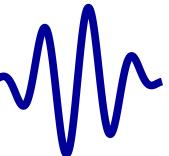
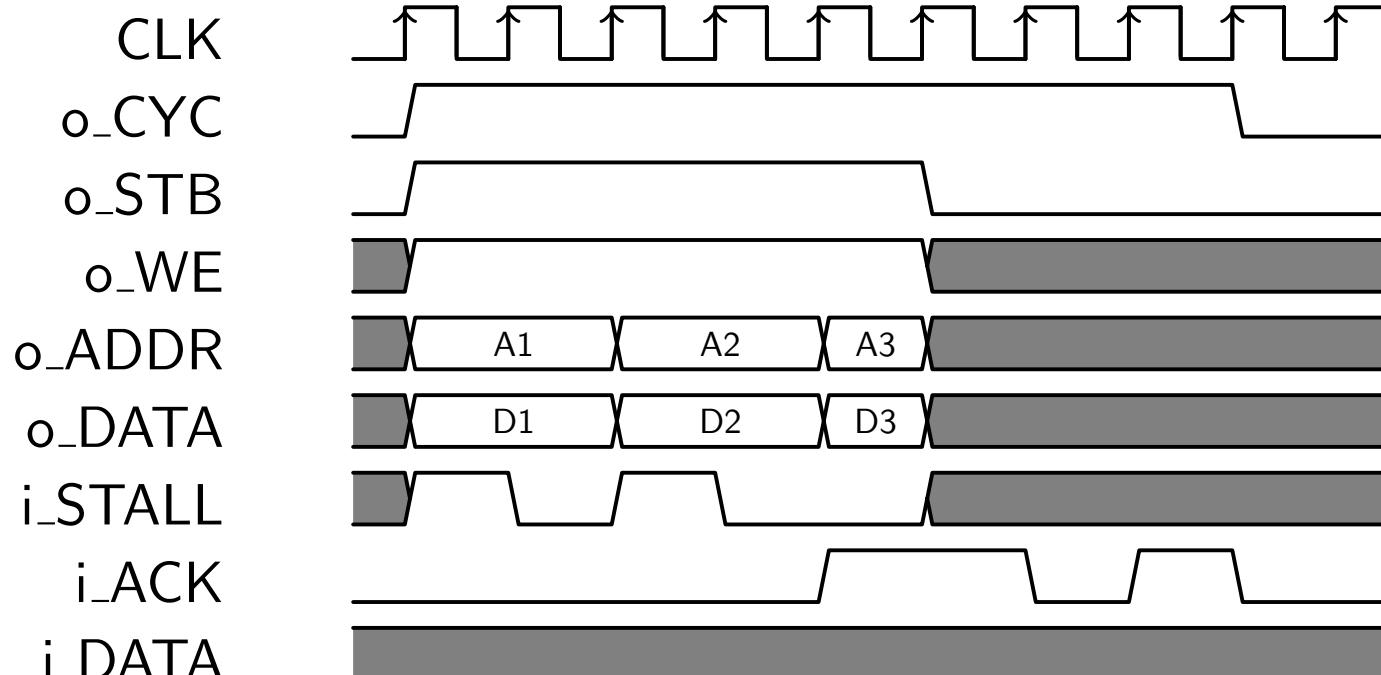
- Swap the port direction
- Swap the **assume()** statements for **assert()**s

[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](#)[Parting Thoughts](#)

- 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](#)[Parting Thoughts](#)

- 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](#)[Parting Thoughts](#)

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

Parting Thoughts

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

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

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

- 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](#)[Parting Thoughts](#)

- 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](#)[Parting Thoughts](#)

- 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

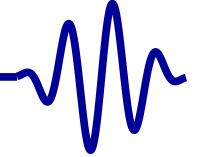
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[▷ WB Basics](#)[WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

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

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

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

# GT CYC and STB



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Ex: WB Bus

AXI

Avalon

Wishbone

WB Basics

▷ WB Basics

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Parting Thoughts

- 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](#)[Parting Thoughts](#)

The rule: the slave (external) cannot stall the master more than F\_OPT\_MAXSTALL counts:

```
if (rising_edge(i_clk)) then
    if ((i_reset = '1') or (o_CYC = '0')
        or ((o_STB='1')
            and (i_stall = '0')))) then
        f_stall_count <= 0;
    else
        f_stall_count <= f_stall_count + 1;
    end if;
end if;
```

```
always @ (posedge 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](#)[Parting Thoughts](#)

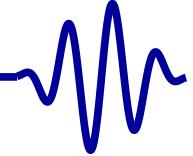
The rule: the slave can only respond to requests

```
process(i_clk)
begin
    if (rising_edge(i_clk)) then
        f_nreqs <= to_unsigned(0,F_LGDEPTH);
    elsif ((i_STB = '1') and (o_STALL = '0')) then
        f_nreqs <= f_nreqs + 1;
    end if;
end process;
```

— Need a similar counter for acknowledgements

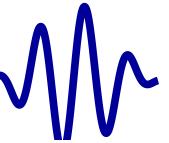
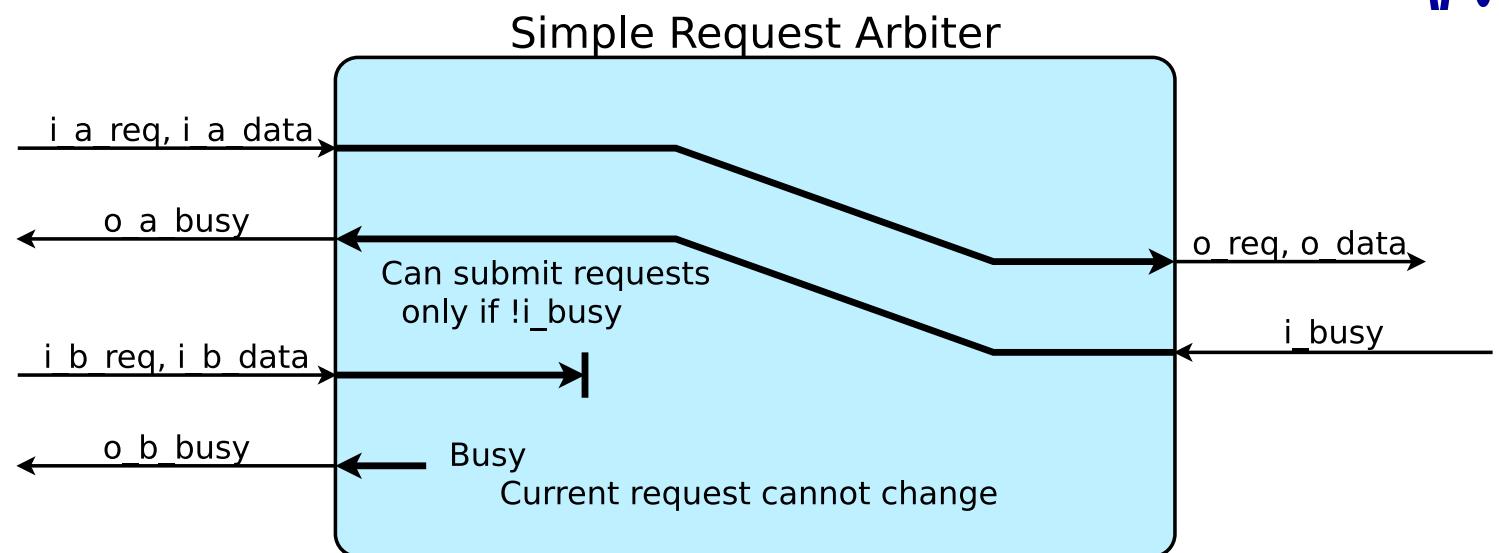
```
always @(*)
if (f_nreqs == f_nacks)
    assert (!o_ACK);
```

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](#)[Parting Thoughts](#)

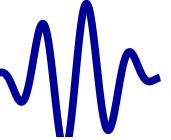
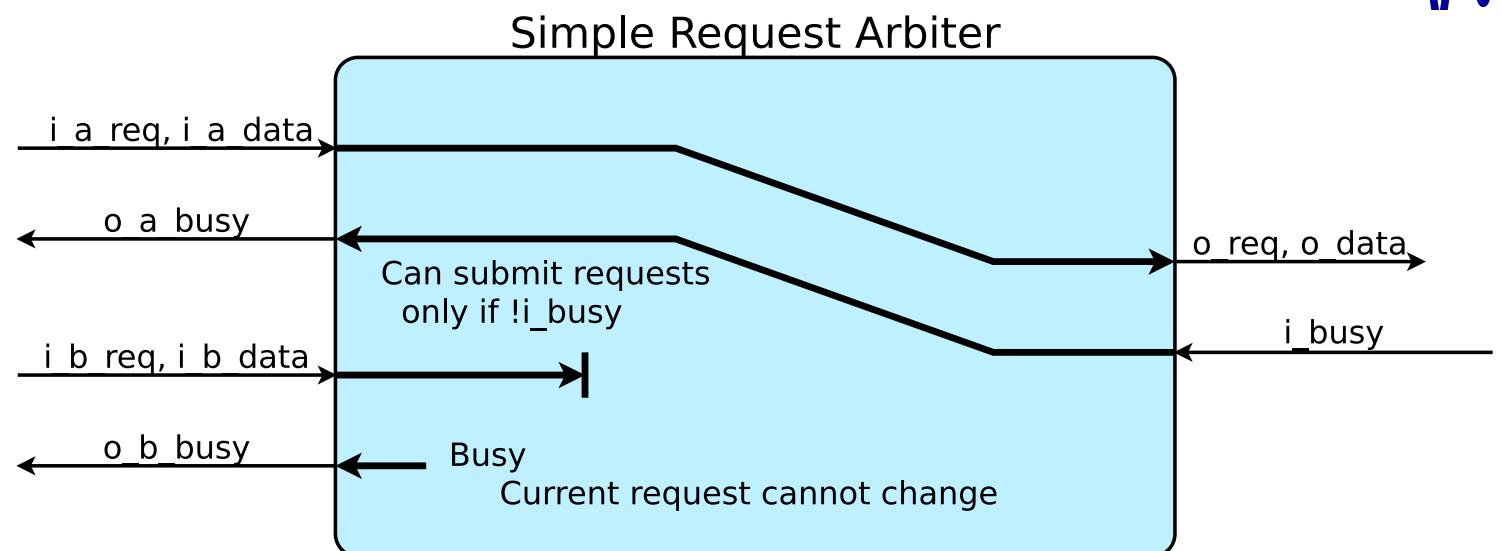
Let's build up to proving a WB arbiter

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

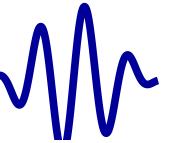
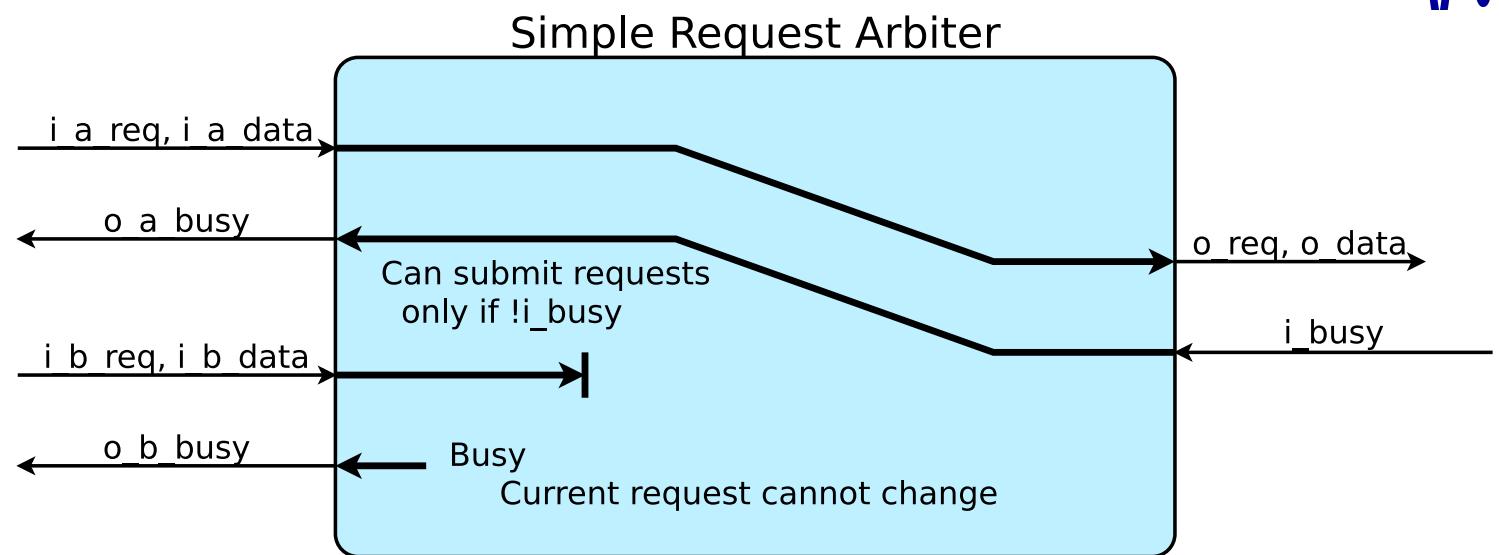
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

## The basics

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

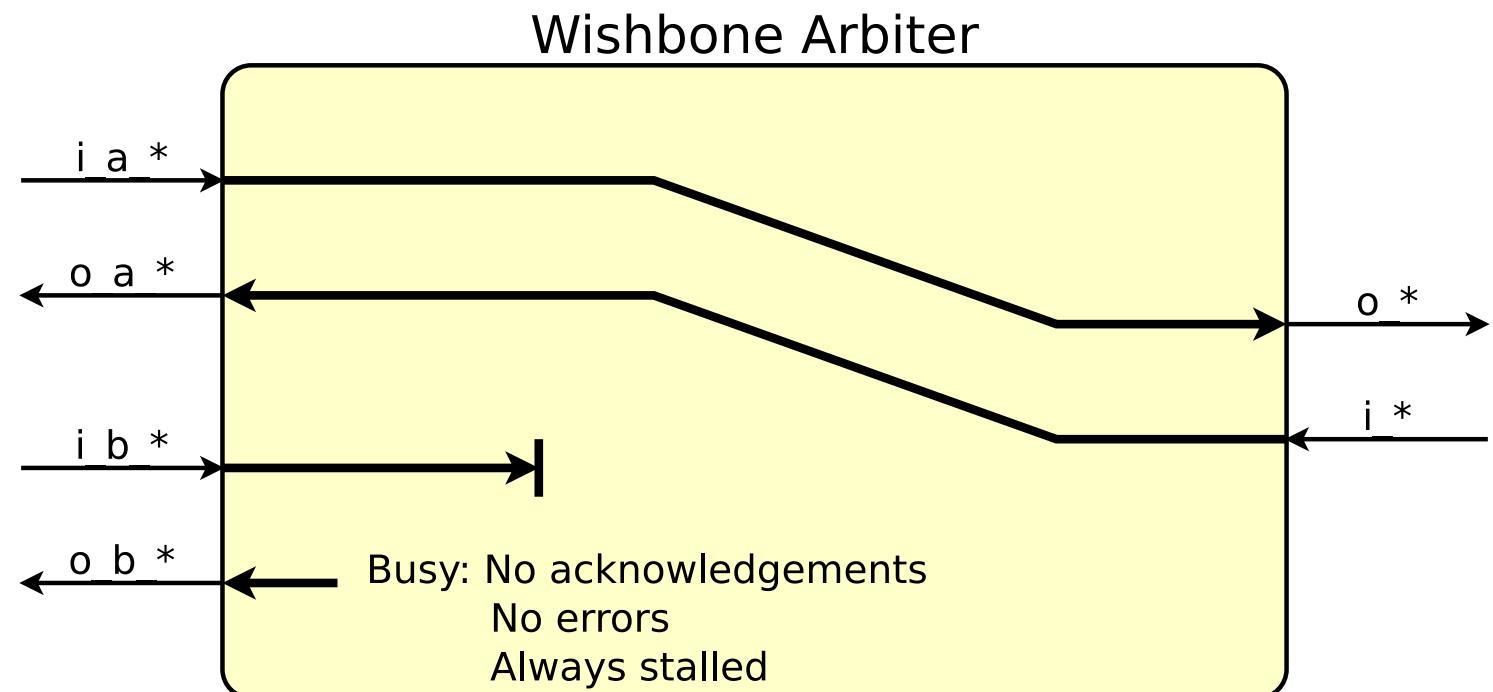
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

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

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

To prove:

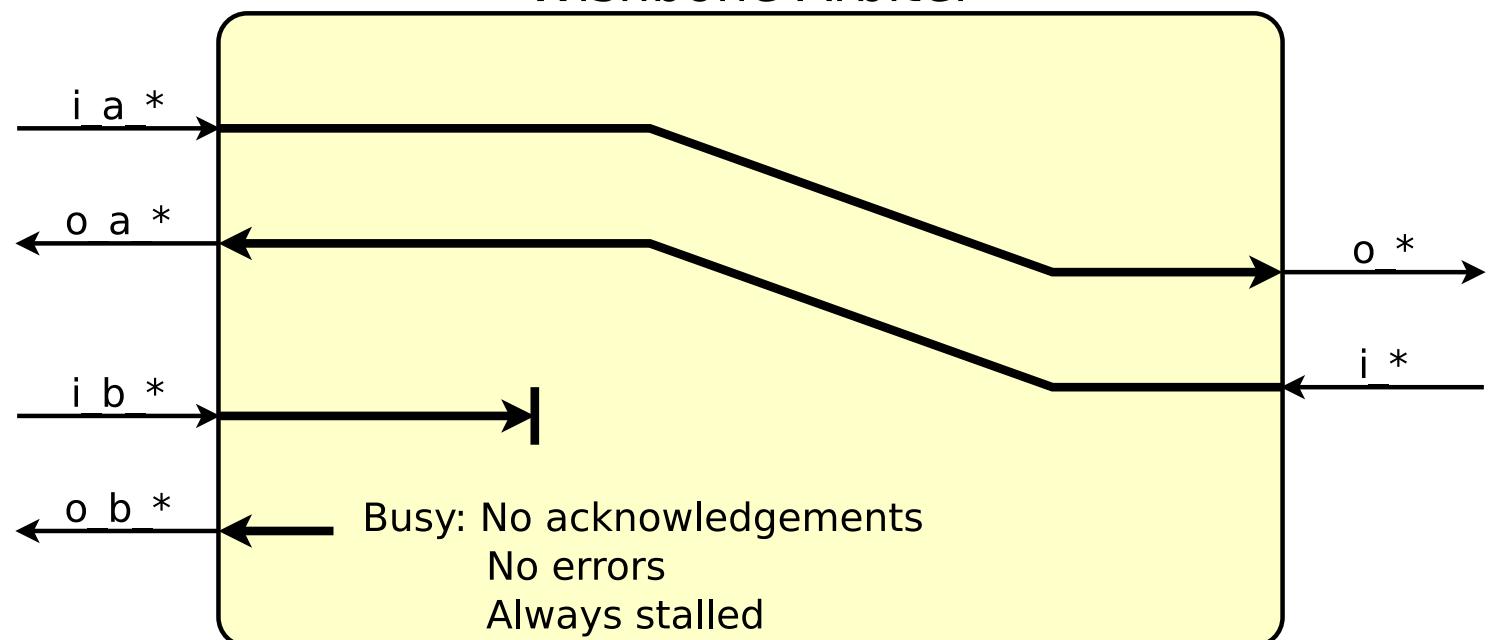
- No data will be lost, no requests will be dropped  
Assume all requests remain stable until accepted
- Only one source ever gets access at a time  
Assert one busy line is always high
- Therefore, all requests go through . . . eventually

[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](#)[Parting Thoughts](#)

Shall we try this with Wishbone?

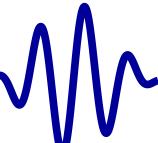
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

## Wishbone Arbiter

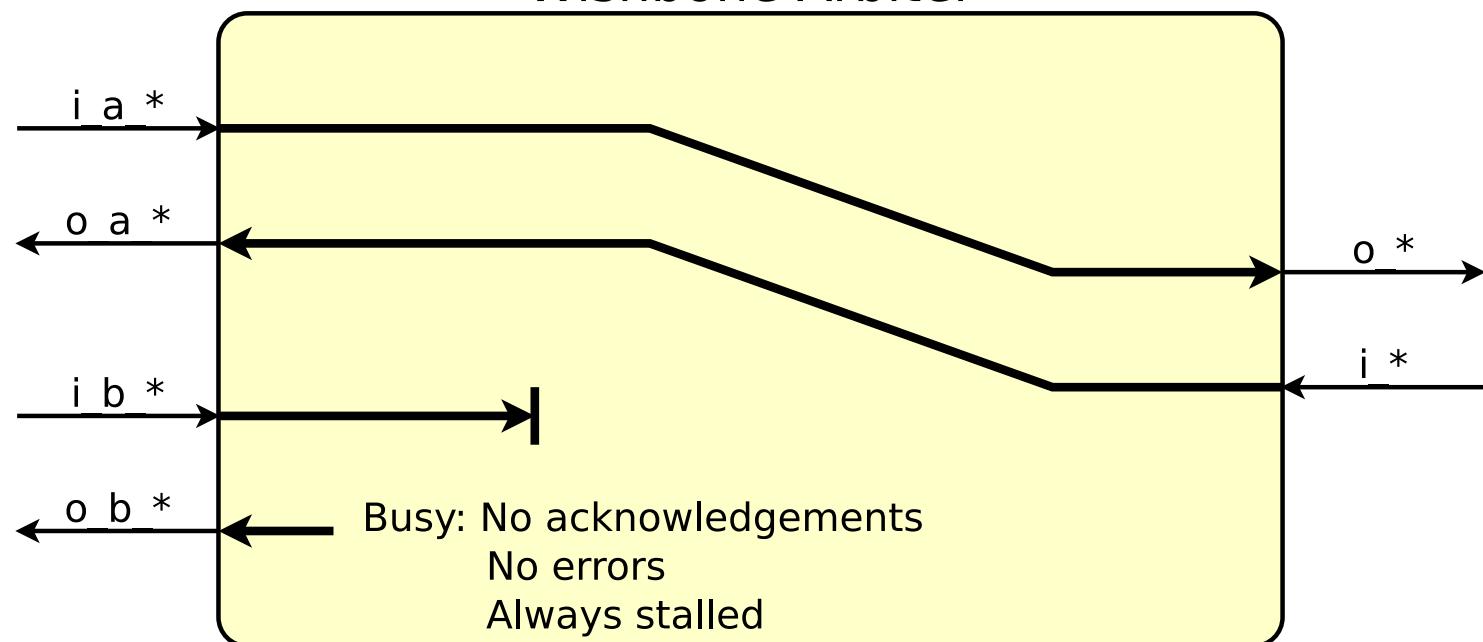


This request side is almost identical

- 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](#)[Parting Thoughts](#)

### Wishbone Arbiter

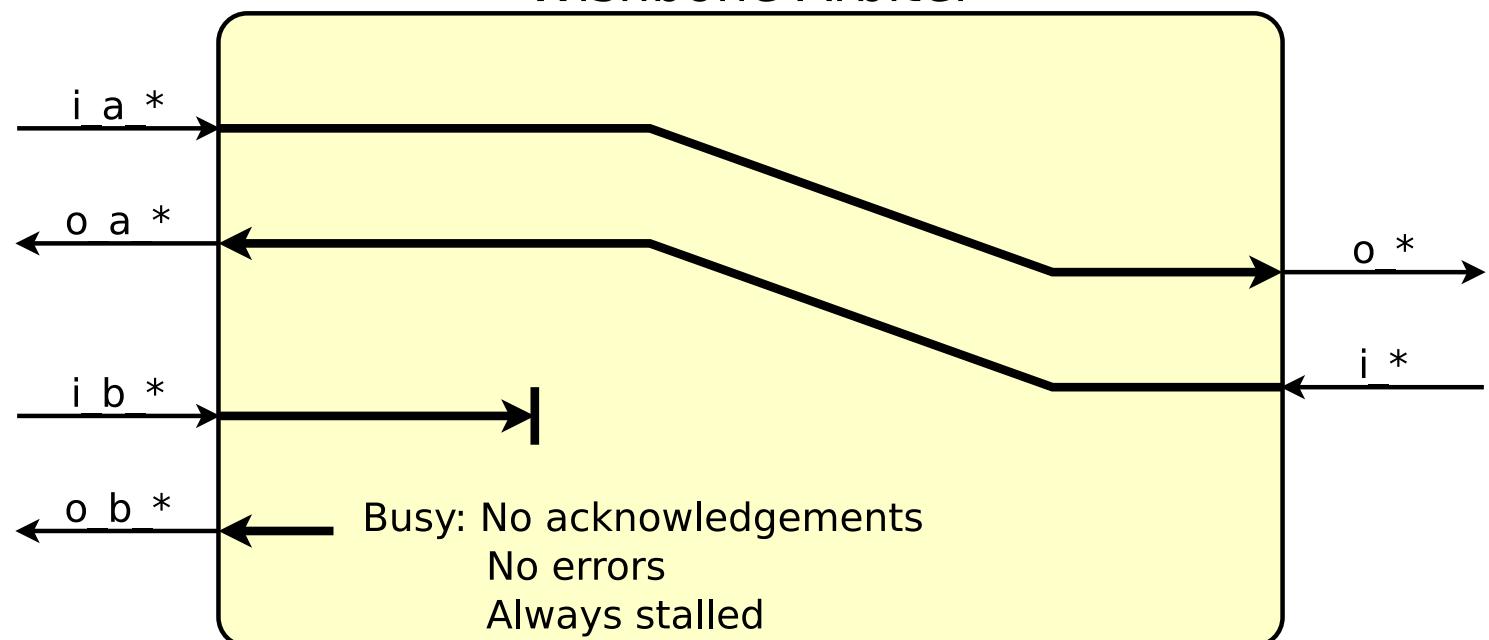


The difference is the acknowledgements

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

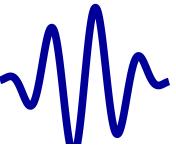
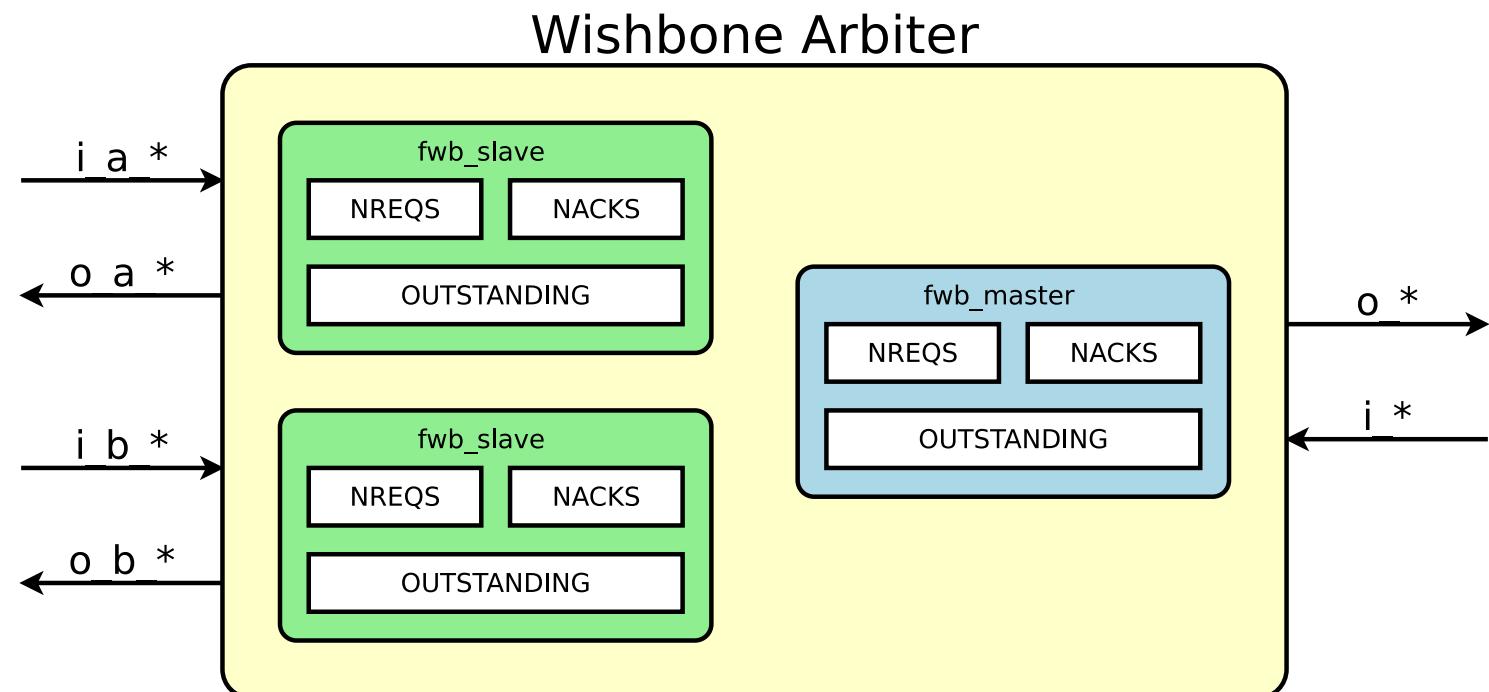
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Ex: WB Bus](#)[AXI](#)[Avalon](#)[Wishbone](#)[WB Basics](#)[▷ WB Basics](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

### Wishbone Arbiter



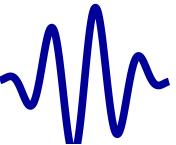
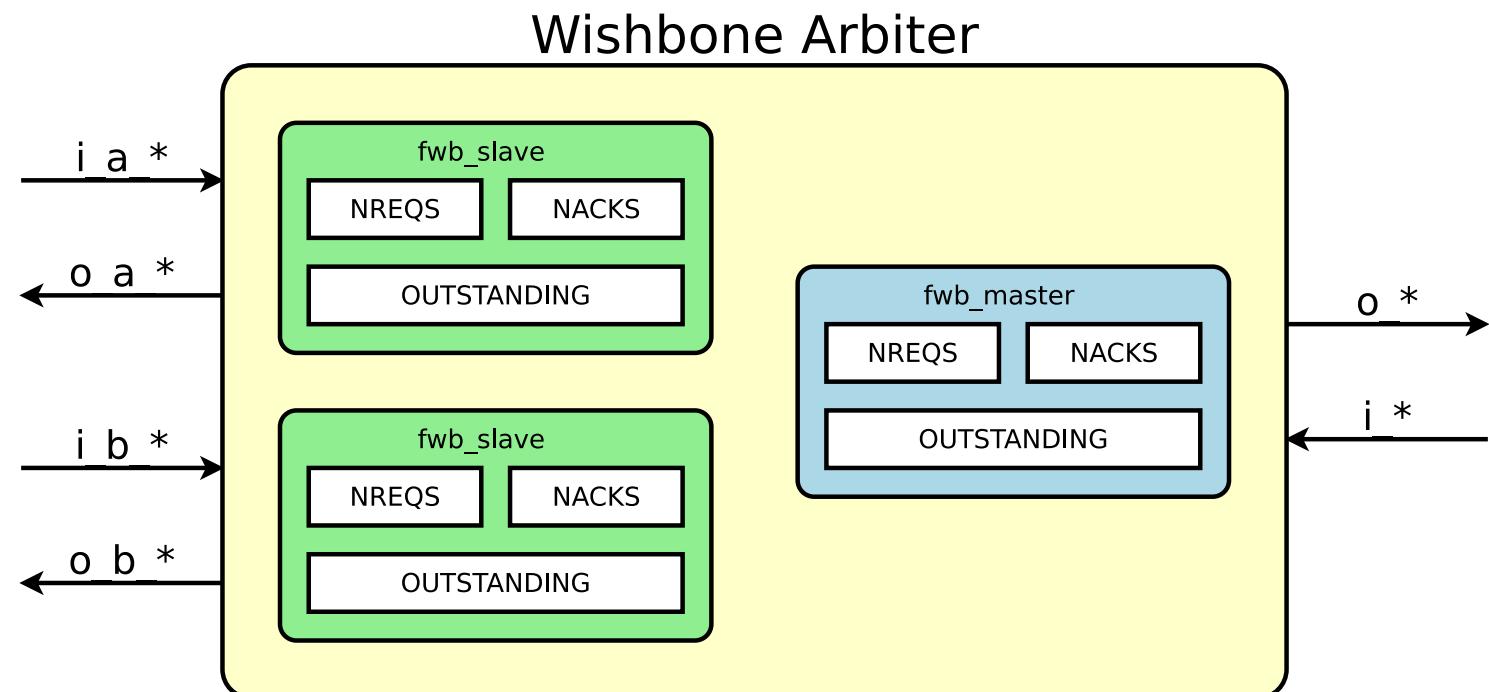
Now, prove that `exercise-07/wbpriarbiter.vhd` works.

- 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](#)[Parting Thoughts](#)

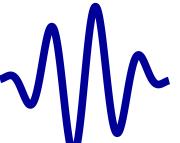
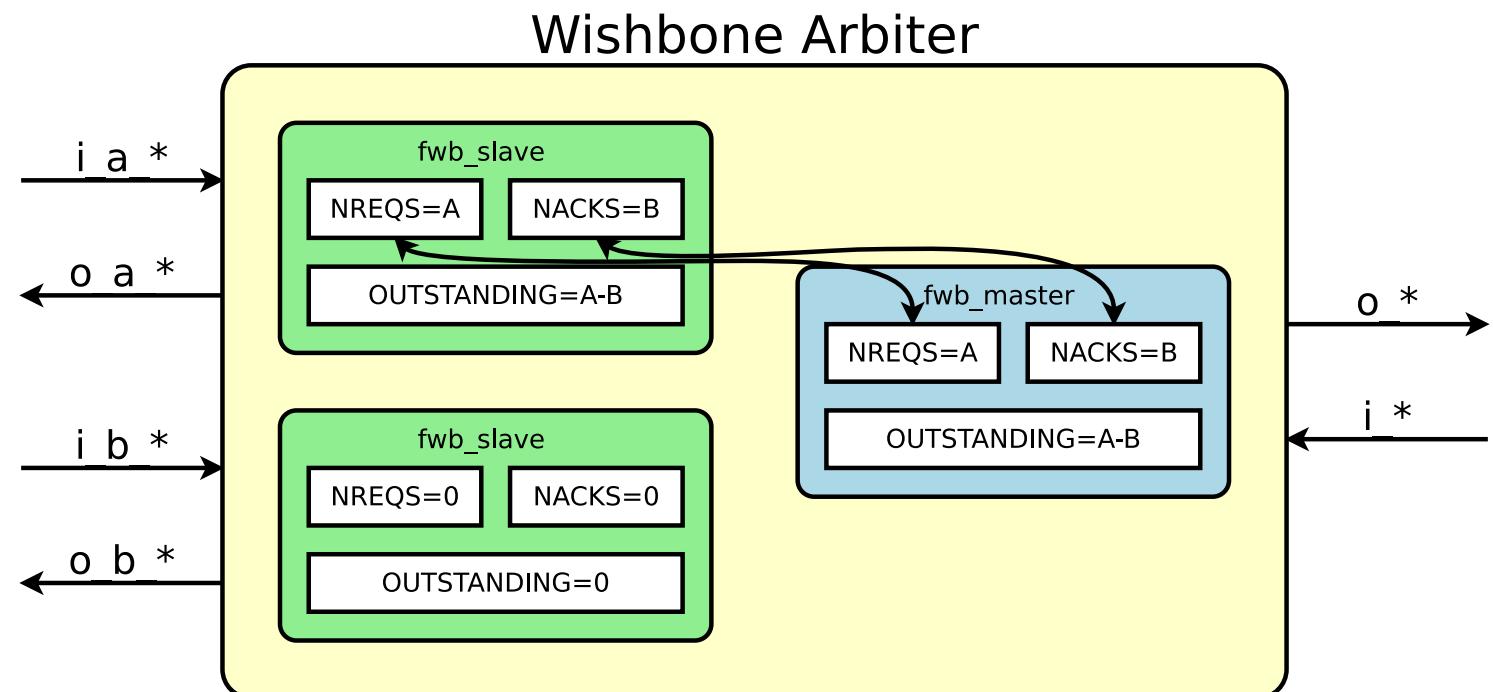
The `fwb_slave.v` properties will

- 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](#)[Parting Thoughts](#)

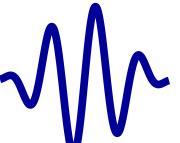
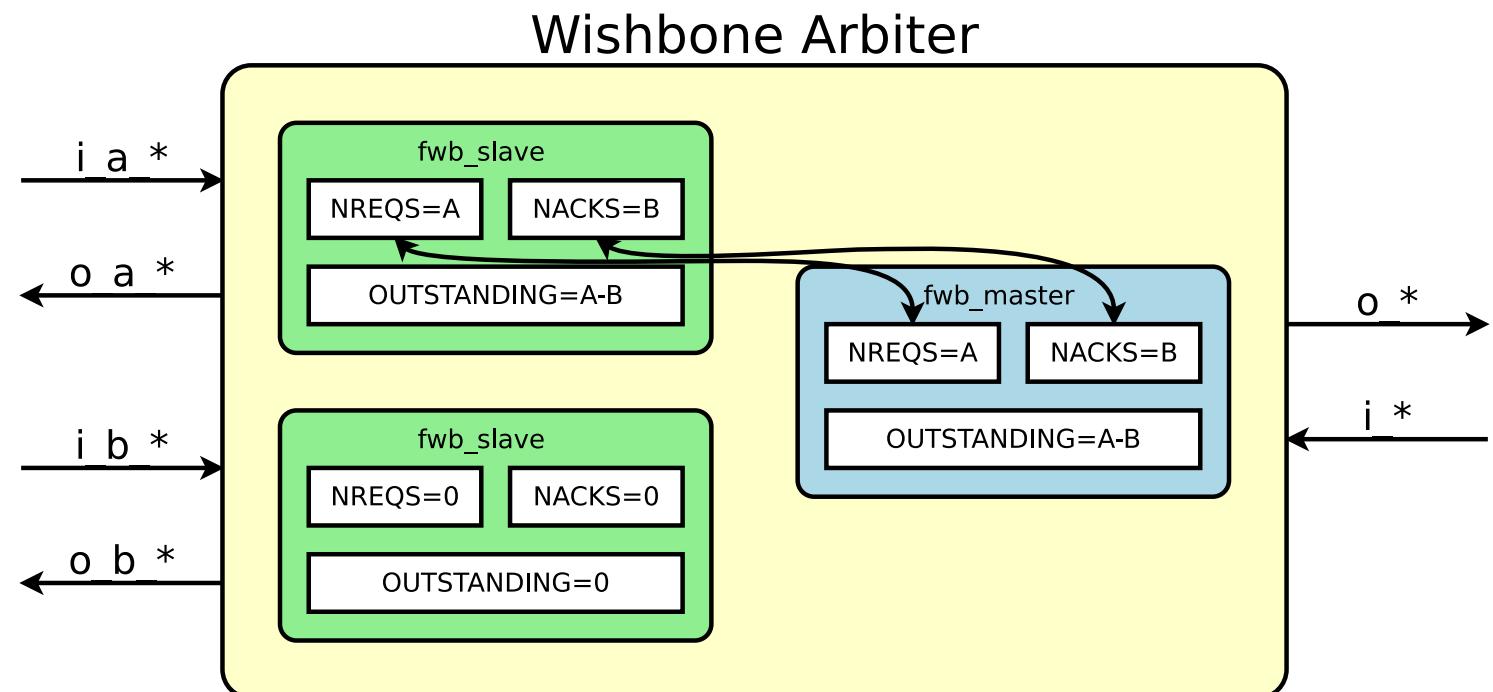
You'll write the `fwb_master.v` properties

- 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](#)[Parting Thoughts](#)

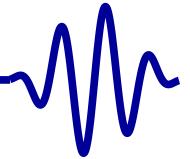
The magic is in how the files are connected

- 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](#)[Parting Thoughts](#)

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

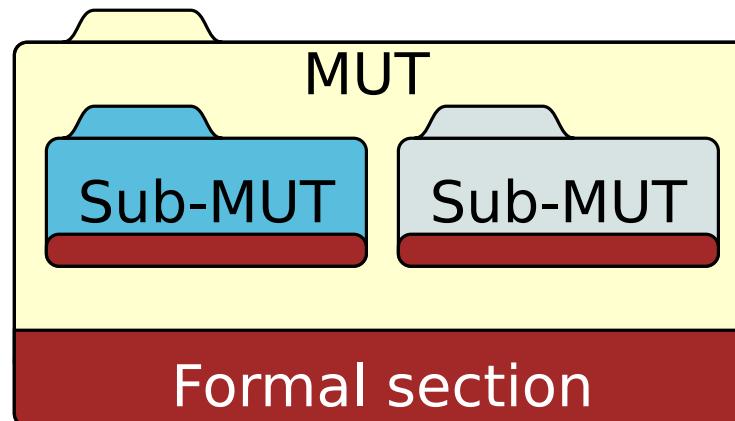
Invariants

Multiple-Clocks

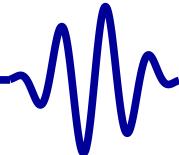
Cover

Sequences

Parting Thoughts



- 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

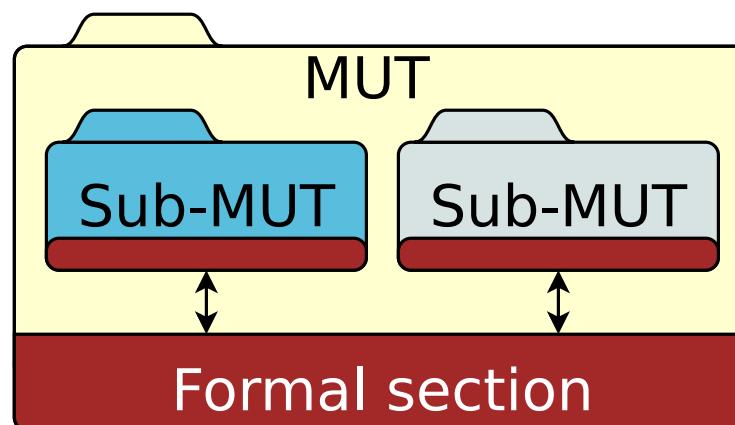
Invariants

Multiple-Clocks

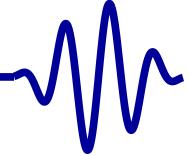
Cover

Sequences

Parting Thoughts



- 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

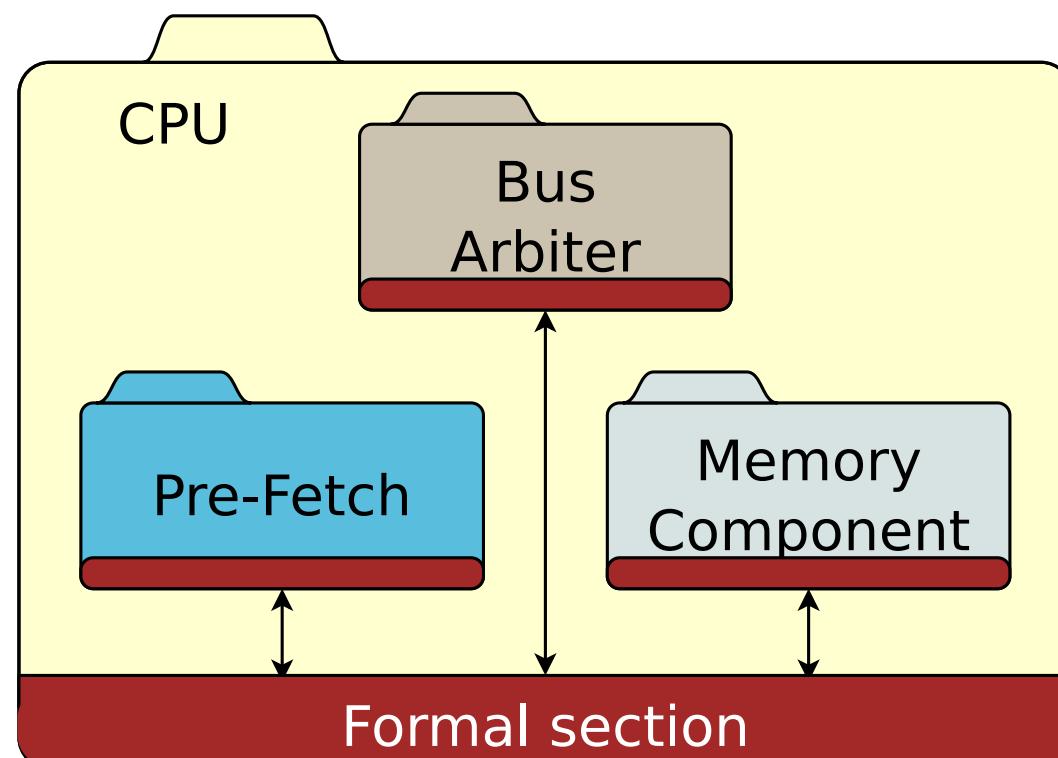
AXI

Avalon

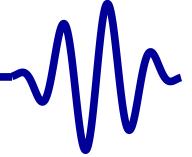
Wishbone

WB Basics

▷ WB Basics

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

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

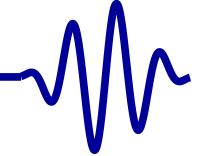


Welcome  
Motivation  
Basics  
Clocked and \$past  
 $k$  Induction  
Bus Properties  
▷ Free Variables  
Lesson Overview  
Formal  
Formal  
Memory  
So what?  
Rule  
Discussion  
Abstraction  
Invariants  
Multiple-Clocks  
Cover  
Sequences  
Parting Thoughts

# Free Variables



# Lesson Overview



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

▷ Lesson Overview

Formal

Formal

Memory

So what?

Rule

Discussion

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Parting Thoughts

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

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

- (\* anyconst \*)

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

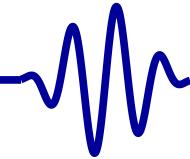
- Can be anything
- Defined at the beginning of time
- Never changed

- (\* 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](#)[▷ Formal](#)[Memory](#)[So what?](#)[Rule](#)[Discussion](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

These properties can be used from within VHDL as well:

- These are VHDL attributes in Yosys
- `anyconst`

```
signal rval : std_logic_vector(N-1 downto 0);

attribute anyconst : bit;
attribute anyconst of rval : signal is '1';
```

- `anyseq`

```
signal sval : std_logic_vector(N-1 downto 0);

attribute anyseq : bit;
attribute anyseq of sval : signal is '1';
```

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

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);
```

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

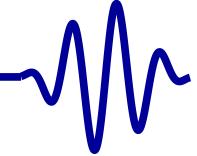
Consider the specification of a prefetch

- The contract

```
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](#)[Formal](#)[Memory](#)[So what?](#)[▷ Rule](#)[Discussion](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

How would our general rule apply here?

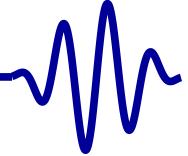
- Assume inputs, assert internal state and outputs
- Both (\* `anyconst` \*) and (\* `anyseq` \*) act like inputs
- You could have written

```
port( ... i_value : in ...; ...);
```

```
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](#)[Formal](#)[Memory](#)[So what?](#)[▷ Rule](#)[Discussion](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

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

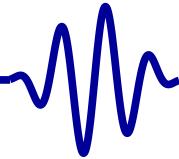
```
attribute anyconst of f_addr: signal is '1';
attribute anyconst of f_valid: signal is '1';
```

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

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](#)[Formal](#)[Memory](#)[So what?](#)[▷ Rule](#)[Discussion](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

You can use this to build a serial port transmitter

```
attribute anyseq of f_tx_start: signal is '1';
attribute anyseq of f_tx_data: signal is '1';
```

```
always @(*)
if (f_tx_busy)
    assume (!f_tx_start);

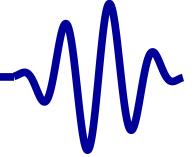
always @ (posedge f_txclk)
if (f_tx_busy)
    assume (f_tx_data == $past(f_tx_data));
```

You can then

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



# Discussion



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

Formal

Memory

So what?

Rule

▷ Discussion

Abstraction

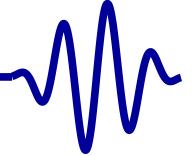
Invariants

Multiple-Clocks

Cover

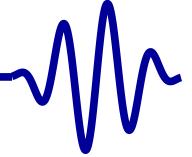
Sequences

Parting Thoughts

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

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

Parting Thoughts

# 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

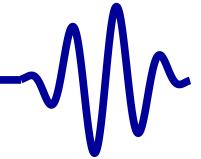
Parting Thoughts

- 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](#)[Parting Thoughts](#)

Formally, if

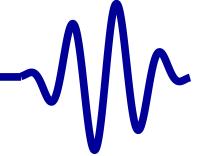
$$A \rightarrow C$$

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

Parting Thoughts

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

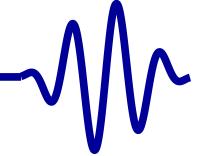
$$\neg A \vee \neg B \vee C$$

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

Expressing as an implication

$$(AB) \rightarrow C$$

Q.E.D.!

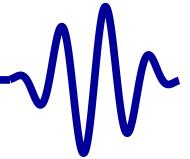
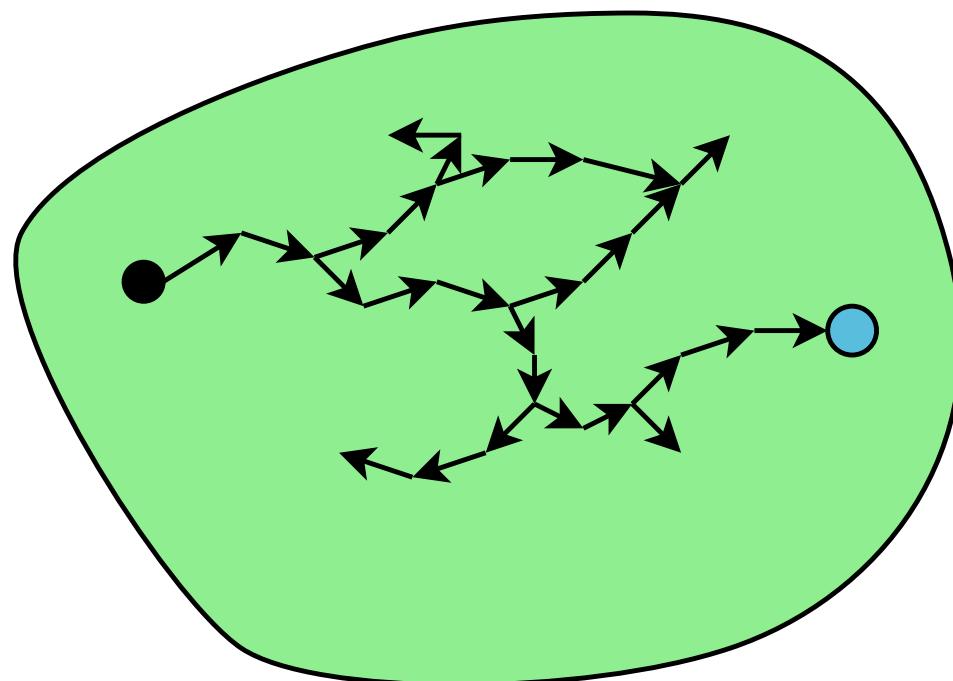
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[▷ Proof](#)[Pictures](#)[Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

With every additional module,

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

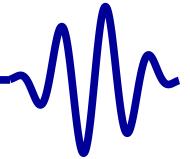
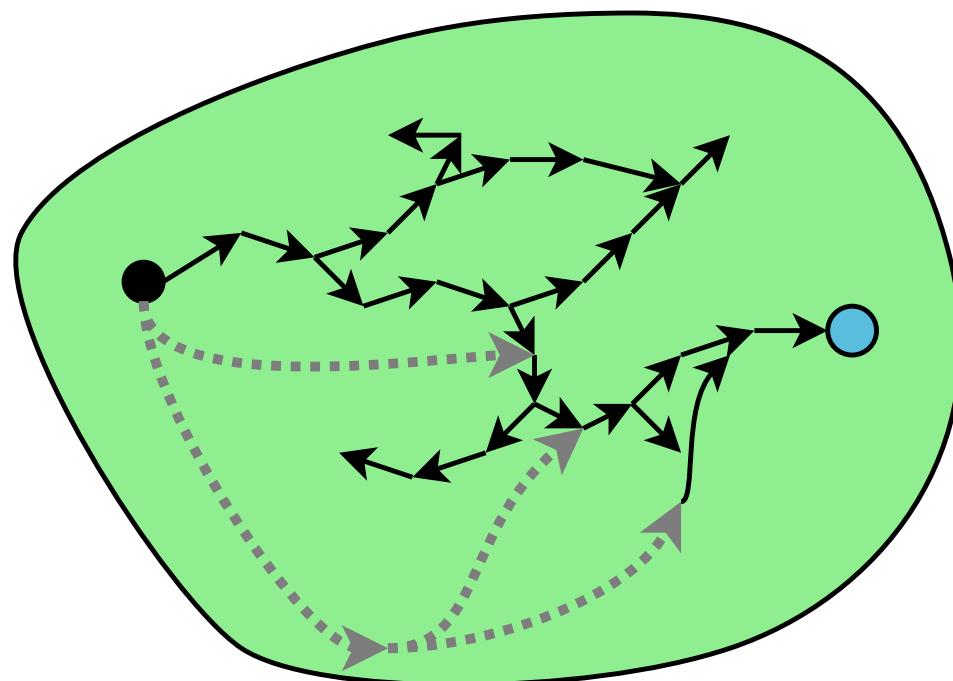
On the other hand,

- 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](#)[Parting Thoughts](#)

Suppose your state space looked like this

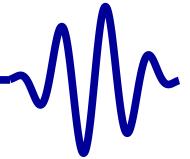
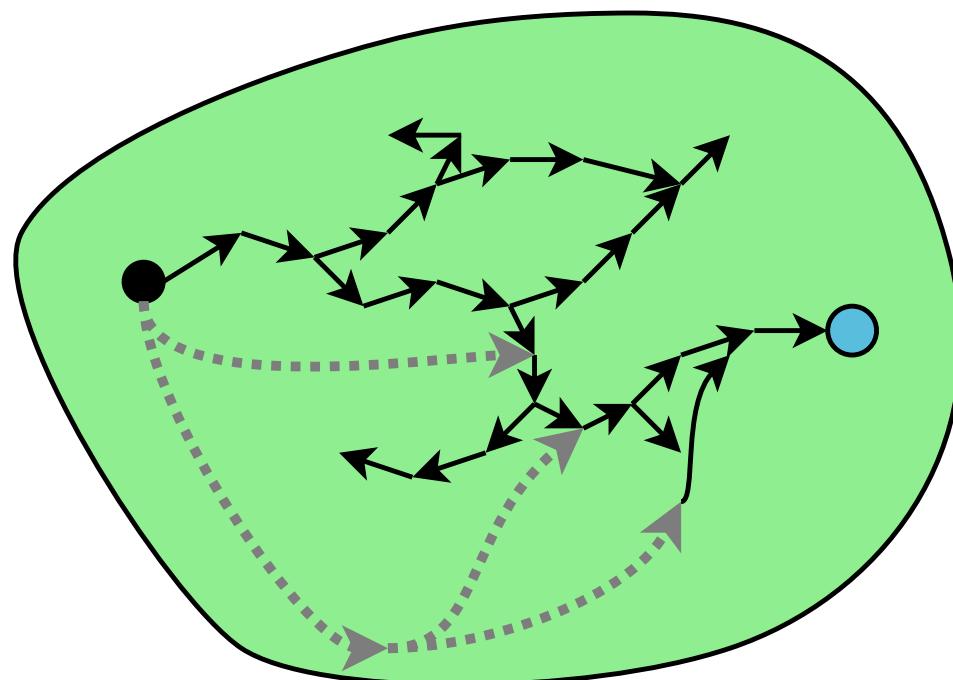
- It takes many transitions required to get to interesting states

[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](#)[Parting Thoughts](#)

Suppose we added to this design ...

- Some additional states, and
- Additional transitions

The *real* states and transitions must still remain

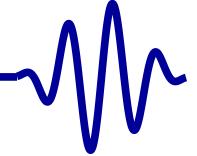
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[▷ Pictures](#)[Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

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

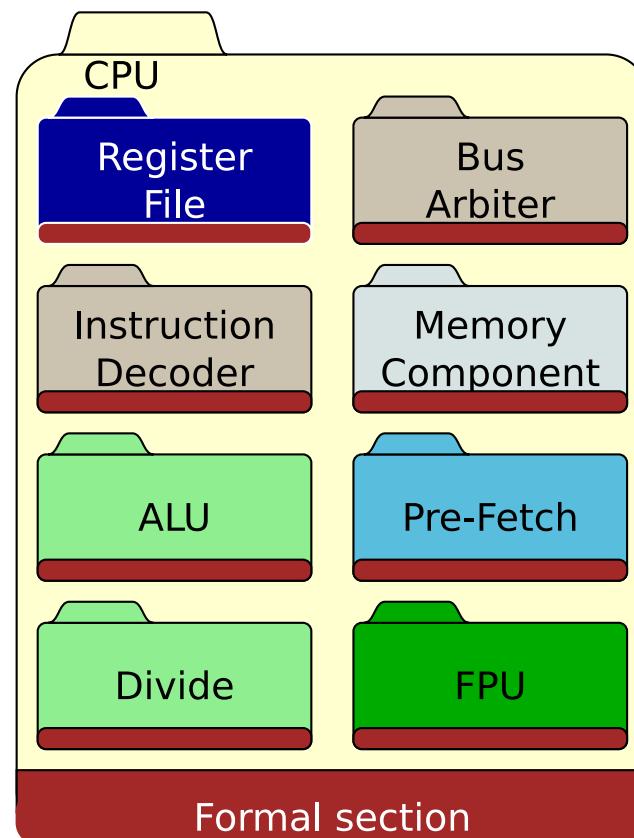
Multiple-Clocks

Cover

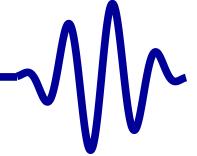
Sequences

Parting Thoughts

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

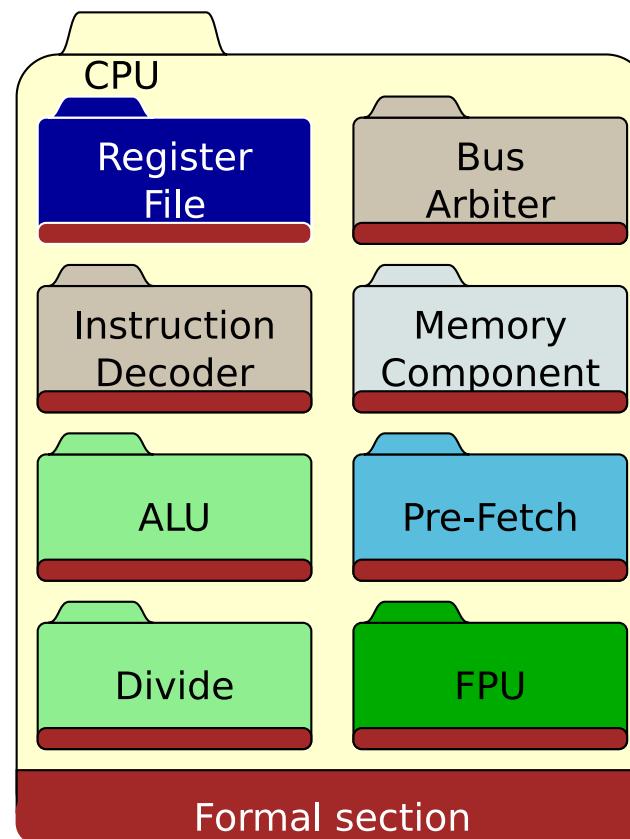
Multiple-Clocks

Cover

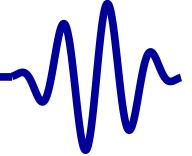
Sequences

Parting Thoughts

Where would you start?



At the interfaces!



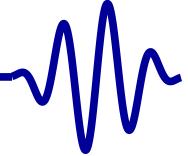
Let's consider a prefetch module as an example.



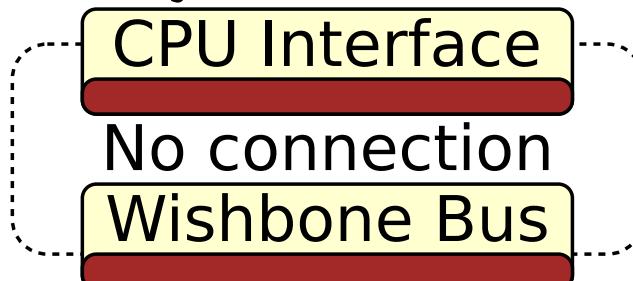
If you do this right,

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

Care to try a different prefetch approach?

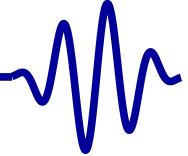
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[▷ Pictures](#)[Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

Suppose the prefetch was just a shell

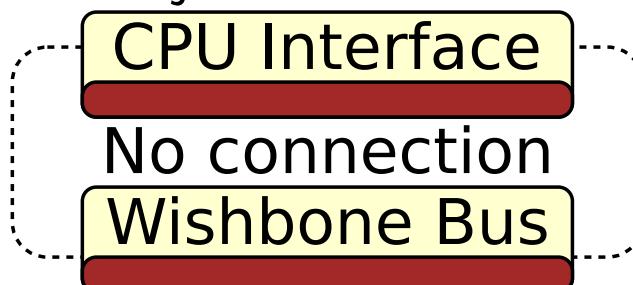


It would still interact properly with

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

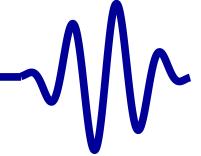
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[▷ Pictures](#)[Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

Suppose the prefetch was just a shell



If the CPU still acted “correctly”

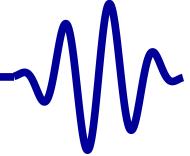
- With either the right, or the wrong instructions, then
- The CPU *must act correctly with the right instructions*

[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](#)[Parting Thoughts](#)

Consider these statements:

□

If  
And  
Then

[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](#)[Parting Thoughts](#)

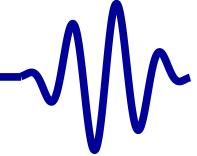
Consider these statements:

- 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)

[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](#)[Parting Thoughts](#)

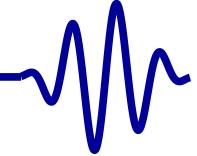
Consider these statements:

- 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](#)[Parting Thoughts](#)

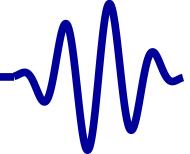
Consider these statements:

- 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](#)[Parting Thoughts](#)

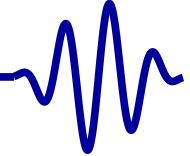
Consider these statements:

- 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](#)[Parting Thoughts](#)

Consider these statements:

- The divide must return a calculated number

If (Divide returns a value when requested)

And (It is the right value)

Then (The divide works within the design)

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

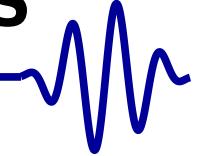
Consider these statements:

- Formal solvers break down when applied to multiplies

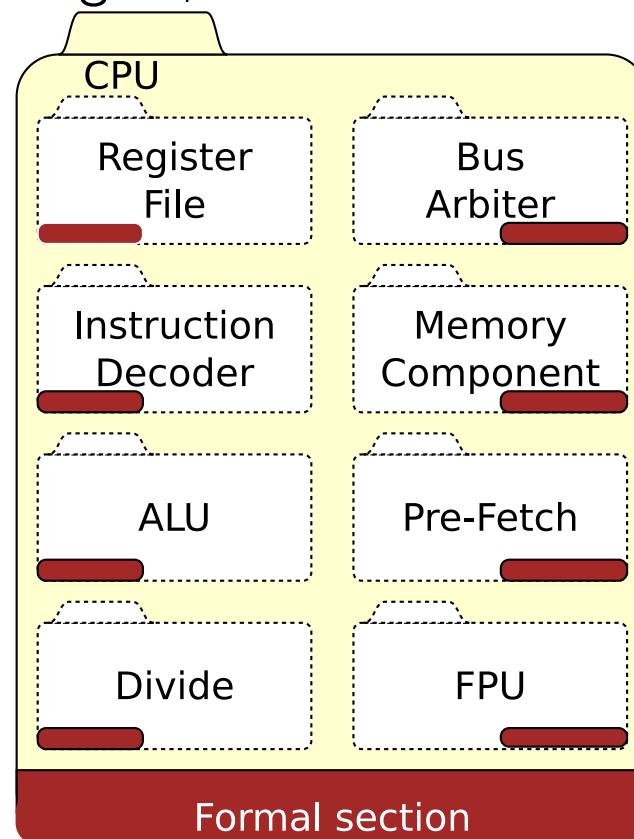
If (Multiply unit returns an answer  $N$  clocks later)

And (It is the right value)

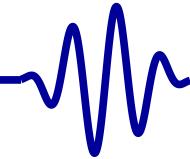
Then (The multiply works within the design)

[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](#)[Parting Thoughts](#)

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](#)[Parting Thoughts](#)

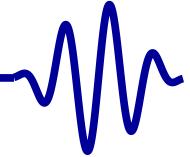
Let's consider a fractional counter:

```
signal r_count : unsigned(31 downto 0) := 0;  
  
process(i_clk)  
begin  
    if (rising_edge(i_clk)) then  
        (o_pps, r_count) <= resize(r_count, 33) + 43;  
    end if;  
end process;
```

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](#)[Parting Thoughts](#)

How might we build an abstract counter?

- First, create an arbitrary counter increment

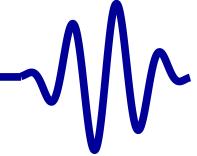
```
signal increment : unsigned(31 downto 0)
              := to_unsigned(1,32);
attribute anyseq of increment : signal is '1';

process(i_clk)
begin
  if (rising_edge(i_clk)) then
    (o_pps, r_count) <= resize(r_count, 33)
                           + increment;

  end if;
end process;

rollover <= -r_count;
```

We'll constrain this increment next

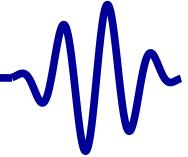
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[Pictures](#)[▷ Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

How might we build an abstract counter?

- First, create an arbitrary counter increment
- Then constrain the arbitrary increment

```
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, 43, must be a possibility

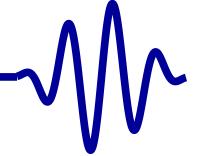
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[Pictures](#)[▷ Examples](#)[Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

Will this work?

- Let's try this to see!

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

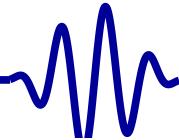
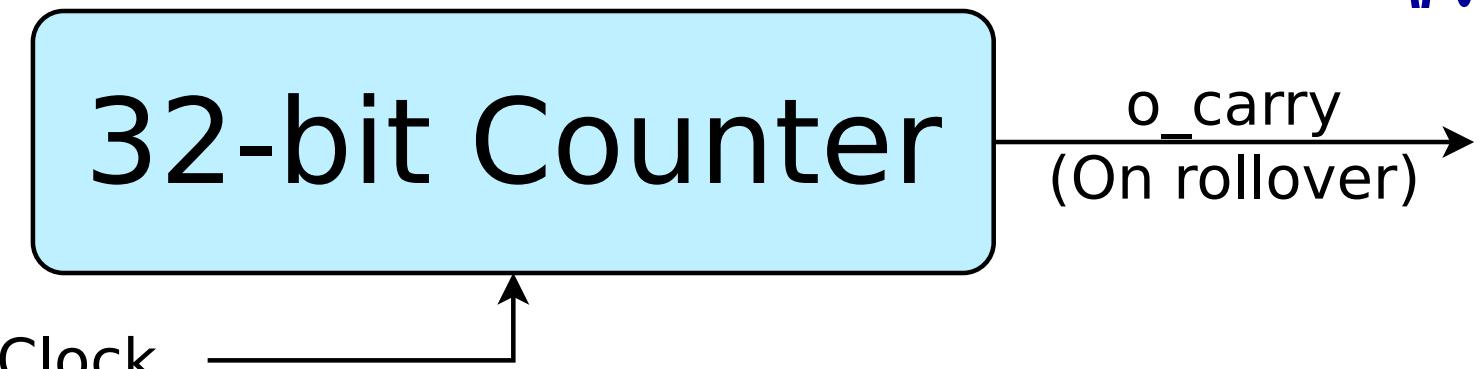
always @(posedge i_clk)
  if ((f_past_valid)&&(r_count < $past(r_count)))
    assert(o_pps);
```

[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](#)[Parting Thoughts](#)

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?

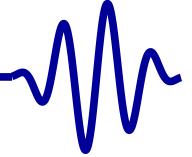
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[Pictures](#)[Examples](#)[▷ Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

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](#)[Parting Thoughts](#)

## Your task:

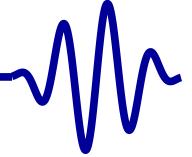
- Rebuild the counter
- Make it increment by one
- Build it so that ...

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

// and

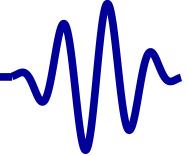
```
always @(posedge i_clk)  
    if ((f_past_valid)&&(!$past(&r_count)))  
        assert(!o_carry);
```

- 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](#)[Parting Thoughts](#)

## Your task:

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

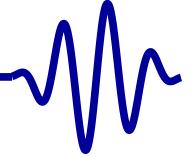
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Lesson Overview](#)[Formal](#)[Proof](#)[Pictures](#)[Examples](#)[▷ Exercise](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

### 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

Parting Thoughts

# Invariants



# Lesson Removed



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

▷ Lesson Removed

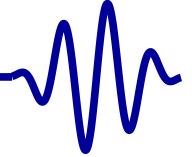
Multiple-Clocks

Cover

Sequences

Parting Thoughts

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



---

Welcome

---

Motivation

---

Basics

---

Clocked and \$past

---

$k$  Induction

---

Bus Properties

---

Free Variables

---

Abstraction

---

Invariants

---

▷ Multiple-Clocks

Basics

SBY File

(\* gclk \*)

\$rose

\$stable

Examples

Exercises

---

Cover

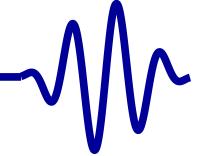
---

Sequences

---

Parting Thoughts

# Multiple-Clocks

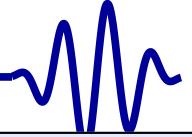
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[▷ Basics](#)[SBY File](#)[\(\\* gclk \\*\)](#)[\\$rose](#)[\\$stable](#)[Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

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](#)[Parting Thoughts](#)

```
[options]
mode prove
multiclock on

[engines]
smtbmc

[script]
read -vhdl module.vhd
read -formal module_vhd.sv
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  
Parting Thoughts

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

[engines]
smtbmc

[script]
read -vhdl module.vhd
read -formal module_vhd.sv
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

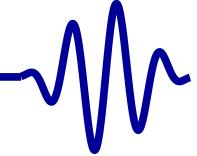
Cover

Sequences

Parting Thoughts

- (\* 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

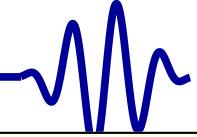
Examples

Exercises

Cover

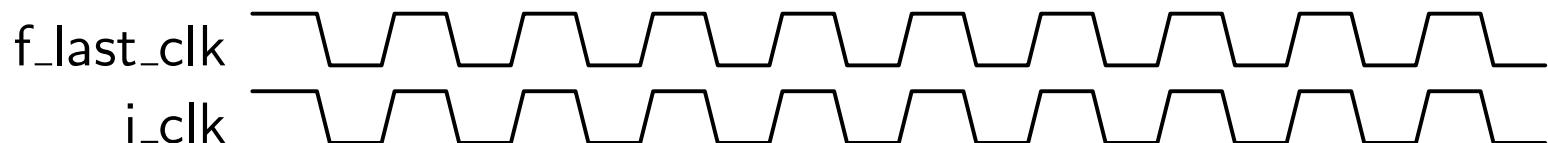
Sequences

Parting Thoughts

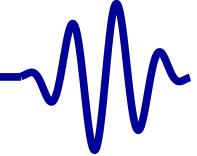


- Welcome
- Motivation
- Basics
- Clocked and \$past
- $k$  Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Basics
- SBY File
- ▷ (\* gclk \*)
- \$rose
- \$stable
- Examples
- Exercises
- Cover
- Sequences
- Parting Thoughts

```
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

Parting Thoughts

[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](#)[Parting Thoughts](#)

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

sync

f\_clk\_counter



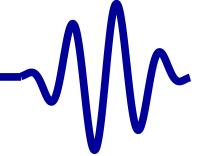
i\_clk\_fast



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

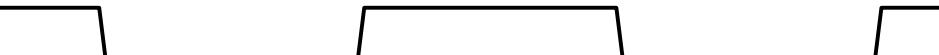
Parting Thoughts

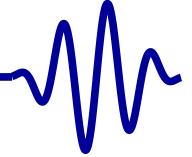
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[▷ \(\\* gclk \\*\)](#)[\\$rose](#)[\\$stable](#)[Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

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

$\triangleright$  (\* gclk \*)

\$rose

\$stable

Examples

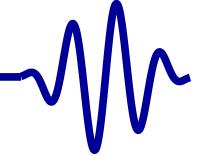
Exercises

Cover

Sequences

Parting Thoughts

# (\* gclk \*)



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

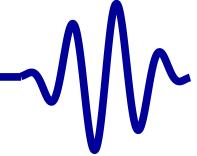
Examples

Exercises

Cover

Sequences

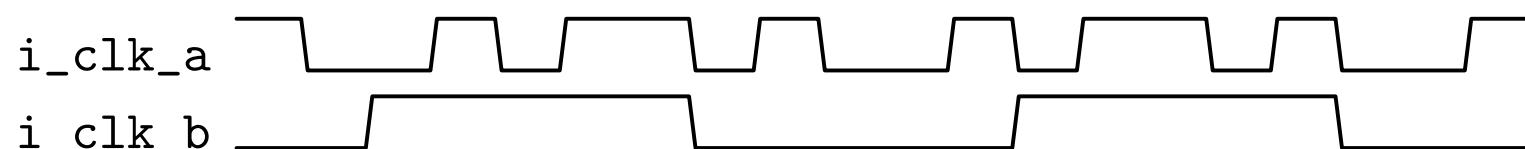
Parting Thoughts

[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](#)[Parting Thoughts](#)

The timing relationship between these two clocks can be anything

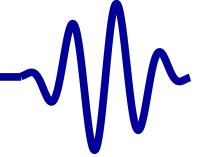
- Each clock can have an arbitrary frequency
- Each clock can have an arbitrary phase

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

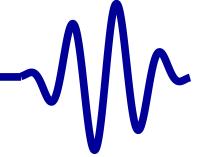
Parting Thoughts

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

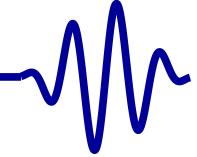
Examples

Exercises

Cover

Sequences

Parting Thoughts



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

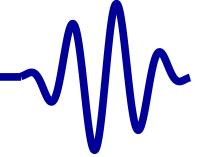
Examples

Exercises

Cover

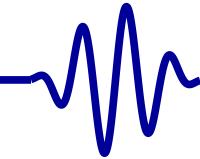
Sequences

Parting Thoughts

[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](#)[Parting Thoughts](#)

Could we do it this way?

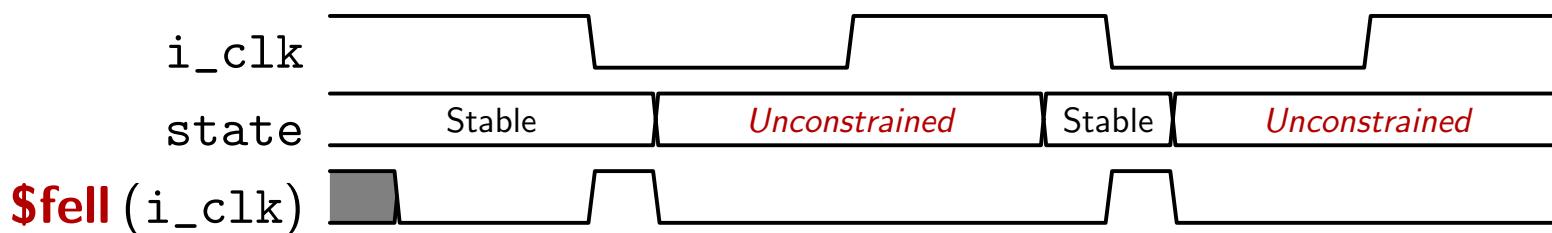
```
always @(posedge gbl_clk)
if ($fell(i_clk))
    assert(state == $past(state));
```



Could we do it this way?

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

- No, this doesn't work either



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Basics

SBY File

(\* gclk \*)

▷ \$rose

\$stable

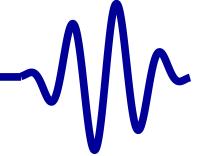
Examples

Exercises

Cover

Sequences

Parting Thoughts



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

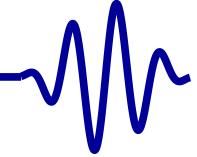
Examples

Exercises

Cover

Sequences

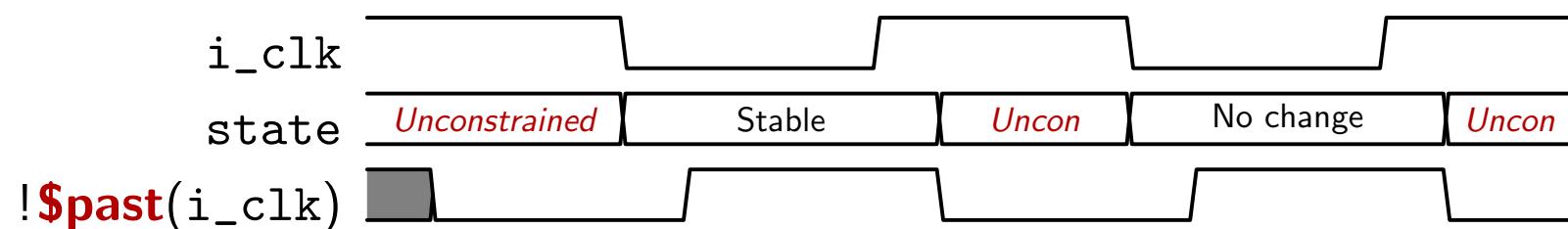
Parting Thoughts

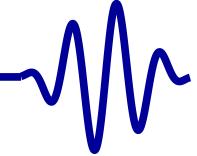
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\(\\* gclk \\*\)](#)[▷ \\$rose](#)[\\$stable](#)[Examples](#)[Exercises](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

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

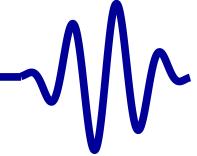
Examples

Exercises

Cover

Sequences

Parting Thoughts



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

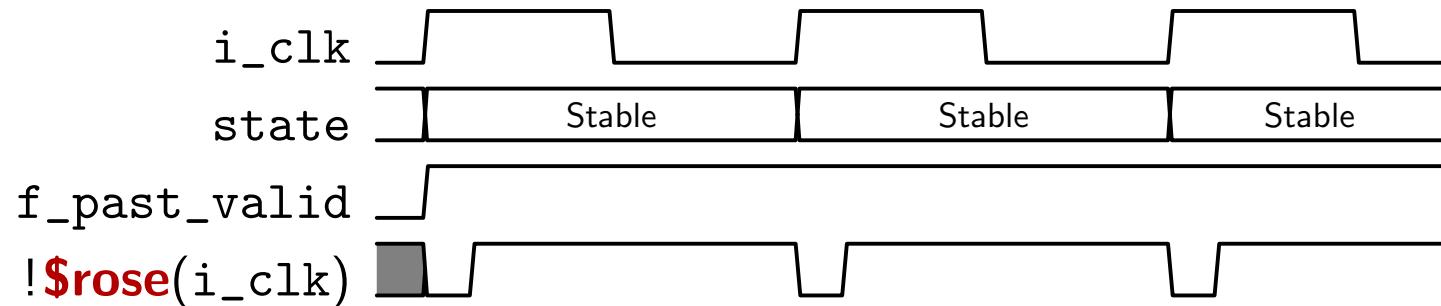
Parting Thoughts

[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](#)[Parting Thoughts](#)

- State/outputs should be clock synchronous

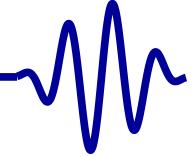
```
always @ (posedge gbl_clk)
  if ((f_past_valid)&&(!$rose(i_clk))
      assert(state == $past(state));
```

- With f\_past\_valid this works



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

# \$stable



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

Parting Thoughts

Describes a signal which has not changed

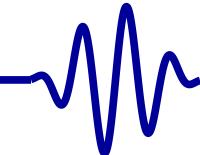
```
always @(posedge gbl_clk)
if ((f_past_valid)&&(!$rose(i_clk)))
    assert($stable(state));
```

- Requires a clock edge

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

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

# \$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

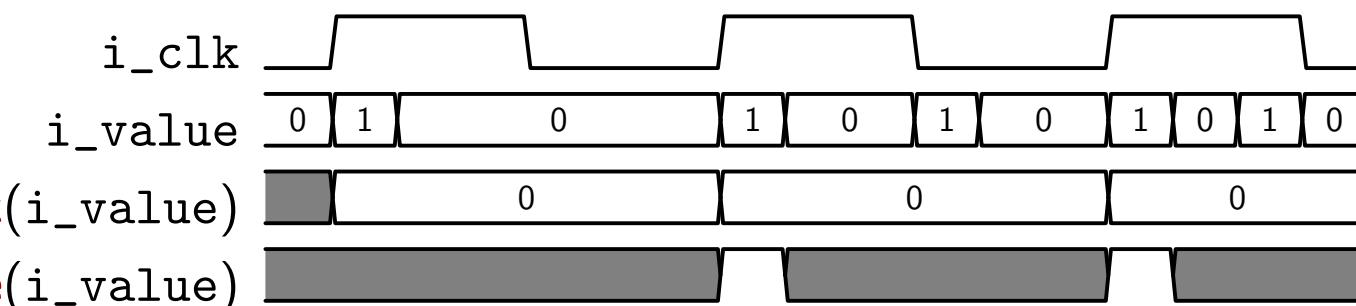
Sequences

Parting Thoughts

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

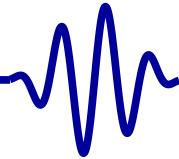
```
always @(posedge i_clk)
    assume($stable(i_value));
```

The waveform below would satisfy the assumption above

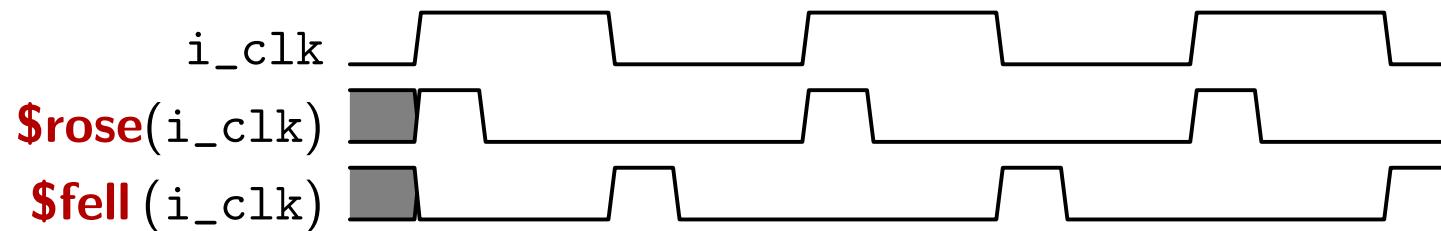


The key to understanding what's going on is to realize . . .

- 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](#)[Parting Thoughts](#)

**\$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

Parting Thoughts

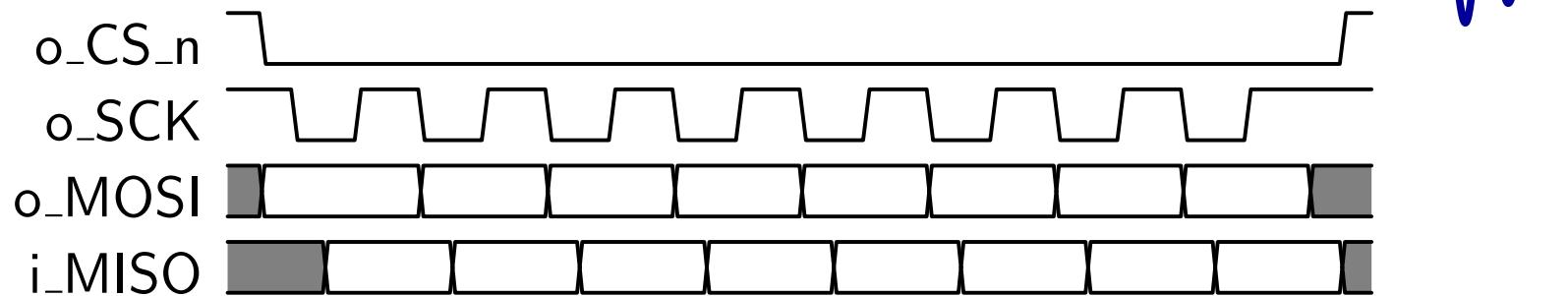
- Most logic doesn't need the multiclock option
- To help with logic that might need it, I use a parameter

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

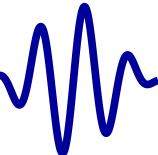
generate if (F_OPT_CLK2FFLOGIC)
begin
    (* gclk *) wire gbl_clk;

    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
```

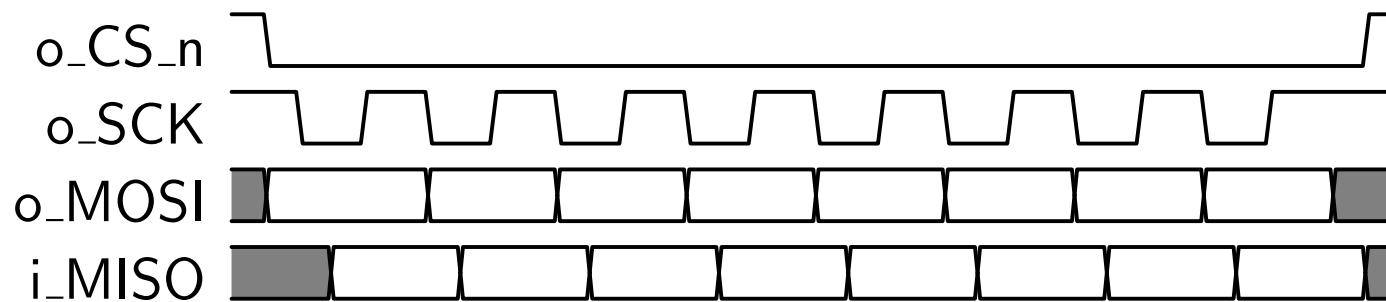
Welcome
Motivation
Basics
Clocked and \$past
$k$ Induction
Bus Properties
Free Variables
Abstraction
Invariants
Multiple-Clocks
Basics
SBY File
(* gclk *)
\$rose
\$stable
▷ Examples
Exercises
Cover
Sequences
Parting Thoughts



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



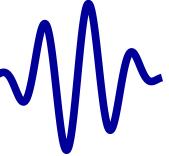
- Welcome
- Motivation
- Basics
- Clocked and \$past
- $k$  Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Basics
- SBY File
- (\* gclk \*)
- \$rose
- \$stable
- ▷ Examples
- Exercises
- Cover
- Sequences
- Parting Thoughts



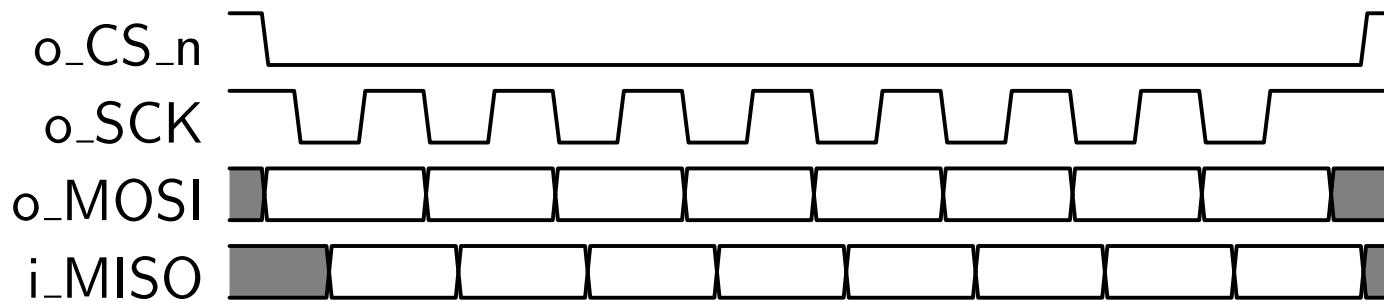
- How would you formally describe the o\_SCK and o\_CS\_n relationship?

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

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



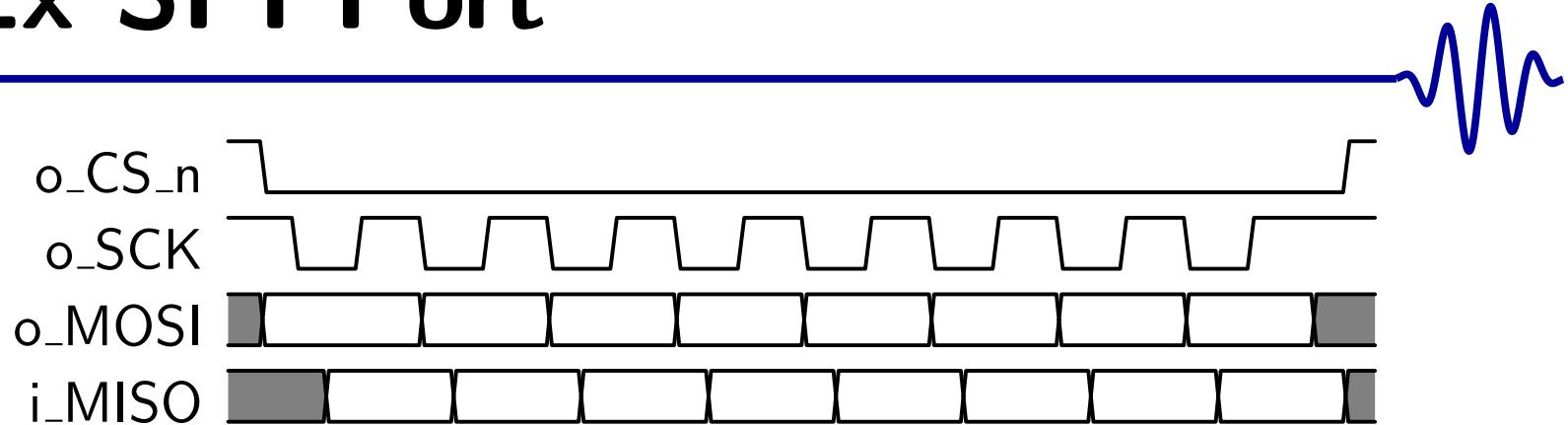
- Welcome
- Motivation
- Basics
- Clocked and \$past
- $k$  Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Basics
- SBY File  
(\* gclk \*)
- \$rose
- \$stable
- ▷ Examples
- Exercises
- Cover
- Sequences
- Parting Thoughts



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

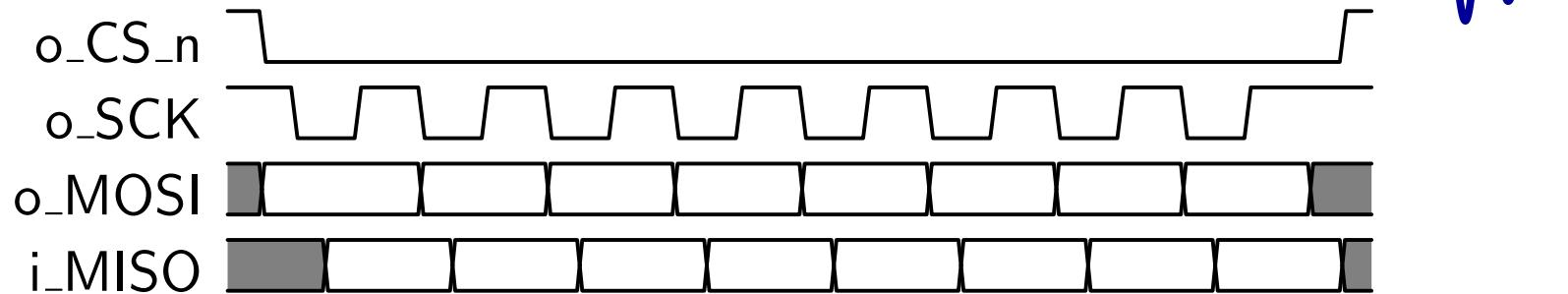
```
always @(`posedge` gbl_clk)
if ((f_past_valid)
    &&(`$rose(o_CS_n)) || (`$fell(o_CS_n)))))
    assert ((o_SCK)&&(`$stable(o_SCK)));
```

- 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
- Parting Thoughts



- How would you describe `o_MOSI`?

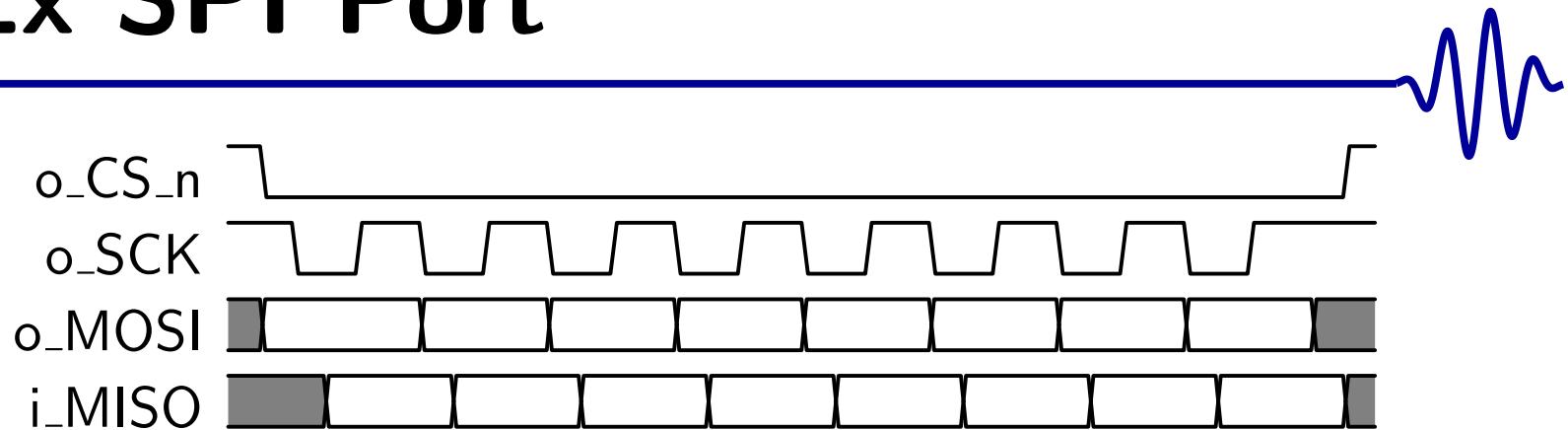
- Welcome
- Motivation
- Basics
- Clocked and \$past
- $k$  Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Basics
- SBY File
- (\* gclk \*)
- \$rose
- \$stable
- ▷ Examples
- Exercises
- Cover
- Sequences
- Parting Thoughts



- How would you describe o\_MOSI?

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

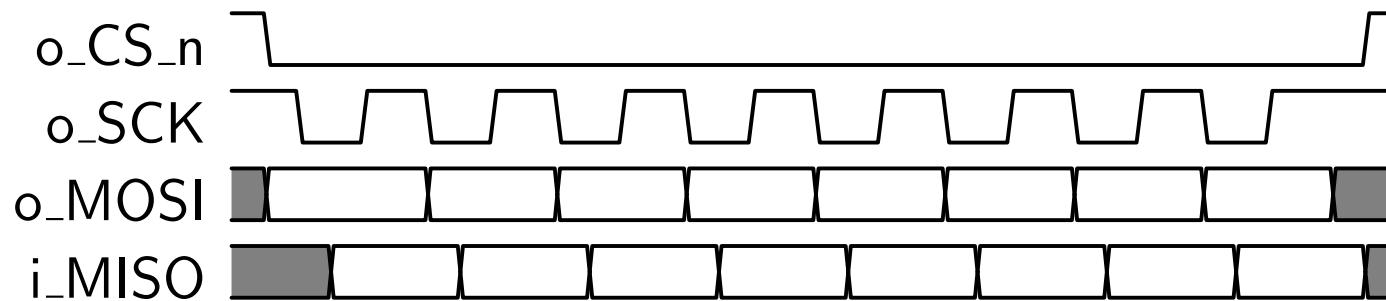
- Welcome
- Motivation
- Basics
- Clocked and \$past
- $k$  Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Basics
- SBY File
- (\* gclk \*)
- \$rose
- \$stable
- ▷ Examples
- Exercises
- Cover
- Sequences
- Parting Thoughts



- How would you describe i\_MISO?



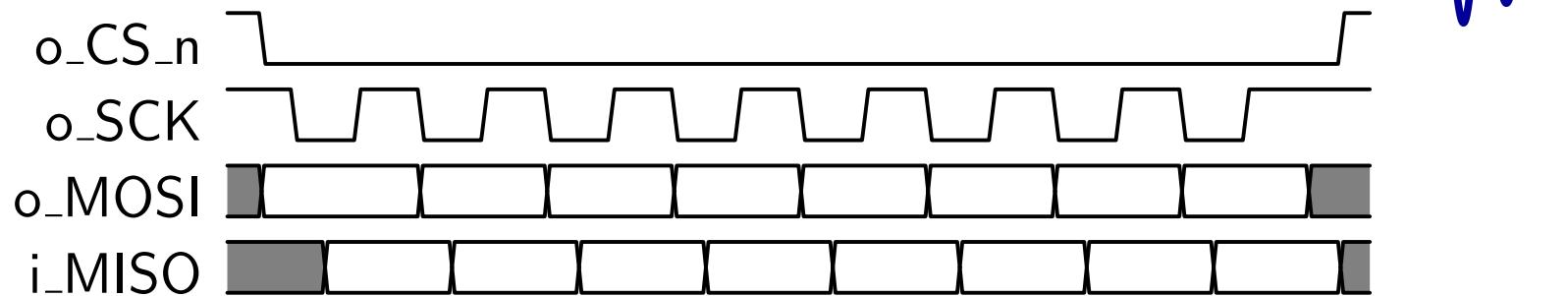
- Welcome
- Motivation
- Basics
- Clocked and \$past
- $k$  Induction
- Bus Properties
- Free Variables
- Abstraction
- Invariants
- Multiple-Clocks
- Basics
- SBY File
- (\* gclk \*)
- \$rose
- \$stable
- ▷ Examples
- Exercises
- Cover
- Sequences
- Parting Thoughts



- 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
Parting Thoughts



- 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](#)[Parting Thoughts](#)

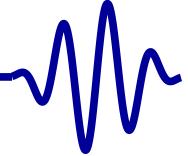
- A little logic will force `i_MISO` to have only one transition per clock

```
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?

# Ex SPI Port



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

Parting Thoughts

- Forcing exactly 8 clocks

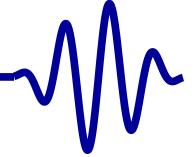
```
always @ (posedge gbl_clk)
  if (o_CS_n)
    f_spi_bits <= 0;
  else if ($rose(o_SCK))
    f_spi_bits <= f_spi_bits + 1'b1;
```

```
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

Parting Thoughts

Three exercises, chose one to verify:

1. Input serdes

`exercises-09/iserdes.vhd`

2. Clock gate

`exercises-10/clkgate.vhd`

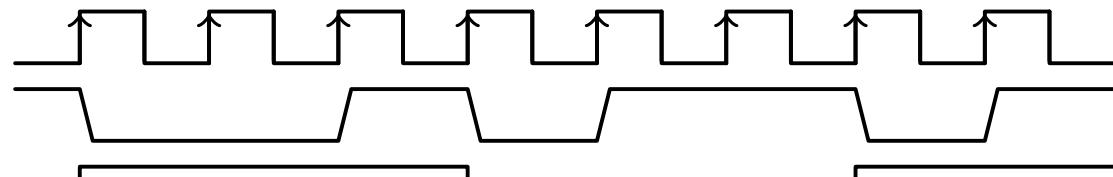
3. Clock Switch

`exercises-11/clkswitch.vhd`

[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](#)[Parting Thoughts](#)

Getting a SERDES right is a good example of multiple clocks

i\_fast\_clk



i\_pin

i\_slow\_clk

o\_word

0x0b

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

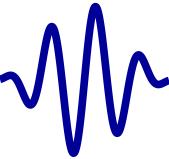
Getting a SERDES right is a good example of multiple clocks

- 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](#)[Parting Thoughts](#)

Be aware of the asynchronous reset signal!

i\_areset\_n



i\_fast\_clk



i\_pin



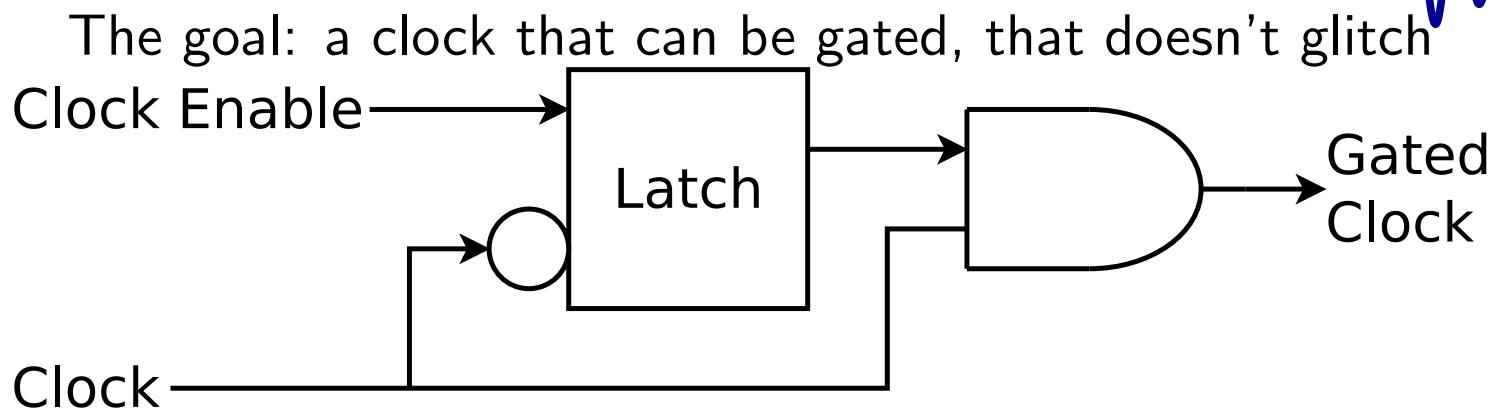
i\_slow\_clk



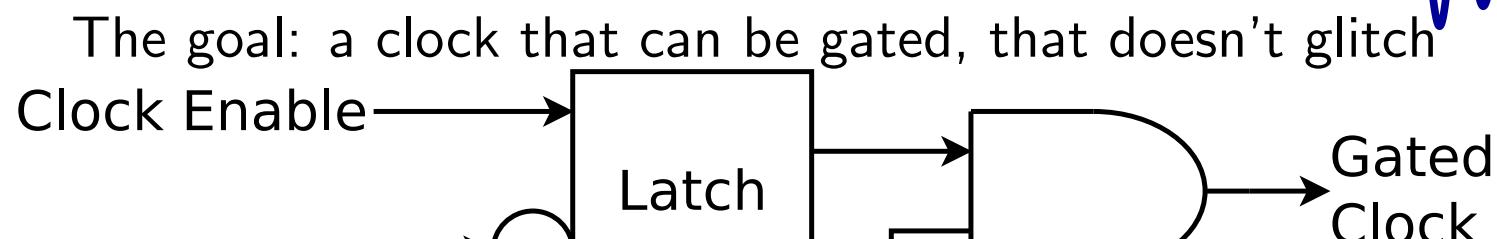
o\_word



- 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](#)[Parting Thoughts](#)

- exercise-10/ Contains the file clkgate.vhd

[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](#)[Parting Thoughts](#)

Clock

i\_clk

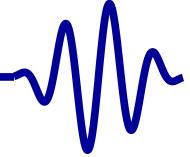


i\_en



o\_clk

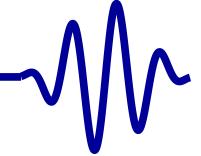


[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Basics](#)[SBY File](#)[\(\\* gclk \\*\)](#)[\\$rose](#)[\\$stable](#)[Examples](#)[▷ Exercises](#)[Cover](#)[Sequences](#)[Parting Thoughts](#)

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

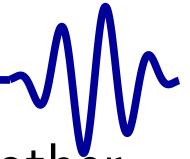
See `exercise-10/clkgate.v`

[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](#)[Parting Thoughts](#)

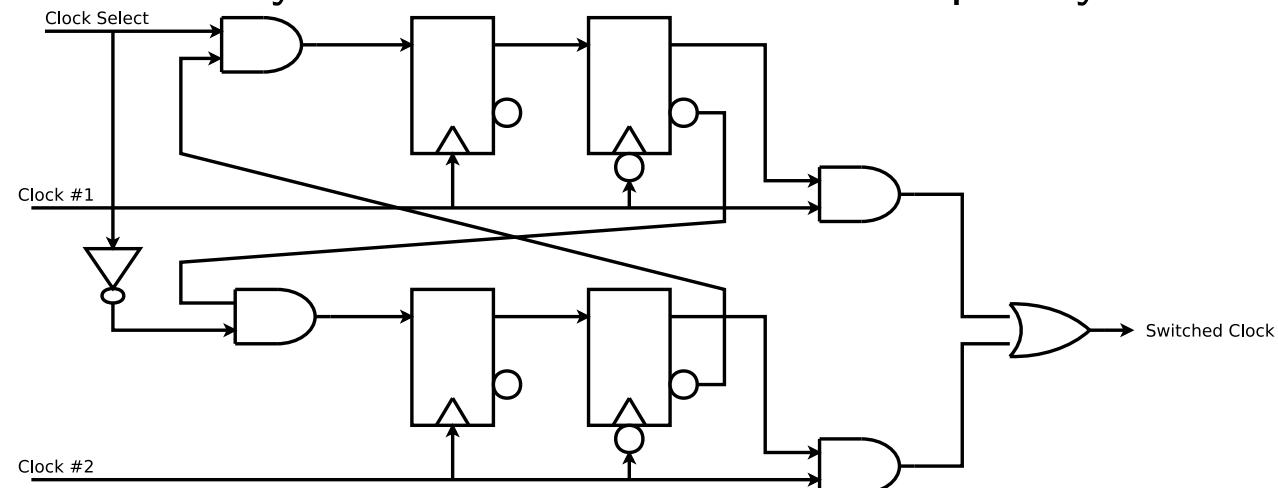
### Hints:

- 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](#)[Parting Thoughts](#)

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](#)[Parting Thoughts](#)

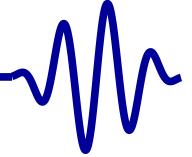
Goal: To safely switch from one clock frequency to another

- Inputs
  - Two arbitrary clocks
  - One select line

Prove that the output clock

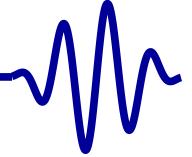
- Is always high (or low) for at least the duration of one of the clocks
- Doesn't stop

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](#)[Parting Thoughts](#)

## 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

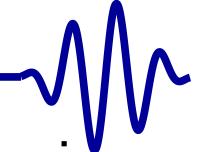
Sequences

Parting Thoughts

# 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

Parting Thoughts

The cover element is used to make certain something remains possible

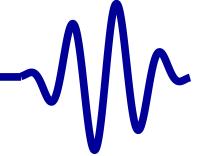
- BMC and induction test *safety* properties  
They prove that something *will not* happen
- Cover tests a *liveness* property  
It proves that something *may* happen

## 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

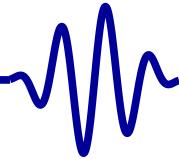
Parting Thoughts

## 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?

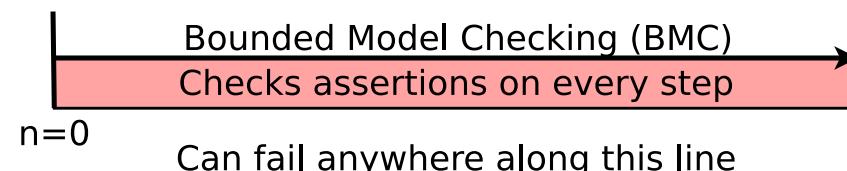
What is cover good for? Catching the *careless assumption!*

What else? Ad hoc simulation traces!

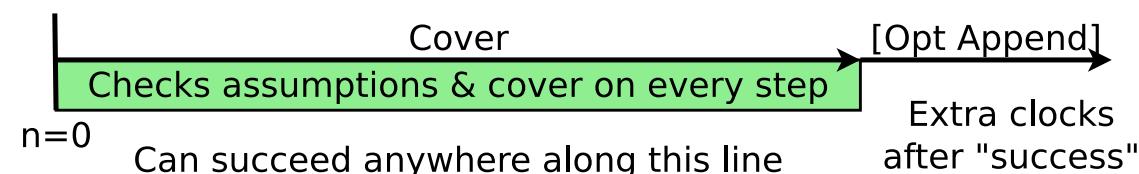
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[▷ BMC vs Cover](#)[Cover in Verilog](#)[State Space](#)[SymbiYosys](#)[Examples](#)[Counter](#)[Sequences](#)[Parting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

Cover is more like BMC than Induction is

- BMC



- Cover



- BMC searches for failures

- Cover searches for a success

Formally, we might say . . .

- BMC +  $k$ -Induction: proof for all

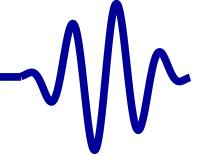
$$\forall \text{assume}() \Rightarrow \forall \text{assert}()$$

- Cover: there exists one

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



# Cover in Verilog



Welcome

Motivation

Basics

Clocked and \$past

k Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Lesson Overview

BMC vs Cover

▷ Cover in Verilog

State Space

SymbiYosys

Examples

Counter

Sequences

Parting Thoughts

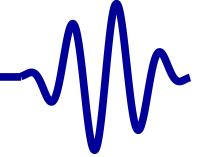
Just like an assumption or an assertion

```
// 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](#)[Parting Thoughts](#)

Assert and cover handle surrounding logic differently

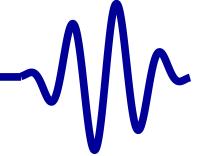
- Assert logic

```
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](#)[Parting Thoughts](#)

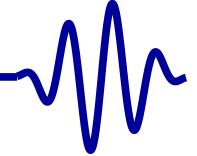
Assert and cover handle surrounding logic differently

- 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

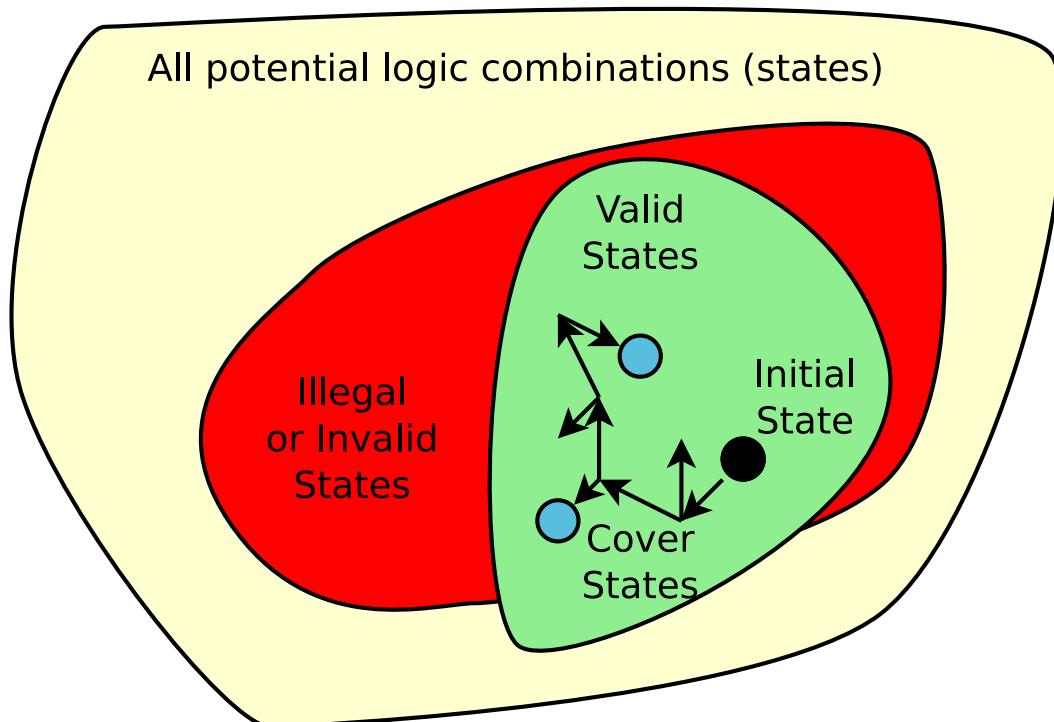
Examples

Counter

---

Sequences

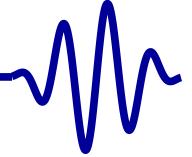
---

Parting Thoughts

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



# SymbiYosys



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Lesson Overview

BMC vs Cover

Cover in Verilog

State Space

▷ SymbiYosys

Examples

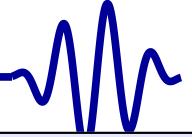
Counter

Sequences

Parting Thoughts

The SymbiYosys script for cover needs to change as well

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

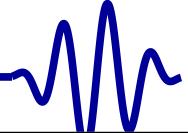


```
[options]
mode cover
depth 40
append 20

[engines]
smtbmc

[script]
read -vhdl module.vhd
read -formal module_vhd.sv
prep -top module

[files]
# file list
```



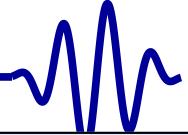
Welcome  
Motivation  
Basics  
Clocked and \$past  
*k* Induction  
Bus Properties  
Free Variables  
Abstraction  
Invariants  
Multiple-Clocks  
Cover  
Lesson Overview  
BMC vs Cover  
Cover in Verilog  
State Space  
▷ SymbiYosys  
Examples  
Counter  
Sequences  
Parting Thoughts

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

[engines]
smtbmc

[script]
read -vhdl module.vhd
read -formal module_vhd.sv
prep -top module

[files]
# file list
```



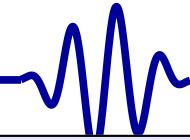
Welcome  
Motivation  
Basics  
Clocked and \$past  
 $k$  Induction  
Bus Properties  
Free Variables  
Abstraction  
Invariants  
Multiple-Clocks  
Cover  
Lesson Overview  
BMC vs Cover  
Cover in Verilog  
State Space  
▷ SymbiYosys  
Examples  
Counter  
Sequences  
Parting Thoughts

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

[engines]
smtbmc

[script]
read -vhdl module.vhd
read -formal module_vhd.sv
prep -top module

[files]
# file list
```



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

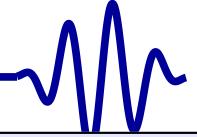
[ engines ]
smtbmc

[ script ]
read -vhdl module.vhd
read -formal module_vhd.sv
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
- 
- Parting Thoughts

[ **tasks** ]

prf

cvr

[ **options** ]

prf : **mode** prove

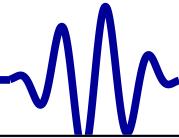
cvr : **mode** cover

**depth** 40

# . . .



# SymbiYosys tasks

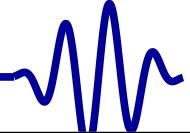


- >Welcome
- 
- Motivation
- 
- Basics
- 
- Clocked and \$past
- 
- $k$  Induction
- 
- Bus Properties
- 
- Free Variables
- 
- Abstraction
- 
- Invariants
- 
- Multiple-Clocks
- 
- Cover
- 
- Lesson Overview
- BMC vs Cover
- Cover in Verilog
- State Space
- ▷ SymbiYosys
- Examples
- Counter
- Sequences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

```
[ tasks ]
prf ←———— Run two tasks: prf and cvr
cvr

[ options ]
prf: mode prove
cvr: mode cover
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
- Parting Thoughts

### [ tasks ]

prf

cvr

### [ options ]

prf : mode prove ← The prf tasks runs induction

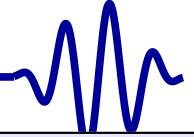
cvr : mode cover

depth 40

# . . .



# SymbiYosys tasks

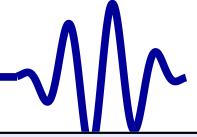


Welcome  
Motivation  
Basics  
Clocked and \$past  
 $k$  Induction  
Bus Properties  
Free Variables  
Abstraction  
Invariants  
Multiple-Clocks  
Cover  
Lesson Overview  
BMC vs Cover  
Cover in Verilog  
State Space  
▷ SymbiYosys  
Examples  
Counter  
Sequences  
Parting Thoughts

```
[ 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  
Parting Thoughts

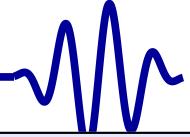
**[ tasks ]**

prf

cvr

**[ options ]**prf : **mode** provecvr : **mode** cover**depth** 40 ← The same depth can apply to both

# . . .

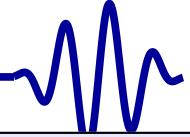
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[BMC vs Cover](#)[Cover in Verilog](#)[State Space](#)[▷ SymbiYosys](#)[Examples](#)[Counter](#)[Sequences](#)[Parting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

[ **tasks** ]

prf

cvr

[ **options** ]

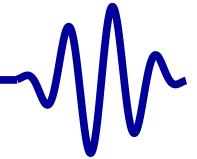
prf: **mode** prove

cvr: **mode** cover

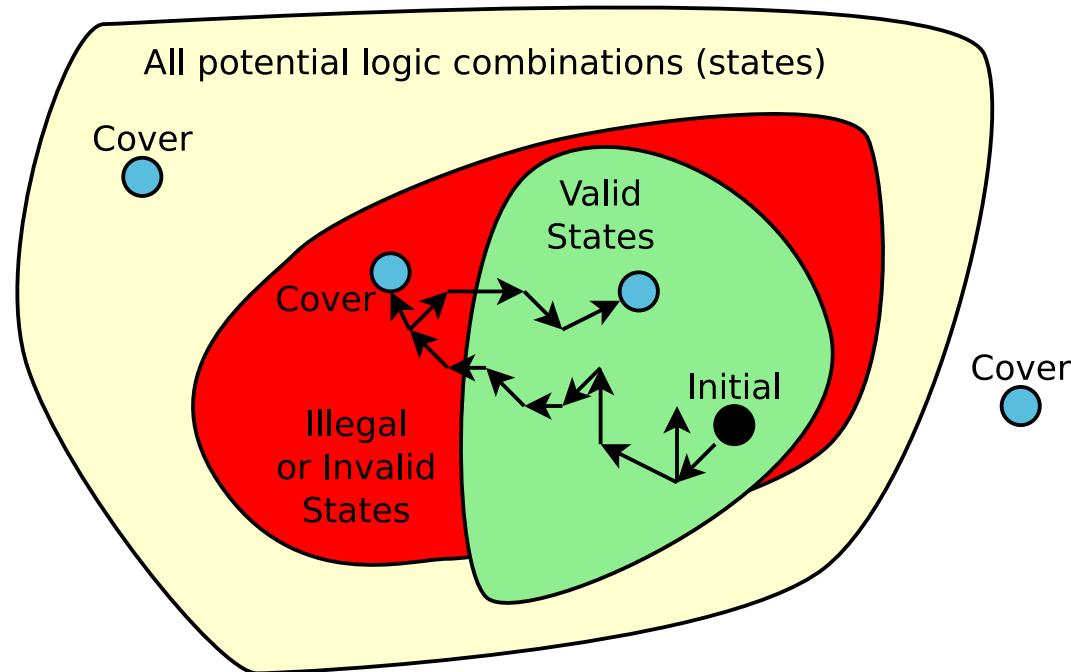
**depth** 40

# . . .

% 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
- Parting Thoughts

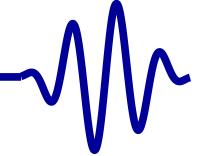


Two basic types of cover failures

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

Parting Thoughts

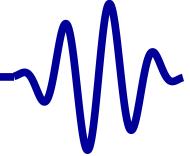
Consider a CPU I-cache:

```
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](#)[Parting Thoughts](#)

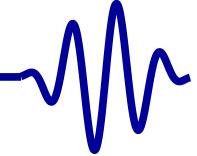
Consider a Flash controller:

```
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

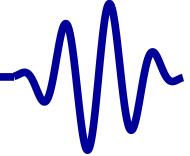
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Lesson Overview](#)[BMC vs Cover](#)[Cover in Verilog](#)[State Space](#)[SymbiYosys](#)[▷ Examples](#)[Counter](#)[Sequences](#)[Parting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

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](#)[Parting Thoughts](#)

How about an SDRAM controller?

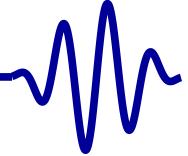
```
always @(posedge i_clk)
    cover(o_wb_ack);
```

The controller must,

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



# Counter



Welcome

Motivation

Basics

Clocked and \$past

*k* Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Lesson Overview

BMC vs Cover

Cover in Verilog

State Space

SymbiYosys

Examples

▷ Counter

Sequences

Parting Thoughts

Remember our counter?

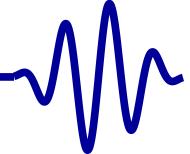
```
signal  counts : unsigned(15 downto 0)
        := to_unsigned(0,16);

process(i_clk)
begin
  if (rising_edge(i_clk)) then
    if ((i_start_signal = '1')
        and (0 = counts)) then
      counts <= to_unsigned(MAX_AMOUNT-1, 16);
    else
      counts <= counts - 1;
    end if;
  end if;
end process;

o_busy <= '1' when (0 = counts) else '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

Parting Thoughts

Let's add some cover statements...

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

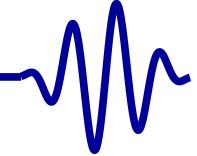
// Transition back to idle
always @(posedge i_clk)
if ((f_past_valid)&&($past(o_busy)))
    cover(!o_busy);

// 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

Parting Thoughts

How about now?

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



# Examples



Welcome

Motivation

Basics

Clocked and \$past

*k* Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Lesson Overview

BMC vs Cover

Cover in Verilog

State Space

SymbiYosys

Examples

▷ Counter

Sequences

Parting Thoughts

How about now?

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

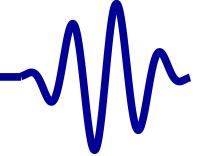
Or this one,

```
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

Parting Thoughts

How about now?

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

Or this one,

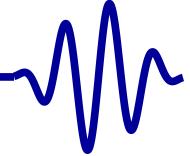
```
always @(posedge i_clk)
    cover(counter == MAX_AMOUNT);
```

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

Parting Thoughts

What if the state is unreachable?

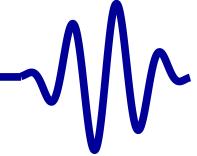
```
// Keep the counter from ever starting
always @(*)
    assume (!i_start_signal);

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

Parting Thoughts

What if the state is unreachable?

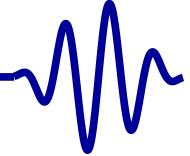
```
// Keep the counter from ever starting
always @(*)  
    assume (!i_start_signal);  
  
always @(posedge i_clk)  
    cover(counter != 0);
```

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

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



# Examples



Welcome

Motivation

Basics

Clocked and \$past

*k* Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Lesson Overview

BMC vs Cover

Cover in Verilog

State Space

SymbiYosys

Examples

▷ Counter

Sequences

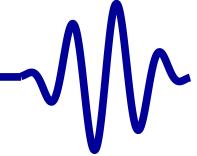
Parting Thoughts

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](#)[Parting Thoughts](#)

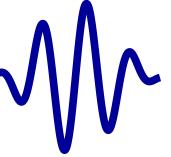
What if an assertion needs to be violated?

```
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](#)[Parting Thoughts](#)

## 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](#)[Parting Thoughts](#)

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](#)[Parting Thoughts](#)

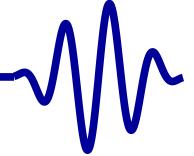
Notice what we just proved:

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

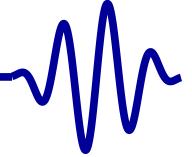
Examples

▷ Counter

Sequences

Parting Thoughts

## When should you use cover?



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

▷ Sequences

Overview

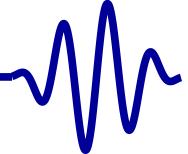
Clocking

Sequences

Exercise

Parting Thoughts

# Sequences

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

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](#)[Sequences](#)[Exercise](#)[Parting Thoughts](#)

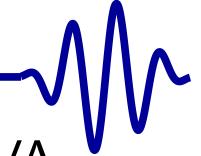
Much of what we've written can easily be rewritten in SVA

```
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](#)[Sequences](#)[Exercise](#)[Parting Thoughts](#)

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](#)[Sequences](#)[Exercise](#)[Parting Thoughts](#)

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

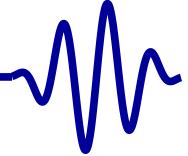
```
always @(posedge i_clk)
if ((f_past_valid)&&($past(A)))
    assert(B);
```

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](#)[Sequences](#)[Exercise](#)[Parting Thoughts](#)

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](#)[Sequences](#)[Exercise](#)[Parting Thoughts](#)

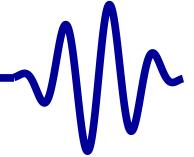
You could also do something like:

```
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);
```

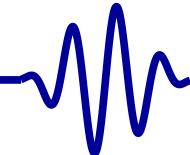
This would work quite nicely for a bus property file

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

Properties can also accept parameters

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

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

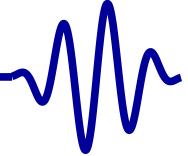
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[▷ Overview](#)[Clocking](#)[Sequences](#)[Exercise](#)[Parting Thoughts](#)

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](#)[Sequences](#)[Exercise](#)[Parting Thoughts](#)

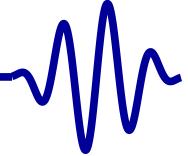
Getting tired of writing `@(posedge i_clk)`?

- 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

Sequences

Exercise

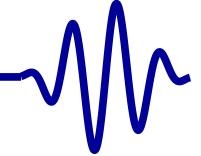
Parting Thoughts

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

▷ Sequences

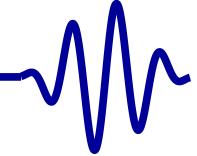
Exercise

Parting Thoughts

So far with properties,

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

Sequences are something new

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

With sequences, you can

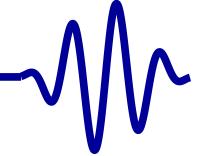
- Specify a series of actions

**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

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

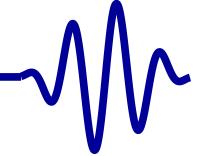
With sequences, you can

- 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

▷ Sequences

Exercise

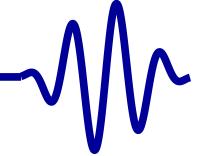
Parting Thoughts

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

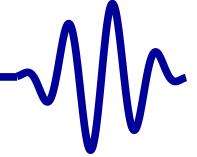
▷ Sequences

Exercise

Parting Thoughts

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](#)[▷ Sequences](#)[Exercise](#)[Parting Thoughts](#)

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](#)[▷ Sequences](#)[Exercise](#)[Parting Thoughts](#)

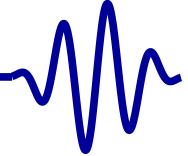
- Throughout

```
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

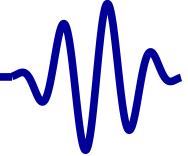
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[▷ Sequences](#)[Exercise](#)[Parting Thoughts](#)

- 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](#)[▷ Sequences](#)[Exercise](#)[Parting Thoughts](#)

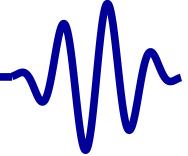
- Throughout
- Until

```
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

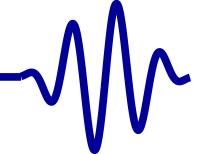
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[▷ Sequences](#)[Exercise](#)[Parting Thoughts](#)

- 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](#)[▷ Sequences](#)[Exercise](#)[Parting Thoughts](#)

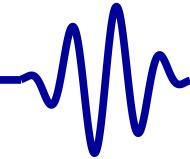
Properties can reference sequences

- 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

▷ Sequences

Exercise

Parting Thoughts

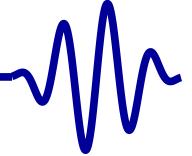
Properties can include . . .

- **if** statements

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

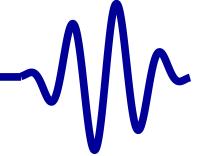
- **not**, **and**, or even **or** statements

```
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);
```

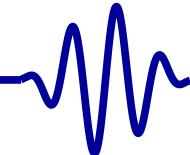
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[▷ Sequences](#)[Exercise](#)[Parting Thoughts](#)

A request persists until it is accepted

```
sequence BUS_REQUEST;
  @(posedge i_clk)
    // Repeat up to MAX_STALL clk
    (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](#)[▷ Sequences](#)[Exercise](#)[Parting Thoughts](#)

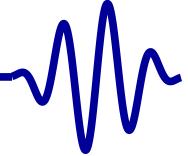
A request persists until it is accepted

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

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

You no longer need to count stalls yourself.

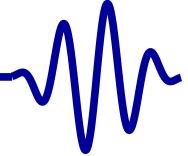
Could we do this with an **until** statement?

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

A request persists until it is accepted

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

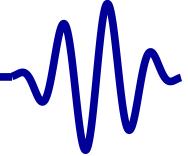
What is the difference?

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

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.

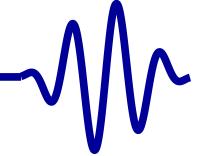
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[▷ Sequences](#)[Exercise](#)[Parting Thoughts](#)

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?

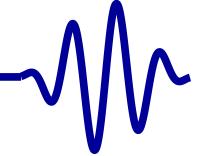
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[▷ Sequences](#)[Exercise](#)[Parting Thoughts](#)

A property can be conditionally disabled

```
sequence BUS_REQUEST;
    // Repeat up to MAX_STALL c/ks
    (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?

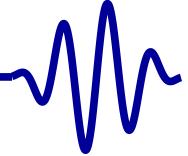
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[▷ Sequences](#)[Exercise](#)[Parting Thoughts](#)

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](#)[▷ Sequences](#)[Exercise](#)[Parting Thoughts](#)

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](#)[▶ Sequences](#)[Exercise](#)[Parting Thoughts](#)

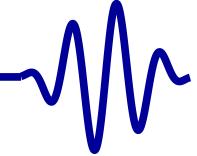
Some peripherals will only ever accept one request

```
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](#)[▷ Sequences](#)[Exercise](#)[Parting Thoughts](#)

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](#)[▷ Sequences](#)[Exercise](#)[Parting Thoughts](#)

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](#)[▷ Sequences](#)[Exercise](#)[Parting Thoughts](#)

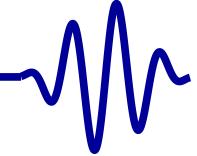
Some peripherals will

- 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



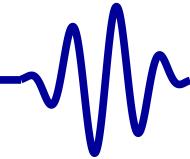
Cannot ACK or ERR when no request is pending

```
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));
```

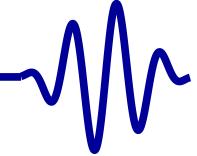
Which is simpler to understand?



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

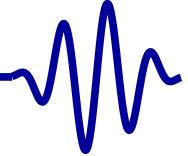
- Output data is constant
- Logic doesn't change state
- Internal shift register value is known
- Ends with zero\_baud\_counter

```
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](#)[▷ Sequences](#)[Exercise](#)[Parting Thoughts](#)

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
```

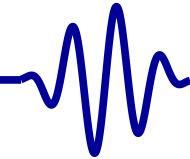
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[▷ Sequences](#)[Exercise](#)[Parting Thoughts](#)

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

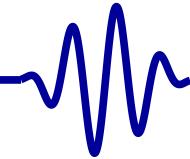
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[▷ Sequences](#)[Exercise](#)[Parting Thoughts](#)

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](#)[▷ Sequences](#)[Exercise](#)[Parting Thoughts](#)

## SystemVerilog Concurrent Assertions . . .

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

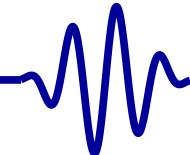
You can keep using the simpler property form we've been using

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

Let's formally verify a synchronous FIFO

```
entity sfifo is
    generic(BW : natural := 8;
            LGFLEN : natural := 4);

    port(i_clk, i_reset : in std_logic;
        — The incoming (write) interface
        i_wr      : in std_logic;
        i_data    : in std_logic_vector(BW-1 downto 0);
        o_full    : out std_logic := '0';
        — The outgoing (read) interface
        i_rd      : in std_logic;
        o_data    : out std_logic_vector(BW-1 downto 0);
        o_empty   : out std_logic := '1';
        o_err     : out std_logic := '0');
    end entity sfifo;
```

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

Let's formally verify a synchronous FIFO

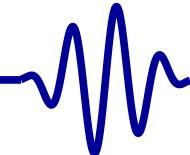
```
architecture behavior of sfifo is
  constant FLEN : natural := 2 ** LGFLEN;

  type data_type is
    std_logic_vector (BW-1 downto 0);
  type mem_type is
    array (FLEN-1 downto 0) of data_type;
  type ptr_type is
    unsigned (LGFLEN downto 0);

  signal mem : mem_type;

  signal r_first: ptr_type
    := to_unsigned(0,LGFLEN+1);
```

See the problem?

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

Let's formally verify a synchronous FIFO

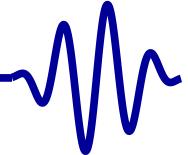
```
architecture behavior of sfifo is
  constant FLEN : natural := 2 ** LGFLEN;

  type data_type is
    std_logic_vector (BW-1 downto 0);
  type mem_type is
    array (FLEN-1 downto 0) of data_type;
  type ptr_type is
    unsigned (LGFLEN downto 0);

  signal mem : mem_type;

  signal r_first: ptr_type
    := to_unsigned(0, LGFLEN+1);
```

See the problem? You can't pass memories through ports!

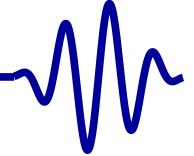
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Sequences](#)[▷ Exercise](#)[Parting Thoughts](#)

How will you pass the memory to the formal tool?

- You might pass an arbitrary address and value instead

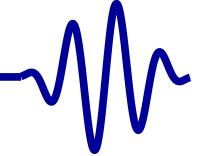
```
signal test_address: ptr_type;
attribute anyconst of test_address
    : signal is '1';
signal test_value: data_type;

test_value <= mem(to_integer(unsigned(
    test_address(LGFLEN-1 downto 0))));
```



Welcome  
Motivation  
Basics  
Clocked and \$past  
 $k$  Induction  
Bus Properties  
Free Variables  
Abstraction  
Invariants  
Multiple-Clocks  
Cover  
Sequences  
Overview  
Clocking  
Sequences  
▷ Exercise  
Parting Thoughts

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

Sequences

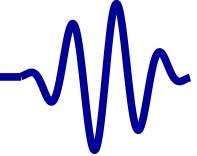
▷ Exercise

Parting Thoughts

Let's formally verify a synchronous FIFO

What properties do you think would be appropriate?

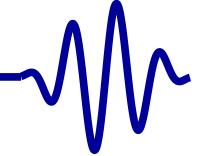
- Should never go from full to empty

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

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

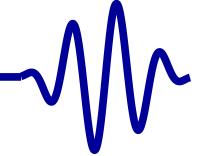
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Sequences](#)[▷ Exercise](#)[Parting Thoughts](#)

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

# Last Exercise



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Overview

Clocking

Sequences

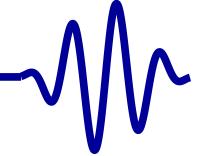
▷ Exercise

Parting Thoughts

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](#)[Sequences](#)[▷ Exercise](#)[Parting Thoughts](#)

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)

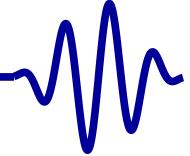
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Sequences](#)[▷ Exercise](#)[Parting Thoughts](#)

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

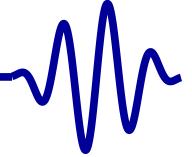
[Welcome](#)[Motivation](#)[Basics](#)[Clocked and \\$past](#)[k Induction](#)[Bus Properties](#)[Free Variables](#)[Abstraction](#)[Invariants](#)[Multiple-Clocks](#)[Cover](#)[Sequences](#)[Overview](#)[Clocking](#)[Sequences](#)[▷ Exercise](#)[Parting Thoughts](#)

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



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

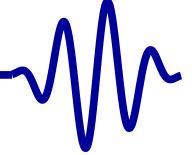
Parting  
▷ Thoughts

Questions?

# Parting Thoughts



# Questions?



Welcome

Motivation

Basics

Clocked and \$past

$k$  Induction

Bus Properties

Free Variables

Abstraction

Invariants

Multiple-Clocks

Cover

Sequences

Parting Thoughts

▷ Questions?