6.S965
Digital Systems Laboratory II

Lecture 3:

More Simulation Thoughts

and

Zynq Architecture

Administrative

- Week 1’s stuff due Friday at 5pm
- Week 2’s stuff should be out at noon on Friday
- If you find yourself thinking, “I’m probably doing something stupid…” in the context of Vivado, the problem may not be you, it may be Vivado. Please ask for help

CocoTb Stuff

How some of it works…

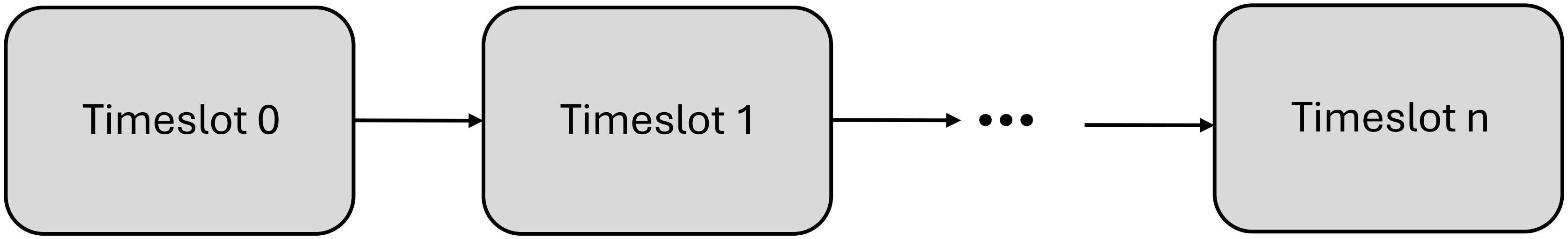
More on Cocotb and the Event Loop

- As we start to write more with Cocotb, we need to see what’s going on with how it is interacting with the Verilog VPI/simulation cycle.

- Understanding its underlying interactions with the Verilog simulations will help us write less problematic code.

Verilog Simulation

- A standard Verilog engine runs through a series of time slots.
- Within each time slot are regions in which different evaluations and updates are made



Timeslot 0

Timeslot 1

…

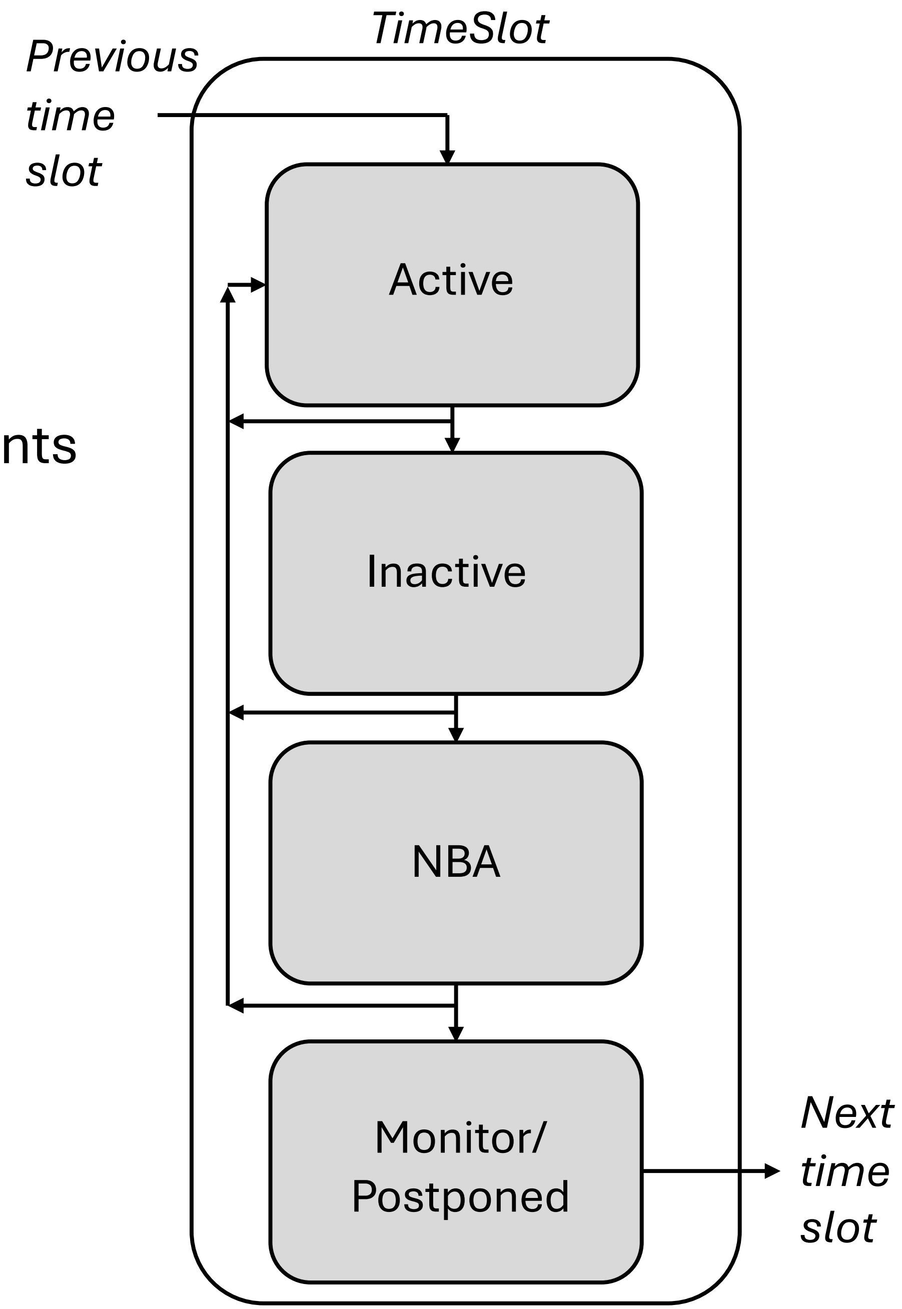
Timeslot n

- The size of the simulation timeslot will be based off the timescale specified. For example:
  - `` `timescale 1ns/1ps ``
  - Means we have basically 1ps time step/slot size

"Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!"
Clifford E. Cummings Sunburst Design, Inc.

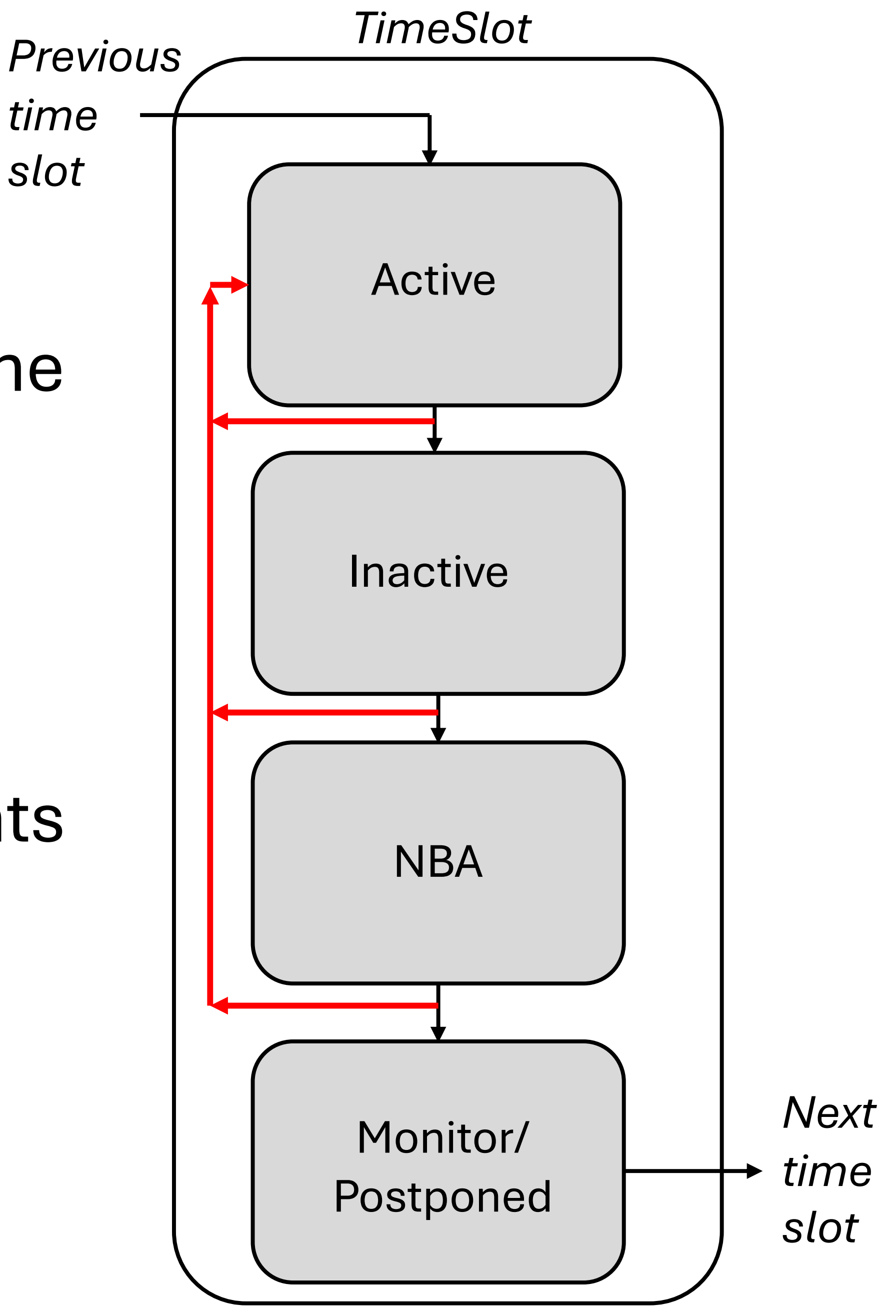
Verilog Simulation Time Step

- Active:
  - Blocking Assignments
  - RHS of non-blocking assignments
  - Continuous assignments
- Inactive Region:
  - “#0 Blocking Assignments” (ignore)
- Non-Blocking Region:
  - LHS updating of non-blocking assignments
- Monitor/Postponed Region:
  - Meant for evaluation of results



Iterative Nature

- The simulation may do run through multiple cycles of the three big stages until things resolve
- Each iteration is a ”delta-cycle” or “delta-step”
- These are all *zero-time* events



Example Code

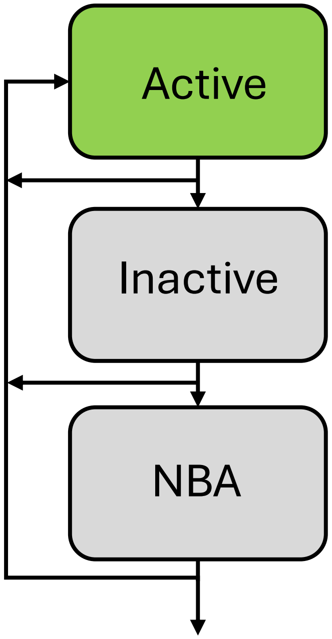
- Here’s some simple SystemVerilog:
- It has a variety of things being done here
- One a simulation step, all of the HDL is evaluated and processed in steps

```
logic clk;
logic [1:0] a,b,c,d,e;

assign a = b + c;

always_comb begin
  b = c + d;
end

always_comb begin
  d=e+2;
end

always_ff @(posedge clk)begin
  c <= e+2;
end
```

Active Region

- Fully do (in any order) non-deterministic:
  - assign a = b + c;
  - b = c + d;
  - d=e+2;
- Evaluate RHS of:
  - c <= e+2;

d updated and b updated. *Because other lines use them in their RHS, the simulator will need to go back through again*

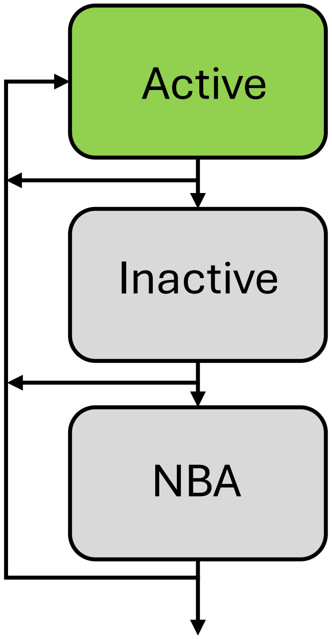


```
logic clk;
logic [1:0] a,b,c,d,e;

assign a = b + c;

always_comb begin
  b = c + d;
end

always_comb begin
  d=e+2;
end

always_ff @(posedge clk)begin
  c <= e+2;
end
```

Active Region II

- `Redo (in any order) non-deterministic:`
  - `assign a = b + c;`
  - `b = c + d;`
- `Evaluate RHS of:`
  - `c <= e+2;`

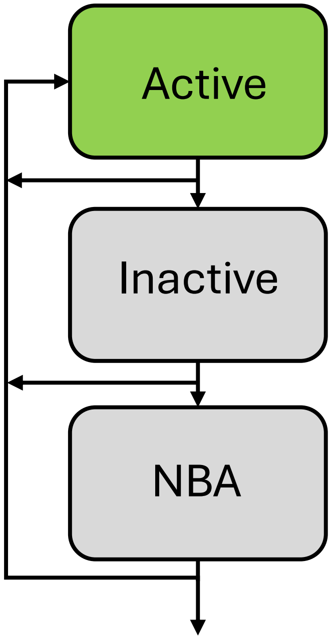
b updated. *Depending on the order these lines were evaluated in, the first one might need to run again given the new value of b*



```
logic clk;
logic [1:0] a,b,c,d,e;

assign a = b + c;

always_comb begin
  b = c + d;
end

always_comb begin
  d=e+2;
end

always_ff @(posedge clk)begin
  c <= e+2;
end
```

Active Region III

- Redo (in any order) non-deterministic:
  - `assign a = b + c;`
- Evaluate RHS of:
  - `c <= e+2;`

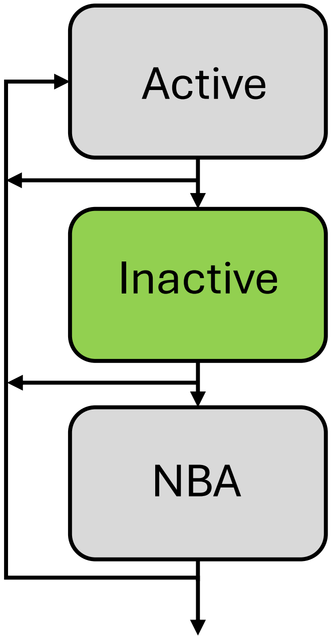


```
logic clk;
logic [1:0] a,b,c,d,e;

assign a = b + c;

always_comb begin
  b = c + d;
end

always_comb begin
  d=e+2;
end

always_ff @(posedge clk)begin
  c <= e+2;
end
```

Shouldn’t be anything left dangling

InActive Region

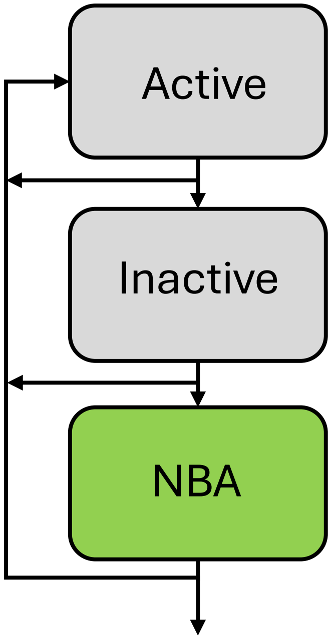
- Highly discourage to use this.
- Just skip this…is a weird delayed region that people use to force order on assignments
- Kinda like `!important` in CSS if anybody does webdev



```
logic clk;
logic [1:0] a,b,c,d,e;

assign a = b + c;

always_comb begin
  b = c + d;
end

always_comb begin
  d=e+2;
end

always_ff @(posedge clk)begin
  c <= e+2;
end
```

NBA Region

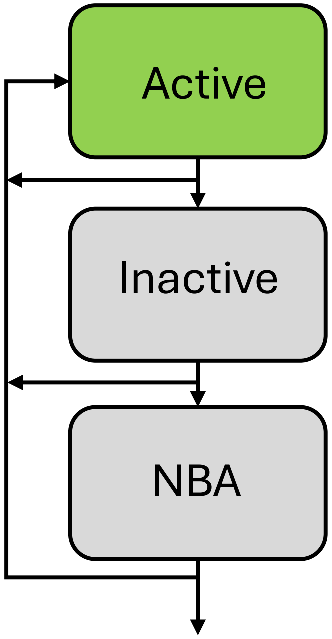
- `Transfer result of e+2 to c`



c updated. *Because c was used in the assignments of b and a, we will return back to the Active region to recalculate*

```
logic clk;
logic [1:0] a,b,c,d,e;

assign a = b + c;

always_comb begin
  b = c + d;
end

always_comb begin
  d=e+2;
end

always_ff @(posedge clk)begin
  c <= e+2;
end
```

Active Region IV



- `Fully do (in any order) non-deterministic:`
  - `assign a = b + c;`
  - `b = c + d;`

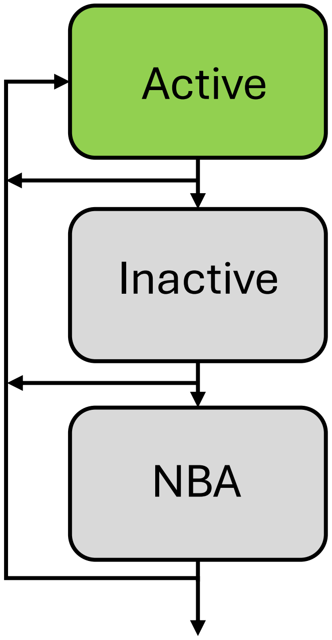
b updated. *Depending on the order these lines were evaluated in, the first one might need to run again given the new value of* ***b***

```
logic clk;
logic [1:0] a,b,c,d,e;

assign a = b + c;

always_comb begin
  b = c + d;
end

always_comb begin
  d=e+2;
end

always_ff @(posedge clk)begin
  c <= e+2;
end
```

Active Region V

- `Redo (in any order) non-deterministic:`
  - `assign a = b + c;`
- `Evaluate RHS of:`
  - `c <= e+2;`

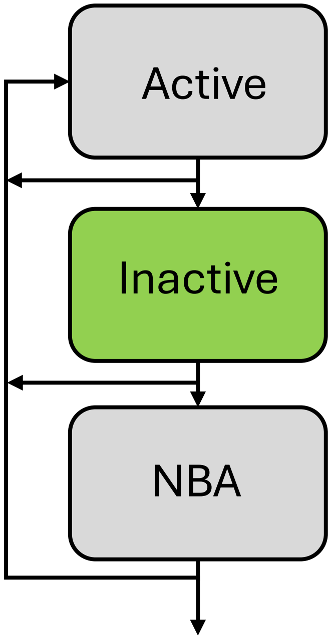
Shouldn't be anything left dangling



```
logic clk;
logic [1:0] a,b,c,d,e;

assign a = b + c;

always_comb begin
  b = c + d;
end

always_comb begin
  d=e+2;
end

always_ff @(posedge clk)begin
  c <= e+2;
end
```

InActive Region II

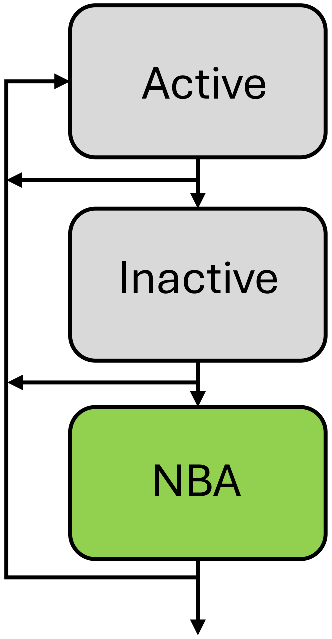
- Highly discourage to use this.
- Just skip this…is a weird delayed region that people use to force order on assignments
- Kinda like `!important` in CSS if anybody does webdev



```
logic clk;
logic [1:0] a,b,c,d,e;

assign a = b + c;

always_comb begin
  b = c + d;
end

always_comb begin
  d=e+2;
end

always_ff @(posedge clk)begin
  c <= e+2;
end
```

NBA Region II

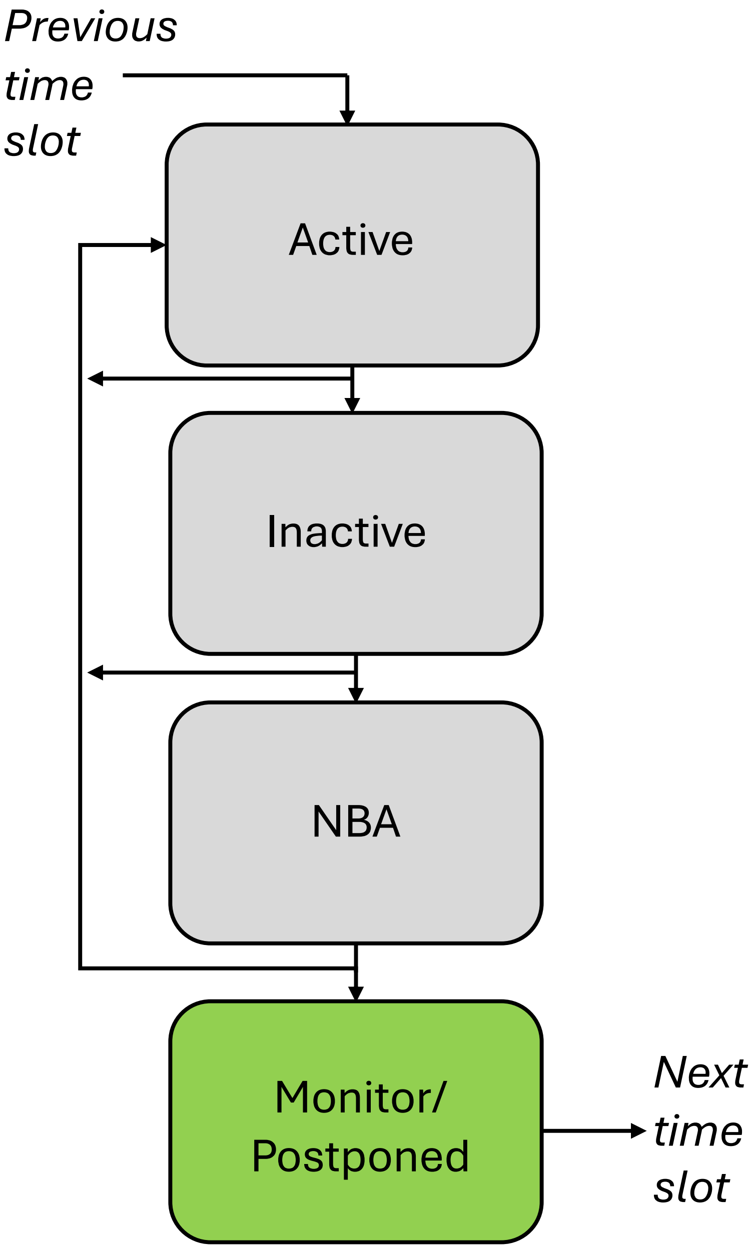
- Nothing new this time through



```
logic clk;
logic [1:0] a,b,c,d,e;

assign a = b + c;

always_comb begin
  b = c + d;
end

always_comb begin
  d=e+2;
end

always_ff @(posedge clk)begin
  c <= e+2;
end
```

Monitor Region

- Only after we've completely "stabilized" in all of our calculations.



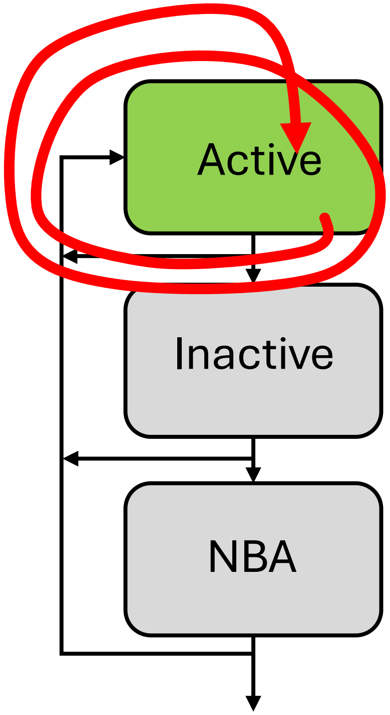
*NBA = Non-blocking Assignments

It is absolutely possible to have unsolveable Verilog

- A simple combinational loop is technically infinite:

```
logic [7:0] d;
assign d = d + 1;
```

- Might get some warnings about combinational loop
- In simulation iVerilog will just say this is always **X**



Would you ever run through NBA more than twice?

- In the earlier “good” example we ran through the NBA region twice:
  - First time through it we did a LHS update…this triggered a loopback since some continual assignments used the thing updated in their RHS
  - Second time through nothing was left to do.
- Once a non-blocking assignment is carried out, it is removed from the list of things that needs to be addressed (prevents it from being done multiple times…)
- So question: Could you ever run through the NBA more than twice on a time step?

Some messed up Verilog:

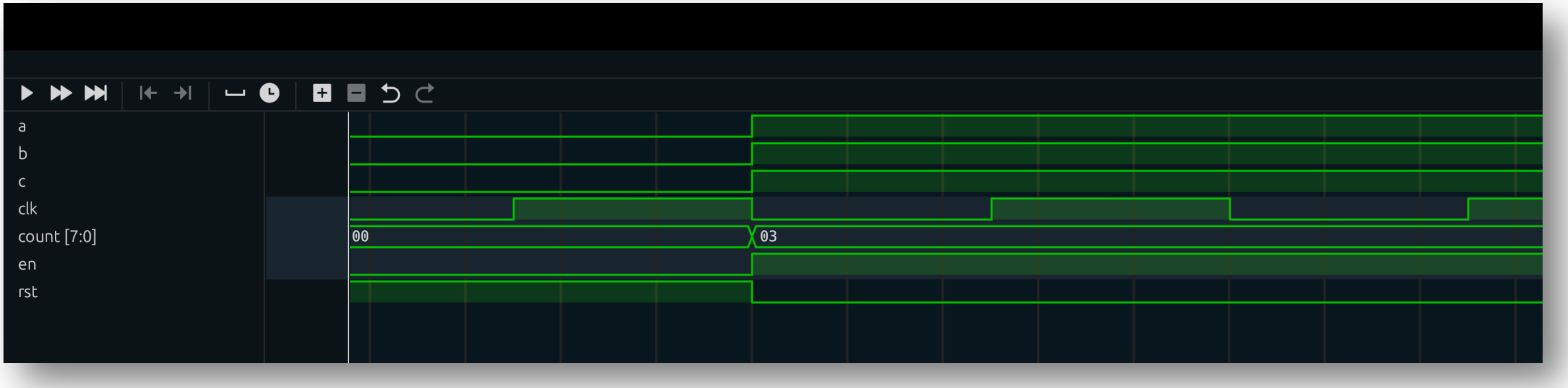
```
module messed_up(
     input wire clk,
     input wire rst,
     input wire en,
     output logic [7:0] count
     );
     logic a,b,c;
     initial begin
          //way to say these values start at something rather than X
          a = 0;
          b = 0;
          c = 0;
          count = 0;
     end
     always_ff @(posedge en)begin
          count <= count +1;
          a <= ~a;
     end
     always_ff @(posedge a)begin
          count <= count+1;
          b <= ~b;
     end
     always_ff @(posedge b) begin
          count <= count + 1;
     c <= ~c;
     end
endmodule
```

Wrote this Test Script:

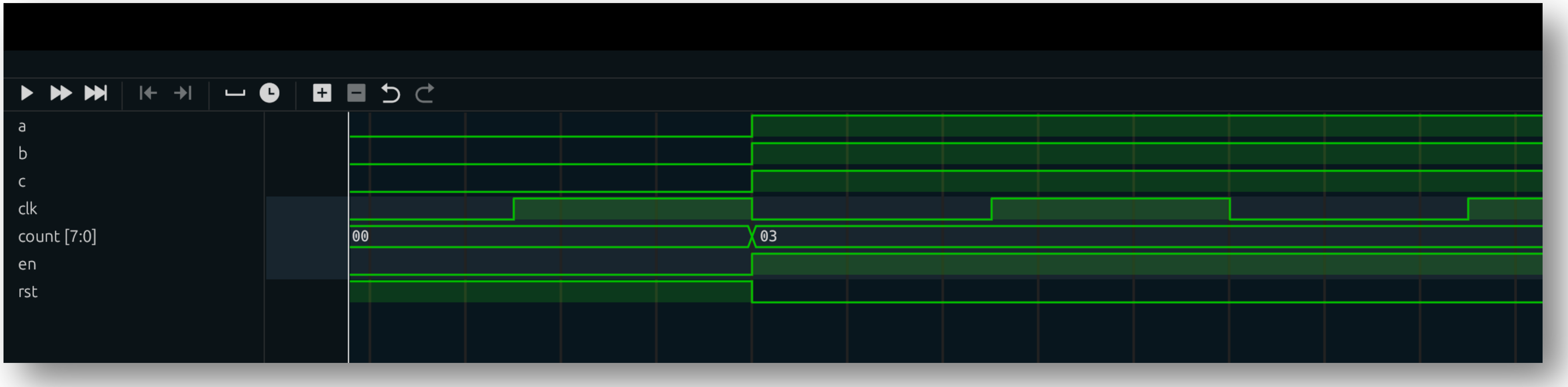
```
@cocotb.test()
async def test_a(dut):
     """cocotb test for seven segment controller"""
     dut._log.info("Starting...")
     cocotb.start_soon(Clock(dut.clk, 10, units="ns").start(start_high=False))
     dut.en.value = 0;
     dut.rst.value = 1;
     await Timer(10, "ns")
     dut.rst.value = 0;
     dut._log.info(f"About to set en to 1.")
     dut.en.value = 1;
     await RisingEdge(dut.en)
     dut._log.info(f"Rising Edge of en caught!")
     dut._log.info(f" Values: en: {dut.en.value}, a: {dut.a.value}, b: {dut.b.value}")
     await RisingEdge(dut.a)
     dut._log.info(f"Rising Edge of a caught!")
     dut._log.info(f" Values: en: {dut.en.value}, a: {dut.a.value}, b: {dut.b.value}")
     await RisingEdge(dut.b)
     dut._log.info(f"Rising Edge of b caught!")
     dut._log.info(f" Values: en: {dut.en.value}, a: {dut.a.value}, b: {dut.b.value}")
     await Timer(99, "ns")
```

When Run

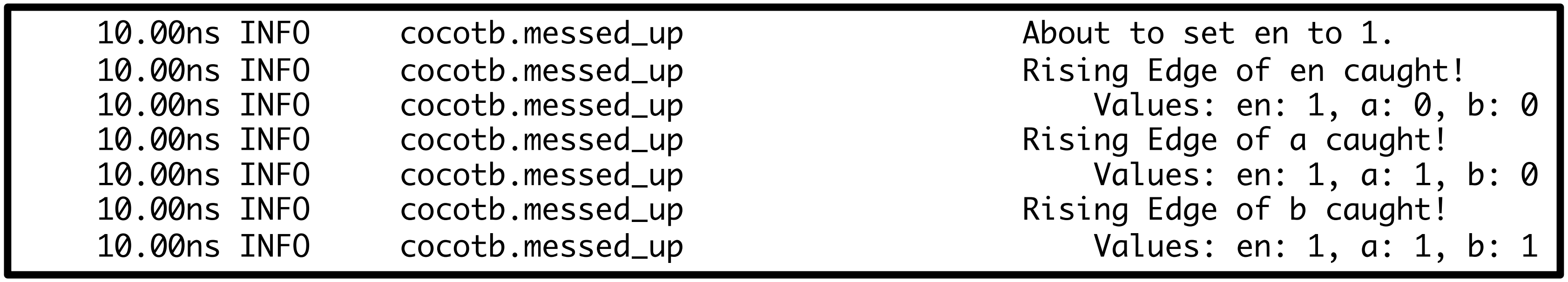
- en goes high…
- count goes from 00 to 03 instantaneously at 10ns…



With Readout…



```
    10.00ns INFO     cocotb.messed_up                   About to set en to 1.
    10.00ns INFO     cocotb.messed_up                   Rising Edge of en caught!
    10.00ns INFO     cocotb.messed_up                     Values: en: 1, a: 0, b: 0
    10.00ns INFO     cocotb.messed_up                   Rising Edge of a caught!
    10.00ns INFO     cocotb.messed_up                     Values: en: 1, a: 1, b: 0
    10.00ns INFO     cocotb.messed_up                   Rising Edge of b caught!
    10.00ns INFO     cocotb.messed_up                     Values: en: 1, a: 1, b: 1
```



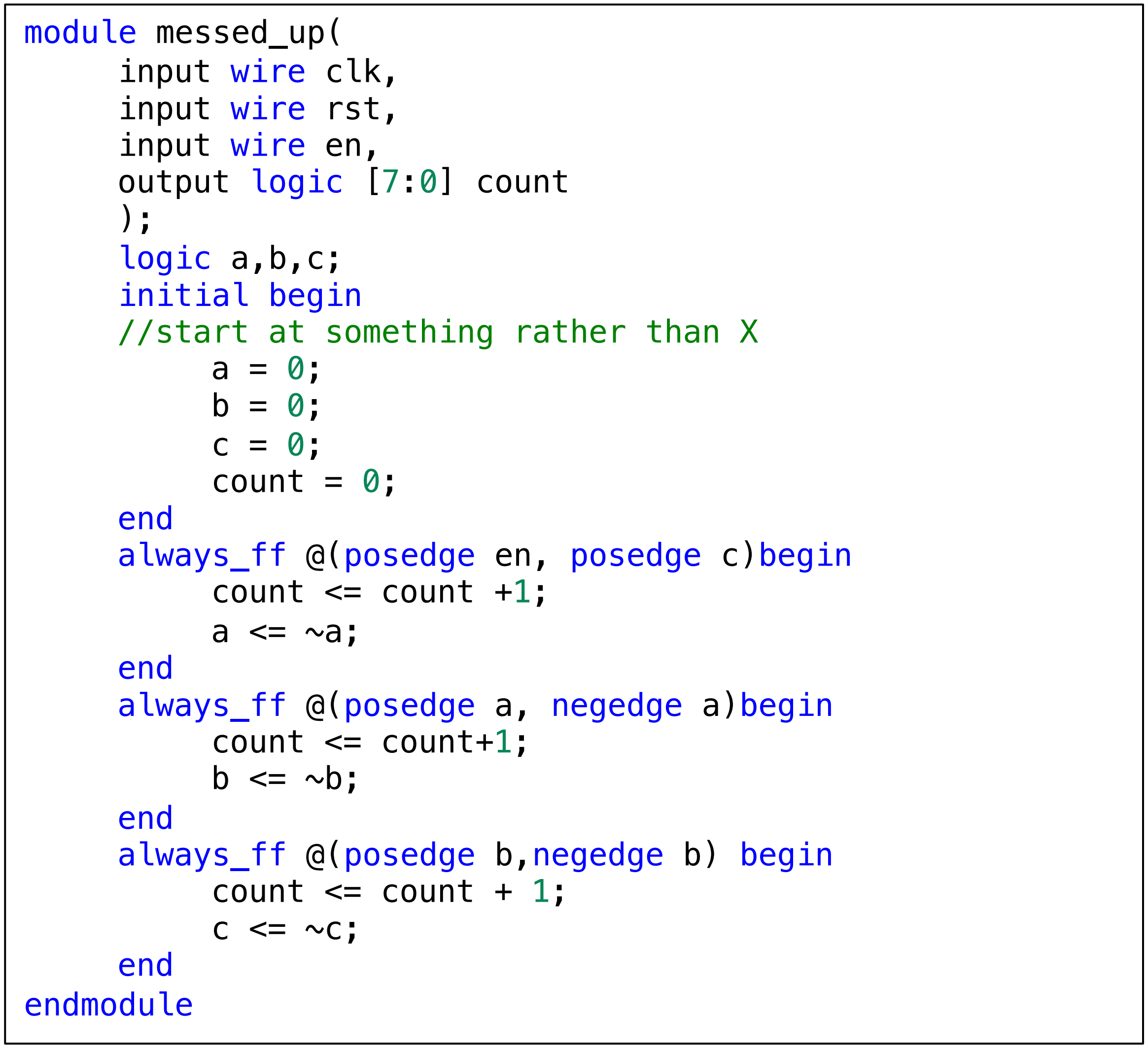
```
    10.00ns INFO     cocotb.messed_up                                    About to set en to 1.
    10.00ns INFO     cocotb.messed_up                                    Rising Edge of en caught!
    10.00ns INFO     cocotb.messed_up                                        Values: en: 1, a: 0, b: 0
    10.00ns INFO     cocotb.messed_up                                    Rising Edge of a caught!
    10.00ns INFO     cocotb.messed_up                                        Values: en: 1, a: 1, b: 0
    10.00ns INFO     cocotb.messed_up                                    Rising Edge of b caught!
    10.00ns INFO     cocotb.messed_up                                        Values: en: 1, a: 1, b: 1
```

- For several cycles there the delta steps finish and then the NBA is applied. That RHS assignment is to a value in the sensitivity list of a `always_ff`, *bringing its lines of stuff* onto the "queue" of stuff to evaluate. Forcing another full round (at least)
- Notice how the the timestamp of all of this is *exactly the same*!
- This is all happening in zero-time as the simulator resolves itself.
- So it went through NBA like four/five times

Could You Mess This *Really* Up?

- Oh yeah…

- This runs fine but…



```
module messed_up(
     input wire clk,
     input wire rst,
     input wire en,
     output logic [7:0] count
     );
     logic a,b,c;
     initial begin
     //start at something rather than X
          a = 0;
          b = 0;
          c = 0;
          count = 0;
     end
     always_ff @(posedge en, posedge c)begin
          count <= count +1;
          a <= ~a;
     end
     always_ff @(posedge a, negedge a)begin
          count <= count+1;
          b <= ~b;
     end
     always_ff @(posedge b,negedge b) begin
          count <= count + 1;
          c <= ~c;
     end
endmodule
```

Could You Really Mess This Up?

- Hard Crash!

- Gets stuck in a death spiral it never gets out of

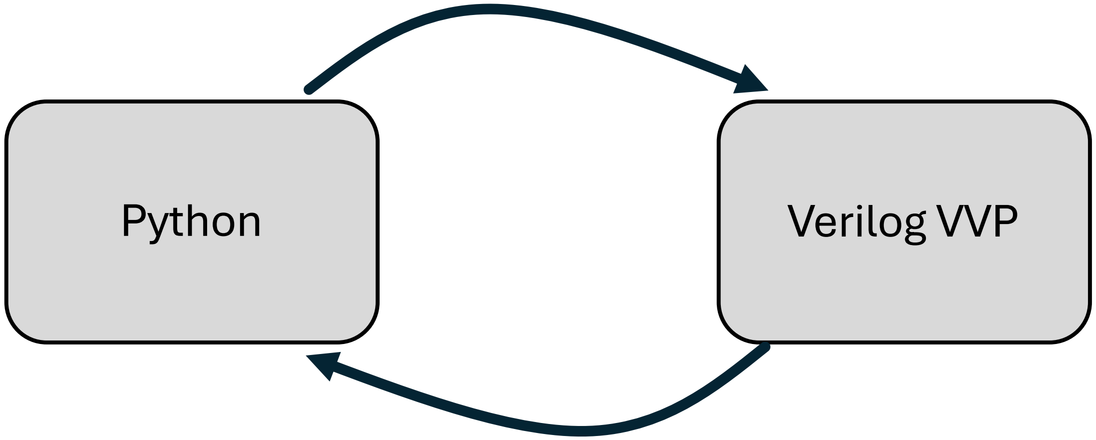
```
module messed_up(
     input wire clk,
     input wire rst,
     input wire en,
     output logic [7:0] count
     );
     logic a,b,c;
     initial begin
     //start at something rather than X
          a = 0;
          b = 0;
          c = 0;
          count = 0;
     end
     always_ff @(posedge en, posedge c, negedge c)begin
          count <= count +1;
          a <= ~a;
     end
     always_ff @(posedge a, negedge a)begin
          count <= count+1;
          b <= ~b;
     end
     always_ff @(posedge b,negedge b) begin
          count <= count + 1;
          c <= ~c;
     end
endmodule
```

Conclusion

- You can break the simulator in many different ways.
- Thankfully if you write good HDL, it minimizes the chances of that.

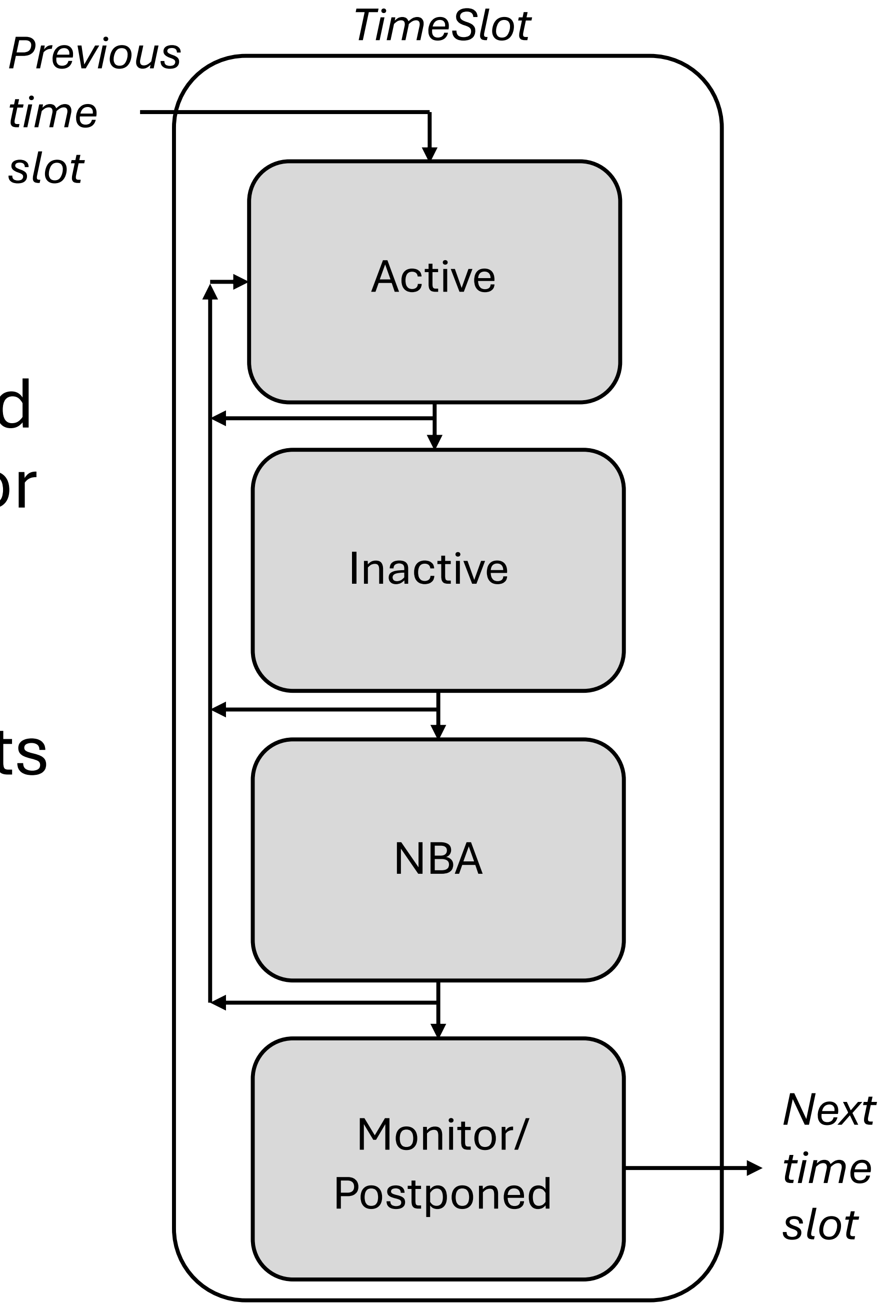
So How Does CocoTB interact with Verilog?

- It is interesting.
- You have two separate programs running and they are effectively ping-ponging control back and forth between each other.



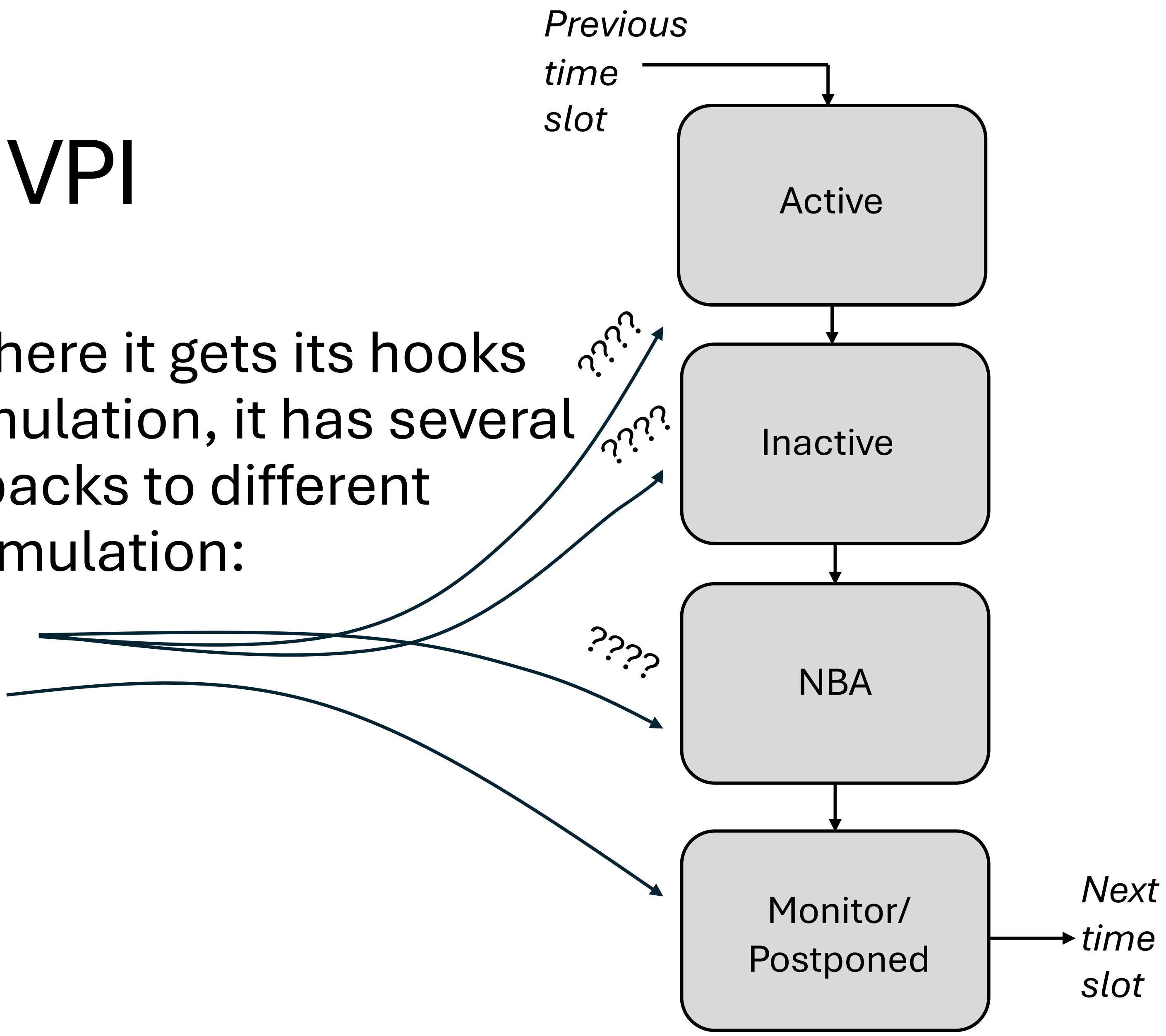
PLI/VPI Usage

- You use PLI/VPI to run and interact with the simulator
- And you also set up ***callbacks*** to pause the simulator at various points to allow you to check in



Verilog VPI

- As far as where it gets its hooks into the simulation, it has several major callbacks to different points in simulation:
  - others
  - readonly



Some Common VPI Callbacks

- Taken from cocotb source code
- Also refer to icarusVerilog source code

```
/****************************** CALLBACK REASONS ******************************/
/***************************** Simulation related *****************************/

#define cbValueChange                 1
#define cbStmt                        2
#define cbForce                       3
#define cbRelease                     4

/******************************** Time related ********************************/

#define cbAtStartOfSimTime            5
#define cbReadWriteSynch              6
#define cbReadOnlySynch               7
#define cbNextSimTime                 8
#define cbAfterDelay                  9

/******************************* Action related *******************************/

#define cbEndOfCompile               10
#define cbStartOfSimulation          11
#define cbEndOfSimulation            12
#define cbError                      13
#define cbTchkViolation              14
#define cbStartOfSave                15
#define cbEndOfSave                  16
#define cbStartOfRestart             17
#define cbEndOfRestart               18
#define cbStartOfReset               19
#define cbEndOfReset                 20
#define cbEnterInteractive           21
#define cbExitInteractive            22
#define cbInteractiveScopeChange     23
#define cbUnresolvedSystf            24
```

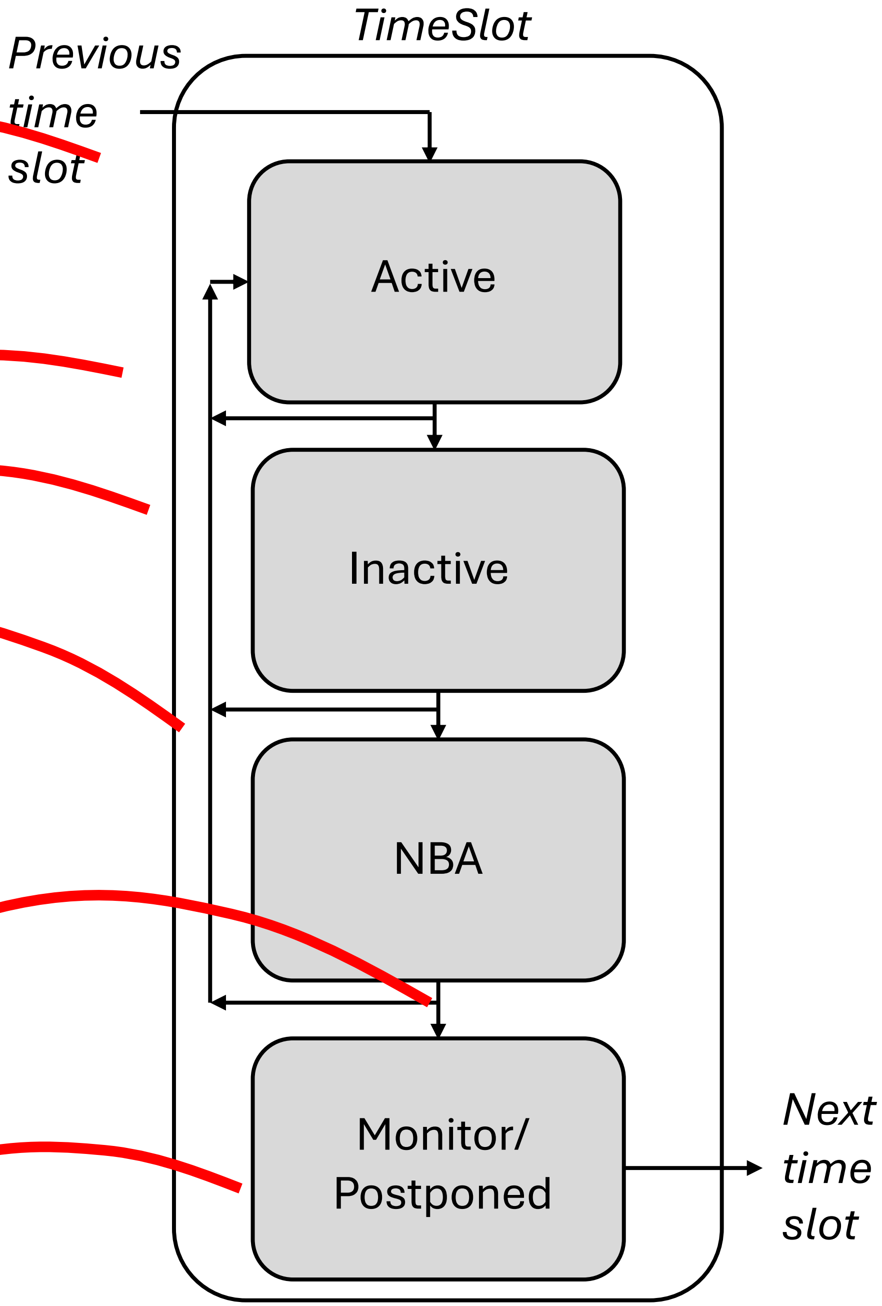
Callbacks

`cbNextTimeStep,`
`cbAfterDelay`

`cbValueChange`

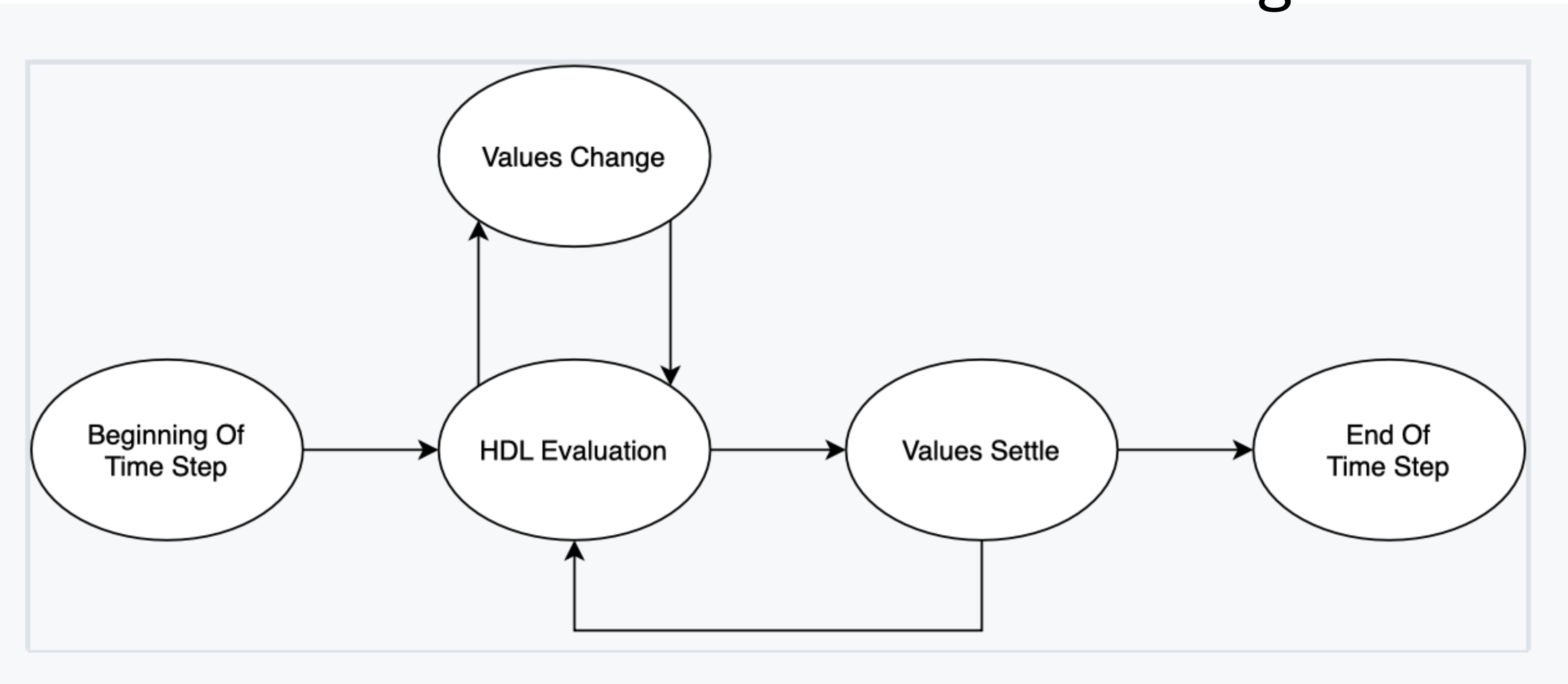
`cbReadWriteSynch`

`cbReadOnlySynch`



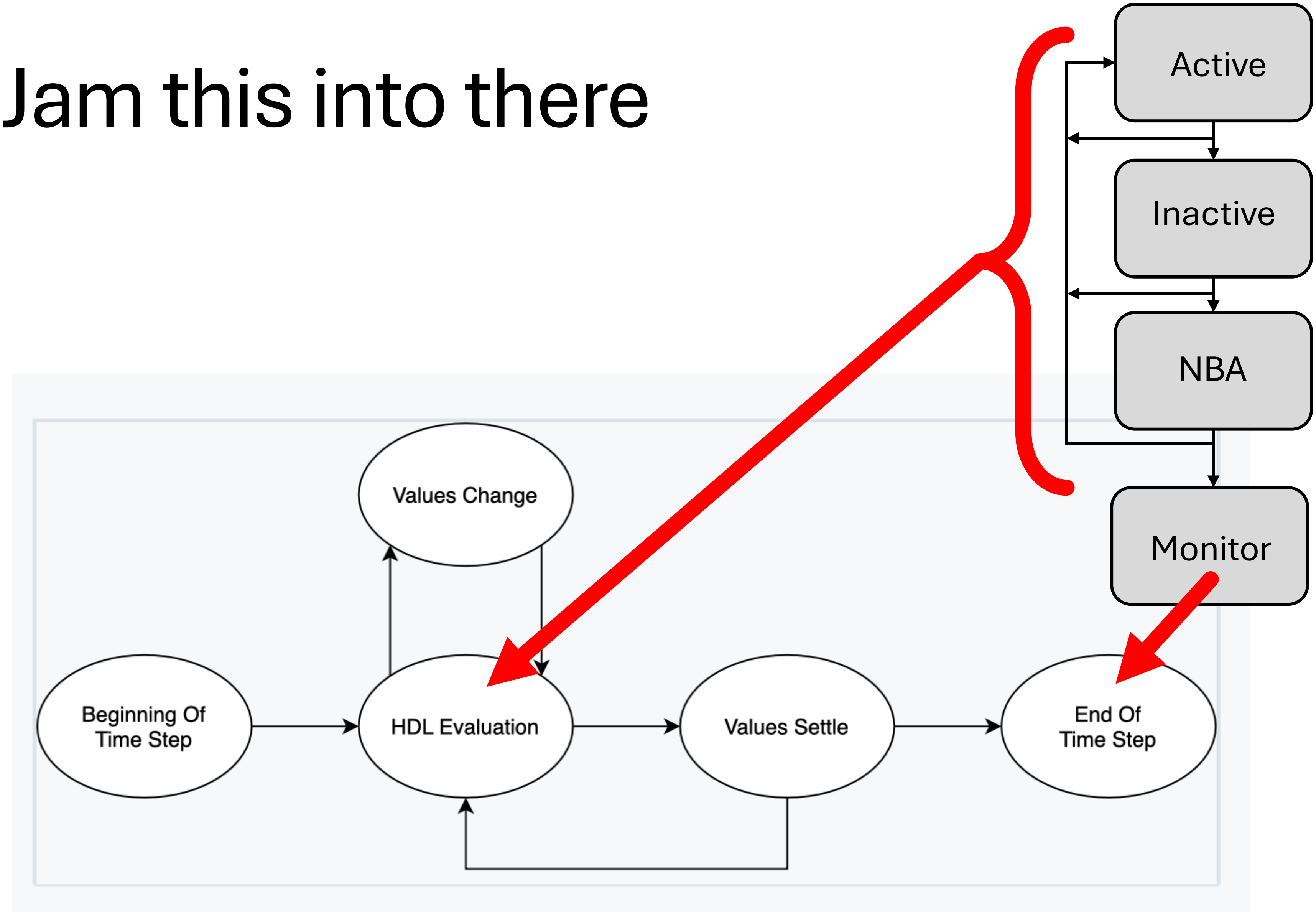
CocoTb's Official^{TM} Timing Model

- Cocotb uses these callbacks to integrate a layer of control around the standard Verilog Simulation Construct
- The Result is their Own Simulation/Timing Model



https://github.com/cocotb/cocotb/blob/master/docs/source/timing_model.rst

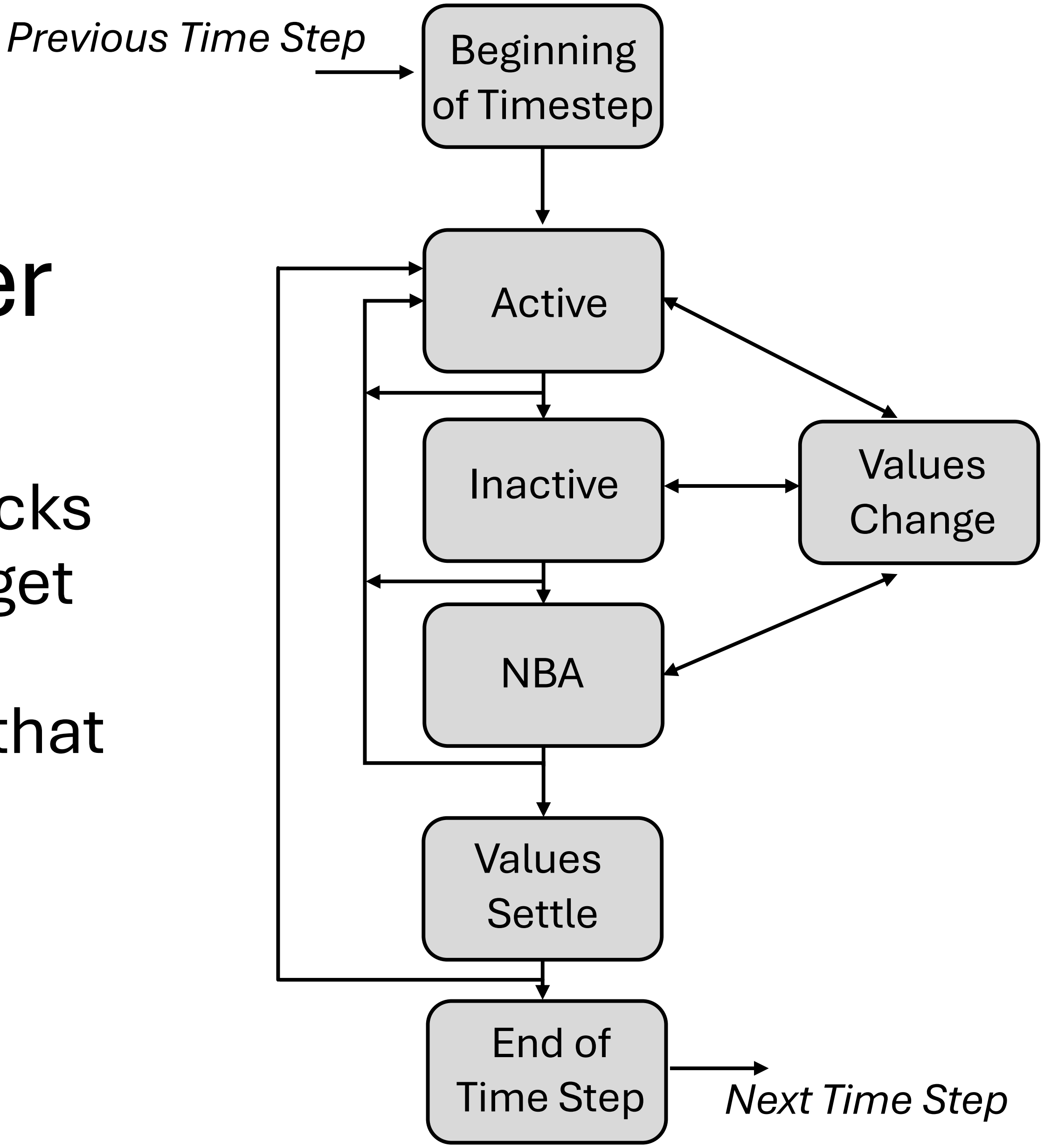
Jam this into there



https://github.com/cocotb/cocotb/blob/master/docs/source/timing_model.rst

Drawing the Two previous slides together

- Using VPI callbacks and Python you get this hybrid simulation step that runs



So in CocoTb when you want to set a Callback…

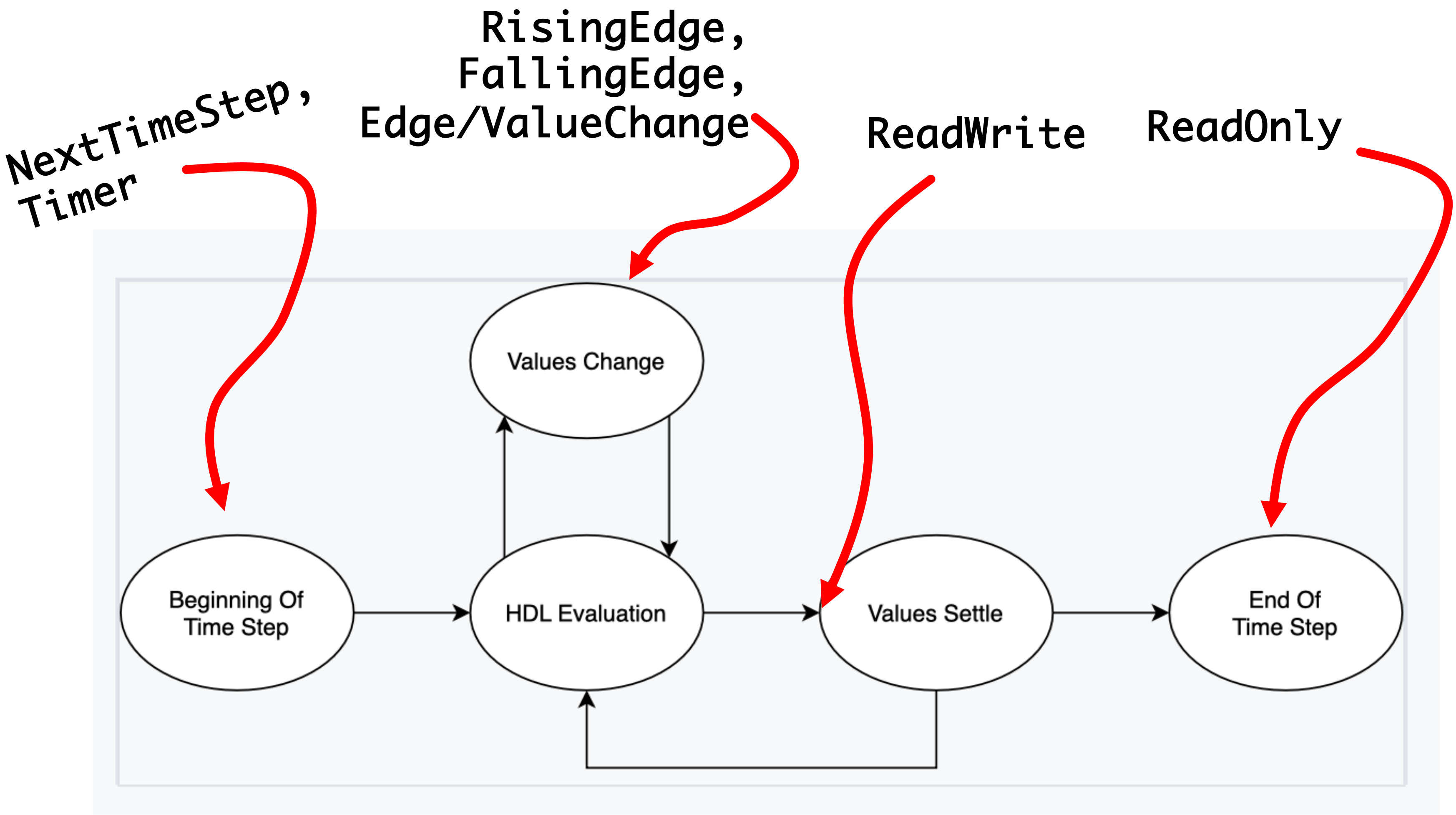
- You’re going to do that through issuing ***await***

- In Cocotb when you issue an await, what you’re doing is throwing control to the underlying Verilog simulator and it will then return control back to you when it hits that event and the await returns

List of Events to await for

- `NextTimeStep`
- `Timer`
- `RisingEdge`, `FallingEdge`, `Edge` (soon to be `ValueChange`)
- `ReadWrite`
- `ReadOnly`
- These all have almost 1:1 mappings to VPI callbacks

CocoTb's Timing Model



NextTimeStep, Timer

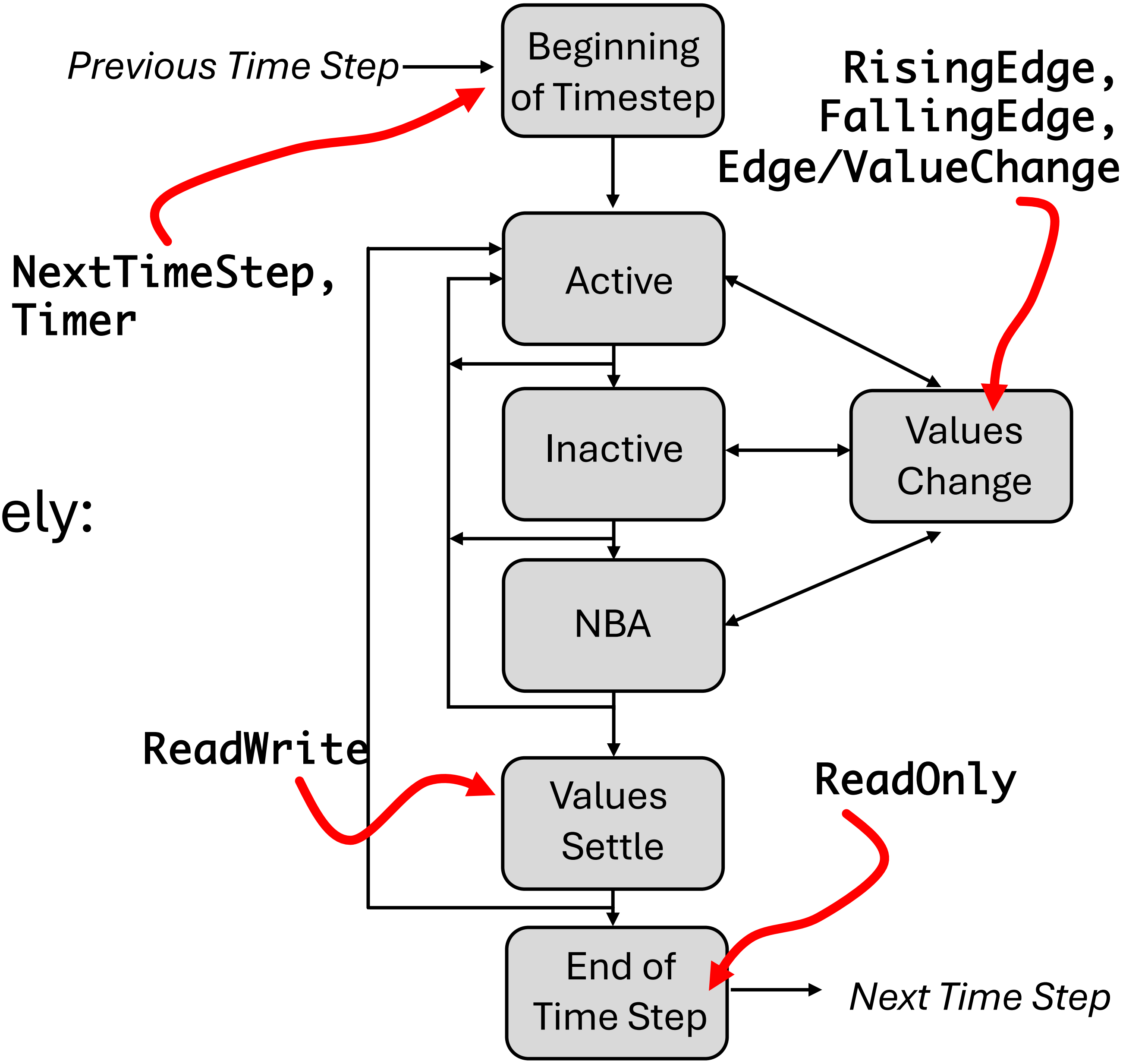
RisingEdge, FallingEdge, Edge/ValueChange

ReadWrite

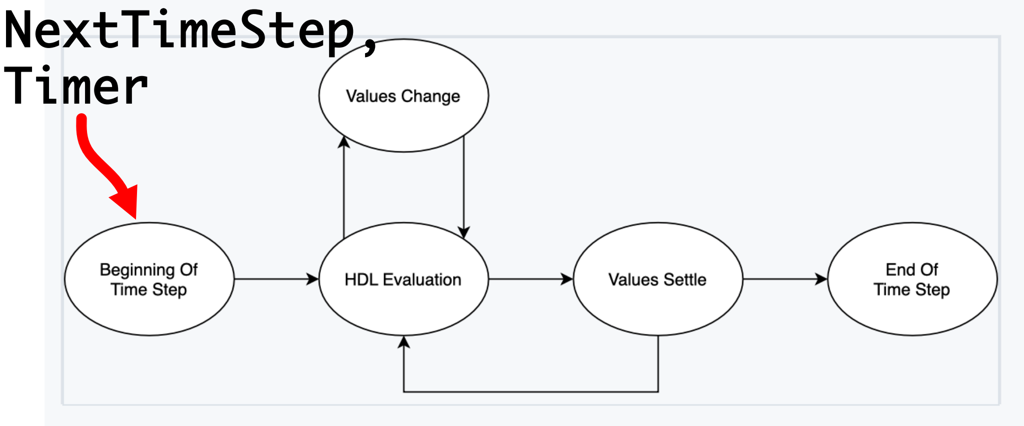
ReadOnly

https://github.com/cocotb/cocotb/blob/master/docs/source/timing_model.rst

- Or alternatively:



Beginning of TimeStep



- Start of a step.
- You get here by awaiting a `Timer` or awaiting the `NextTimeStep` the simulator encounters!
- You can Read or Write values here.
- When `await Timer(10, "ns")` returns, you're in the beginning of the timestep.

All Value Assignments in Cocotb

- All Value Assignments made through Cocotb are *non-blocking* and do not apply immediately!
- Run this code:

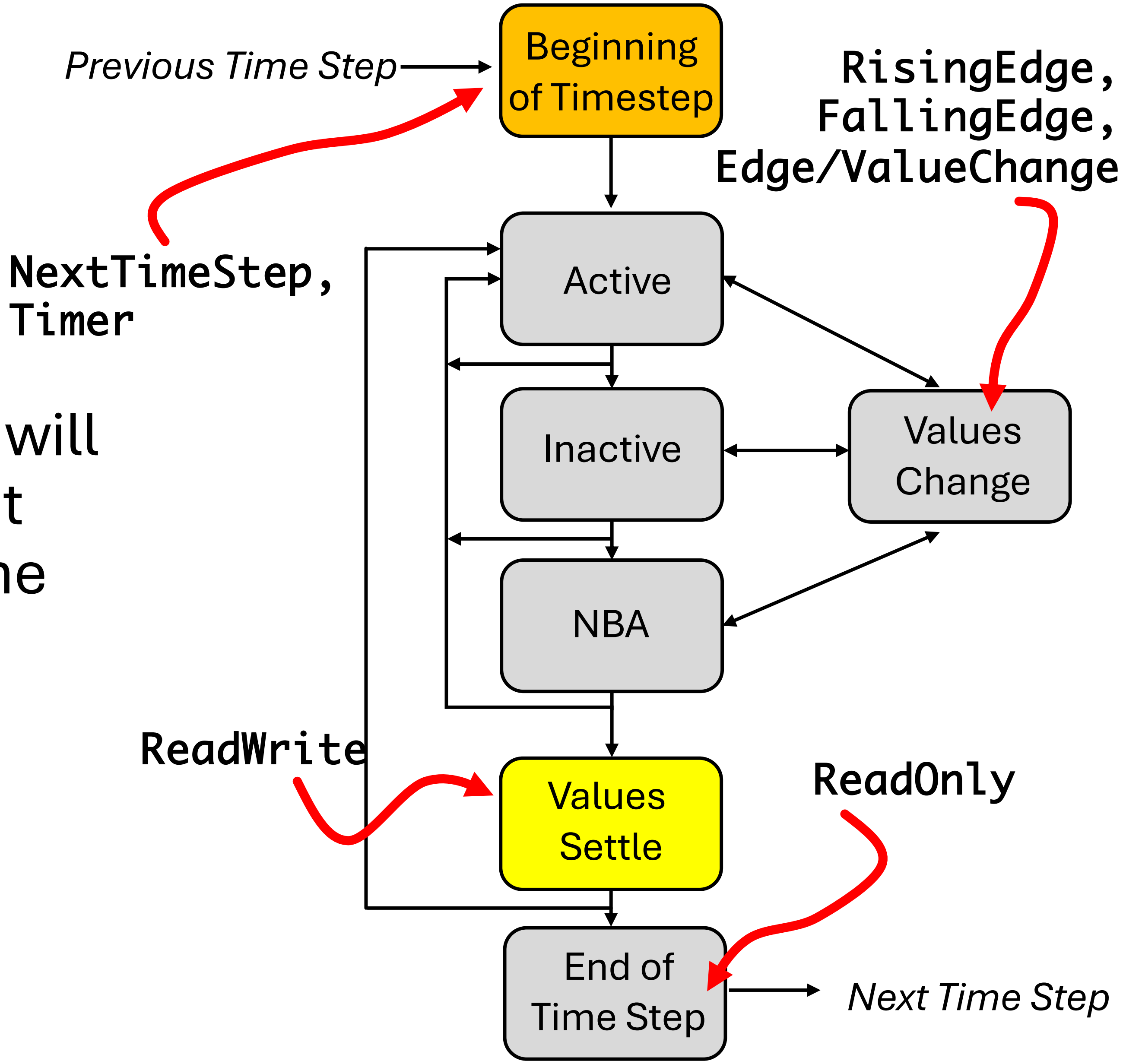
```
await Timer(10, "ns") #in start of timestep:
dut._log.info(f"clk: {dut.clk.value} en:{dut.en.value} count: {dut.count.value.integer}")
dut.en.value = 1 #set en to be 1...now check its value...
dut._log.info(f"clk: {dut.clk.value} en:{dut.en.value} count: {dut.count.value.integer}")
```

- Will return this:

```
20.00ns INFO     cocotb.simple_logic2                        clk: 1 en:0 count: 0
20.00ns INFO     cocotb.simple_logic2                        clk: 1 en:0 count: 0
```

Delayed Write

- Values written will not actually get applied until the **Values Settle** state



One Exception

- In simulation states where you can write the following will get queued up:

```
dut.some_signal.value = 1
```

- However, you can override this and set immediately:

```
dut.some_signal.setimmediatevalue(1)
```

- This will immediately change the value in the simulation.
- Be very very careful…only useful for initial conditions (maybe)…can cause race conditions and/or non-determinism

All Value Assignments in Cocotb

- If I change that code from the previous page:

```
await Timer(10, "ns") #in start of timestep:
dut._log.info(f"clk: {dut.clk.value} en:{dut.en.value} count: {dut.count.value.integer}")
dut.en.setimmediatevalue(1) #set en to be 1...now check its value...
dut._log.info(f"clk: {dut.clk.value} en:{dut.en.value} count: {dut.count.value.integer}")
```

- Will return this:

```
~~20.00ns INFO     cocotb.simple_                          clk: 1 en:0 count: 0~~
~~20.00ns INFO     cocotb.simple_l                         clk: 1 en:1 count: 0~~
```

JK it didn't do this.
the simulation hangs forever and I'm not sure why.
Must have introduced a race condition iunno

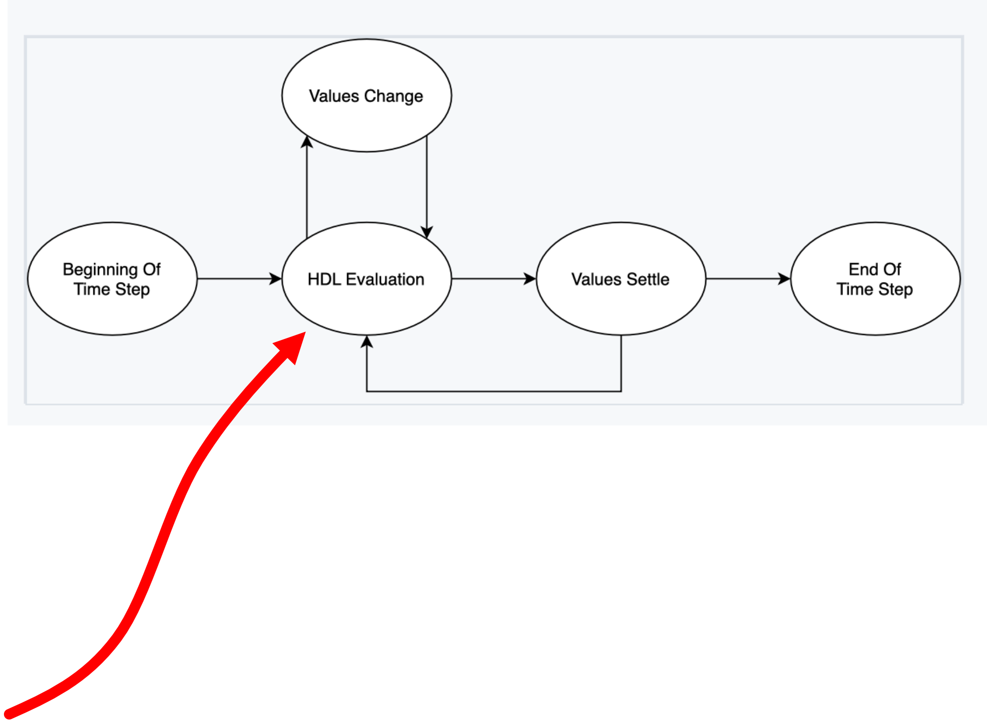
NextTimeStep()!

- It will not just jump 1ns or 1ps forward
- SystemVerilog/Verilog are not simulating useless time in between events. There is a schedule of events and it will jump to the next even that is scheduled to happen.
- In many clocked systems (in simulation), after everything has settled on the rising edge of a clock, the next even will likely be the falling edge.

When Done with Beginning…

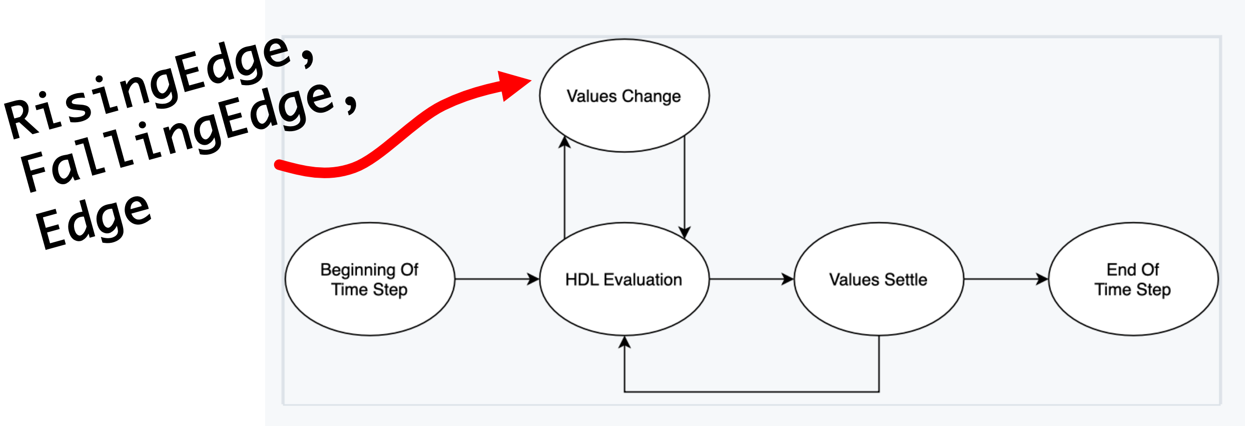
- Move on…can never go back (within this timestep)

HDL Evaluation Step

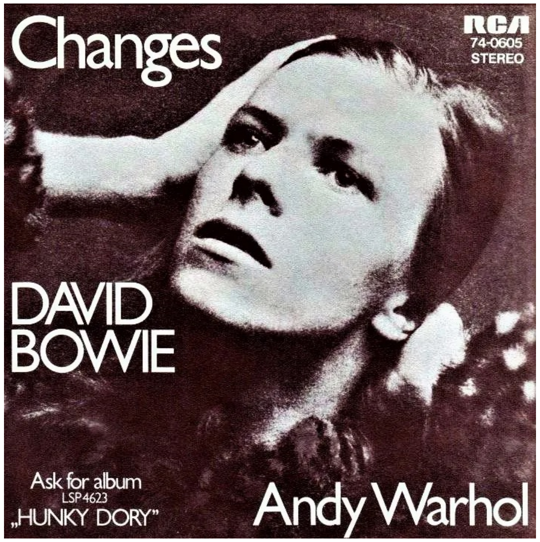


- No way to get direct insight into the progress of the HDL evaluation (vvp engine). That will run until resolved and values settle…
- *Or* until it encounters a change callback

Changes

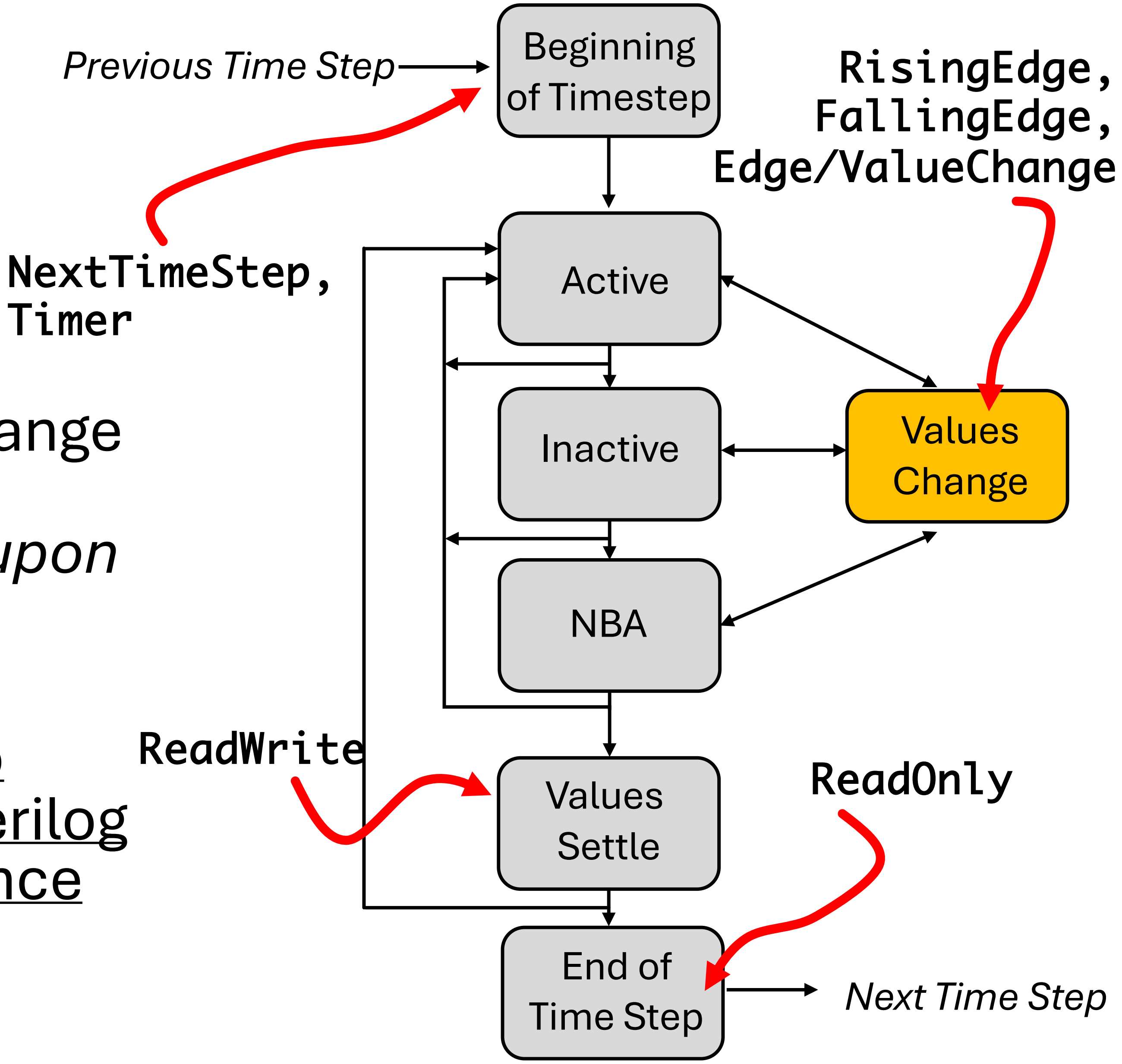

RisingEdge, FallingEdge, Edge

- During HDL Evaluation if a signal with a callback on it changes appropriately, the callback fires and control returns to Cocotb/Python
- You can do `RisingEdge`, `FallingEdge`, and `Edge`
- `Edge` will be renamed `ValueChange` in Cocotb 2.0

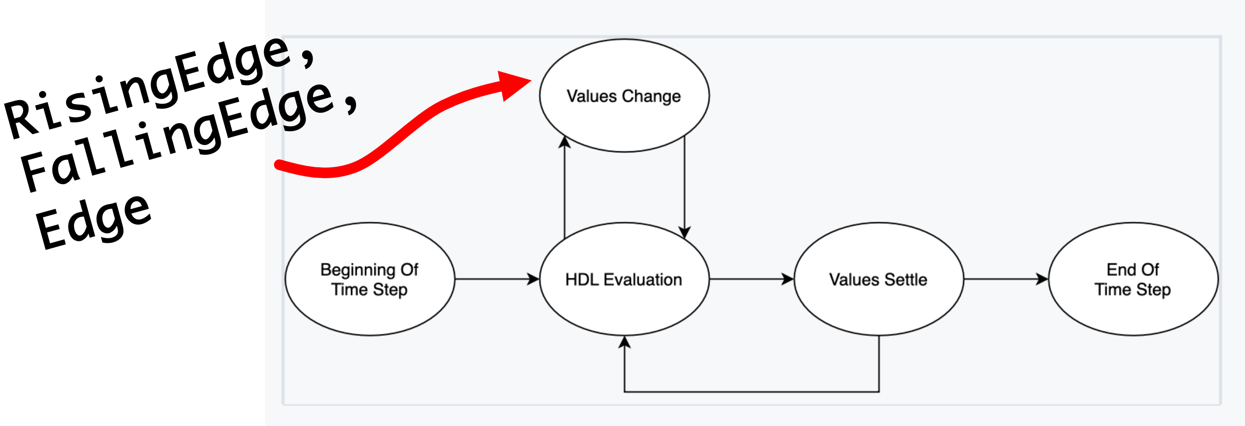


Values Change

- The Values Change will happen ***<u>immediately</u>*** *upon the event happening.*
- <u>That means no other line of Verilog has had a chance to react to it!</u>



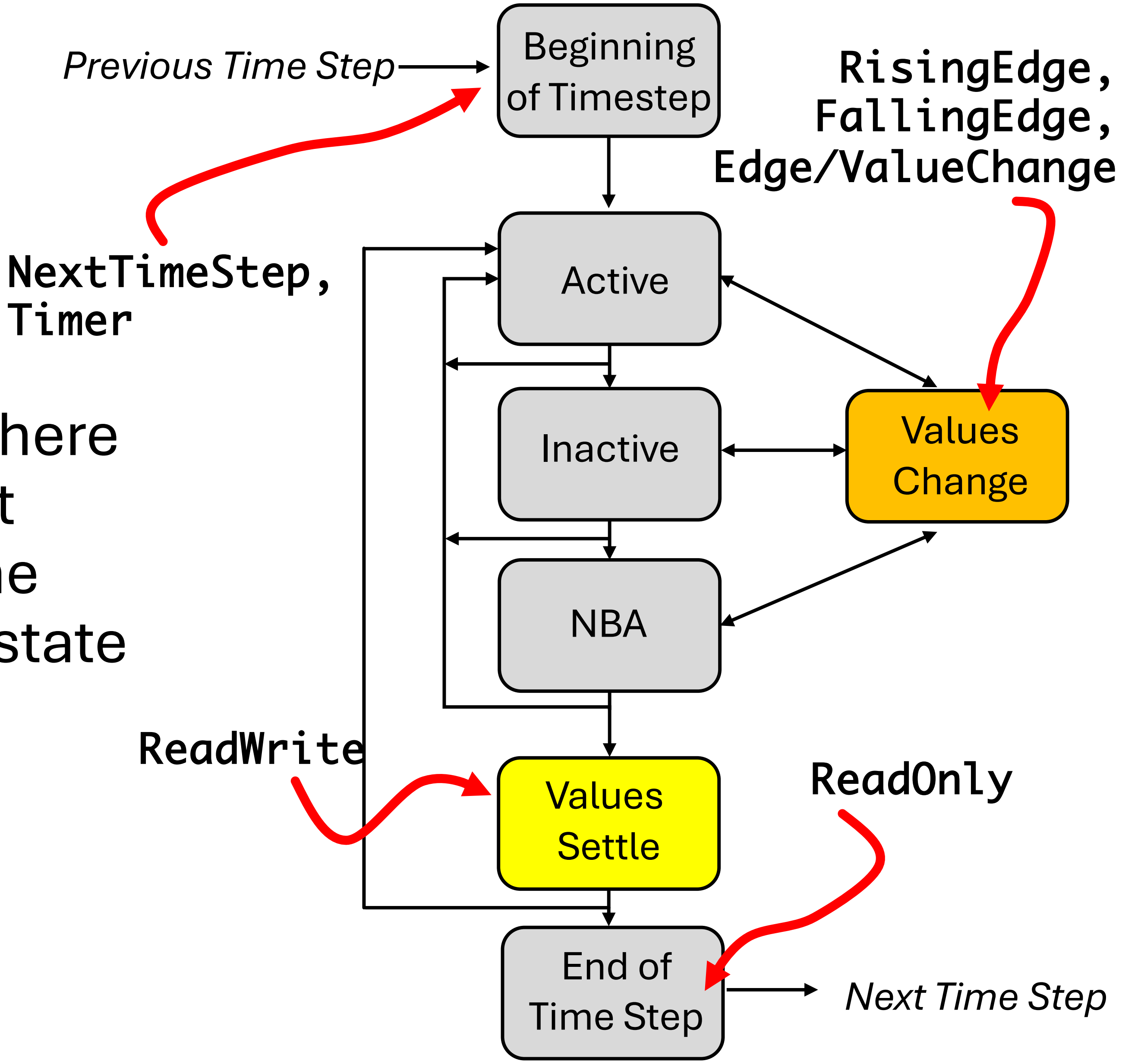
Value Change


RisingEdge, FallingEdge, Edge

- You can *read* and *write* signals in the Values Change state.
- However, the state of values may be non-deterministic here.
- BUT what you read here was unaffected by whatever change brought you in…

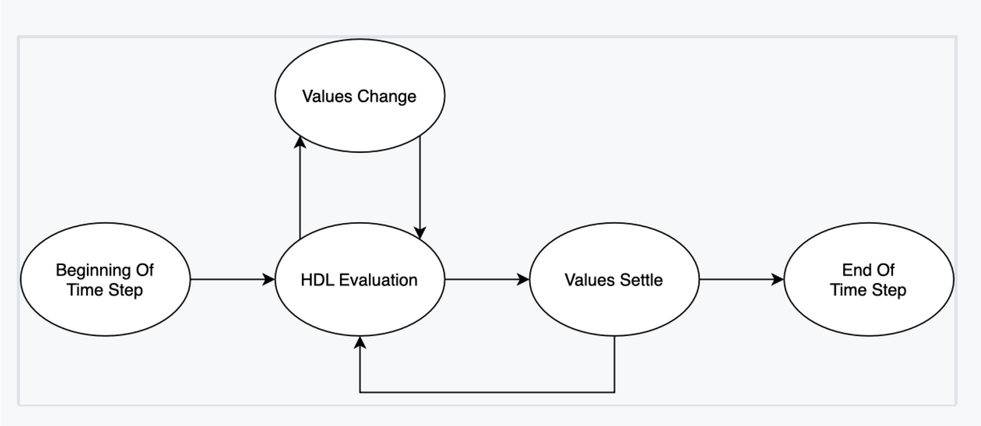
Delayed Write

- Values written here not actually get applied until the **Values Settle** state

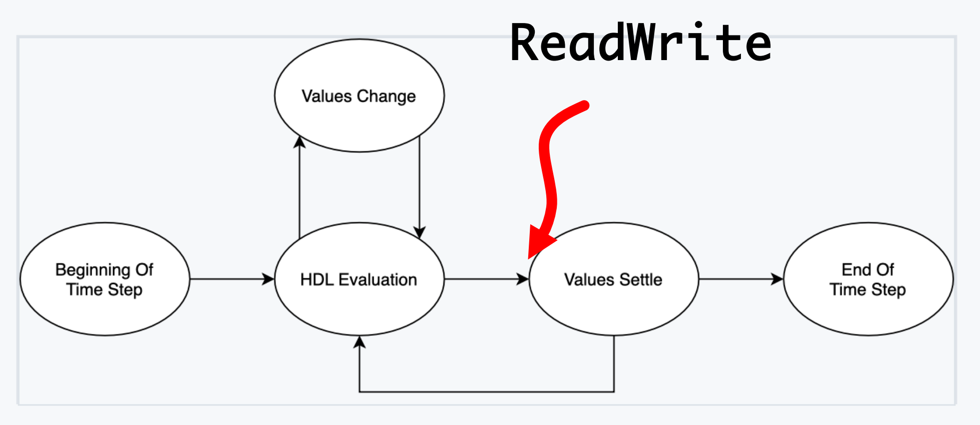


Delta Steps Happen

- When HDL Evaluation is done and no more Values Change, simulation is stable.
- Moves to Values Settle

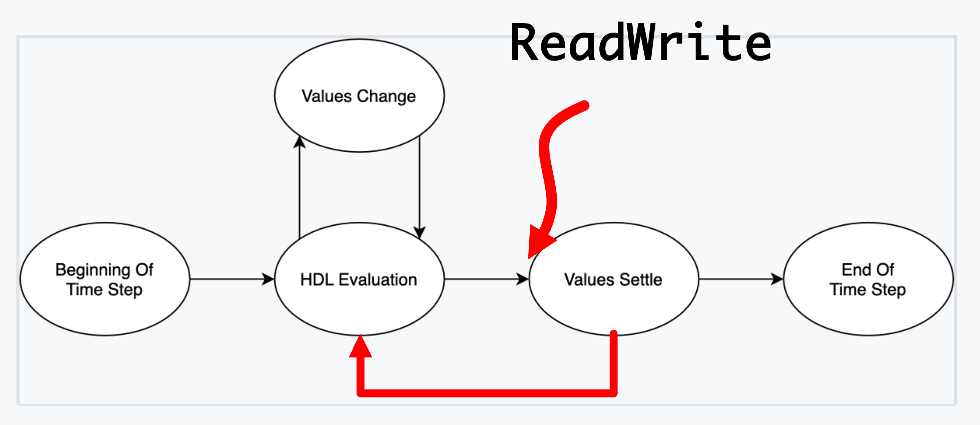


Values Settle



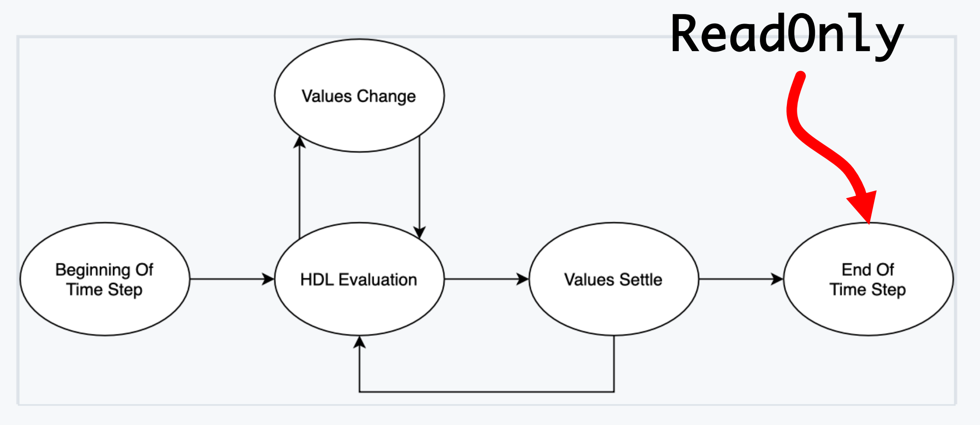
- awaiting ReadWrite puts you in this point of your simulation and pauses things.
- Everything, including non-blocking assignments internal to the DUT will have resolved.
- External signals (*specified from Cocotb*) will not have been applied yet!
- You can read/write values here.
  - Reads are stable
  - Writes after you're done here and sim moves on.

Values Settle



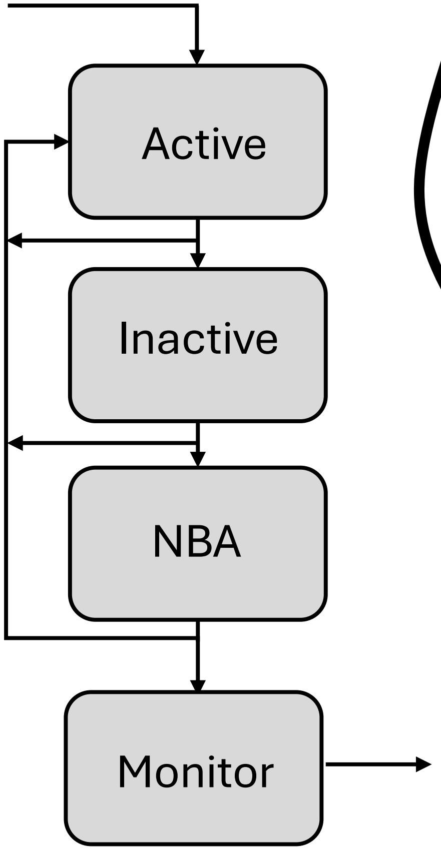
- You can read/write values
- When the simulator continues, those delayed writes from Cocotb will be applied
- The Verilog simulator will determine if it needs to revisit the HDL evaluation stage
- If not, it goes to `EndofTimeStep`

EndOfTimeStep



- Effectively the original Monitor Phase
- This is the only state/phase where you can only Read signals. Writing is not allowed since you have already passed the spot in the timestep where you can apply external signals (Values Settle)
- Can’t go back.
- `await ReadOnly()` gets you here.

Original:

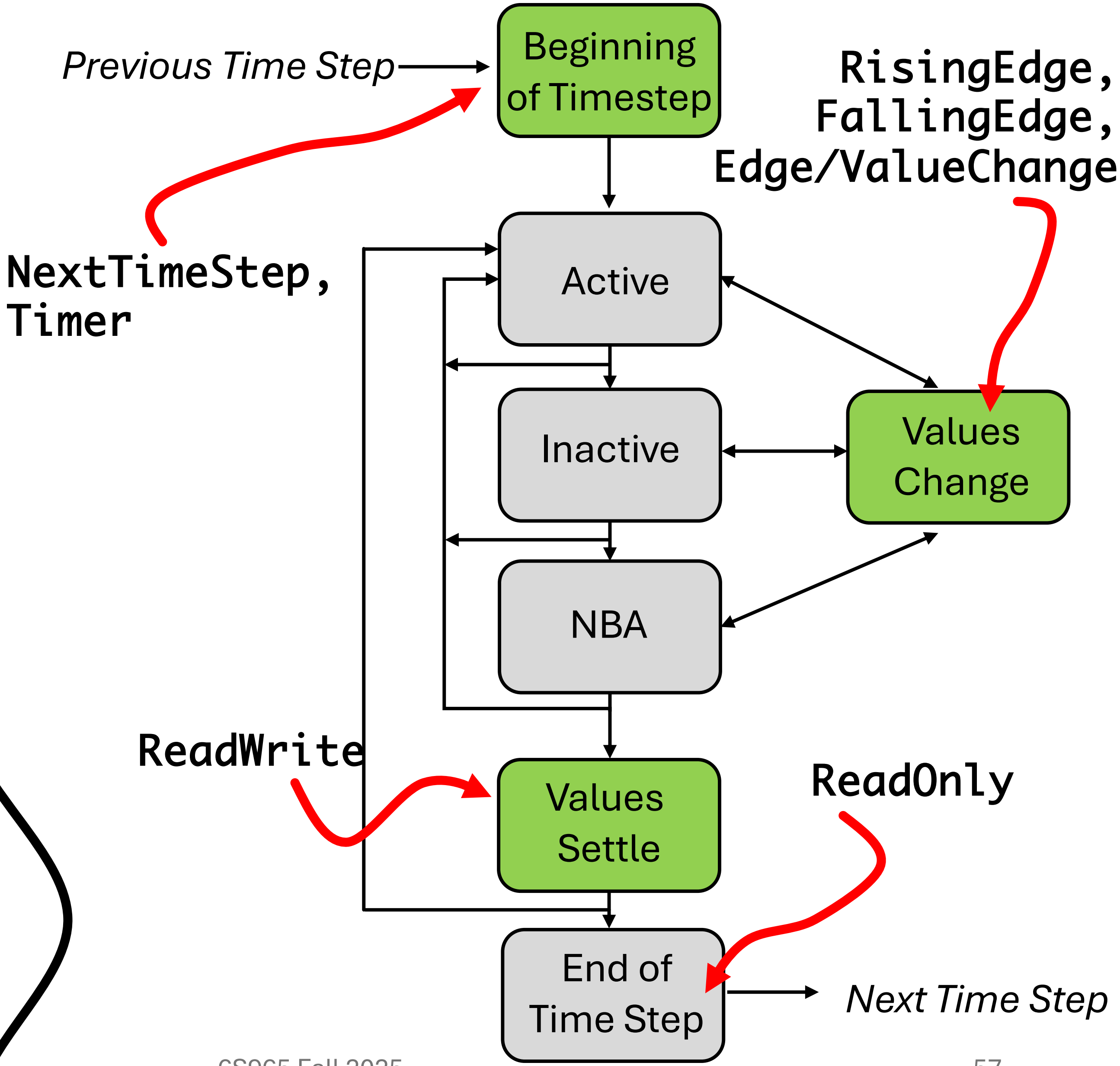

Active

Inactive

NBA

Monitor

Now with Cocotb:


Previous Time Step

Beginning of Timestep

`NextTimeStep,`
`Timer`

`RisingEdge,`
`FallingEdge,`
`Edge/ValueChange`

Active

Inactive

Values Change

NBA

`ReadWrite`

Values Settle

`ReadOnly`

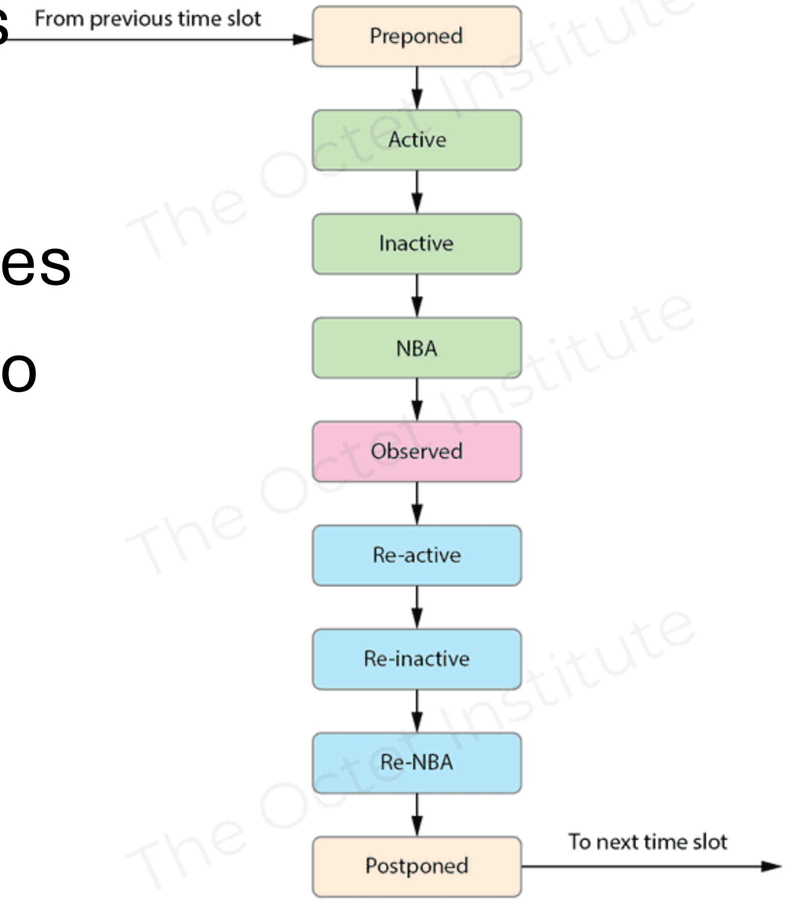
End of Time Step

Next Time Step

Cocotb and SystemVerilog

- Both simulation structures have tried to achieve the same thing and added on many of the same structures
- Clearly broken out region to react and inject external inputs to DUT
- Clear Begin/End Spots

Simplified SV time step (loopbacks not shown)



www.theoctetinstitute.com

Cocotb 2.0 Updates

- The Naming and Everything is Very Confusing
- Cocotb 2.0 is making an effort to:
    - Have the current step of a simulation be more transparent
    - Have the naming be more clear
    - We’ll see how that evolves in coming months/years

Maybe Cocotb Demo

- Try to walk through the simulation trying to mess with it.