# 6.S965
# Digital Systems Laboratory II

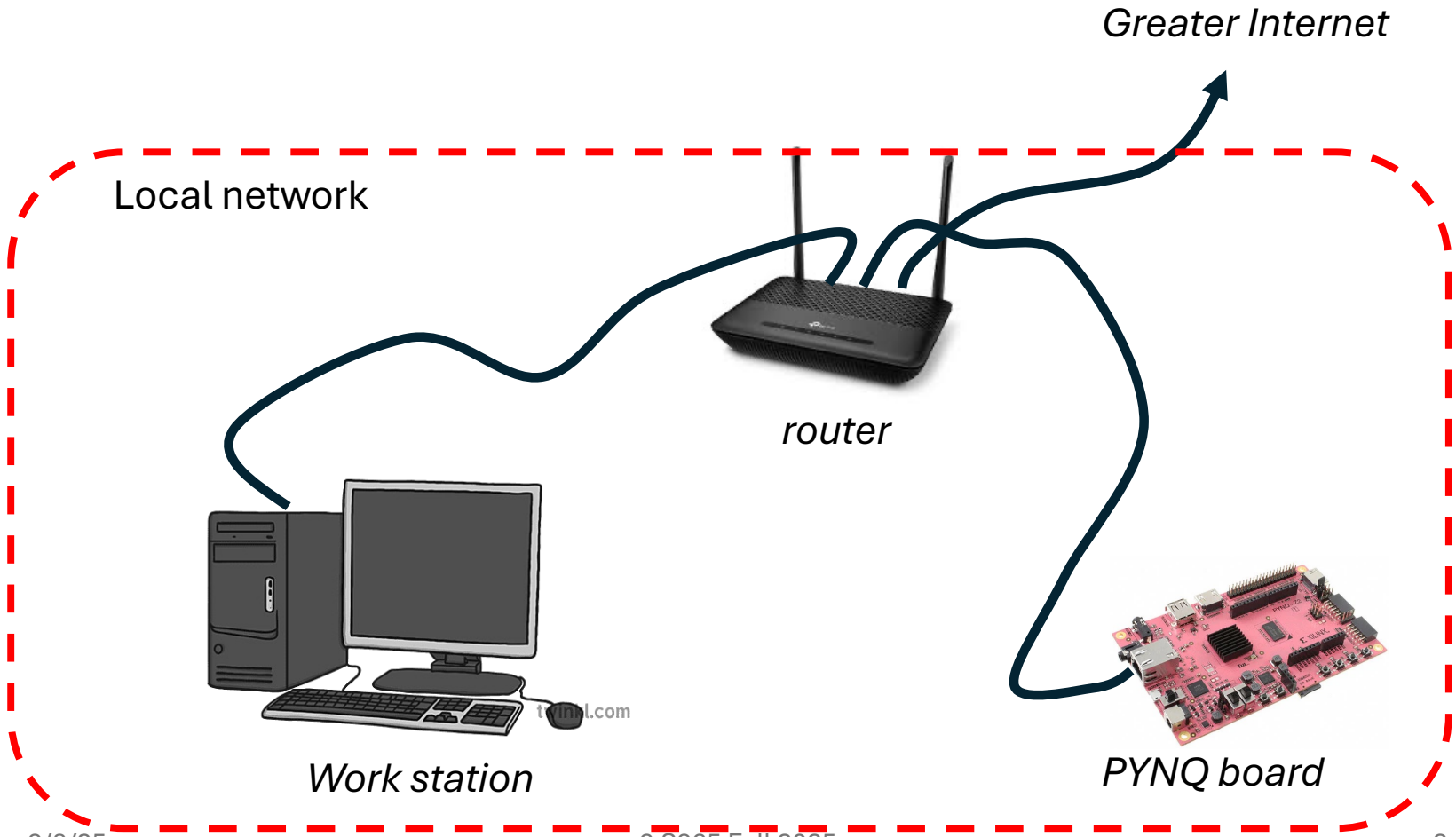
Lecture 2:
Verilog, Simulation, and Cocotb

# Administrative

- Week 1's material is out.  Some students actually already got through it, which means it is doable.
- Lab stations 33 through 46 have user accounts, Pynq boards set up for you.
  - Username: kerb, password: MIT ID number
- Reach out for help on Piazza
- I've thrown some office hours in the calendar for the week too.

# Note on PYNQ Deployment

- Each Lab station set up like this:



# Note on PYNQ Deployment

- Each router has an IP address and login credentials on it.
- In the browser at a work station go to router IP address, log in and see what other devices are on network.
- PYNQ should show up (when powered and booted) and it’ll tell you what IP address it has been assigned.
- Use that that sign into it/transfer files/etc…

# How Does Verilog Actually Work?

- We spent all of 6.205 using Verilog and SystemVerilog to write things, but we never really spent any time thinking about how it actually simulates

- If we're going to spend a lot of time learning how to thinking about simulation at least quasi-formally we should at least.

# Let's look at a relatively simple chunk of code

- Here's some simple SystemVerilog:
- It has a variety of things being done here

```
`timescale 1ns/1ps

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
```

# Compile and Run

- If you just wanted to simulate with icarusVerilog we'd do:

```
iverilog –g2012 –o example.out example_tb.sv example.sv
```

- This would then produce a file we would run with vvp (Verilog Virtual Processor) like so:

```
vvp example.out
```

- Simulation runs…prints, waveforms generated, etc…

# The Fundamental Problem

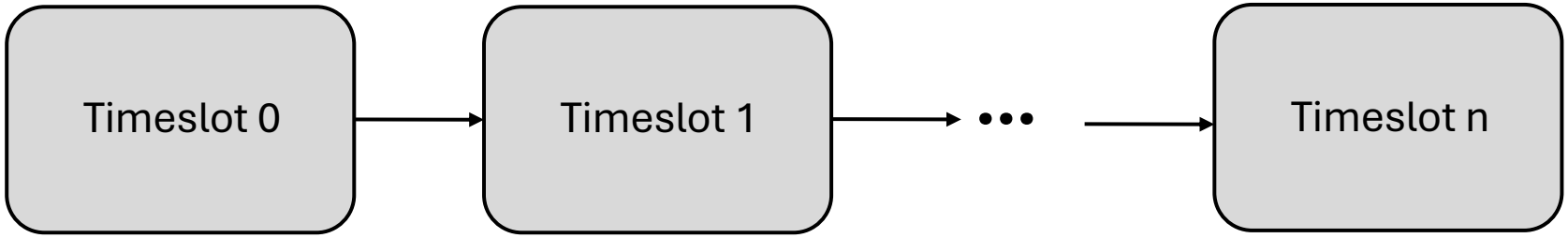
- When we design hardware we're designing systems that work "in parallel" or "at the same time".
- But when we simulate we can't actually do that. Simulation is really just a program that runs one instruction after the other.
- As a result we need to fake the "at-the-same-time" thing.

# The Fundamental Problem II

- What happens if two things are supposed to happen at the same time?
- Which one will get simulated first will be non-deterministic
- That may or may not matter depending on the design.

# 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

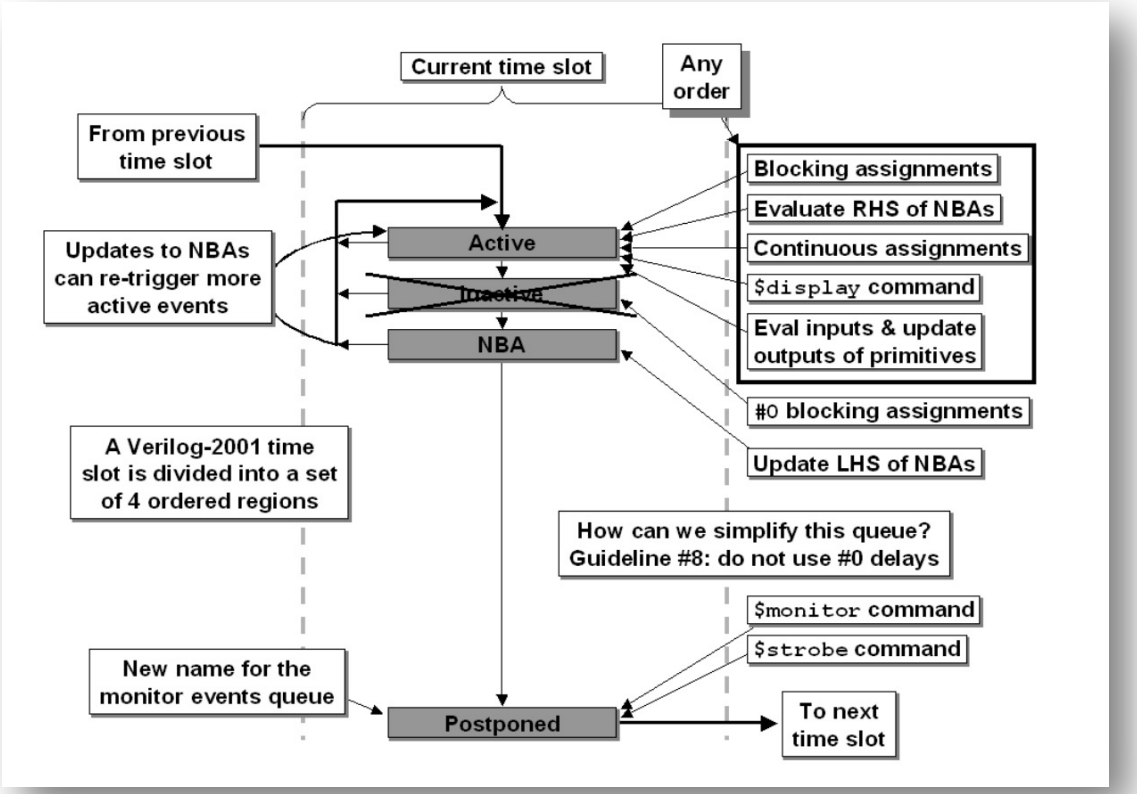



- The size of the simulation timeslot will be based off the timescale specified. For example:
  - `` `timescale 1ns/1ps ``
  - Means we have basically 1ps time step/slot size

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

# The Verilog Simulation Time Step

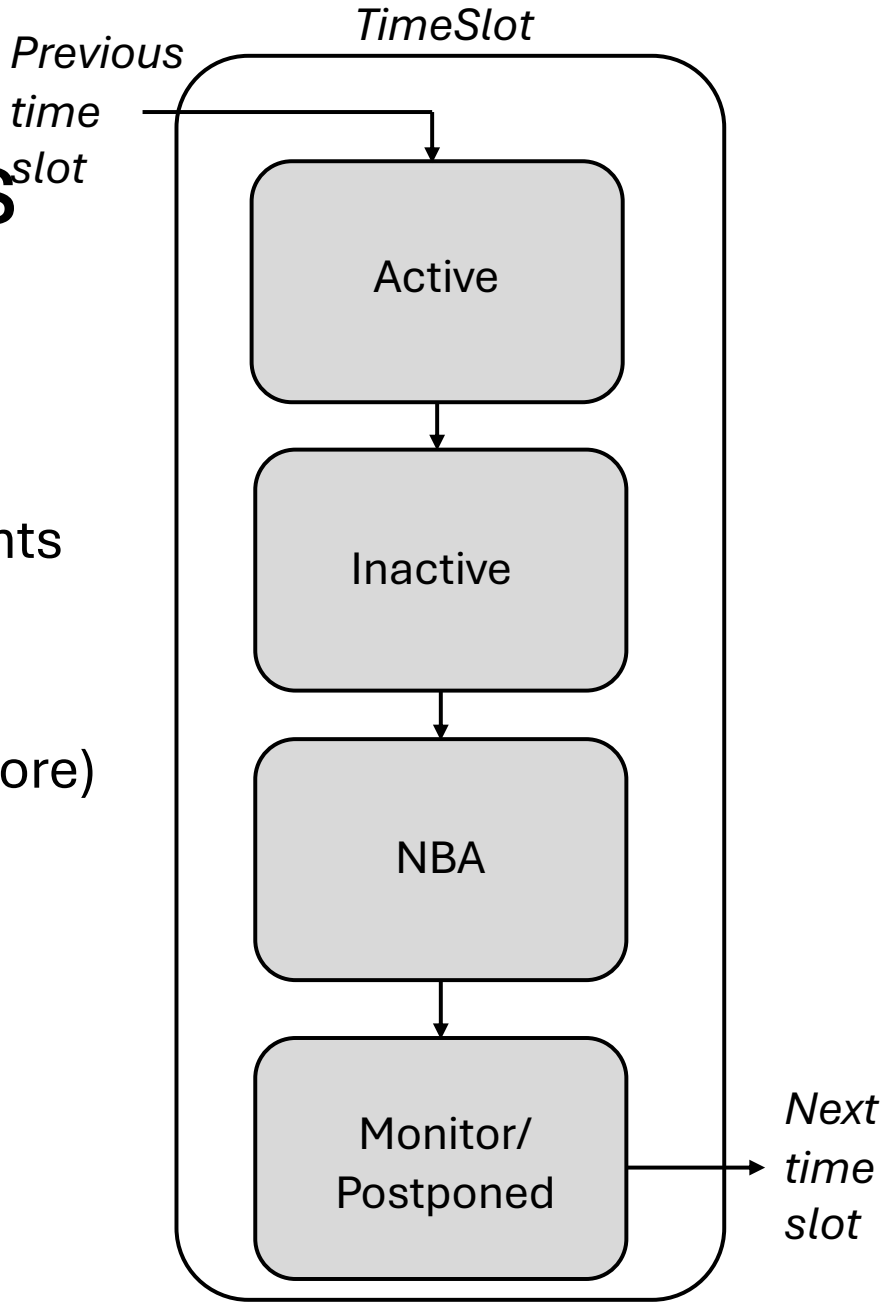
- Within each time slot are regions in which different evaluations and updates are made
- They go through a specific order




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

# Four Main Regions

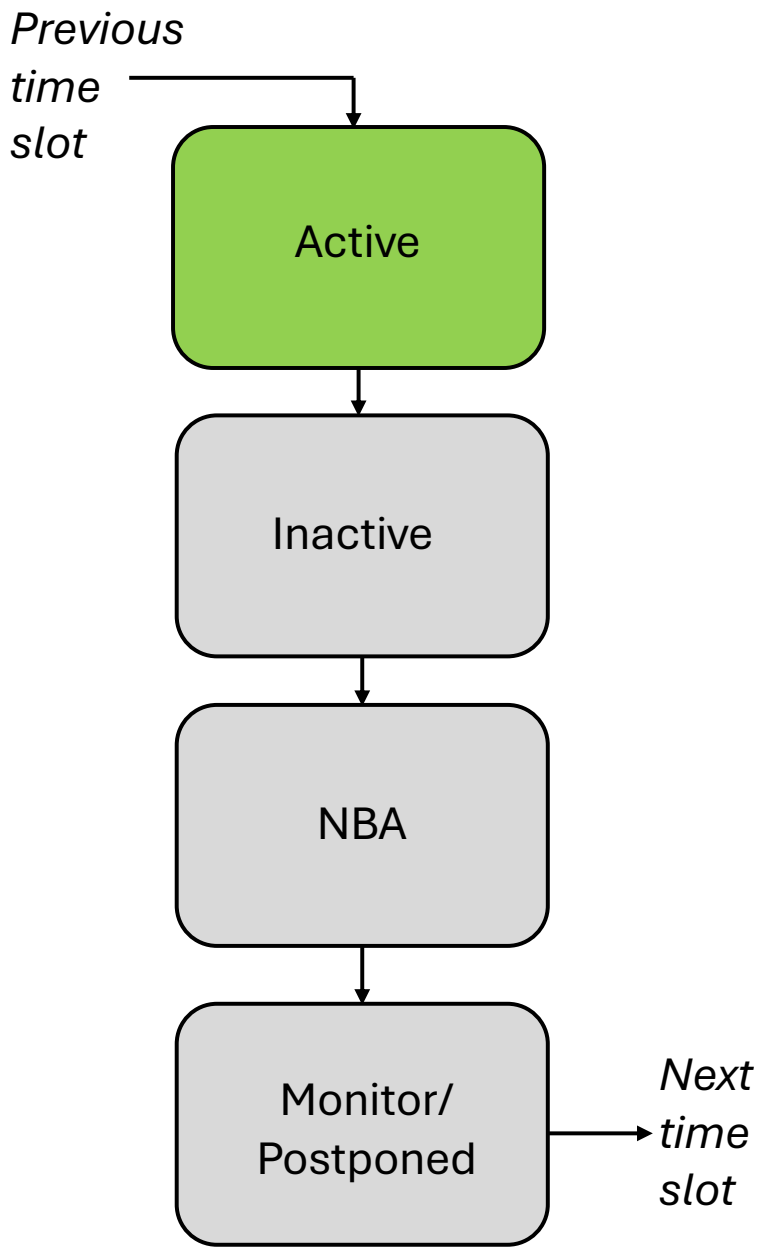
- Active:
  - Blocking Assignments
  - RHS of non-blocking assignments
  - Continuous assignments
- Inactive Region:
  - “#0 Blocking Assignments” (ignore)
- Non-Blocking Region:
  - LHS updating of non-blocking assignments
- Monitor/Postponed Region:
  - Meant for evaluation of results




https://medium.com/@vritvlsi/verilog-event-scheduler-88e5e18e4afd

# Active Region

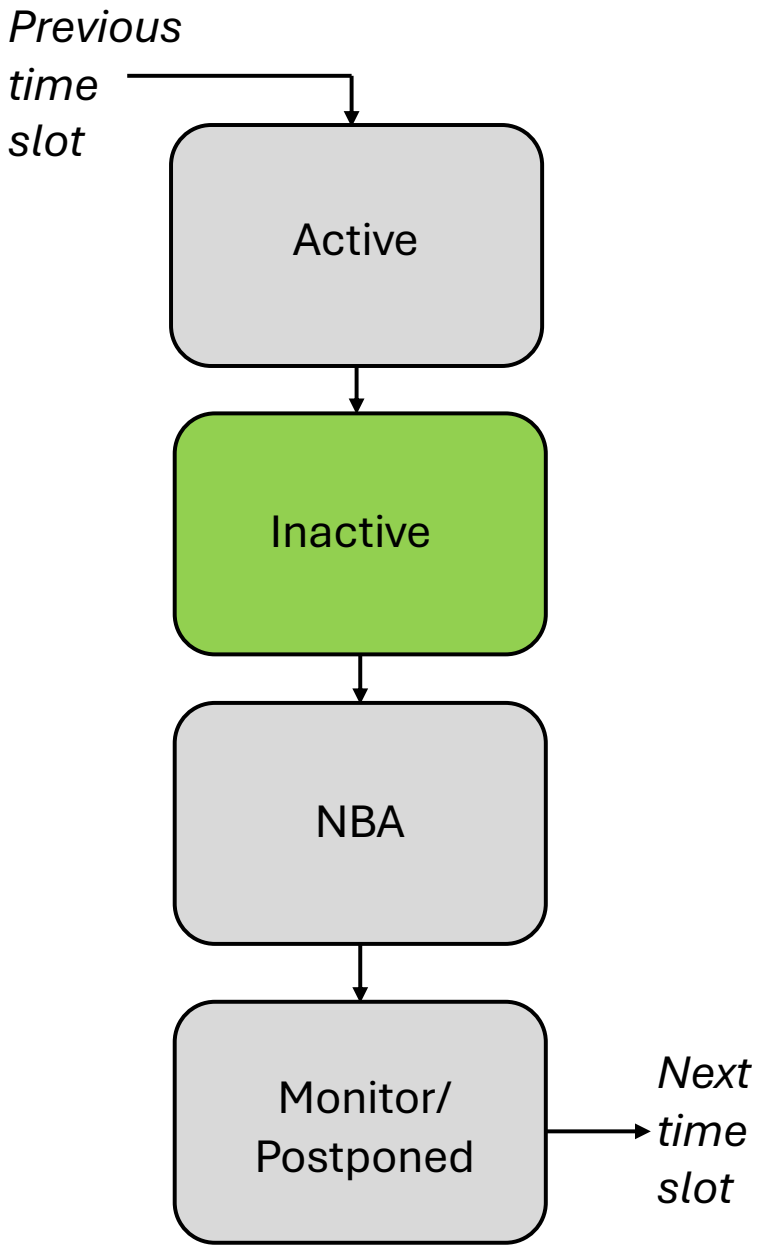
- Blocking Assignments (=)
- RHS of Non-Blocking Assignments (<=)
- Continuous Assignments (assign)
- Primitive/User-Defined Primitives
- $display commands




https://medium.com/@vritvlsi/verilog-event-scheduler-88e5e18e4afd

# Inactive Region

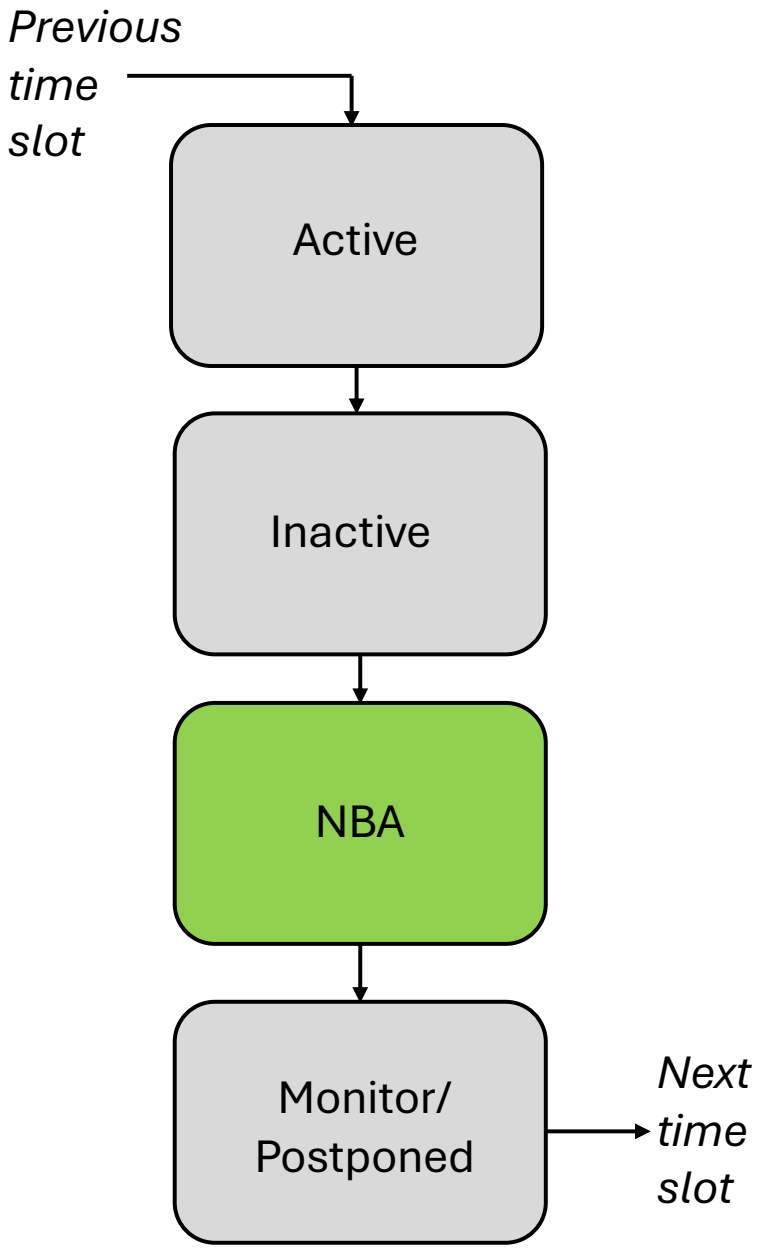
- Blocking Assignments (=) that take place after #0
- Make sense? It shouldn't…everyone always says to ignore this.




https://medium.com/@vritvlsi/verilog-event-scheduler-88e5e18e4afd

# Non-Blocking Assignment

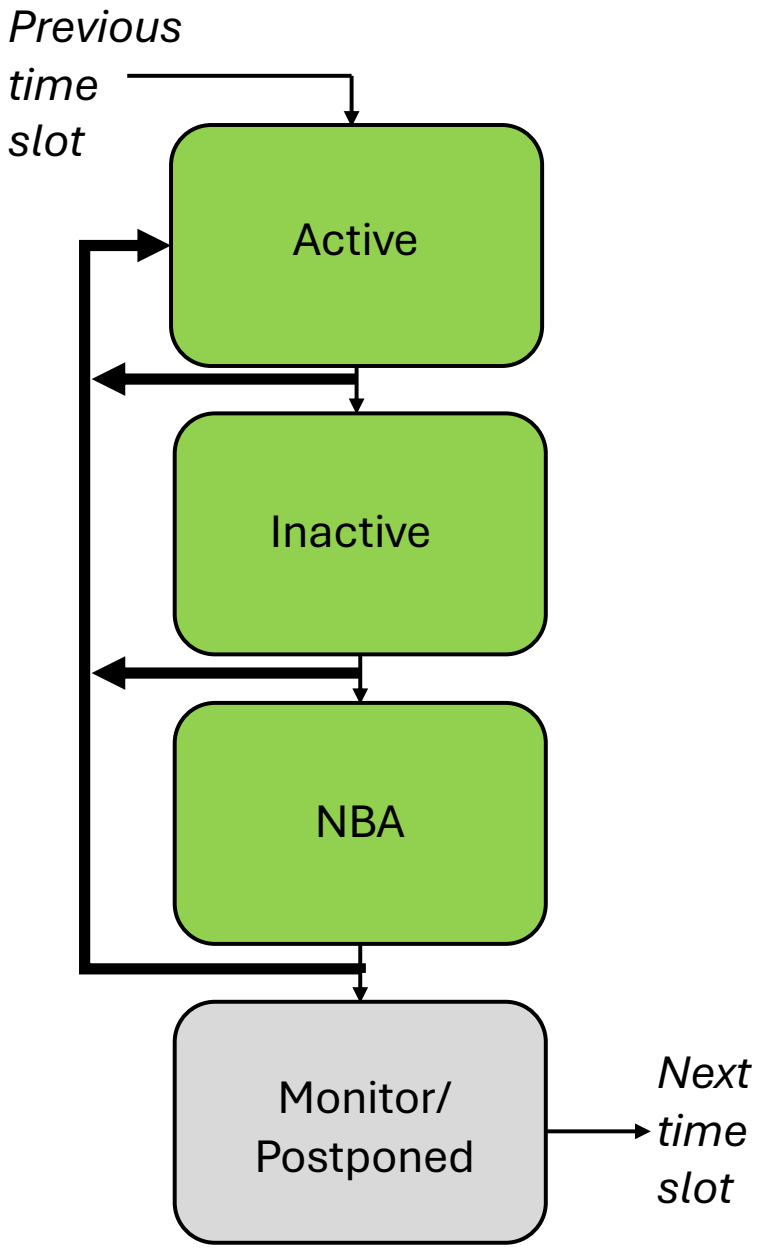
- Updating the LHS of non-blocking assignments




https://medium.com/@vritvlsi/verilog-event-scheduler-88e5e18e4afd

# Go-Back Triggers

- The simulation does not go necessarily go through each stage once
- It monitors the changes to things. If a change in one region means another change should happen, different stages may be restart
- Each restart/iteration is a “delta-step”




*NBA = Non-blocking Assignments

# Return to the code…

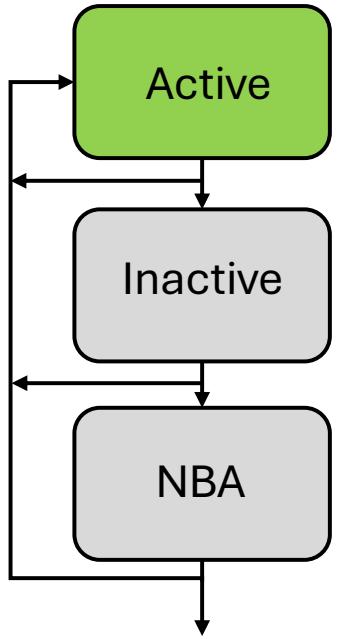
- Here's some simple SystemVerilog:
- It has a variety of things being done here

```
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

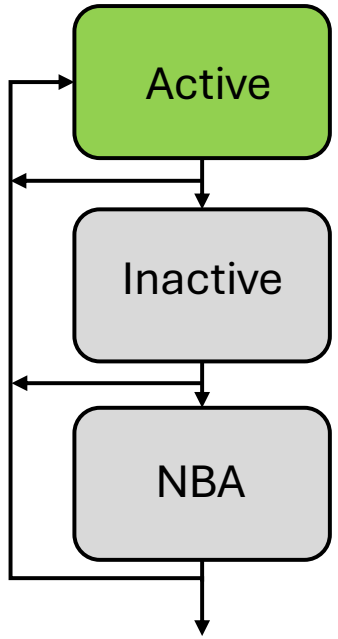



```
logic clk;
logic [1:0] a,b,c,d,e;

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;

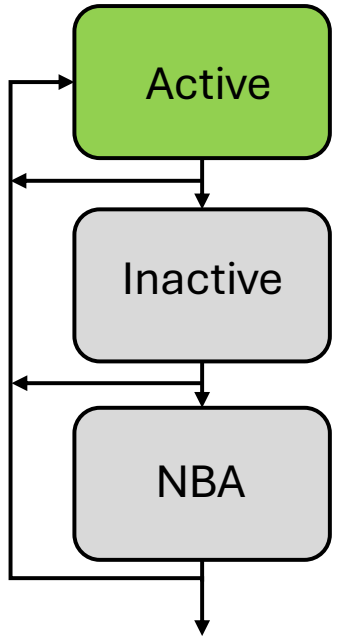
b updated. Depending on the order these lines were evaluated in, the first one might need to run again given the new value of b




```
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;

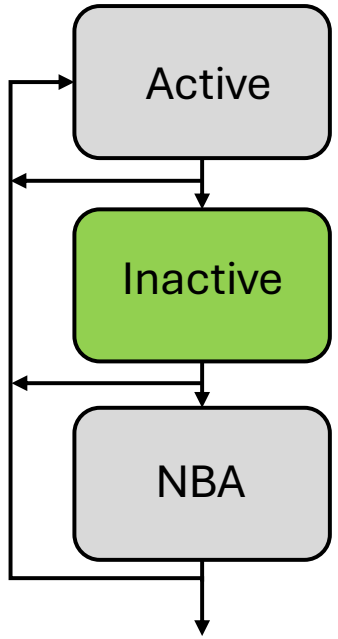



```
logic clk;
logic [1:0] a,b,c,d,e;

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

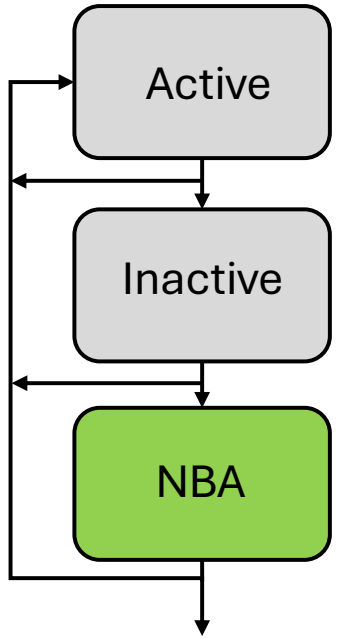
- Highly discourage to use this.
- Just skip this…is a weird delayed region that people use to force order on assignments
- Kinda like `!important` in CSS if anybody does webdev




```
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`



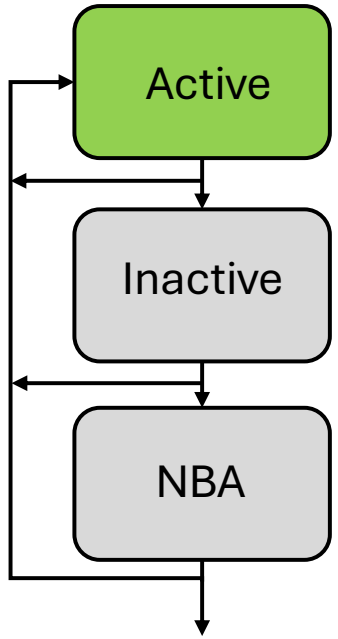

```
logic clk;
logic [1:0] a,b,c,d,e;

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
```

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

# 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

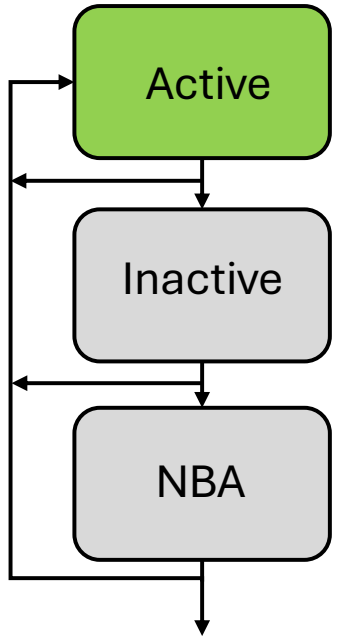



```
logic clk;
logic [1:0] a,b,c,d,e;

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;

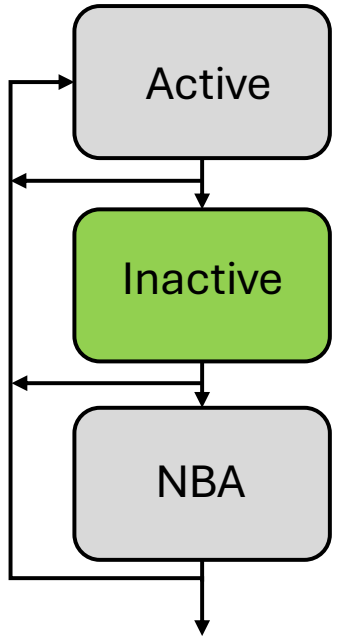
Shouldn't be anything left dangling




```
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

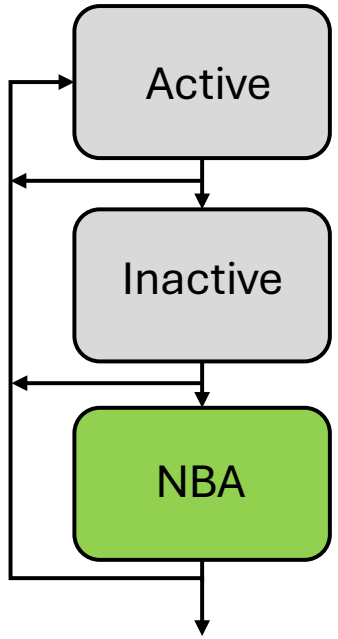
- Highly discourage to use this.
- Just skip this…is a weird delayed region that people use to force order on assignments
- Kinda like `!important` in CSS if anybody does webdev




```
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

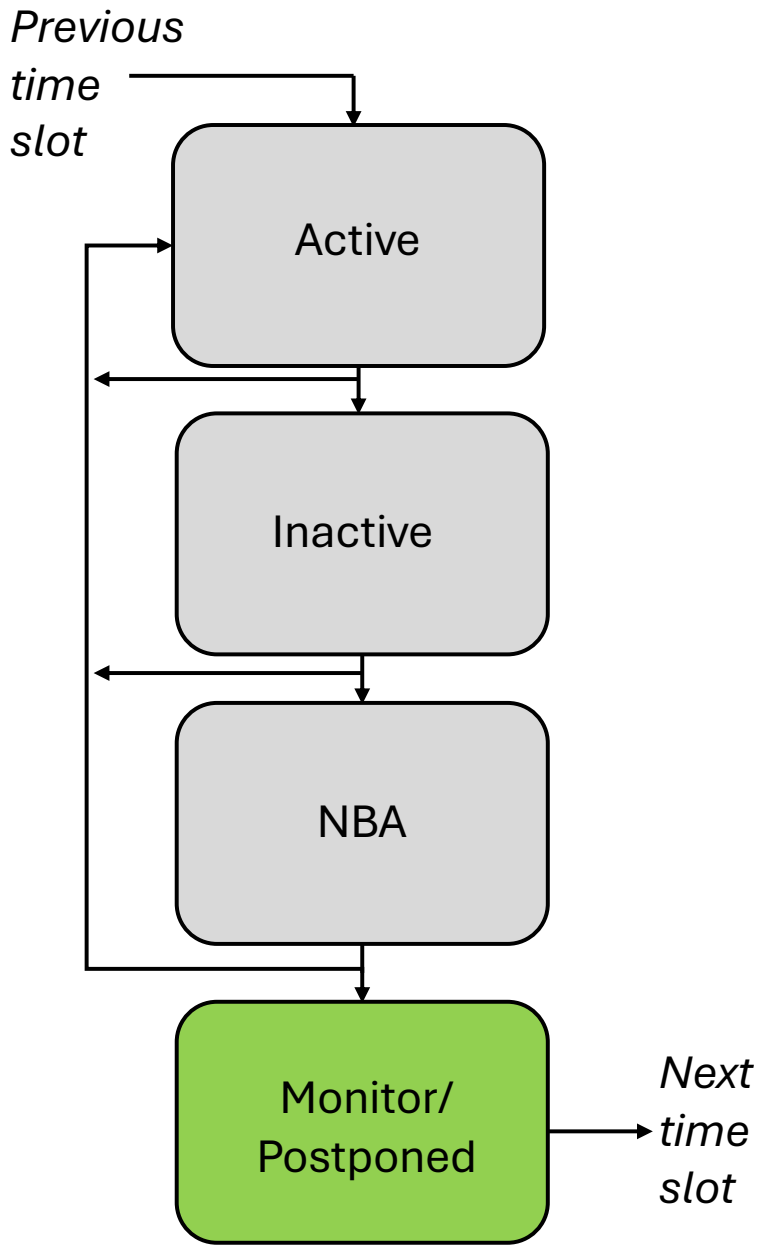
- Nothing new this time through




```
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

# Non-Determinism

- Blocking Lines within an always block will be evaluated in order
- Blocking Lines across multiple always blocks will be analyzed in a non-deterministic fashion
- Might require iterations in stages

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

//fine:
always_comb begin
  b = c + d;
  e = 1+b;
end



//Alternative:
//annoying but will resolve
always_comb begin
  b = c + d;
end

always_comb begin
  e = 1+b;
end
```

# Non-Determinism

- Very possible to have conflicting assignments across multiple blocks
- Non-deterministic which will “win”

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

//not great, but will resolve:
always_comb begin
  b = c + d;
  b = 1 + e;
end

//Alternative:
//bad...actually non-deterministic
always_comb begin
  b = c + d;
end

always_comb begin
  b = 1 + e;
end
```

# Avoid Issues?

- In Verilog, there are ways to end up in non-determinism hell when you have very complicated designs and are lazy with blocking/non-blocking
- The language requires you to follow rules in order for things to work properly.
- Good reading: “Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!” by Clifford E. Cummings Sunburst Design, Inc.

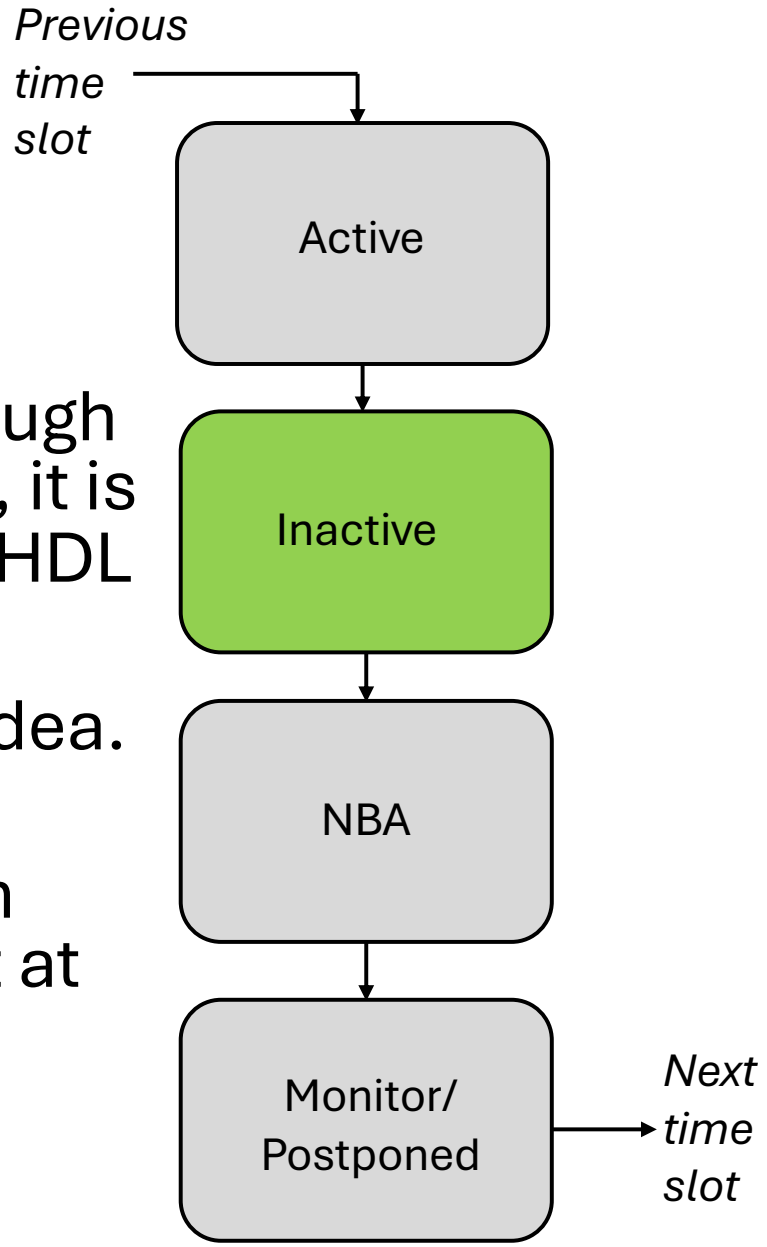
# Bad Fix

- You will sometimes see this is really messed up code where clearly somebody just wanted to clock out and get the thing working

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


//Alternative:
//bad...actually non-deterministic
always_comb begin
  b = c + d;
end

always_comb begin
  #0; //There. fixed it:
  b = 1 + e;
end
```

# How Does #0 ”fix” it?

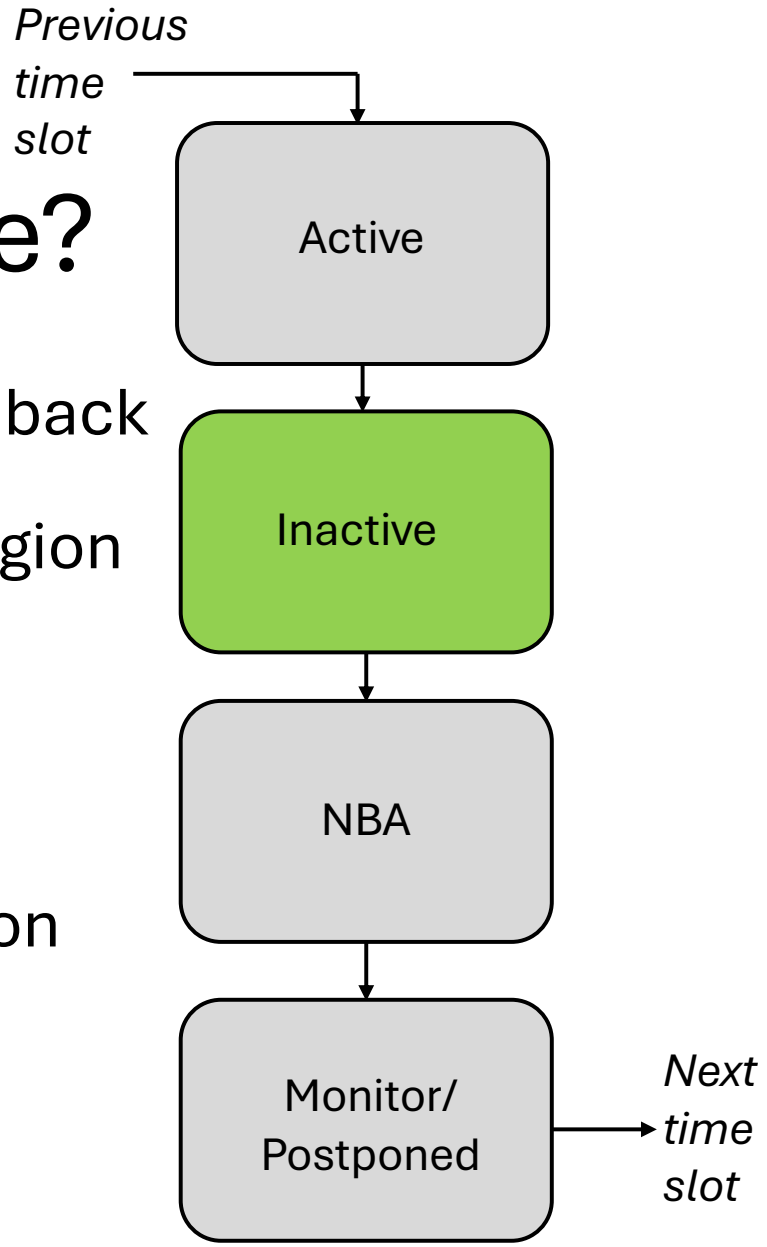
- Since you always have to go through inactive after you’ve done active, it is a way to guarantee that a line of HDL gets the “final say” in step.
- This is rarely, rarely ever a good idea.
- Inactive region dates back to the original Verilog (1984) simulation engine and was an early attempt at easing integration with external interfaces and non-blocking assignments




https://medium.com/@vritvlsi/verilog-event-scheduler-88e5e18e4afd

# Why Even Have Inactive?

- I believe the original inactive dates back to when there was only Active and Inactive regions and the Inactive region was a way to have flip-flops work
- The NBA was brought in later.
- Some Verilog simulators have their external programming hooks in the Inactive region as well (VPI…more on that in a bit).
- And inactive is kept to this day for reverse compatibility.




https://medium.com/@vritvlsi/verilog-event-scheduler-88e5e18e4afd

# Cocotb

- As discussed in Week 1's assignments, for the purposes of observing your design is extremely critical not to be evaluating signals before they resolve (in active/inactive/nba regions).
- Simply grabbing signal value in cocotb is not enough!
- Must wait until you are in the Monitor/Postponed region

# await ReadOnly()

- awaiting the ReadOnly() event in Cocotb allows you to ensure you're only observing signals after they've stabilized and that you're in the monitor/postponed region of any simulation step.

*class* **cocotb.triggers.ReadOnly** [source]

Fires when the current simulation timestep moves to the read-only phase.

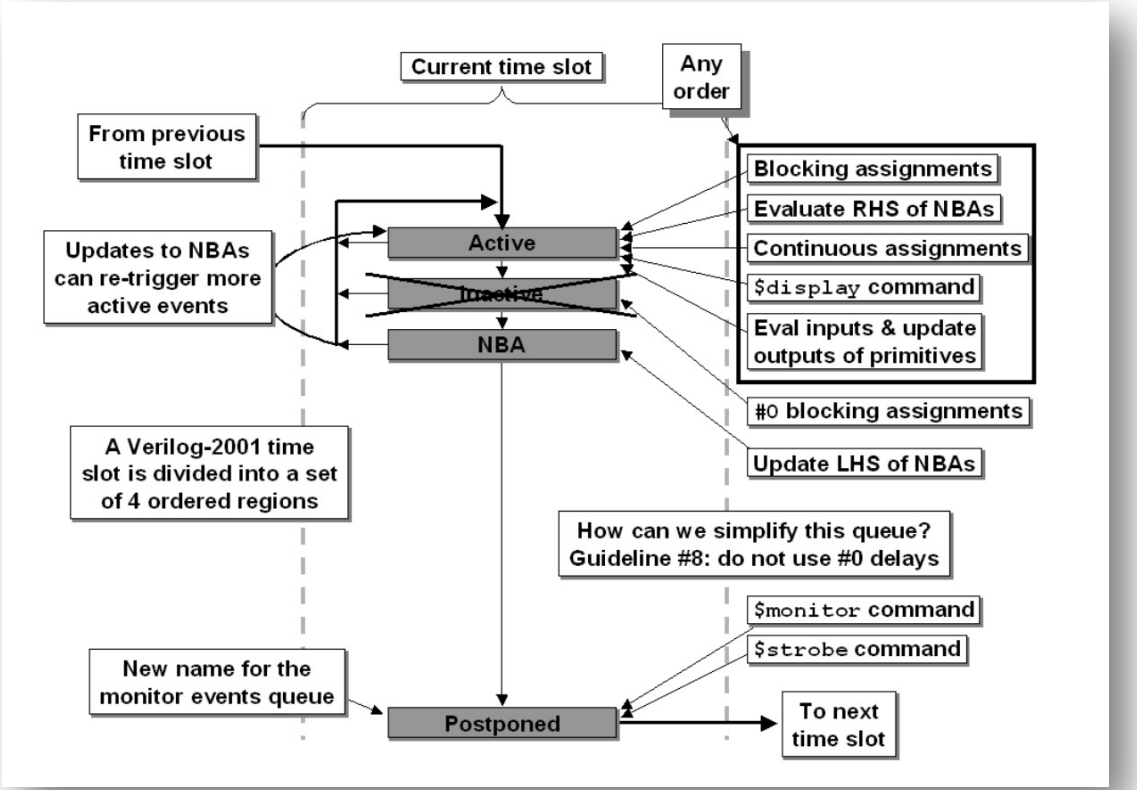
The read-only phase is entered when the current timestep no longer has any further delta steps. This will be a point where all the signal values are stable as there are no more RTL events scheduled for the timestep. The simulator will not allow scheduling of more events in this timestep. Useful for monitors which need to wait for all processes to execute (both RTL and cocotb) to ensure sampled signal values are final.

# Interestingly…VHDL

- Interestingly, VHDL largely avoids the issues with non-determinism in its design largely through its use of syntax and how its simulator does updates (no need to iterate back since it forces you to specify causality even when doing combinational logic)

# The Verilog Simulation Time Step

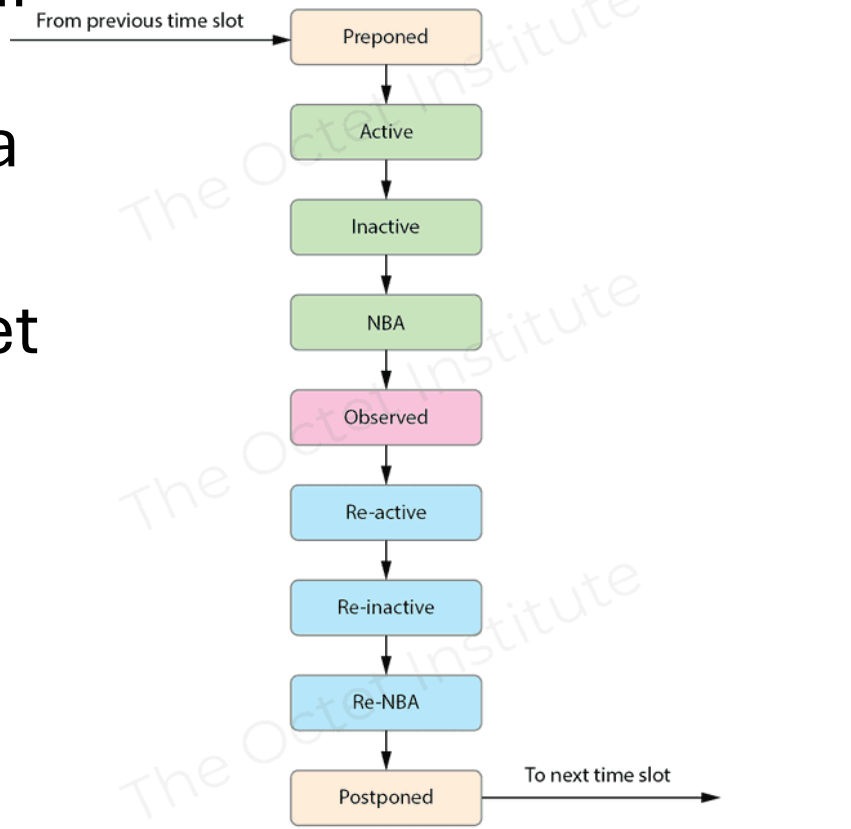
- Within each time slot are regions in which different evaluations and updates are made
- They go through a specific order and may iterate until values have “settled”




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

# SystemVerilog to the… Rescue?

- Whereas Verilog has like four-ish stages in a simulation step, SystemVerilog added a ton more on top
- Just like language is superset of Verilog, so is simulation engine. New:
  - Preponed
  - Observed region
  - Re regions…active region



# The SystemVerilog Simulation Time Step

- Just like languag
  is a superset of
  Verilog
- Simulation in
  System

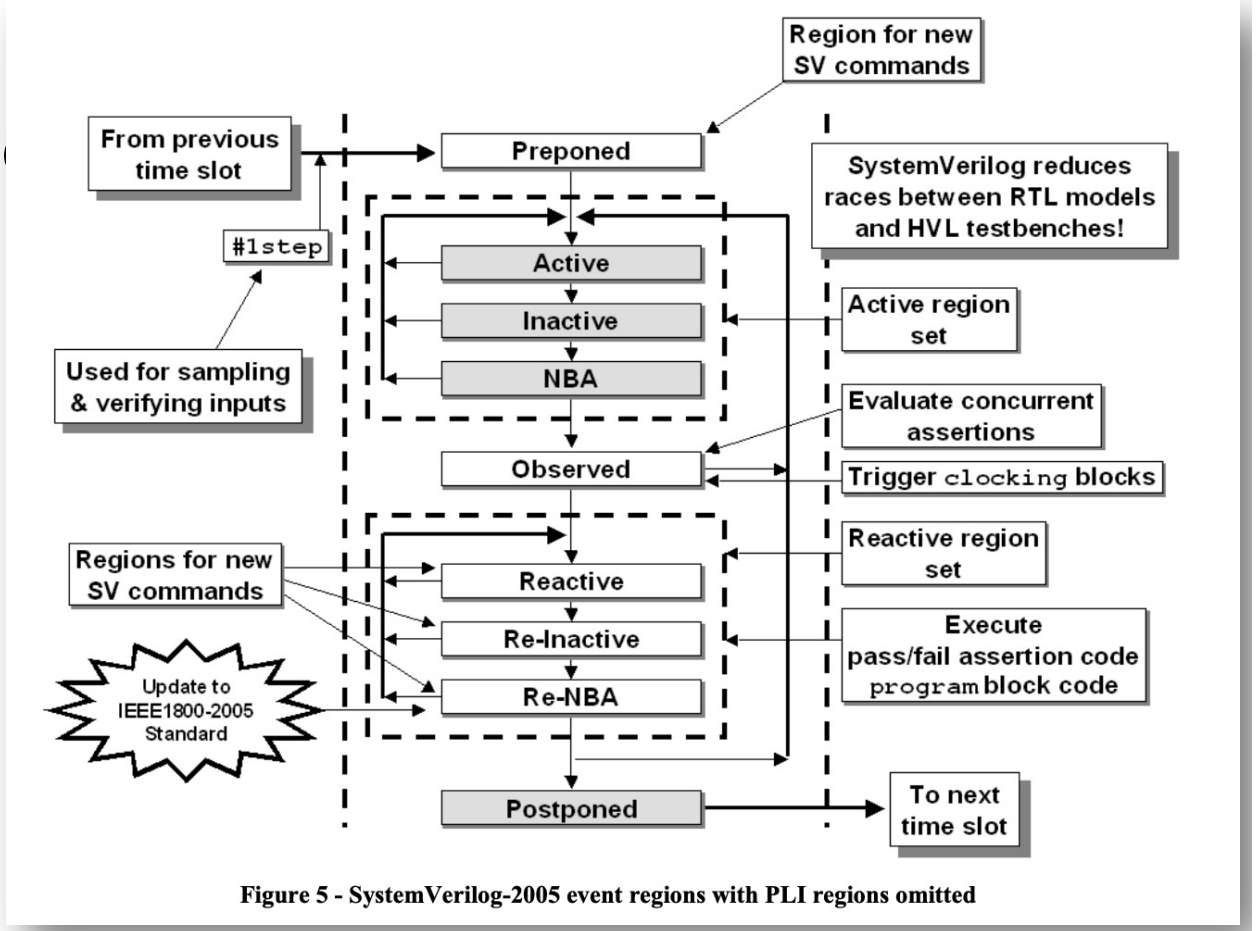



**Figure 5 - SystemVerilog-2005 event regions with PLI regions omitted**

"SystemVerilog Event Regions, Race Avoidance & Guidelines"
Clifford E. Cummings Arturo Salz

# System Verilog Simulation

- Still basic Active/Inactive/NBA region,
- But additional regions added in for more reliable simulation and control interfacing

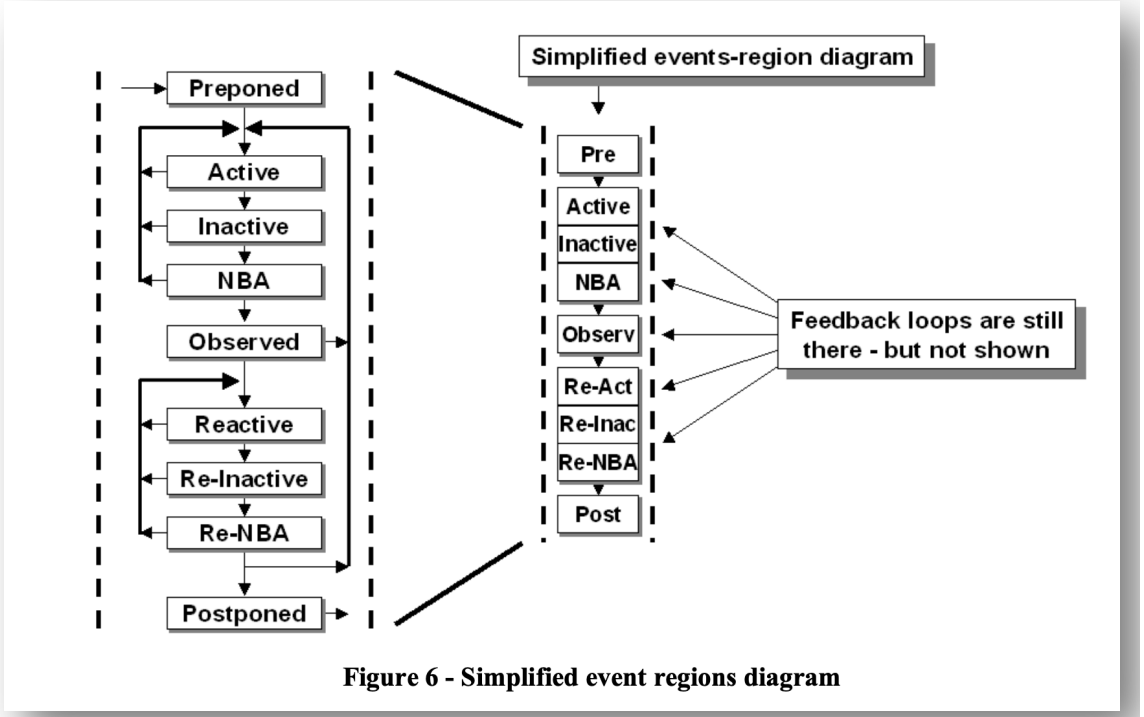



**Figure 6 - Simplified event regions diagram**

"SystemVerilog Event Regions, Race Avoidance & Guidelines"
Clifford E. Cummings Arturo Salz

# System Verilog Simulation

- Active/Inactive/NBA region meant for the hardware under simulation
- Re-active/Re-inactive/Re-NBA region meant for simulation/testing code. A testbench will set inputs in the reactive region, for example
- Separating the two was an attempt at avoiding bugs that showed up when mixing simulation with synthesis Verilog

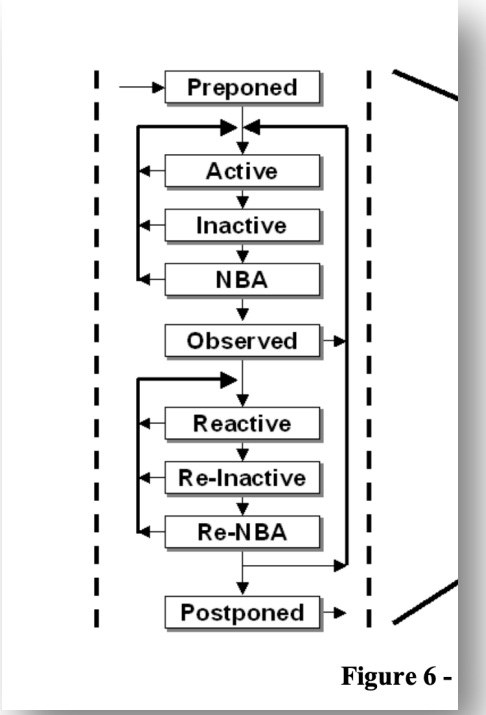


Figure 6 -

"SystemVerilog Event Regions, Race Avoidance & Guidelines"
Clifford E. Cummings Arturo Salz

# SV Time Slot Expanded Out

- 17-total stages in a single time slot now

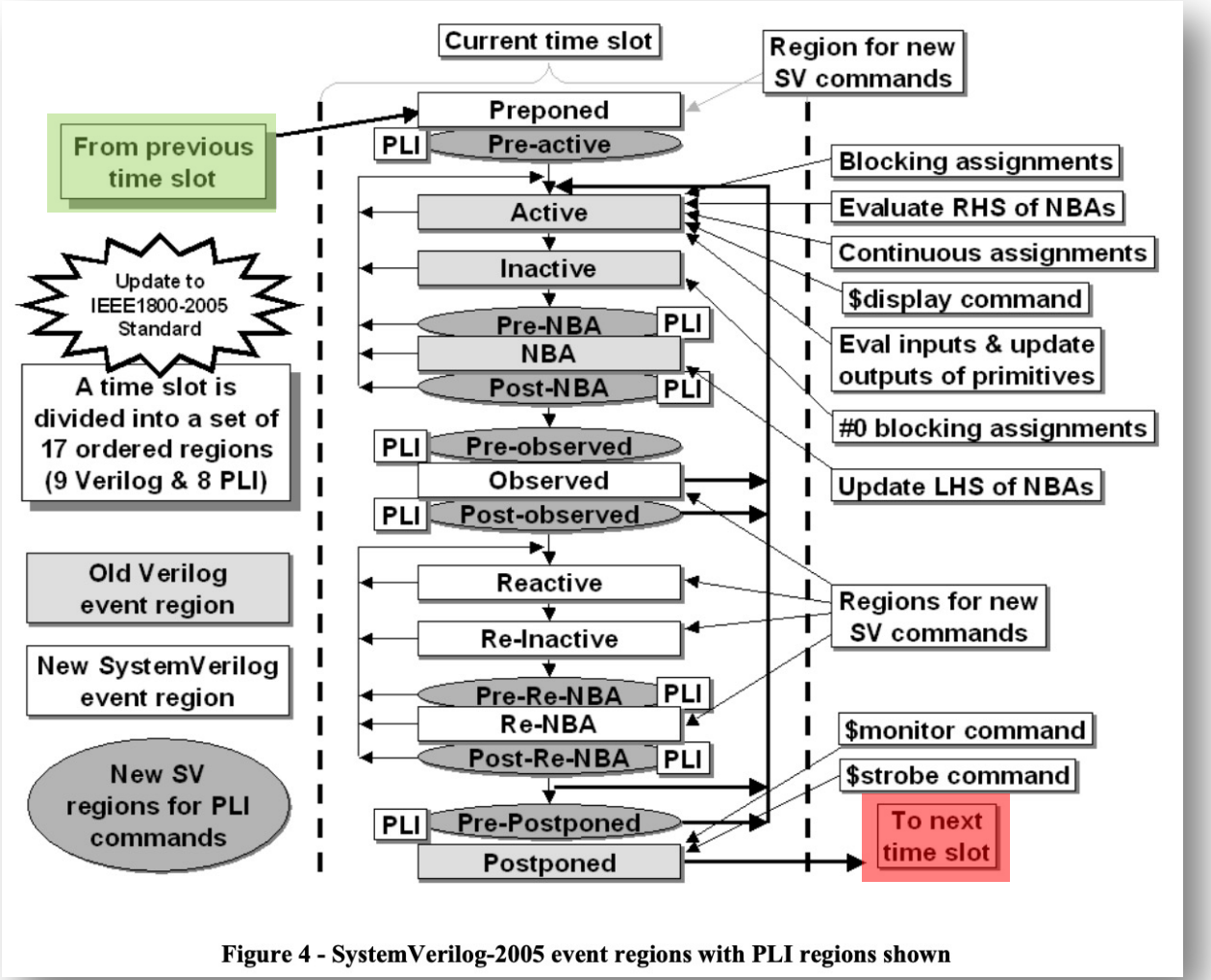



**Figure 4 - SystemVerilog-2005 event regions with PLI regions shown**

"SystemVerilog Event Regions, Race Avoidance & Guidelines"
Clifford E. Cummings Arturo Salz

# PLI regions?

- Sprinkled throughout are PLI regions
- These are actually regions to allow you to interface with the simulation

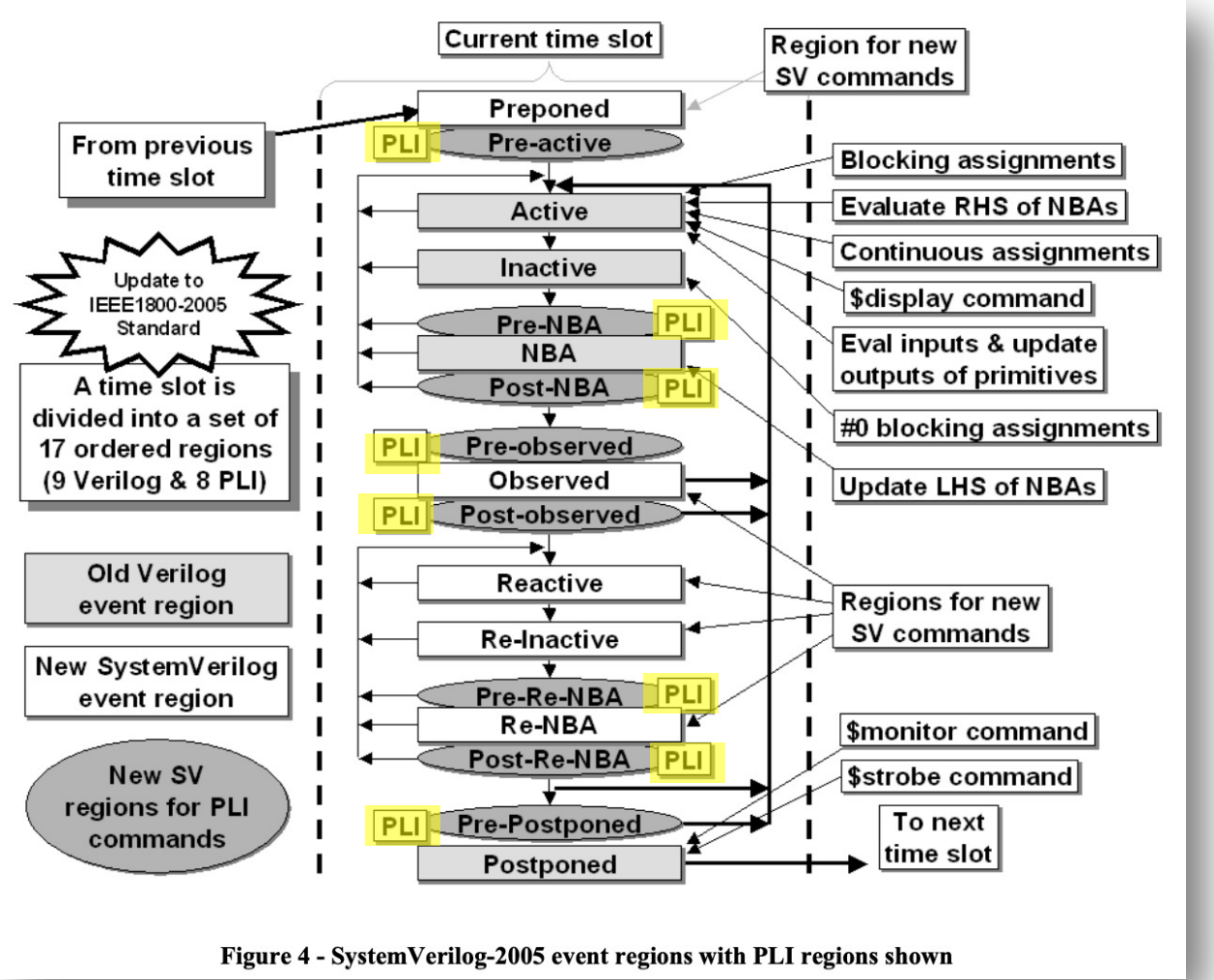



**Figure 4 - SystemVerilog-2005 event regions with PLI regions shown**

"SystemVerilog Event Regions, Race Avoidance & Guidelines"
Clifford E. Cummings Arturo Salz

# Program Language Interface (PLI)

- Original set of hooks built into the Verilog spec that would allow you to read out data from the Verilog simulation engine to C program
- You could also use them to inject inputs into Verilog
- In the original Verilog standard, the PLI integration was mixed in with the simulation part which could lead to bugs.

# PLI Died

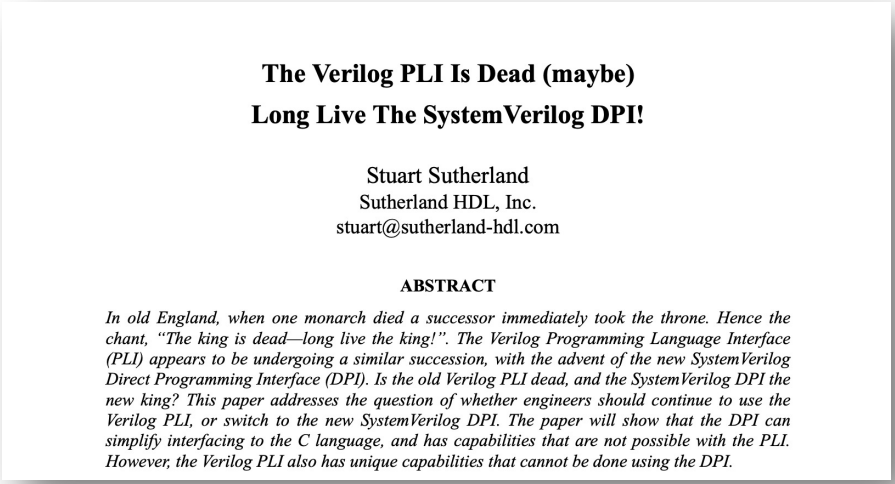
- PLI is deprecated and was replaced with the **Verilog Procedural Interface** or **VPI**
- You'll still hear PLI and VPI used interchangeably, but VPI is kinda the newer term.
- VPI is sometimes called "PLI 2.0"

# VPI vs. DPI

- The VPI allows a set of C/C++ functions that can be used to hook into various points in the simulation timesteps (basically at those PLI points)
- SystemVerilog has its own thing which is actually called a Direct Programming Interface (DPI)
- DPI is higher level and nominally cleaner

https://www.asic-world.com/systemverilog/dpi1.html

# Comparing and Contrasting…



**The Verilog PLI Is Dead (maybe)**
**Long Live The SystemVerilog DPI!**

Stuart Sutherland
Sutherland HDL, Inc.
stuart@sutherland-hdl.com

**ABSTRACT**

*In old England, when one monarch died a successor immediately took the throne. Hence the chant, "The king is dead—long live the king!". The Verilog Programming Language Interface (PLI) appears to be undergoing a similar succession, with the advent of the new SystemVerilog Direct Programming Interface (DPI). Is the old Verilog PLI dead, and the SystemVerilog DPI the new king? This paper addresses the question of whether engineers should continue to use the Verilog PLI, or switch to the new SystemVerilog DPI. The paper will show that the DPI can simplify interfacing to the C language, and has capabilities that are not possible with the PLI. However, the Verilog PLI also has unique capabilities that cannot be done using the DPI.*

- It is complicated.
- I've also seen VPI backronymed to Verification Peripheral Interface (Verilator)
- The big thing I want to point out here is pretty much all System/Verilog simulation engines (and VHDL too) have hooks into them that can allow C or other languages to interface with them.

# The Point of PLI/VPI/DPI

- All of these simulation entry points were created to provide ways to automate simulation.
- But also they found use in allowing you to interface non-HDL models of very complicated things:
  - Memory
  - CPU's
  - Other bit-accurate models
- Key component of the developing Verification field

# Building Up

- About ten years ago, some folks started to wrap up the VPI C material with Python and that ended up evolving into what Cocotb is today

# CoCoTb

Python for Simulation

# Two Big Parts to Cocotb

- Takes advantage of Python's asynchronous programming capabilities to launch many parallel running processes.
  - Very nice…can spawn off lots of tasks to take care of different jobs and not worry about having to task switch between tracking them.
- It utilizes the **Verilog Procedural Interface**
  - A built-in interface to the simulator's runtime environment that allows manipulation within the environment by outside forces

# CocoTb

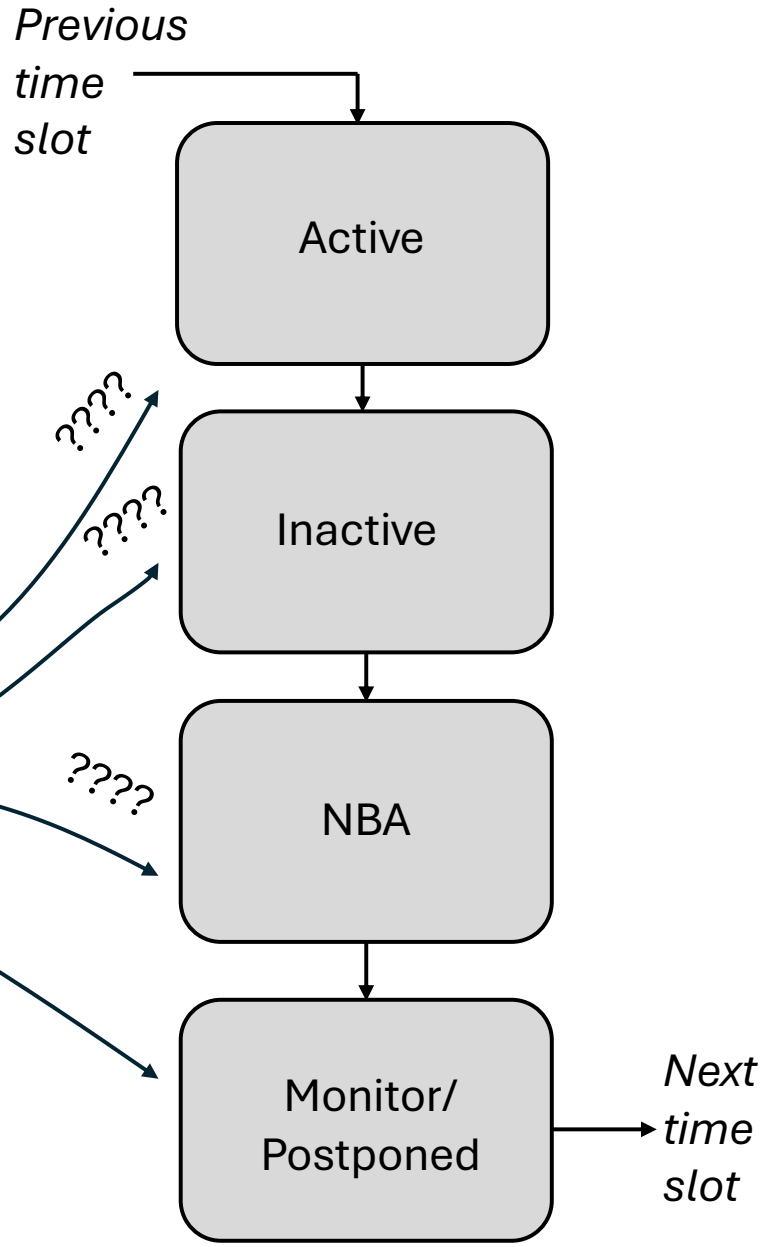
- Uses a GPI (general programming interface) which can then be further specified to use:
  - VPI: for Verilog-type simulators (icarus Verilog)
  - VHPI: for some VHDL type simulators
  - FLI: for other weirder formats (specifically Mentor/Siemans Questa)

# What can you do through the VPI that Cocotb builds upon?

- Basically anything you want.
- It gives you full access to all signals within the simulated environment
- It also has built a lot of utility logic that will “watch for events” through the use of callbacks…which can basically trigger the listening program to when things happen

# Verilog VPI

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



# Does Vivado Support VPI?

- No
- Whereas many other simulators (Mentor/Siemens, Cadence, Synopsis included) have maintained a VPI for backwards support Xilinx xsim did not

# Does Vivado Support VPI?

- No
- Vivado does support ”DPI” though
- So that means Cocotb does not work with Vivado

https://docs.amd.com/r/en-US/ug900-vivado-logic-simulation/Direct-Programming-Interface-DPI-in-Vivado-Simulator

# Attempts to link Cocotb to Vivado

- There's an open project on github of someone trying to connect Cocotb to Vivado's DPI:*:
  - https://github.com/themperek/cocotb-vivado/tree/main

- Kiran Vuksanaj then built upon this in project here through some further reverse-engineering of Vivado's DPI:
  - https://github.com/kiran-vuksanaj/vicoco/tree/main

*doesn't look touched in the half-year...

# Cocotb and Vivado

- There's some interesting parallel projects out there in this space:
  - https://github.com/fvutils/pyhdl-if

- I think what will eventually be needed is some sort of "buy-in" from Xilinx/AMD and I need to keep pushing on this because right now some of the important pieces of their DPI aren't really well documented, if at all.