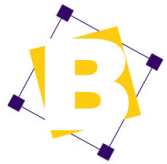
Boyd Technology Inc.

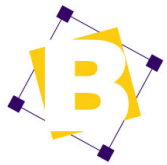
Program Block

Program Purpose



- Philisophical Motivation
  - "Entry point where test-bench begins execution"
  - "Scope that encapsulates program-wide data"
  - Assume software oriented view of chip environment vs modeling external hardware
    - Allows "programmers" to verify using blocking assignments without worrying about race conditions (no DAVEs just DE and VE)
  - Last paragraph of 15.1 (CH67) tries to address modeling
- Technical
  - Avoid races by giving testbench code access to reactive queue
    - Event & timing control will wait until reactive
    - Assignments will execute in active and NBA region **after** reactive queue

Syntax 1

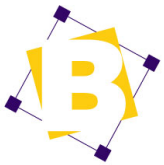


Boyd Technology Inc.

```
program foo (input x, y, output z);
...
    c.z <= c.x;
    @(myclking);
    c.z <= c.y;
    @(myclking);
    c.z = c.x;
    ...
endprogram
```

First wait until reactive region
Assignments evaluated in active region, update immediately or scheduled for NBA

Syntax 2



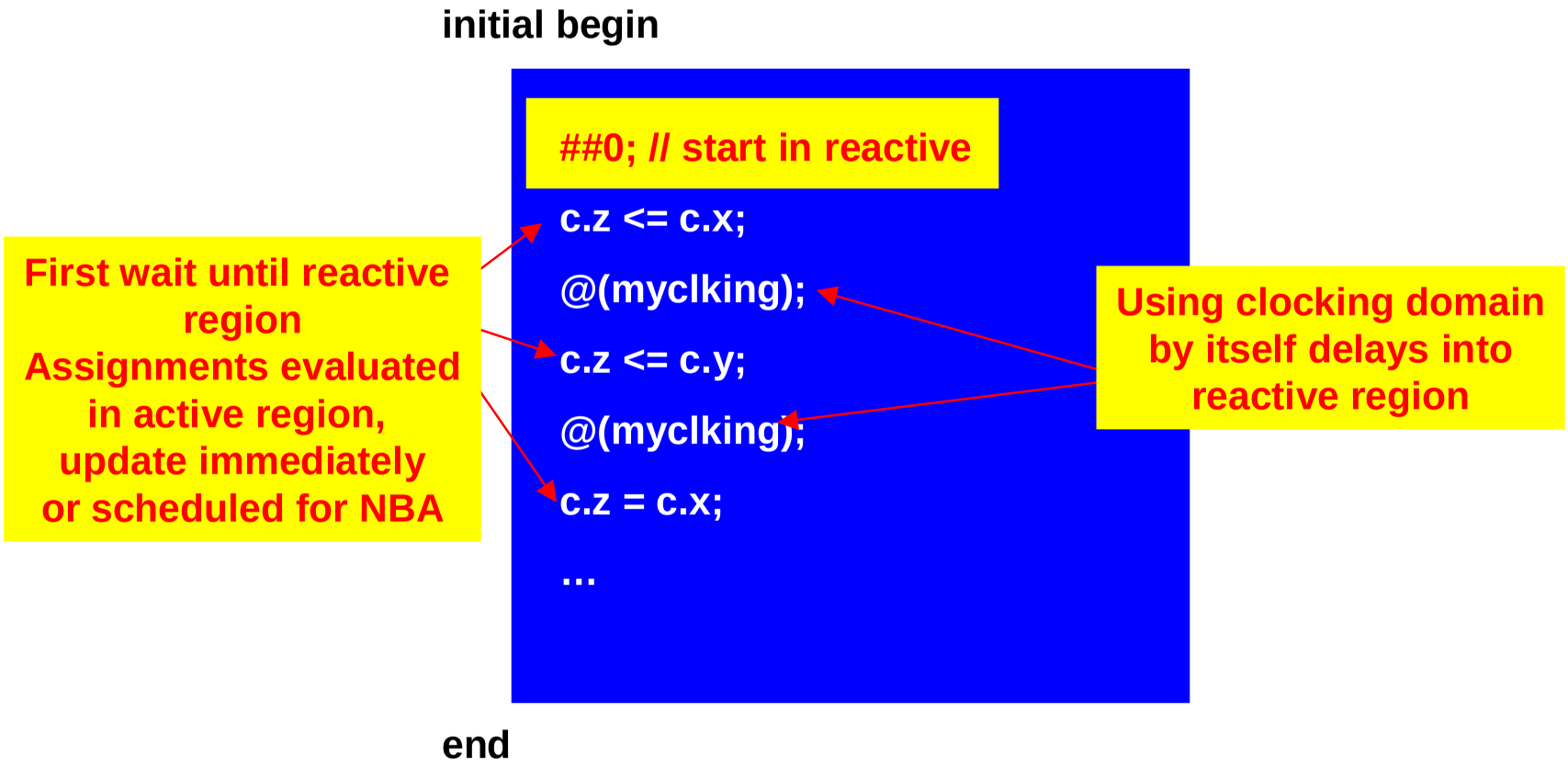
Boyd Technology Inc.

```
module foo (input x, y, output z);
program begin
  c.z <= c.x;
  @(myclking);
  c.z <= c.y;
  @(myclking);
  c.z = c.x;
  ...
end
```

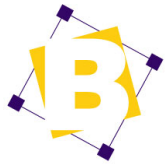
First wait until reactive region
Assignments evaluated in active region, update immediately or scheduled for NBA

Syntax 3

Boyd Technology Inc.

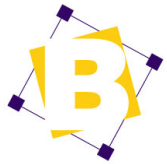


Accessing Reactive Region



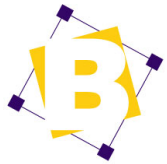
- Assertions
  - Code using output of assertion
  - Isn’t in Section 16 – understanding from other sources
- PLI
  - But SV isn’t defining the PLI – leaving that to later
- Testbench
  - Program
  - event/delay control

Testbench Access to Reactive



- New testbench code needs to run in reactive
  - Including new tasks and functions
  - Functions not problem – they always run in zero time
  - Tasks – need to make sure they “wake up” after event/delay control in reactive region
    - #1: It’s in the program block
    - #2: It was called from program block
    - #3: It has to use special delay/event control
- New code needs to use clocking domain version of signals
  - Need values sampled from one time step before clock
  - Don’t want values after NBA
    - more later

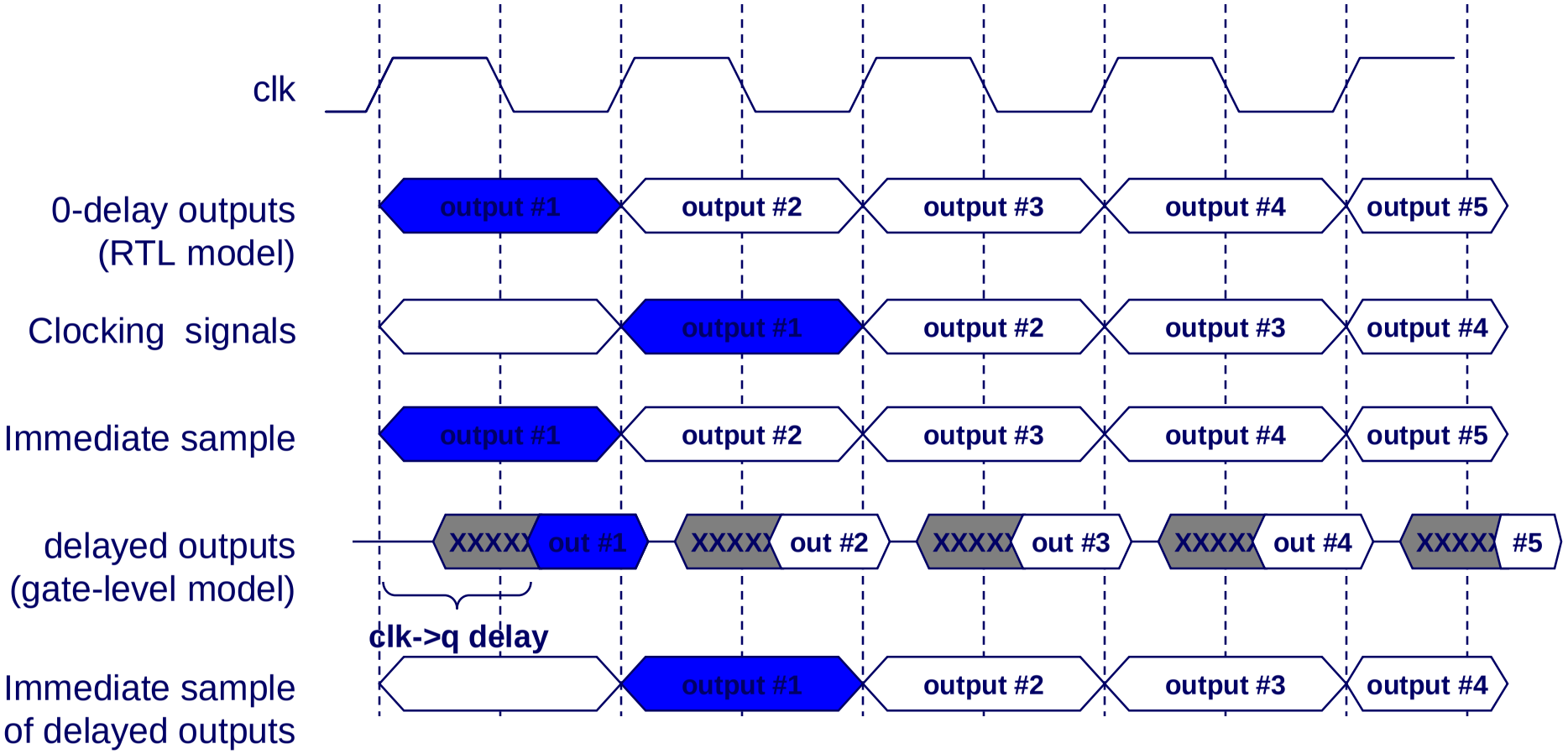
What about **OLD** tasks?



- BFM written for Verilog-1995 (or 2001) shouldn't use reactive region
- Uses NBA (or at least delays its outputs) to prevent races
- Expects to sample during active region of queue (the first time through – before NBAs get updated)

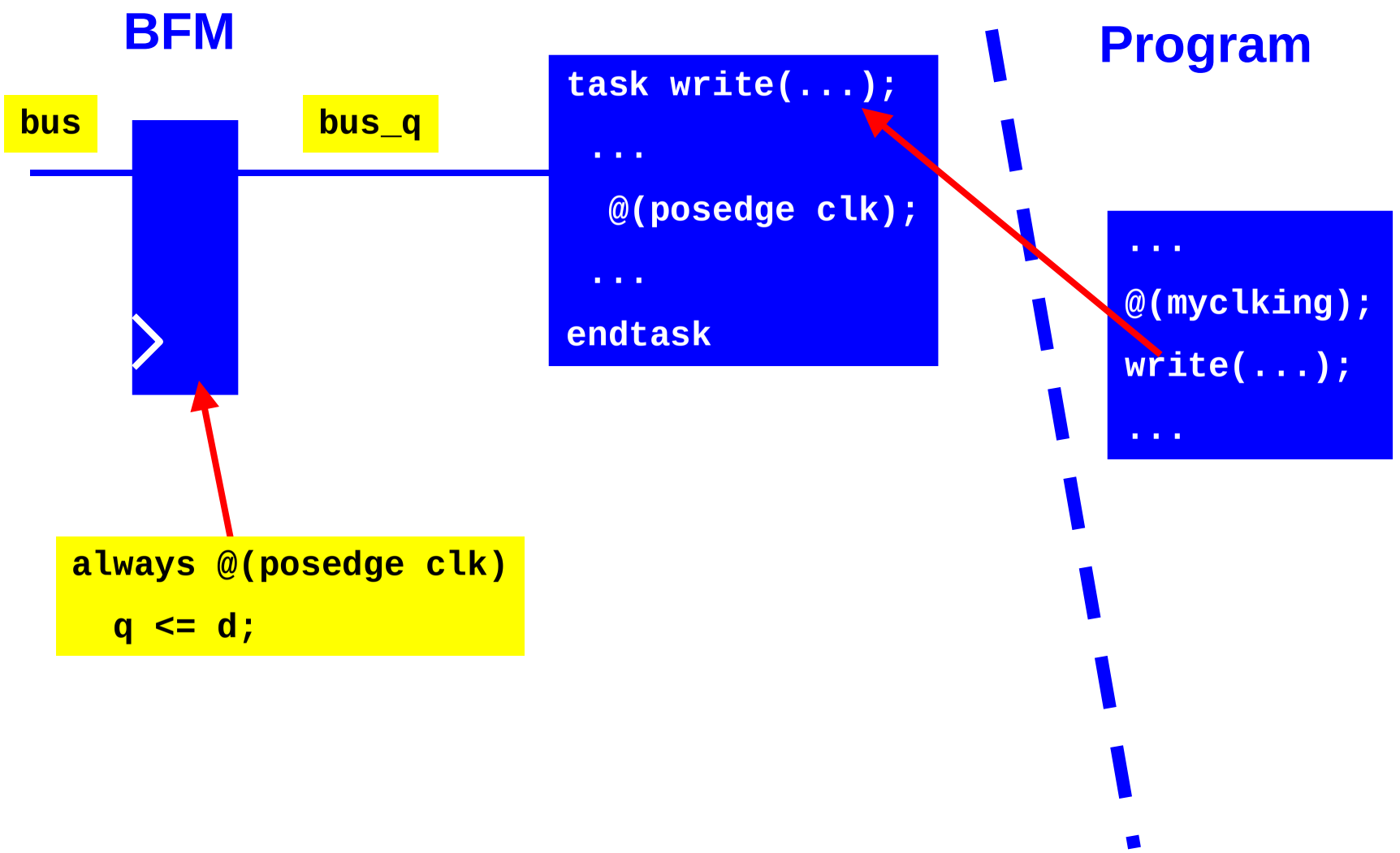
Mixing Reactive Execution With Immediate Sample

Boyd Technology Inc.

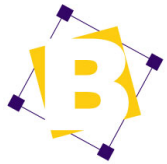


PCIX BFM

Boyd Technology Inc.

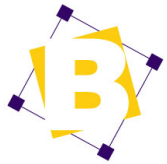


What about those OLD BFM Models?

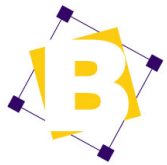


- Proposal #1
  - Will allow them to wake up ok
    - Event and delay control will work since task isn't declared in program
  - Fail on all samples taken before first event/delay control
- Proposal #2
  - Fails on first samples
    - Sampling after NBA when designed to sample ***before*** NBA
- Proposal #3
  - Normal event control ok (it's just an initial block!)
  - After special event/delay control
    - Once in reactive region, you had better not call a legacy task that expected to be called in first pass through active region!
    - At least you had to think about entering reactive – so maybe you'll be more careful

Program Summary

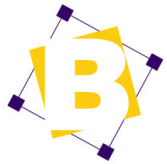


- Is it really ready yet?
- Proposals #1 and #2 don't solve issues with legacy testbenches
- Proposal #3 doesn't address motivation for adding the construct to the language
- Testbench code that reacts to assertions will already run in reactive region
  - Waits on assertion outputs which can't change until reactive
- Postpone program until SV3.2?
  - Need to address motivation
  - Need to support legacy code in way that supports the "programmer doing verification" model



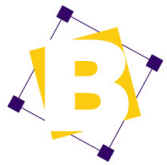
Clocking Domains

Clocking Domain and Reactive Region



- Clocking Domain important part of use of reactive region
- Failure to use clocking domain leads to pre vs post synthesis differences
- How do you get to reactive region?
  - Event/delay control inside special construct (program)
  - Special event/delay controls
- How easy is it to accidently use non-clocked version of signals?
  - Most common place for clocking domain is in program
  - Both clocking signal and regular signal available
  - Non-clocking signal **easiest** to use

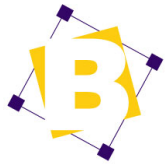
Syntax 3 – Forget “c.”




```
initial begin
  ##0; // start in reactive
  c.z <= c.x;
  @(myclking);
  c.z <= y;
  @(myclking);
  c.z = c.x;
  …
end
```

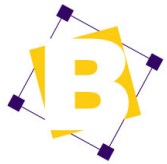
Ooops, I forgot to include clocking domain “c.” when sampling signal!

Clocking Domain and Extensibility / Reusability



- 24 Port Gigabit Ethernet
  - Need 24 clocking domains – one for each port
- Put each port's clocking domain in interface
  - Instantiate clocking domain in interface
    - Have to use interface.clocking.signal – yuck!
    - I already disliked clocking.signal!
  - Change syntax so clocking domain is just a view of signals in an interface
    - Only have to type interface.signal
      - **`Still would like enhancement in 3.2 to fix this`**
    - Can't accidentally get non-clocked signal

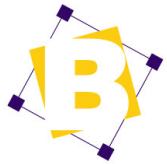
Current Syntax



Boyd Technology Inc.

```
module/program tb (
  logic req, gnt,
  logic [7:0] addr, data,
  logic [1:0] mode,
  logic start, rdy);

  clocking clk1 @(posedge clk);
    input #1step gnt, rdy, clk;
    output #2 req, addr, mode, start;
    inout data;
  endclocking



...

endmodule/endprogram
```

Proposed Syntax




```
interface simple_bus (
  logic req, gnt,
  logic [7:0] addr, data,
  logic [1:0] mode,
  logic start, rdy);

  modport tb @(posedge clk) (
    input #1step gnt, rdy, clk,
    output #2 req, addr, mode, start,
    inout data);
endinterface: simple_bus

module/program tb(simple_bus.tb bus);

...

endmodule/endprogram
```

Can't accidently get version that didn't go through clocking