Configuring a Date with a Model

A Guide to Configuration Objects and Register Models

Jeff Montesano, Jeff Vance
Verilab, Inc.

September 29, 2016
SNUG Austin

Agenda

Testbench and DUT Synchronization Problems

Register Models and Configuration Objects

Synchronizing Designs with Triggers

Synchronizing Designs without Triggers

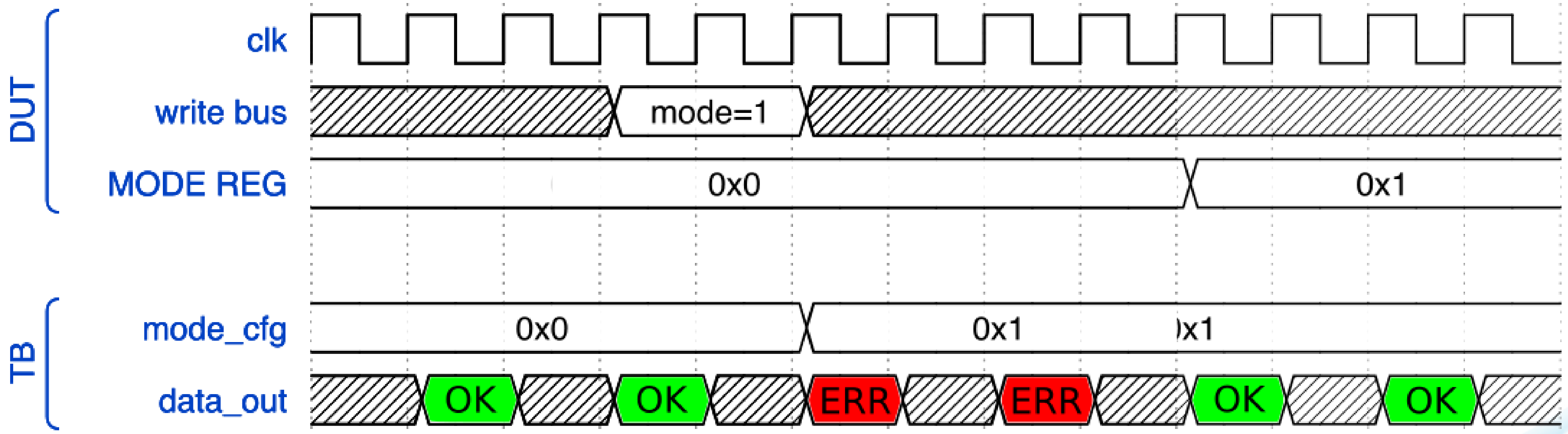
Conclusions

All source code is available for download.
www.verilab.com/resources/source-code

Testbench and DUT Synchronization

Checks Must use the Same Configuration as DUT

1. Initially checks mode 0, then mode 1
2. Testbench assumes mode 1 is active (false errors)
3. DUT starts using mode 1

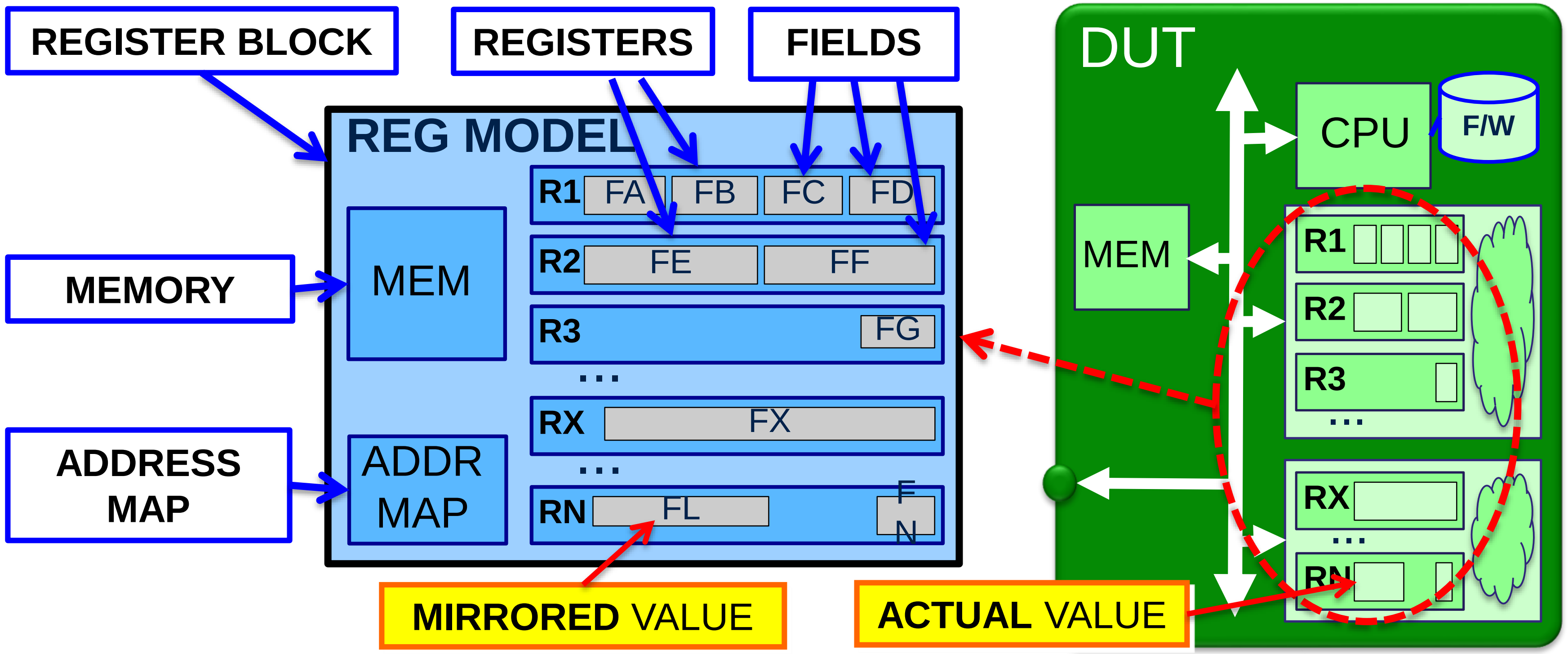


Register Models & Configuration Objects

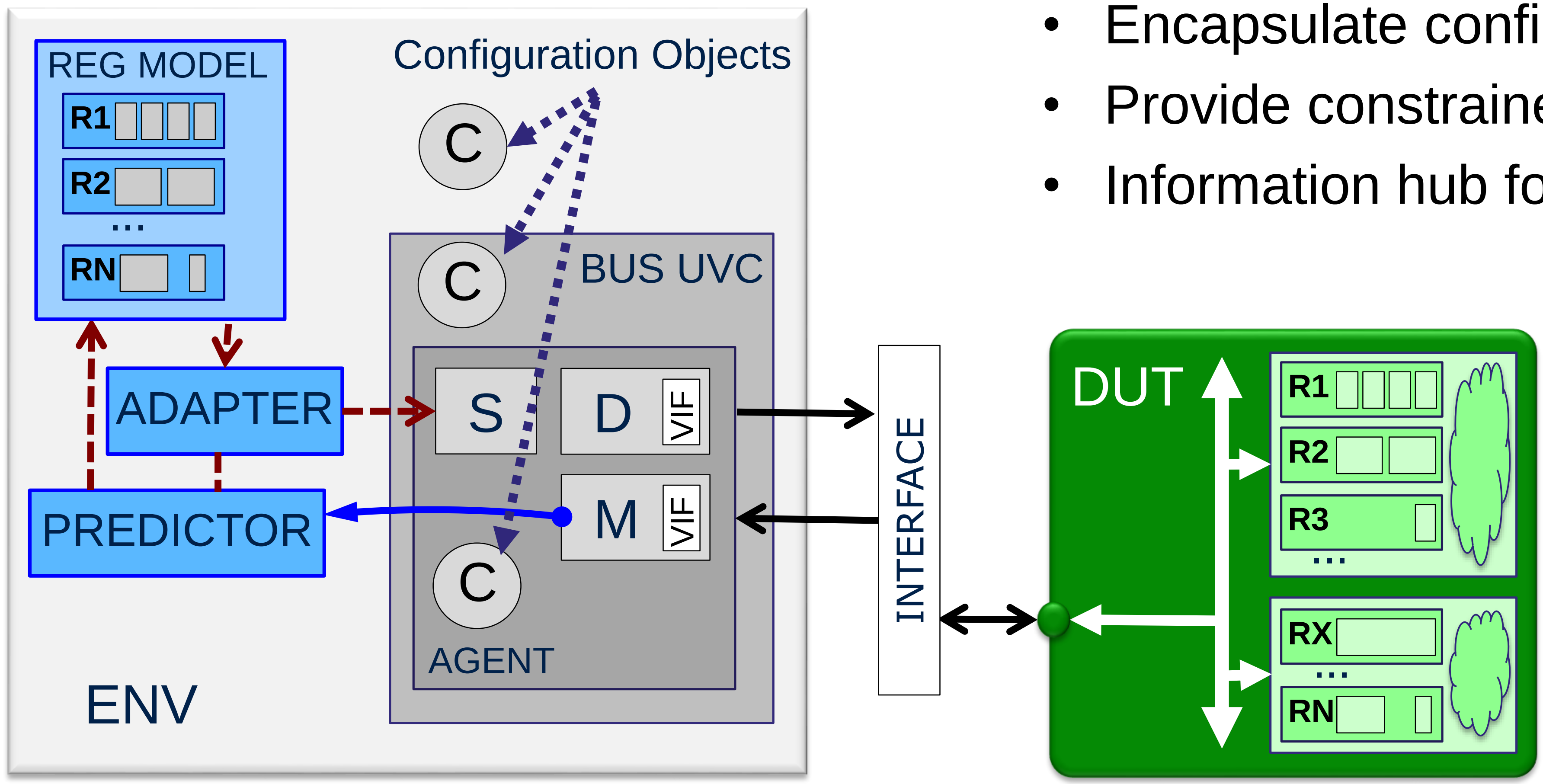
A Quick Overview

Role of a Register Model

- Database of registers and fields
- API for register access from tests (back-door and front-door)



Role of a Configuration Object



- Encapsulate configuration info
- Provide constrained-random values
- Information hub for testbench

Types of Configuration Fields

Pseudo-static

- Can change, but *infrequently*
- Design might only use one setting
- Examples:
  - operating speed / mode
  - number of data lanes
  - ADC settings (sampling rate)

Dynamic

- Can change frequently
- Normal operation expects this
- Examples:
  - start processing bit
  - counter reset
  - configuration trigger

Static

Design constants that can’t change

Comparing Roles

Register Models vs. Configuration Objects

CTRL1:0x0	15:1	0
	RESERVED	COUNTER_EN

CTRL2:0x2	15:1	0
	RESERVED	PARITY

CTRL3:0x4	15:1	0
	RESERVED	RESET_COUNTER

CTRL4:0x4	15:0
	MAX_COUNT_L

CTRL5:0x5	15:2	1:0
	RESERVED	MAX_COUNT_H

- Both hold configuration data
- A register model is tied to the *design*
  - Mirrors field positions per register
  - Mirrors address, width, and access rights
  - These details change per design
- A config object is tied to the *protocol*
  - Not tied to implementation details
  - Design changes have no impact
  - Can be reused for other designs

Configuration Objects are Generic

```
class block_config  extends uvm_object;
 rand bit           counter_en;
 rand bit           parity;          // 0=odd, 1=even
 rand bit [17:0] max_count;          // 0x0=1, 0x3FFFF=262144
 rand bit           reset_counter;
 …
endclass
```

Tied to one implementation!

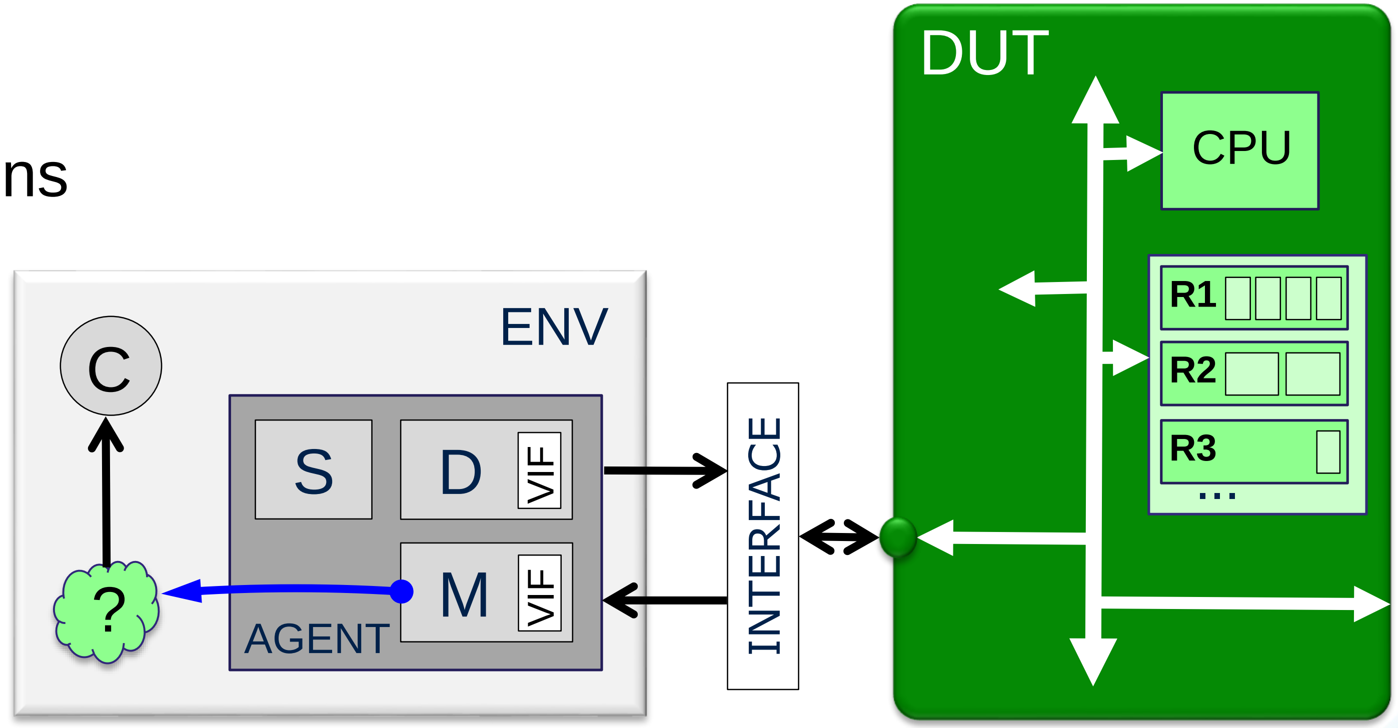
```
class block_config extends uvm_object;
 rand bit                       counter_en;
 rand my_parity_enum            parity;        //EVEN, ODD
 rand int                       max_count;
 rand bit                       reset_counter;
 …
 constraint c_max_count {
   max_count >= 1;
   max_count <= 262144;
 }
endclass
```

This is more generic

Basic Testbench Synchronization

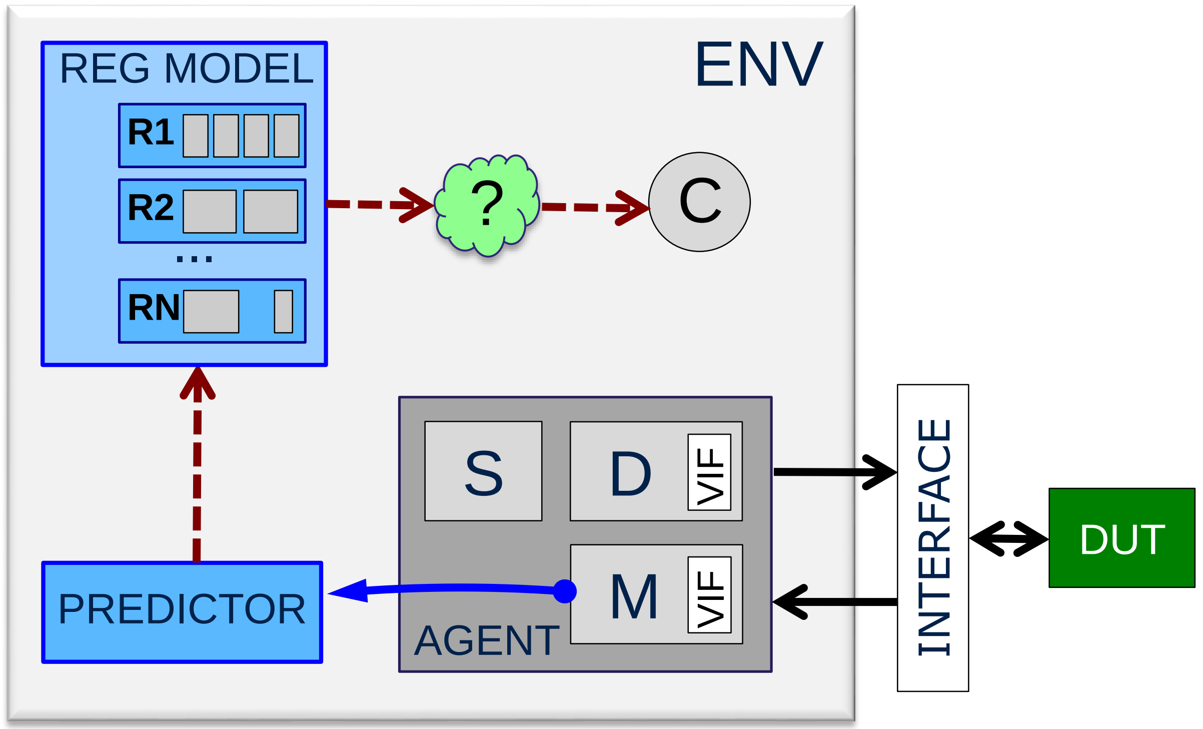
Must use Passive Monitoring

- Monitor observes register bus
  - sends transactions to testbench
  - configuration object is updated
- Handles *all* sources of transactions
  - BFM (active)
  - other design blocks (passive)
  - other interfaces (passive)



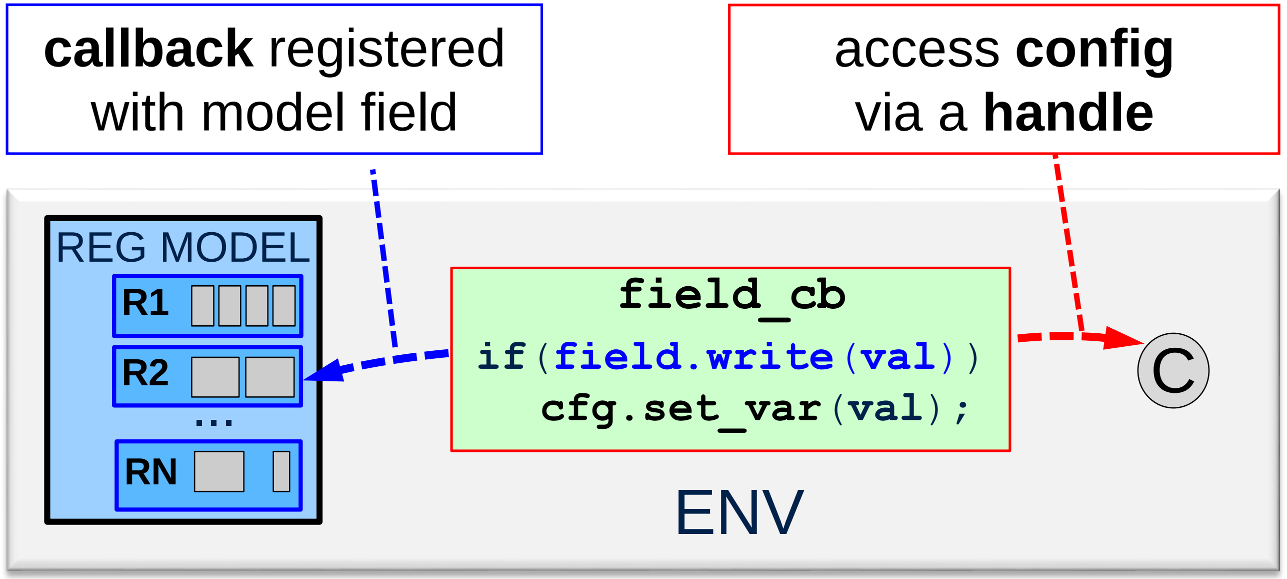
Register Model Helps Synchronize

- Predictor translates transaction
  - Address is mapped to a register
  - Fields are mapped to registers
- Register model mirrors data
- Configuration object needs data to stay in sync



Callbacks

- Callback classes must be extended
- Callbacks must be registered with registers or fields
- UVM Register model has many
- *post_predict()* is most important
  - Only callback that works in passive context
  - Can only be registered with ***fields***.



callback registered with model field
access **config** via a **handle**
REG MODEL
R1
R2
...
RN
field_cb
if(field.write(val))
cfg.set_var(val);
C
ENV

Mark Litterick, “Advanced UVM Register Modeling – There’s More Than One Way to Skin a Reg,”

DVCon 2014

Callbacks

```
class block_config extends uvm_object;

  bit                 counter_en;
  my_parity_enum      parity;
  ...
endclass
```

Configuration Object Field

```
class my_counter_en_cb extends uvm_reg_cbs;
  block_config config_obj;

  function void post_predict(input uvm_reg_field  fld,…);
    if (kind == UVM_PREDICT_WRITE) begin
      config_obj.counter_en  =  value;
    end
  endfunction
endclass
```

Corresponding register model callback

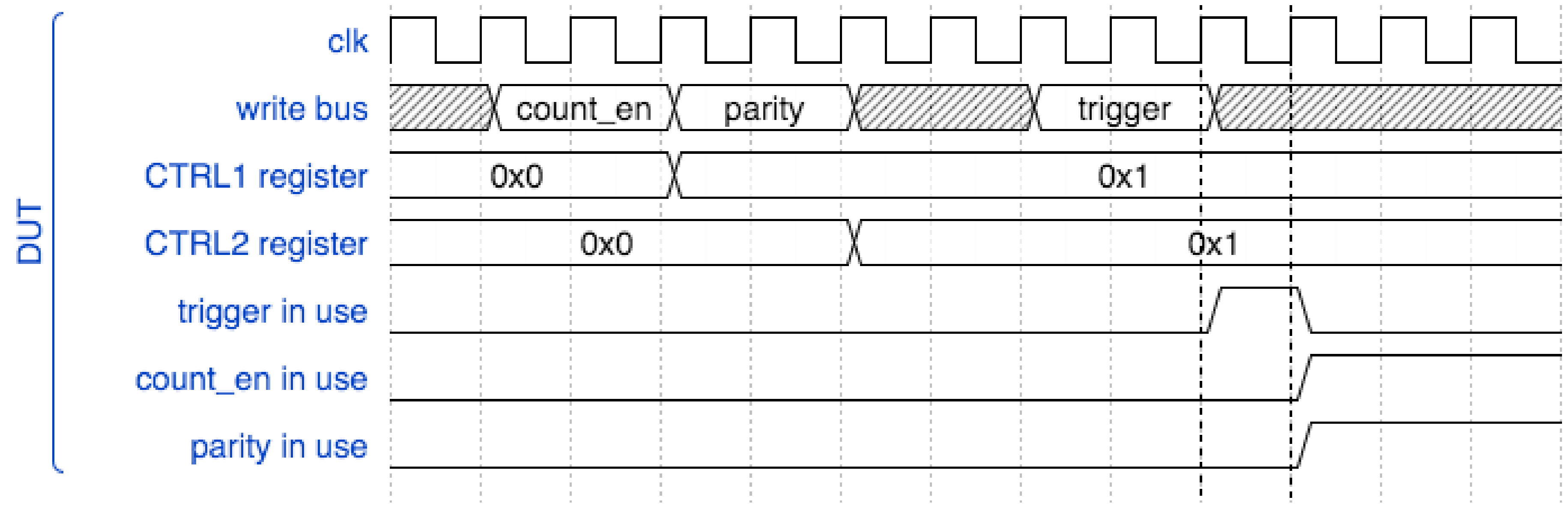
Synchronization Challenges

Designs with Triggers Events

Designs with Triggers

Transactions are Not Stopped

- A new DUT configuration is written
- The new configuration will *not* be used immediately
- A trigger makes the new configuration active



Handling Triggers

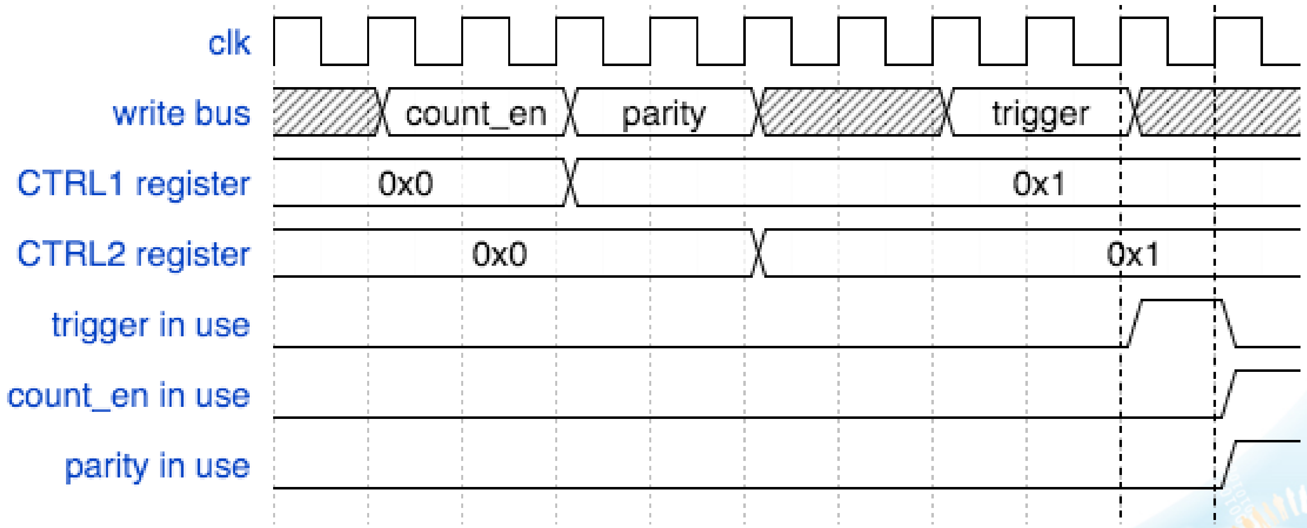
Pending Variables Added

- DUT isn't using *new* config yet
- Checks must keep using *old* config
- Field callbacks only update the *pending* variables!

```
class block_config extends uvm_object;

  rand bit  counter_en_pending;
  bit       counter_en;

  rand my_parity_enum   parity_pending;
  my_parity_enum        parity;
  ...
endclass
```

Trigger Field

- Needs its own callback
- Assigns pending value to actual



Trigger Callback

Make Pending Values Active

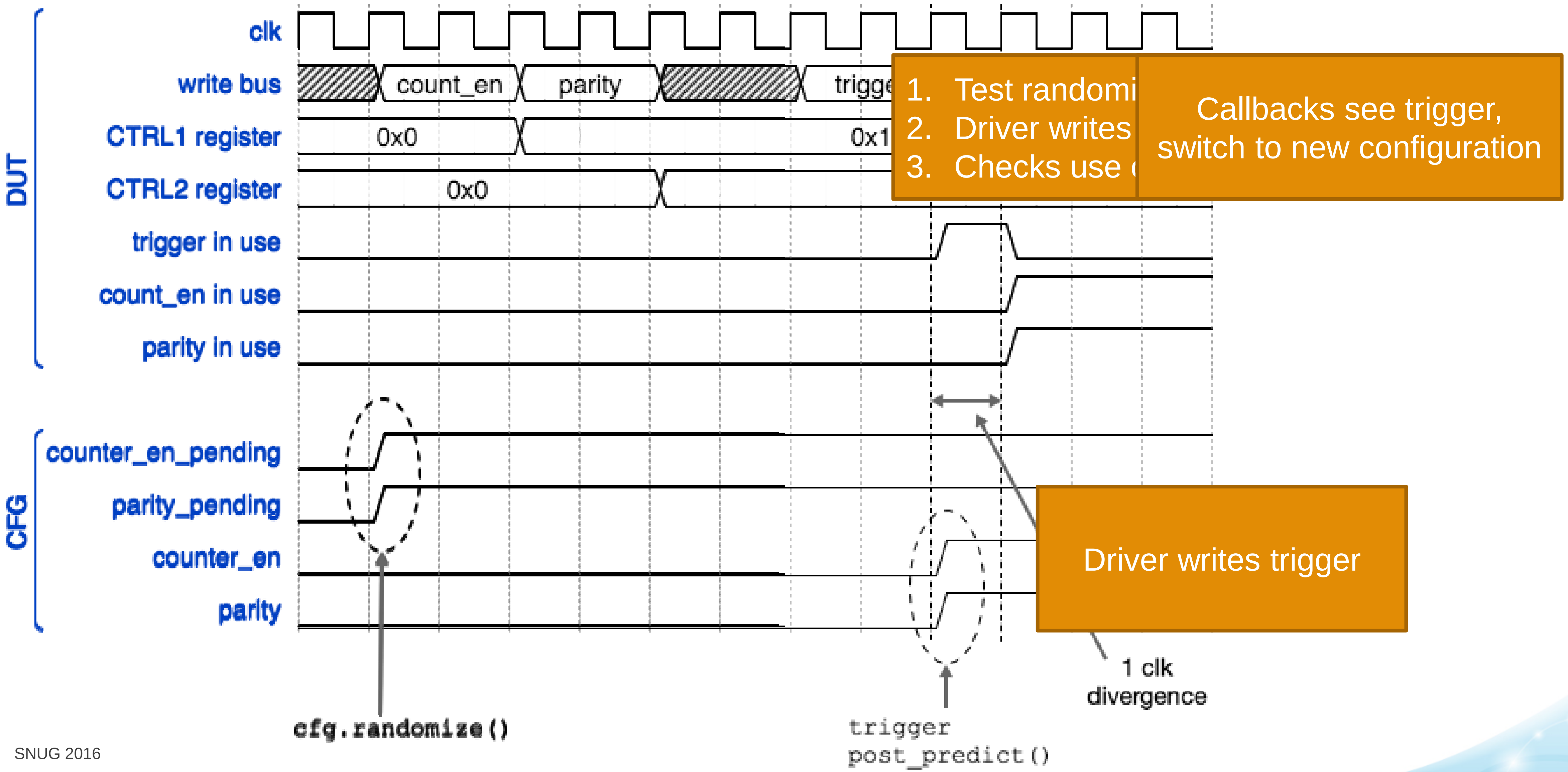
Testbench switches to the new configuration

```
class block_config extends uvm_object;

 rand bit   counter_en_pending;
 bit        counter_en;

 rand my_parity_enum    parity_pending;
 my_parity_enum         parity;
 ...
endclass
```

```
class my_trigger_cb extends uvm_reg_cbs;
 block_config config_obj;

 function void post_predict(input uvm_reg_field  fld,
                            …);
   if (kind == UVM_PREDICT_WRITE) begin
     if (value == 1)
       config_obj.counter_en = config_obj.counter_en_pending;
       config_obj.parity     = config_obj.parity_pending;
       …
   end
 endfunction
endclass
```

Designs with Triggers

Active Scenario



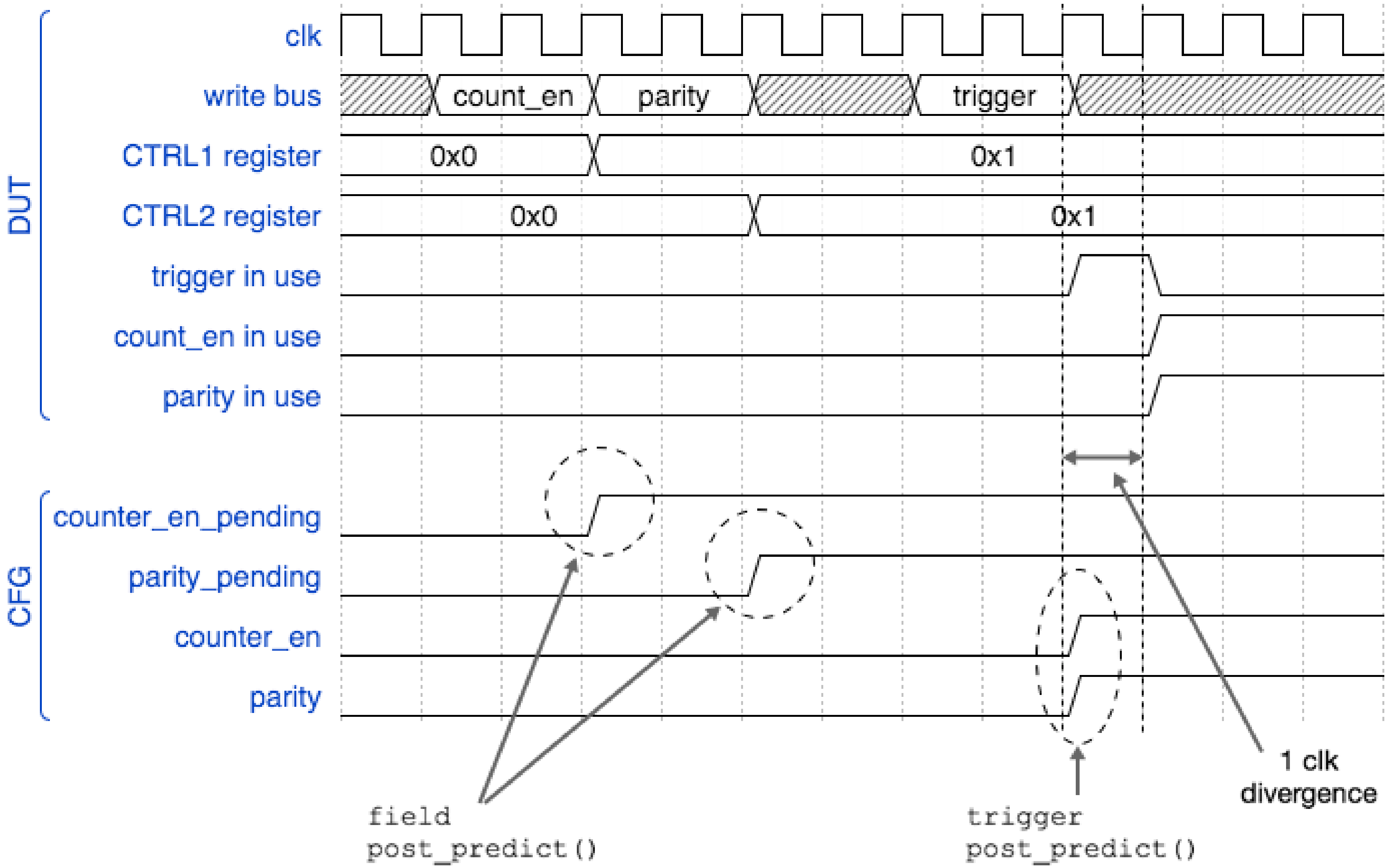
1. Test randomi
2. Driver writes
3. Checks use (

Callbacks see trigger, switch to new configuration

Driver writes trigger

Designs with Triggers

Passive Scenario

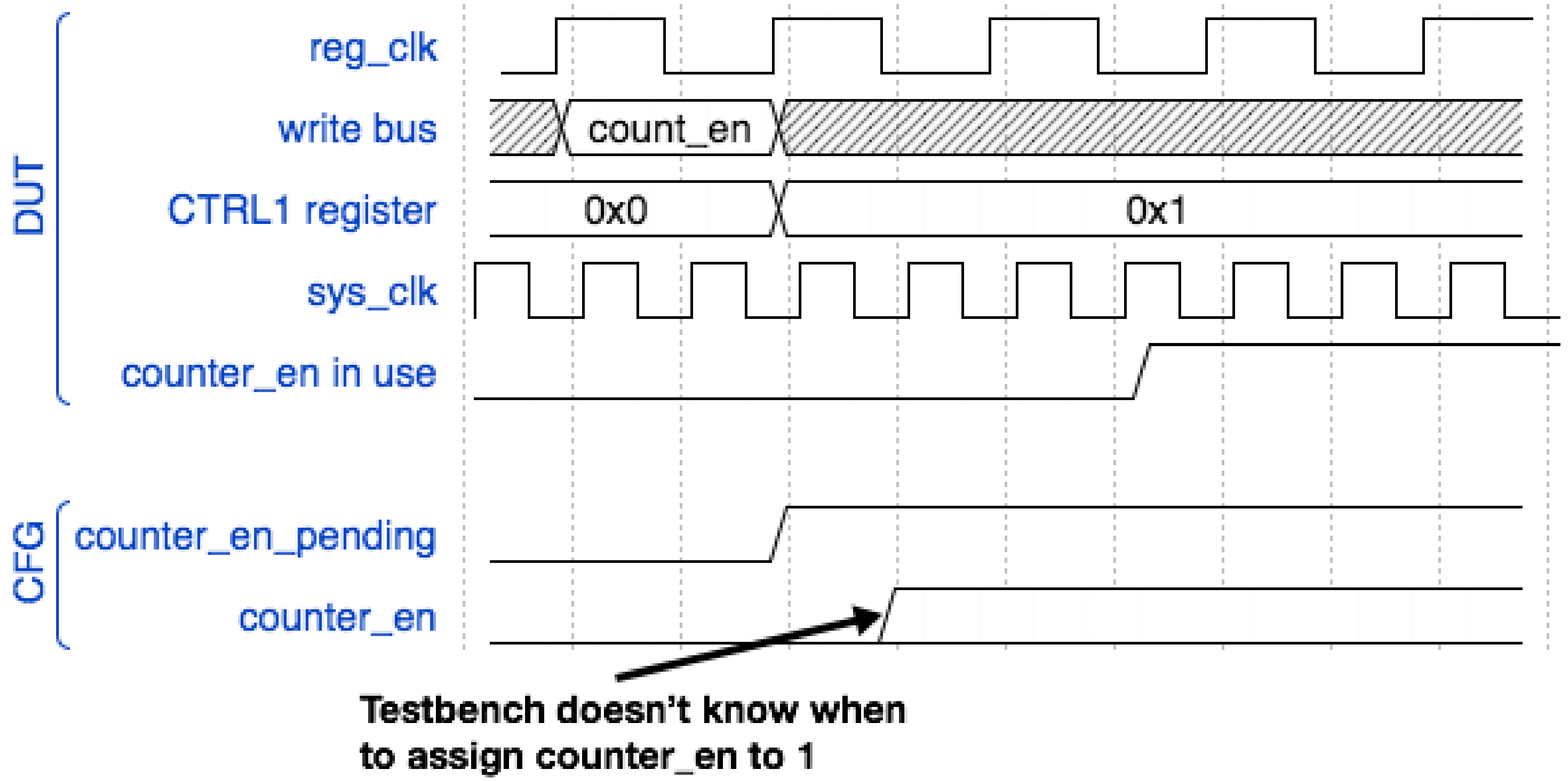


Synchronization Challenges

Designs Without Triggers

Designs Without Trigger

Clock Domain Crossing



Synchronize Configuration

No Trigger Event

```
class block_config extends uvm_object;
  …
  task keep_config_in_sync_with_dut();
    fork
      forever begin
        @(vif.counter_en);
        counter_en = counter_en_pending;
        if (counter_en != vif.counter_en)
          if (check_register_values)
            `uvm_error(…, "wrong value")
      end
    join_none
  endtask

  function new(…)
    …
    keep_config_in_sync_with_dut();
  endfunction
```

- Probe DUT for the moment to sync
- The probed value is *not* copied
  - This is only for getting a sync event
  - Use the *pending* value for new config
  - Check if this value matches the internal DUT configuration
- We need one thread per field

Synchronize Configuration

No Trigger Event

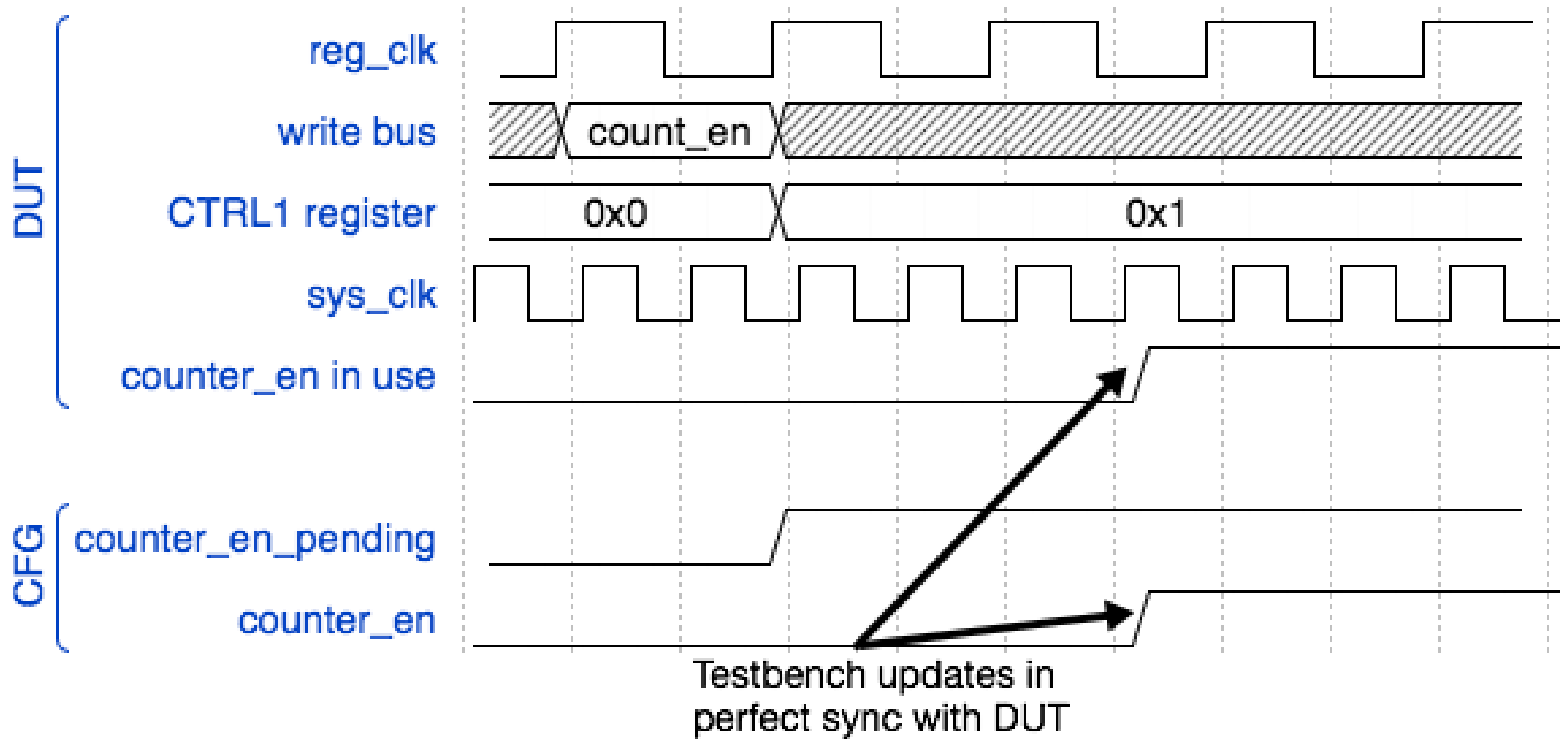
Use macros to create synchronizing threads for all fields.

```
class block_config extends uvm_object;
  …
  task keep_config_in_sync_with_dut();
    `SYNCHRONIZE_CONFIG_FIELD(counter_en)
    `SYNCHRONIZE_CONFIG_FIELD(parity)
    …
  endtask
  …
  function new(…)
    …
    keep_config_in_sync_with_dut();
  endfunction
```

```
`define SYNCHRONIZE_CONFIG_FIELD(field)\
fork                                  \
 forever begin                        \
   @(vif.field);                      \
   field = field``_pending;           \
   if (field != vif.field)            \
     if (check_register_values)       \
       `uvm_error(…, "wrong value")   \
 end                                  \
join_none
```

Configuration Synchronized

Testbench in Sync with DUT



More Info and Examples

- See paper for more examples
  - Signals that aren't resynchronized to system clock
  - Handling X's at time 0
  - Handling resets
  - Handling auto-clear registers

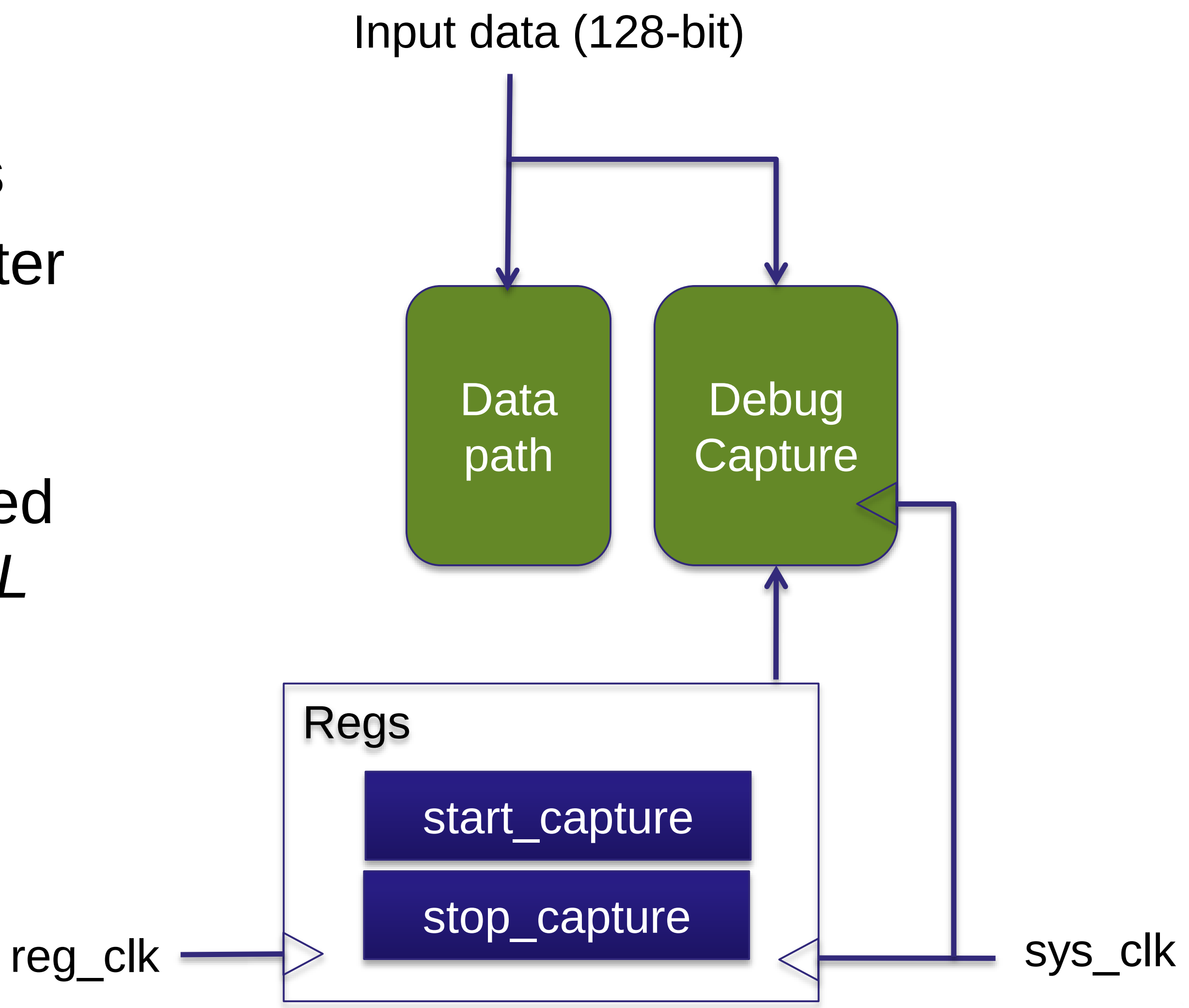
- Source code available
  - Fully working examples
  - Makefiles included
  - Get running in VCS in just minutes

Download code examples from:
www.verilab.com/resources/source-code

Real World Design

Debug Data Capture Block

- Data stored in "capture memory"
- "start/stop capture" dynamic fields
- Clock-domain crossing from register clock to system clock domains
- Testbench scoreboard needs to know exact moment capture started and stopped *without snooping RTL values from the design*



Conclusions

- Configuration objects are more abstract than register models and serve different roles - Don’t confuse them!
- A register model and configuration object can work together to ensure testbench is in sync with DUT
  Save time by avoiding debug of false errors!
- We can look into the design to get sync information without compromising the verification environment
- A framework of macros can simplify code

Synopsys Users Group
Austin

Thank You