

# Don't Forget the Holes in the Box: Utilizing Embedded Components for Verification

## Authors:

Ralph Martin (Honeywell, ASIC/FPGA Design, [Ralph.Martin@honeywell.com](mailto:Ralph.Martin@honeywell.com))

Brett Oliver (Honeywell, ASIC/FPGA Design, [Brett.Oliver@honeywell.com](mailto:Brett.Oliver@honeywell.com))

John Profumo (Honeywell, ASIC/FPGA Design, [John.Profumo@honeywell.com](mailto:John.Profumo@honeywell.com))

Lucas Roosevelt (Honeywell, ASIC/FPGA Design, [Lucas.Roosevelt@honeywell.com](mailto:Lucas.Roosevelt@honeywell.com))

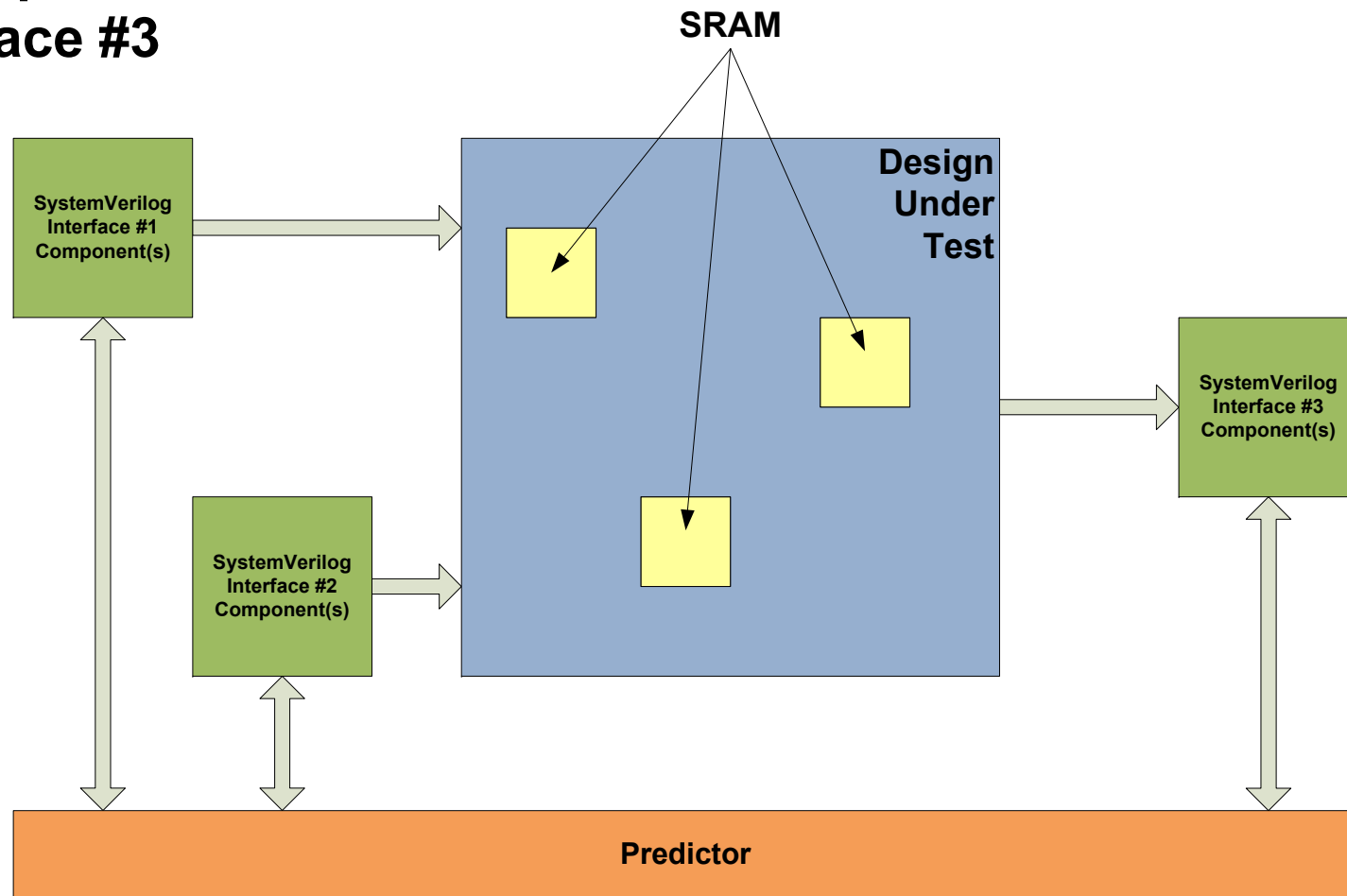
Presenter: Lucas Roosevelt

The Honeywell logo, consisting of the word "Honeywell" in a bold, red, sans-serif font.

- The goal of any verification effort is **Black-Box** verification
- **Black-Box** is defined as looking at the **Design Under Test (DUT)** only through its externally defined interfaces
- This can be difficult for many reasons
  - Large designs have vast amounts of internal state
  - Input effects can take large amounts of time to propagate to outputs
  - Temporal aspects can be hard to account for
  - Avoiding cycle accurate modeling
- **White-Box** testing can be used to bypass some of problems associated with **Black-Box** testing
  - Involves reaching in and looking at internal state
- Most people forget that there are generally holes in the **Black-Box** that still constitute external interfaces
- Internal Hard Macros, Memories, SerDes, are nothing but models
- They can easy be replaced/overloaded with better models
  - You can even do this with gates, its still just a model

# Typical “Black-Box” Testbench Strategy

- This example device has 3 external interfaces
- The testing is performed by
  - Sending in transactions on interfaces #1, and #2
  - Checking / Scoreboarding is performed when the results occur on interface #3.
- DUT operation can not be checked until results are seen on interface #3

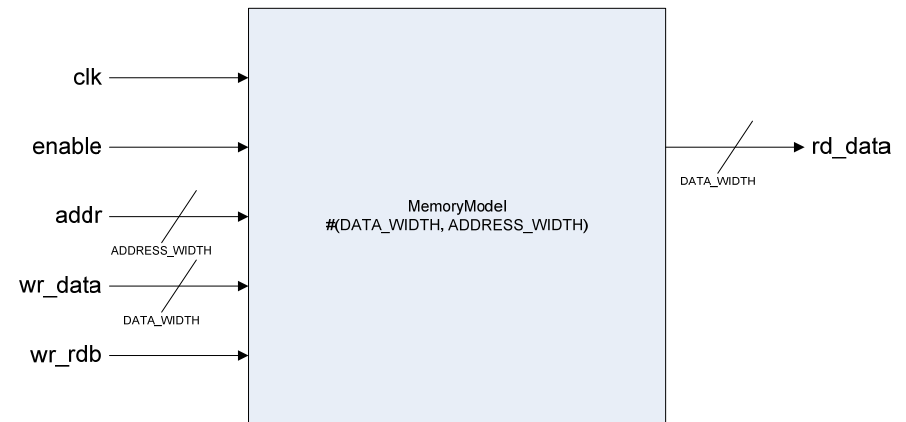


# Filling Memory Holes

- Many designs have internal memory

- These memories are used for

- Configuration (routing tables)
- Status (vast statistics)
- Transient Data Memory
- Queues



- These internal SRAMs are just models

- You can exploit this SRAMs to provide a look into your design
- This converts your black box model into something with holes

- A generic SRAM model can be created

- You can model whatever you need
- As of Questa 6.4 you can “dot-into” your VHDL and assign handles

- This allows you to ...

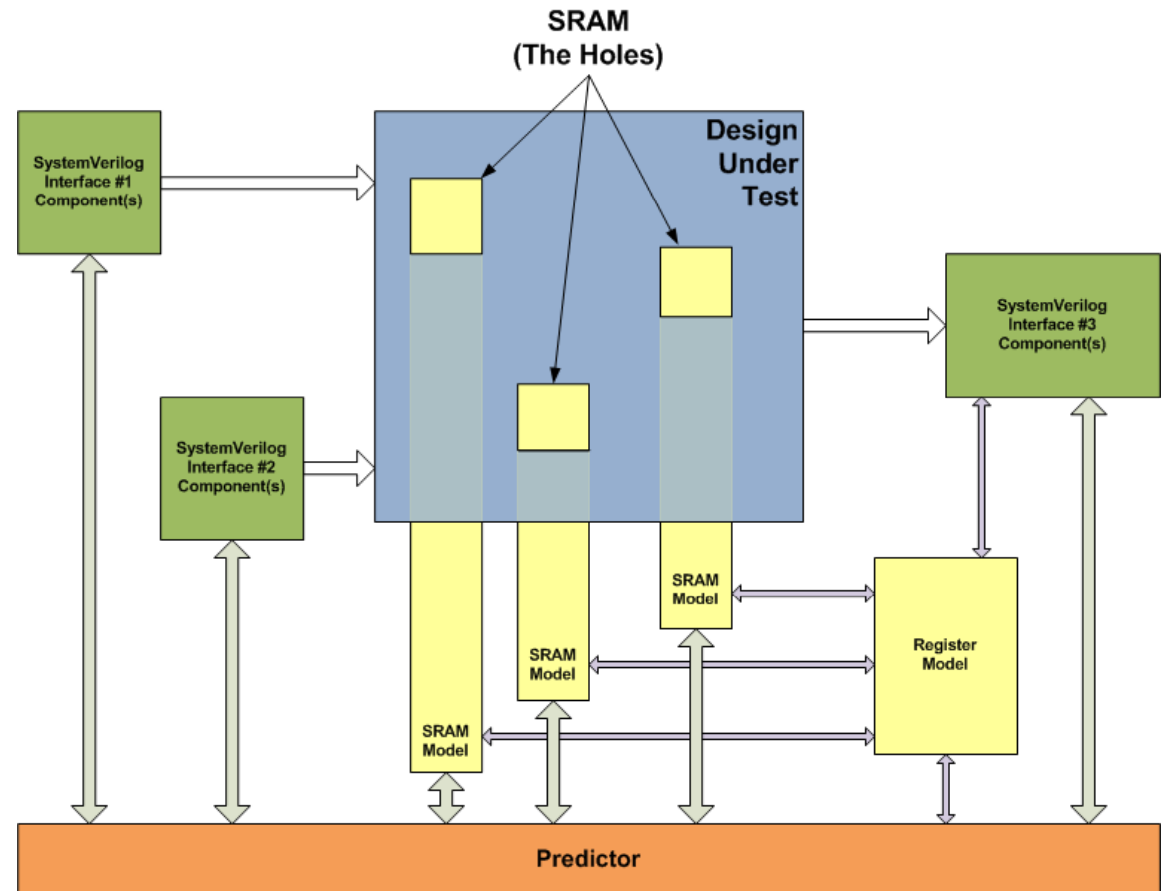
- Preload configuration (random or static)
- Observe status updates (without performing reads)
- Check various features at the SRAM boundary

- We have found the optimal solution is to accept handles to a base memory object

- The base memory class implements read(), write(), and reset().

# Utilizing the Memory Holes

- This time we exploit our memory holes
- Packets can be checked as they are written into the memories
  - Predictor and Model can scoreboard packets
- Configuration can be randomized, changed, and loaded by the test
- Incremental prediction
  - Interface #1,2 to memory
  - Memory to Interface #3
- Much faster error visibility
- Time to Bug is faster!



# Interacting with Data Storage

- **Say you have a large internal memory which holds packet data**
- **Using a memory hole you can scoreboard packets**
- **The predicted memory writes can be posted to a scoreboard**
  - Packet errors can be detected early
- **Provides an error injection source**
  - Requirements to
  - Errors injected into packet contents while in the memory
- **If the memory contains queues or other control structures**
  - Can be exploited to help predict arbitration, temporal events
- **If the storage structure of the memory is known it can be modeled and used!**

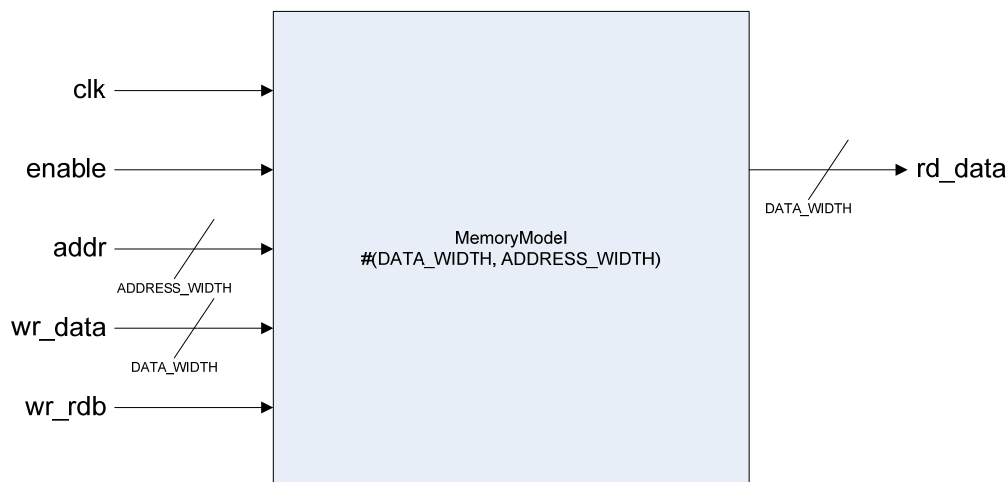
# Real-Time Status Checking

- **Checking large amounts of status can be tedious**
- **If its located in a drop-in memory it can be a breeze**
- **Say you have vast arrays of packet error counters**
  - Mapped into internal SRAM
- **Typically you would perform the following actions**
  - Generate packets with errors
  - Read all the status registers
  - Verify they are correct (predicted or directed)
- **The read-back is time consuming**
- **Utilizing a memory Hole you could scoreboard the error and never perform a read-back!**
- **Your tests can focus more time on generating good and error packets (testing)**
- **Less time wasted checking registers**

- **You can preload memories you have always done**
  - Write File I/O methods and load file contents during new()
  - Or make a function you call to load the file
- **Why preload using files if you don't have to?**
- **You can create a class with random variables**
  - Randomize the object
  - Set read() and write() to interact with the variables
- **You can then easily manipulate the configuration in the test**
  - Constrain the variables
  - Force them manually
- **This also allows the predictor and the DUT to physically have the exact same configuration registers.**
  - The DUT changes the registers, updates are reflected instantly
- **Far superior to preloading static files!**

# Simple Synchronous SRAM Model

- An example simple synchronous SRAM module
- Parameterizable and reusable
- This one SRAM model can be used everywhere
- Accepts a MemoryBlock handle “memory”
- Read and writes defer to the MemoryBlock
- Allows the test to hand in custom Memory Blocks
- We find it easy to quickly transition to a class



```
module SRAMModel #(int DATA_WIDTH,
                   int ADDR_WIDTH)
    input  logic clk, enable, wr_rdb;
    input  logic [ADDR_WIDTH-1:0] addr;
    input  logic [DATA_WIDTH-1:0] wr_data;
    output logic [DATA_WIDTH-1:0] rd_data;

    MemoryBlock #(DATA_WIDTH, ADDR_WIDTH)
                memory;

    always_ff @(posedge clk)
    begin
        if (enable) begin
            if (wr_rdb)
                memory.write(addr, wr_data);
            else
                rd_data = memory.read(addr);
        end
    end

endmodule
```

# MemoryBlock Detail

- **MemoryBlock** is an abstract (virtual) class
- Implements 3 pure virtual functions common to all memories
  - read(), write(), reset()
- Parameterized for Data and Address Width
- Ultimately use any memory model you prefer
- The SRAMModel accepts objects of base MemoryBlock
- You extend and implement the required functionality

```
virtual class MemoryBlock #(int DATA_WIDTH, int ADDR_WIDTH);  
  
    pure virtual function bit [DATA_WIDTH-1:0] read(bit [ADDR_WIDTH-1:0] A);  
    pure virtual function void write (bit [ADDR_WIDTH-1:0] A, bit [DATA_WIDTH-1:0] D);  
    pure virtual function void reset();  
  
endclass
```

# User Supplied MemoryBlock

- A routing Table
- Native width is 64-bits
- Randomized table contents
- Test constrains entry zero to assure it is valid
- Handle is given to MemoryBlock

```
module tb;
    MyConfig config;
    initial begin
        config = new();
        assert( config.randomize() with
                { rt[0].valid == 1'b1; } );
        $top.dut.mysram.memory = config;
        do_test();
    end
endmodule
```

```
class MyConfig extends MemoryBlock #(64, 10);
    rand struct {
        bit valid;
        bit [15:0] ports;
        bit [31:0] address;
    } rt [1024]; // Routing Table

    virtual function bit [63:0] read(bit [9:0] A);
        return({rt[A].valid, 15'h0, rt[A].ports, rt[A].address});
    endfunction

    virtual function void write(bit [9:0] A, bit [63:0] D);
        {rt[A].valid, rt[A].ports, rt[A].address} = {D[63], D[47:32], D[31:0]};
    endfunction
endclass
```

# MemoryBlock Helper Classes

- **Physical memories often have different organizations virtually and physically**
  - An internal configuration memory might be 72-bits wide
  - Register reads might map this to 32-bit accesses
- **Helper Classes can be developed to provide different organizations of the same data**
  - Map a wide memory into two smaller width memories
  - Map a deep memory into two shallow memories
  - These can be made generic and re-usable
- **Helper classes can also add functionality**
  - Convert a MemoryBlock to Read-Only
  - Add read-to-clear functionality
  - Add checkers (assert failures on invalid DUT access)
  - Add logging (log writes to a file)
- **You can tie multiple classes and/or memories together**
  - Writes to one effect both
- **Whatever is required to be modeled by your design**

# There is no limit to what you can do

- You have a device with vast configuration in internal memory
- At Power-up the configuration is read from an external flash
- The external flash is an 8-bit interface
- The internal configuration is in a 64-bit wide memory
- The device interface #1 provides a read/write protocol that operates on 32-bits
- You have the same MemoryBlock handle in all locations
- You can quickly change from preloading to loading
- The DUT SRAM block can even check the configuration as it loads.
- Predictor can perform reads of the model to generated expected responses

