Results Checking Strategies with Portable Stimulus

Tom Fitzpatrick
Matthew Balance
Mentor, A Siemens Business
Portable Stimulus Vision

• Single specification of test intent
  – With capabilities supporting block to SoC
  – Support for multiple engines

• Capture "scenario space"
  – Data space
  – Operational sequence
  – Resource dependencies

• Enable automated test generation
  – Goal-driven stimulus generation
  – Target stimulus across environments
Elements of a Scenario-Level Test

- Identify target design behaviors to be exercised
- What data do these actions require/produce?
- With what component(s) of the design are these actions associated?
- What system resources are required to accomplish these actions?

These questions are independent of the implementation details of the DUT
Scenario Creation (UART)

- Design is a UART
  - Transmits and receives data serially
  - Wishbone bus interface to access registers
  - Interrupt request output
  - rx_ready and tx_ready output signals

- IP-level testbench is entirely VIP based
  - Wishbone VIP to program registers
  - Serial VIP to receive and transmit data
Simple Example, Step 1: Identify Behaviors

- Transmit data packet via Serial port
  - WB VIP: Generate packet
  - UART: Get packet from WB, send to Serial port
  - Serial VIP: Receive & check packet

- Receive data packet via Serial port
  - Serial VIP: Generate packet, send to UART
  - UART: Receive packet from Serial port, put to WB
  - WB VIP: Get packet from UART, check data

- Behaviors are modeled in PSS as actions
Data Flow Objects Imply Scheduling

- genpkt
- rx
- tx_info
- mem2tx
- dat_i

Stream implies parallel
Buffer implies serial
Activities Define Explicit Scheduling

```plaintext
action test_tx_a {
    gendata_a      genpkt;
    mem2tx_a       mem2tx;
    uart_agent_rx_a rx;

    activity {
        genpkt;
        parallel {
            mem2tx;
            rx;
        }
        bind genpkt.dat_o mem2tx.dat_i;
        bind mem2tx.tx_info rx.rx_info;
    }
}
```
Putting It Together
Data Generation & Checking – Data Types

```
struct data_mem_t {
  rand bit[31:0] addr;
  rand bit[31:0] sz;
}
```

```
buffer data_ref_mem_b : data_mem_t {
  rand bit[31:0] ref;
}
```

```
stream data_ref_s {
  rand bit[31:0] sz;
  rand bit[31:0] ref;
}
```

Notice anything about these data types? No data fields!
Mapping to UVM

Serial VIP

- tx
- data_ref_s
- tx
- uvm_serial_tx_seq
- UART
- scoreboard
- Most likely not present at sub/system-level

WB VIP

- rx2mem
- data_ref_mem_b
- chkpkt

Most likely not present at sub/system-level
Data Generation

class uart_serial_tx_seq extends uart_serial_seq_base;
byte unsigned data[$];
task body();
  uart_serial_seq_item item =
    uart_serial_seq_item::type_id::create("item");

  for (int i=0; i<data.size(); i++) begin
    start_item(item);
    item.randomize();
    finish_item(item);
  end
endtask

endclass
Data Generation

```plaintext
class uart_serial_tx_seq extends uart_serial_seq_base;
byte unsigned data[$];
task body();
    uart_serial_seq_item item =
        uart_serial_seq_item::type_id::create("item");

    for (int i=0; i<data.size(); i++) begin
        item.data = data[i];  // repeatable
        start_item(item);
        finish_item(item);
    end
endtask
...
endclass
```
Data Generation

task tx(int unsigned seed, int unsigned bytes, int unsigned stride);
    uart_serial_tx_seq tx_seq = uart_serial_tx_seq::type_id::create();
    pvm_rand r = new(seed);

    for (int i=0; i<bytes; i+=stride) begin
        tx_seq.data[0] = r.next(); // Generate next repeatable-random value for r
        tx_seq.start(m_agent.m_seqr); // tx_seq will transmit one byte for each call
        // Spin the counter ahead for nonzero strides
        for (int j=1; j<stride; j++) begin
            void'(r.next());
        end
    end
endtask
Data Generation

class pvm_rand;
    local int unsigned m_seed;

    function new(int unsigned seed);
        m_seed = seed;
    endfunction

    function int unsigned next();
        m_seed ^= (m_seed << 13);
        m_seed ^= (m_seed >> 17);
        m_seed ^= (m_seed << 5);
        return m_seed;
    endfunction
endclass
Data Checking

```verilog
task automatic checkdata(bit[31:0] seed, addr, sz)
    pvm_rand r = new(seed);

    for (int i=0; i<sz; i++) begin
        int unsigned v = r.next(); // Generate same data stream from given seed
        byte unsigned d;
        pvm_ioread8(d, addr+i);
        if (d != v) begin
            `uvm_error("checkdata_mask" ...);
        end
    end
endtask
```
Generating the UVM Test Sequence

```
begin
  fork
    wb_uart_dev_rx2mem(0, 27566, 5);
    uart_agent_dev_tx(1, 3027603715, 5, 1);
  join
    pvm_uvm_pkg::checkdata(3027603715, 27566, 5);
end

action test_rx_a {
  rx2mem_a          rx2mem;
  tx_a              tx;
  chkdata_a         chkpkt;

  activity {
    parallel {
      tx;
      rx2mem;
    }
    chkpkt;
    bind chkpkt.dat_i rx2mem.dat_o;
    bind rx2mem.rx_info tx.tx_info;
  }
}
```
The Rules: Think Ahead!

- IP and Subsystem often use a scoreboard for checking
  - Observe data going in and out of the block
  - Observe status signals
  - Compare data to confirm correctness
  - Compare status signals with a reference model

- This isn’t feasible in an SoC-level environment
  - Passing data is time consuming
  - Visibility is limited

- A portable checking strategy works in all environments

- Can be augmented by detailed checkers at IP
  - Example: check that UART status signals match FIFO fill
Additional DMA Actions

- **data_ref_mem_b** → **mem2mem** → **data_ref_mem_b**
- **data_ref_s** → **dev2mem** → **data_ref_mem_b**
- **data_ref_mem_b** → **mem2dev** → **data_ref_s**

Still just passing *ref* pointers around
void wb_periph_subsys_test(void) {
...{
    infactpss_thread_t branch_1 = infactpss_thread_create(&thread_func_5);
    infactpss_thread_t branch_2 = infactpss_thread_create(&thread_func_6);
    infactpss_thread_join(branch_1);
    infactpss_thread_join(branch_2);
}
pvm_checkdata_mask(2362696682, 1612443855, 14, 255);
...
}
void thread_func_5(void) {
    uart_agent_dev_tx(0, 2362696682, 14, 1);
}
void thread_func_6(void) {
    wb_uart_dev_rx2mem(2, (void *)1612443855, 14);
}
The Keys to Results Checking in PSS

• Keep PSS data model as simple as possible
• Rely on target-specific implementation for generation and checking
  – Based on key information passed from PSS model
  – Easily ported across platforms
• Base checking strategy on limited visibility in system-level model
  – Block-level scoreboards or other checking likely won't be available
• A little forethought goes a long way