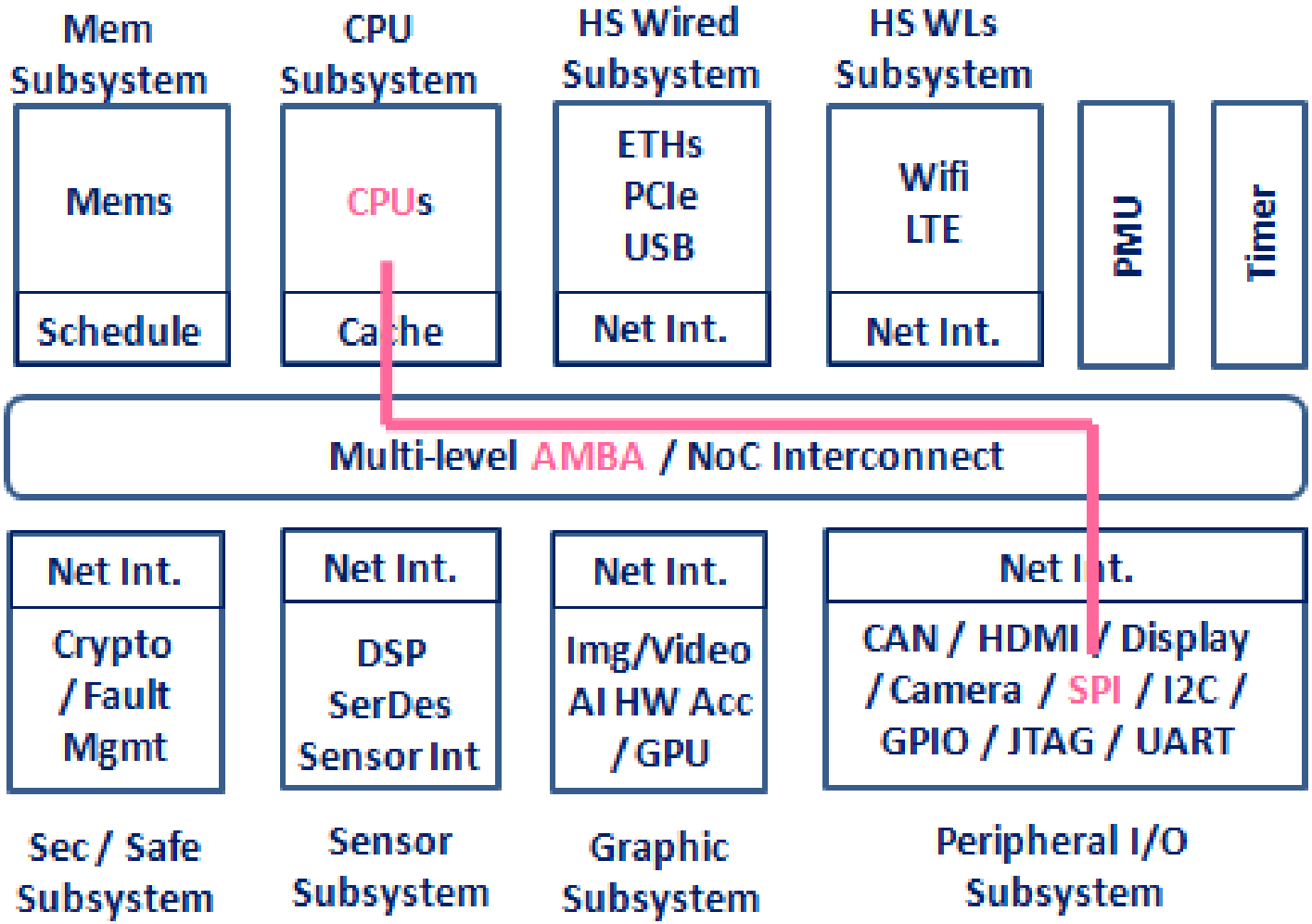
SPI Bus (Serial Peripheral Interface Bus)

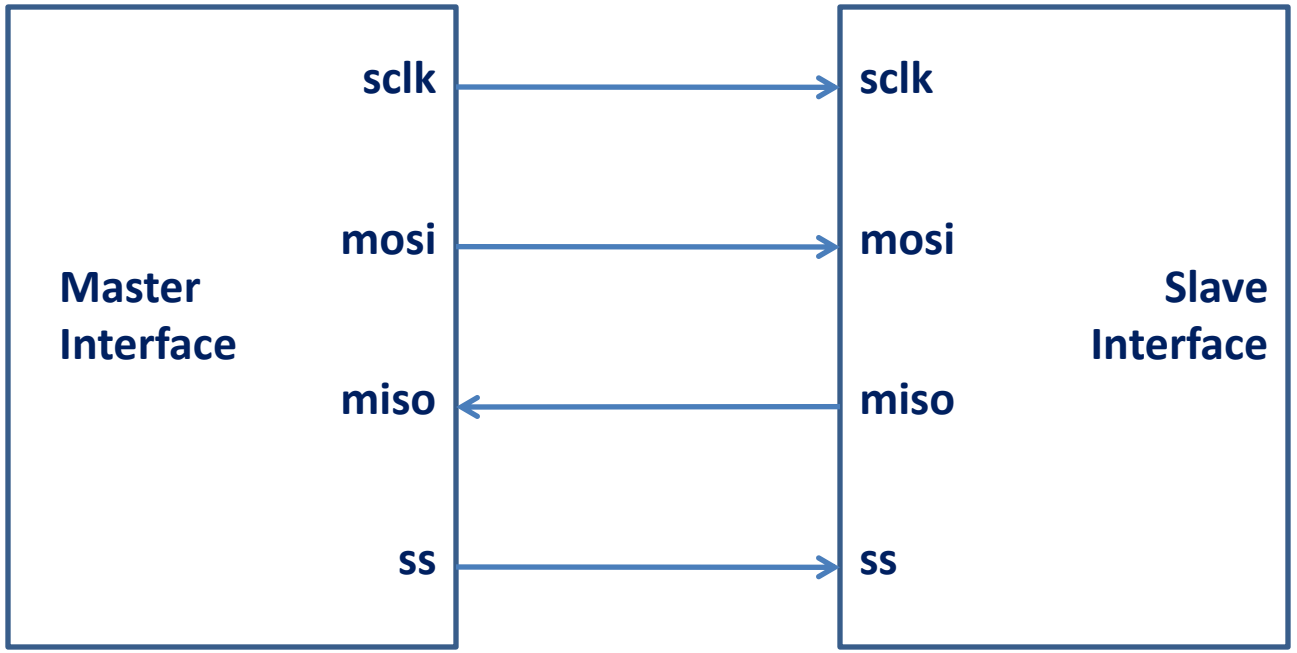
SoC – SPI (for slow PCs)

Tuan Nguyen-viet

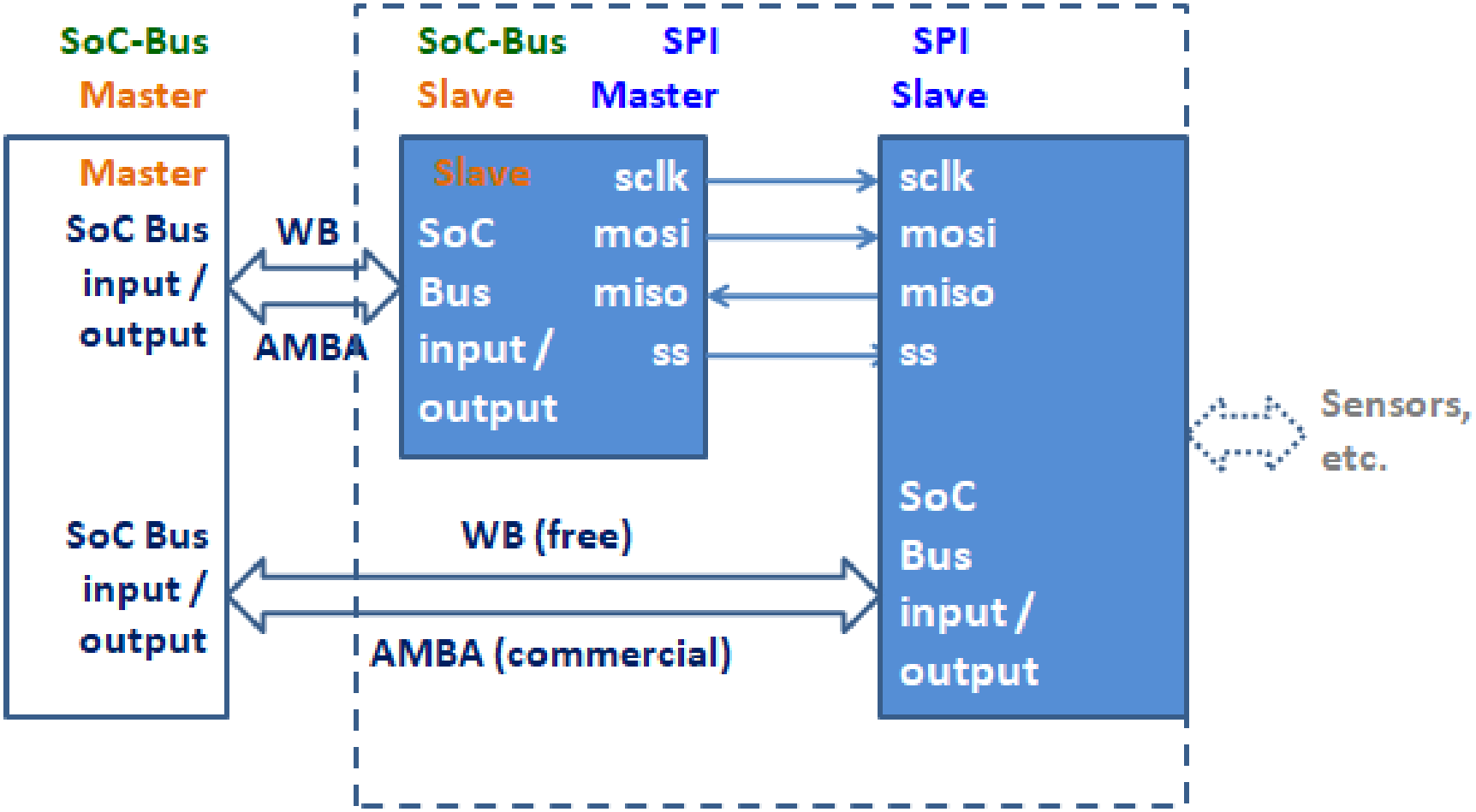
Processor-SPI Interaction



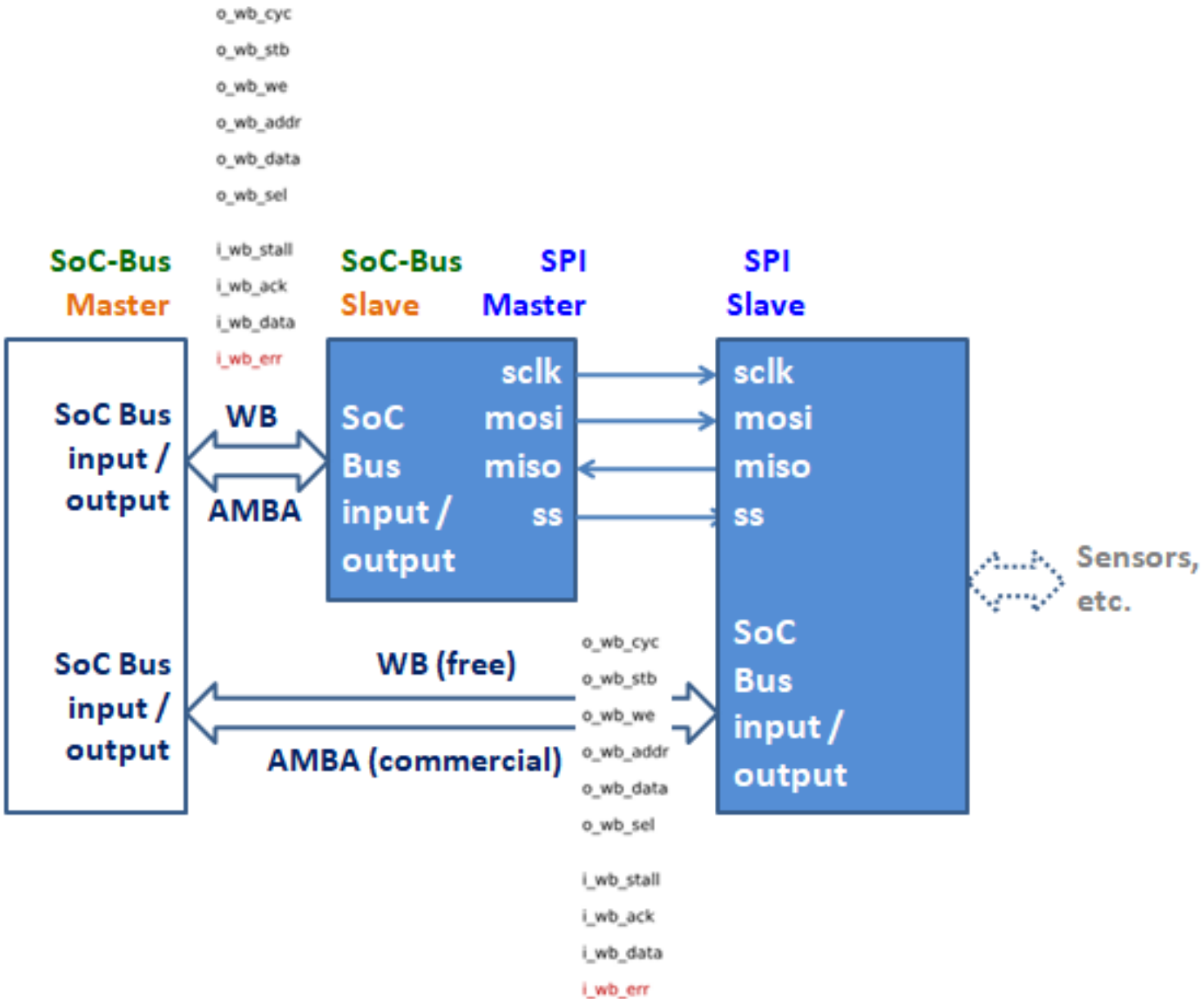
SPI Master / Slave



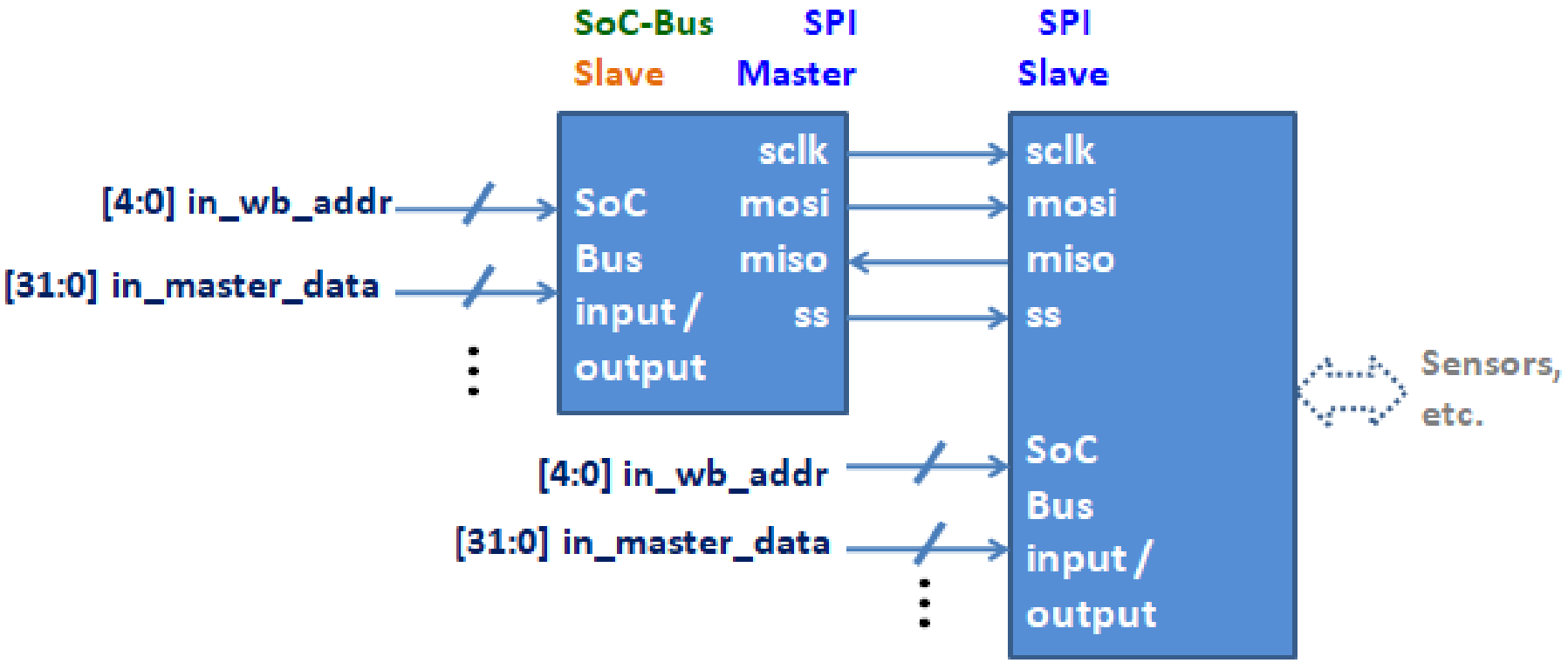
DUT – SoC SPI


SoC-Bus
Master
SoC-Bus
Slave
SPI
Master
SPI
Slave
Master
SoC Bus
input /
output
WB
AMBA
Slave
SoC
Bus
input /
output
sclk
mosi
miso
ss
sclk
mosi
miso
ss
Sensors,
etc.
SoC Bus
input /
output
WB (free)
AMBA (commercial)
SoC
Bus
input /
output

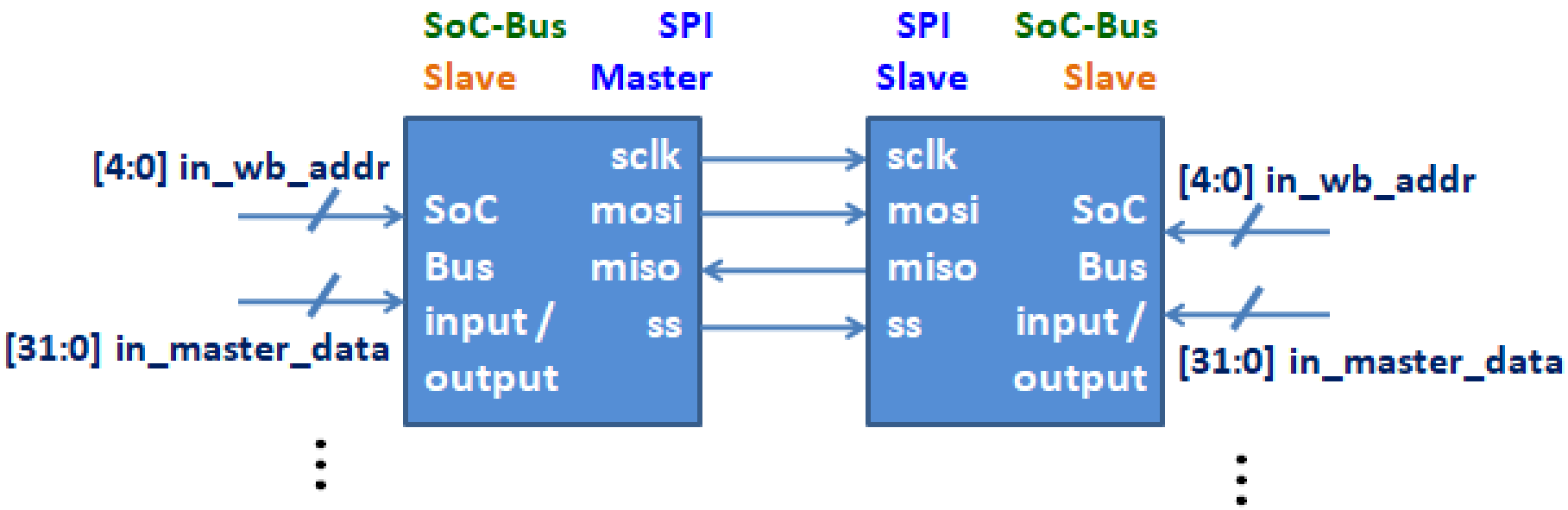
DUT – SoC SPI


o_wb_cyc
o_wb_stb
o_wb_we
o_wb_addr
o_wb_data
o_wb_sel
i_wb_stall
i_wb_ack
i_wb_data
i_wb_err
SoC-Bus Master
SoC-Bus Slave
SPI Master
SPI Slave
SoC Bus input / output
WB
AMBA
SoC Bus input / output
sclk
mosi
miso
ss
sclk
mosi
miso
ss
Sensors, etc.
SoC Bus input / output
WB (free)
AMBA (commercial)
o_wb_cyc
o_wb_stb
o_wb_we
o_wb_addr
o_wb_data
o_wb_sel
i_wb_stall
i_wb_ack
i_wb_data
i_wb_err
SoC Bus input / output

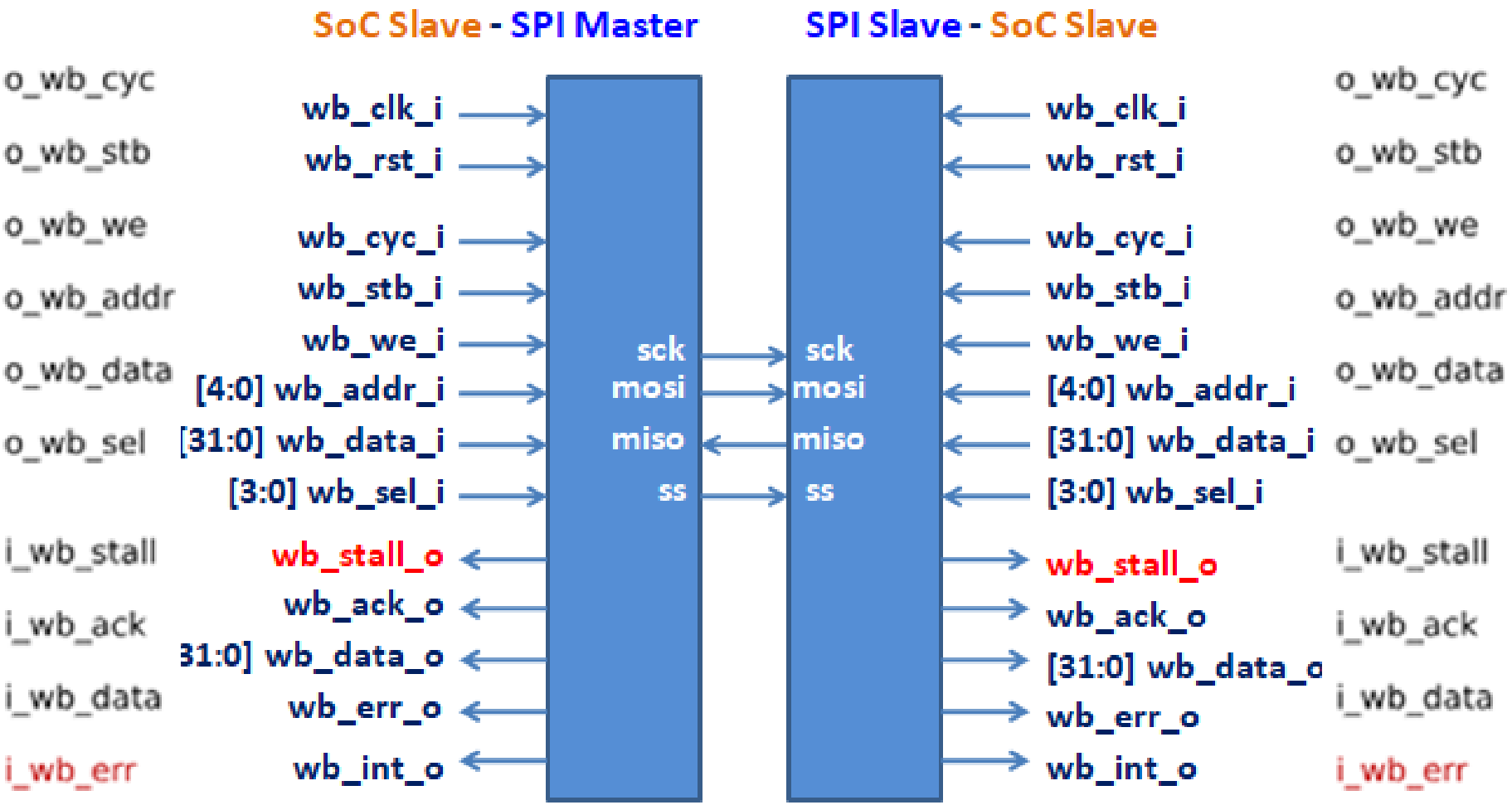
DUT – SoC SPI


SoC-Bus
Slave
SPI
Master
SPI
Slave
[4:0] in_wb_addr
[31:0] in_master_data
SoC
Bus
input /
output
sclk
mosi
miso
ss
sclk
mosi
miso
ss
Sensors,
etc.
[4:0] in_wb_addr
[31:0] in_master_data
SoC
Bus
input /
output

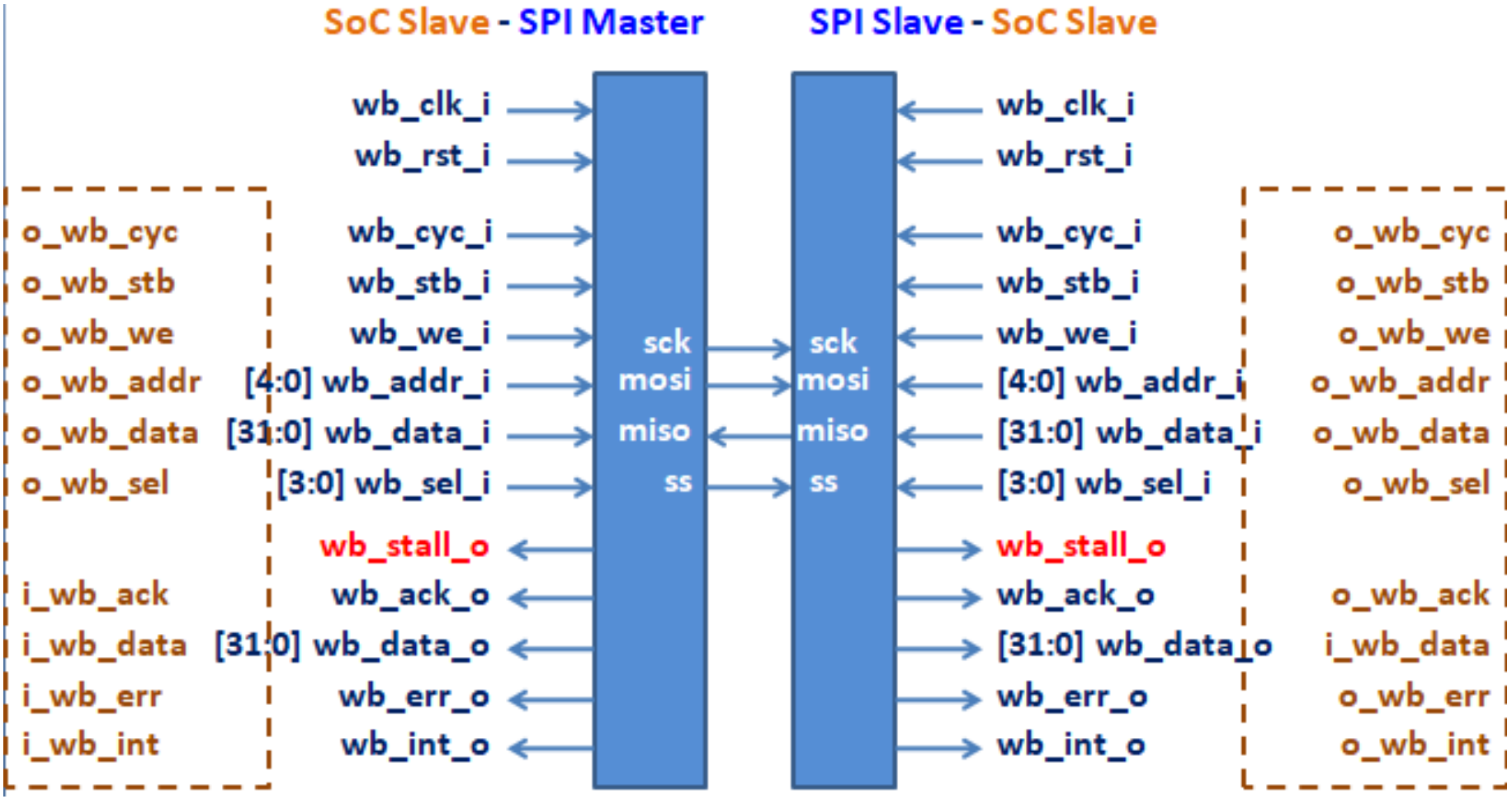
DUT – SoC SPI


SoC-Bus
Slave
SPI
Master
SPI
Slave
SoC-Bus
Slave
[4:0] in_wb_addr
[31:0] in_master_data
SoC
Bus
input /
output
sclk
mosi
miso
ss
sclk
mosi
miso
ss
SoC
Bus
input /
output
[4:0] in_wb_addr
[31:0] in_master_data

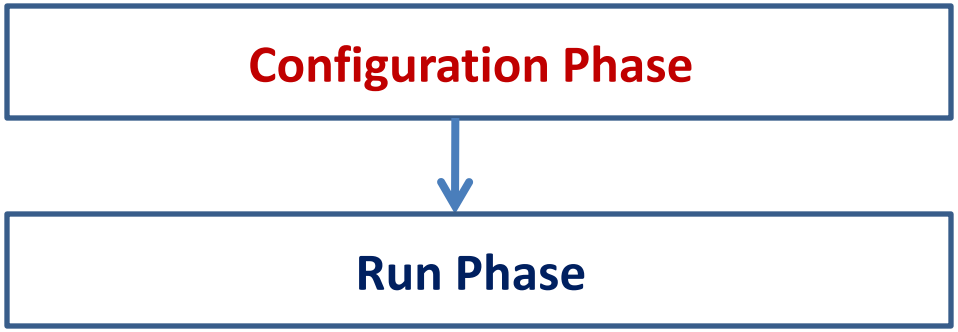
DUT – SoC SPI


SoC Slave - SPI Master
SPI Slave - SoC Slave
o_wb_cyc
o_wb_stb
o_wb_we
o_wb_addr
o_wb_data
o_wb_sel
i_wb_stall
i_wb_ack
i_wb_data
i_wb_err
wb_clk_i
wb_rst_i
wb_cyc_i
wb_stb_i
wb_we_i
[4:0] wb_addr_i
[31:0] wb_data_i
[3:0] wb_sel_i
wb_stall_o
wb_ack_o
31:0] wb_data_o
wb_err_o
wb_int_o
sck
mosi
miso
ss
sck
mosi
miso
ss
wb_clk_i
wb_rst_i
wb_cyc_i
wb_stb_i
wb_we_i
[4:0] wb_addr_i
[31:0] wb_data_i
[3:0] wb_sel_i
wb_stall_o
wb_ack_o
[31:0] wb_data_o
wb_err_o
wb_int_o
o_wb_cyc
o_wb_stb
o_wb_we
o_wb_addr
o_wb_data
o_wb_sel
i_wb_stall
i_wb_ack
i_wb_data
i_wb_err

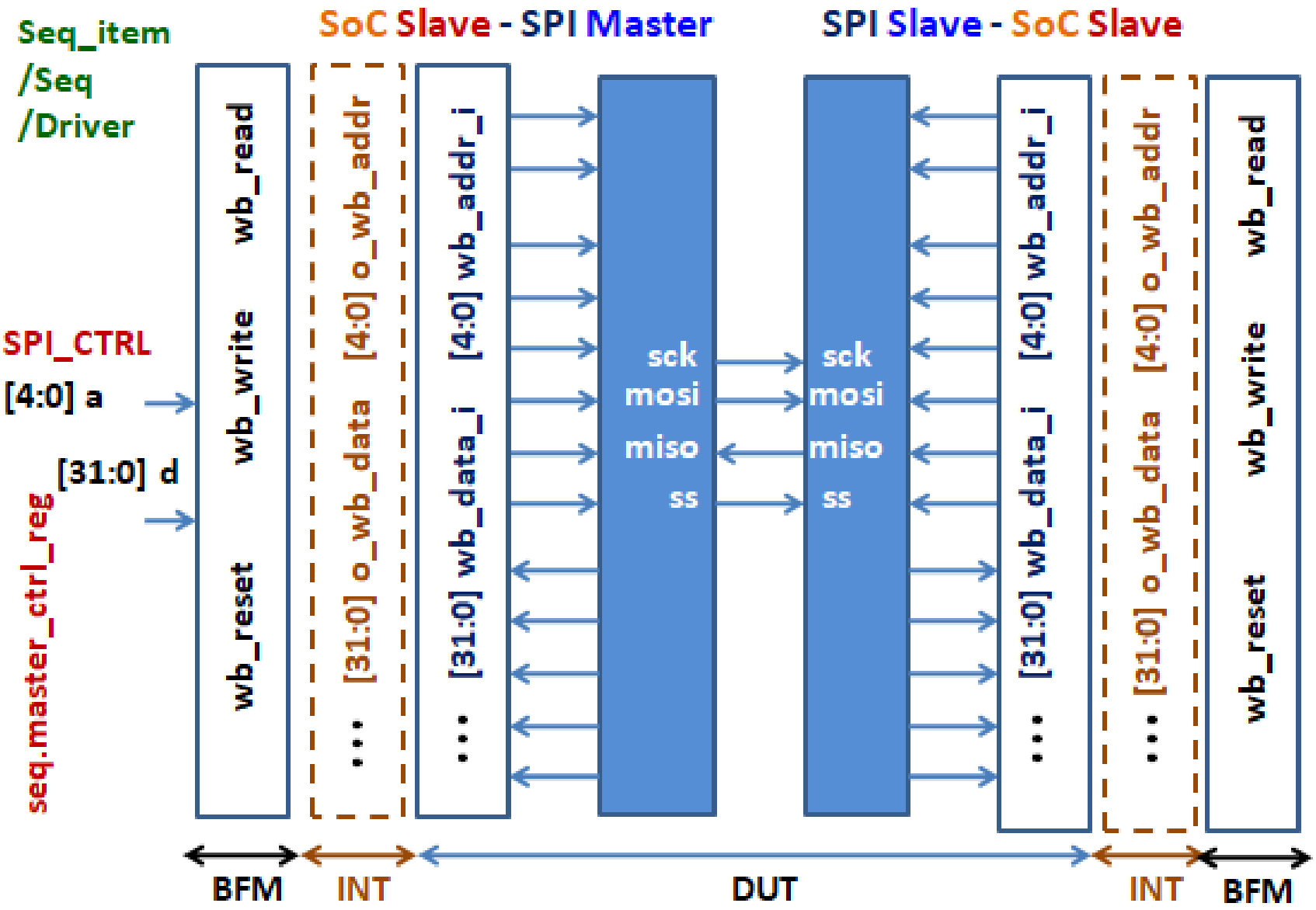
Interface


SoC Slave - SPI Master
SPI Slave - SoC Slave
wb_clk_i
wb_rst_i
wb_cyc_i
wb_stb_i
wb_we_i
[4:0] wb_addr_i
[31:0] wb_data_i
[3:0] wb_sel_i
wb_stall_o
wb_ack_o
[31:0] wb_data_o
wb_err_o
wb_int_o
sck
mosi
miso
ss
o_wb_cyc
o_wb_stb
o_wb_we
o_wb_addr
o_wb_data
o_wb_sel
i_wb_ack
i_wb_data
i_wb_err
i_wb_int

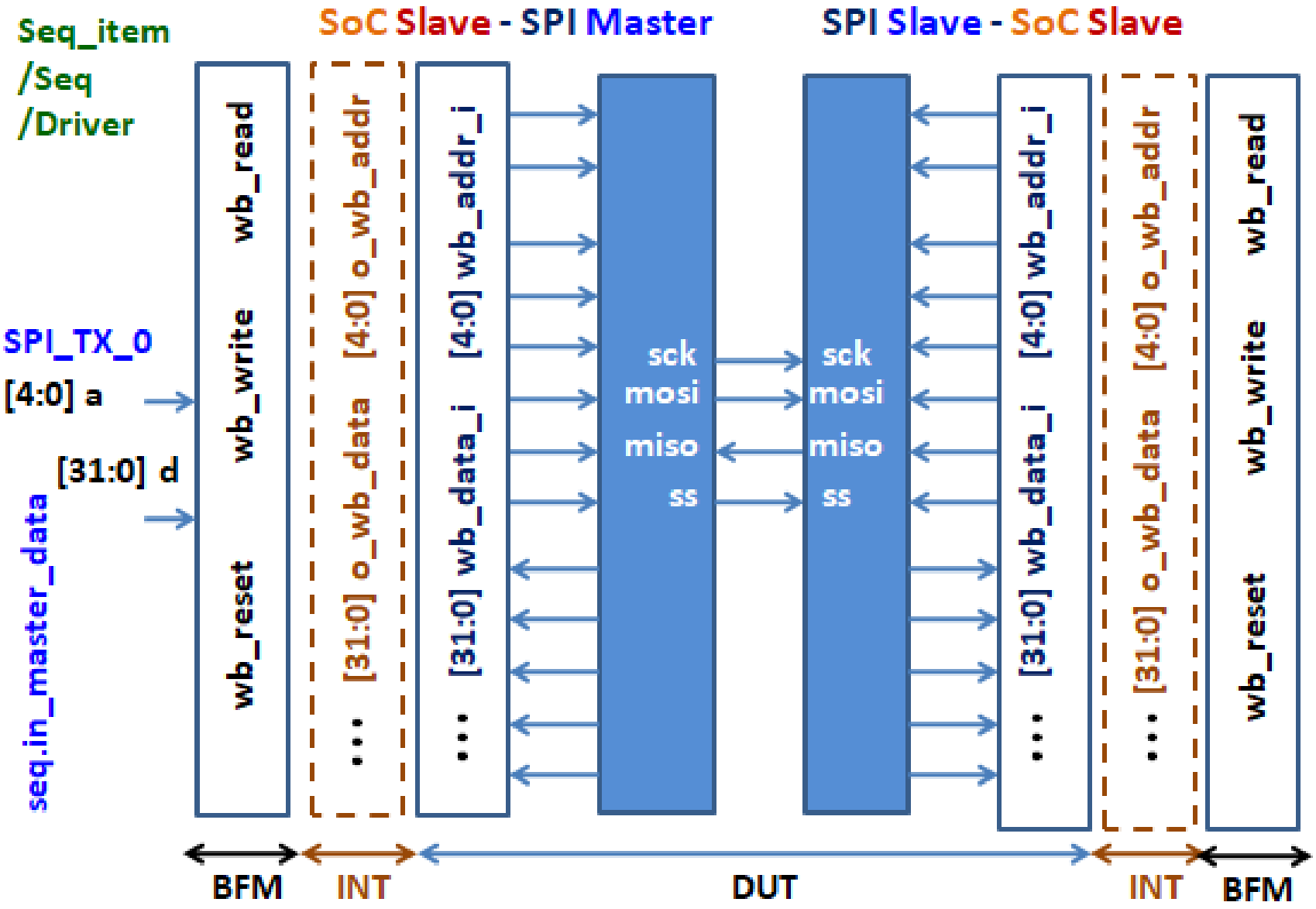
SPI Phases

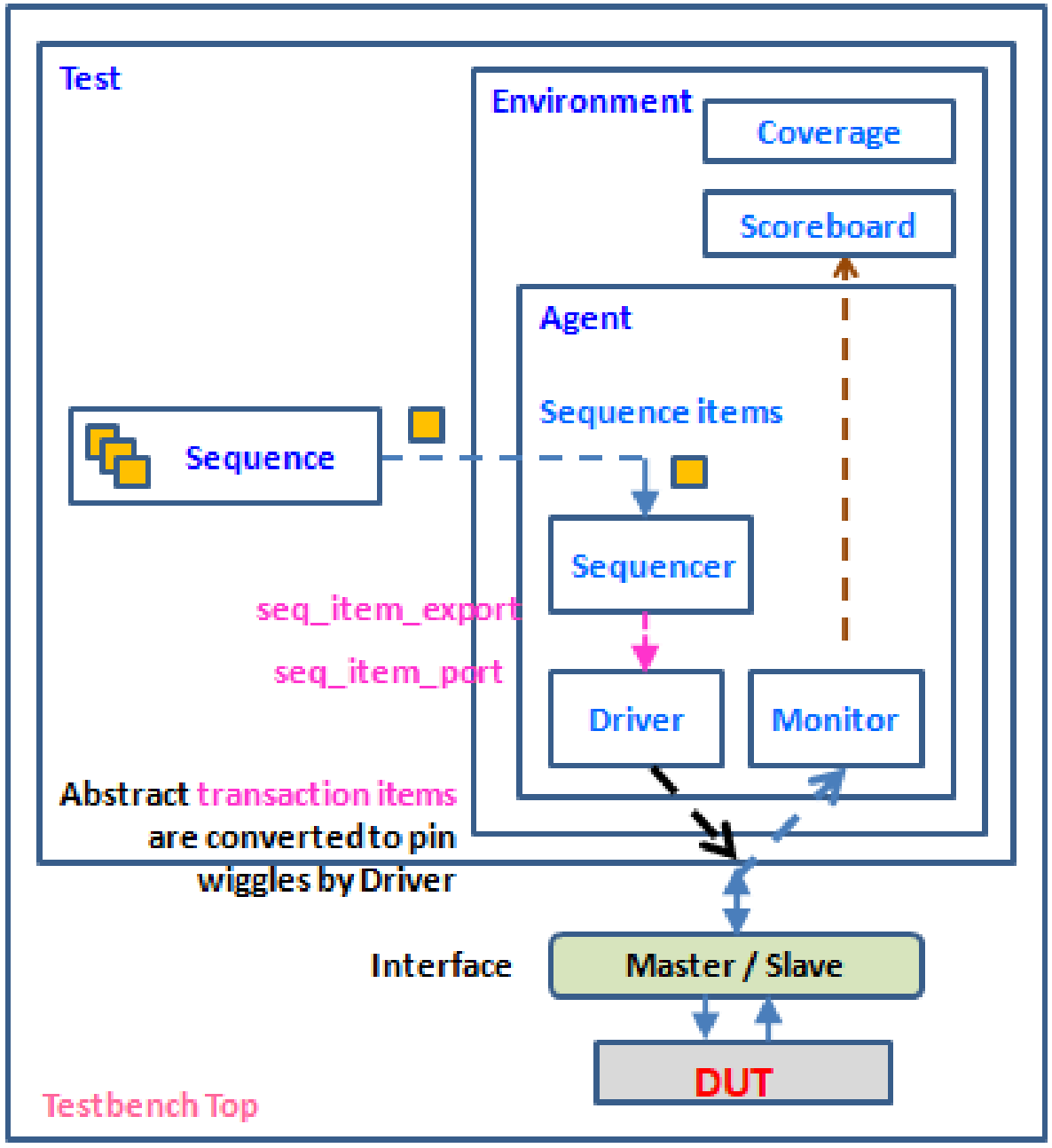


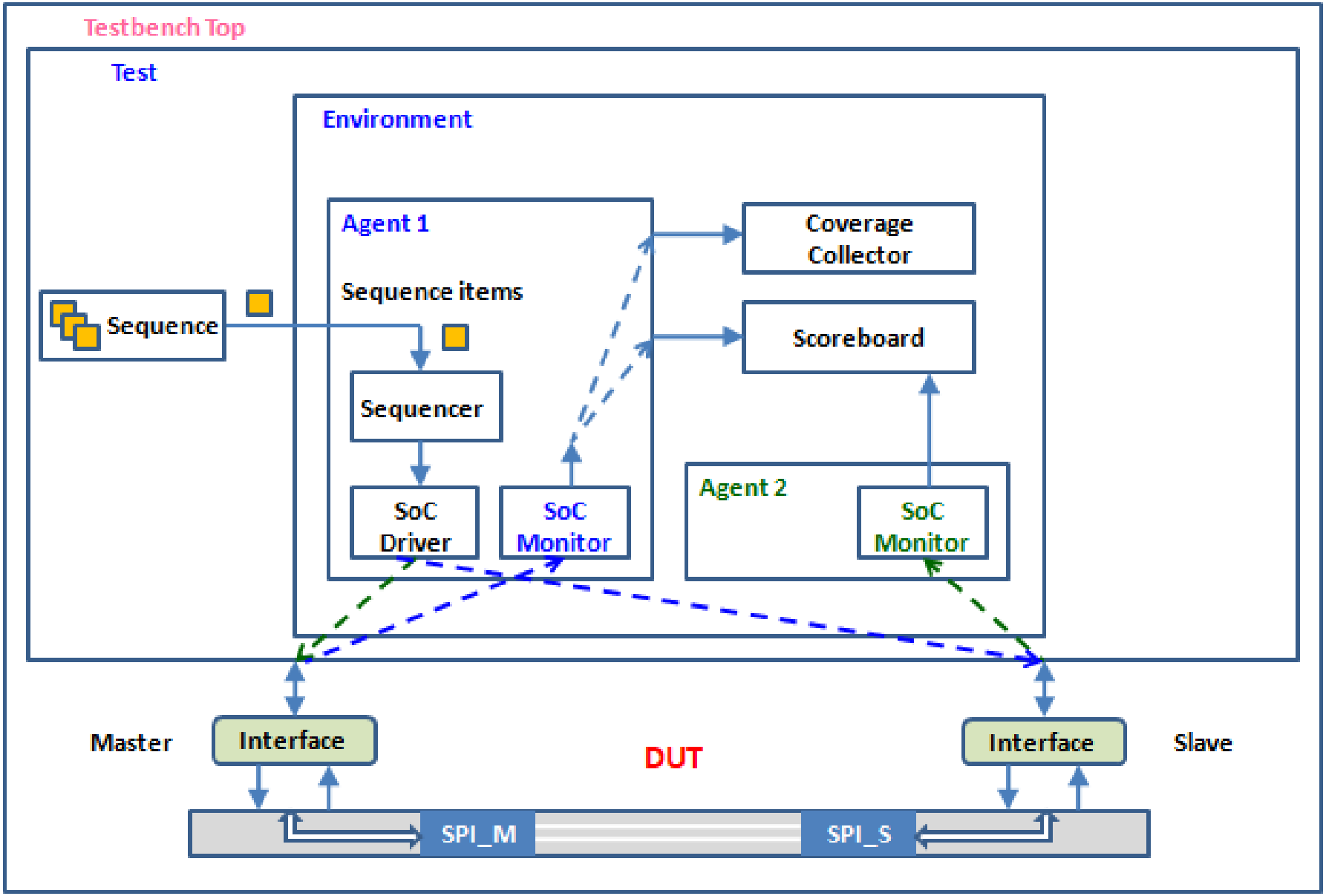
SoC BFM/Interface/DUT

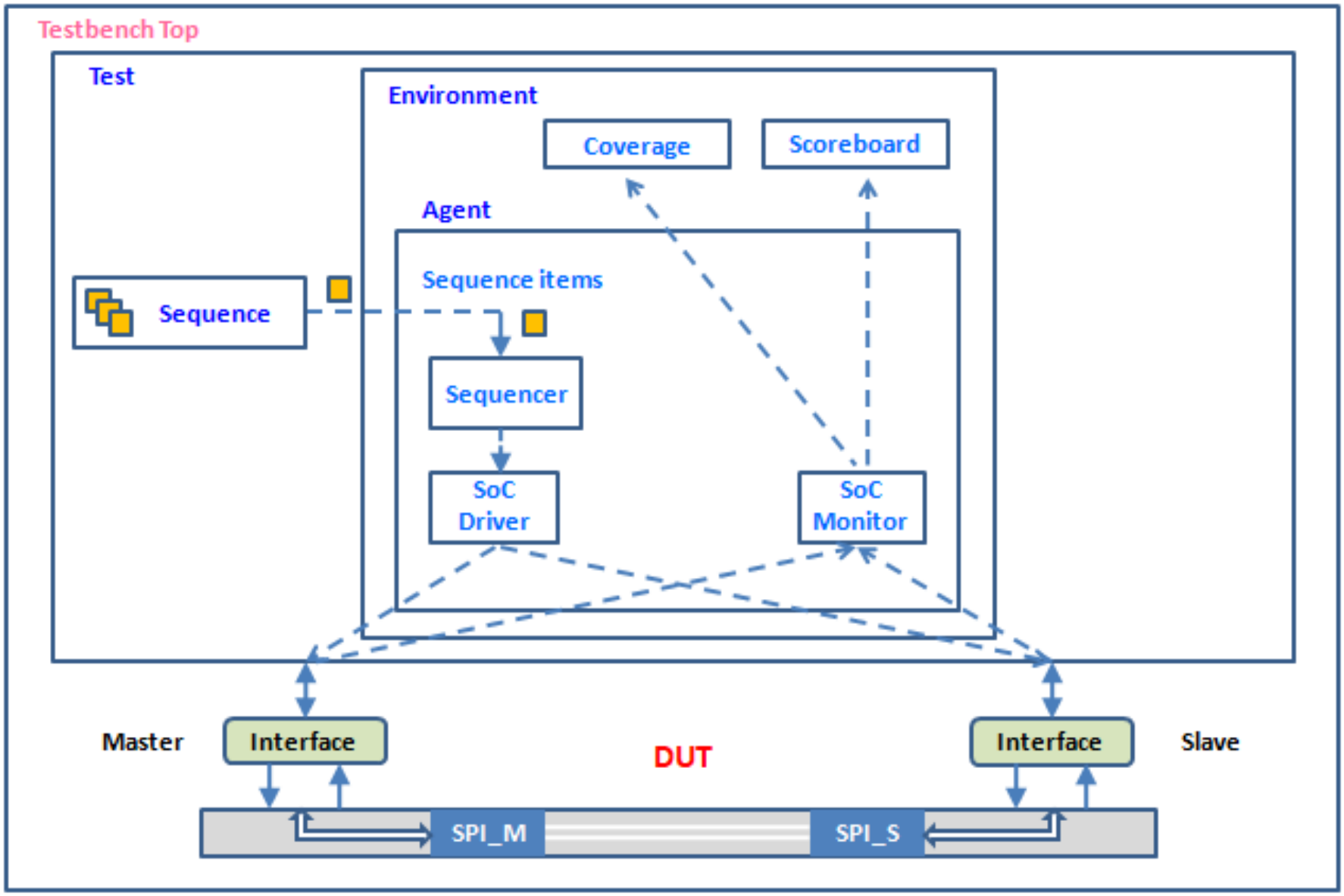


SoC BFM/Interface/DUT

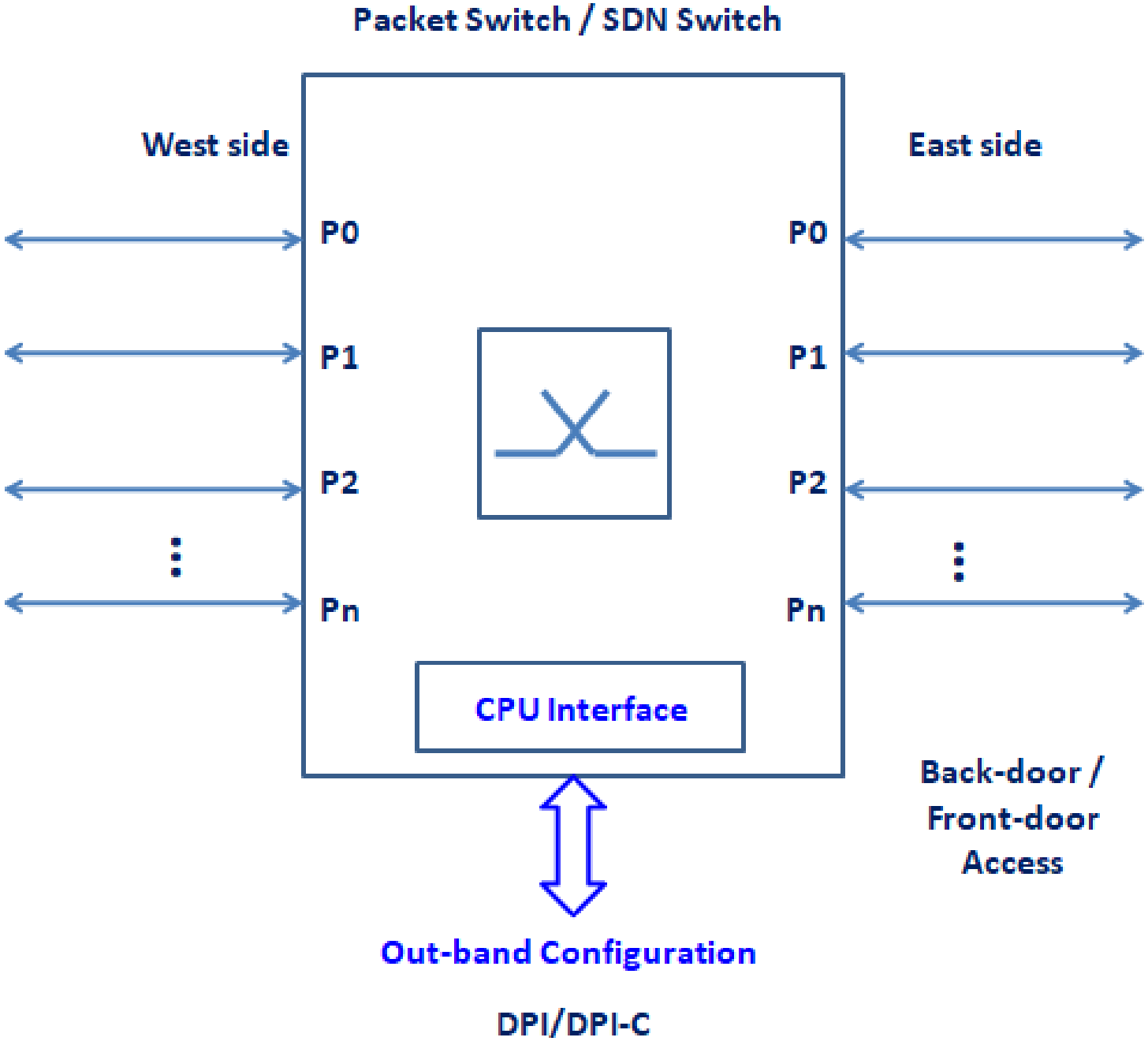


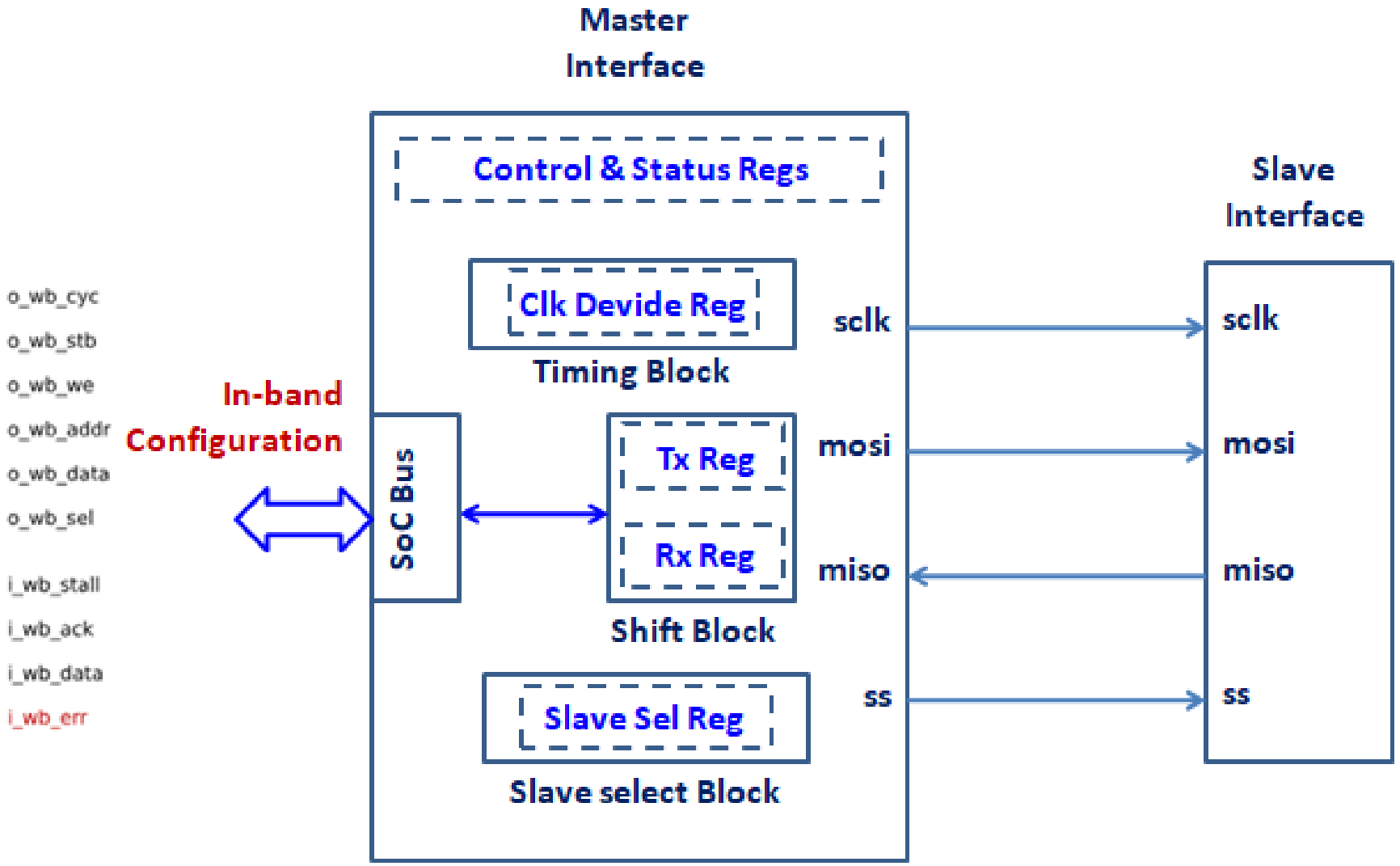

Test
Environment
Coverage
Scoreboard
Agent
Sequence items
Sequence
Sequencer
seq_item_export
seq_item_port
Driver
Monitor
Abstract transaction items
are converted to pin
wiggles by Driver
Interface
Master / Slave
DUT
Testbench Top


Testbench Top
Test
Environment
Agent 1
Sequence items
Sequence
Sequencer
SoC Driver
SoC Monitor
Coverage Collector
Scoreboard
Agent 2
SoC Monitor
Master
Interface
DUT
Interface
Slave
SPI_M
SPI_S

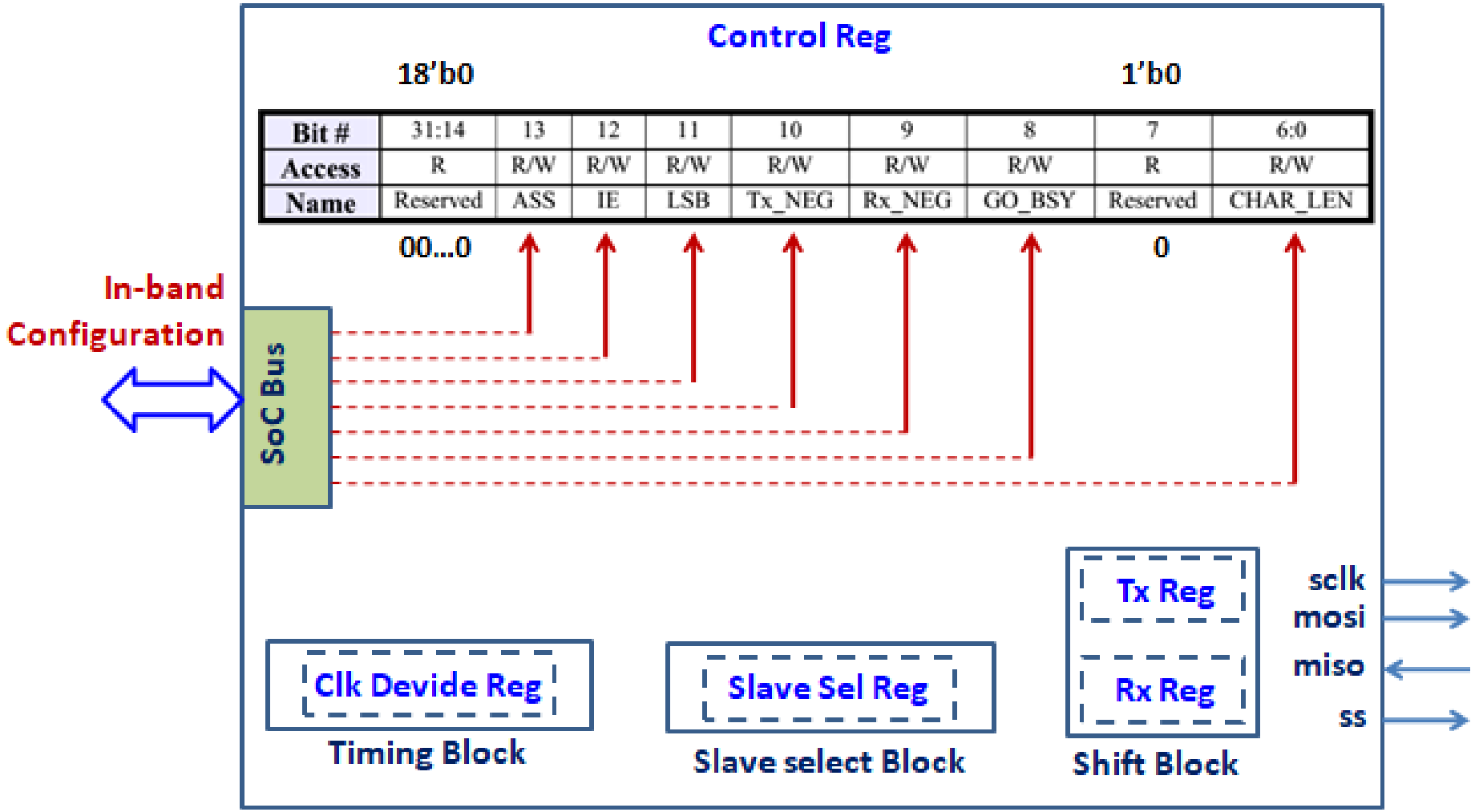

Testbench Top
Test
Environment
Coverage
Scoreboard
Agent
Sequence items
Sequence
Sequencer
SoC
Driver
SoC
Monitor
Master
Interface
DUT
Interface
Slave
SPI_M
SPI_S

SPI Registers

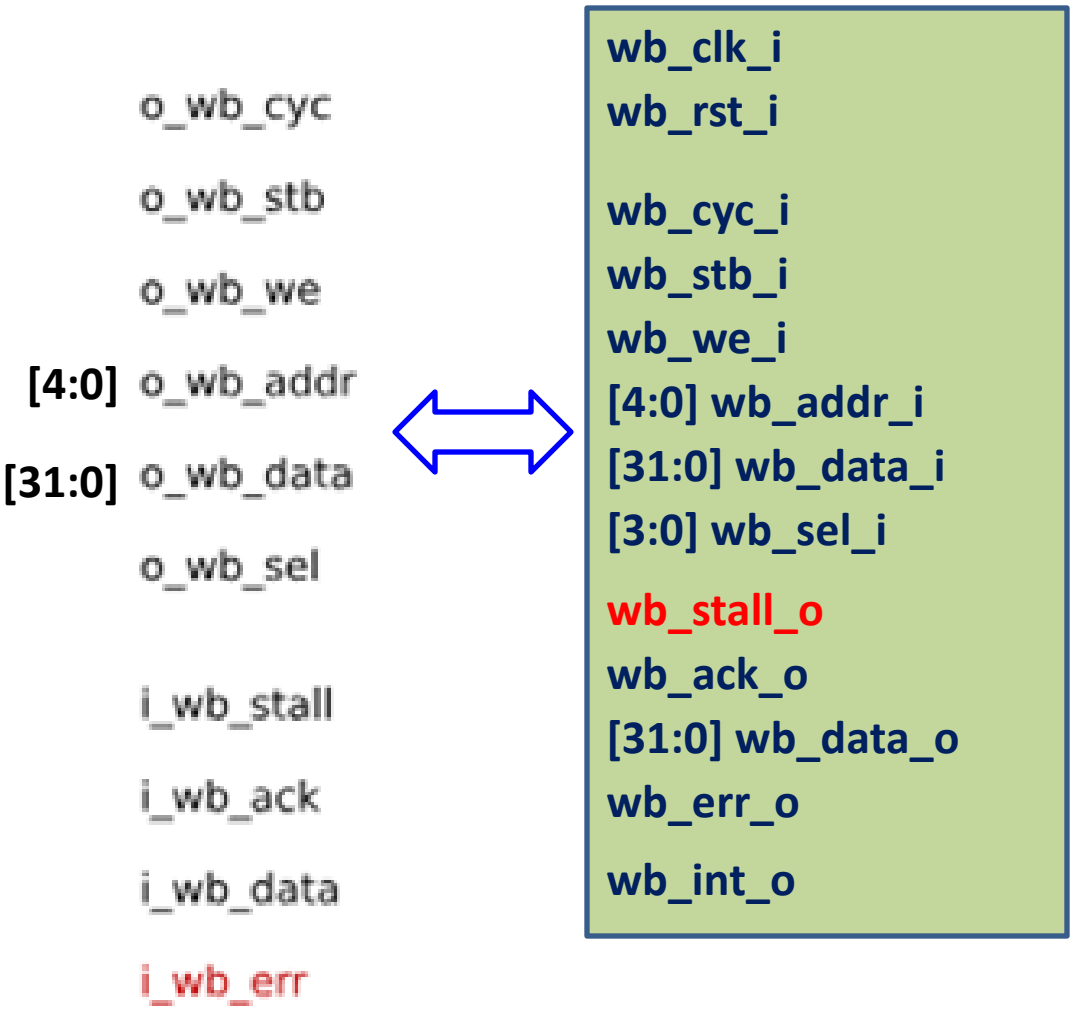

Packet Switch / SDN Switch
West side
East side
P0
P1
P2
Pn
P0
P1
P2
Pn
CPU Interface
Back-door /
Front-door
Access
Out-band Configuration
DPI/DPI-C


Master
Interface
Control & Status Regs
Clk Devide Reg
Timing Block
Tx Reg
Rx Reg
Shift Block
Slave Sel Reg
Slave select Block
SoC Bus
In-band
Configuration
o_wb_cyc
o_wb_stb
o_wb_we
o_wb_addr
o_wb_data
o_wb_sel
i_wb_stall
i_wb_ack
i_wb_data
i_wb_err
sclk
mosi
miso
ss
Slave
Interface
sclk
mosi
miso
ss

SPI Master Interface



SoC Bus (Slave) of SPI Master Interface



DUT – SoC WB Bus Interface

Port	Width	Direction	Description
wb_clk_i	1	Input	Master clock
wb_rst_i	1	Input	Synchronous reset, active high
wb_adr_i	5	Input	Lower address bits
wb_dat_i	32	Input	Data towards the core
wb_dat_o	32	Output	Data from the core
wb_sel_i	4	Input	Byte select signals
wb_we_i	1	Input	Write enable input
wb_stb_i	1	Input	Strobe signal/Core select input
wb_cyc_i	1	Input	Valid bus cycle input
wb_ack_o	1	Output	Bus cycle acknowledge output
wb_err_o	1	Output	Bus cycle error output
wb_int_o	1	Output	Interrupt signal output

- May use a typical at: https://opencores.org/projects/spi
- And https://github.com/xfguo/spi/tree/master/rtl/verilog
- And https://opencores.org/websvn/filedetails?repname=spi&path=%2Fspi%2Ftrunk%2Fdoc%2Fspi.pdf

DUT – SoC SPI

Reg Name	Access Address	Bus Width	Access Mode	Description
Rx0	0x00	32	R	Receive data register 0
Rx1	0x04	32	R	Receive data register 1
Rx2	0x08	32	R	Receive data register 2
Rx3	0x0C	32	R	Receive data register 3
Tx0	0x00	32	W/R	Transmit data register 0
Tx1	0x04	32	W/R	Transmit data register 1
Tx2	0x08	32	W/R	Transmit data register 2
Tx3	0x0C	32	W/R	Transmit data register 3
CTRL	0x10	32	W/R	Control and Status register
DIVIDER	0x14	32	W/R	Clock dividing register
SS	0x18	32	W/R	Slave chip select register

- May use a typical at: https://opencores.org/projects/spi
- And https://github.com/xfguo/spi/tree/master/rtl/verilog

32-bit Receive Data Reg

- Reset Value: 0x00000000

Bit #	31:0
Access	R
Name	Rx

32-bit Transmit Data Reg

- Reset Value: 0x00000000

Bit #	31:0
Access	R/W
Name	Tx

Configuration Note

- We use **Receive data register 0** that holds the value of data received from the last transfer.
- **CTRL register** holds the data word length field:
  - E.g., if use CTRL [9:3] (… 00 0000 1000) that is set to 0x10,
    - Then, bits Rx[15:0] (16 low bits) can hold 16-bit received data.
  - If CTRL[9:3] is set to 0x08, bit RxL[7:0] holds the received data

- We use **Transmit data register 0** that holds the value of data received from the last transfer.
- **CTRL register** holds the data word length field:
  - E.g., if use CTRL [9:3] (… 00 0000 1000) that is set to 0x10,
    - Then, bits Tx[15:0] (16 low bits) can hold 16-bit received data.
  - if CTRL[9:3] is set to 0x08, the bit Tx0[7:0] will be transmitted in next transfer

Control and Status register [CTRL]

- Reset Value: 0x00000000

Bit #	31:14	13	12	11	10	9	8	7	6:0
Access	R	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
Name	Reserved	ASS	IE	LSB	Tx_NEG	Rx_NEG	GO_BSY	Reserved	CHAR_LEN

Configuration Note (2)

- Automatic **SS** (ASS) bit
  - When the automatic **SS bit** (ASS) is set, the **o_ss** signal is generated automatically.
  - Data transfer is started by setting **CTRL**[GO_BSY],
    - the slave select (SS) signal which is selected in **SS** register
      - is asserted by the **SPI module**
      - and is de-asserted after the transfer is fnished.
  - If ASS bit is cleared, then
    - the slave select (SS) signals are asserted and de-asserted
    - by writing and clearing the bits in **SS** register.

Configuration Note (3)

IE

If this bit is set, the interrupt output is set active after a transfer is finished.

- The Interrupt signal is deasserted after a Read or Write to any register.

LSB

If this bit is set, the LSB is sent first on the line (bit TxL[0]), and the first bit received from the line will be put in the LSB position in the Rx register (bit RxL[0]).

- If this bit is cleared, the MSB is transmitted/received first (which bit in Tx/Rx register that is depends on the CHAR_LEN field in the CTRL register).

Configuration Note (4)

Tx_NEG

If this bit is set, the **mosi** signal is changed on the falling edge of a **sclk** clock signal, or otherwise the **mosi** signal is changed on the rising edge of **sclk**.

Rx_NEG

If this bit is set, the **miso** signal is latched on the falling edge of a **sclk** clock signal, or otherwise the miso signal is latched on the rising edge of **sclk**.

GO_BSY

Writing 1 to this bit starts the transfer.

- This bit remains set during the transfer and is automatically cleared after the transfer finished.
- Writing 0 to this bit has no effect.

Configuration Note (5)

NOTE:

- *All registers, including the CTRL register, should be set before writing 1 to the GO_BSY bit in the CTRL register.*
- *The configuration in the CTRL register must be changed with the GO_BSY bit cleared, i.e. two Writes to the CTRL register must be executed when changing the configuration and performing the next transfer, firstly with the GO_BSY bit cleared and secondly with GO_BSY bit set to start the transfer.*
- *When a transfer is in progress, writing to any register of the SPI Master core has no effect.*

Configuration Note (6)

- **CHAR_LEN**
  This field specifies how many bits are transmitted in one transfer.
- Up to 64 bits can be transmitted.
  CHAR_LEN = 0x01 … 1 bit
  CHAR_LEN = 0x02 … 2 bits
  …
  CHAR_LEN = 0x7f … 127 bits
  CHAR_LEN = 0x00 … 128 bits

Configuration Note (6bis)

0x09 – 9 bits

0x0A-10 bits

0x0B-11 bits

0x0C-12 bits

0x0D-13 bits

0x0E-14 bits

0x0F-15 bits

0x10-16 bits

0x11-17 bits

0x12-18 bits

0x13-19 bits

0x14-20 bits

0x15-21 bits

0x16-22 bits

0x17-23 bits

0x18-24 bits

0x19-25 bits

0x1A-26 bits

0x1B-27 bits

0x1C-28 bits

0x1D-29 bits

0x1E-30 bits

0x1F-31 bits

0x20-32 bits

Clk dividing register [DIVIDER]

- Reset Value: 0x0000ffff

Bit #	31:16	15:0
Access	R	R/W
Name	Reserved	DIVIDER

- The value in this field is the frequency divider of the system clock **i_wb_clk** to generate the serial clock on the output **o_sclk**.
- The desired frequency is obtained according to the following equation:

$$f_{sclk} = \frac{f_{wb\_clk}}{(DIVIDER+1)*2}$$

SS Reg

Bit #	31:8	7:0
Access	R	R/W

Configuration Note (7)

- **SS Register**
  - When **CTRL**[ASS] bit is cleared, writing 0x1 (0001) to any of the bit locations of this feld sets the proper o_ss line to an active state and writing 0x0 sets the line back to the inactive state.
  - When **CTRL** [ASS] bit is set, writing 1 to any bit location of this feld will select appropriate o_ss line to be automatically driven to an active state for the duration of the transfer, and will be driven to an inactive state for the rest of the time.

BFM

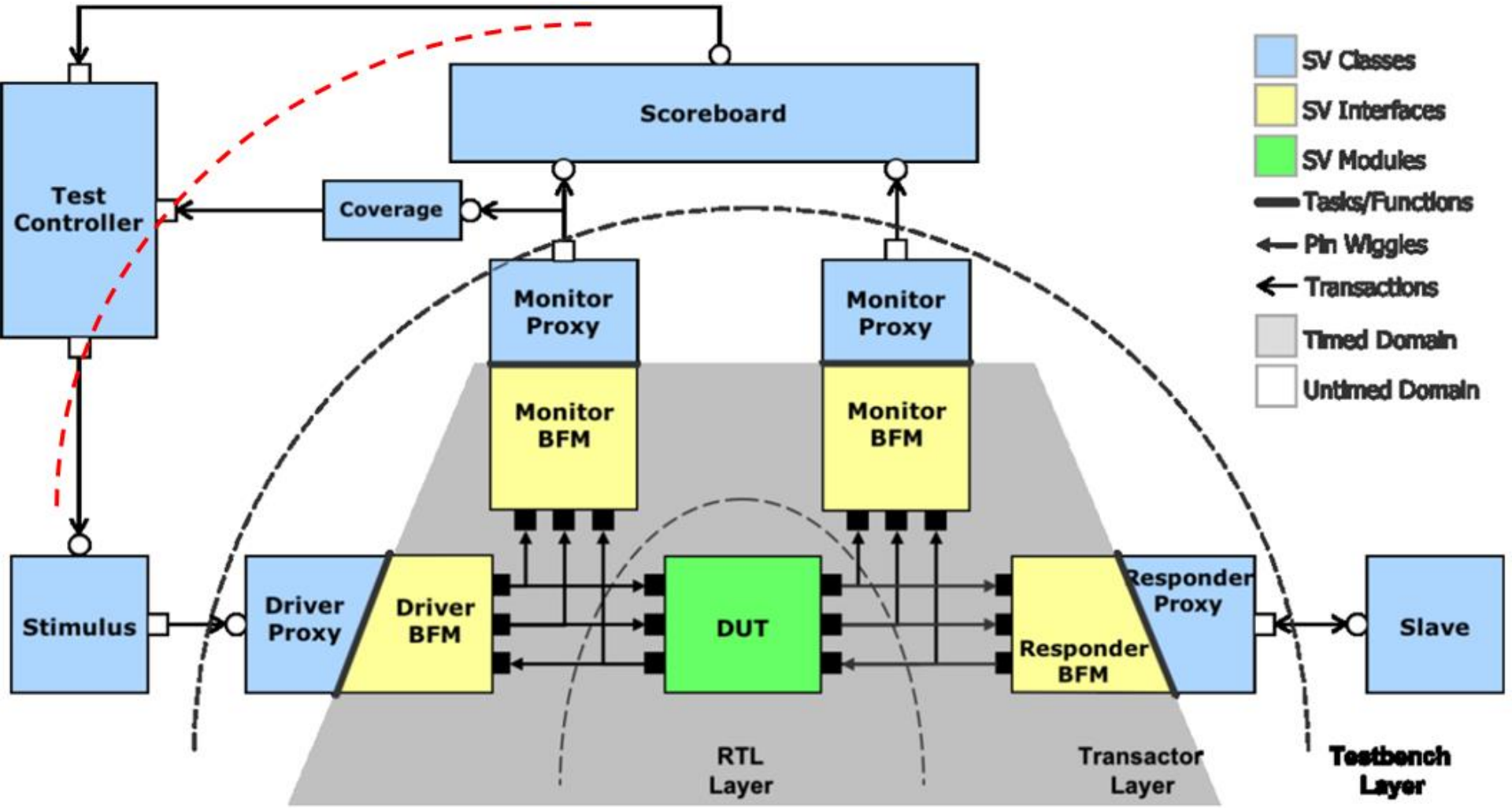
- Bus functional model (BFM) is a model of physical interfaces of the DUT.
- It ppresents all the bus level scenario that DUT can experience on the attached bus.
- BFMs
  1. on one side provide the logical interface for the high level **transactions** interface to test bench components
  2. and on another side connect to the physical interface of the DUT.

BFM (2)

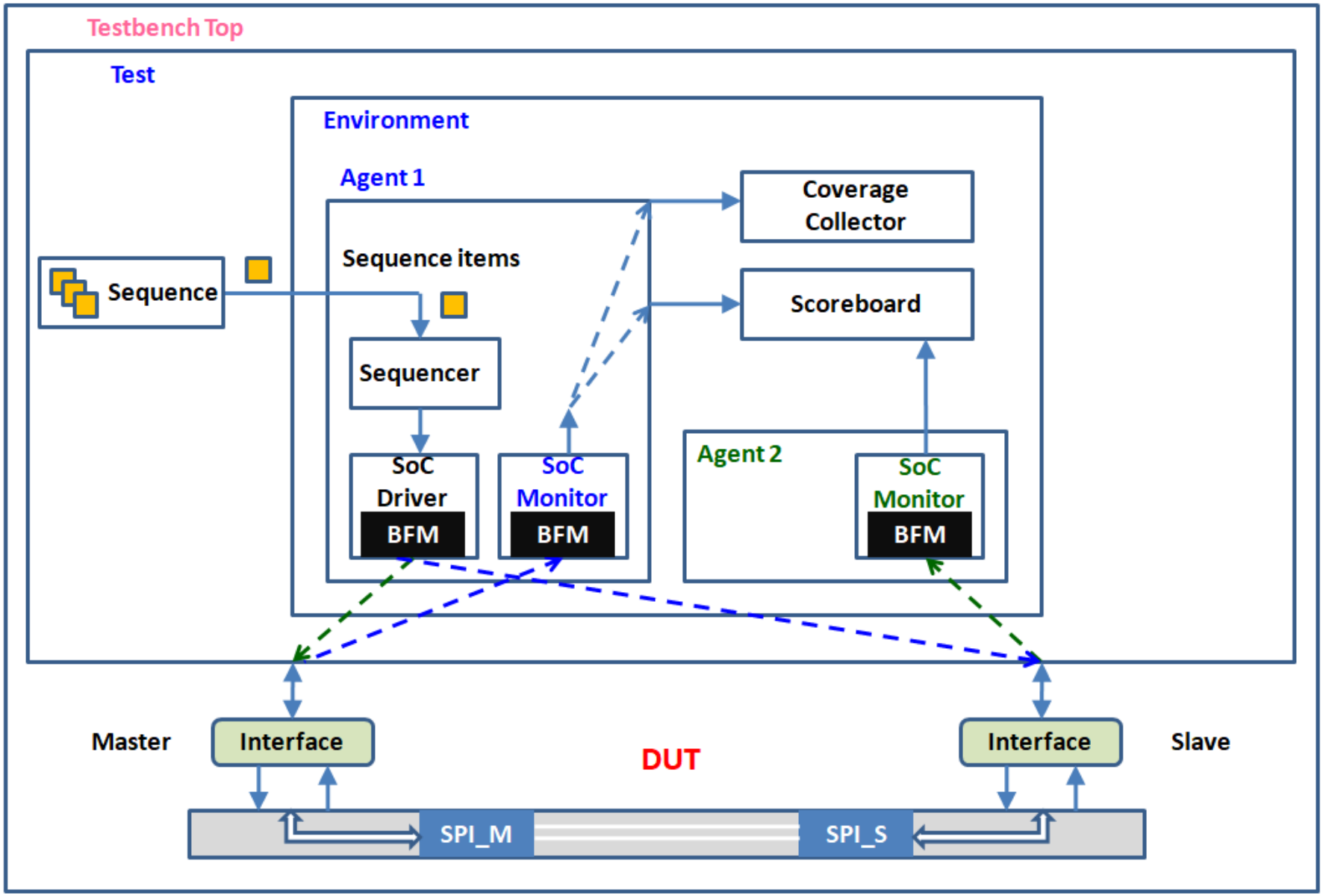
- Bus Functional Model (BFM) simulates **transactions/sequence items** of a bus, like READ and WRITE,
  - reducing the overhead of a testbench of taking care of the timing analysis for the same.

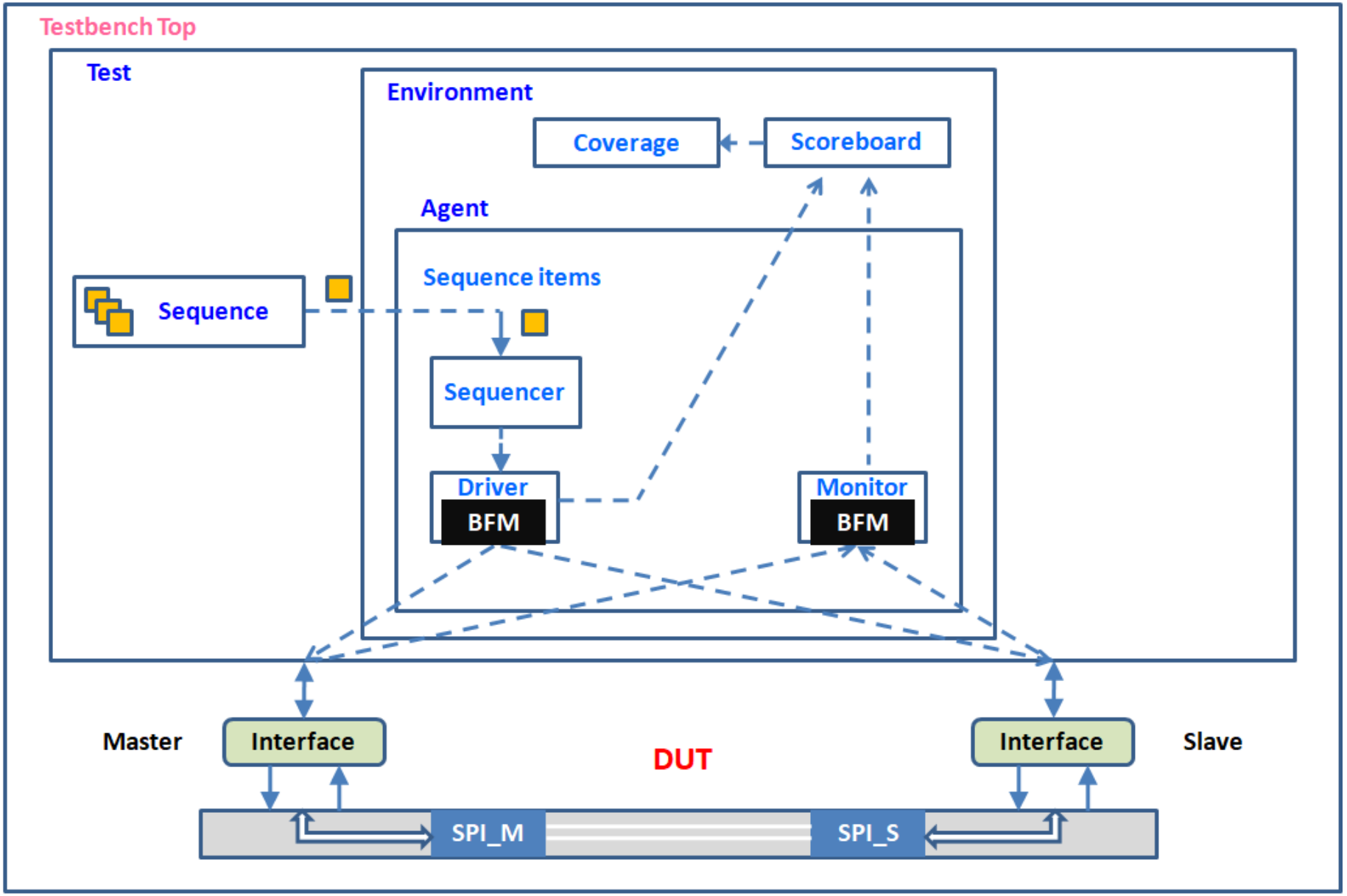
- There are a lot more interpretations of a BFM,
  - BFMs typically reduce the job of a testbench by making it more data focused.

UVM Testbench Architecture (Cookbook)

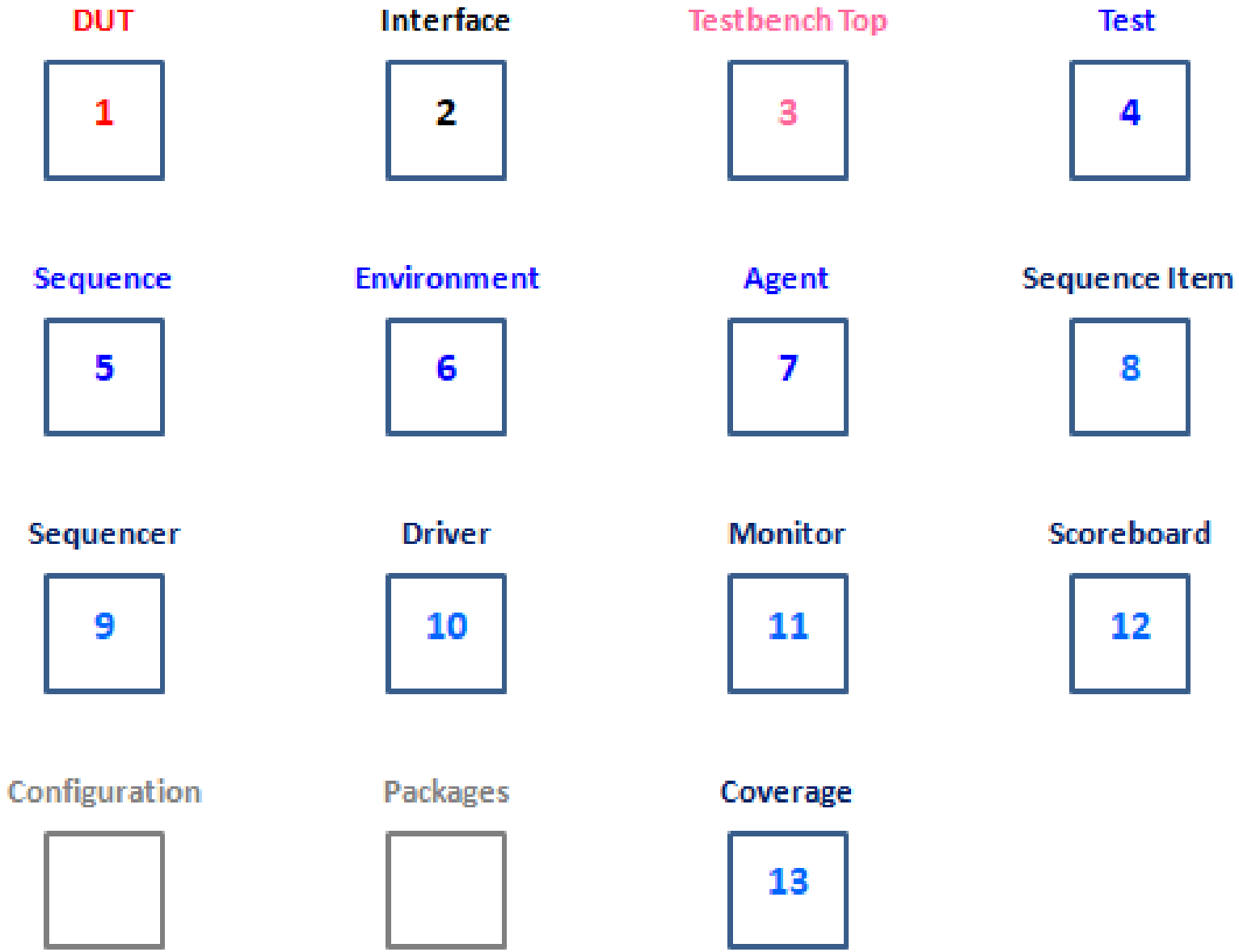


Bus functional model(BFM) is a model of physical interfaces of the DUT


Testbench Top
Test
Environment
Agent 1
Coverage Collector
Sequence items
Sequence
Scoreboard
Sequencer
SoC Driver
BFM
SoC Monitor
BFM
Agent 2
SoC Monitor
BFM
Master
Interface
DUT
Interface
Slave
SPI_M
SPI_S


Testbench Top
Test
Environment
Coverage
Scoreboard
Agent
Sequence items
Sequence
Sequencer
Driver
BFM
Monitor
BFM
Master
Interface
DUT
Interface
Slave
SPI_M
SPI_S

Implementation


DUT
1
Interface
2
Testbench Top
3
Test
4
Sequence
5
Environment
6
Agent
7
Sequence Item
8
Sequencer
9
Driver
10
Monitor
11
Scoreboard
12
Configuration
Packages
Coverage
13

```
interface spi_interface (input bit clk);
// NOTE: interface in uvm side/domain
// SoC signals:
logic [4:0] o_wb_addr;  // lower address bits
logic [31:0] i_wb_data; // data bus input
logic [31:0] o_wb_data; // data bus output
logic [3:0] o_wb_sel;   // select inputs
logic o_wb_we;          // write enable input
logic o_wb_stb;         // strobe / core select signal
logic o_wb_cyc;         // valid bus cycle input
logic i_wb_ack;         // bus cycle acknowledge output
logic i_wb_err;         // termination w/ error
logic i_wb_int;         // interrupt request signal output input
logic tick;             // transfer in complete known

clocking drive_cb @(posedge clk);
    input i_wb_data, i_wb_ack, i_wb_err, i_wb_int, tick;
    output o_wb_addr, o_wb_data, o_wb_sel, o_wb_we, o_wb_stb, o_wb_cyc;
endclocking

clocking monitor_cb @(posedge clk);
    input i_wb_data, i_wb_ack, i_wb_err, i_wb_int, tick;
    output o_wb_addr, o_wb_data, o_wb_sel, o_wb_we, o_wb_stb, o_wb_cyc;
endclocking

endinterface
```

```
`include "uvm_macros.svh"
`include "spi_pkg.sv"
`include "spi_interface.sv"

module tb_top;
import uvm_pkg::*;
import spi_pkg::*;

spi_interface master (clock); // master int declaration
spi_interface slave (clock);  // slave int declaration
```

```
// SPI master core:
spi spi_master (
    // UVM TB to DUT connection:
    .wb_clk_i(clock),
    .wb_rst_i(rstn),
    .wb_adr_i(master.o_wb_addr[4:0]),
    .wb_dat_i(master.o_wb_data),
    .wb_sel_i(master.o_wb_sel),
    .wb_we_i(master.o_wb_we),
    .wb_stb_i(master.o_wb_stb),
    .wb_cyc_i(master.o_wb_cyc),
    .wb_dat_o(master.i_wb_data),
    .wb_ack_o(master.i_wb_ack),
    .wb_err_o(master.i_wb_err),
    .wb_int_o(master.i_wb_int),
    // master to slave connection:
    .ss_pad_o(ss),
    .sclk_pad_o(sclk),
    .mosi_pad_o(mosi),
    .miso_pad_i(miso),
    .tip(master.tick)
    );
```

```
// SPI slave core:
spi_slave spi_slave (
    // UVM TB to DUT connection:
    .wb_clk_i(clock),
    .wb_rst_i(rstn),
    .wb_adr_i(slave.o_wb_addr[4:0]),
    .wb_dat_i(slave.o_wb_data),
    .wb_sel_i(slave.o_wb_sel),
    .wb_we_i(slave.o_wb_we),
    .wb_stb_i(slave.o_wb_stb),
    .wb_cyc_i(slave.o_wb_cyc),
    .wb_dat_o(slave.i_wb_data),
    .wb_ack_o(slave.i_wb_ack),
    .wb_err_o(slave.i_wb_err),
    .wb_int_o(slave.i_wb_int),
    // slave to master connection
    .ss_pad_i(ss),
    .sclk_pad_i(sclk),
    .mosi_pad_i(mosi),
    .miso_pad_o(miso)
);
```

```
initial begin
    generate_clock();
    reset_dut();
    reg_2ints_to_config_db();
    run_test();
end

task generate_clock();
    fork
        forever begin
            clock = 0;
            #(25);
            clock = 1;
            #(25);
        end
    join_none
endtask
```

May not needed ‘cause we can use test with reset phase

```
always #25 clock = ~clock;
```

```
task reset_dut();
    rstn <= 0;
    repeat (25) @(posedge clock);
    rstn <= 1;
    repeat (25) @(posedge clock);
    rstn = 0;
endtask

function void reg_2ints_to_config_db();
    // Registers two interfaces in the configuration data base so that
    // other blocks can use it retrived using get method
    uvm_config_db #( virtual spi_interface)::set(null, "*", "m_if", master);
    uvm_config_db #( virtual spi_interface)::set(null, "*", "s_if", slave);
endfunction

endmodule
```

```
class wb_bfm extends uvm_object;
`uvm_object_utils (wb_bfm)

function new (string name = "wb_bfm");
    super.new(name);
endfunction

static task wb_reset;
    input spi_vif vif;
    vif.o_wb_addr <= {5{1'bx}};
    vif.o_wb_data <= {32{1'bx}};
    vif.o_wb_cyc <= 1'b0;
    vif.o_wb_stb <= 1'bx;
    vif.o_wb_we <= 1'hx;
    vif.o_wb_sel <= {4{1'bx}};
endtask
```

```
static task wb_read;
    input spi_vif vif;
    input integer delay;
    input logic [4:0] a;
    output logic [31:0] d;

    begin
        // wait initial delay
        repeat (delay) @(vif.monitor_cb);
        // assert wishbone signals
        repeat (1) @(vif.monitor_cb);
        vif.monitor_cb.o_wb_addr <= a;
        vif.monitor_cb.o_wb_data <= {32{1'bx}};
        vif.monitor_cb.o_wb_cyc <= 1'b1;
        vif.monitor_cb.o_wb_stb <= 1'b1;
        vif.monitor_cb.o_wb_we <= 1'b0;
        vif.monitor_cb.o_wb_sel <= {4{1'b1}};
        @(vif.monitor_cb);
        // wait for acknowledge from slave
        wait (vif.monitor_cb.i_wb_ack == 1'b1)
        // negate wishbone signals
        repeat (1) @(vif.monitor_cb);
        vif.monitor_cb.o_wb_cyc <= 1'b0;
        vif.monitor_cb.o_wb_stb <= 1'bx;
        vif.monitor_cb.o_wb_addr <= {5{1'bx}};
        vif.monitor_cb.o_wb_data <= {32{1'bx}};
        vif.monitor_cb.o_wb_we <= 1'hx;
        vif.monitor_cb.o_wb_sel <= {4{1'bx}};
        d = vif.monitor_cb.i_wb_data;
    end
endtask
```

```
static task wb_write;
    input spi_vif vif;
    input integer delay;
    input logic [4:0] a;
    input logic [31:0] d;

    begin
        // wait initial delay
        repeat (delay) @(vif.drive_cb);
        // assert wishbone signal
        vif.drive_cb.o_wb_addr <= a;
        vif.drive_cb.o_wb_data <= d;
        vif.drive_cb.o_wb_cyc <= 1'b1;
        vif.drive_cb.o_wb_stb <= 1'b1;
        vif.drive_cb.o_wb_we <= 1'b1;
        vif.drive_cb.o_wb_sel <= {4{1'b1}};
        @(vif.drive_cb);
        // wait for acknowledge from slave
        //@(vif.drive_cb);
        wait (vif.drive_cb.i_wb_ack == 1'b1)
        // negate wishbone signals
        repeat (2) @(vif.drive_cb);
        vif.drive_cb.o_wb_cyc <= 1'b0;
        vif.drive_cb.o_wb_stb <= 1'bx;
        vif.drive_cb.o_wb_addr <= {5{1'bx}};
        vif.drive_cb.o_wb_data <= {32{1'bx}};
        vif.drive_cb.o_wb_we <= 1'hx;
        vif.drive_cb.o_wb_sel <= {4{1'bx}};
    end
endtask
endclass
```

External Note

- TASK:
  - Write a BFM class used for transactions at SoC AMBA bus.
  - Typical:
    - AXI Bus
      - **ab_bfm.svh**
      - or **axi_bfm.svh**
  - Specifications:
    - https://kolegite.com/EE_library/datasheets_and_manuals/FPGA/AMBA/IHI0022H_c_amba_axi_protocol_spec.pdf (2021)
  - Overview:
    - https://www.cis.upenn.edu/~cis5710/current/slides/13_axi.pdf

```
class spi_seq_item extends uvm_sequence_item;

// Registers configuration:
rand logic [31:0] master_ctrl_reg;
rand logic [31:0] slave_ctrl_reg;
rand logic [31:0] divider_reg;
rand logic [31:0] slave_select_reg;
rand logic [31:0] start_dut_reg;
// DUT output:
logic [31:0] out_master_data;
logic [31:0] out_slave_data;
// Expected data:
rand logic [31:0] exp_master_data;
rand logic [31:0] exp_slave_data;
// DUT input:
rand logic [31:0] in_master_data;
rand logic [31:0] in_slave_data;
```

```
`uvm_object_utils_begin(spi_seq_item)
`uvm_field_int(master_ctrl_reg, UVM_ALL_ON)
`uvm_field_int(slave_ctrl_reg, UVM_ALL_ON)
`uvm_field_int(divider_reg, UVM_ALL_ON)
`uvm_field_int(slave_select_reg, UVM_ALL_ON)
`uvm_field_int(start_dut_reg, UVM_ALL_ON)
`uvm_field_int(out_master_data, UVM_ALL_ON)
`uvm_field_int(out_slave_data, UVM_ALL_ON)
`uvm_field_int(exp_master_data, UVM_ALL_ON)
`uvm_field_int(exp_slave_data, UVM_ALL_ON)
`uvm_field_int(in_master_data, UVM_ALL_ON)
`uvm_field_int(in_slave_data, UVM_ALL_ON)
`uvm_object_utils_end

function new (string name="spi_seq_item");
    super.new (name);
endfunction

endclass
```

NOTE for v5

```
class spi_seq extends uvm_sequence#(spi_seq_item);
`uvm_object_utils(spi_seq)
spi_seq_item req;

function new (string name="spi_seq");
    super.new(name);
endfunction

virtual task body();
    req = spi_seq_item::type_id::create("req");
    start_item(req);
    //configure_dut_register();
    set_dut_data();
    finish_item(req);
endtask
```

```
virtual function void configure_dut_register();
    assert (req.randomize() with {req.master_ctrl_reg == 32'h00002208;
    //32'b0010 0010 0000 1000
    req.slave_ctrl_reg == 32'h00000200;//32'b0000001000000000 (CTRL[9]=1'b1, Rx_NEG)
    req.divider_reg == 32'h00000000; // DIVIDER = 0
    req.slave_select_reg == 32'h00000001;//32'b0000000000000001 (Slv needs the bit set)
    req.start_dut_reg == 32'h00000320;});//32'b0000 0011 0010 0000
    //0x20(char length)-32 bits; CTRL[9]=1,Rx_NEG; CTRL[8]=1,GO_BSY
endfunction

virtual function void set_dut_data();
    assert (req.randomize() with {req.divider_reg == 32'h00000000;
    req.master_ctrl_reg == 32'h00002208;
    req.slave_ctrl_reg == 32'h00000200;
    req.slave_select_reg == 32'h00000001;
    req.start_dut_reg == 32'h00000320;
    req.exp_master_data == req.in_slave_data;   //
    req.exp_slave_data == req.in_master_data;});//
endfunction

endclass
```

- uvm_analysis_port #(seq_item)

```
class spi_driver extends uvm_driver #(spi_seq_item);
 `uvm_component_utils(spi_driver)
 spi_vif m_vif, s_vif;
 spi_seq_item packet;
 uvm_analysis_port #(spi_seq_item) dut_in_pkt;

function new (string name="spi_driver", uvm_component parent);
     super.new(name, parent);
     dut_in_pkt = new ("dut_in_pkt", this);
endfunction

function void build_phase (uvm_phase phase);
     super.build_phase(phase);
     `uvm_info (get_full_name(), "Build phase called in spi_driver", UVM_LOW)
     if (!uvm_config_db #(virtual spi_interface)::get(this, "", "m_if", m_vif))
     `uvm_fatal("NO_VIF", {"virtual interface must be set for:", get_full_name(), ".m_vif"})
     if (!uvm_config_db #(virtual spi_interface)::get(this, "", "s_if", s_vif))
     `uvm_fatal("NO_VIF", {"virtual interface must be set for:", get_full_name(), ".s_vif"})
endfunction
```

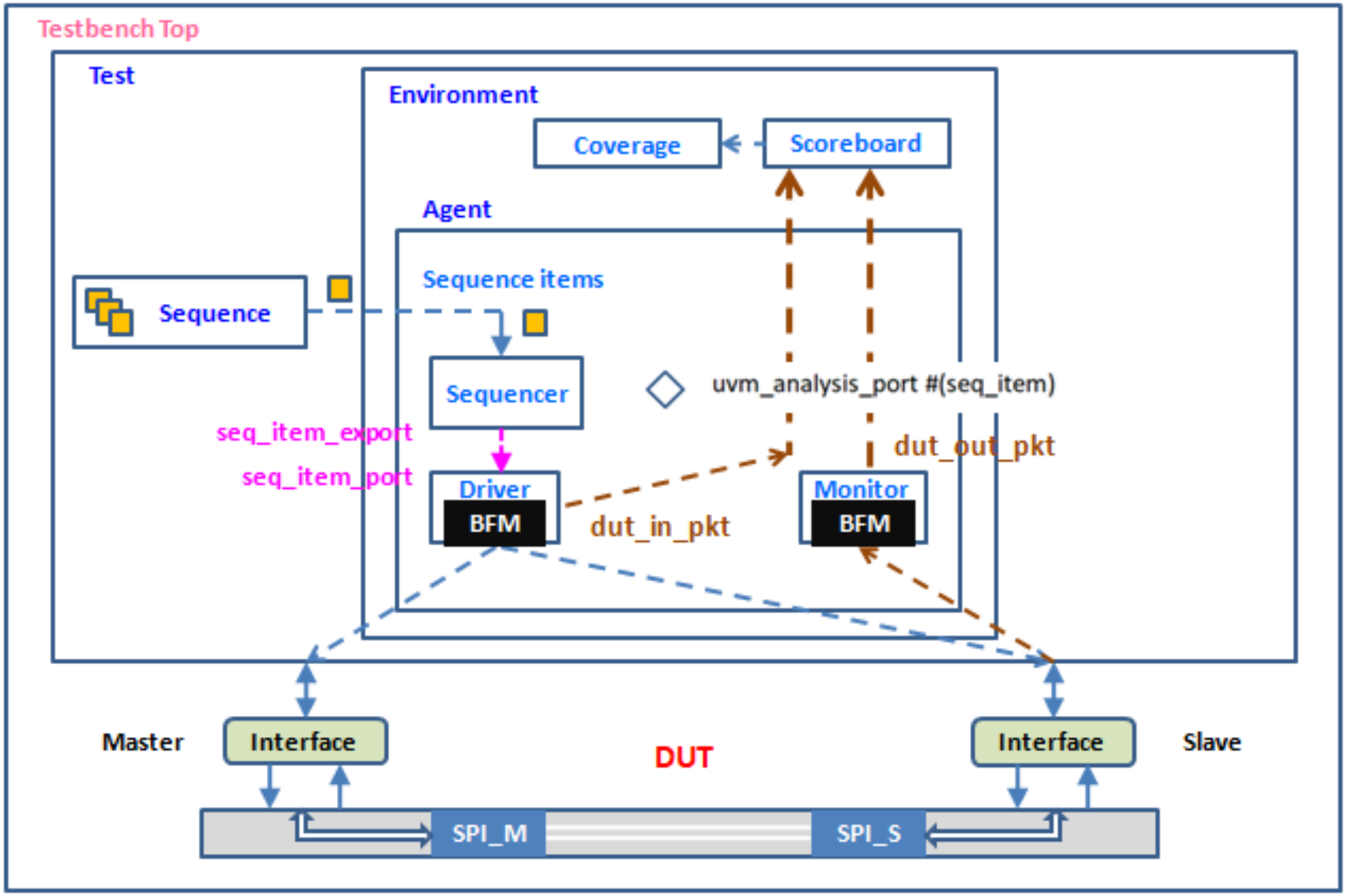
```
task run_phase (uvm_phase phase);
    packet = spi_seq_item::type_id::create("packet");
    wb_bfm::wb_reset(m_vif);
    wb_bfm::wb_reset(s_vif);
    fork
        forever begin
        seq_item_port.get_next_item(req);
        drive_transfer(req);
        $cast(packet, req.clone());
        packet = req;
        dut_in_pkt.write(packet);
        seq_item_port.item_done();
        wait(m_vif.monitor_cb.tick == 1'b0);
        end
    join_none
endtask

task drive_transfer (spi_seq_item seq);
wb_bfm::wb_write(m_vif, 0, SPI_DIVIDE, seq.divider_reg);  // set divider register
wb_bfm::wb_write(m_vif, 0, SPI_SS, seq.slave_select_reg); // set ss 0
wb_bfm::wb_write(m_vif, 0, SPI_TX_0, seq.in_master_data); // set master data register
wb_bfm::wb_write(m_vif, 0, SPI_CTRL, seq.master_ctrl_reg);// set master ctrl register
wb_bfm::wb_write(s_vif, 0, SPI_CTRL, seq.slave_ctrl_reg); // set slave ctrl register
wb_bfm::wb_write(s_vif, 0, SPI_TX_0, seq.in_slave_data);  // set slave data register
wb_bfm::wb_write(m_vif, 0, SPI_CTRL, seq.start_dut_reg);  // start data transfer
endtask

endclass
```

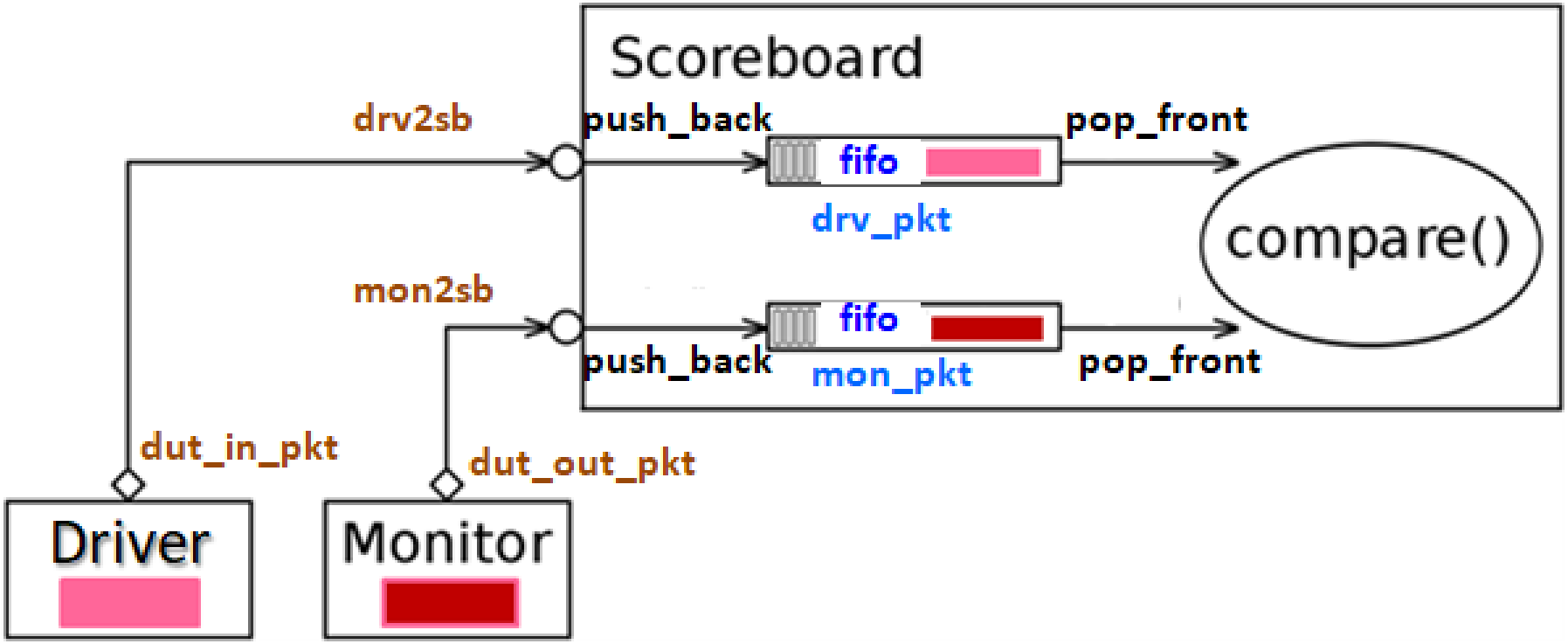
```
class spi_monitor extends uvm_monitor;
 `uvm_component_utils(spi_monitor)
 spi_vif m_vif, s_vif;
 spi_seq_item packet;
 uvm_analysis_port#(spi_seq_item) dut_out_pkt;

function new (string name="spi_monitor", uvm_component parent);
     super.new(name, parent);
     dut_out_pkt = new("dut_out_pkt", this);
endfunction

function void build_phase(uvm_phase phase);
     super.build_phase(phase);
     `uvm_info(get_full_name(), "Build phase called in spi_monitor", UVM_LOW)
     if (!uvm_config_db#(virtual spi_interface)::get(this, "", "m_if", m_vif))
     `uvm_fatal("NO_VIF", {"virtual interface must be set for:", get_full_name(), ".m_vif"})
     if (!uvm_config_db#(virtual spi_interface)::get(this, "", "s_if", s_vif))
     `uvm_fatal("NO_VIF", {"virtual interface must be set for:", get_full_name(), ".s_vif"})
endfunction

task run_phase (uvm_phase phase);
     packet = spi_seq_item::type_id::create("packet");
     wait (m_vif.monitor_cb.tick ==1'b1) // wait_to_start
     forever begin
          wait (m_vif.monitor_cb.tick ==1'b0) // wait_to_complete
          wb_bfm::wb_read(m_vif, 1, SPI_RX_0, packet.out_master_data);
          wb_bfm::wb_read(s_vif, 1, SPI_RX_0, packet.out_slave_data);
          dut_out_pkt.write(packet);
          wait (m_vif.monitor_cb.tick ==1'b1); // wait_to_start
     end
endtask
endclass
```

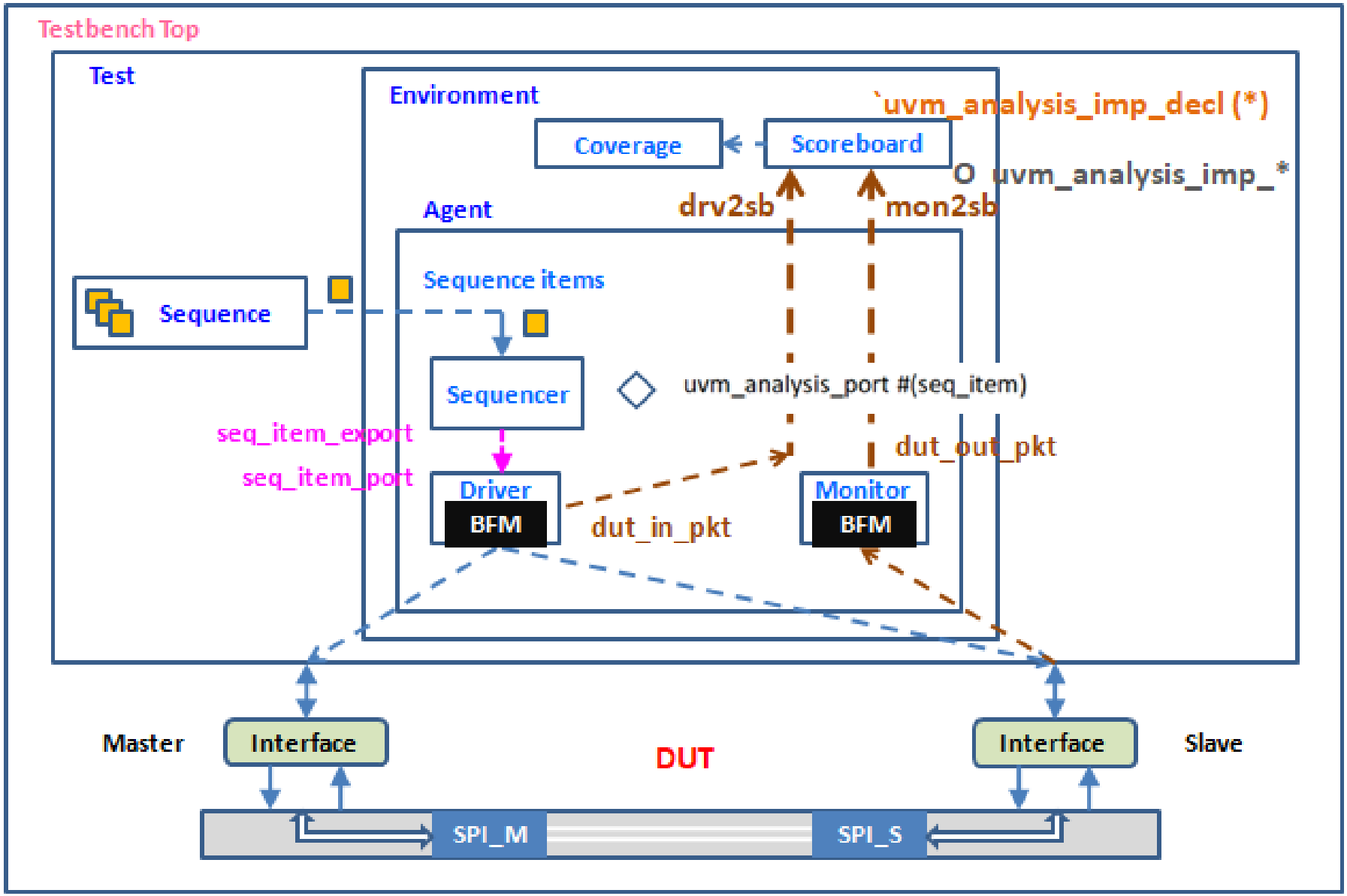
◇ uvm_analysis_port #(seq_item)

```
class spi_coverage extends uvm_component;
 `uvm_component_utils (spi_coverage)
 spi_seq_item c_pkt;

covergroup cov_inst;
 cp_dut_mosi: coverpoint c_pkt.exp_master_data
     {
     bins byte0_7 = {[0:255]};
     bins byte8_15 = {[256:65535]};
     bins byte16_23 = {[65536:16777215]};
     bins byte24_31 = {[16777216:$]};
     }
endgroup

function new (string name="spi_coverage", uvm_component parent=null);
     super.new(name, parent);
     cov_inst = new();
endfunction

function void perform_coverage(spi_seq_item pkt);
     this.c_pkt = pkt;
     cov_inst.sample();
endfunction

endclass
```

```
extends uvm_subscriber #(spi_seq_item);
```


Testbench Top
Test
Environment
Coverage
Scoreboard
Agent
Sequence items
Sequence
Sequencer
uvm_analysis_port #(seq_item)
seq_item_export
seq_item_port
dut_out_pkt
Driver
BFM
dut_in_pkt
Monitor
BFM
Master
Interface
DUT
Interface
Slave
SPI_M
SPI_S


Scoreboard
drv2sb
push_back
pop_front
fifo
drv_pkt
compare()
mon2sb
fifo
push_back
mon_pkt
pop_front
dut_in_pkt
dut_out_pkt
Driver
Monitor


Testbench Top
Test
Environment
`uvm_analysis_imp_decl (*)
Coverage
Scoreboard
uvm_analysis_imp_*
drv2sb
mon2sb
Agent
Sequence items
Sequence
Sequencer
uvm_analysis_port #(seq_item)
seq_item_export
seq_item_port
dut_out_pkt
Driver
BFM
dut_in_pkt
Monitor
BFM
Master
Interface
DUT
Interface
Slave
SPI_M
SPI_S

```
class spi_scoreboard extends uvm_scoreboard;
`uvm_component_utils (spi_scoreboard)

uvm_tlm_analysis_fifo #(spi_seq_item) drv2sb; //
uvm_tlm_analysis_fifo #(spi_seq_item) mon2sb; //
spi_seq_item drv_pkt[$]; // unbounded
spi_seq_item mon_pkt[$]; // unbounded

spi_seq_item ip_pkt;
spi_seq_item op_pkt;


static string report_tag;
spi_coverage spi_covg;
int pass = 0;
int fail = 0;

function new (string name="spi_scoreboard", uvm_component parent);
    super.new(name, parent);
    report_tag = $sformatf("%0s", name);
    drv2sb = new ("drv2sb", this);
    mon2sb = new ("mon2sb", this);
endfunction

function void build_phase (uvm_phase phase);
    super.build_phase(phase);
    `uvm_info(get_full_name(), "Build phase called in spi_scoreboard", UVM_LOW)
    spi_covg = spi_coverage::type_id::create("spi_covg", this);
endfunction
```

```
function void connect_phase (uvm_phase phase);
    super.connect_phase(phase);
    `uvm_info(get_full_name(), "Connect phase called in spi_scoreboard", UVM_LOW)
endfunction

task run_phase (uvm_phase phase);
    forever begin
        perform_check(ip_pkt, op_pkt);
        perform_coverage(ip_pkt);
        uvm_test_done.drop_objection(this);
    end
endtask

function void perform_coverage (spi_seq_item pkt);
    spi_covg.perform_coverage(pkt);
endfunction

function void perform_check (spi_seq_item ip_pkt, spi_seq_item op_pkt);
    forever begin
    fork
        drv2sb.get(ip_pkt);
        `uvm_info("SB","TRANSACTIONS FROM DRIVER",UVM_NONE);
        mon2sb.get(op_pkt);
        `uvm_info("SB","TRANSACTIONS FROM MONITOR",UVM_NONE);
    join
    if (ip_pkt.exp_master_data == op_pkt.out_master_data && ip_pkt.exp_slave_data == op_pkt.out_slave_data)
        begin
        `uvm_info(get_full_name(), "Master PASSED", UVM_MEDIUM)
        `uvm_info(get_full_name(), "Slave PASSED", UVM_MEDIUM)
        pass++;
        end
    else
        begin
        `uvm_info(get_full_name(), $sformatf("Slave FAILED: exp data=%0h and out data=%0h", ip_pkt.exp_slave_data, op_pkt.out_slave_data), UVM_MEDIUM)
        `uvm_info(get_full_name(), $sformatf("Master FAILED: exp data=%0h and out master data=%0h", ip_pkt.exp_master_data, op_pkt.out_master_data), UVM_MEDIUM)
        fail++;
        end
    end
endfunction
```

Note

```
function void extract_phase(uvm_phase phase);
endfunction

function void report_phase(uvm_phase phase);
    if (fail==0) begin
        $display("---------------------- 32 bit --MSB First --TX: posedge --RX: negedge ------------------------------");
        $display("----------------------------------------- TEST PASSED -----------------------------------------");
        $display("***********************************");
        uvm_report_info("Scoreboard Report", $sformatf("Transactions PASS = %0d FAIL = %0d", pass, fail), UVM_MEDIUM);
        $display("***********************************************");
        $display("-----------------------------------");
        $display("-----------------------------------");
    end
    else begin
        $display("----------------------- 32 bit --MSB First --TX: posedge--RX: negedge  ------------------------------");
        $display("----------------------------------------- TEST FAILED -----------------------------------------");
        $display("***********************************");
        uvm_report_info("Scoreboard Report", $sformatf("Trasactions PASS = %0d FAIL = %0d", pass, fail), UVM_MEDIUM);
        $display("***********************************");
        $display("-----------------------------------");
        $display("-----------------------------------");
    end
endfunction

endclass
```

```
class spi_agent extends uvm_agent;
`uvm_component_utils(spi_agent)
spi_sequencer sequencer;
spi_monitor monitor;
spi_driver driver;
spi_vif m_vif, s_vif;

function new (string name="spi_agent", uvm_component parent);
    super.new(name, parent);
endfunction

function void build_phase (uvm_phase phase);
    super.build_phase(phase);
    `uvm_info(get_full_name(), "Build phase called in spi_agent", UVM_LOW)
    if (!uvm_config_db#(virtual spi_interface)::get(this, "", "m_if", m_vif))
    `uvm_fatal("NO_VIF", {"virtual interface must be set for:", get_full_name(), ".m_vif"})
    if (!uvm_config_db#(virtual spi_interface)::get(this, "", "s_if", s_vif))
    `uvm_fatal("NO_VIF", {"virtual interface must be set for:", get_full_name(), ".s_vif"})
    sequencer = spi_sequencer::type_id::create("sequencer", this);
    driver = spi_driver::type_id::create("driver", this);
    driver.m_vif = m_vif;
    driver.s_vif = s_vif;
    monitor = spi_monitor::type_id::create("monitor", this);
    monitor.m_vif = m_vif;
    monitor.s_vif = s_vif;
endfunction

function void connect_phase (uvm_phase phase);
    super.connect_phase(phase);
    `uvm_info(get_full_name(), "Connect phase called in spi_agent", UVM_LOW)
    driver.seq_item_port.connect(sequencer.seq_item_export);
endfunction

endclass
```

```
class spi_env extends uvm_env;

 `uvm_component_utils (spi_env)
 spi_agent agent;
 spi_scoreboard scoreboard;

 function new (string name="spi_env", uvm_component parent);
     super.new(name, parent);
 endfunction

 function void build_phase (uvm_phase phase);
     super.build_phase(phase);
     `uvm_info(get_full_name(), "Build phase called in spi_environment", UVM_LOW)
     // Build agent and scoreboard components:
     agent = spi_agent::type_id::create("agent", this);
     scoreboard = spi_scoreboard::type_id::create("scoreboard", this);
 endfunction

 function void connect_phase (uvm_phase phase);
     super.connect_phase(phase);
     `uvm_info(get_full_name(), "Connect phase called in spi_environment", UVM_LOW)
     // Connect the analysis ports at driver and monitor with scoreboard:
     agent.driver.dut_in_pkt.connect(scoreboard.drv2sb.analysis_export);
     agent.monitor.dut_out_pkt.connect(scoreboard.mon2sb.analysis_export);
 endfunction

 endclass
```

```
class spi_test extends uvm_test;
`uvm_component_utils(spi_test)
spi_env env;
spi_seq t_seq;

function new (string name="spi_test", uvm_component parent);
    super.new(name, parent);
endfunction

function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info(get_full_name(), "Build phase called in spi_test", UVM_LOW)
    // Build environment component:
    env = spi_env::type_id::create("env", this);
endfunction

function void connect_phase (uvm_phase phase);
    super.connect_phase (phase);
    `uvm_info (get_full_name(), "Connect phase called in spi_test", UVM_LOW)
endfunction
```

```
task reset_phase (uvm_phase phase);
    phase.raise_objection(this);
    rstn <= 0;
    repeat (25) @(posedge clock);
    rstn <= 1;
    repeat (25) @(posedge clock);
    rstn = 0;
    phase.drop_objection(this);
endtask

virtual task main_phase (uvm_phase phase);
    `uvm_info (get_full_name(), "in main phase", UVM_LOW)
    phase.raise_objection(this);
    t_seq = spi_seq::type_id::create("t_seq");
    repeat(100)
    t_seq.start(env.agent.sequencer);
    phase.drop_objection(this);
endtask

endclass
```

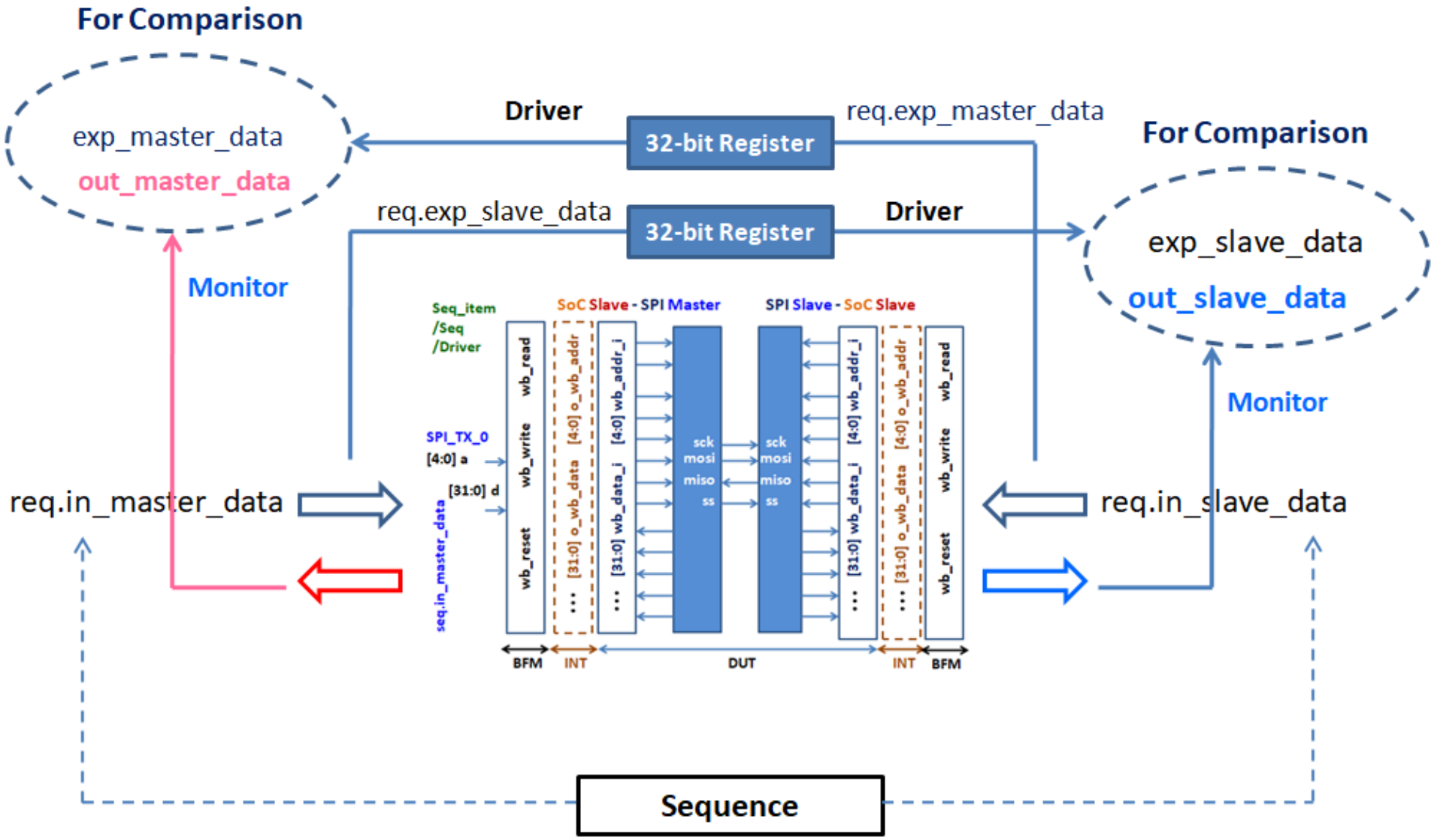
```
package spi_pkg;

import uvm_pkg::*;
`include "uvm_macros.svh"

`include "tb_defines.svh"
`include "spi_seq_item.svh"
`include "wb_bfm.svh"
`include "spi_driver.svh"
`include "spi_monitor.svh"
`include "spi_sequencer.svh"
`include "spi_agent.svh"
`include "spi_coverage.svh"
`include "spi_scoreboard.svh"
`include "spi_seq.svh"
`include "spi_env.svh"
`include "spi_test.svh"

endpackage
```

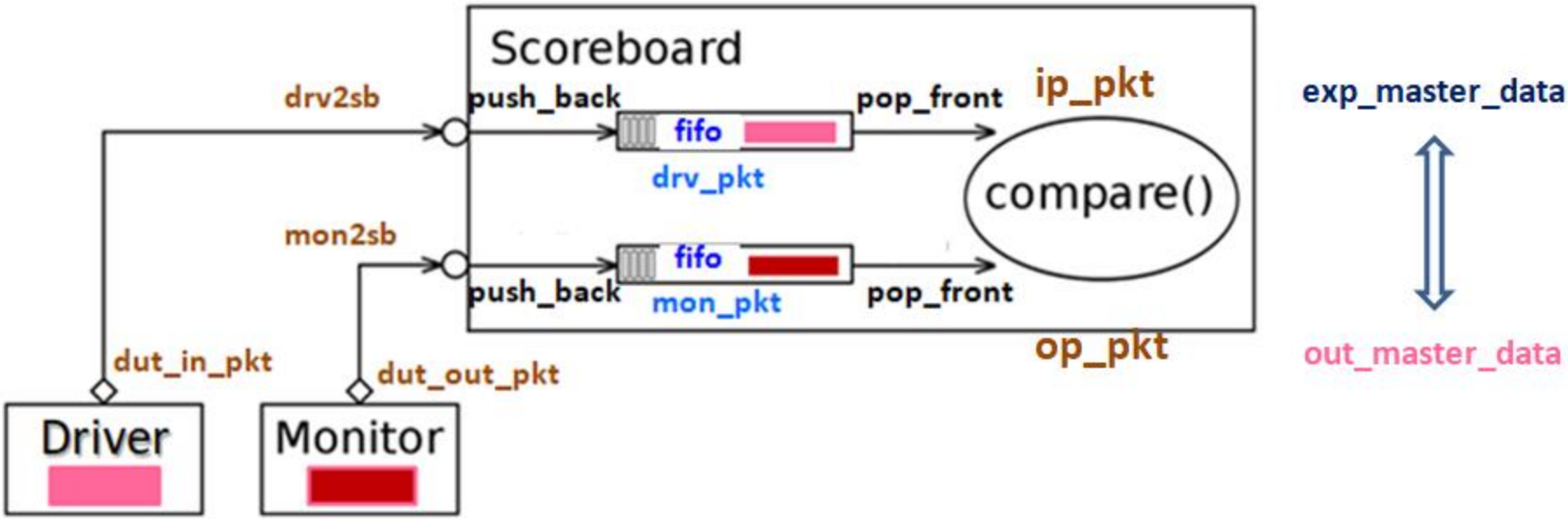
```
1   // Register address values:
2
3   parameter SPI_RX_0 = 5'h0;
4   parameter SPI_RX_1 = 5'h4;
5   parameter SPI_RX_2 = 5'h8;
6   parameter SPI_RX_3 = 5'hc;
7
8   parameter SPI_TX_0 = 5'h0;     // 0000 0000
9   parameter SPI_TX_1 = 5'h4;     // 0000 0100
10  parameter SPI_TX_2 = 5'h8;     // 0000 1000
11  parameter SPI_TX_3 = 5'hc;     // 0000 1100
12  parameter SPI_CTRL = 5'h10;    // 0001 0000
13  parameter SPI_DIVIDE = 5'h14; // 0001 0100
14  parameter SPI_SS = 5'h18;      // 0001 1000
15
16  logic clock, rstn;
17  wire logic sclk, mosi, miso;
18  wire logic [31:0]ss;
19  logic tip;
20
21  // Virtual interface:
22  typedef virtual spi_interface spi_vif;
```

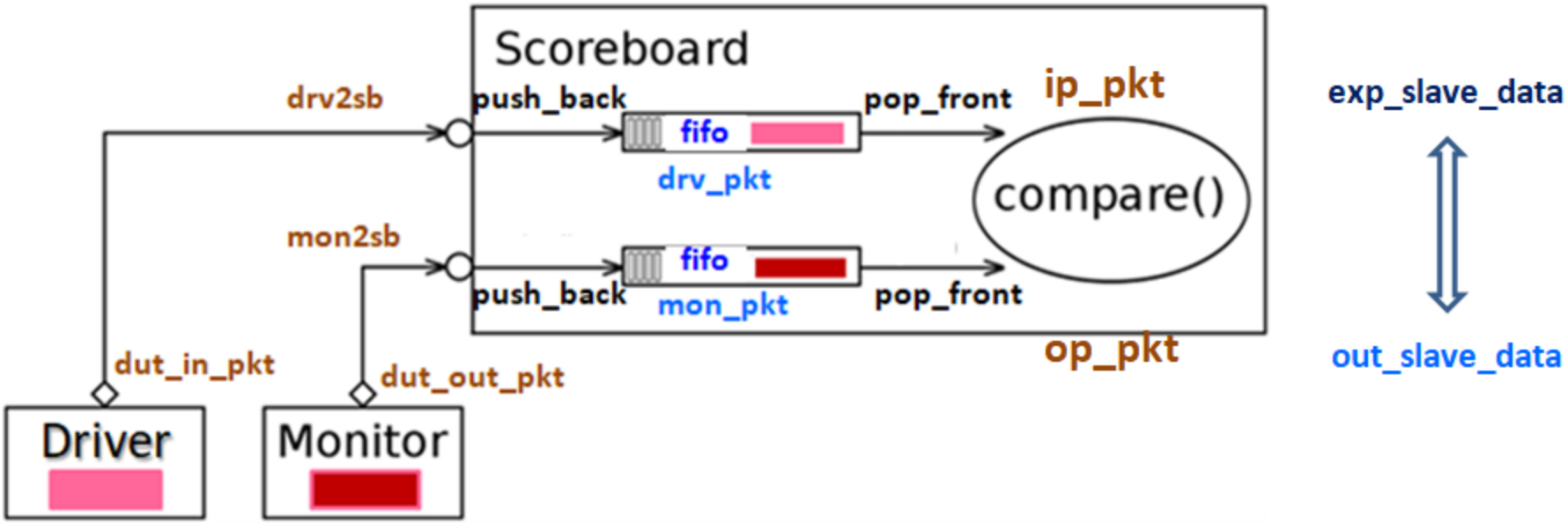
Backup Slide

```
class spi_sequencer extends uvm_sequencer #(spi_seq_item);
 `uvm_component_utils(spi_sequencer)

function new (string name="spi_sequencer", uvm_component parent);
    super.new(name, parent);
endfunction

endclass
```

```
# Setting directories: ------------------------------------------------
set RTL ./dut

# Library and mapping: ------------------------------------------------
vlib work
vmap work work

# Compilation: --------------------------------------------------------
vlog $RTL/*.v +acc +cover=bcefst
vlog spi_interface.sv -timescale 1ns/10ps
vlog spi_pkg.sv
vlog tb_top.sv +acc +cover=bcefst

# Simulation: ---------------------------------------------------------
vsim -coverage -novopt tb_top +UVM_TESTNAME=spi_test

# Coverage report: ----------------------------------------------------
# Coverage save ucdb file:
coverage save -onexit -assert -directive -cvg -codeAll spi_test.ucdb

# Coverage reports with html and text files:
vcover report -html spi_test.ucdb -htmldir covhtmlreport
vcover report -file cov_report.txt spi_test.ucdb

add wave -r sim:/tb_top/*
run -all
```


For Comparison
exp_master_data
out_master_data
Driver
32-bit Register
req.exp_master_data
For Comparison
req.exp_slave_data
32-bit Register
Driver
exp_slave_data
out_slave_data
Monitor
Monitor
Seq_item
/Seq
/Driver
SoC Slave - SPI Master
SPI Slave - SoC Slave
SPI_TX_0
[4:0] a
[31:0] d
seq.in_master_data
wb_read
wb_write
wb_reset
[4:0] o_wb_addr
[31:0] o_wb_data
[4:0] wb_addr_i
[31:0] wb_data_i
sck
mosi
miso
ss
BFM
INT
DUT
INT
BFM
req.in_master_data
req.in_slave_data
Sequence


Scoreboard
drv2sb
push_back
fifo
drv_pkt
pop_front
ip_pkt
compare()
mon2sb
fifo
push_back
mon_pkt
pop_front
op_pkt
dut_in_pkt
dut_out_pkt
Driver
Monitor
exp_master_data
out_master_data


Scoreboard
drv2sb
push_back
fifo
drv_pkt
pop_front
ip_pkt
compare()
mon2sb
fifo
push_back
mon_pkt
pop_front
op_pkt
dut_in_pkt
dut_out_pkt
Driver
Monitor
exp_slave_data
out_slave_data

Thank You