Chap 4 Connecting the Testbench and Design

- Interfaces
- Clocking blocks
- Program blocks
- The end of simulation
- Top level scope
- Assertions
4 Connecting the Testbench and Design

• Testbench wraps around the Design Under Test (DUT)
• The testbench mimics the environment around the DUT

- The testbench is separate from the design
Connecting all the blocks of a design/testbench is tedious/error prone

module top;
    stim stim(...I/O...);
    Digital DUT(...I/O...);
endmodule

module stim(... I/O...);
endmodule

module Digital(... I/O...);
    heater heater(... I/O...);
endmodule

module heater(... I/O...);
    heater_sm heater_sm(... I/O...);
endmodule

module heater_sm(...I/O...)
endmodule
4.2 The Interface Construct

- An intelligent bundle of signals. Contains:
  - connectivity
  - synchronization
  - functionality
- Can be used for testbench as well as RTL connections

```vhdl
interface arb_if(input bit clk);
  logic [1:0] grant, request;
  bit rst;
endinterface
```
4.2.1 Using an Interface to Simplify Conn.

module arb(
    output logic [1:0] grant,
    input logic [1:0] request,
    input logic rst,
    input logic clk);

    always @(posedge clk or
              posedge rst) begin
        if (rst)
            grant <= 2′b00;
        else
            ....
    end
endmodule

interface arb_if(input bit clk);
    logic [1:0] grant, request;
    bit rst;
endinterface

module arb (arb_if arb_bus);
    always @(posedge arb_bus.clk or
              posedge arb_bus.rst) begin
        if (arb_bus.rst)
            arb_bus.grant <= 2′b00;
        else
            ....
    end
endmodule

module top;
    bit clk;
    always #5 clk = ~clk;
    arb_if arb_bus (clk);
    arb a1(arb_bus);
    test t2(arb_bus);
endmodule
4.2.2 Connecting Interfaces and Ports

If the ports of a legacy design cannot be changed to use an interface...

```verilog
interface arb_if(input bit clk);
    logic [1:0] grant, request;
    bit rst;
endinterface

module top;
    bit clk;
    always #5 clk = ~clk;
    arb_if arb_bus(clk);
    arb a1(
        .grant(arb_bus.grant),
        .request(arb_bus.request),
        .rst(arb_bus.rst),
        .clk(arb_bus.clk)
    );
    test t2(arb_bus);
endmodule
```
4.2.3 Group Signals in I/F using modport

• Default direction of signals in interface is bi-directional
• Probably not what you want for RTL or testbench.
• Specify direction using modport

```vhdl
interface arb_if(input bit clk);
  logic [1:0] grant, request;
  bit rst;

  modport DUT (input request, rst, clk, output grant);
endinterface

module arb(arb_if.DUT arb_bus);
  arb a1 (arb_bus.DUT);
endmodule
```

Usage:

```vhdl
module arb(arb_if.DUT arb_bus);

or

arb a1 (arb_bus.DUT);
```
4.2.3 Group Signals in I/F using modport

• Another example:

```verilog
interface ifx;
    logic clk, data_in, data_out csel;
    modport MASTER (input data_in, output clk, data_out, csel);
    modport SLAVE  (output data_in, input clk, data_out, csel);
endinterface

module m(ifx.MASTER mport);
...
@ (posedge ifx.clk);
...
endmodule

module s(ifx.SLAVE sport);
...
endmodule

module top
    ifx ifx1;
    m m1 (mport(ifx1));
    s s1 (sport(ifx1));
...
endmodule
```

Usage (in the module header):

Note: modport name not required
Mixing interface and ordinary ports

Typically only common busses will be encapsulated in an interface

```
`default_nettype none
`include "arb_if.v"

module test;
    bit clk;
    bit [7:0] data_in, data_out;
    always #5 clk = ~clk;
    arb_if arb_bus(clk);
    arb arb (.data_in(data_in), arb_bus.DUT,
             .data_out(data_out));
endmodule

module arb(input [7:0] data_in, arb_if arb_bus, 
           output [7:0] data_out);
```
Functions in an Interface

Why?
Protocol checkers!

```verilog
interface mem_bus_if(input bit clk, input bit reset);
    reg [7:0] data_in;
    reg [3:0] addr;
    always @(posedge clk) check_addr(addr);

    function check_addr(input reg [3:0] addr);
        if (addr == 4'hf) $display("Invalid Address");
    endfunction

endinterface
```
RTL functions in Interface

• Why?
  – Provides common functions.
  – Is only synthesized if called by module

```verilog
function automatic reg parity(input reg[7:0] data);
    return (^data);
endfunction
```

• Need to modify modport

```verilog
modport DUT (input clk, reset, data_in, addr,
             output data_out, import parity);
```

• Call in RTL

```verilog
assign mem_bus.data_out[8] = mem_bus.parity(.....);
```
Synthesis

• Interface synthesizes exactly as DUT without an interface.
• No warning on print statement in check_addr of interface.
• Need automatic keyword for functions. This makes functions use automatic storage
• An interface function only gets synthesized if called by module.
4.2.6 Interface trade-offs

• Advantages:
  • Design reuse – communication with an oft-used protocol
  • Adding/changing/deleting a signal is easy
  • Enforcing naming conventions, size, direction

• Disadvantages
  • Little savings for point to point connections
  • Must use interface name in addition to the signal name
  • Difficult to connect 2 different interfaces.
  • All variables in interface are accessible
    • No privacy/control
Interface Exercise 1

1. Design an interface and testbench for the following ARM High Speed Bus (AHB).
2. You are provided an AHB master as verification IP.
3. You are testing an AHB slave design.
4. Your interface will display an error if the transaction type is not IDLE or NONSEQ on the negative edge of HCLK.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Width</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCLK</td>
<td>1</td>
<td>output</td>
<td>Clock</td>
</tr>
<tr>
<td>HADDR</td>
<td>21</td>
<td>output</td>
<td>Address</td>
</tr>
<tr>
<td>HWRITE</td>
<td>1</td>
<td>output</td>
<td>Write flag. 1=write, 0=read</td>
</tr>
<tr>
<td>HTRANS</td>
<td>2</td>
<td>output</td>
<td>The transaction type. 2’b00 = IDLE, 2’b10 = NONSEQ</td>
</tr>
<tr>
<td>HWDATA</td>
<td>8</td>
<td>output</td>
<td>Write data</td>
</tr>
<tr>
<td>HRDATA</td>
<td>8</td>
<td>input</td>
<td>Read data</td>
</tr>
</tbody>
</table>
// Compile order
`default_nettype none
`include "ahb_if.sv"

module test;
    ahb_if ahb_bus();
    ahb_slave ahb_slave(ahb_bus.slave);
    ahb_master ahb_master(ahb_bus);
endmodule

module ahb_master(ahb_if ahb_bus);
    localparam IDLE = 2'b00, NONSEQ = 2'b10;
    always #50ns ahb_bus.HCLK = ~ahb_bus.HCLK;
    assign ahb_bus.HTRANS = NONSEQ;
endmodule

module ahb_slave(ahb_if ahb_bus);
endmodule // ahb_slave

interface ahb_if;
    bit     HCLK;
    bit [20:0]  HADDR;
    bit        HWRITE;
    bit [1:0]   HTRANS;
    bit [7:0]   HWDATA;
    logic [7:0] HRDATA;
    localparam IDLE = 2'b00, NONSEQ = 2'b10;
    // At the negative edge of the clock ensure HTRANS is either IDLE or NONSEQ
    always @(negedge HCLK) begin
        if ((HTRANS != IDLE) && (HTRANS != NONSEQ)) $display("Error: HTRANS = %h", HTRANS);
    end
endinterface
4.3 Stimulus Timing

• Controlling the timing in a testbench is difficult
• Easy to have race conditions between testbench and DUT.

```verilog
for (int i=TESTS-1;i>=0;i--) begin
    @(negedge clk);
    read = 1;
    address = address_array[i];
    @(posedge clk); // read is complete
    #5ns;
    Check9Bits(data_read_expect_assoc[address], data_out, address);
    data_read_queue.push_back(data_out);
end
```
4.3.1 Control Timing with a clocking block

• A clocking block:
  • Separates the timing and synchronization details from the functionality of your tests.
  • Enables modeling at higher level of abstraction

• Use a clocking block:
  • To define the clock domain of the testbench
  • Specify timing of synchronous signals relative to the clock domain
  • Drive inputs just after the active edge
  • Sample outputs just before the active edge

• A clocking block can be located in:
  • Modules
  • Interfaces
  • Programs
interface arb_if(input bit clk);
    logic [1:0] grant, request;
    bit rst;
    clocking cb @(posedge clk);
        default input #2ns output #3ns;
        output request;
        input grant;
    endclocking
modport TEST(clocking cb, output rst);
modport DUT(input request, rst, clk, output grant);
endinterface
module test (arb_if.TEST arb_bus);
  initial begin
    arb_bus.cb.request <= 0;
    @arb_bus.cb;
    $display("@%0t: Grant = %b", $time, arb_bus.cb.grant);
  end
endmodule
Another Example

Suppose we want to test the following design

```
initial begin
  @(negedge reg_bus.clk);
  for (int i=0;i<=15;i++) begin
    reg_bus.data_in <= i;
    @(negedge reg_bus.clk);
    check_data_in(i, reg_bus.data_out);
  end
end
```
Timing of stimulus/checker

```verbatim
initial begin
    @(negedge reg_bus.clk);
    for (int i=0; i<=15; i++) begin
        reg_bus.data_in <= i;
        @(negedge reg_bus.clk);
        check_data_in(i, reg_bus.data_out);
    end
end
```

apply stimulus and check output
Issues with this timing

- Inputs to the design are treated as asynchronous signals
- Time from active edge to check edge is only $\frac{1}{2}$ a clock cycle.
- Ideally want to give signals the full clock cycle to settle
The Verilog 2001 event queue

- Active region
  - Blocking assignments occur
    - The right hand side of non-blocking assignments are evaluated
  - #0 blocking assignments are scheduled in the inactive region.
- Left hand side of non-blocking assignments occur in NBA.

---

Diagram:

- Active
- Inactive
- NBA

- From previous time slot to active
- From inactive to next time slot
Moving stimulus/check to pos edge

```verilog
initial begin
    @(posedge reg_bus.clk);
    for (int i=0;i<=15;i++) begin
        reg_bus.data_in <= i;
        @(posedge reg_bus.clk);
        check_data_in(i, reg_bus.data_out);
    end
end
```
Separating stimulus and checking

initial begin
    @(negedge reg_bus.reset);
    @(posedge reg_bus.clk);
    for (int j=0; j<=15; j++) begin
        reg_bus.data_in <= j;
        @(posedge reg_bus.clk);
    end
end // initial begin

initial begin
    @(negedge reg_bus.reset);
    repeat (2) @(posedge reg_bus.clk);
    for (int i=0; i<=15; i++) begin
        @(posedge reg_bus.clk);
        check_data_in(i, reg_bus.data_out);
    end
end

........
Using a Clocking block

interface reg_if(input bit clk);
    bit reset; bit [3:0] data_in; logic [3:0] data_out;
    modport slave (input clk, reset, data_in, output data_out);
    modport master(input clk, data_out, output reset, data_in);
endinterface

interface reg_if(input bit clk);
    bit reset; bit [3:0] data_in; logic [3:0] data_out;

    clocking cb @(posedge clk);
        input data_out;
        output data_in;
        output reset;
    endclocking;

    modport slave (input clk, reset, data_in, output data_out);
    modport master(clocking cb);
endinterface
Clocking Block Example

clocking cb @(posedge clk);
  input data_in;
  input data_out;
endclocking // cb

//Check results
initial begin
  @(negedge reg_bus.cb.reset);
  @reg_bus.cb;
  for (int i=0;i<=15;i++) begin
    reg_bus.cb.data_in <= i;
    @reg_bus.cb;
  end
  $display("At the end of the test the error count is %d", error_count);
  $finish;
end

clocking cb @(posedge clk);
  input data_in;
  input data_out;
endclocking // cb

//Check results
initial begin
  @(negedge reg_bus.cb.reset);
  @reg_bus.cb;
  for (int i=0;i<=15;i++) begin
    @reg_bus.cb;
    check_data_in(i, reg_bus.cb.data_out);
  end
  $display("At the end of the test the error count is %d", error_count);
  $finish;
end
Clocking Block exercise 2

For the following interface:

```verbatim
interface my_if(input bit clk);
    bit write; bit [15:0] data_in; bit [7:0] address;
    logic [15:0] data_out;
endinterface
```

Add a clocking block that:

1. Is sensitive to the negative edge of clock
2. All I/O are synchronous to the clock
3. Creates a modport for the testbench called master and for the DUT called slave
4. Will be used in the master
Clocking Block exercise 2

Add a clocking block that:

1. Is sensitive to the negative edge of clock
2. All I/O are synchronous to the clock
3. Creates a modport for the testbench called master and for the DUT called slave
4. Will be used in the master

interface my_if(input bit clk);
    bit write; bit [15:0] data_in; bit [7:0] address;
    logic [15:0] data_out;

clocking cb @(negedge clk);
    output write, data_in, address;
    input data_out;
endclocking;

modport slave (input clk, write, data_in, address, output data_out);
modport master(clocking cb);
endinterface
CB forces I/O to be synchronous

```verilog
initial begin
  @(negedge reg_bus.cb.reset);
  @reg_bus.cb;
  for (int i=0;i<15;i++) begin
    #25ns;
    reg_bus.cb.data_in <= i;
    @reg_bus.cb;
  end
end
```

```verilog
module my_reg (reg_if reg_bus);
  always @(posedge reg_bus.clk)
    reg_bus.data_out <= #30ns reg_bus.data_in;
endmodule
```
Clocking Block exercise 3

From the clocking block in the last exercise, fill in the following timing diagram.
4.4.6 Specifying delays in clocking blocks

• By default inputs are sampled “just before” the clock and outputs driven “just after” the clock.
• “just before” and “just after” can be explicitly set by specifying input and output skew.

```
clocking cb @(posedge clk);
```

![Diagram of clocking block with input and output skew](image)

![Diagram of test and DUT connections](image)
Input and Output Skew Example

clocking cb @(posedge clk);
    input #75ns data_out; // Sample data_out 75ns before the posedge clk
    output #25ns data_in; // Drive data 25ns after positive edge
    output reset;
endclocking;
Sample 4-26

Can specify skew for all outputs and inputs using `default`

clocking cb @(posedge clk);
    default input #15ns output #10ns;
    output request;
    input grant;
endclocking
Overriding the Default Edge

Can override the default edge for signals in the clock block.

clocking cb @(posedge clk);
    input data_out;
    output data_in;
    output negedge reset;
endclocking;

clocking cb @(posedge clk);
    input data_out;
    output data_in;
    output negedge reset;
endclocking;
Clocking Block exercise 4

• Modify the clocking block in the last exercise to have:
  • Output skew of 25ns for outputs write and address
  • Input skew of 15ns
  • Restrict data_in to only change on the positive edge of the clock
  • Assume the period of test/reg_bus/clk is 100ns.

• Fill in the following timing diagram
Clocking Block Summary

• 1st pass at using clocking blocks:
  • Define a clock edge
    ```
    clocking cb @(posedge clk);
    ```
  • Define the outputs to your DUT and inputs from your DUT
    ```
    input data_out; output data_in; output reset;
    ```
  Use `@<interface_name>.<clock block name>` to advance time
    ```
    @reg_bus.cb;
    ```

• 2nd pass
  • Define input/output skews if necessary
    ```
    input #75ns data_out;
    ```
  • Define alternate edges for testbench outputs if necessary
    ```
    output negedge reset;
    ```

• Rules of thumb
  • Direction in clocking block is referenced to the block that uses it.
  • Put the clocking block in an interface
  • Use non-blocking assignments (`<=`) when assigning to/from signals in a clocking block.
The Verilog 2001 event queue, again

The problem

- Clocking blocks do not eliminate the possibility of race conditions
- Due to the testbench and RTL sharing the same region in the Verilog event queue.
The SystemVerilog event queue

- active
- inactive
- NBA
- observed
- reactive
- re-inactive

from previous time slot

to next time slot
4.5 Program Block considerations

A program block cannot contain:
1. always blocks
2. Instantiated modules
3. other programs

An interface or clocking block is not required to use a program block.
All programs need to finish before $finish is implicitly called
  • Can terminate early using $exit.

```verilog
program automatic test (arb_if.TEST arb_bus);
    initial $display("Hello World");
    ..... 
endprogram
```
4.5.1 The end of simulation

• Simulation ends when the last initial block finishes in a program
• At the end of which initial do you put the error report?
• A program can have a final block
  • final block runs just before simulator terminates
  • Permitted statements same as function to ensure executes in single simulation cycle (zero execution time).

```verbatim
program automatic test;
int errors, warnings;
    initial begin
        .... // Main program activity.
    end
    final
        $display(“Test done with %0d errors and %0d warnings”,
                 errors, warnings);
endprogram
```
program bad_generator (output bit clk, data);
    initial
        forever #5 clk <= ~clk ;
    initial
        forever @(posedge clk)
            data <= ~data;
endprogram
Program Blocks Summary

• Moves testbench into an event region separate from RTL.
• Useful for implicitly ending simulation -> no $finish needed
• Put clock generators in top level module, not program
• Restricts language usage -> no always, instantiated modules, etc
• Not a silver bullet for race conditions.

Follow the controversy
1. Go to wwwverificationguild.com and search for the thread
   Program block: Necessary?
2. Doulos doesn’t teach programs anymore
3. The authors say programs are useful for portability across platforms
4. No consensus across authors of the LRM
Section 4.6 Connecting it all together

module top;
    bit clk;
    always #5 clk = ~clk;

    arb_if arb_bus(.*)
    arb a1 (.*)
    test t1 (.*)
endmodule
4.7 Top-Level Scope

- Compilation unit – the code compiled and treated as 1 logical unit
- Useful for accessing constants in earlier compiled files
- If your simulator supports multi-file compilation

```verilog
`timescale 1ns/1ns
parameter int TIMEOUT = 1_000_000;
const string time_out_msg = "ERROR: Time out";
module top; // top.sv
    test t1();
endmodule

program automatic test; // test.sv
initial begin
    #TIMEOUT;
    $display("%s", time_out_msg); # ERROR: Time out
end
endprogram
```
4.7 Top-Level Scope (cont.)

• By default QuestaSim’s vlog command operates in Single File Compilation Unit mode.
• Use –mfcu switch to vlog to compile in Multi File Compilation Unit mode
• Need to be conscious of compile order.

```
vlog -mfcu test.sv top.sv
```

Undefined variable: TIMEOUT.

• Correct compile order

```
vlog -mfcu top.sv test.sv
```
4.7 Top-Level Scope - $\text{root}$

- $\text{root}$ allows you to unambiguously refer to names starting with the top level scope.
- Compiler goes up in scope to find names.

```plaintext
module top;
  string my_string = "top";

  middle middle();
  string my_string = "middle";

  bottom top();
  string my_string = "bottom";
```

Chapter 4 Copyright 2012 G. Tumbush, C. Spear, v1.5
module top;
  string my_string = "top";
  middle middle();
  initial begin
    #10ns;
    $finish;
  end
endmodule

module bottom;
  string my_string = "bottom";
endmodule

module middle;
  string my_string = "middle";
  bottom top();
  initial begin
    $display("%s", my_string);
    $display("%s", $root.top.my_string);
    $display("%s", $root.top.middle.my_string);
    $display("%s", $root.top.middle.top.my_string);
    $display("%s", top.my_string);
  end
endmodule
4.9 SystemVerilog Assertions

- An assertion succinctly specifies the behavior of the system
- 2 types of assertions
  - Immediate -> Operates in procedural code
  - Simulation use primarily
  - Concurrent -> Operates outside procedural code
  - Usable by other tools as well such as formal verification
  - Describe behavior over time

```verilog
always @(posedge clk)
  traf_light : assert ( green && !red && !yellow && go)
endproperty

property traf_light
  @ (posedge clk) (green && !red && !yellow && go)
endproperty

do_traf_prop : assert property (traf_light);
```
4.9 SystemVerilog Assertions (cont.)

- Property to check: If request is 1 on the positive edge of the clock grant must be 1 on the next positive edge of the clock.

- Verilog 2001 checker:

```verilog
always @(posedge clk) begin
    if (request) begin
        repeat (1) @(posedge clk);
        if (grant !== 1)
            $display("%t: Error, grant != 1", $time);
    end
end
```
4.8.1 Immediate Assertions

Immediate Assertion check:

```verilog
always @(posedge clk) begin
    if (request) begin
        repeat (1) @(posedge clk);
        reg_grnt_assert: assert (grant == 1);
    end
end
```

# ** Error: Assertion error.
# Time: 250 ns Scope: test.reg_grnt_assert File: test.sv Line: 30
4.8.3 Concurrent Assertions

From previous time slot → Preponed

active → NBA → observed → reactive → re-inactive → Postponed → To next time slot

inactive → NBA → observed → reactive → re-inactive → Postponed → To next time slot

Assertion variable values sampled In the Preponed region

Assertion are evaluated In the Observed region
4.8.3 Concurrent Assertions (cont.)

Variable must be static
0, X or Z are interpreted as false

Delay:

```
##N    // sequential delay of N cycles
```

Implication:

```
|->     // Overlapping

( red ##1 green ##1 blue ) |-> ( orange ##1 yellow );

If red/green/blue matches then d is evaluated on same tick

|=>     // Non – Overlapping

( red ##1 green ##1 blue ) |=> ( orange ##1 yellow );

If red/green/blue matches then d is evaluated on next tick
Concurrent Assertion checks:

```vhdl
reg_grnt_assert: assert property (@(posedge clk) (request |-> #1 grant));

x_req_assert: assert property (@(posedge clk) (request !== 1'bx));

x_grnt_assert: assert property (@(posedge clk) (grant !== 1'bx));
```
Viewing Assertions in QuestaSim