Chapter 10
Advanced Interfaces

In Chapter 4 you learned how to connect the design and testbench with interfaces. These physical interfaces represent real signals, similar to the wires that connected ports in Verilog-1995. A testbench uses these interfaces by statically connecting to them through ports. However, for many designs, the testbench needs to connect dynamically to the design.

For example, in a network switch, a single driver class may connect to many interfaces, one for each input channel of the DUT. You wouldn't want to write a unique driver for each channel — instead you want to write a generic driver, instantiate it N times, and have it connect to each of the N physical interfaces. You can do this in SystemVerilog by using a virtual interface, which is merely a handle or pointer to a physical interface. A better name for a virtual interface would be a "ref interface."

You may need to write a testbench that attaches to several different configurations of your design. In another example, a chip may have multiple configurations. In one, the pins might drive a USB bus, whereas in another the same pins may drive an I2C serial bus. Once again, you can use a virtual interface so you can decide at run time which drivers to run in your testbench.

A SystemVerilog interface is more than just signals — you can put executable code inside. This might include routines to read and write to the interface, initial and always blocks that run code inside the interface, and assertions to constantly check the status of the signals. However, do not put testbench code in an interface. Program blocks have been created expressly for building a testbench, including scheduling their execution in the Reactive region, as described in the SystemVerilog LRM.

10.1 Virtual Interfaces with the ATM Router

The most common use for a virtual interface is to allow objects in a testbench to refer to items in a replicated interface using a generic handle rather than the actual name. Virtual interfaces are the only mechanism that can bridge the dynamic world of objects with the static world of modules and interfaces.

10.1.1 The Testbench with Just Physical Interfaces

Chapter 4 showed how to build an interface to connect a 4x4 ATM router to a testbench. Sample 10.1 and 10.2 show the ATM interfaces for the receive and transmit directions.

Sample 10.1 Rx interface with clocking block

```
// Rx interface with modports and clocking block
interface Rx_if (input logic clk);
  logic [7:0] data;
  logic soc, en, clav, rclk;

  clocking cb @(posedge clk);
    output data, soc, clav;  // Directions are relative
    input  en;               // to the testbench
  endclocking : cb

  modport TB (clocking cb);

  modport DUT (output en, rclk,
               input  data, soc, clav);
endinterface : Rx_if
```

Sample 10.2 Tx interface with clocking block

```
// Tx interface with modports and clocking block
interface Tx_if (input logic clk);
  logic [7:0] data;
  logic soc, en, clav, tclk;

  clocking cb @(posedge clk);
      input  data, soc, en;
      output clav;
  endclocking : cb
  modport TB (clocking cb);

  modport DUT (output data, soc, en, tclk,
               input  clav);
endinterface : Tx_if
```

These interfaces can be used in a program block shown in Sample 10.3. This procedural code is hard coded with interface names such as `Rx0` and `Tx0`. Note that in these examples, the top module does not pass a clock to the testbench; instead the tests synchronize with clocking blocks in the interfaces, thus allowing you to work at a higher level of abstraction.

Sample 10.3 Testbench using physical interfaces

```
program automatic test(Rx_if.TB Rx0, Rx1, Rx2, Rx3,
                       Tx_if.TB Tx0, Tx1, Tx2, Tx3,
                       output logic rst);

  bit [7:0] bytes[`ATM_SIZE];

  initial begin
    // Reset the device
    rst <= 1;
    Rx0.cb.data <= '0;
    ...
    receive_cell0;
    ...
  end

  task receive_cell0();
    @(Tx0.cb);
    Tx0.cb.clav <= 1;          // Assert ready to receive
    wait (Tx0.cb.soc == 1);    // Wait for Start of Cell

    for (int i=0; i<`ATM_SIZE; i++) begin
      wait (Tx0.cb.en == 0);   // Wait for enable
        @(Tx0.cb);

      bytes[i] = Tx0.cb.data;
      @(Tx0.cb);
      Tx0.cb.clav <= 0;        // Deassert flow control
    end
  endtask

endprogram
```

Figure 10.1 shows the testbench communicating with the design through virtual interfaces.

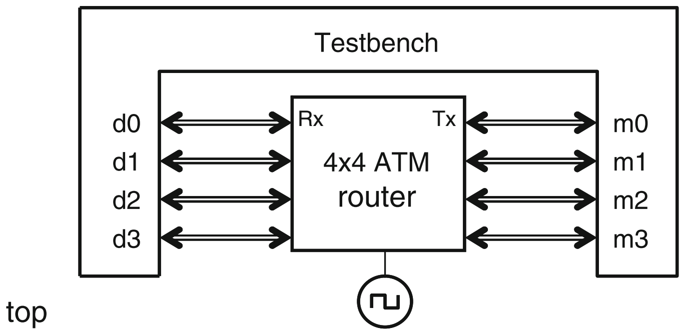

Fig. 10.1 Router and testbench with interfaces

The top level module must connect an array of interfaces to work with the testbench in Sample 10.6. The module in Sample 10.4 instantiates an array of interfaces, and passes this array to the testbench. Since the DUT was written with four RX and four TX interfaces, you need to pass the individual interface array elements into the DUT instance.

Sample 10.4 Top level module with array of interfaces

```
module top;
  logic clk, rst;

  Rx_if Rx[4] (clk);
  Tx_if Tx[4] (clk);

  test       t1 (Rx, Tx, rst);  // See testbench in Sample 10-6
  atm_router a1 (Rx[0], Rx[1], Rx[2], Rx[3],
                 Tx[0], Tx[1], Tx[2], Tx[3],
                 clk, rst);

  initial begin
    clk = 0;
    forever #20 clk = !clk;
    end
endmodule : top
```

10.1.2 Testbench with Virtual Interfaces

A good OOP technique is to create a class that uses a handle to reference an object, rather than a hard-coded object name. In this case, you can make a single Driver class and a single Monitor class, have them operate on a handle to the data, and then pass in the handle at run time.

The program block in Sample 10.5 is still passed the `4 Rx` and `4 Tx` interfaces as ports, as in Sample 10.3, but it creates an array of virtual interfaces, `vRx` and `vTx`. These can now be passed into the constructors for the drivers and monitors.

Sample 10.5 Testbench using virtual interfaces

```
program automatic test(Rx_if.TB Rx0, Rx1, Rx2, Rx3,
                       Tx_if.TB Tx0, Tx1, Tx2, Tx3,
                       output logic rst);

  Driver drv[4];
  Monitor mon[4];
  Scoreboard scb[4];

  virtual Rx_if.TB vRx[4] = '{Rx0, Rx1, Rx2, Rx3};
  virtual Tx_if.TB vTx[4] = '{Tx0, Tx1, Tx2, Tx3};

  initial begin
    foreach (scb[i]) begin
      scb[i] = new(i);
      drv[i] = new(scb[i].exp_mbx, i, vRx[i]);
      mon[i] = new(scb[i].rcv_mbx, i, vTx[i]);
    end
    ...
  end
endprogram
```

You can also skip the virtual interface array variables, and make an array in the port list. These interfaces are passed to the constructors as shown in Sample 10.6.

Sample 10.6 Testbench using virtual interfaces

```
program automatic test(Rx_if.TB Rx[4], Tx_if.TB Tx[4],
                       output logic rst);
...
  initial begin
    foreach (scb[i]) begin
      scb[i] = new(i);
      drv[i] = new(scb[i].exp_mbx, i, Rx[i]);
      mon[i] = new(scb[i].rcv_mbx, i, Tx[i]);
    end
    ...
  end
endprogram
```

The task `monitor::receive_cell` in Sample 10.7 is similar to the task **`receive_cell0`** in Sample 10.3, except it uses the virtual interface name `Tx` instead of the physical interface `Tx0`.

Sample 10.7 Monitor class using virtual interfaces

```
typedef virtual Tx_if vTx_t;

class Monitor;
    int     stream_id;
    mailbox rcv_mbx;
    vTx_t   Tx;

function new(input mailbox rcv_mbx,
             input int     stream_id,
             input vTx_t   Tx);
    this.rcv_mbx = rcv_mbx;
    this.stream_id = stream_id;
    this.Tx = Tx;
endfunction // new


task run();
    ATM_Cell ac;

    fork begin

      // Initialize output signals
      Tx.cb.clav <= 0;                 // Not ready to receive
      @Tx.cb;

      $display("@%0d: Monitor::run[%0d] starting",
               $time, stream_id);
      forever begin
        receive_cell(ac);
      end
    end
    join_none

  endtask : run


  task receive_cell(inout ATM_Cell ac);
    bit [7:0] bytes[];

    bytes = new[ATM_CELL_SIZE];
    ac = new();                      // Initialize the cell

    @Tx.cb;
    Tx.cb.clav <= 1;                 // Assert ready to receive
    while (Tx.cb.soc !== 1'b1)       // Wait for Start of Cell
      @Tx.cb;
    foreach (bytes[i]) begin
      while (Tx.cb.en != 0)       // Wait if enable goes away
        @Tx.cb;

      bytes[i] = Tx.cb.data;
      @Tx.cb;
      Tx.cb.clav <= 0;            // Deassert flow control
    end

    ac.byte_unpack(bytes);
    $display("@%0d: Monitor::run(%0d) received cell vci=%h",
             $time, stream_id, ac.vci);

    // Send cell to scoreboard
    rcv_mbx.put(ac);
  endtask : receive_cell

endclass : Monitor
```

A common mistake when creating a testbench is to leave off the modport name from a virtual interface declaration. The program in Sample 10.5 declares `Tx_if.TB Tx0` in the port list, so it can only assign `Tx0` to a virtual interface declared with the `TB` modport. See the declaration of the virtual interface `Tx` in Sample 10.7.

10.1.3 Connecting the Testbench to an Interface in Port List

This book shows tests that connect to the DUT with interfaces in the port list. This style is comfortable to Verilog users who have always connected modules using signals in ports. Sample 10.8 is the top level module, also known as a test harness, which connects the DUT and test using an interface in the port list.

Sample 10.8 Test harness using an interface in the port list

```
module top;
  bus_ifc bus();    // Instantiate the interface
  test t1(bus);     // Pass to test through port list
  dut  d1(bus);     // Pass to DUT through port list
  ...
endmodule
```

Sample 10.9 shows the program block with an interface in the port list.

Sample 10.9 Test with an interface in the port list

```
program automatic test(bus_ifc bus);
  initial $display(bus.data);  // Use an interface signal
endprogram
```

What happens if you add a new interface to your design? The test harness in Sample 10.10 declares the new bus and puts it in the port lists.

Sample 10.10 Top module with a second interface in the test's port list

```
module top;
  bus_ifc bus();       // Instantiate the interface
  new_ifc newb();      // and a new one
  test t1(bus, newb); // Test with two interfaces
  dut  d1(bus, newb); // DUT with two interfaces
  ...
endmodule
```

Now you have to change the test in Sample 10.9 to include another interface in the port list, giving the test in Sample 10.11.

Sample 10.11 Test with two interfaces in the port list

```
program automatic test(bus_ifc bus, new_ifc newb);
  initial $display(bus.data);  // Use an interface signal
endprogram
```

Adding a new interface to your design means you need to edit all existing tests so they can plug into the test harness. How can you avoid this extra work? Avoid port connections!

10.1.4 Connecting the Test to an Interface with an XMR

Your test needs to connect to the physical interface in the harness, so use a cross module reference (XMR) and a virtual interface in the program block as shown in Sample 10.12. You must use a virtual interface so you can assign it the physical interface in the top level module.

Sample 10.12 Test with virtual interface and XMR

```
program automatic test();
  virtual bus_ifc bus = top.bus; // Cross module reference
  initial $display(bus.data);    // Use an interface signal
endprogram
```

The program connects to the test harness shown in Sample 10.13.

Sample 10.13 Test harness without interfaces in the port list

```
module top;
  bus_ifc bus();    // Instantiate the interface
  test t1();        // Don't use port list for test
  dut  d1(bus);     // Still use port list for DUT
  ...
endmodule
```

This approach is recommended by methodologies such as the VMM to make your test code more reusable. If you add a new interface to your design, as shown in Sample 10.14, the test harness changes, but existing tests don't have to change.

Sample 10.14 Test harness with a second interface

```
module top;
  bus_ifc bus();       // Instantiate the interface
  new_ifc newb();      // and a new one
  test t1();           // Instantiaton remains the same
  dut  d1(bus, newb);
  ...
endmodule
```

The harness in Sample 10.14 works with the test in Sample 10.12 that does not know about the new interface, as well as the test in Sample 10.15 that does.

Sample 10.15 Test with two virtual interfaces and XMRs

```
program automatic test();
  virtual bus_ifc bus = top.bus;
  virtual new_ifc newb = top.newb

  initial begin
    $display(bus.data);  // Use existing interface
    $display(newb.addr); // and new one
  end
endprogram
```

Some methodologies have a rule that makes the connection between tests and harnesses slightly more complicated than with traditional ports, but means you won't have to modify existing tests, even if the design changes. The examples in this book use the simple style of interfaces in the port lists, but you should decide if test reuse is important enough to change your coding style.

10.2 Connecting to Multiple Design Configurations

A common challenge to verifying a design is that it may have several configurations. You could make a separate testbench for each configuration, but this could lead to a combinatorial explosion as you explore every alternative. Instead, you can use virtual interfaces to dynamically connect to the optional interfaces.

10.2.1 A Mesh Design

Sample 10.16 is built of a simple replicated component, an 8-bit counter. This resembles a DUT that has a device such as a network chip or processor that is instantiated repeatedly in a mesh configuration. The key idea is that the top-level module creates an array of interfaces and counters. Now the testbench can connect its array of virtual interfaces to the physical ones.

Sample 10.16 shows the code for the counter's interface, `X_if`. If the code printed the signal values with a `$monitor`, they would display when any signal changed. Instead, the `always` block waits until the clocking block changes, then prints the values of the signals at the end of the time slot with `$strobe`. The result is you are now working at a higher level of abstraction, seeing the values cycle by cycle instead of the individual events.

Sample 10.16 Interface for 8-bit counter

```
interface X_if (input logic clk);
  logic [7:0] din, dout;
  logic reset_l, load;

  clocking cb @(posedge clk);
    output din, load;
    input dout;
  endclocking

  always @cb
    $strobe("@%0t: %m: out=%0d, in=%0d, ld=%0d, r=%0d",
            $time, dout, din, load, reset_l);

  modport DUT (input clk, din, reset_l, load,
               output dout);

  modport TB (clocking cb, output reset_l);
endinterface
```

The simple counter is shown in Sample 10.17.

Sample 10.17 Counter model using `X_if` interface

```
// Simple 8-bit counter with load and active-low reset
module counter(X_if.DUT xi);
  logic [7:0] count;
  assign xi.dout = count;

  always @(posedge xi.clk or negedge xi.reset_l)
    begin
      if (!xi.reset_l)  count <= '0;
      else if (xi.load) count <= xi.din;
      else              count <= count+1;
    end
endmodule
```

The top-level module in Sample 10.18 uses a generate statement to instantiate `NUM_XI` interfaces and counters, but only one testbench.

Sample 10.18 Top-level module with an array of virtual interfaces

```
module top;
  parameter NUM_XI = 2;  // Number of design instances

  // Clock generator
  bit clk;
  initial begin
    clk <= '0;
    forever #20 clk = ~clk;
  end

  // Instantiate NUM_XI interfaces
  X_if xi[NUM_XI] (clk);

  // Instantiate the testbench, passing the number of interfaces
  test #(.NUM_XI(NUM_XI)) tb();

  // Generate NUM_XI counter instances
  generate
  for (genvar i=0; i<NUM_XI; i++)
    begin : count_blk
      counter c (xi[i]);
    end
  endgenerate

endmodule : top
```

In Sample 10.19, the key line in the testbench is where the local virtual interface array, `vxi`, is assigned to point to the array of physical interfaces in the top module, `top.xi`. (Note that this example takes some shortcuts compared to the recommendations in Chapter 8. To simplify Sample 10.18, the environment class has been merged with the test, whereas the generator, agent, and driver layers have been compressed into the driver.)

The testbench assumes there is at least one counter and thus at least one X interface. If your design could have zero counters, you would have to use a dynamic array to hold the virtual interfaces, as a fixed-size array cannot have a size of zero. The actual number of interfaces is passed as a parameter from the top-level module.

Sample 10.19 Counter testbench using virtual interfaces

```
program automatic test #(NUM_XI=2);

  virtual X_if.TB vxi[NUM_XI]; // Virtual interface array
  Driver driver[];

  initial begin
    // Connect local virtual interface to top
    vxi = top.xi;

    // Create N drivers
    driver = new[NUM_XI];
    foreach (driver[i])
      driver[i] = new(vxi[i], i);

    foreach (driver[i]) begin
      automatic int j = i;
      fork
        begin
          driver[j].reset();
          driver[j].load_op();
        end
      join_none
    end

    repeat (10) @(vxi[0].cb);
  end

endprogram
```

Of course in this simple example, you could just pass the interface directly into the `Driver`'s constructor, rather than make a separate variable.

In Sample 10.20, the `Driver` class uses a single virtual interface to drive and sample signals from the counter.

Sample 10.20 `Driver` class using virtual interfaces

```
class Driver;
  virtual X_if.TB xi;
  int id;

  function new(input virtual X_if.TB xi, input int id);
    this.xi = xi;
    this.id = id;
  endfunction

  task reset();
    $display("@%0t: Driver[%0d]: Start reset", $time, id);
    // Reset the device
    xi.reset_l <= 1;
    xi.cb.load <= 0;
    xi.cb.din <= '0;
    @(xi.cb) xi.reset_l <= 0;
    @(xi.cb) xi.reset_l <= 1;
    $display("@%0t:Driver[%0d]: End reset", $time, id);
  endtask : reset

  task load_op();
    $display("@%0t: Driver[%0d]: Start load", $time, id);
    ##1 xi.cb.load <= 1;
    xi.cb.din <= id + 10;

    ##1 xi.cb.load <= 0;
    repeat (5) @(xi.cb);
    $display("@%0t: Driver[%0d]: End load", $time, id);
  endtask : load_op

endclass : Driver
```

10.2.2 *Using `Typedefs` with Virtual Interfaces*

You can reduce the amount of typing, and ensure you always use the correct modport by replacing "`virtual X_if.TB`" with a `typedef`, as shown in Sample 10.21 through 10.23, of the interface, testbench, and driver.

Sample 10.21 Interface with a typedef

```
interface X_if (input logic clk);
  //...
endinterface
typedef virtual X_if.TB vx_if;
```

Sample 10.22 Testbench using a typedef for virtual interfaces

```
program automatic test #(NUM_XI=2);
  vx_if vxi[NUM_XI];       // Virtual interface array
  Driver driver[];
  // ...
endprogram
```

Sample 10.23 Driver using a `typedef` for virtual interfaces

```
class Driver;
  vx_if xi;
  int id;

  function new(input vx_if xi, input int id);
    this.xi = xi;
    this.id = id;
  endfunction
  // ...
endclass : Driver
```

10.2.3 Passing Virtual Interface Array Using a Port

The previous examples passed the array of virtual interfaces using a cross module reference (XMR). An alternative is to pass the array in a port. Since the array in the top module is static and so only needs to be referenced once, the XMR style makes more sense than using a port that normally is used to pass changing values.

Sample 10.24 uses a global parameter to define the number of X interfaces. Here is a snippet of the top module.

Sample 10.24 Testbench using an array of virtual interfaces

```
parameter NUM_XI = 2;  // Number of instances

module top;
  // Instantiate N interfaces
  X_if xi [NUM_XI] (clk);

  ...
  // Instantiate the testbench
  test tb(xi);

endmodule : top
```

The testbench that uses the virtual interfaces is shown in Sample 10.25. It creates an array of virtual interfaces so that it can pass them into the constructor for the driver class, or just pass the interface directly from the port.

Sample 10.25 Testbench passing virtual interfaces with a port

```
program automatic test(X_if xi[NUM_XI]);

  vx_if vxi[NUM_XI];
  Driver driver[];

  initial begin
    // Build phase
    // Connect the local virtual interfaces to the top
    vxi = xi;                     // Assign the interface array
    driver = new[NUM_XI];

    foreach (vxi[i])              // Create NUM_XI drivers
      driver[i] = new(vxi[i], i);

    // Reset phase
    foreach (vxi[i])
      fork
        begin
          driver[i].reset();
          driver[i].load_op();
        end
      join
    //...
  end

endprogram
```

10.3 Parameterized Interfaces and Virtual Interfaces

The example in Section 10.2 shows an 8-bit counter and matching busses. What if you want to vary the counter's width? Verilog-1995 allows you to parameterize modules, and System Verilog extends this concept with parameterized interfaces and virtual interfaces.

First, update the counter, originally shown in Sample 10.17 with parameters. This only requires changing the first few lines. Sample 10.26 now passes the number of interfaces in as a parameter too.

Sample 10.26 Parameterized counter model using `X_if` interface

```
// Simple N-bit counter with load and active-low reset
module counter #(BIT_WIDTH = 8) (X_if.DUT xi);
  logic [BIT_WIDTH-1:0] count;
...
```

Next, Sample 10.27 adds the bit width parameter to the interface in Sample 10.16.

Sample 10.27 Parameterized interface for 8-bit counter

```
interface X_if #(BIT_WIDTH=8) (input logic clk);
  logic [BIT_WIDTH-1:0] din, dout;
...
```

Sample 10.28 shows the parameter being passed into the testbench.

Sample 10.28 Parameterized top-level module with an array of virtual interfaces

```
module top;
  parameter NUM_XI = 2;     // Number of design instances
  parameter BIT_WIDTH = 4; // Width of counter and bus

  // Clock generator
  bit clk;
  initial begin
    clk <= '0;
    forever #20 clk = ~clk;
  end

  // Instantiate N interfaces
  X_if #(.BIT_WIDTH(BIT_WIDTH)) xi[NUM_XI] (clk);

  // Testbench with the number of interfaces and bit width
  test #(.NUM_XI(NUM_XI), .BIT_WIDTH(BIT_WIDTH)) tb();

  // Generate N counter instances
  generate
  for (genvar i=0; i<NUM_XI; i++)
    begin : count_blk
      counter #(.BIT_WIDTH(BIT_WIDTH)) c (xi[i]);
    end
  endgenerate

endmodule : top
```

Lastly are the testbench module and `Driver` class are shown in Samples 10.29 and 10.30. These have virtual interfaces that must be parameterized. The syntax for this is a little tricky, especially when you have a modport. First, the testbench, updated from Sample 10.19. Notice how the parameter goes between the type name and the modport.

Sample 10.29 Parameterized counter testbench using virtual interfaces

```
program automatic test #(NUM_XI=2, BIT_WIDTH=8);
  virtual X_if #(.BIT_WIDTH(BIT_WIDTH)).TB vxi[NUM_XI];

...
```

Sample 10.30 `Driver` class using virtual interfaces

```
class Driver;
  virtual X_if  #(.BIT_WIDTH(BIT_WIDTH)) xi;

  //...

endclass
```

10.4 Procedural Code in an Interface

Just as a class contains both variables and routines, an interface can contain code such as routines, assertions, and `initial` and `always` blocks. Recall that an interface includes the signals and functionality of the communication between two blocks. So the interface block for a bus can contain the signals and also routines to perform commands such as a read or write. The inner workings of these routines are hidden from the external blocks, allowing you to defer the actual implementation. Access to these routines is controlled using the `modport` statement, just as with signals. A task or function is imported into a `modport` so that it is then visible to any block that uses the `modport`.

These routines can be used by both the design and the testbench. This approach ensures that both are using the same protocol, eliminating a common source of testbench bugs. However, not all synthesis tools can handle routines in an interface.

A problem with sharing code between the design and testbench is that the independence between the design and verification teams is lost. If only one person implements the interface protocol for both parts, who checks it?

You can verify a protocol with assertions in an interface. An assertion can check for illegal combinations, such as protocol violations and unknown values. These can display state information and stop simulation immediately so that you can easily debug the problem. An assertion can also fire when good transactions occur. Functional coverage code uses this type of assertion to trigger the gathering of coverage information.

10.4.1 Interface with Parallel Protocol

When creating your system, you may not know whether to choose a parallel or serial protocol. The interface in Sample 10.31 has two tasks, `initiatorSend` and `targetRcv`, that send a transaction between two blocks using the interface signals. It sends the address and data in parallel across two 8-bit buses.

Sample 10.31 Interface with tasks for parallel protocol

```
interface simple_if(input logic clk);
  logic [7:0] addr;
  logic [7:0] data;
  bus_cmd_e cmd;
  modport TARGET
    (input  addr, cmd, data,
     import task targetRcv (output bus_cmd_e c,
                            logic [7:0] a, d));
   modport INITIATOR
     (output addr, cmd, data,
      import task initiatorSend(input bus_cmd_e c,
                                logic [7:0] a, d)
      );

  // Parallel send
  task initiatorSend(input bus_cmd_e c,
                         logic [7:0] a, d);
    @(posedge clk);
    cmd <= c;
    addr <= a;
    data <= d;
  endtask

  // Parallel receive
  task targetRcv(output bus_cmd_e c, logic [7:0] a, d);
    @(posedge clk);
    a = addr;          // Use non-blocking assignments to
    d = data;          // immediately sample the bus values
    c = cmd;           // and avoid race conditions
  endtask
endinterface: simple_if
```

10.4.2 Interface with Serial Protocol

The interface in Sample 10.32 implements a serial interface for sending and receiving the address and data values. It has the same interface and routine names as Sample 10.31, so you can swap between the two without having to change any design or testbench code.

Sample 10.32 Interface with tasks for serial protocol

```
interface simple_if(input logic clk);
  logic addr;
  logic data;
  logic start = 0;
  bus_cmd_e cmd;

  modport TARGET(input  addr, cmd, data,
               import task targetRcv (output bus_cmd_e c,
                                      logic [7:0] a, d));
  modport INITIATOR(output addr, cmd, data,
                 import task initiatorSend(input bus_cmd_e c,
                                          logic [7:0] a, d));

  // Serial send
  task initiatorSend(input bus_cmd_e c, logic [7:0] a, d);
    @(posedge clk);
    start <= 1;
    cmd <= c;
    foreach (a[i]) begin
      addr <= a[i];
      data <= d[i];
      @(posedge clk);
      start <= 0;
    end
    cmd <= IDLE;
  endtask

  // Serial receive
  task targetRcv(output bus_cmd_e c, logic [7:0] a, d);
    @(posedge start);
    c = cmd;
    foreach (a[i]) begin
      @(posedge clk);
      a[i] = addr;
      d[i] = data;
    end
  endtask

endinterface: simple_if
```

10.4.3 Limitations of Interface Code

Tasks in interfaces are fine for RTL, where the functionality is strictly defined. However, these tasks are a poor choice for any type of verification IP. Interfaces and their code cannot be extended, overloaded, or dynamically instantiated based on configuration. An interface cannot have private data. Any code for verification needs maximum flexibility and configurability, and so should go in classes that run in a program block.

10.5 Conclusion

The interface construct in SystemVerilog provides a powerful technique to group together the connectivity, timing, and functionality for the communication between blocks. In this chapter you saw how you can create a single testbench that connects to many different design configurations containing multiple interfaces. Your signal layer code can connect to a variable number of physical interfaces at run time with virtual interfaces. Additionally, an interface can have routines that drive the signals and assertions to check the protocol, but put the test in a program block, not an interface.

In many ways, an interface can resemble a class with pointers, encapsulation, and abstraction. This lets you create an interface to model your system at a higher level than Verilog's traditional ports and wires. Just remember to keep the testbench in the program block.

10.6 Exercises

1. Complete the following code, as indicated by the comments.

```
class Driver;
  ...
  // Declare a virtual interface for the DUT
  function new(input inst_mbox #(Instruction) agt2drv,
               /* complete the argument list */);
     this.agt2drv = agt2drv;
     // Save the virtual interface argument in
     // the class-level variable
  endfunction
endclass

class Environment;
  Driver drv;
  .....
  drv = new(agt2drv, /* complete the argument list */);
  .....
endclass
```

2. Using the solution to Exercise 1, complete the following code as indicated by the comments.

```
program automatic test(risc_spm_if risc_bus);
   import my_package::*;
   Environment env;
   initial begin
     // Create object referenced by env handle
   end
endprogram
```

3. Modify the following program declaration to use cross module references (XMR). Assume the top module that contains the interface is named `top`.

```
program automatic test(risc_spm_if risc_bus);
  ...
endprogram
```

Modify the following instantiation of program `test` to use cross module references (XMR).

```
`include "risc_spm_if.sv"
 module top;
   ....
   test t1(risc_bus);
   ....
 endmodule
```

4. Expand the solution to Exercise 3 to create `NUM_RISC_BUS` environments and create `NUM_RISC_BUS` interfaces.

5. Expand the solution to Exercise 3 to use a `typedef` for the virtual interface.

6. Modify the following interface to use a parameter, `ADDRESS_WIDTH`. By default the addressing space supports 256 words.

```
interface risc_spm_if (input bit clk);

  bit rst;
  bit    [7:0] data_out;

  logic [7:0] address;
  logic [7:0] data_in;
  logic       write;
  modport DUT (input clk, data_out,
               output address, data_in, write);
endinterface
```