Chapter 4
Connecting the Testbench and Design

There are several steps needed to verify a design: generate stimulus, capture responses, determine correctness, and measure progress. However, first you need the proper testbench, connected to the design, as shown in Fig. 4.1.

Your testbench wraps around the design, sending in stimulus and capturing the design's response. The testbench forms the "real world" around the design, mimicking the entire environment. For example, a processor model needs to connect to various buses and devices, which are modeled in the testbench as bus functional models. A networking device connects to multiple input and output data streams that are modeled based on standard protocols. A video chip connects to buses that send in commands, and then forms images that are written into memory models. The key concept is that the testbench simulates everything not in the design under test.

Your testbench needs a higher-level way to communicate with the design than Verilog's ports and the error-prone pages of connections. You need a robust way to describe the timing so that synchronous signals are always driven and sampled at the correct time and all interactions are free of the race conditions so common to Verilog models.

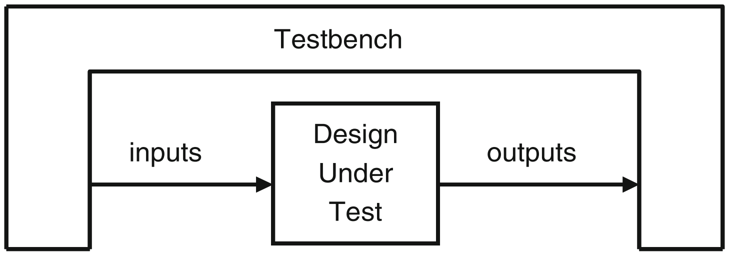


Fig. 4.1 The testbench – design environment

4.1 Separating the Testbench and Design

In an ideal world, all projects have two separate groups: one to create the design and one to verify it. In the real world, limited budgets may require you to wear both hats. Each team has its own set of specialized skills, such as creating synthesizable RTL code, or figuring out new ways to find bugs in the design. These two groups each read the original design specification and make their own interpretations. The designer has to create code that meets that specification, whereas your job as the verification engineer is to create scenarios where the design does not match its description.

Likewise, your testbench code is in a separate block from design code. In classic Verilog, each goes in a separate module. However, using a module to hold the testbench often causes timing problems around driving and sampling, so SystemVerilog introduces the program block to separate the testbench, both logically and temporally. For more details, see Section 4.3.

As designs grow in complexity, the connections between the blocks increase. Two RTL blocks may share dozens of signals, which must be listed in the correct order for them to communicate properly. One mismatched or misplaced connection and the design will not work. You can reduce errors by using the connect-by-name syntax, but this more than doubles your typing burden. If it is a subtle error, such as swapping pins that only toggle occasionally, you may not notice the problem for some time. Worse yet is when you add a new signal between two blocks. You have to edit not only the blocks to add the new port but also the higher-level modules that wire up the devices. Again, one wrong connection at any level and the design stops working. Or worse, the system only fails intermittently!

The solution is the interface, the SystemVerilog construct that represents a bundle of wires. Additionally, you can specify timing, signal direction, and even add functional code. An interface is instantiated like a module but is connected to ports like a signal.

4.1.1 Communication Between the Testbench and DUT

The next few sections show a testbench connected to an arbiter, using individual signals and again using interfaces. Figure 4.2 is a diagram of the top level design including a testbench, arbiter, clock generator, and the signals that connect them. This DUT (Design Under Test) is a trivial design, so you can concentrate on the SystemVerilog concepts and not get bogged down in the design. At the end of the chapter, an ATM router is shown.

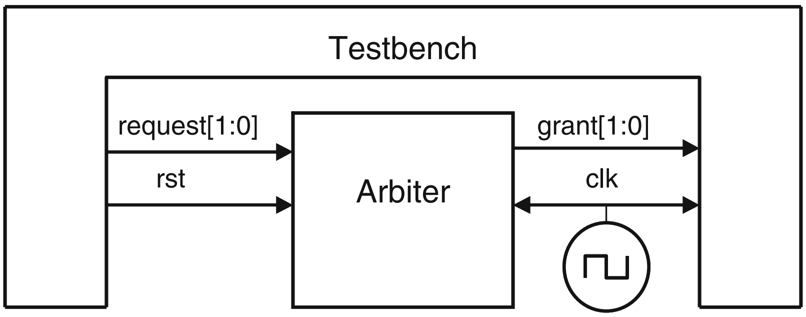


Fig. 4.2 Testbench – Arbiter without interfaces

4.1.2 Communication with Ports

The following code shows the steps needed to connect an RTL block to a testbench. First is the header for the arbiter model, shown in Sample 4.1. This uses the Verilog-2001 style port declarations where the type and direction are in the header. Some code has been left out for clarity.

As discussed in Section 2.1.1, SystemVerilog has expanded the classic `reg` type so that you can use it like a `wire` to connect blocks. In recognition of its new capabilities, the `reg` type has the new name of `logic`. The only place where you cannot use a `logic` variable is a net with multiple structural drivers, where you must use a net such as `wire`.

Sample 4.1 Arbiter model using ports

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

  always @(posedge clk or posedge rst) begin
    if (rst)
      grant <= 2'b00;
    else if (request[0])    // High priority
      grant <= 2'b01;
    else if (request[1])    // Low priority
      grant <= 2'b10;
    else
      grant <= '0;
  end
endmodule
```

The testbench in Sample 4.2 is kept in a module to separate it from the design. Typically, it connects to the design with ports.

Sample 4.2 Testbench module using ports

```
module test_with_port (input  logic [1:0] grant,
                       output logic [1:0] request,
                       output bit   rst,
                       input  bit   clk);
  initial begin
    @(posedge clk);
    request <= 2'b01;
    $display("@%0t: Drove req=01", $time);
    repeat (2) @(posedge clk);
    if (grant == 2'b01)
      $display("@%0t: Success: grant == 2'b01", $time);
    else
      $display("@%0t: Error: grant != 2'b01", $time);
    $finish;
  end
endmodule
```

The top module connects the testbench and DUT, and includes a simple clock generator.

Sample 4.3 Top-level module with ports

```
module top;
  logic [1:0] grant, request;
  bit   clk;
  always #50 clk = ~clk;

  arb_with_port  a1 (grant, request, rst, clk);   // Sample 4-1
  test_with_port t1 (grant, request, rst, clk);   // Sample 4-2
endmodule
```

In Sample 4.3, the modules are simple, but real designs with hundreds of pins require pages of signal and port declarations. All these connections can be error prone. As a signal moves through several layers of hierarchy, it has to be declared and connected over and over. Worst of all, if you just want to add a new signal, it has to be declared and connected in multiple files. SystemVerilog interfaces can help in each of these cases.

4.2 The Interface Construct

Designs have become so complex that even the communication between blocks may need to be separated out into separate entities. To model this, SystemVerilog uses the interface construct that you can think of as an intelligent bundle of wires. It contains the connectivity, synchronization, and optionally, the functionality of the communication between two or more blocks and, optionally, error checking. They connect design blocks and/or testbenches.

Design-level interfaces are covered in Sutherland (2006). This book concentrates on interfaces that connect design blocks and testbenches.

4.2.1 Using an Interface to Simplify Connections

The first improvement to the arbiter example is to bundle the wires together into an interface. Figure 4.3 shows the testbench and arbiter, communicating using an interface. Note how the interface extends into the two blocks, representing the drivers and receivers that are functionally part of both the test and the DUT. The clock can be part of the interface or a separate port.

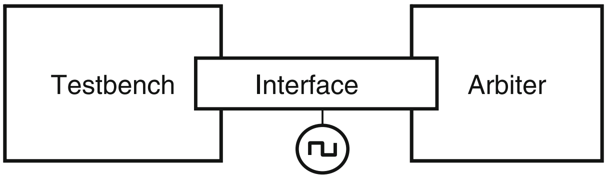


Fig. 4.3 An interface straddles two modules

The simplest interface is just a bundle of bidirectional signals as shown in Sample 4.4. Use the `logic` data type so you can drive the signals from procedural statements.

Sample 4.4 Simple interface for arbiter

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

Sample 4.5 is the device under test, the arbiter, that uses an interface instead of ports.

Sample 4.5 Arbiter using a simple interface

```
module arb_with_ifc (arb_if arbif);
  always @(posedge arbif.clk or posedge arbif.rst)
    begin
    if (arbif.rst)
      arbif.grant <= '0;
    else if (arbif.request[0])     // High priority
      arbif.grant <= 2'b01;
    else if (arbif.request[1])     // Low priority
      arbif.grant <= 2'b10;
    else
      arbif.grant <= '0;
  end
endmodule
```

Sample 4.6 shows the testbench. You refer to a signal in an interface by making a hierarchical reference using the instance name: `arbif.request`. Interface signals should always be driven using nonblocking assignments. This is explained in more detail in Section 4.4.3 and 4.4.4.

Sample 4.6 Testbench using a simple arbiter interface

```
module test_with_ifc (arb_if arbif);
   initial begin
    @(posedge arbif.clk);
    arbif.request <= 2'b01;
    $display("@%0t: Drove req=01", $time);
    repeat (2) @(posedge arbif.clk);
    if (arbif.grant != 2'b01)
      $display("@%0t: Error: grant != 2'b01", $time);
    $finish;
  end
endmodule
```

All these blocks are instantiated and connected in the `top` module as shown in Sample 4.7.

Sample 4.7 Top module with a simple arbiter interface

```
module top;
  bit clk;
  always #50 clk = ~clk;

  arb_if        arbif(clk);    // From Sample 4-4
  arb_with_ifc  a1 (arbif);    // From Sample 4-5
  test_with_ifc t1(arbif);     // From Sample 4-6
endmodule : top
```

You can see an immediate benefit, even on this small device: the connections become cleaner and less prone to mistakes. If you wanted to put a new signal in an interface, you would just have to add it to the interface definition and the modules that actually used it. You would not have to change any module such as `top` that just passes the interface through. This language feature greatly reduces the chance for wiring errors.

This book only shows interfaces with a single clock that is connected to a generator at the top level. If your interface requires multiple clocks, treat them like the other signals inside the interface, and connect the interface to a clock generator. You are more productive if you work at a high level and treat the interface as a cycle based construct. The next level up is transaction-based, which is beyond typical RTL code.

Make sure you declare your interfaces outside of modules and program blocks. If you forget, expect all sorts of trouble. Some compilers may not support defining an interface inside a module. If allowed, the interface would be local to the module and thus not visible to the rest of the design. Sample 4.8 shows the common mistake of including the interface definition right after other include statements.

Sample 4.8 Bad test module includes interface

```
module bad_test(arb_if arbif);
`include "MyTest.sv"   // Legal include
`include "arb_if.sv"   // BAD:Interface hidden in module
...
```

4.2.2 Connecting Interfaces and Ports

If you have a Verilog-2001 legacy design with ports that cannot be changed to use an interface, you can just connect the interface's signals to the individual ports. Sample 4.9 connects the original arbiter from Sample 4.1 to the interface in Sample 4.4.

Sample 4.9 Connecting an interface to a module that uses ports

```
module top;
  bit  clk;
  always #50 clk = ~clk;

  arb_if arbif(clk);
  arb_with_port a1 (.grant  (arbif.grant), // .port (ifc.signal)
                    .request (arbif.request),
                    .rst     (arbif.rst),
                    .clk     (arbif.clk));
  test_with_ifc t1(arbif);   // From Sample 4-6
endmodule : top
```

4.2.3 Grouping Signals in an Interface Using Modports

Sample 4.5 uses a point-to-point connection scheme with no signal directions in the interface. The original modules using ports had this information that the compiler uses to check for wiring mistakes. The `modport` construct in an interface lets you group signals and specify directions. The `MONITOR` modport in Sample 4.10 allows you to connect a monitor module to the interface.

Sample 4.10 Interface with modports

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

  modport TEST (output request, rst,
                input  grant, clk);

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

  modport MONITOR (input request, grant, rst, clk);

endinterface
```

Sample 4.11 shows the arbiter model and testbench, with the modport in their port connection list. Note that you put the modport name, `DUT` or `TEST`, after the interface name, `arb_if`. Other than the modport name, these are identical to the previous examples.

Sample 4.11 Arbiter model with interface using modports

```
module arb_with_mp (arb_if.DUT arbif);
  ...
endmodule
```

Sample 4.12 Testbench with interface using modports

```
module test_with_mp (arb_if.TEST arbif);
  ...
endmodule
```

Even though the code didn't change much (except that the interface grew larger), this interface more accurately represents the real design, especially the signal direction.

There are two ways to use these modport names in your design. You can specify them in the modules that connect to the interface signals. In this case, the top model does not change from Sample 4.7, except for the module names. This book recommends this style, as the modport is an implementation detail that should not clutter the top level module.

The alternative is to specify the modport when you instantiate the module as shown in Sample 4.13.

Sample 4.13 Top level module with modports

```
module top;
  logic [1:0] grant, request;
  bit   clk;
  always #50 clk = ~clk;

  arb_if       arbif(clk);         // Sample 4-10
  arb_with_mp  a1 (arbif.DUT);     // Sample 4-11
  test_with_mp t1 (arbif.TEST);    // Sample 4-12
endmodule
```

With this style, you have the flexibility to instantiate a module more than once, with each instance connected to a different modport, that is, a different subset of interface signals. For example, a byte-wide RAM model could connect to one of four slots on a 32-bit bus. In this case, you would need to specify the modport when you instantiate the module, not in the module itself.

Note that modports are defined in the interface, and specified in the module port list, but never in the signal name. The name `arb_if.TEST.grant` is illegal!

4.2.4 Using Modports with a Bus Design

Not every signal needs to go in every modport. Consider a CPU – memory bus modeled with an interface. The CPU is the bus master and drives a subset of the signals, such as `request`, `command`, and `address`. The memory is a slave and receives those signals and drives `ready`. Both master and slave drive `data`. The bus arbiter only looks at `request` and `grant`, and ignores all other signals. So your interface would have three modports for master, slave, and arbiter, plus an optional monitor modport.

4.2.5 Creating an Interface Monitor

You can create a bus monitor using the `MONITOR` modport. Sample 4.14 shows a trivial monitor for the arbiter. For a real bus, you could decode the commands and print the status: completed, failed, etc.

Sample 4.14 Arbiter monitor with interface using modports

```
module monitor (arb_if.MONITOR arbif);

  always @(posedge arbif.request[0]) begin
    $display("@%0t: request[0] asserted", $time);
    @(posedge arbif.grant[0]);
    $display("@%0t: grant[0] asserted", $time);
  end

  always @(posedge arbif.request[1]) begin
    $display("@%0t: request[1] asserted", $time);
    @(posedge arbif.grant[1]);
    $display("@%0t: grant[1] asserted", $time);
  end
endmodule
```

4.2.6 Interface Trade-Offs

An interface cannot contain module instances, only instances of other interfaces. There are trade-offs in using interfaces with modports as compared with traditional ports connected with signals.

The advantages to using an interface are as follows.

- An interface is ideal for design reuse. When two blocks communicate with a specified protocol using more than two signals, consider using an interface. If groups of signals are repeated over and over, as in a networking switch, you should additionally use virtual interfaces, as described in Chapter 10.
- The interface takes the jumble of signals that you declare over and over in every module or program and puts it in a central location, reducing the possibility of misconnecting signals.
- To add a new signal, you just have to declare it once in the interface, not in higher-level modules, once again reducing errors.
- Modports allow a module to easily tap a subset of signals from an interface. You can specify signal direction for additional checking.

The disadvantages of using an interface are as follows.

- For point-to-point connections, interfaces with modports are almost as verbose as using ports with lists of signals. Interfaces have the advantage that all the declarations are still in one central location, reducing the chance for making an error.
- You must now use the interface name in addition to the signal name, possibly making the modules more verbose, but more readable for debugging.
- If you are connecting two design blocks with a unique protocol that will not be reused, interfaces may be more work than just wiring together the ports.

- It is difficult to connect two different interfaces. A new interface (`bus_if`) may contain all the signals of an existing one (`arb_if`), plus new signals (address, data, etc.). You may have to break out the individual signals and drive them appropriately.

4.2.7 *More Information and Examples*

The SystemVerilog LRM specifies many other ways for you to use interfaces. See Sutherland (2006) for more examples of using interfaces for design.

4.2.8 *Logic vs. Wire in an Interface*

This book recommends declaring the signals in your interface as `logic` while the VMM has a rule that says to use a `wire`. The difference is ease-of-use vs. reusability.

If your testbench drives an asynchronous signal in an interface with a procedural assignment, the signal must be a `logic` type. A `wire` can only be driven with a continuous assignment statement. Signals in a clocking block are always synchronous and can be declared as `logic` or `wire`. Sample 4.15 shows how the `logic` signal can be driven directly, whereas the `wire` requires additional code.

Sample 4.15 Driving logic and wires in an interface

```
interface asynch_if();
  logic l;
  wire w;
endinterface


module test(asynch_if ifc);
  logic local_wire;
  assign ifc.w = local_wire;

  initial begin
    ifc.l <= 0;       // Drive asych logic directly ...
    local_wire <= 1; // but drive wire through assign
    ...
  end
endmodule
```

Another reason to use `logic` for interface signals is that the compiler will give an error if you unintentionally use multiple structural drivers.

The VMM takes a more long-term approach. Take the case where you have created test code that works well on the current project and is later used in a new design.

What if your interface with all its `logic` signals is connected such that now a signal has multiple structural drivers? The engineers will have to change that `logic` to a `wire`, and, if the signal does not go through a clocking block, change the procedural assignment statements. Now there are two versions of the interface, and existing tests must be modified before they can be reused. Rewriting good code goes against the VMM principles.

4.3 Stimulus Timing

The timing between the testbench and the design must be carefully orchestrated. At a cycle level, you need to drive and receive the synchronous signals at the proper time in relation to the clock. Drive too late or sample too early, and your testbench is off a cycle. Even within a single time slot (for example, everything that happens at time 100ns), mixing design and testbench events can cause a race condition, such as when a signal is both read and written at the same time. Do you read the old value, or the one just written? In Verilog, nonblocking assignments help when a test module drives the DUT, but the test could not always be sure it sampled the last value driven by the design. SystemVerilog has several constructs to help you control the timing of the communication.

4.3.1 Controlling Timing of Synchronous Signals with a Clocking Block

An interface should contain a clocking block to specify the timing of synchronous signals relative to the clocks. Clocking blocks are mainly used by testbenches but also allow you to create abstract synchronous models. Signals in a clocking block are driven or sampled synchronously, ensuring that your testbench interacts with the signals at the right time. Synthesis tools do not support clocking blocks, so your RTL code can not take advantage of them. The chief benefit of clocking blocks is that you can put all the detailed timing information in here, and not clutter your testbench.

An interface can contain multiple clocking blocks, one per clock domain, as there is a single clock expression in each block. Typical clock expressions are `@(posedge clk)` for a single edge clock and `@(clk)` for a DDR (double data rate) clock.

You can specify a clock skew in the clocking block using the `default` statement, but the default behavior is that input signals are sampled just before the design executes, and the outputs are driven back into the design during the current time slot. The next section provides more details on the timing between the design and testbench.

Once you have defined a clocking block, your testbench can wait for the clocking expression with `@arbif.cb` rather than having to spell out the exact clock and edge. Now if you change the clock or edge in the clocking block, you do not have to change your testbench.

Sample 4.16 is similar to Sample 4.10 except that the `TEST` modport now treats `request` and `grant` as synchronous signals. The clocking block `cb` declares that the signals are active on the positive edge of the clock. The signal directions are relative to the `modport` where they are used. So `request` is a synchronous output in the `TEST` modport, and `grant` is an synchronous input. The signal `rst` is asynchronous in the `TEST` modport.

Sample 4.16 Interface with a clocking block

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

  clocking cb @(posedge clk);    // Declare cb
    output request;
    input grant;
  endclocking

  modport TEST (clocking cb,     // Use cb
                output rst);

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

// Trivial test, see Sample 4-21 for a better one
module test_with_cb(arb_if.TEST arbif);
  initial begin
    @arbif.cb;
    arbif.cb.request <= 2'b01;
    @arbif.cb;
    $display("@%0t: Grant = %b", $time, arbif.cb.grant);
    @arbif.cb;
    $display("@%0t: Grant = %b", $time, arbif.cb.grant);
    $finish;
  end
endmodule
```

4.3.2 *Timing Problems in Verilog*

Your testbench needs to be separate from the design, not just logically but also temporally. Consider how a hardware tester interacts with a chip for synchronous signals. In a real hardware design, the DUT's storage elements latch their inputs from the tester at the active clock edge. These values propagate through the storage element outputs, and then the logic clouds to the inputs of the next storage element. The time from the input of the first storage to the next must be less than a clock cycle.

So a hardware tester needs to drive the chip's inputs at the clock edge, and read the outputs just before the following edge.

A testbench has to mimic this tester behavior. It should drive on or after the active clock edge, and should sample as late as possible as allowed by the protocol timing specification, just before the active clock edge.

If the DUT and testbench are made of Verilog modules only, this outcome is nearly impossible to achieve. If the testbench drives the DUT at the clock edge, there could be race conditions. What if the clock propagates to some DUT inputs before the testbench stimulus, but is a little later to other inputs? From the outside, the clock edges all arrive at the same simulation time, but in the design, some inputs get the value driven during the last cycle, whereas other inputs get values from the current cycle.

One way around this problem is to add small delays to the system, such as `#0`. This forces the thread of Verilog code to stop and be rescheduled after all other code. Invariably though, a large design has several sections that all want to execute last. Whose `#0` wins out? It could vary from run to run and be unpredictable between simulators. Multiple threads using `#0` delays cause indeterministic behavior. Avoid using `#0` as it will make your code unstable and not portable.

The next solution is to use a larger delay, `#1`. RTL code has no timing, other than clock edges, so one time unit after the clock, the logic has settled. However, what if one module uses a time precision of 1ns, whereas another used a resolution of just 10ps? Does that `#1` mean 1ns, 10ps, or something else? You want to drive as soon as possible after the clock cycle with the active clock edge, but not during that time, and before anything else can happen. Worse yet, your DUT may contain a mix of RTL code with no delays and gate code with delays. Just as you should avoid using `#0`, stay away from `#1` delays to fix timing problems. See Cummings (2000) and other papers by him for additional guidelines.

4.3.3 Testbench – Design Race Condition

Sample 4.17 shows a potential race condition between the testbench and design. The race condition occurs when the test drives the `start` signal and then the other ports. The memory is waiting on the `start` signal and could wake up immediately, whereas the `write` signal still has its old value, while `addr` and `data` have new values. This behavior is perfectly legal according to the LRM. You could delay all these signals slightly by using nonblocking assignments, as recommended by Cummings (2000), but remember that the testbench and the design are both using these assignments. It is still possible to get a race condition between the testbench and design.

Sampling the design outputs has a similar problem. You want to grab the values at the last possible moment, just before the active clock edge. Perhaps you know the next clock edge is at 100ns. You can't sample right at the clock edge at 100ns, as some design values may have already changed. You should sample at `Tsetup` just before the clock edge.

Sample 4.17 Race condition between testbench and design

```
module memory(input wire start, write,
              input wire [7:0] addr,
              inout wire [7:0] data);
  logic [7:0] mem[256];
  always @(posedge start) begin
    if (write)
      mem[addr] <= data;
    ...
  end
endmodule

module test(output logic start, write,
            output logic [7:0] addr, data);
  initial begin
    start = 0;            // Initialize signals
    write = 0;
    #10;                  // Short delay
    addr = 8'h42;         // Start first command
    data = 8'h5a;
    start = 1;
    write = 1;
    ...
  end
endmodule
```

4.3.4 The Program Block and Timing Regions

The root of the problem is the mixing of design and testbench events during the same time slot, though even in pure RTL the same problem can happen. Good coding guidelines such as proper use of nonblocking assignments can reduce these race conditions, but improperly coded assignments have the habit of creeping in. What if there were a way you could separate these events temporally, just as you separated the code? At 100ns, your testbench could sample the design outputs before the clock has had a chance to change and any design activity has occurred. By definition, these values would be the last possible ones from the previous time slot. Then, after all the design events are done, your testbench would start.

How does SystemVerilog know to schedule the testbench events separately from the design events? In SystemVerilog, your testbench code is in a program block, which is similar to a module in that it can contain code and variables and be instantiated in other modules. However, a program cannot have any hierarchy such as instances of modules, interfaces, or other programs.

A new region of the time slot was introduced in SystemVerilog as shown in Fig. 4.4. In Verilog, most events are executed in the Active region. There are dozens of other regions for nonblocking assignments, PLI execution, etc., but they can be

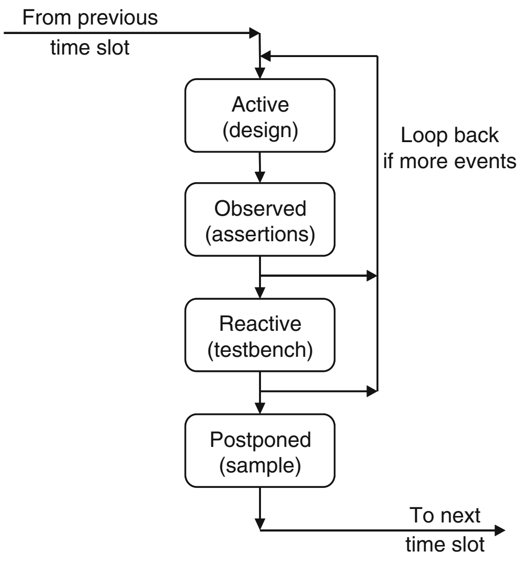


Fig. 4.4 Main regions inside a SystemVerilog time step

ignored for the purposes of this book. See Table 4.1, the LRM, and Cummings (2006) for more details on the SystemVerilog event regions.

First to execute during a time slot is the Active region, where design events run. These include your traditional RTL and gate code plus the clock generator. The second region is the Observed region, where SystemVerilog Assertions are evaluated. Following that is the Reactive region where the testbench code in a program executes. Note that time does not strictly flow forwards — events in the Observed and Reactive regions can trigger further design events in the Active region in the current cycle. Last is the Postponed region, which samples signals at the end of the time slot, in the readonly period, after design activity has completed.

Table 4.1 Primary SystemVerilog scheduling regions

Name	*Activity*
Active	Simulation of design code in modules
Observed	Evaluation of SystemVerilog Assertions
Reactive	Execution of testbench code in programs
Postponed	Sampling design signals for testbench input

Sample 4.18 shows part of the testbench code for the arbiter. Note that the statement `@arbif.cb` waits for the active edge of the clocking block, `@(posedge clk)`, as shown in Sample 4.16. This sample shows that your testbench code is

written at a slightly higher level of abstraction, using cycle-by-cycle timing instead of worrying about individual clock edges.

Section 4.4 explains more about the driving and sampling of interface signals.

Sample 4.18 Testbench using interface with clocking block

```
program automatic test (arb_if.TEST arbif);
  initial begin
    @arbif.cb;
    arbif.cb.request <= 2'b01;
    $display("@%0t: Drove req=01", $time);
    repeat (2) @arbif.cb;
    if (arbif.cb.grant == 2'b01)
      $display("@%0t: Success: grant == 2'b01", $time);
    else
      $display("@%0t: Error: grant != 2'b01", $time);
    end
endprogram : test
```

Your test should be contained in a single program. You should use OOP to build a dynamic, hierarchical testbench from objects instead of modules. A simulation may have multiple program blocks if you are using code from other people or combining several tests.

As discussed in Section 3.6.1, you should always declare your program block as `automatic` so that it behaves more like the routines in stack-based languages you may have worked with, such as C.

Note that not all vendors regard program blocks equally. See Rich (2009) for an alternate opinion.

4.3.5 Specifying Delays Between the Design and Testbench

The default timing of the clocking block is to sample inputs with a skew of `#1step` and to drive the outputs with a delay of `#0`. The `1step` delay specifies that signals are sampled in the Postponed region of the previous time slot, before any design activity. So you get the output values just before the clock changes. The testbench outputs are synchronous by virtue of the clocking block, so they flow directly into the design. The program block, running in the Reactive region, generates the stimulus that is applied to the DUT, which is then evaluated in the Active region during the same time slot. The DUT evaluates its logic and drives its outputs, which are the inputs to the testbench through the clocking blocks. These are then sampled in the Postponed region and the cycle repeats. If you have a design background, you can remember this by imagining that the clocking block inserts a synchronizer between the design and testbench, as shown in Fig. 4.5. With proper use of program and clocking blocks, race conditions between the testbench and DUT can be all but eliminated.

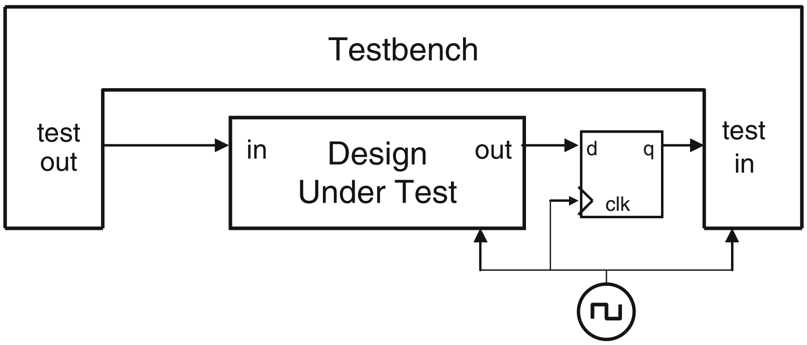


Fig. 4.5 A clocking block synchronizes the DUT and testbench

4.4 Interface Driving and Sampling

Your testbench needs to drive and sample signals from the design, primarily through interfaces with clocking blocks. The next section uses the arbiter interface from Sample 4.16 and the top-level module from Sample 4.9.

Asynchronous signals such as `rst` pass through the interface with no delays. The signals in the clocking block get synchronized as shown in the sections below.

4.4.1 Interface Synchronization

You can use the Verilog `@` and `wait` constructs to synchronize the signals in a testbench. Sample 4.19 shows the various constructs.

Sample 4.19 Signal synchronization

```
program automatic test(bus_if.TB bus);
  initial begin
    @bus.cb;                    // Continue on active edge
                                // in clocking block
    repeat (3) @bus.cb;         // Wait for 3 active edges
    @bus.cb.grant;              // Continue on any edge
    @(posedge bus.cb.grant);    // Continue on posedge
    @(negedge bus.cb.grant);    // Continue on negedge
    wait (bus.cb.grant==1);     // Wait for expression
                                // No delay if already true
    @(posedge bus.cb.grant or
      negedge bus.rst);         // Wait for several signals
  end
endprogram
```

4.4.2 *Interface Signal Sample*

When you read a signal from a clocking block, you get the value sampled from just before the last clock edge, i.e., from the Postponed region. Sample 4.20 shows a program block that reads the synchronous grant signal from the DUT. The `arb` module drives `grant` to 1 & 2 in the middle of the 100ns cycle, and then to 3 exactly at the clock edge. This code is for illustration only and is not real, synthesizable RTL.

Sample 4.20 Synchronous interface sample and drive from module

```
program automatic test(arb_if.TEST arbif);
  initial begin
    $monitor("@%0t: grant=%h", $time, arbif.cb.grant);
    #500ns $display("End of test");
  end
endprogram

module arb_dummy(arb_if.DUT arbif);
  initial
    fork
      # 70ns arbif.grant = 1;
      #170ns arbif.grant = 2;
      #250ns arbif.grant = 3;
    join
endmodule
```

The waveforms in Fig. 4.6 show that in the program, `arbif.cb.grant` gets the value from just before the clock edge. When the interface input changes right at a clock edge, such as 250ns, the value does not propagate to the testbench until the next cycle, which starts at 350ns.

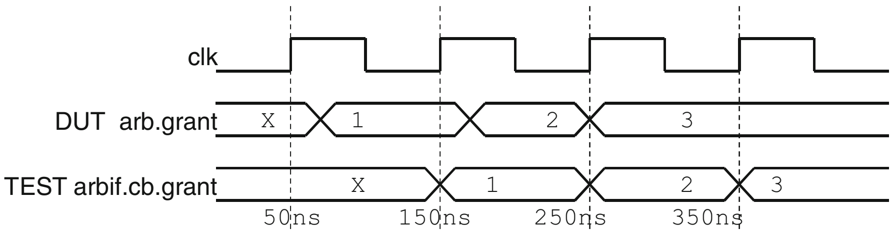


Fig. 4.6 Sampling a synchronous interface

4.4.3 *Interface Signal Drive*

Sample 4.21 has an abbreviated version of the arbiter test program, which uses the arbiter interface defined in Sample 4.16.

Sample 4.21 Testbench using interface with clocking block

```
program automatic test_with_cb (arb_if.TEST arbif);

  initial begin
    @arbif.cb;
    arbif.cb.request <= 2'b01;
    $display("@%0t: Drove req=01", $time);
    repeat (2) @arbif.cb;
    if (arbif.cb.grant == 2'b01)
      $display("@%0t: Success: grant == 2'b01", $time);
    else
      $display("@%0t: Error: grant != 2'b01", $time);
  end

endprogram : test_with_cb
```

When using modports with clocking blocks, a synchronous interface signal such as `request` must be prefixed with both the interface name, `arbif`, and the clocking block name, `cb`. So in Sample 4.21, `arbif.cb.request` is legal, but `arbif.request` is not. This is the most common coding mistake with interfaces and clocking blocks.

4.4.4 *Driving Interface Signals Through a Clocking Block*

You should always drive interface signals in a clocking block with a synchronous drive using a nonblocking assignment. This is because the design signal does not change immediately after your assignment – remember that your testbench executes in the Reactive region while design code is in the Active region. If your testbench drives `arbif.cb.request` at 100ns, the same time as `arbif.cb` (which is `@(posedge clk)` according to the clocking block), `request` changes in the design at 100ns. However, if your testbench tries to drive `arbif.cb.request` at time 101ns, between clock edges, the change does not propagate until the next clock edge. In this way, your drives are always synchronous. In Sample 4.20, `arbif.grant` is driven by a module and can use a blocking assignment.

If the testbench drives the synchronous interface signal at the active edge of the clock, as shown in Sample 4.22, the value propagates immediately to the design. This is because the default output delay is `#0` for a clocking block. If the testbench drives the output just after the active edge, the value is not seen in the design until the next active edge of the clock.

Sample 4.22 Interface signal drive

```
busif.cb.request <= 1;      // Synchronous drive
busif.cb.cmd <= cmd_buf;    // Synchronous drive
```

Sample 4.23 shows what happens if you drive a synchronous interface signal at various points during a clock cycle. This uses the interface from Sample 4.16 and the top module and clock generator from Sample 4.9.

Sample 4.23 Driving a synchronous interface

```
program automatic test_with_cb(arb_if.TEST arbif);
  initial fork
    # 70ns arbif.cb.request <= 3;
    #170ns arbif.cb.request <= 2;
    #250ns arbif.cb.request <= 1;
    #500ns finish;
  join
endprogram

module arb(arb_if.DUT arbif);
  initial
    $monitor("@%0t: req=%h", $time, arbif.request);
endmodule
```

Note that in Fig. 4.7, the value 3, driven in the middle of the first cycle, is seen by the DUT at the start of the second cycle. The value 2 is driven in the middle of the second cycle. It is never seen by the DUT as the testbench drives a 1 at the end of the second cycle.

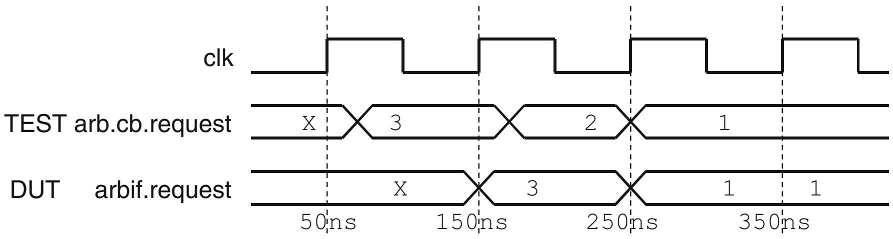


Fig. 4.7 Driving a synchronous interface

Driving clocking block signals asynchronously can lead to dropped values. Instead, drive at the clock edge by using a cycle delay prefix on your drives as shown in Sample 4.24.

Sample 4.24 Interface signal drive

```
##2 arbif.cb.request <= 0;  // Wait 2 cycles then assign
##3;         // Illegal - must be used with an assignment
```

If you want to wait for two clock cycles before driving a signal, you can either use "`repeat (2) @arbif.cb;`" or use the cycle delay `##2`. This latter delay only works as a prefix to a drive of a signal in a clocking block, as it needs to know which clock to use for the delay.

The cycle delay of `##0` in an assignment that drives the value immediately if the clock was asserted in this time slot, according to the clocking block. If the clock was not just asserted, the signal is driven at the next active edge of the clock. The cycle delay of `##1` always waits for the next active edge of the clock, even if the clock was asserted in the current time slot.

The naked cycle delay statement `##3`; works if you have a default clocking block for your program or module. This book only recommends putting a clocking block in an interface and not creating a default clocking block. You should always be specific about which clock is being referenced.

4.4.5 Bidirectional Signals in the Interface

In Verilog-1995, if you want to drive a bidirectional signal such as a port from procedural code, you need a continuous assignment to connect the `reg` to the `wire`. In SystemVerilog, synchronous bidirectional signals in interfaces are easier to use as the continuous assignment is added for you, as shown in Sample 4.25. When you write to the net from a program, SystemVerilog actually writes to a temporary variable that drives the net. Your program reads directly from the wire, seeing the value that is resolved from all the drivers. Design code in a module still uses the classic register plus continuous assignment statement.

Sample 4.25 Bidirectional signals in a program and interface

```
interface bidir_if (input bit clk);
  wire [7:0] data;  // Bidirectional signal
  clocking cb @(posedge clk);
    inout data;
  endclocking
  modport TEST (clocking cb);
endinterface

program automatic test(bidir_if.TEST mif);
  initial begin
    mif.cb.data <= 'z;          // Tri-state the bus
    @mif.cb;
    $displayh(mif.cb.data);     // Read from the bus
    @mif.cb;
    mif.cb.data <= 7'h5a;       // Drive the bus
    @mif.cb;
    $displayh(mif.cb.data);     // Read from the bus
    mif.cb.data <= 'z;          // Release the bus
  end
endprogram
```

The SystemVerilog LRM is not clear on driving an asynchronous bidirectional signal using an interface. Two possible solutions are to use a cross-module reference and continuous assignment or to use a virtual interface as shown in Chapter 10.

4.4.6 Specifying Delays in Clocking Blocks

A clocking block ensures that your signals are driven and sampled at the specified clock edge. You can skew these times with either a `default` statement, or by specifying the delays for individual signals. This can be useful when simulating netlists with real delays. Sample 4.26 shows a clocking block with a default statement that has the skews for all signals. In this example, the inputs are sampled 15ns before the posedge of the clock and the outputs are driven 10ns after the posedge of the clock.

Sample 4.26 Clocking block with default statement

```
clocking cb @(posedge clk);
  default input #15ns output #10ns;
  output request;
  input grant;
endclocking
```

Sample 4.27 shows the equivalent clocking block, but with the delays specified on the individual signals.

Sample 4.27 Clocking block with delays on individual signals

```
clocking cb @(posedge clk);
  output #10ns request;
  input  #15ns grant;
endclocking
```

4.5 Program Block Considerations

4.5.1 The End of Simulation

In Verilog, simulation continues while there are scheduled events, or until a `$finish` is executed. SystemVerilog adds an additional way to end simulation. A program block is treated as if it contains a test. If there is only a single program, simulation ends when you complete the last statement in every `initial` block in the program, as this is considered the end of the test. Simulation ends even if there are threads still running in the program or modules. As a result, you don't have to shut down every monitor and driver when a test is done.

If there are several program blocks, simulation ends when the last program completes. This way simulation ends when the last test completes. You can terminate any program block early by executing `$exit`. Of course you can still explicitly call `$finish` to end simulation, but this might cause issues if you have multiple programs.

However, simulation is not yet over. A module or program can have a `final` block that contains code to be run just before the simulator terminates, as shown in Sample 4.28. This is a great place to perform clean up work such as closing files, and printing a report of the number of errors and warnings encountered. You cannot schedule any events, or have any delays in a `final` block that could cause time to elapse. You do not have to worry about freeing any memory that was allocated as this will be done automatically.

Sample 4.28 A `final` block

```
program automatic test;
  int errors, warnings;

  initial begin
    ... // Main program activity
  end

  final
    $display("Test completed with %0d errors and %0d warnings",
             errors, warnings);
endprogram
```

4.5.2 *Why are* `Always` *Blocks not Allowed in a Program?*

In SystemVerilog you can put `initial` blocks in a program, but not `always` blocks. This may seem odd if you are used to Verilog modules, but there are several reasons SystemVerilog programs are closer to a program in C, with one (or more) entry points, than Verilog's many small blocks of concurrently executing hardware. In a design, an `always` block might trigger on every positive edge of a clock from the start of simulation. In contrast, a testbench has the steps of initialization, stimulate and respond to the design, and then wrap up simulation. An `always` block that runs continuously would not work.

When the last `initial` block completes in the program, simulation implicitly ends just as if you had executed `$finish`. If you had an `always` block, it would run for ever, so you would have to explicitly call `$exit` to signal that the program completed. But don't despair. If you really need an `always` block, you can use `initial forever` to accomplish the same thing.

4.5.3 *The Clock Generator*

Now that you have seen the program block, you may wonder if the clock generator should be in a module. The clock is more closely tied to the design than the testbench, and so the clock generator should remain in a module. The generator should be instantiated at the same level as the DUT so it can drive both the DUT and testbench As you refine the design, you create clock trees, and you have to carefully control the skews as the clocks enter the system and propagate through the blocks.

The testbench is much less picky. It just wants a clock edge to know when to drive and sample signals. Functional verification is concerned with providing the right values at the right cycle, not with fractional nanosecond delays and relative clock skews.

The program block is not the place to put a clock generator. Sample 4.29 tries to put the generator in a program block but just causes a race condition. The `clk` and `data` signals both propagate from the Reactive region to the design in the Active region and could cause a race condition depending on which one arrived first.

Sample 4.29 Bad clock generator in program block

```
program automatic bad_generator (output bit clk, data);
  initial
    forever #5 clk <= ~clk ;

  initial
    forever @(posedge clk)
      data <= ~data;
endprogram
```

Avoid race conditions by always putting the clock generator in a module. If you want to randomize the generator's properties, create a class with random variables for skew, frequency, and other characteristics, as shown in Chapter 6. You can use this class in the generator module, or in the testbench.

Sample 4.30 shows a good clock generator in a module. It deliberately avoids an edge at time 0 to prevent race conditions. All clock edges are generated with a blocking assignment to trigger events during the Active region. If you must generate a clock edge at time 0, use a nonblocking assignment to set the initial value so all clock sensitive logic such as `always` blocks will have started before the clock changes value.

Sample 4.30 Good clock generator in module

```
module clock_generator (output bit clk);
  bit local_clk = 0;
  assign clk = local_clk;  // Drive port from local signal
  always #50 local_clk = ~local_clk;
endmodule
```

Lastly, don't try to verify the low-level timing with functional verification. The testbenches described in this book check the behavior of the DUT but not the timing, which is better done with a static timing analysis tool. Your testbenches should be flexible enough to be compatible with gate-level simulations run with back-annotated timing.

4.6 Connecting It All Together

Now you have a design described in a module, a testbench in a program block, and interfaces that connect them together. Sample 4.31 has the top-level module that instantiates and connects all the pieces.

Sample 4.31 `Top` module with implicit port connections

```
module top;
  bit  clk;
  always #50 clk = ~clk;

  arb_if arbif(.*);      // ... arbif(clk)  From Sample 4-4
  arb_with_ifc  a1(.*); // ... a1(arbif)   From Sample 4-5
  test_with_ifc t1(.*); // ... t1(arbif)   From Sample 4-6
endmodule : top
```

This is almost identical to Sample 4.7. It uses a shortcut notation.* (implicit port connection) that automatically connects module instance ports to signals at the current level if they have the same name and data type.

4.6.1 An Interface in a Port List Must Be Connected

The SystemVerilog compiler won't let you compile a single module or program that uses an interface in the port list. Why not? After all, a module or program with ports made of individual signals can be compiled without being instantiated, as shown in Sample 4.32.

Sample 4.32 Module with just port connections

```
module uses_a_port(inout bit not_connected);
  ...
endmodule
```

The compiler creates wires and connects them to the dangling signals. However, a module or program with an interface in its port list must be connected to an instance of the interface.

Sample 4.33 Module with an interface

```
// This will not compile without interface declaration
module uses_an_interface(arb_if.DUT arbif);
  initial arbif.grant = 0;
endmodule
```

For Sample 4.33, the compiler is not able to build even a simple interface. If you have modports or a program block using clocking blocks in an interface, the compiler has an even more difficult time. Even if you are just looking to wring out syntax bugs, you must complete the connections. This can be done as shown in Sample 4.34.

Sample 4.34 Top module connecting DUT and interface

```
module top;
  bit clk;
  always #50 clk = !clk;

  arb_if arbif(clk);            // Interface with modport
  uses_an_interface u1(arbif);  //    needed to compile this
endmodule
```

4.7 Top-Level Scope

Sometimes you need to create things in your simulation that are outside of a program or module so that they are seen by all blocks. In Verilog, only macros extend across module boundaries, and so are used for creating global constants. SystemVerilog introduces the *compilation unit*, that is a group of source files that are compiled together. The scope outside the boundaries of any `module`, `macromodule`, `interface`, `program`, `package`, or `primitive` is known as the *compilation-unit scope*, also referred to as `$unit`. Anything such as a parameter defined in this scope is similar to a global because it can be seen by all lower-level blocks. However, it is not truly global as the `parameter` cannot be seen during compilation of other files.

This leads to some confusion. Some simulators compile all the SystemVerilog code together, so `$unit` is global. Other simulators and synthesis tools compile a single module or group of modules at a time, so `$unit` may be just the contents of one or a few files. As a result, `$unit` is not portable. Packages allow you to have code outside of a program or module while eliminating the requirement of compiling all the modules at the same time.

This book calls the scope outside blocks the "top-level scope." You can define variables, parameters, data types and even routines in this space. Sample 4.35 declares a top-level parameter, `TIMEOUT`, that can be used anywhere in the hierarchy. This example also has a `const` string that holds an error message. You can declare top-level constants either way.

Sample 4.35 Top-level scope for arbiter design

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

program automatic test;
  ...
  initial begin
    #TIMEOUT;
    $display("%s", time_out_msg);
    $finish;
    end
endprogram
```

The instance name `$root` allows you to unambiguously refer to names in the system, starting with the top-level scope. In this respect, `$root` is similar to "/" in the Unix file system. For tools such as VCS that compile all files at once, `$root` and `$unit` are equivalent. The name `$root` also solves an old Verilog problem.

When your code refers to a name in another module, such as i1.var, the compiler first looks in the local scope, then looks up to the next higher scope, and so on until it reaches the top. You may have wanted to use i1.var in the top module, but an instance named i1 in an intermediate scope may have sidetracked the search, giving you the wrong variable. You use $root to make unambiguous cross module references by specifying the absolute path.

Sample 4.36 shows a program that is instantiated in the module top that is implicitly instantiated in the top-level scope. The program can use a relative or absolute reference to the clk signal in the module. You can use a macro to hold the hierarchical path so that when the path changes, you only have to change one piece of code. The LRM does not allow modules to be explicitly instantiated in the top-level scope.

Sample 4.36 Cross-module references with $root

```
module top;
  bit clk;
  test t1(.*);
endmodule

`define TOP $root.top
program automatic test;
  initial begin
    // Absolute reference
    $display("clk=%b", $root.top.clk);
    $display("clk=%b", `TOP.clk);    // With macro

    // Relative reference
    $display("clk=%b", top.clk);
    end
endprogram
```

4.8 Program–Module Interactions

The program block can read and write all signals in modules, and can call routines in modules, but a module has no visibility into a program. This is because your testbench needs to see and control the design, but the design should not depend on anything in the testbench.

A program can call a routine in a module to perform various actions. The routine can set values on internal signals, also known as “backdoor load.” Next, because the current SystemVerilog standard does not define how to force signals from a program block, you need to write a task in the design to do the force, and then call it from the program.

Lastly, it is a good practice for your testbench to use a function to get information from the DUT. Reading signal values can work most of the time, but if the design code changes, your testbench may interpret the values incorrectly. A function in the module can encapsulate the communication between the two and make it easier for your testbench to stay synchronized with the design. Chapter 10 shows how to embed functions and SystemVerilog Assertions in an interface.

4.9 SystemVerilog Assertions

You can create temporal assertions about signals in your design to check their behavior and temporal relationship with SystemVerilog Assertions (SVA). The simulator keeps track of what assertions have triggered, so you can gather functional coverage data on them.

4.9.1 Immediate Assertions

An immediate assertion checks if an expression is true when the statement is executed. Your testbench procedural code can check the values of design signals and testbench variables and take action if there is a problem. For example, if you have asserted the bus request, you expect that grant will be asserted two cycles later. You could use an `if`-statement as shown in Sample 4.37.

Sample 4.37 Checking a signal with an `if`-statement

```
arbif.cb.request <= 2'b01;
repeat (2) @arbif.cb;
if (arbif.cb.grant != 2'b01)
  $display("Error, grant != 2'b01");
```

An assertion is more compact than an `if`-statement. However, note that the logic is reversed compared to the `if`-statement above. You want the expression inside the parentheses to be true; otherwise, print an error as shown in Sample 4.38.

Sample 4.38 Simple immediate assertion

```
arbif.cb.request <= 2'b01;
repeat (2) @arbif.cb;
a1: assert (arbif.cb.grant == 2'b01);
```

If the `grant` signal is asserted correctly, the test continues. If the signal does not have the expected value, the simulator produces a message similar Sample 4.39.

Sample 4.39 Error from failed immediate assertion

```
"test.sv", 7: top.t1.a1: started at 55ns failed at 55ns
Offending '(arbif.cb.grant == 2'b1)'
```

This says that on line 7 of the file `test.sv`, the assertion `top.t1.a1` started at 55ns to check the signal `arbif.cb.grant`, but failed immediately. The label `a1` should be unique so that you can quickly locate the failing assertion.

You may be tempted to use the full SystemVerilog Assertion syntax to check an elaborate sequence over a range of time, but use care as they can be hard to debug. Assertions are declarative code, and execute very differently than the surrounding procedural code. In just a few lines of assertions, you can verify temporal relations; the equivalent procedural code would be more complicated and verbose, but easier for the next person to understand when they have to read your code.

If you are a VHDL programmer, you may be tempted at this point to start sprinkling immediate assertions across your code. Resist the temptation! Your code will work correctly for weeks or months until someone decides to improve simulation performance by disabling assertions. The simulator will no longer execute the expression in the assertion. If the expression has a side effect such as incrementing a value or calling a function, it will no longer occur.

4.9.2 Customizing the Assertion Actions

An immediate assertion has optional then- and else-clauses. If you want to augment the default message, you can add your own as shown in Sample 4.40.

Sample 4.40 Creating a custom error message in an immediate assertion

```
a40: assert (arbif.cb.grant == 2'b01)
else $error("Grant not asserted");
```

If `grant` does not have the expected value, you'll see an error message similar to Sample 4.41.

Sample 4.41 Error from failed immediate assertion

```
"test.sv", 7: top.t1.a40: started at 55ns failed at 55ns
Offending '(arbif.cb.grant == 2'b01)'
Error: "test.sv", 7: top.t1.a40: at time 55 ns
Grant not asserted
```

SystemVerilog has four functions to print messages: `$info`, `$warning`, `$error`, and `$fatal`. These are allowed only inside an assertion, not in procedural code, though future versions of SystemVerilog may allow this.

You can use the `then`-clause to record when an assertion completed successfully as shown in Sample 4.42.

Sample 4.42 Creating a custom error message

```
a42: assert (arbif.cb.grant == 2'b01)
  grants_received++;        // Another succesful result!
else
  $error("Grant not asserted");
```

4.9.3 Concurrent Assertions

The other type of assertion is the concurrent assertion that you can think of as a small model that runs continuously, checking the values of signals for the entire simulation. These are instantiated similarly to other design blocks and are active for the entire simulation. You need to specify a sampling clock in the assertion. Sample 4.43 has a small assertion to check that the arbiter request signal does not have X or Z values except during reset. This code is placed outside of procedural blocks such as `initial` and `always`. Sample 4.43 is for illustration only. See one of the books listed below for a more information.

Sample 4.43 Concurrent assertion to check for X/Z

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

  property request_2state;
    @(posedge clk) disable iff (rst)
    $isunknown(request) == 0;       // Make sure no Z or X found
  endproperty
  assert_request_2state: assert property (request_2state);

endinterface
```

4.9.4 Exploring Assertions

There are many other uses for assertions. For example, you can put assertions in an interface. Now your interface not only transmits signal values but also checks the protocol.

This Section has provided a brief introduction to SystemVerilog Assertions. For more information, see Vijayaraghhavan and Ramanathan (2005) and Haque et al. (2007).

4.10 The Four-Port ATM Router

The arbiter example is a good introduction to interfaces, but real designs have more than a single input and output. This section discusses a four-port ATM (Asynchronous Transfer Mode) router, shown in Fig. 4.8.

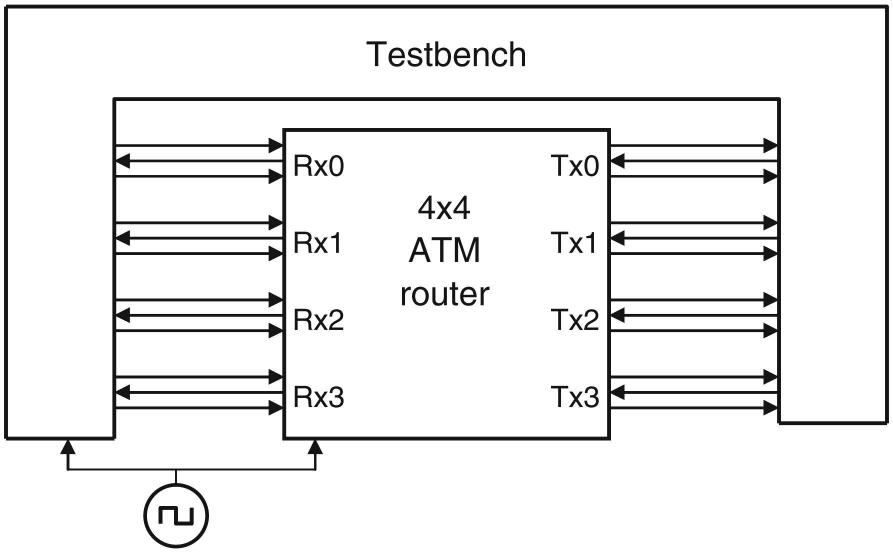


Fig. 4.8 Testbench – ATM router diagram without interfaces

4.10.1 ATM Router with Ports

The following code fragments show the tangle of wires you would have to endure to connect an RTL block to a testbench. First is the header for the ATM router model. This uses the Verilog-1995 style port declarations, where the type and direction are separate from the header.

The actual code for the router in Sample 4.44 is crowded out by nearly a page of port declarations.

Sample 4.44 ATM router model header with ports

```
module atm_router_ports(
  // 4 x Level 1 Utopia ATM layer Rx Interfaces
     Rx_clk_0,  Rx_clk_1,  Rx_clk_2,  Rx_clk_3,
     Rx_data_0, Rx_data_1, Rx_data_2, Rx_data_3,
     Rx_soc_0,  Rx_soc_1,  Rx_soc_2,  Rx_soc_3,
     Rx_en_0,   Rx_en_1,   Rx_en_2,   Rx_en_3,
     Rx_clav_0, Rx_clav_1, Rx_clav_2, Rx_clav_3,

  // 4 x Level 1 Utopia ATM layer Tx Interfaces
     Tx_clk_0,   Tx_clk_1,   Tx_clk_2,   Tx_clk_3,
     Tx_data_0,  Tx_data_1,  Tx_data_2,  Tx_data_3,
     Tx_soc_0,   Tx_soc_1,   Tx_soc_2,   Tx_soc_3,
     Tx_en_0,    Tx_en_1,    Tx_en_2,    Tx_en_3,
     Tx_clav_0,  Tx_clav_1,  Tx_clav_2,  Tx_clav_3,

  // Miscellaneous control interfaces
     rst, clk);

// 4 x Level 1 Utopia Rx Interfaces
  output       Rx_clk_0, Rx_clk_1, Rx_clk_2, Rx_clk_3;
  input [7:0]  Rx_data_0,Rx_data_1,Rx_data_2,Rx_data_3;
  input        Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3;
  output       Rx_en_0,  Rx_en_1,  Rx_en_2,  Rx_en_3;
  input        Rx_clav_0,Rx_clav_1,Rx_clav_2,Rx_clav_3;

// 4 x Level 1 Utopia Tx Interfaces
  output       Tx_clk_0, Tx_clk_1, Tx_clk_2, Tx_clk_3;
  output [7:0] Tx_data_0,Tx_data_1,Tx_data_2,Tx_data_3;
  output       Tx_soc_0, Tx_soc_1, Tx_soc_2, Tx_soc_3;
  output       Tx_en_0,  Tx_en_1,  Tx_en_2,  Tx_en_3;
  input        Tx_clav_0,Tx_clav_1,Tx_clav_2,Tx_clav_3;

// Miscellaneous control interfaces
  input        rst, clk;
  ...
endmodule
```

So what sort of synthesizable code goes in the "…" at the end of Sample 4.44 ? See Sutherland (2006) for more information and examples of using interfaces in modules and other SystemVerilog design constructs.

4.10.2 ATM Top-Level Module with Ports

Sample 4.45 contains the top-level module.

Sample 4.45 Top-level module without an interface

```
module top;
  bit clk, rst;
  always #5 clk = !clk;
  wire Rx_clk_0,  Rx_clk_1,  Rx_clk_2,  Rx_clk_3,
       Rx_soc_0,  Rx_soc_1,  Rx_soc_2,  Rx_soc_3,
       Rx_en_0,   Rx_en_1,   Rx_en_2,   Rx_en_3,
       Rx_clav_0, Rx_clav_1, Rx_clav_2, Rx_clav_3,
       Tx_clk_0,  Tx_clk_1,  Tx_clk_2,  Tx_clk_3,
       Tx_soc_0,  Tx_soc_1,  Tx_soc_2,  Tx_soc_3,
       Tx_en_0,   Tx_en_1,   Tx_en_2,   Tx_en_3,
       Tx_clav_0, Tx_clav_1, Tx_clav_2, Tx_clav_3;

  wire [7:0] Rx_data_0, Rx_data_1, Rx_data_2, Rx_data_3,
             Tx_data_0, Tx_data_1, Tx_data_2, Tx_data_3;

  atm_router_ports
             a1(Rx_clk_0, Rx_clk_1, Rx_clk_2, Rx_clk_3,
                Rx_data_0,Rx_data_1,Rx_data_2,Rx_data_3,
                Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3,
                Rx_en_0,  Rx_en_1,  Rx_en_2,  Rx_en_3,
                Rx_clav_0,Rx_clav_1,Rx_clav_2,Rx_clav_3,
                Tx_clk_0, Tx_clk_1, Tx_clk_2, Tx_clk_3,
                Tx_data_0,Tx_data_1,Tx_data_2,Tx_data_3,
                Tx_soc_0, Tx_soc_1, Tx_soc_2, Tx_soc_3,
                Tx_en_0,  Tx_en_1,  Tx_en_2,  Tx_en_3,
                Tx_clav_0,Tx_clav_1,Tx_clav_2,Tx_clav_3,
                rst, clk);

  test_ports t1 (Rx_clk_0, Rx_clk_1, Rx_clk_2, Rx_clk_3,
                Rx_data_0,Rx_data_1,Rx_data_2,Rx_data_3,
                Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3,
                Rx_en_0,  Rx_en_1,  Rx_en_2,  Rx_en_3,
                Rx_clav_0,Rx_clav_1,Rx_clav_2,Rx_clav_3,
                Tx_clk_0, Tx_clk_1, Tx_clk_2, Tx_clk_3,
                Tx_data_0,Tx_data_1,Tx_data_2,Tx_data_3,
                Tx_soc_0, Tx_soc_1, Tx_soc_2, Tx_soc_3,
                Tx_en_0,  Tx_en_1,  Tx_en_2,  Tx_en_3,
                Tx_clav_0,Tx_clav_1,Tx_clav_2,Tx_clav_3,
                rst, clk);
endmodule
```

Sample 4.46 shows the top of the testbench module. Once again, note that the ports and wires take up the majority of the module.

Sample 4.46 Verilog-1995 testbench using ports

```
module test_ports(
    // 4 x Level 1 Utopia ATM layer Rx Interfaces
    Rx_clk_0,  Rx_clk_1,  Rx_clk_2,  Rx_clk_3,
    Rx_data_0, Rx_data_1, Rx_data_2, Rx_data_3,
    Rx_soc_0,  Rx_soc_1,  Rx_soc_2,  Rx_soc_3,
    Rx_en_0,   Rx_en_1,  Rx_en_2,   Rx_en_3,
    Rx_clav_0, Rx_clav_1, Rx_clav_2, Rx_clav_3,

    // 4 x Level 1 Utopia ATM layer Tx Interfaces
    Tx_clk_0,  Tx_clk_1,  Tx_clk_2,  Tx_clk_3,
    Tx_data_0, Tx_data_1, Tx_data_2, Tx_data_3,
    Tx_soc_0,  Tx_soc_1,  Tx_soc_2,  Tx_soc_3,
    Tx_en_0,   Tx_en_1,  Tx_en_2,   Tx_en_3,
    Tx_clav_0, Tx_clav_1, Tx_clav_2, Tx_clav_3,

    rst, clk);    // Miscellaneous control signals

// 4 x Level 1 Utopia Rx Interfaces
  input        Rx_clk_0, Rx_clk_1, Rx_clk_2, Rx_clk_3;
  output [7:0] Rx_data_0,Rx_data_1,Rx_data_2,Rx_data_3;
  reg  [7:0]   Rx_data_0,Rx_data_1,Rx_data_2,Rx_data_3;
  output       Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3;
  reg          Rx_soc_0, Rx_soc_1, Rx_soc_2, Rx_soc_3;
  input        Rx_en_0,  Rx_en_1,  Rx_en_2,  Rx_en_3;
  output       Rx_clav_0,Rx_clav_1,Rx_clav_2,Rx_clav_3;
  reg          Rx_clav_0,Rx_clav_1,Rx_clav_2,Rx_clav_3;

// 4 x Level 1 Utopia Tx Interfaces
  input       Tx_clk_0,  Tx_clk_1, Tx_clk_2, Tx_clk_3;
  input [7:0] Tx_data_0, Tx_data_1,Tx_data_2,Tx_data_3;
  input       Tx_soc_0,  Tx_soc_1, Tx_soc_2, Tx_soc_3;
  input       Tx_en_0,  Tx_en_1,  Tx_en_2,   Tx_en_3;
  output      Tx_clav_0, Tx_clav_1,Tx_clav_2,Tx_clav_3;
  reg         Tx_clav_0, Tx_clav_1,Tx_clav_2,Tx_clav_3;

// Miscellaneous control interfaces
  output       rst;
  reg          rst;
  input        clk;

  initial begin
    // Reset the device
    rst <= 1;
    Rx_data_0 <= 0;
    ...
    end
endmodule
```

You just saw three pages of code, and it was all just connectivity — no testbench, no design! Interfaces provide a better way to organize all this information and eliminate the repetitive parts that are so error prone.

4.10.3 Using Interfaces to Simplify Connections

Figure 4.9 shows the ATM router connected to the testbench, with the signals grouped into interfaces.

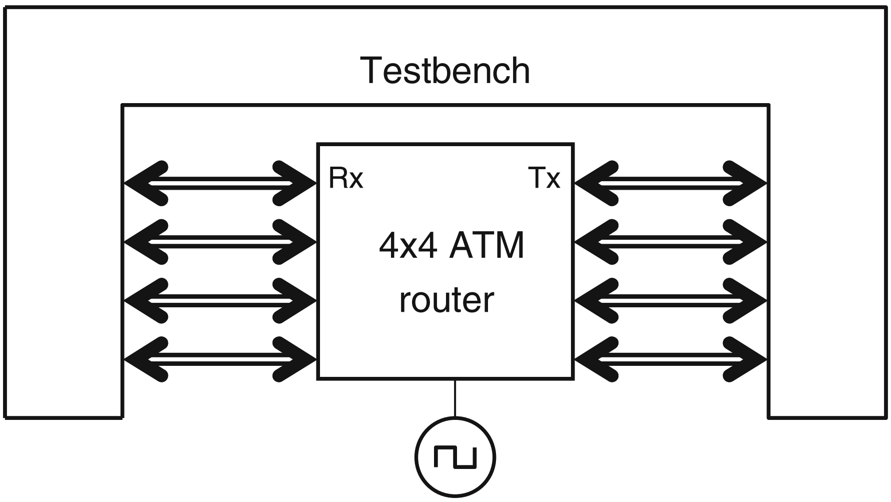


Fig. 4.9 Testbench - router diagram with interfaces

4.10.4 ATM Interfaces

Sample 4.47 and 4.48 show the `Rx` and `Tx` interfaces with modports and clocking blocks.

Sample 4.47 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 DUT (output en, rclk,
               input  data, soc, clav);

  modport TB (clocking cb);
endinterface : Rx_if
```

Sample 4.48 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 DUT (output data, soc, en, tclk,
               input  clk, clav);

  modport TB (clocking cb);
endinterface : Tx_if
```

4.10.5 ATM Router Model Using an Interface

Sample 4.49 contains the ATM router model and testbench, which need to specify the `modport` in their port connection list. Note that you put the `modport` name after the interface name, `Rx_if`.

Sample 4.49 ATM router model with interface using modports

```
module atm_router(Rx_if.DUT Rx0, Rx1, Rx2, Rx3,
                  Tx_if.DUT Tx0, Tx1, Tx2, Tx3,
                  input logic clk, rst);
  ...
endmodule
```

4.10.6 ATM Top Level Module with Interfaces

The top module, shown in Sample 4.50, has shrunk considerably, along with the chances of making a mistake.

Sample 4.50 Top-level module with interface

```
module top;
  bit clk, rst;
  always #5 clk = !clk;

  Rx_if Rx0 (clk), Rx1 (clk), Rx2 (clk), Rx3 (clk);
  Tx_if Tx0 (clk), Tx1 (clk), Tx2 (clk), Tx3 (clk);

  atm_router a1 (Rx0, Rx1, Rx2, Rx3,             // or just (.*)
                 Tx0, Tx1, Tx2, Tx3, clk, rst);

  test       t1 (Rx0, Rx1, Rx2, Rx3,             // or just (.*)
                 Tx0, Tx1, Tx2, Tx3, clk, rst);
endmodule : top
```

4.10.7 ATM Testbench with Interface

Sample 4.51 shows the part of the testbench that captures cells coming in from the TX port of the router. Note that the interface names are hard-coded, so you have to duplicate the same code four times for the 4×4 ATM router. For example, only the task `receive_cell0` is shown, and the final code would also have `receive_cell1`, `receive_cell2`, and `receive_cell30`. Chapter 10 shows how to simplify the code by using virtual interfaces.

Sample 4.51 Testbench using an interface with a clocking block

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

  bit [7:0] bytes[ATM_CELL_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

    foreach (bytes[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 : receive_cell0

endprogram : test
```

4.11 The Ref Port Direction

SystemVerilog introduces a new port direction for connecting modules: `ref`. You should be familiar with the `input`, `output`, and `inout` directions. The last is for modeling bidirectional connections. If you drive a signal with multiple `inout` ports, SystemVerilog will calculate the value of the signal by combining the values of all drivers, taking in to account driver strengths and Z values.

A `ref` port is a different beast. It is essentially a way to make two names that both reference the same variable. There is only one storage location, but multiple aliases. Ref ports can only connect to variables, not signals. See Section 3.4.3 for information on the `ref` direction for routine arguments.

In Sample 4.52, the `incr` module has two ref ports, `c` and `d`. These two variables share storage with the `c` and `d` variables in the top module. When `top` changes the value of `c`, it is seen immediately by `incr`. Then `incr` increments `c` and the result is seen back in the `top` module. If the port `c` was declared as `inout`, you would have had to build tristate drivers such as continuous assignment statements, and make sure you properly drove an enable signal and Z values. Don't consider `ref` ports as a convenient replacement for `inout` ports as only the latter are supported for synthesis.

Sample 4.52 Ref ports

```
module incr(ref int c, d);  // c variable is read and written
  always @(c)
    #1 d = c++; // d=c; c=c+1;
endmodule

module top;
  int c, d;
  incr i1(c, d);
  initial begin
    $monitor("@%0d: c=%0d, d=%0d", $time, c, d);
    c = 2;
    #10;
    c = 8;
    #10;
  end
endmodule
```

4.12 Conclusion

In this chapter you have learned how to use SystemVerilog's interfaces to organize the communication between design blocks and your testbench. With this design construct, you can replace dozens of signal connections with a single interface, making your code easier to maintain and improve, and reducing the number of wiring mistakes.

SystemVerilog also introduces the program block to hold your testbench and to reduce race conditions between the device under test and the testbench. With a clocking block in an interface, your testbenches will drive and sample design signals correctly relative to the clock.

4.13 Exercises

1. Design an interface and testbench for the ARM Advanced High-performance Bus (AHB). You are provided a bus master as verification IP that can initiate AHB transactions. You are testing a slave design. The testbench instantiates the interface, slave, and master. Your interface will display an error if the transaction type is not IDLE or NONSEQ on the negative edge of HCLK. The AHB signals are described in Table 4.2.

Table 4.2 AHB Signal Description

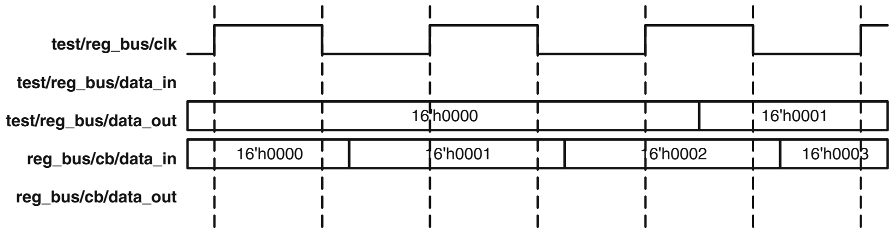
Signal	*Width*	*Direction*	*Description*
HCLK	1	Output	Clock
HADDR	21	Output	Address
HWRITE	1	Output	Write flag: 1=write, 0=read
HTRANS	2	Output	Transaction type: 2′b00=IDLE, 2′b10=NONSEQ
HWDATA	8	Output	Write data
HRDATA	8	Input	Read data

2. For the following interface, add the following code.

   a. A clocking block that is sensitive to the negative edge of the clock, and all I/O that are synchronous to the clock.
   b. A modport for the testbench called `master`, and a modport for the DUT called `slave`
   c. Use the clocking block in the I/O list for the `master` modport.

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

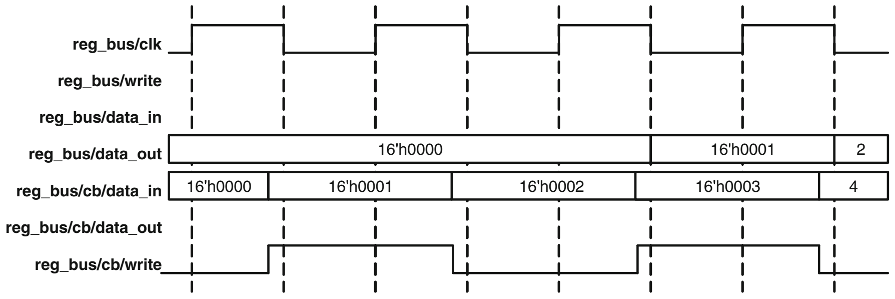
3. For the clocking block in Exercise 2, fill in the `data_in` and `data_out` signals in the following timing diagram.



4. Modify the clocking block in Exercise 2 to have:

   a. output skew of 25ns for output write and address
   b. input skew of 15ns
   c. restrict `data_in` to only change on the positive edge of the clock

5. For the clocking block in Exercise 4, fill in the following timing diagram, assuming a clock period of 100ns.


reg_bus/clk
reg_bus/write
reg_bus/data_in
reg_bus/data_out
16'h0000
16'h0001
2
reg_bus/cb/data_in
16'h0000
16'h0001
16'h0002
16'h0003
4
reg_bus/cb/data_out
reg_bus/cb/write