Introduction

Verilog-XL is a simulator that allows you to test the logic of a design. The process of logic simulation in Verilog-XL is as follows:

1.   Describe the design to Verilog-XL.
2.   Tell Verilog-XL how to apply stimuli to the design.
3.   Tell Verilog-XL how to report its results.

A description of a design is called a model. In Verilog-XL, you write models in the Verilog Hardware Description Language (the Verilog HDL).

You tell Verilog-XL how to apply stimuli to a model of your design and how to report the results by writing a model of a test fixture for your device. Verilog-XL has a unified modeling and command language.

Purpose

The purpose of this tutorial is to show you how to model and simulate a basic design. It shows you how to use the Verilog HDL to model and apply stimuli to a device and tell Verilog-XL how to report the results.

Intended Audience

This tutorial is intended for new users of Verilog-XL who understand digital logic and have experience with a high-level programming language. This tutorial is also intended for users who are evaluating Verilog-XL.

Conventions

This tutorial contains a number of graphics to help you find important information and key concepts. These graphics and their purpose are as follows:

• **Notes** (indicated by a pointing finger)—suggest different ways to cover the material in this tutorial or point out important facts about the concepts discussed.

1

- **Concepts** (in a folder marked CONCEPTS)—define ideas that are essential to your understanding of the Verilog-XL modeling and simulation process.

> **gate-level model** – a model that describes the gates, and the connections between gates, in a design.
> **keyword** – a term that has a special meaning in the Verilog HDL. A keyword tells Verilog-XL that it is beginning to read a statement, declaration or some other kind of language construct. Keywords are the essential parts of language constructs.

- **Comparisons** (indicated by a balance scale) — compare important concepts

Method

This tutorial uses the following methods:

- Present the modeling and simulating processes one step at a time, explaining why you take each step.
- Present concepts as they are needed to perform a step.
- Compare concepts when the comparison is worthwhile.

This tutorial explains how to develop three models. Two of these models describe the design. The third model describes how a test fixture drives the design and reports the simulation results. These models work together in a hierachy. After the explanation of the models, this tutorial explains how to run the simulation of these models.

Each step in the modeling process is an entry in a model of a statement, declaration, or some other language construct or part of a language construct. With each new entry, this tutorial shows the entire model with the new entry in boldface type. The tutorial then explains or illustrates the contents and purpose of each entry.
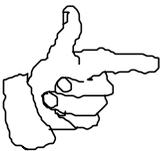
Please note: This tutorial provides an explanation of each entry or step in the modeling and simulation processes. Some entries are very similar to previous entries and you may be able to understand them without reading the explanation. If you understand an entry you can save time by skipping the explanation and moving on to the next entry. If you are not sure about what an entry does or why you made an entry, you can read the explanation that follows it.

The Functional Spec for the Design

This section describes the design that this tutorial models and simulates.

The design is a 4-bit clocked register that operates on both phases of the clock. The design includes a clear line that is active when low. Its components are D-type master-slave flip-flops.Figure 1 shows a schematic for this register.
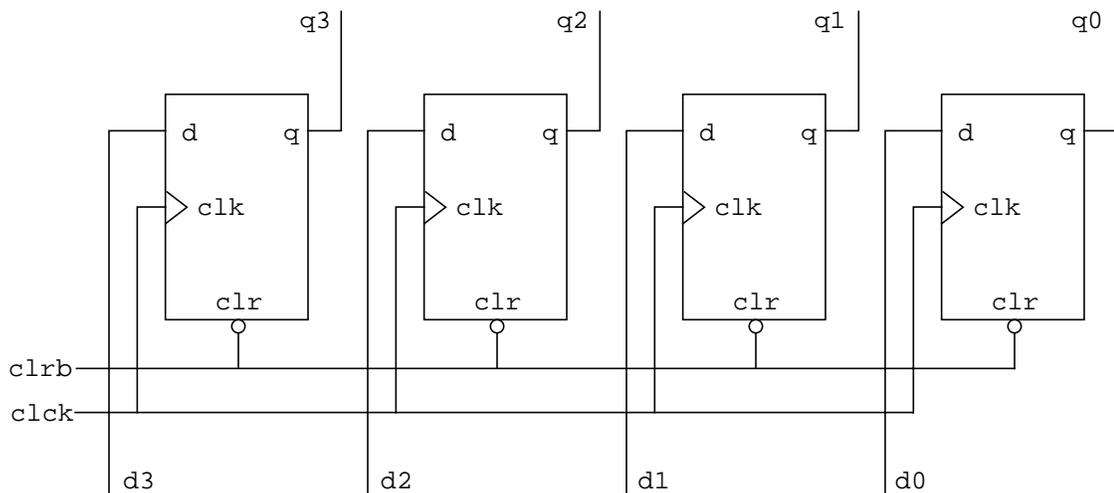

Figure 1:A 4-bit register

The inputs to this register are the following signals:

• a single bit clear line

• a single bit clock line

• a 4-bit data bus

The output from this register is a 4-bit bus.

Figure 2 shows a schematic of the D-type master-slave flip-flop that is the basic component of the register.

3

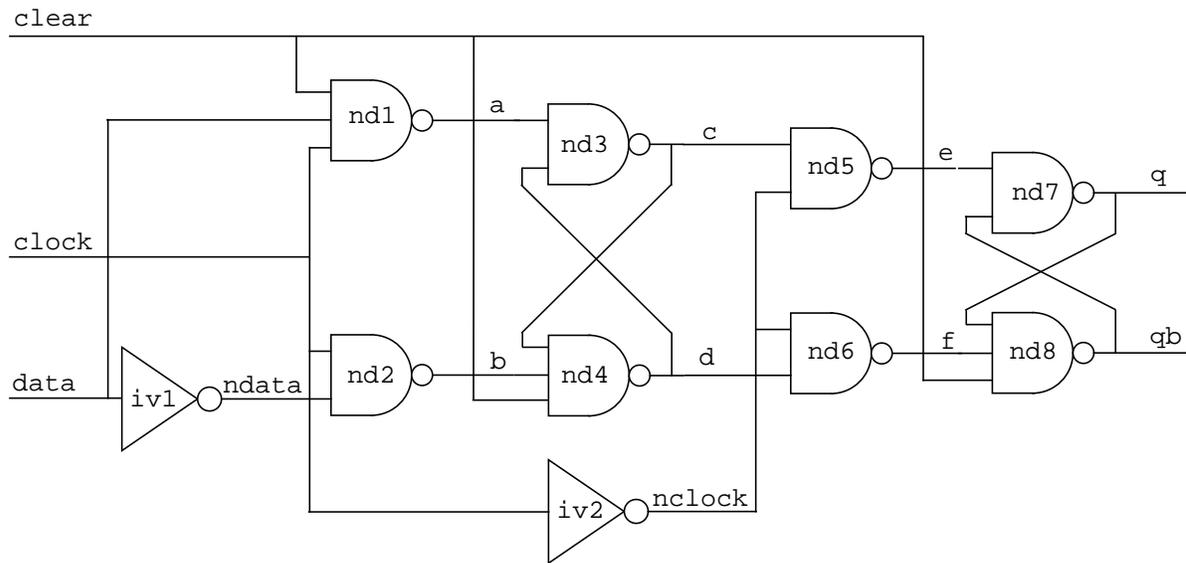*Figure 2: A D-type master-slave flip-flop*

In this D-type master-slave flip-flop, the NAND gates labelled `nd3` and `nd7` have a propagation delay of nine nanoseconds. All other NAND gates and inverters have a propagation delay of 10 nanoseconds.

Writing a Hierarchical Model

This tutorial uses the "bottom-up" approach to a design. It shows you how to write a hierarchical model of this 4-bit register by first modeling the D-type master-slave flip-flop and using copies of that model as building blocks.

Before you begin to write this model you must understand the following concepts:

CONCEPTS

gate-level model – a model that describes the gates and the connections between gates in a design.

keyword – A term that has a special meaning in the Verilog HDL. A keyword tells Verilog-XL that it is beginning to read a statement, declaration or some other kind of language construct. Keywords are the essential parts of language constructs.

logic values – Verilog-XL has four basic logic values:

1   logic 1, high or true condition
0   logic 0, low or false condition
x   unknown value
z   high-impedance state

Verilog-XL reports a logic value of x for most objects in a design that store or transmit values at the start of a simulation, before you apply stimulus. It also reports a logic value of x when it cannot decide between a 1 and a 0.

Verilog-XL reports a logic value of z when an object in a design that transmits a value has no connection through which Verilog-XL can assign or propagate a value of 1, 0 or x to that object.

module – The basic construct that you use to build Verilog HDL models. All the elements of the Verilog language must be contained within a module. All modules contain descriptions of a design or part of a design, or describe how a test fixture for the device drives the device and reports its results.

port – A connection to a module. Logic values propagate into and out of a module through its ports.

scalar – An attribute of a design component that stores or propagates only one bit of data.

vector – An attribute of a design component that can store or propagate more that one bit of data.

identifier – A name that you assign to an object. Identifiers allow you to refer to one object in more than one place in your design. Some of the objects to which you can assign an identifier are modules and gates.

The following comparison may be helpful:

keyword vs. identifier – Keywords are the pre-defined identifying elements of language constructs—they tell Verilog-XL that is has encountered a language construct or how to execute a language construct. Identifiers are the identifying elements of design objects such as modules or gates—you use them to distinguish between design objects. You tell Verilog-XL how to define an identifier.

The Flip-Flop Model

The first model is a gate-level model of a D-type master-slave flip-flop. You write this model as a Verilog HDL module.

1. The first step is to open a text file. Use the commands and text editor that applies to your platform and operating system to open this file. You can assign any name to this file, but for purposes of this tutorial, enter the name `flop.v`.

The Module Header

All modules begin with a *module header.* The following steps show you the entries in a module header.

2. The first entry is the first term in a module.

```
module
```

All module headers begin with the keyword `module`.

3. The next entry is the module identifier.

```
module flop
```

This entry assigns the identifier `flop` to the module. You enter this identifier when you use a copy of this module in another module.

6

4. The next entry is a port connection list.

```
module flop (data,clock,clear,q,qb);
```

All modules that have ports must include a port connection list in their module header.

Port connection lists begin with a left parenthesis and end with a right parenthesis.

This port connection list contains five ports; their names or *identifiers* are as follows:

1. data
2. clock
3. clear
4. q
5. qb

In a port connection list, you separate port names with commas.

The port connection list is the last entry in this module header so the port connection list is followed by a semicolon(;). A semicolon ends the module header.

Port Declarations

The following entries are port declarations. Port declarations tell Verilog-XL the following information about a port:

- the port's type
- the port's size

A module can have three types of ports:

1. input ports
2. output ports
3. inout ports

Logic values propagate into a module through an input port, and propagate out of a module through an output port. Logic values can propagate both into and out of a module through an inout port.

Port declarations can also specify the size of the port. There are *scalar* and *vector* ports. Scalar ports are connections from and to one-bit objects in another module. Vector ports are connection from and to multibit objects in another module. If a port is a vector port, you must specify its size in its port declaration.

7

5. Now enter an input port declaration.

```
    module flop (data,clock,clear,q,qb);
    input data,clock,clear;
```

Input port declarations begin with the keyword input. This declaration specifies that logic values propagate into the module through the ports whose identifiers are data, clock, and clear.

Separate the ports in a port declaration with commas.

End port declarations with a semicolon.

6. Now enter an output port declaration.

```
    module flop (data,clock,clear,q,qb);
    input data,clock,clear;
    output q,qb;
```

Output port declarations begin with the keyword output. This declaration specifies that logic values propagate out of the module through ports with the identifiers q and qb.

This entry also ends the port declarations in this example. The labeled items in the following copy of the schematic shows the objects that are now in module `flop`.

clear

clock

data

q

qb

Gate Declarations

The entries that you make next specify gates and their connections. Before you make these entries, you must understand the following concepts:

CONCEPTS

data type – A classification of a variable that can store or propagate values. The main groups of data types are *nets* and *registers*.

net – A group of data types that represents physical connections between objects in your design.

register – In the Verilog HDL registers are a group of data types that stores data; a register is a programming variable, not a hardware register that can be driven by another part of a design.

reg – A register data type that you can use for general purposes such as holding a value.

wire – A net data type that models a physical connection that neither performs wired logic nor stores a charge. It is typically used to connect a gate to the gate's load.

primitive – A gate or a transistor.

primitive declaration – A statement that tells Verilog-XL to apply its definition for a type of gate or transistor to certain objects. Primitive declarations contain *primitive instances* that specify the identifiers of the objects to which Verilog-XL applies the definition and the connections to those objects. Primitive declarations are sometimes called *primitive instantiations*. Primitive declarations of gates are sometimes called *gate instantiations*.

primitive instance – A single use of Verilog-XL's definition for a type of gate or transistor. A primitive instance includes a terminal list that specifies the input and output connections to the gate or the source, drain, and gate connections to a transistor. Primitive instances of gates are sometimes called g*ate instances*. Verilog-XL applies the values on the gate's input terminals to the definition to determine the value of the output terminal.

delay expression – An expression that specifies an amount of simulation time that elapses between two events. In a gate declaration, a delay expression specifies the interval of simulation time between a transition of the value of a gate's input terminal and the subsequent transition of the value of the gate's output terminal. The term "delay expression" is sometimes abbreviated to "delay".

7. Now begin to enter gate declarations.

```
      module flop (data,clock,clear,q,qb);
      input data,clock,clear;
      output q,qb;


      nand
```

The keyword nand begins a declaration for a NAND gate.

8. Now enter a delay expression for this gate declaration.

```
      module flop (data,clock,clear,q,qb);
      input data,clock,clear;
      output q,qb;


      nand #10
```

This entry tells Verilog-XL that in the gate instances that follow in this declaration, the value of the gate's output terminal changes 10 time units after a change in the value of the gate's input terminal. This model does not specify the scale of the time unit so you can assume that a time unit is one nanosecond.

9. Now enter a gate instance.

```
    module flop (data,clock,clear,q,qb);
    input data,clock,clear;
    output q,qb;


    nand #10 nd1 (a,data,clock,clear),
```

This entry consists of one gate instance followed by a comma to separate this gate instance from other gate instances in the declaration. This gate instance has two parts:

nd1   The gate instance identifier. Gate instance names are optional; they are useful when you debug your model.

(a,data,clock,clear)   The gate's *terminal connection list*. Terminal connection lists are the required part of a gate instance.

A terminal connection list specifies how a gate connects to the rest of the model. This terminal connection list contains four net identifiers, separated by commas. Verilog-XL connects the first net in the list to the gate output terminal and the remaining nets to its input terminals. A four-terminal connection list tells Verilog-XL that this gate connects to the rest of your model through one output terminal and three input terminals.

The first terminal in a terminal connection list is always an output terminal. This output terminal identifier, a, the first identifier in the terminal connection list, is an identifier that you have not used before in this module. When Verilog-XL encounters this identifier, it assumes that a is a scalar wire that connects NAND gate nd1 to some other primitive in your design. When you enter in a terminal connection list an identifier that you have not used before, you make by default an *implicit declaration* of a wire.

The remaining terminal identifiers in this terminal connection list are input terminals. Terminal identifiers data, clock and clear have the same identifiers as the input ports of this module. By entering these identifiers in the terminal connection list for gate nd1, you specify a connection between the module's input ports and the gate's input terminals.

The following copy of the schematic shows the identifiers in this instance.

clear

nd1

a
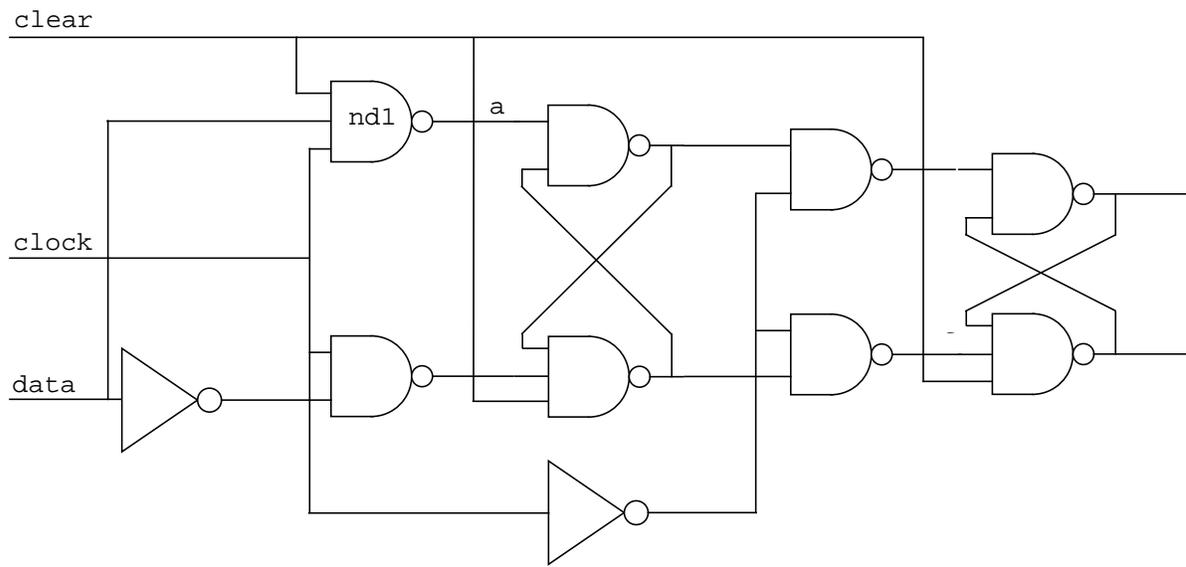
clock

data

The following steps are entries of more gate instances in this gate declaration.

10. Now instantiate NAND gate `nd2`.
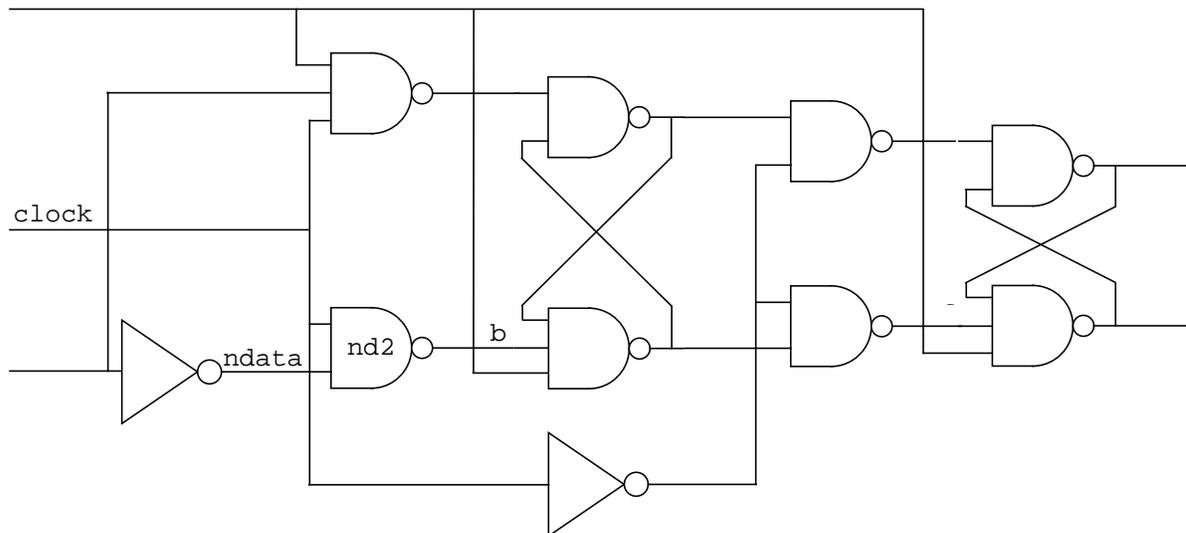
```
    module flop (data,clock,clear,q,qb);
    input data,clock,clear;
    output q,qb;

    nand #10 nd1 (a,data,clock,clear),
            nd2 (b,ndata,clock),
```

When you *instantiate* you make a use of a definition.

The following copy of the schematic shows the identifiers in this instance.

11. Now instantiate NAND gate nd4.

```
module flop (data,clock,clear,q,qb);
input data,clock,clear;
output q,qb;


nand #10 nd1 (a,data,clock,clear),
         nd2 (b,ndata,clock),
         nd4 (d,c,b,clear),
```

You do not enter a gate instance for NAND gate nd3 in this declaration because the design specifies a nine nanosecond propagation delay for gate nd3 and this declaration is for NAND gates with a 10 nanosecond delay.

The following copy of the schematic shows the identifiers in this instance.

15

12. Now instantiate NAND gate nd5.

```
module flop (data,clock,clear,q,qb);
input data,clock,clear;
output q,qb;

nand #10 nd1 (a,data,clock,clear),
          nd2 (b,ndata,clock),
          nd4 (d,c,b,clear),
          nd5 (e,c,nclock),
```
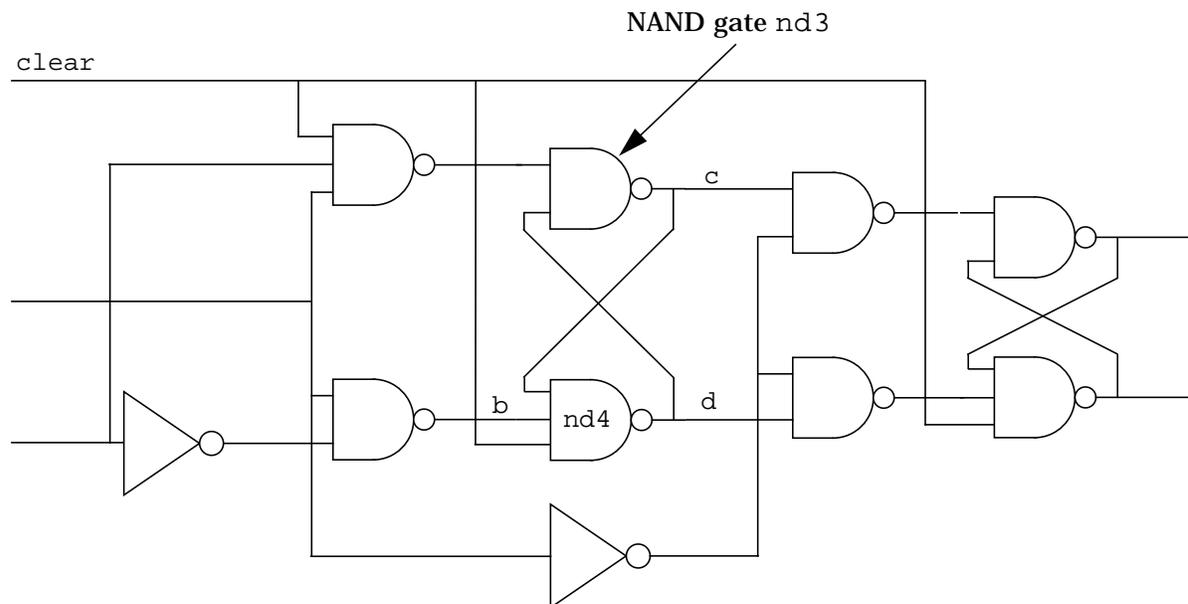
The following copy of the schematic shows the identifiers in this instance.

13. Now instantiate NAND gate nd6.

```
module flop (data,clock,clear,q,qb);
input data,clock,clear;
output q,qb;

nand #10 nd1 (a,data,clock,clear),
        nd2 (b,ndata,clock),
        nd4 (d,c,b,clear),
        nd5 (e,c,nclock),
        nd6 (f,d,nclock),
```

The following copy of the schematic shows the identifiers in this instance.

14. Now instantiate NAND gate nd8.
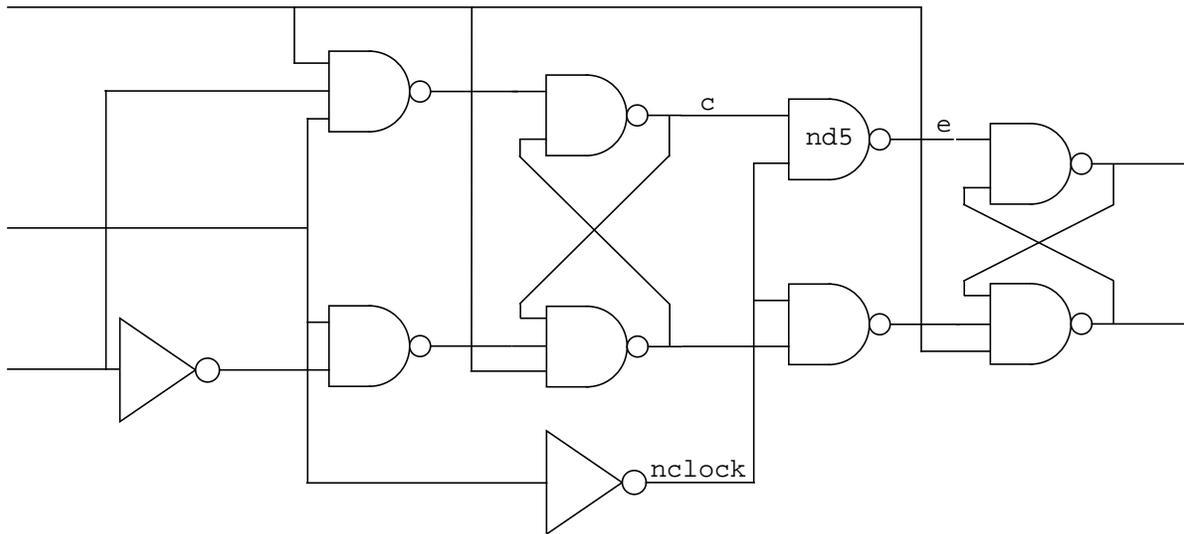
```
module flop (data,clock,clear,q,qb);
input data,clock,clear;
output q,qb;

nand #10 nd1 (a,data,clock,clear),
         nd2 (b,ndata,clock),
         nd4 (d,c,b,clear),
         nd5 (e,c,nclock),
         nd6 (f,d,nclock),
         nd8 (qb,q,f,clear);
```

You do not instantiate NAND gate nd7 in this declaration because it also does not have a delay of 10 time units.

This gate instance is the last entry in this NAND gate declaration so it is followed by a semicolon instead of a comma. Semicolons terminate gate declarations.

This NAND gate declaration contains gate instances for all the NAND gates in the model whose delays are 10 time units.

The following copy of the schematic shows the identifiers in this instance.

18

15. Begin the declaration for NAND gates with a nine time unit delay.
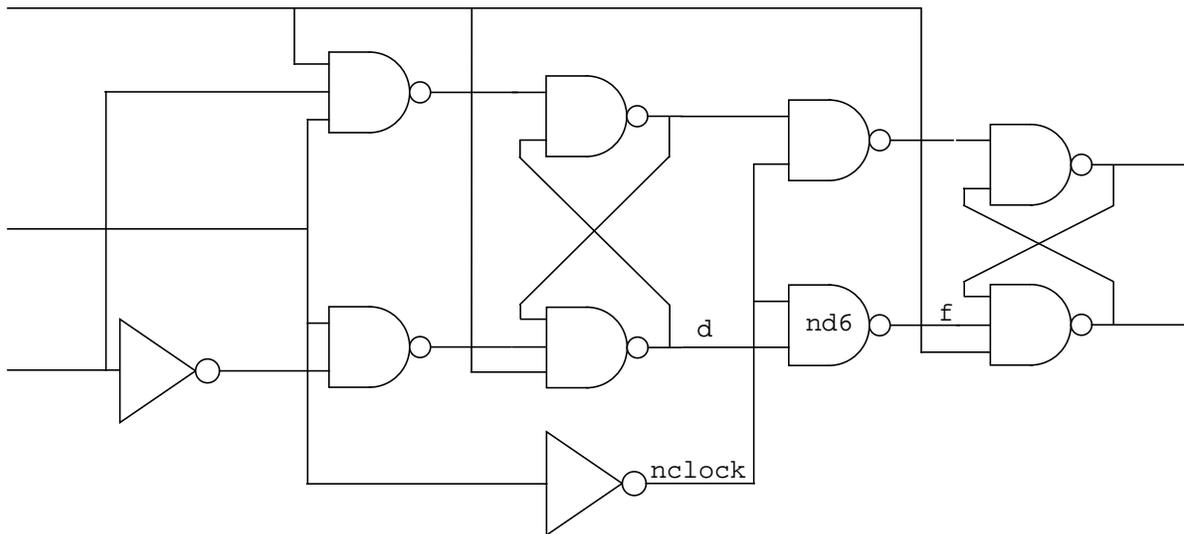
```
module flop (data,clock,clear,q,qb);
input data,clock,clear;
output q,qb;

nand #10 nd1 (a,data,clock,clear),
         nd2 (b,ndata,clock),
         nd4 (d,c,b,clear),
         nd5 (e,c,nclock),
         nd6 (f,d,nclock),
         nd8 (qb,q,f,clear);
nand #9
```

The NAND gate instances that you enter in this declaration have a propagation delay of nine time units.

16. Instantiate NAND gate `nd3`.

```
    module flop (data,clock,clear,q,qb);
    input data,clock,clear;
    output q,qb;

    nand #10 nd1 (a,data,clock,clear),
             nd2 (b,ndata,clock),
             nd4 (d,c,b,clear),
             nd5 (e,c,nclock),
             nd6 (f,d,nclock),
             nd8 (qb,q,f,clear);
    nand #9 nd3 (c,a,d),
```

The following copy of the schematic shows the identifiers in this instance.

.

17. Instantiate NAND gate `nd7`
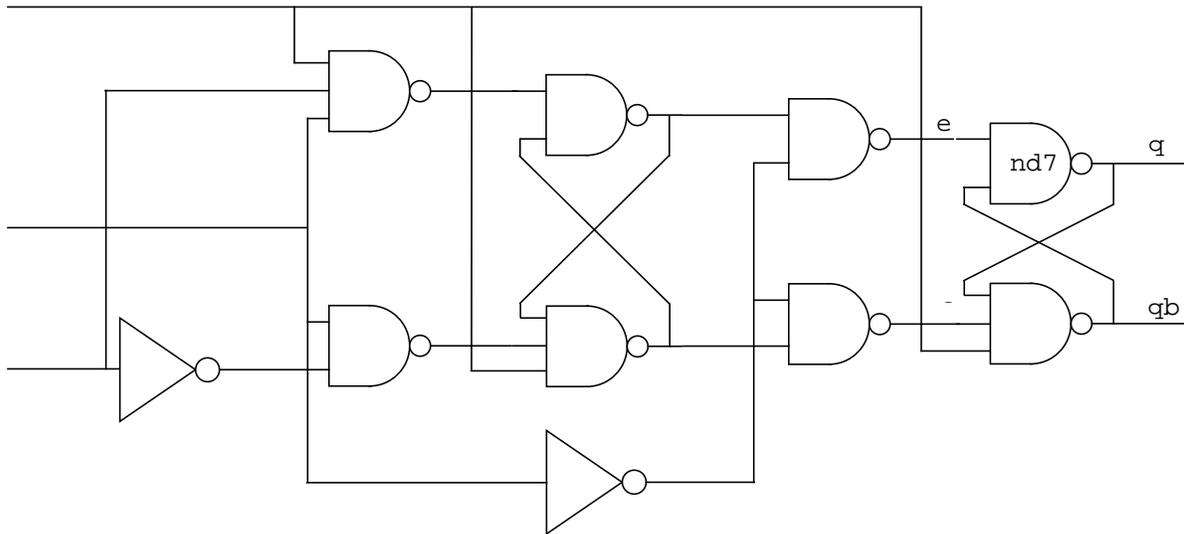
```
      module flop (data,clock,clear,q,qb);
      input data,clock,clear;
      output q,qb;

      nand #10 nd1 (a,data,clock,clear),
               nd2 (b,ndata,clock),
               nd4 (d,c,b,clear),
               nd5 (e,c,nclock),
               nd6 (f,d,nclock),
               nd8 (qb,q,f,clear);
      nand #9 nd3 (c,a,d),
              nd7 (q,e,qb);
```

The following copy of the schematic shows the identifiers in this instance.

18. Now begin to declare the inverters.

```
    module flop (data,clock,clear,q,qb);
    input data,clock,clear;
    output q,qb;

    nand #10 nd1 (a,data,clock,clear),
             nd2 (b,ndata,clock),
             nd4 (d,c,b,clear),
             nd5 (e,c,nclock),
             nd6 (f,d,nclock),
             nd8 (qb,q,f,clear);
    nand #9  nd3 (c,a,d),
             nd7 (q,e,qb);
    not #10
```

You declare inverters with a NOT gate declaration. NOT gate declarations begin with the keyword `not`.

The design specifies a propagation delay of 10 time units on these inverters.

22
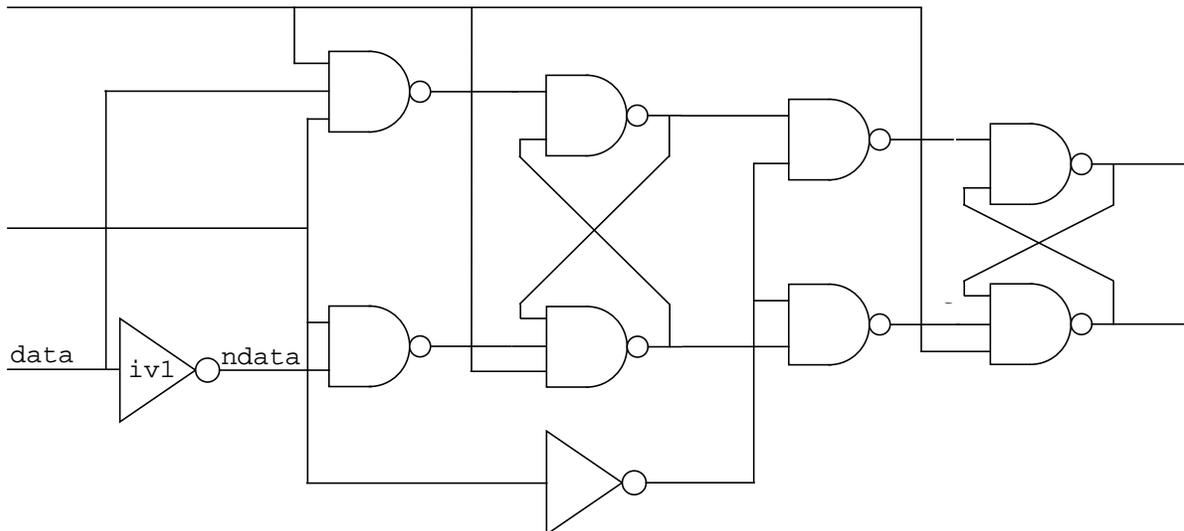
19. Instantiate inverter `iv1`.

```
module flop (data,clock,clear,q,qb);
input data,clock,clear;
output q,qb;

nand #10 nd1 (a,data,clock,clear),
         nd2 (b,ndata,clock),
         nd4 (d,c,b,clear),
         nd5 (e,c,nclock),
         nd6 (f,d,nclock),
         nd8 (qb,q,f,clear);
nand #9  nd3 (c,a,d),
         nd7 (q,e,qb);
not #10 iv1 (ndata,data),
```

The following copy of the schematic shows the identifiers in this instance.

20. Instantiate inverter `iv2`.
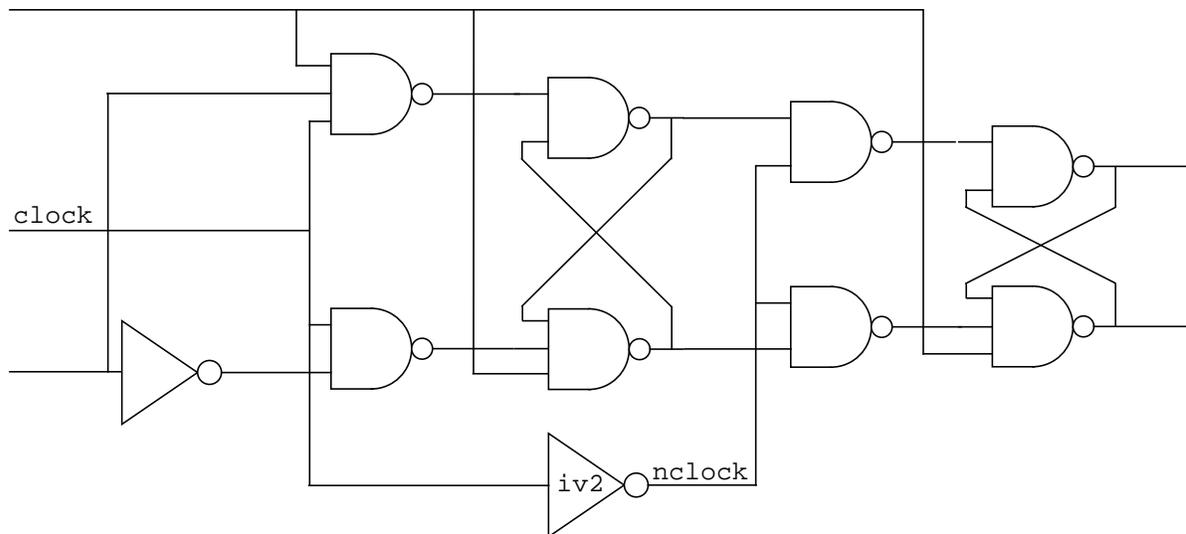
```
module flop (data,clock,clear,q,qb);
input data,clock,clear;
output q,qb;

nand #10 nd1 (a,data,clock,clear),
         nd2 (b,ndata,clock),
         nd4 (d,c,b,clear),
         nd5 (e,c,nclock),
         nd6 (f,d,nclock),
         nd8 (qb,q,f,clear);
nand #9  nd3 (c,a,d),
         nd7 (q,e,qb);
not #10  iv1 (ndata,data),
         iv2(nclock,clock);
```

The following copy of the schematic shows the identifiers in this instance.

The End of the Module

21. Now end the module.

```
module flop (data,clock,clear,q,qb);
input data,clock,clear;
output q,qb;


nand #10 nd1 (a,data,clock,clear),
         nd2 (b,ndata,clock),
         nd4 (d,c,b,clear),
         nd5 (e,c,nclock),
         nd6 (f,d,nclock),
         nd8 (qb,q,f,clear);
nand #9  nd3 (c,a,d),
         nd7 (q,e,qb);
not #10 iv1 (ndata,data),
         iv2(nclock,clock);
endmodule
```

This entry completes the definition of the module of the flip-flop. All modules end with the keyword `endmodule`.

Writing a Module Definition for the Register

The next model is the 4-bit register. Before you begin to write this model, you must understand the following concepts:

CONCEPTS

hierarchical model – A model that consists, at least in part, of other models.

module hierarchy – A system of modules and hierarchical modules that describe a design. A typical module hierarchy contains the following models:

- a top-level hierarchical model that describes the entire design; its description includes references to models of large parts of a design

- models that describe large parts of the design; their descriptions include references to models of small parts of the design

- bottom-level or leaf models that describe small parts of the design

The following figure shows a module hierarchy.

other modules in the hierarchy

top-level module

bottom-level modules

26
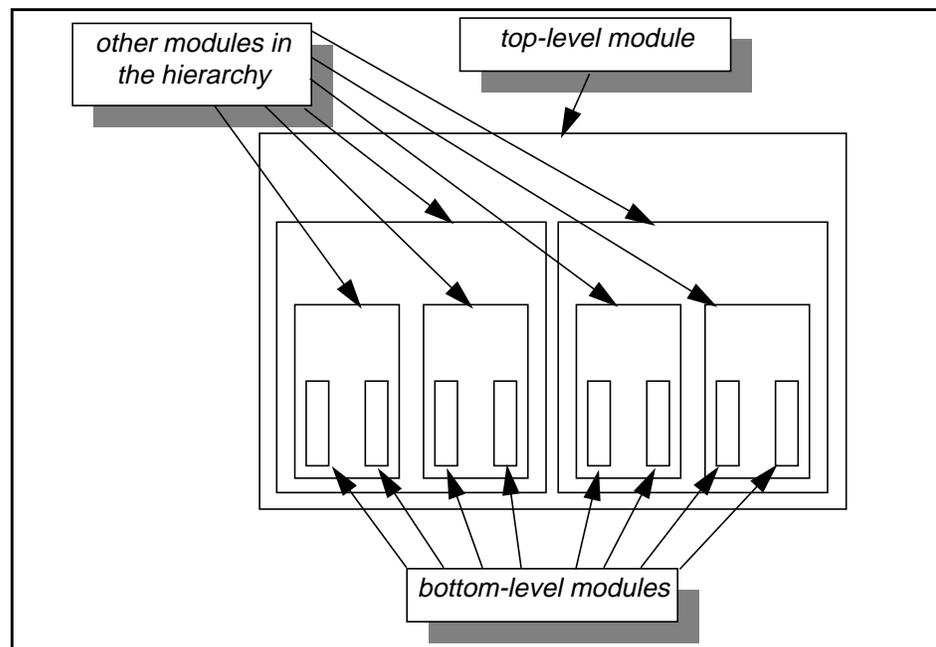
module definition – an entire module comprises a module definition. The entire contents of a module describe how it works or how it is put together.

module instance – A reference to a module definition to describe a part of a hierarchical model. Entering a module instance in a source description is called *instantiating* a module definition. When you *instantiate* a module definition you tell Verilog-XL that a part of a design is a copy of, or behaves like, the design described in that module.

The following comparison may be helpful:

module instance vs. gate instance – Both a module instance and a gate instance are uses of a definition—gate definitions are in the executable file that is Verilog-XL, you write module definitions in your source description.

The following schematic illustrates this model of a 4-bit register:

27
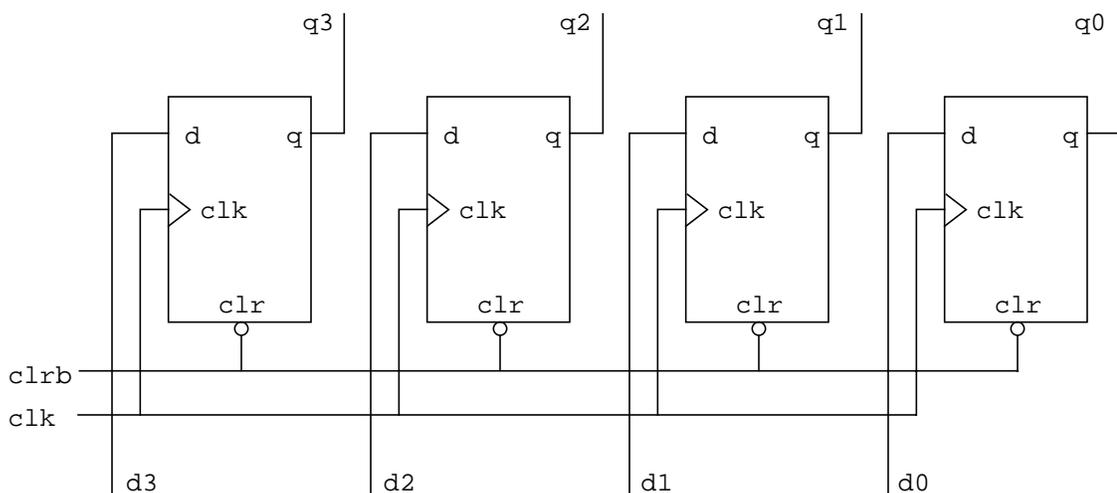
1. Open a text file with the name `hardreg.v`.

The Module Header

2. Enter the module header.

```
module hardreg (d,clk,clrb,q);
```

This entry specifies that there is a module with the identifier `hardreg` whose ports have the identifiers d, clk, clrb, and q.

Port Declarations

3. Declare the scalar input ports.

```
module hardreg (d, clk,clrb,q);
input clk, clrb;
```

This input port declaration specifies no bit width so Verilog-XL uses the default width of one bit for ports `clk` and `clrb`. This input port declaration does not include d because d is four bits wide.

4. Declare the vector input port.

```
module hardreg (d, clk,clrb,q);
input clk, clrb;
input [3:0] d;
```

The part of this entry that is enclosed in brackets is a range specification. A range specification establishes the bit width of an object such as a port and specifies the indices. A range specification begins with the bit number of the object's most significant bit (MSB), followed by a colon (:) and the bit number of the object's least significant bit (LSB).

This input port declaration specifies that d is an input port that is four bits wide, and that from the MSB to the LSB the bits are numbered 3, 2, 1 and 0.
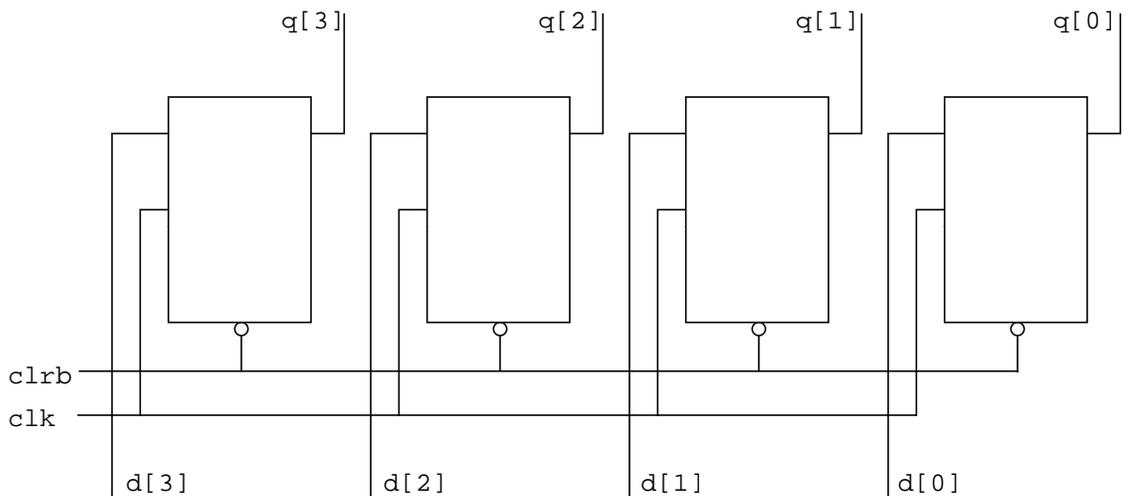
28

5. Declare the vector output port.

```
module hardreg (d, clk,clr,q);
input clk, clrb;
input [3:0] d;
output [3:0] q;
```

This output port declaration specifies that q is an output port that is four bits wide and that from the MSB to the LSB, the bits are numbered 3, 2, 1, and 0.

This entry ends the port declarations. The labeled items in the following copy of the schematic show the identifiers that are now in module hardreg.

29

Module Instantiation and Module Instances

6. Begin to instantiate the module of the flip-flop.

```
module hardreg (d, clk,clrb,q);
input clk, clrb;
input [3:0] d;
output [3:0] q;

flop
```

This entry begins a module instantiation. It tells Verilog-XL that one or more parts of module `hardreg` are instances of a module definition named `flop`.

7. Enter one instance of module `flop`.

```
module hardreg (d, clk,clrb,q);
input clk, clrb;
input [3:0] d;
output [3:0] q;

flop f1 (d[0],clk,clrb,q[0],),
```

This entry is a module instance of module `flop`. Because it is the first of a series of instances of `flop`, it is followed by a comma instead of a semicolon. Commas separate module instances in a module instantiation.

Module instances have two parts. In this instance the parts are as follows:

f1   The identifier of this instance of module
                                       `flop`.

`(d[0],clk,clrb,q[0],)` The port connection list.

You must enter a module instance name. A module instance name can help you to debug a hierarchical module that contains more than one instance of a module.
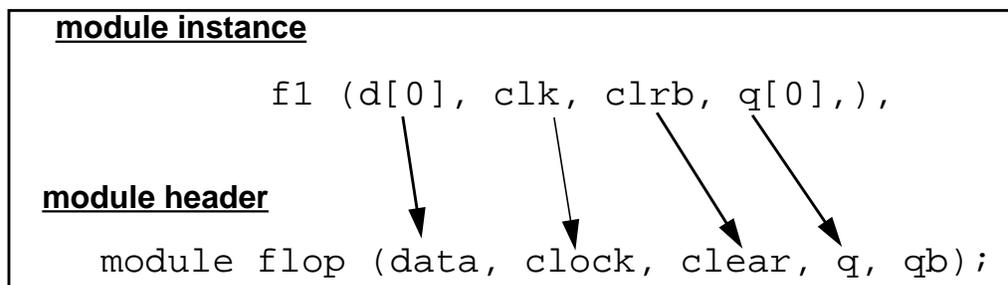
This port connection list is an *order-based* port connection list. The Verilog HDL has two types of port connection lists:

- name-based

- order-based

In a name-based port connection list, you explicitly specify the connection between the variable or port in the module that contains the instance and the port in the instantiated module.

In an order-based port connection list, you specify connections by matching the order-based port connection list in the module instance with the order of the port connection list in the module header of the instantiated module.

Compare module `flop`'s module header with this instance of module `flop` to see how the instance's port connection list shows Verilog-XL how to connect `hardreg` to this instance of `flop`.

---

module instance

```
            f1 (d[0], clk, clrb, q[0],),
```

module header

```
    module flop (data, clock, clear, q, qb);
```
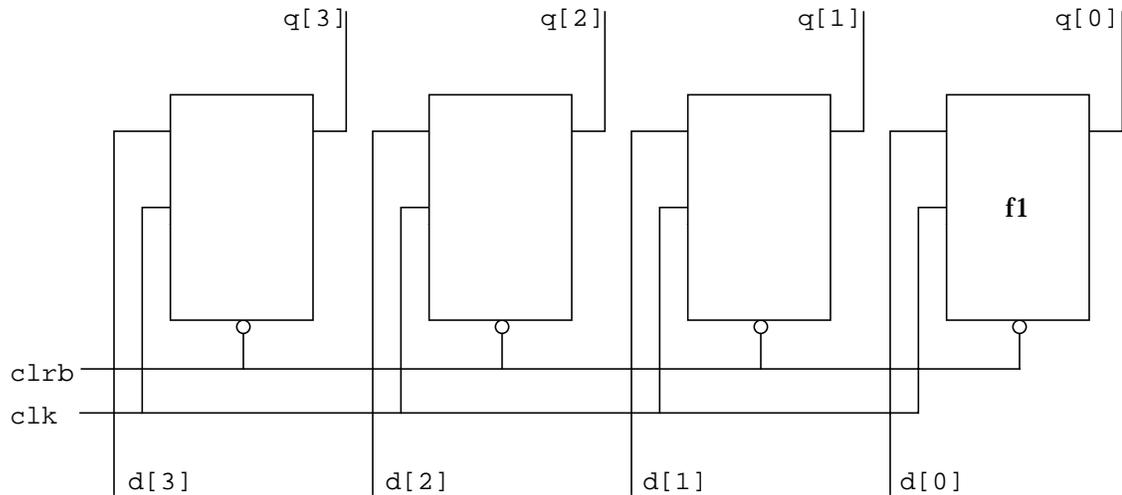

The instance's order-based port connection list tells Verilog-XL to make the following connections:

- connect bit number 0 of module `hardreg`'s 4-bit input port `d` to this instance of module `flop`'s input port `data`

- connect module `hardreg`'s input port `clk` to this instance of module `flop`'s input port `clock`

- connect module `hardreg`'s input port `clrb` to this instance of module `flop`'s input port `clear`

- connect bit number 0 of module `hardreg`'s 4-bit output port `q` to this instance of module `flop`'s output port `q`
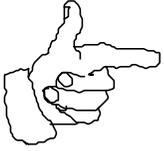
Please note: The comma between q[0] and the right parenthesis in the module instance tells Verilog-XL that you know that module flop has a fifth port, but that you make no connection to that port. Module hardreg does not need the data on module flop's output port qb.

The following schematic shows instance f1.

8. Enter another instance of module `flop`.

```
module hardreg (d, clk,clrb,q);
input clk, clrb;
input [3:0] d;
output [3:0] q;

flop f1 (d[0],clk,clrb,q[0],),
      f2 (d[1],clk,clrb,q[1],),
```

Here, the instance name is `f2`. The instance's port connection list tells Verilog-XL to make the following connections:

- connect bit number 1 of module `hardreg`'s 4-bit input port `d` to this instance of module `flop`'s input port `data`
- connect module `hardreg`'s input port `clk` to this instance of module `flop`'s input port `clock`
- connect module `hardreg`'s input port `clrb` to this instance of module `flop`'s input port `clear`
- connect bit number 1 of module `hardreg`'s 4-bit output port `q` to this instance of module `flop`'s output port `q`

Also in this instance you make no connection to module `flop`'s output port `qb`.

The following schematic shows instance `f2`.
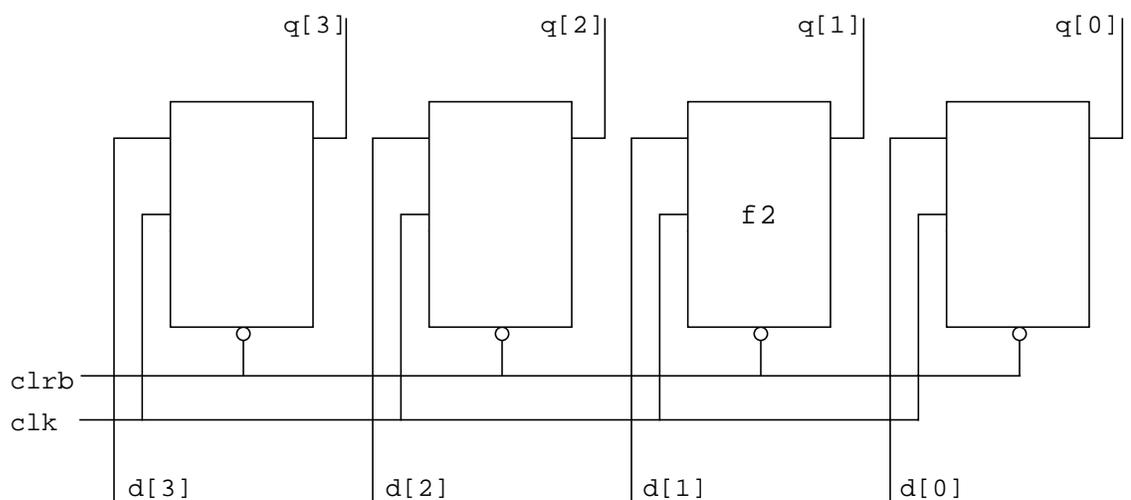
9. Enter a third instance of module `flop`.

```
module hardreg (d, clk,clrb,q);
input clk, clrb;
input [3:0] d;
output [3:0] q;

flop f1 (d[0],clk,clrb,q[0],),
     f2 (d[1],clk,clrb,q[1],),
     f3 (d[2],clk,clrb,q[2],),
```
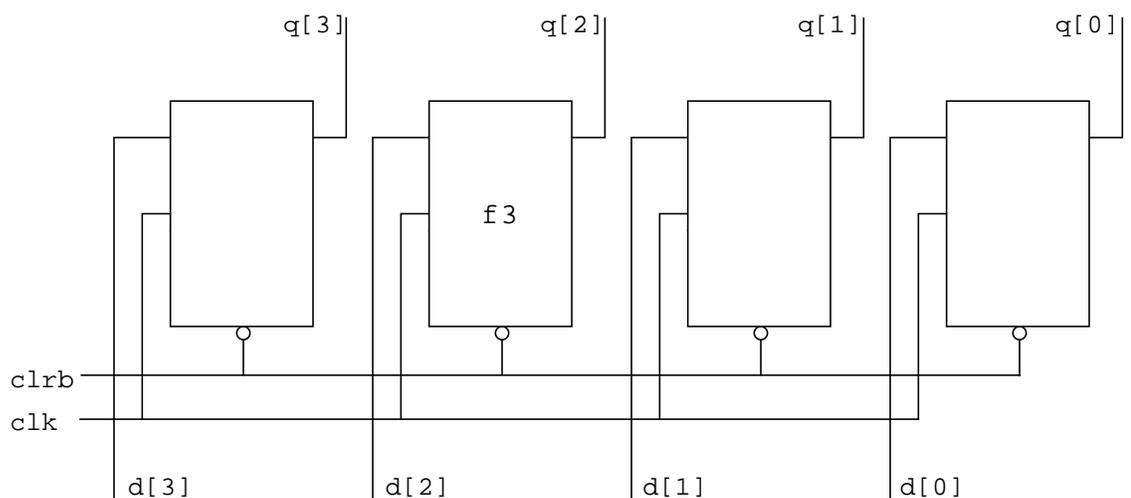
In this entry, the instance name is `f3`. The instance's port connection list tells Verilog-XL to make the following connections:

- connect bit number **2** of module `hardreg`'s 4-bit input port `d` to this instance of module `flop`'s input port `data`
- connect module `hardreg`'s input port `clk` to this instance of module `flop`'s input port `clock`
- connect module `hardreg`'s input port `clrb` to this instance of module `flop`'s input port `clear`
- connect bit number **2** of module `hardreg`'s 4-bit output port `q` to this instance of module `flop`'s output port `q`

Also in this instance you make no connection to module `flop`'s output port `qb`.

The following schematic shows instance `f3`.

10. Enter the fourth and final instance of module `flop`.

```
module hardreg (d, clk,clrb,q);
input clk, clrb;
input [3:0] d;
output [3:0] q;

flop f1 (d[0],clk,clrb,q[0],),
     f2 (d[1],clk,clrb,q[1],),
     f3 (d[2],clk,clrb,q[2],),
     f4 (d[3],clk,clrb,q[3],);
```

This entry ends with a semicolon because this is the last instance in this instantiation of module flop. The instance name is `f4`. The instance's port connection list tells Verilog-XL to make the following connections:

- connect bit number 3 of module `hardreg`'s 4-bit input port `d` to this instance of module `flop`'s input port `data`

- connect module `hardreg`'s input port `clk` to this instance of module `flop`'s input port `clock`

- connect module `hardreg`'s input port `clrb` to this instance of module `flop`'s input port `clear`

- connect bit number 3 of module `hardreg`'s 4-bit output port `q` to this instance of module `flop`'s output port `q`

Also in this instance you make no connection to module `flop`'s output port `qb`.

The following schematic shows instance `f4`.
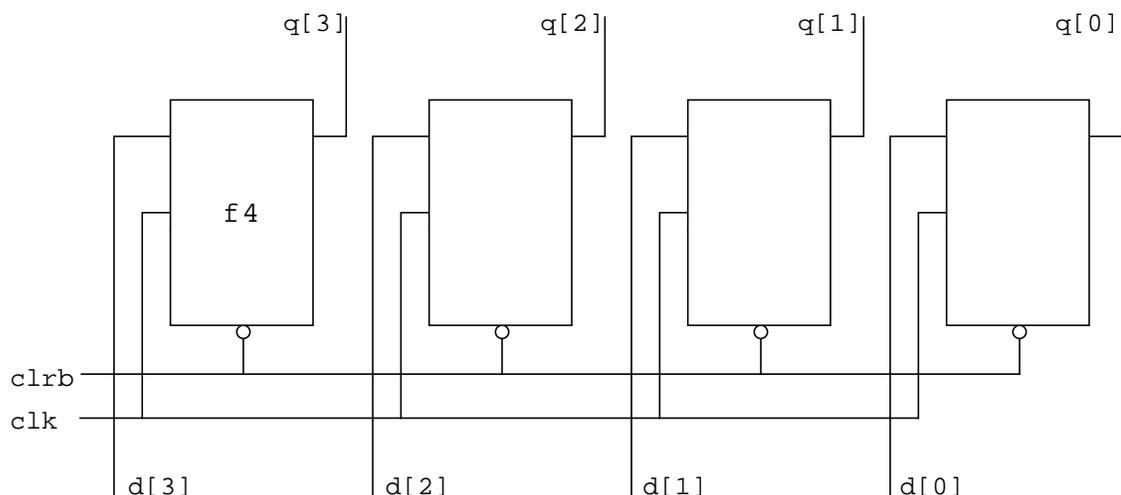
11. Enter the keyword to end the module.

```
module hardreg (d,clk,clrb,q);
input clk, clrb;
input [3:0] d;
output [3:0] q;

flop f1 (d[0],clk,clrb,q[0],),
     f2 (d[1],clk,clrb,q[1],),
     f3 (d[2],clk,clrb,q[2],),
     f4 (d[3],clk,clrb,q[3],);

endmodule
```

This entry ends the definition of module `hardreg`.

Writing a Test Fixture Model

Now that you have completed your model of the 4-bit register, the next step is to write a model of a test fixture that tests the logic of the design.

1. Open another text file named `harddrive.v`.

The test fixture model has the following parts:

• module header, variable declarations, and design instance
• application of stimulus—the design driving part
• results monitoring part

The test fixture model is a *behavioral* model, not a *structural* model. The following comparison describes the difference between these types of models:

behavior model vs. structural model – A behavioral model describes the behavior of a device—it contains a procedure that performs the operations of the device. It does not describe how you connect components to perform that behavior. A structural model describes how you connect the component gates or transistors in the device.

The test fixture model applies values to your design and displays the results on your screen.
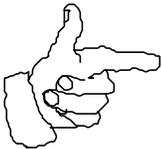
Please note: You can also model your design with a behavioral model. A behavioral model of a design models at a more abstract level.

Module Header, Declarations, and Design Instance

This section shows how to enter the module header, the variable declarations and the design instance.

Top-Level Module Header

2. Enter the module header.

```
module hardrive;
```

This entry is the module header for a top-level module. Top-level module headers do not contain a port connection list.

37

Data Type Declarations

The next steps are declarations of variables of the data type `reg` and `wire`. A `reg` is an abstract storage element. No part of a design propagates values to a `reg` but you can connect a `reg` to part of your design to propagate values to that part. A `wire` is a physical connection between objects.

3. Declare the scalar `regs` `clr` and `clk`.

```
module hardrive;
reg clr, clk;
```

This entry declares two variables with the name `clk` and `clr` that have the `reg` data type. Since you entered no bit width, they are one bit wide.

4. Declare the vector `reg` `data`.

```
module hardrive;
reg clr, clk;
reg [3:0] data;
```

This entry declares a `reg` named `data` that is four bits wide. From the MSB to the LSB, the bits are numbered 3, 2,1 and 0.

38

5. Declare the vector wire `q`.

```
    module hardrive;
    reg clr, clk;
    reg [3:0] data;
    wire [3:0] q;
```

The Design Instance

6. Enter an instance of module `hardreg`.

```
    module hardrive;
    reg clr, clk;
    reg [3:0] data;
    wire [3:0] q;

    hardreg h1 (data, clk, clr, q);
```

This entry has the following parts:

`hardreg`	the module identifier of the module that contains the model of the 4-bit register
`h1`	the name of this instance of module `hardreg`
`(data, clk, clr, q);`	the order-based port connection list for this instance of module `hardreg`

The port connection list makes the following connections from module `hardrive` to module `hardreg`:

- connect module `hardrive`'s 4-bit reg `data` to this instance of module `hardreg`'s 4-bit input port d

- connect module `hardrive`'s one-bit reg `clk` to this instance of module `hardreg`'s one-bit input port `clk`

- connect module `hardrive`'s one-bit reg `clr` to this instance of module `hardreg`'s one-bit input port `clr`

- connect module `hardrive`'s four-bit wire `q` to this instance of module `hardreg`'s output port `q`

Figure 3 shows how data values propagate into the 4-bit register from the model of the test fixture.
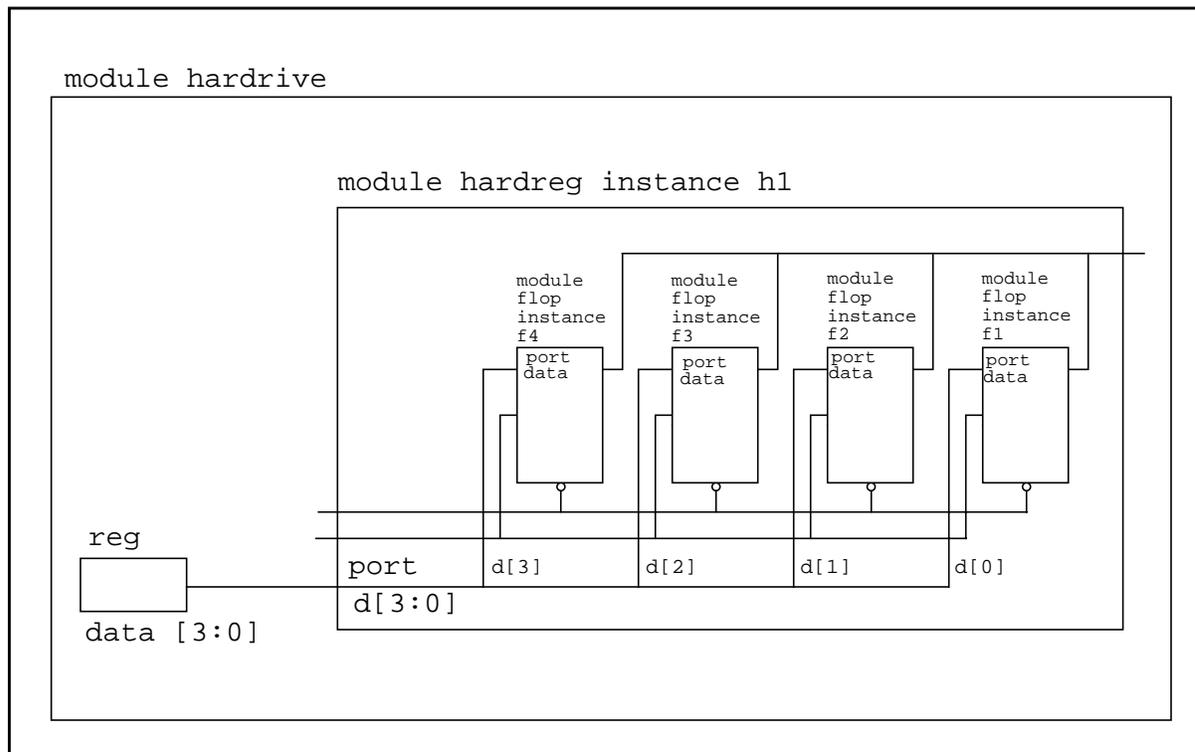
Figure 3: Propagating data values

Data values propagate down the hierarchy along the following paths:

1.   Data values originate in module `hardrive`'s 4-bit reg `data`.
2.   Data values propagate from module `hardrive`'s 4-bit reg `data` through the hierarchically lower-level module `hardreg`'s 4-bit input port `d`.
3.   The values on each bit of module `hardreg`'s 4-bit input port `d` propagate through the single bit ports named `data` in each instance of module `flop`.

Applying Stimuli to the Design

This test fixture module uses the following strategy to apply stimulus:

1.   Initialize `clr` to 1, `clk` to 0, and apply a test pattern vector of all 0's to the four bits of data.
2.   Toggle `clk` every 50 time units.
3.   Apply a new test pattern vector to `data` every 100 time units until `hardrive` applies test patterns from 0000 to 1111.
4.   After the last test pattern vector propagates to the output, toggle `clr` and once again apply the test pattern vectors from 0000 to 1111.

Initializing the Clock and Clear Lines

The next step is to make the entries that model the behavior of applying the first values to `clk` and `clr`. Before you make these entries you must understand the following concepts about behavior modeling:

CONCEPTS

procedural block–One or more procedural statements that specify an activity or activities that Verilog-XL performs during the simulation. Procedural blocks are sometimes called procedures. You specify procedural blocks in one of the following statements:

> `initial` statement
> `always` statement
> `task`
> `function`

This tutorial includes `initial` and `always` statements.

procedural statement– A statement that controls the simulation or manipulates variables such as registers and nets.

procedural assignment statement– A procedural statement that assigns a value to a register.

initial statement– A statement that specifies that Verilog-XL executes—starting at simulation time 0—one or more procedural statements when the simulation starts, and does not execute them again.

always statement– A statement that specifies that Verilog-XL executes—starting at simulation time 0—one or more procedural statements continuously until the simulation stops.

block statement– A statement that permits multiple statements in places where the Verilog HDL syntax calls for one statement. The Verilog HDL contains two block statements:

`begin-end`   The sequential block. Verilog-XL executes the procedural statements inside this block statement one after the other.

`fork-join`   The parallel block. Verilog-XL executes the procedural statements inside this block concurrently.
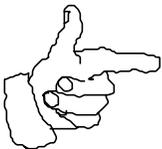
Please note: You can enter more than one procedural block. If you have multiple `initial` or `always` procedural blocks, Verilog-XL executes them concurrently.

7. Begin an `initial` procedural block.

```
    module hardrive;
    reg clr, clk;
    reg [3:0] data;
    wire [3:0] q;

    hardreg h1 (data, clk, clr, q);

    initial
```

This entry is a keyword. It specifies that Verilog-XL executes once—only one time—the procedural statement or block of procedural statements that follow this keyword.

This step is the beginning of a procedural block that assigns logic values to some of module `hardrive`'s regs.

8. This step enters a block statement that includes assignment statements to assign values to `clr` and `clk`. The Verilog HDL has three kinds of assignment statements:

- continuous assignments
- procedural continuous assignments
- procedural assignments

This tutorial only includes procedural assignments. In all three kinds of assignments there are two basic parts that are on opposite sides of the equal sign (=), the left-hand side and the right-hand side.

In a procedural assignment statement these parts must contain the following types of variables:

The left-hand side	This side must be all or part of a vector or scalar reg or an element of a memory (a memory is an array of registers with both a bit-size and depth). The left-hand side can also contain a concatenation or grouping of all or part of more than one reg or memory element.
The right-hand side	This side can contain any expression.

Enter a begin-end block that contains procedural assignment statements that assign values to `clr` and `clk` when the simulation starts.

```
module hardrive;
reg clr, clk;
reg [3:0] data;
wire [3:0] q;

hardreg h1 (data, clk, clr, q);

initial
begin
     clr = 1;
     clk = 0;
end
```

This entry specifies that when the simulation time is 0, Verilog-XL takes the following steps:

1.   assigns a value of 1 to reg `clr`
2.   assigns a value of 0 to reg `clk`

All assignment statements end with a semicolon(;). The begin-end block does *not* end with a semicolon.

This procedural block has the following purposes:

• The functional spec specifies that the clear line is active low. The value in reg `clr` propagates onto the net named `clrb` in `module hardreg`. Assigning a 1 to `clr` makes the clear line inactive so that the data input can propagate through this model of a register.

• The clock line begins the simulation with no known logic value. The value you assign to reg `clk` propagates onto the clock line so that it has an initial value and so that another part of the model can toggle that value.

Toggling the Clock

9. Enter a procedural block that toggles the clock.

```
module hardrive;
reg clr, clk;
reg [3:0] data;
wire [3:0] q;

hardreg h1 (data, clk, clr, q);

initial
begin
    clr = 1;
    clk = 0;
end

always #50 clk = ~clk;
```

This entry is an `always` procedural block that includes a delay expression and a procedural assignment.

The tilde (~) in the expression on the right-hand side of the procedural assignment is the bit-wise unary operator that inverts all the bits in the operand. The execution of this procedural assignment toggles `clk`; it assigns a new value of 0 when the value of `clk` is 1, and assigns a new value of 1 when the value of `clk` is 0.
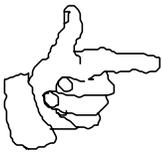
Please note: This entry specifies that as long as the simulation runs, Verilog-XL inverts the value of `clk` after every 50 time units. This entry would not change the value of `clk` at any time in the simulation if you had not first changed the value of `clk` from x to 0 with the procedural assignment in the `initial` block, because the inverted value of x is x.

Applying Test Vectors

The next steps apply the test pattern vectors. There are several ways that you can apply stimuli to a design; the method used in this module is called *vectored patterns*.

44

10. The strategy to apply stimulus to the design specifies that the test fixture model applies the test pattern vectors twice. To accomplish this task, enter a procedural block that schedules the application of the test pattern vectors twice.

```
module hardrive;
reg clr, clk;
reg [3:0] data;
wire [3:0] q;

hardreg h1 (data, clk, clr, q);

initial
begin
      clr = 1;
      clk = 0;
end

always #50 clk = ~clk;

initial
begin
      repeat (2)
```

This entry specifies an `initial` procedural block that contains a `begin-end` block statement. The first statement in this block statement is a `repeat` statement. A `repeat` statement begins with the keyword `repeat` followed by an expression in parentheses. The value of this expression specifies the number of times that Verilog-XL executes the statement that follows.

11. Begin another begin-end block and include in it a procedural assignment statement.

```
module hardrive;
reg clr, clk;
reg [3:0] data;
wire [3:0] q;

hardreg h1 (data, clk, clr, q);

initial
begin
      clr = 1;
      clk = 0;
end

always #50 clk = ~clk;

initial
begin
      repeat (2)
            begin
            data = 4'b0000;
```

This entry begins another begin-end block statement. The first procedural statement in this block statement is a procedural assignment to all four bits in the vector reg named `data`.

The expression on the right-hand side of this procedural assignment statement is a constant number specification; it has three parts:

4   The size—the specification of the number of bits Verilog-XL needs for the constant. You specify the size with base 10 digits.

'b   The base format—base formats consists of a single quote character (') followed by a character that specifies the base, or radix, of the number. The character `b` indicates that this is a binary number. The default base format is `d` for a decimal number.

0000   The number value.

With this entry, you schedule Verilog-XL to apply a test pattern of all zeroes to the input data bus of the design at the start of the simulation.

46

12. The strategy to apply stimulus calls for the test fixture to apply a new test pattern vector that is the next higher binary number after every 100 time units. The most elementary way to do this is to make the following entry:

```
#100 data = 4'b0001
```

You can reduce the amount of typing needed to enter all the test pattern vectors from binary 0 to binary 15 by defining a text macro.

```
module hardrive;
reg clr, clk;
reg [3:0] data;
wire [3:0] q;
'define stim #100 data = 4'b

hardreg h1 (data, clk, clr, q);

initial
begin
     clr = 1;
     clk = 0;
     nd

always #50 clk = ~clk;

initial
begin
     repeat (2)
          begin
          data = 4'b0000;
```

This entry is a compiler directive that defines a text macro. Compiler directives and text macros have the following definitions:

CONCEPTS

compiler directive – an accent grave character (') followed by a keyword that controls how Verilog-XL compiles and simulates your source description.

text macro – A character string that Verilog-XL substitutes for another character string when it compiles your source description.

This entry has three parts:

`'define`	the single quote character and keyword that tell Verilog-XL that this line is a compiler directive that defines a text macro
`stim`	the text macro name
`#100 data = 4'b`	the macro text

During compilation of the source description, when Verilog-XL encounters the text macro name preceded by an accent grave character (')—`'stim`— it substitutes the macro text—`#100 data = 4'b`.

Unlike statements, declarations, and instantiations, compiler directives do not end with a semicolon.

48

13. Use this text macro to apply after every 100 time units test vectors from binary 1 to binary 15.

```
module hardrive;
reg clr, clk;
reg [3:0] data;
wire [3:0] q;
'define stim #100 data = 4'b

hardreg h1 (data, clk, clr, q);

initial
begin
      clr = 1;
      clk = 0;
end

always #50 clk = ~clk;

initial
begin
      repeat (2)
            begin
            data = 4'b0000;
            'stim 0001;
            'stim 0010;
            'stim 0011;
            'stim 0100;
            'stim 0101;
            'stim 0110;
            'stim 0111;
            'stim 1000;
            'stim 1001;
            'stim 1010;
            'stim 1011;
            'stim 1100;
            'stim 1101;
            'stim 1110;
            'stim 1111;
```

These entries tell Verilog-XL when to apply the test patterns.

49

14. The strategy to apply stimulus specifies that after the test fixture applies a series of test pattern, it toggles `clr` and applies all the test patterns again. To tell Verilog-XL when to toggle `clr` you can use a named event after the last test pattern. A named event is a data type that has the following definition:

CONCEPTS

named event – an event that you trigger in a procedural block to enable actions.

You must declare a named event before you can schedule it to occur in a procedural block.

50

```
        module hardrive;
        reg clr, clk;
        reg [3:0] data;
        wire [3:0] q;
        'define stim #100 data = 4'b
        event end_first_pass;

        hardreg h1 (data, clk, clr, q);

        initial
        begin
             clr = 1;
             clk = 0;
        end

        always #50 clk = ~clk;

        initial
        begin
             repeat (2)
             begin
                  data = 4'b0000;
                  'stim 0001;
                          •
                          •
                          •
                  'stim 1111;
```

This entry declares a named event with the name end_first_pass.

15. Schedule Verilog-XL to trigger the event and end the block statement.

```
module hardrive;
reg clr, clk;
reg [3:0] data;
wire [3:0] q;
'define stim #100 data = 'b
event end_first_pass;

hardreg h1 (data, clk, clr, q);

initial
begin
     clr = 1;
     clk = 0;
end

always #50 clk = ~clk;

initial
begin
     repeat (2)
     begin
          data = 'b0000;
          'stim 0001;
               •
               •
               •
          'stim 1111;
          #200 ->end_first_pass;
     end
```

The dash and right angle bracket (->) compose the event trigger operator. This entry tells Verilog-XL to trigger the event named `end_first_pass` after a delay of 200 time units and ends the block statement.

16. Enter a procedural block that toggles `clr` at the right simulation time.

```
module hardrive;
reg clr, clk;
reg [3:0] data;
wire [3:0] q;
'define stim #100 data = 4'b
event end_first_pass;

hardreg h1 (data, clk, clr, q);

initial
begin
     clr = 1;
     clk = 0;
end

always #50 clk = ~clk;

always @(end_first_pass)
     clr = ~clr;

initial
begin
     repeat (2)
     begin
          data = 4'b0000;
          'stim 0001;
                 •
                 •
                 •
          'stim 1111;
          #200 ->end_first_pass;
     end
```

53

This entry is an `always` procedural block that contains an *event controlled statement* that is a procedural assignment. The definition of an event controlled statement is as follows:

CONCEPTS

event controlled statement – a statement whose execution Verilog-XL synchronizes with a value change of a net or register, or with the occurrence of a named event.

This entry includes the "at" symbol (@) that is the event control operator. This operator is followed by a named event in parentheses. Verilog-XL executes the procedural assignment when Verilog-XL triggers the named event.

When Verilog-XL triggers the event named `end_first_pass`, it toggles `clr`.

54

17. Schedule an end to the simulation and close the block statement.

```
module hardrive;
reg clr, clk;
reg [3:0] data;
wire [3:0] q;
'define stim #100 data = 'b
event end_first_pass;

hardreg h1 (data, clk, clr, q);

initial
begin
     clr = 1;
     clk = 0;
end

always #50 clk = ~clk;

always @(end_first_pass)
     clr = ~clr;

initial
begin
     repeat (2)
     begin
          data = 'b0000;
          'stim 0001;
                 •
                 •
                 •
          'stim 1111;
          #200 ->end_first_pass;
     end
     $finish;
end
```

The first line in this entry is a *system task*. To use system tasks you must know the following concepts:

task – a built-in or user-defined procedure.

user-defined task – a procedural block that begins with the keyword `task` followed by the name of the task and ends with the keyword `endtask`. This procedural block specifies the user-defined task so that you can enable it in another procedural block.

system task – a task that is built into Verilog-XL instead of defined in a procedural block in your source description. All system tasks begin with a dollar sign ($). Some system tasks take arguments such as character strings, variable names, or expressions. All system task calls end with a semicolon.

The `$finish` system task tells Verilog-XL to end the simulation. Verilog-XL executes this system task after it executes the `repeat` statement twice.

The `end` keyword in the second line of this entry ends the block statement that includes the `repeat` statement and the `$finish` system task.

Monitoring the Results

There are several methods that you can use to monitor the results of the simulation. You can tell Verilog-XL to write the results to a file or display the results on the screen. You can also use the graphical output facility to draw waveforms on the screen. This tutorial describes how you can display the values of the inputs and outputs on the screen.

There are also several methods that you can use to display these values on the screen. The tutorial uses the `$strobe` system task.

18. Begin to enter an `always` procedural block that contains an event controlled $strobe **system task.**

```
module hardrive;
reg clr, clk;
reg [3:0] data;
wire [3:0] q;
`define stim #100 data = 4'b
event end_first_pass;

hardreg h1 (data, clk, clr, q);

initial
begin
      clr = 1;
      clk = 0;
end

always #50 clk = ~clk;

always @(end_first_pass)
      clr = ~clr;

always @(posedge clk)
      $strobe

initial
begin
      repeat (2)
      begin
            data = 4'b0000;
            `stim 0001;
                  •
                  •
                  •
            `stim 1111;
            #200 ->end_first_pass;
      end
      $finish;
end
```

The `$strobe` system task displays a message on your screen. Unlike other system tasks that display messages, if other events occur at the same time during the simulation—events such as value changes and the execution of other system tasks—the display of the `$strobe` message occurs last. You use the `$strobe` system task to guarantee that you see stable values—the settled values after Verilog-XL executes all other events during that time in the simulation.

The `$strobe` system task takes a list of arguments and expression parameters. The arguments specify the text of the message that the system task displays and the expression parameters supply values that Verilog-XL displays in the message.

The event that controls the statement is a value change in `clk`. The keyword `posedge` qualifies this value change. The `posedge` keyword specifies that Verilog-XL synchronizes the execution of the statement with the following value changes of `clk`:

- 0 to 1
- 0 to x
- x to 1

58

Menus ▶ **Index** **Table of Contents** **Go Back** ◀ 58 of 68 ▶ **Close**

19. Enter a character string argument to the `$strobe` system task.

```
module hardrive;
reg clr, clk;
reg [3:0] data;
wire [3:0] q;
'define stim #100 data = 4'b
event end_first_pass;

hardreg h1 (data, clk, clr, q);

initial
begin
     clr = 1;
     clk = 0;
end

always #50 clk = ~clk;

always @(end_first_pass)
     clr = ~clr;

always @(posedge clk)
     $strobe("at time %0d clr =%b data=%d q=%d",

initial
begin
     repeat (2)
     begin
          data = 4'b0000;
          'stim 0001;
                    •
                    •
                    •
          'stim 1111;
          #200 ->end_first_pass;
     end
     $finish;
end
```

This entry begins the list of arguments and expression parameters to the `$strobe` system task.

 The list is preceded by a left parenthesis character.

The part of the entry that is enclosed in double quotation marks is a character string argument to the system task. All character string arguments are enclosed in double quotation marks.

This character string includes the percent sign (%) which is a special character to Verilog-XL. The percent sign is the beginning of a *format specification*. When Verilog-XL displays this character string, it replaces the format specification with a value.

The type of format specification determines the type of value that replaces it. This character string contains the following format specifications:

%d   The decimal format specification—Verilog-XL replaces this format specification with a decimal number.

%b   The binary format specification—Verilog-XL replaces this format specification with a binary number.

In a format specification, including a zero (0) between the percent sign (%) and the alphabetic character specifies that Verilog-XL uses the minimum number of spaces to display the value. If you omit the zero, Verilog-XL uses the number of spaces required by the largest possible size of the value.

Character string arguments must be on one line.

This argument is followed by a comma to separate it from the expression parameters that you enter next.

20. Enter the expression parameters to the system task.

```
module hardrive;
reg clr, clk;
reg [3:0] data;
wire [3:0] q;
'define stim #100 data = 4'b
event end_first_pass;

hardreg h1 (data, clk, clr, q);

initial
begin
     clr = 1;
     clk = 0;
end

always #50 clk = ~clk;

always @(end_first_pass)
     clr = ~clr;

always @(posedge clk)
     $strobe("at time %0d clr =%b data=%d q=%d",
               $time,clr,data,q);

initial
begin
     repeat (2)
     begin
          data = 4'b0000;
          'stim 0001;
                •
                •
                •
          'stim 1111;
          #200 ->end_first_pass;
     end
     $finish;
end
```

61

This entry specifies the expression parameters to the $strobe system task. You separate them with commas.

The $time expression parameter is a *system function*, clr and data are registers and q is a net. Functions and system functions have the following definitions:

CONCEPTS

function – A means, like tasks, of executing a common procedure from several different places in the source description. Functions differ from tasks in the following ways:

- Functions are invoked as operands in expressions, usually on the right hand side of an assignment; tasks are enabled from a statement.

- You must pass at least one input parameter to a function and you can only pass input parameters; you can pass input, output, inout, or no parameters to a task.

- Functions return a single value; tasks can return multiple values.

- Functions cannot have timing controls, they must occur in zero simulation time; tasks can use timing controls.

system function – a function that is built into Verilog-XL instead of defined in a procedural block in your source description. All system functions begin with a dollar sign ($). Only some system functions take arguments such as character strings, variable names, or expressions.

The $time system function returns the current simulation time.

You must enter an expression parameter for each format specification in the preceding character string argument.

Compare the order of the format specifications in the character string argument to the order of the expression parameters, to see the nets or registers whose values replace the format specifications when Verilog-XL displays the message.

```
$strobe ("at time %0d clr=%b data=%d q=%d",

                $time,clr,data,q);
```

In this entry, after the expression parameters, is a right parenthesis to close the list of arguments. The semicolon ends the $strobe system task.

21. Enter endmodule.

```
    module hardrive;
    reg clr, clk;
    reg [3:0] data;
    wire [3:0] q;
    `define stim #100 data = 4`b
    event end_first_pass;

    hardreg h1 (data, clk, clr, q);

    initial
    begin
        clr = 1;
        clk = 0;
    end

    always #50 clk = ~clk;

    always @(end_first_pass)
        clr = ~clr;

    always @(posedge clk)
        $strobe("at time %0d clr =%b data=%d q=%d",
                  $time,clr,data,q);

    initial
    begin
        repeat (2)
        begin
            data = 4`b0000;
            `stim 0001;
                    •
                    •
                    •
            `stim 1111;
            #200 ->end_first_pass;
        end
        $finish;
    end
    endmodule
```

Running Verilog-XL

Now that you have completed the models of the design and the test fixture, you can run Verilog-XL to see how the 4-bit register works. To accomplish this, you must first take some preliminary steps and then enter a Verilog-XL command line.

Preliminary Steps

To run Verilog-XL, you need information that is only available at your site, as follows:

• the path to your site's version of the Verilog-XL release

• a passcode

After you obtain this information, you need to enter it every time you run Verilog-XL. This information can be very long so you should define an `alias` to stand for this information. (An `alias` is Unix-specific.)

The Path to Verilog-XL

The executable file for Verilog-XL is stored in a particular device and a particular directory at your site. You need to tell the system the path to this executable file. Ask your system manager or the person who installed Verilog-XL for this path.

The following is an example of a path to Verilog-XL:

    /installed/design/cae_tools/sim/ex/verilog

The Passcode

You need a passcode to run Verilog-XL. Ask your system manager for a passcode. The following is an example of a passcode:

    3af838e0

The passcode works with the `-p` command line option. Verilog-XL works with several command line options. A command line option is a way to pass information to Verilog-XL before it compiles your source description and simulates your design. This information can be instructions that control how Verilog-XL compiles and simulates. In the case of the `-p` option, it permits you to invoke Verilog-XL. The following example shows how you can use the `-p` option to provide Verilog-XL with your passcode:

    -p3af838e0

Make sure that you enter no spaces between the `-p` option and the passcode.

The Alias

To simulate your design in Verilog-XL you must provide your system with the path to Verilog-XL, the passcode and the names of the source description files. You can define an `alias` to provide the system with a short character string that stands for both the path and the information in the `-p` option (an `alias` is Unix-specific). To define an `alias` named `verilog` to stand for this information enter the following at the system prompt:

```
alias verilog /installed/design/cae_tools/sim/ex/verilog –p3af838e0
```

The Verilog-XL Command Line

To invoke Verilog-XL to simulate the design, enter at the system prompt your alias and the source description file names:

```
verilog harddrive.v hardreg.v flop.v
```

The Results

Verilog-XL displays the following information about the simulation of modules `hardrive`, `hardreg`, **and** `flop`:

- a banner
- compilation information
- simulation information
- diagnostic messages

The Banner

The banner provides the following information:

- the version number of Verilog-XL
- the time and date of the simulation
- the software copyright notice

Verilog-XL displayed the following banner in a simulation of modules `hardrive`, `hardreg` **and** `flop`.

```
VERILOG-XL 1.5c :12:07
  * Copyright Cadence Design Systems, Inc. 1985, 1988.    *
  *      All Rights Reserved.       Licensed Software.    *
  * Confidential and proprietary information which is the *
  *        property of Cadence Design Systems, Inc.       *
```

Compilation Information

The compilation information includes the following information:

- the names of the source description files in the order that Verilog-XL compiles them
- the names of the top-level modules in your source description

Verilog-XL displayed the following compilation information for modules `hardrive`, `hardreg`, **and** `flop`:

```
Compiling source file "hardrive.v"
Compiling source file "hardreg.v"
Compiling source file "flop.v"
Highest level modules:
hardrive
```

66

Simulation Information

The simulation information is all the values and messages from a system task that displays information. Verilog-XL displayed the following simulation information from the $strobe system task:

```
at time 50 clr=1 data= 0 q= x
at time 150 clr=1 data= 1 q= 0
at time 250 clr=1 data= 2 q= 1
at time 350 clr=1 data= 3 q= 2
at time 450 clr=1 data= 4 q= 3
at time 550 clr=1 data= 5 q= 4
at time 650 clr=1 data= 6 q= 5
at time 750 clr=1 data= 7 q= 6
at time 850 clr=1 data= 8 q= 7
at time 950 clr=1 data= 9 q= 8
at time 1050 clr=1 data=10 q= 9
at time 1150 clr=1 data=11 q=10
at time 1250 clr=1 data=12 q=11
at time 1350 clr=1 data=13 q=12
at time 1450 clr=1 data=14 q=13
at time 1550 clr=1 data=15 q=14
at time 1650 clr=1 data=15 q=15
at time 1750 clr=0 data= 0 q= 0
at time 1850 clr=0 data= 1 q= 0
at time 1950 clr=0 data= 2 q= 0
at time 2050 clr=0 data= 3 q= 0
at time 2150 clr=0 data= 4 q= 0
at time 2250 clr=0 data= 5 q= 0
at time 2350 clr=0 data= 6 q= 0
at time 2450 clr=0 data= 7 q= 0
at time 2550 clr=0 data= 8 q= 0
at time 2650 clr=0 data= 9 q= 0
at time 2750 clr=0 data=10 q= 0
at time 2850 clr=0 data=11 q= 0
at time 2950 clr=0 data=12 q= 0
at time 3050 clr=0 data=13 q= 0
at time 3150 clr=0 data=14 q= 0
at time 3250 clr=0 data=15 q= 0
at time 3350 clr=0 data=15 q= 0
```

This design reguires a non-zero number of time units for values to propagate from reg `data` to wire `q`. A complete clock cycle is 100 time units. This explains why, for example, the value in reg `data` at simulation time 50 does not appear on wire `q` until simulation time 150.

This design also contains a clear line that is active when low. The simulation information shows that when reg `clr` has a value of 0, wire `q` always has a value of 0.

Diagnostic Messages

Verilog-XL always tells you the following information at the end of simulation:

- the number of events in the simulation
- the CPU time used during compilation, linking, and simulation
- the date and time of the end of the simulation

By default, Verilog-XL also tells you the file name and line number of the `$finish` system task that ends the simulation, and the simulation time of the call to this system task.

The following are the diagnostic messages from a simulation of modules `hardrive`, `hardreg`, and `flop`:

```
L45 "hardrive.v": $finish at simulation time 3400
1517 simulation events
CPU time: 1 secs to compile + 0 secs to link + 1 secs in
simulation
End of VERILOG-XL 1.5c   Jul  4, 1990  12:12:13
```