EE178 Lecture
Verilog Review

Eric Crabill

SJSU / Xilinx

Fall 2007

XILINX®

Lecture Agenda

- Suggested reference material.
- This course and my personal experience.
- The Verilog Hardware Description Language.
  - Background on hardware description languages.
  - Overview of Verilog language constructs.
  - Synthesis considerations.
  - FPGA considerations.

This Course

- This course is not an introduction to Verilog; you should have some prior experience.
- We will use Verilog as a tool to describe and model digital circuits which we will implement in FPGA devices.
- If you can’t use the tool effectively, you will have problems completing the assignments.

Reference Material

- HDL Chip Design, by Douglas J. Smith.
  - Covers VHDL and Verilog-HDL side by side.
  - Excellent reference for both languages.
  - Shows many schematic implementations.
  - Out of print…
- You can look for good books on Amazon.
- You may also find many useful references and tutorials by searching the internet.

Background on HDLs

- Hardware description languages (HDLs) are languages used to describe and model the operation of digital circuits.
  - You can use an HDL to describe a circuit.
  - You can also use an HDL to describe how to stimulate the circuit and check its response.
- Simulation of the above requires a logic simulator.

Background on HDLs

- An HDL circuit description may be used as an input to a synthesis tool. Such a tool transforms the HDL description into a representation that may be physically implemented (transistors, or logic gates…)
- When a human does this, it is called logic design.
- When a machine does this, it is called synthesis.
- Synthesis is algorithmic “design”.

My Bad Analogy

- Programming languages are languages used to describe and model behaviors and algorithms to be executed by a processor.
  - You can describe a behavior or algorithm.
  - You could also describe how to exercise your behavior / algorithm and check its response.
- It is possible to methodically simulate your behavior or algorithm and check its response.

My Bad Analogy

- A program description may be used as an input to a compiler. Such a tool transforms the description into a representation that may be executed by a processor (executable instructions…)
- When a human does this, it is called assembly.
- When a machine does this, it is compilation.
- Compilation is algorithmic “assembly”.

Background on HDLs

- There are a fair number of HDLs, but two are by far most prevalent in use:
  - Verilog-HDL, the Verilog Hardware Description Language, not to be confused with Verilog-XL, a logic simulator program sold by Cadence.
  - VHDL, or VHSIC Hardware Description Language and VHSIC is Very High Speed Integrated Circuit.
- In this class, we will be using only Verilog-HDL.
  - Specifically, Verilog-2001 wherever possible…

Background on HDLs

- Verilog history.
  - 1983 Gateway Design Automation released the Verilog HDL and a simulator for it.
  - 1989 Cadence acquired Gateway.
  - 1990 Cadence separated the Verilog HDL from their simulator product, Verilog-XL, releasing the HDL into the public domain, guarded by OVI.
  - 1995 IEEE adopted Verilog as standard 1364.
  - 2001 IEEE ratified 1364-2001 (Verilog-2001).

Background on HDLs

- VHDL history.
  - 1983 VHDL was developed under the VHSIC program of the Department of Defense.
  - 1987 The IEEE adopted the VHDL language as standard 1076. The DOD mandates that all digital electronic circuits be described in VHDL.
  - 1993 The VHDL language was slightly revised into what is often referred to as VHDL93 (versus the previous VHDL87).

Let's Fight!

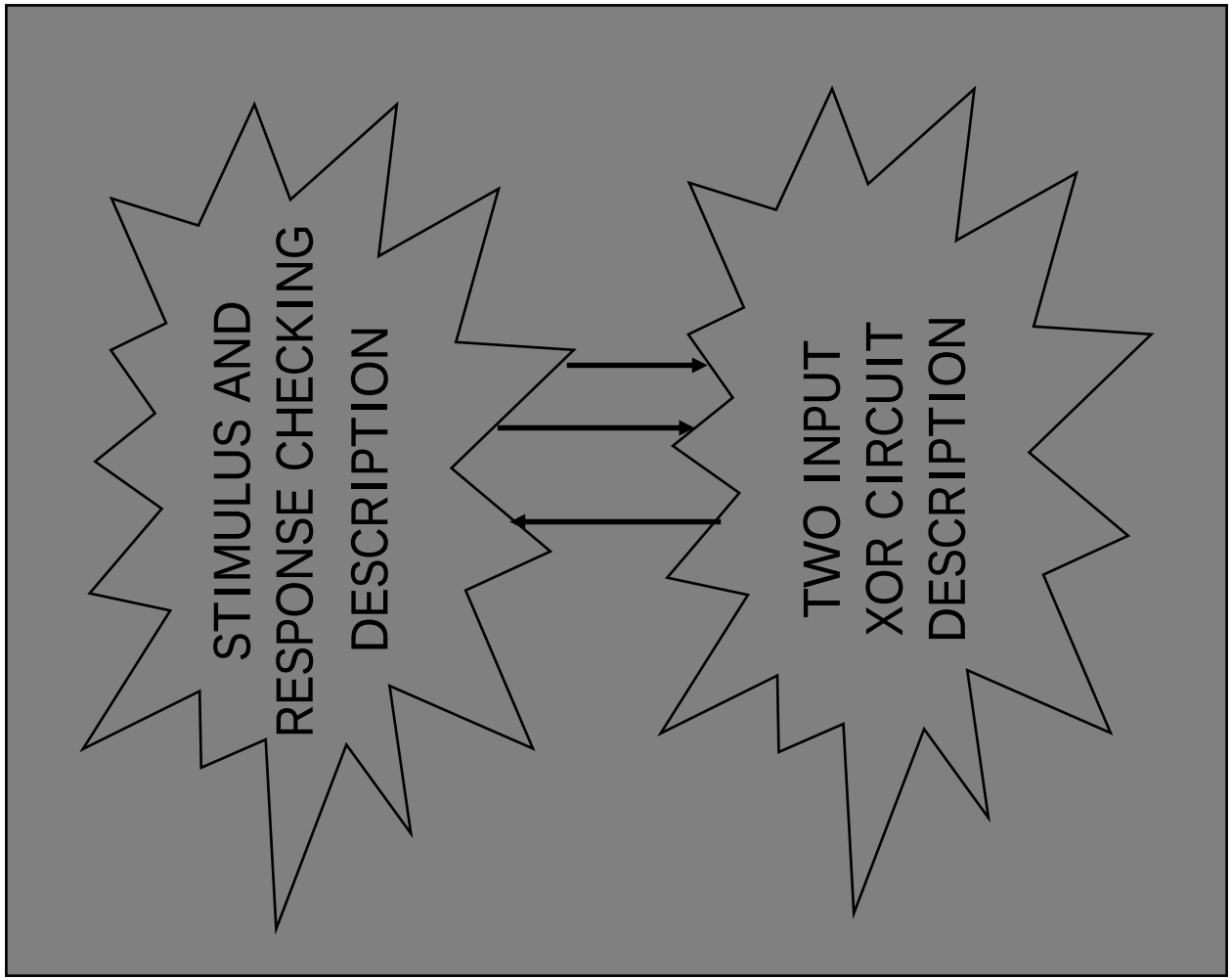
- Which is "better" Verilog or VHDL?
  - Both are adequate for our purposes…
  - What you use in industry may be dictated by company preference or government requirement.
  - VHDL may be more powerful but very rigid.
    - Very Hard and Difficult to Learn?
    - For beginners, do you want to spend your time in logic design, or fighting the language?
  - Verilog may be easier, but you can hang yourself if you are not careful...

Usage

- Verilog may be used to model circuits and behaviors at various levels of abstraction:
  - Transistor. LOW
  - Gate.
  - Logic.
  - Behavioral.
  - Algorithmic. HIGH
- For design with FPGA devices, transistor and gate level modeling is not appropriate.

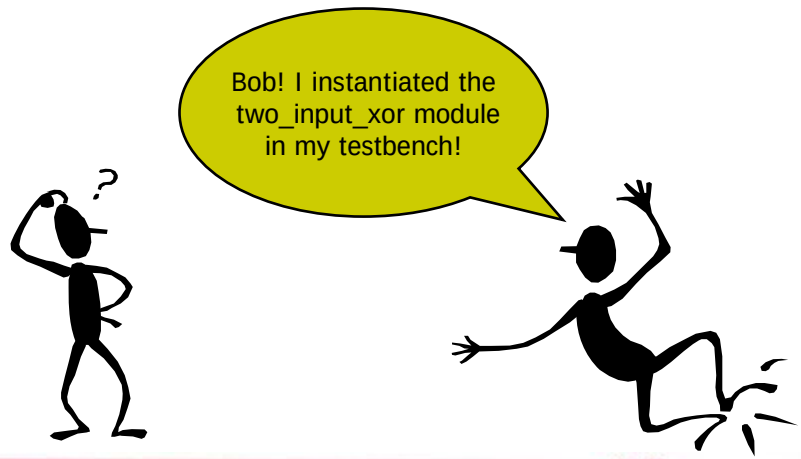
Usage

testbench.v

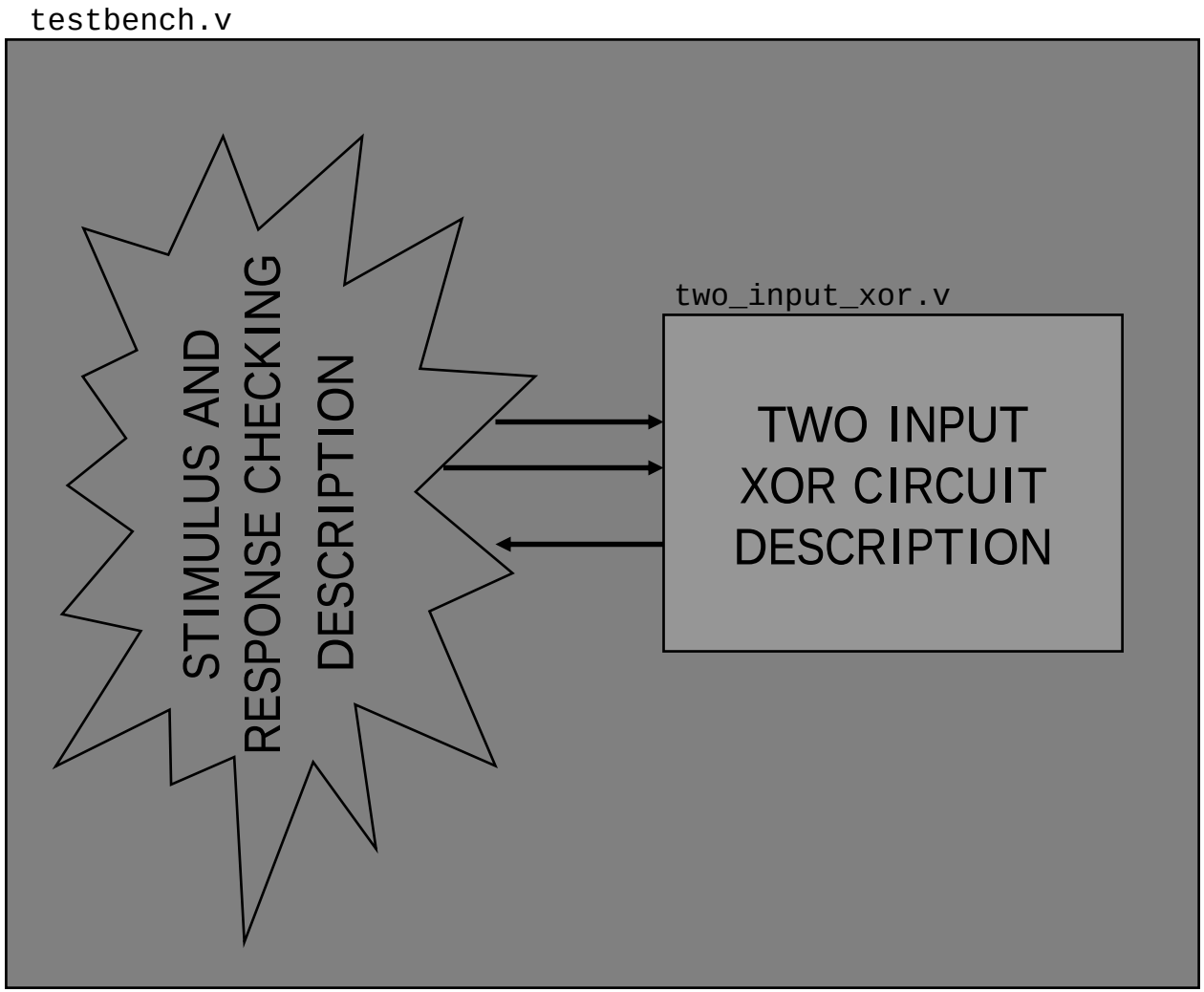


Usage

- The act of using a module within another module (as a sub-module) is called instantiation.
- This permits hierarchical design and the re-use of modules.



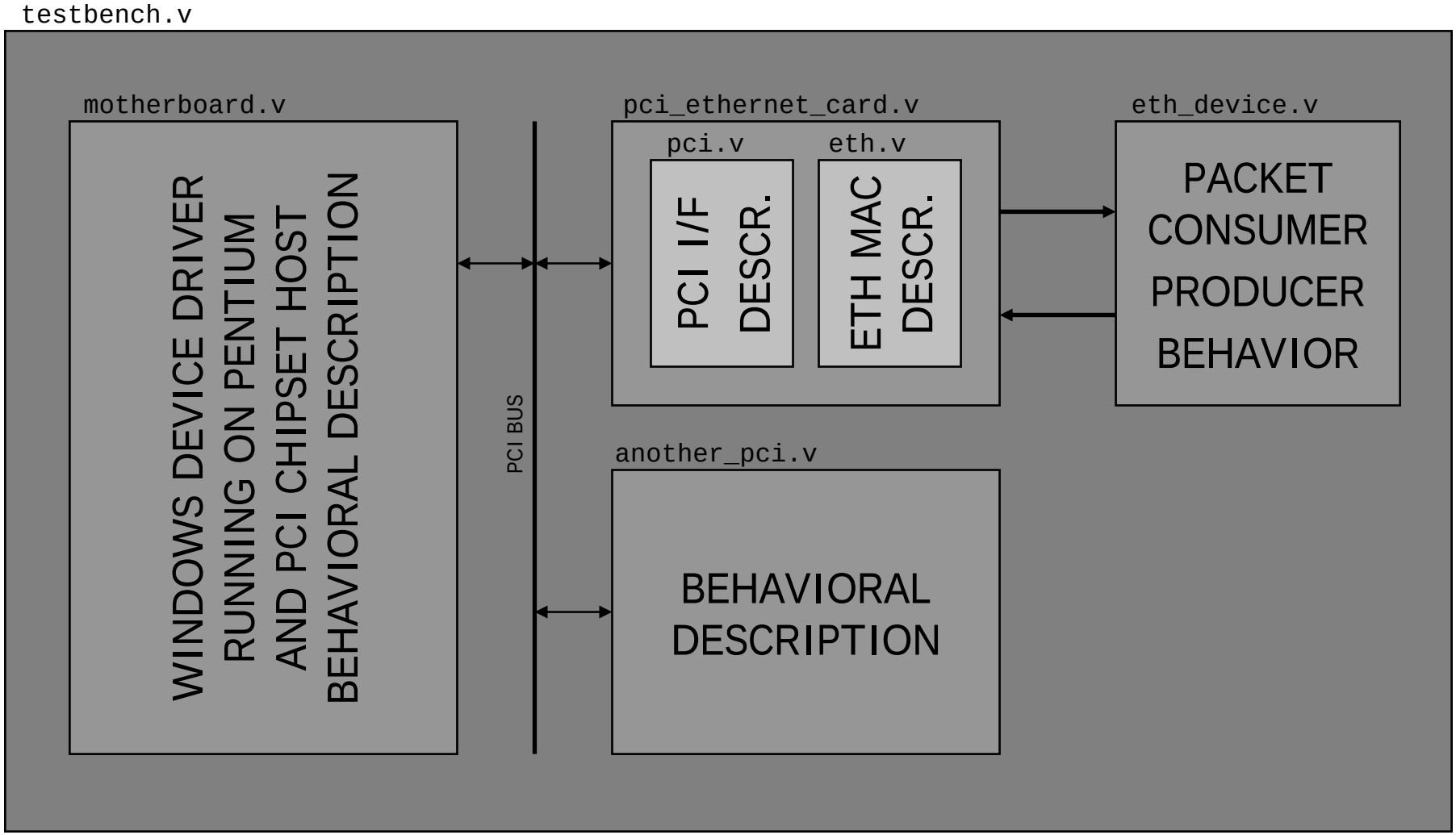
Usage


testbench.v
STIMULUS AND
RESPONSE CHECKING
DESCRIPTION
two_input_xor.v
TWO INPUT
XOR CIRCUIT
DESCRIPTION

Usage

- The stimulus and response checking descriptions may also be in separate modules, if desired.
- For synthesis purposes, you will want to keep the description of the circuit in a separate module from the stimulus and response checking.
  - The circuit can consist of any number of sub-modules.
  - When you synthesize the circuit, you will only provide the synthesis tool with the modules that correspond to the actual design.

Usage


testbench.v
motherboard.v
WINDOWS DEVICE DRIVER RUNNING ON PENTIUM AND PCI CHIPSET HOST BEHAVIORAL DESCRIPTION
PCI BUS
pci_ethernet_card.v
pci.v
PCI I/F DESCR.
eth.v
ETH MAC DESCR.
eth_device.v
PACKET CONSUMER
PRODUCER
BEHAVIOR
another_pci.v
BEHAVIORAL DESCRIPTION

Data Values

- For our logic design purposes, we'll consider Verilog to have four different bit values:
  - 0, logic zero.
  - 1, logic one.
  - z, high impedance.
  - x, unknown.

Data Values

- When specifying constants, whether they be single bit or multi-bit, you should use an explicit syntax to avoid confusion:
  - 4'd14 // 4-bit value, specified in decimal
  - 4'he // 4-bit value, specified in hex
  - 4'b1110 // 4-bit value, specified in binary
  - 4'b10xz // 4-bit value, with x and z, in binary
- The general syntax is:
  - {bit width}'{base}{value}

Data Types

- There are two main data types in Verilog.
  - Wires.
  - Regs.
- These data types may be single bit or multi-bit.

Data Types

- Wires are “continuously assigned” values and do not “remember”, or store, information.
- Wires may have multiple drivers assigning values.
- When multiple drivers exist, the simulator will resolve them into a single value for the wire.
- Every time a driver changes its output value, the resulting wire value is re-evaluated.
- This behaves much like an electrical wire...

Data Types

- Regs are “procedurally assigned” values and “remember”, or store, information until the next value assignment is made.
  - This can be used to model combinational logic.
  - This can be used to model sequential logic.
- The name “reg” does **not** mean it is a register!
- A reg may be assigned by multiple processes.
- Other reg varieties include integer, real, and time.

Modules and Ports

- Consider a top level module declaration:

```
module testbench;
  // Top level modules do not have ports.
endmodule
```

- Consider a module declaration with ports:

```
module two_input_xor (
  input  wire in1,
  input  wire in2,
  output wire out
  );
  // We'll add more later…
endmodule
```

Modules and Ports

- Ports may be of type {input, output, inout} and can also be multiple bits wide.

```
module some_random_design (
  input  wire       fred,  // 1-bit input port
  input  wire [7:0] bob,   // 8-bit input port
  output wire       joe,   // 1-bit output port
  output wire [1:0] sam,   // 2-bit output port
  inout  wire       tom;   // 1-bit bidirectional
  inout  wire [3:0] ky     // 4-bit bidirectional
  );

  // Some design description would be here…

endmodule
```

Port and Data Types

- An input port can be driven from outside the module by a wire or a reg, but inside the module it can only drive a wire (implicit wire).
- An output port can be driven from inside the module by a wire or a reg, but outside the module it can only drive a wire (implicit wire).
- An inout port, on both sides of a module, may be driven by a wire, and drive a wire.

Ports and Data Types

- Data type declaration syntax and examples:

```
module some_random_design (
  input  wire       fred,  // 1-bit input port
  input  wire [7:0] bob,   // 8-bit input port
  output wire       joe,   // 1-bit output port
  output wire [1:0] sam,   // 2-bit output port
  inout  wire       tom;   // 1-bit bidirectional
  inout  wire [3:0] ky     // 4-bit bidirectional
  );

  // In the above port declarations, note that you
  // can declare output ports to be of type reg.
  // Below are declarations of more regs and wires.

  wire       dork;         // 1-bit wire declaration
  wire [7:0] hoser;        // 8-bit wire declaration
  reg        state;        // 1-bit reg declaration
  reg  [7:0] lame;         // 8-bit reg declaration

endmodule
```

Instantiation

- Here is how you do it:

```
module testbench;
  wire    sig3;                 // wire driven by submodule
  reg     sig1, sig2;           // regs assigned by testbench

  two_input_xor my_xor (.in1(sig1), .in2(sig2), .out(sig3));

  // The format is <module> <instance_name> <port mapping>;
  // Inside the port mapping, there is a comma separated
  // list of .submodule_port(wire_or_reg_in_this_module) and
  // you include all ports of the submodule in this list.
endmodule

module two_input_xor (
  input  wire in1,
  input  wire in2,
  output wire out
  );
  // We'll add more later…
endmodule
```

Operators

- Operators in Verilog are similar to operators you might find in other programming languages.
- Operators may be used in both procedural and continuous assignments.
- The following pages present them in order of evaluation precedence.

Operators

- { } is used for concatenation.

  Say you have two 1-bit data objects, sam and bob.

  {sam, bob} is a 2-bit value from concatenation.

- {{ }} is used for replication.

  Say you have a 1-bit data object, ted.

  {4{ted}} is a 4-bit value, ted replicated four times.

Operators

- Unary operators:

  ! Performs logical negation (test for non-zero).

  ~ Performs bit-wise negation (complements).

  & Performs unary reduction of logical AND.

  | Performs unary reduction of logical OR.

  ^ Performs unary reduction of logical XOR.

Operators

- Dyadic arithmetic operators:
  * Multiplication.
  / Division.
  % Modulus.
  + Addition.
  - Subtraction.

Operators

- Dyadic logical shift operators:
  - << Shift left.
  - >> Shift right.
- Dyadic relational operators:
  - > Greater than.
  - < Less than.
  - >= Greater than or equal.
  - <= Less than or equal.

Operators

- Dyadic comparison operators:

  == Equality operator (compares to z, x are invalid).

  === Identity operator (compares to z, x are valid).

  != Not equal.

  !=== Not identical.

Operators

- Dyadic binary bit-wise operators:
  - & Bit-wise logical AND.
  - | Bit-wise logical OR.
  - ^ Bit-wise logical XOR.
  - ~^ Bit-wise logical XNOR.
- Dyadic binary logical operators:
  - && Binary logical AND.
  - || Binary logical OR.

Operators

- Ternary operator for conditional selection:

  ? :

- May be used for description of multiplexers and three state logic.

  sel ? value_if_sel_is_one : value_if_sel_is_zero

  oe ? driven_value : 1'bz

Continuous Assignment

- Continuous assignments are made with the assign statement:
- assign LHS = RHS;
  - The left hand side, LHS, must be a wire.
  - The right hand side, RHS, may be a wire, a reg, a constant, or expressions with operators using one or more wires, regs, and constants.
- You can model combinational logic with assign.

Continuous Assignment

- Two examples to complete two_input_xor:

```
// example 1

assign out = in1 ^ in2;

// example 2

wire product1, product2;
assign product1 = in1 & !in2;      // could have done all in
assign product2 = !in1 & in2;      // assignment of out with
assign out = product1 | product2;  // bigger expression
```

Continuous Assignment

- Two more examples to complete two_input_xor:

```
// example 3

assign out = (in1 != in2);

// example 4

assign out = in1 ? (!in2) : (in2);
```

- Please note – there are often many ways to do the same thing!

Continuous Assignment

- The value of the RHS is continuously driven onto the wire of the LHS.
- Whenever elements in the RHS change, the simulator re-evaluates the result and updates the value driven on the LHS wire.
- Values x and z are allowed and processed.
- All assign statements operate concurrently.

Procedural Assignment

- Procedural assignments operate concurrently and are preceded by event control.
- Procedural assignments are done in block statements which start with “begin” and end with “end”.
- Single assignments can omit begin and end.

Procedural Assignment

- Syntax examples:

```
initial
begin
     // These procedural assignments are executed
     // one time at the beginning of the simulation.
end

always @(sensitivity list)
begin
     // These procedural assignments are executed
     // whenever the events in the sensitivity list
     // occur.
end
```

Procedural Assignment

- A sensitivity list is used to qualify when the block should be executed by providing a list of events which cause the execution to begin:
  - always @(a or b) // any changes in a or b
  - always @(posedge a) // a transitions from 0 to 1
  - always @(negedge a) // a transitions from 1 to 0
  - always @* // any changes in “inputs”
- You can model combinational and sequential logic using procedural assignments.

Procedural Assignment

- Inside a block, two types of assignments exist:
  - LHS = RHS; // blocking assignment
  - LHS <=RHS; // non-blocking assignment
  - Do not mix them in a given block.
- Assignment rules:
  - The left hand side, LHS, must be a reg.
  - The right hand side, RHS, may be a wire, a reg, a constant, or expressions with operators using one or more wires, regs, and constants.

Procedural Assignment

- From *HDL Chip Design*:

  A blocking procedural assignment must be executed before the procedural flow can pass to the subsequent statement. This means that any timing delay associated with such statements is related to the time at which the previous statements in the particular procedural block are executed.

  A non-blocking procedural assignment is scheduled to occur without blocking the procedural flow to subsequent statements. This means the timing in an assignment is relative to the absolute time at which the procedural block was triggered.

Procedural Assignment

- Do I use blocking or non-blocking assignments?
  - Blocking assignments are especially useful when describing combinational logic.
    - You can “build up” complex logic expressions.
    - Blocking assignments make your description evaluate in the order you described it.
  - Non-blocking assignments are useful when describing sequential logic.
    - At a clock or reset event, all flops change state at the same time, best modeled by non-blocking assignments.

Procedural Assignment

- In procedural assignments, you may also use if-else and various types of case statements for conditional assignments.
- You also can make use of additional timing control like wait, #delay, repeat, while, etc…
- While powerful and flexible, this grows confusing so let's look at some simple examples of using procedural assignments.

Procedural Assignment

- Examples to complete two_input_xor using procedural assignment. This assumes the module output, “out” is declared as type reg.

```
// example 1
always @(in1 or in2)        // Note that all input terms
begin                       // are in sensitivity list!
     out = in1 ^ in2;       // Or equivalent expression...
end

// example 2
always @(in1 or in2) out = in1 ^ in2;
// no begin-end pair needed for single statements
```

Procedural Assignment

- More examples:

```
// example 3
always @* out = in1 ^ in2;
// use of wildcard prevents sensitivity list errors

// example 4
always @*
begin
     if (in1 == in2) out = 1'b0;  // very complex decision
     else out = 1'b1;             // logic is possible
end
```

Procedural Assignment

- More examples:

```
// example 5
always @*
begin
     case ({in2, in1})      // Concatenated 2-bit selector
       2'b01:   out = 1'b1;
       2'b10:   out = 1'b1;
       default: out = 1'b0;
     endcase
end
endmodule
```

Procedural Assignment

- Sequential logic example. What is this?

```
module counter (
  input  wire       clk,
  input  wire       rst,
  input  wire       ce,
  output reg  [7:0] out
  );

  always @(posedge clk or posedge rst)
  begin
    if (rst) out <= 0;
    else if (ce) out <= out + 1;
  end

endmodule
```

Delay Control

- You can add delay to continuous assignments.
- assign #delay LHS = RHS;
  - The #delay indicates a time delay in simulation time units; for example, #5 is a delay of five.
  - This can be used to model physical delays of combinational logic.
- The simulation time unit can be changed by the Verilog “`timescale” directive.

Delay Control

- You can add control the timing of assignments in procedural blocks in several ways:
  - Simple delays.
    - #10;
    - #10 a = b;
  - Edge triggered timing control.
    - @(a or b);
    - @(a or b) c = d;
    - @(posedge clk);
    - @(negedge clk) a = b;

Delay Control

- You can add control the timing of assignments in procedural blocks in several ways:
  - Level triggered timing control.
    - wait (!reset);
    - wait (!reset) a = b;

Delay Control

- Generation of clock and resets in testbench:

```
reg rst, clk;
initial              // this happens once at time zero
begin
     rst = 1'b1;     // starts off as asserted at time zero
     #100;           // wait for 100 time units
     rst = 1'b0;     // deassert the rst signal
end
always               // this repeats forever
begin
     clk = 1'b1;     // starts off as high at time zero
     #25;            // wait for half period
     clk = 1'b0;     // clock goes low
     #25;            // wait for half period
end
```

System Tasks

- The $ sign denotes Verilog system tasks, there are a large number of these, most useful being:
  - $display("The value of a is %b", a);
    - Used in procedural blocks for text output.
    - The %b is the value format (binary, in this case…)
  - $finish;
    - Used to finish the simulation.
    - Use when your stimulus and response testing is done.
  - $stop;
    - Similar to $finish, but doesn't exit simulation.

Suggested Coding Style

- Write one module per file, and name the file the same as the module. Break larger designs into modules on meaningful boundaries.
- Always use formal port mapping of sub-modules.
- Use parameters for commonly used constants.
- For this course, do not write modules for stuff that is easy to express (multiplexers, flip flops).
- Be careful to create correct sensitivity lists.

Suggested Coding Style

- Don’t ever just sit down and “code”. Think about what hardware you want to build, how to describe it, and how you should test it.
- You are not writing a computer program, you are describing hardware… *Verilog is not C!*
- Only you know what is in your head. If you need help from others, you need to be able to explain your design -- either verbally, or by detailed comments in your code.

Synthesis Considerations

- As you may now appreciate, Verilog provides a wide variety of means to express yourself.
- Some modules you create, like test benches, are never intended to be synthesized into actual hardware.
- In these types of modules, feel free to use the complete and terrible power of Verilog.

Synthesis Considerations

- For modules you intend on synthesizing, you should apply a coding style to realize:
  - Efficiency.
  - Predictability.
  - Synthesizability.

Synthesis Considerations

- Efficiency is important, for both performance and cost concerns.
  - Use multiplication, division, and modulus operators sparingly.
  - Use vectors / arrays to create “memories” only if you are sure the synthesis tool does it properly.
  - Case may be better than large if-else trees.
  - Keep your mind focused on what hardware you are implying by your description.

Synthesis Considerations

- Predictability is important.
  - Predictability of what hardware may be created by what you describe.
  - Predictability of hardware behavior.
    - Hardware behavior will match your description.
    - No dependency on event ordering in code.
  - Understanding of your description and how it may map into hardware is important when you are debugging the physical implementation.

Synthesis Considerations

- Not everything you can write in Verilog can be turned into hardware by a synthesis tool.
  - Initial blocks have no hardware equivalent.
  - Most types of timing control have no equivalent.
  - There is no hardware for comparisons to z and x. Or assignments to x...
  - Real world electrical considerations -- Verilog will let you describe and simulate multiple non-tristateable drivers on a wire. Fire!

Synthesis Considerations

- Realize that synthesis is machine design.
  - Based on various algorithms.
  - Do you think they included every one known to man?
  - Don’t ever assume, the machine is not a mind reader.
  - Garbage in, garbage (or errors) out.
- Read the manual for your particular synthesis tool to understand what it can and cannot do.
- Carefully read synthesis warnings and errors to identify problems you may be unaware of.

FPGA Considerations

- There are four major considerations to keep in mind while creating synthesizable code for FPGA devices.
  - Understand how to use vendor specific primitives.
  - Code for the resources in the FPGA architecture.
  - Use static timing analysis with design constraints.
  - I/O insertion.

FPGA Considerations

- FPGA vendors usually provide a library of primitive modules that can be instantiated in your design.
- These primitives usually represent device features that cannot be efficiently described in an HDL, or cannot be inferred by synthesis.
- For Xilinx designs, this library is called the “unisim” library.

FPGA Considerations

- Examples of unisim library components:
  - Large memories, called BlockRAM.
  - Smaller memories, called DistributedRAM.
  - I/O buffers, global clock buffers, three-state buffers.
  - Clock management circuits, many simple gates.
- It is possible to create designs using only the unisim library -- using Verilog as a tool to enter a text-based schematic design!
- Consult the Xilinx Library Guide for more info.

FPGA Considerations

- FPGA devices typically have look up tables for implementing combinational logic, and D flip flops and D latches for sequential logic, in addition to a few other resource types...
- You won't find other types of sequential elements, or weird RAMs -- avoid describing things that will be implemented inefficiently or require sequential elements to be built from combinational logic.

FPGA Considerations

Subject: XST Error: It's interesting and surprising!

An interesting problem occurred when I used "more than three" signals in always block sensitivity list. Same code is successfully compiled and simulated. What kind of problem is this? I put the code below.

ERROR:XST:1468 - dummy.v line 25: Unexpected event in always block sensitivity list.

```
always @(posedge a or posedge b or posedge c or posedge d)
begin
  if (a)      z=~z;
  else if (b) z=~z;
  else if (c) z=~z;
  else if (d) z=~z;
end
```

Subject: Re: XST Error: It's interesting and surprising!

What the hell is this? A quad-clock toggle flip-flop?

Subject: Re: XST Error: It's interesting and surprising!

Your results are not surprising. Although your code is legal Verilog and is able to simulate, it's not synthesizable by XST. What type of flip-flop would you expect XST to infer from the "always @(posedge a or posedge b or posedge c or posedge d)" construct? You need to re-think your code!

FPGA Considerations

- There are a limited number of global clock buffers in most FPGA devices.
  - We will cover why these are important.
  - Don’t use multiple clocks. For this course, the synthesizable designs are allowed only one clock.
  - The synthesis tool should recognize clock signals and automatically insert global clock buffers.
  - You can manually insert global clock buffers by instantiation from the unisim library.

FPGA Considerations

- Most designs have frequency and I/O timing requirements. FPGA vendors provide analysis tools called static timing analyzers.
  - Don't use multiple clocks. For this course, the synthesizable designs are allowed only one clock.
  - Don't use latches. Latches greatly complicate static timing analysis. For this course, you should not have ANY latches in your design at all.
  - Don't use combinational feedback loops.

FPGA Considerations

- Where do these evil latches come from?
  - You directly instantiated them or inferred them in your code on purpose.
  - You inferred them in your code by accident.
    - For procedural blocks modeling combinational logic, all the regs and wires on RHS must appear in sensitivity list.
    - For procedural block modeling combinational logic, you have incompletely specified if-else or case statements.
    - For modeling sequential logic, you have more than a single clock and asynchronous control in the sensitivity list.

FPGA Considerations

- Templates for sensitivity lists:
  - always @(posedge clk)
    - Synchronous logic without asynchronous reset.
    - Use of “negedge clk” is legal, but not in this class.
  - always @(posedge clk or posedge rst)
    - Synchronous logic with asynchronous reset.
    - Use of “negedge clk” is legal, but not in this class.
    - Can use “negedge rst_n” for active low reset.

FPGA Considerations

- Templates for sensitivity lists:
  - always @(sig_1 or sig_2 or … or sig_n)
    - Combinational logic description; whenever an input changes, the output must be evaluated.
    - Every signal used as an “input” in description must appear in the sensitivity list.
  - always @*
    - A very welcome Verilog-2001 enhancement!
    - Combinational logic description, process is sensitive to any signal used as an “input” in the description.

FPGA Considerations

- I/O insertion is the process where the synthesis tool looks at the ports of the top-level module that is synthesized and decides what FPGA primitives to insert to interface with the outside world…
- FPGA devices typically have:
  - Input buffers (IBUFs, in unisim library).
  - Output buffers (OBUFs, in unisim library).
  - Three-state output buffers (OBUFTs in unisim library).

FPGA Considerations

- Compare this to the Verilog port types:
  - input ports map into IBUFs.
  - output ports map into OBUFs.
  - inout ports map into… what?
- How do you create bidirectional functionality at a chip boundary?