Computer Access

• Sun workstations may be accessed remotely in the open computing lab (JD1622C) via Hummingbird.

• Directions are available from computer support (JD 1109 or 1112) and will be available on the course web site.
Turn ‘em OFF!

It seriously annoys your instructor, not a Good Thing.
EDA : Electronic Design Automation

The process of using computer-based software systems to design very large-scale integrated (VLSI) circuits.

All modern integrated circuits are designed with EDA tools.

This course uses Verilog for hardware description and the NC Verilog simulator from Cadence for simulation.
WHAT IS A HARDWARE DESCRIPTION LANGUAGE (HDL)?

• COMPUTER-BASED LANGUAGE
  • Syntactically similar to programming languages
• MODEL, REPRESENT, AND SIMULATE DIGITAL HARDWARE
  • HARDWARE CONCURRENCY
  • PARALLEL ACTIVITY FLOW
  • SEMANTICS FOR SIGNAL VALUE AND TIME

• SPECIAL CONSTRUCTS AND SEMANTICS

EXAMPLES

HILO2, HARDWARE C, AHDL (U of A 1970), ISPS (CMU, 1971)

VERILOG,
VHDL (Very High Speed Integrated Circuit Hardware Description Language)
Current HDL Usage

• All current digital IC development is done with either VHDL or Verilog.
• Other scripting languages (perl, make, Tk/tcl) may be used to supplement them.
• All FPGA, ASIC and 3rd party tool vendors (Synopsys, Cadence, Mentor Graphics, etc.) support both Verilog and VHDL.
Why Use an HDL?

• As the complexity of systems increases, it becomes more difficult to design directly on hardware.

• Exploring different design options is easier and cheaper because you only need to change the HDL description. It is much easier to change the description than to reconfigure the prototype.

• You can try different design options quickly and easily. The time to fix a design problem is reduced and so is the cost.
Test Early and Often

Finding defects early is way cheaper than finding them after they are out in the field. Image © Texas Instruments.
Transistor Density

• First microprocessor (circa 1971) had 2,300 transistors.

• It was done by a team of four in about four months.

• That’s less than 150 transistors per engineer per month.
Density Increase

- Current state of the art in processors is approaching a billion transistors.
- At 150 transistors per engineer per month, something like 6 million man-months.
- Design at the gate level: about 10 transistors/gate.
- A team of 100 could then get a processor done in a mere 5,000 years.
Area vs. Operating Frequency

One HDL encoding can produce many designs through logic synthesis.

Image © John Cooley “The Great ESDA Shootout”
HDL Design Flow

1. Design Specification
2. HDL Coding
3. HDL Simulation
4. Simulation Passed?
   - Yes: Logic Synthesis
   - No: Simulation Vectors

Diagram:
- Design Specification ➔ HDL Coding ➔ HDL Simulation ➔ Simulation Passed?
- Simulation Passed? ➔ Logic Synthesis
- Simulation Passed? ➔ No ➔ Simulation Vectors

Popularity of Verilog HDL

- It is a general-purpose hardware description language that is easy to learn and use.
- It is similar to the C programming language.
- It allows for different levels of abstraction to be mixed in the same model.
- Most popular logic synthesis tools support Verilog HDL.
- All fabrication vendors provide Verilog HDL libraries for post logic synthesis simulation.
- The Programming Language Interface (PLI) allows the user to write custom C code to interact with the internal data structures of Verilog.
Typical Design Flow

- The behavioral description is manually converted to a Register Transfer Level (RTL) description in an HDL.
- Designer has to describe the data flow that implements the desired digital circuit.
- Logic synthesis tools convert the RTL description to a gate-level netlist (a description of the circuit in terms of gates and connections between them).
- The gate-level netlist is input to an Automatic Place and Route tool to create a layout.
- The layout is verified then fabricated on chip.
Detailed Routing

Post-layout STA

Timing OK?

Design Rule Check

Passed?

Yes

Back to Salt Mine

Tapeout

Collect Bonus, Vacation

Yes

Back to Salt Mine
# Terms and Definitions

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hard ware description language (HDL)</strong></td>
<td>A programming language that can describe the functionality and timing of hardware circuits.</td>
</tr>
<tr>
<td><strong>Simulator</strong></td>
<td>Software which reads the HDL and emulates the hardware described by the HDL.</td>
</tr>
<tr>
<td><strong>Bottom-Up design flow</strong></td>
<td>A design methodology in which you build the low-level components first and then connect them to make large systems.</td>
</tr>
<tr>
<td><strong>Top-down design flow</strong></td>
<td>A design methodology in which you define the system at a very high level of abstraction and refine it at the lower levels of abstraction by partitioning the design into logical, functional, or physical units.</td>
</tr>
</tbody>
</table>
Key Features of HDL’s

- Typically an HDL contains some high level programming language constructs along with constructs to describe the connectivity of hardware design.

- An HDL allows you to describe the design at various levels of abstraction using structural or behavioral constructs.

- An HDL allows you to describe the functionality of hardware along with its timing constraints.

- Concurrency is the ability to perform multiple tasks at the same time. Typically, programming languages are not concurrent, but in hardware, a number of operations happen at the same time. Thus, an HDL must be concurrent.

- Typically, programming languages have no concept of time. In hardware there are delays associated with going from an input to an output. An HDL allows you to model these delays because it has a concept of time.
Simulation Algorithms

• Time Driven

— Each circuit element is evaluated at each time point, producing a new circuit state at that point.

— Inefficient, because at any time only 2 to 10 percent of elements in a circuit need to be evaluated.

— The change of the value of any of the variable will result in the calculation of the logical value at all points.
Simulation Algorithms

- **Event driven**
  - Changes in circuit state are recorded. Only those elements that might cause a change in circuit state, during time, are simulated. The simulation propagates values forward, through the circuit, in response to input pin events.
  
  - Most practical and widely used simulation algorithm.
  
  - Efficient, because it “evaluates when necessary.”

Assume that at time “t = T”: A=1, B=0 and C = 1, Now if at time “t=T+1” only A changed to “0” this will require the calculation of only the output of this gate.
Simulation Algorithms

- **Demand driven**
  - Further refines “evaluates when necessary.”
  - Evaluates nets and gates only when their values are needed to provide simulation output.
  - Propagates requests for simulation values, backwards through circuit and time.

  Assume that at time “t = T”: W= 1. Now, if at time “t=T+1” W does not change then the changed of any of the other variable will not require any calculation.
Simulation Algorithms

• Cycle-based

Collapse all logic between registers.
Only calculated final values.
Intermediate timing and delay data are lost.
Fastest, but not suitable for all simulations.
Cycle Based Algorithm

CLK
Q1
S1
S2
S3
Q2
Cycle Based Algorithm

S1 = Q1 \& 1
S2 = S1 \| 0
S3 = S2 ^ 0
\implies S3 = Q1

Only calculate final value at clock edge, not all intermediate values at the time at which they transition. Timing may be verified with a static timing analyzer.
Combined Algorithms

• Perfectly synchronous circuits may be effectively simulated with cycle-based simulation combined with STA.
• Real designs tend not to be totally synchronous (though ECE526 designs will be).
• Sophisticated simulators combine algorithms: parts that are suitable for cycle-based use cycle-based, parts that are not use event-driven.
Time Wheel in Event-Driven Simulation

- When the simulator compiles its data structures, it creates the initial queues (time=0) for the time wheel based on the HDL.
- The simulation starts when the current simulation time is 0.
- When all the events scheduled at time 0 are executed, the simulation time advances to the next time step.
- Events at each time step can append new events to event queues at a later time.
Time Wheel in Event-Driven Simulation

- The time wheel can only go forward.
- Time advances only when every event scheduled at that time is executed.
Verilog Event Queues

Active Events
- Blocking assignments
- Evaluate RHS of nonblocking assignments
- Continuous assignments
- $\text{display command execution}$
- Evaluate inputs and change outputs of primitives

Inactive Events
- $\#0$ blocking assignments

Nonblocking Events
- Update LHS of nonblocking assignments

Monitor Events
- $\text{$\$$monitor command execution}$
- $\text{$\$$strobe command execution}$
- Other specific PLI commands

These events may be scheduled in any order

Different Levels of Abstraction

• Architectural/Algorithmic
• Register Transfer level (RTL)
• Gate
• Switch
Different Levels of Abstraction

• Architectural/Algorithmic
  – The system is described in the terms of the algorithms it performs.
  – The aim is to study the data flow of the system and potential bottlenecks.

• Register Transfer Level (RTL)
  – Describes the flow of data and control signals within and between functional blocks.
  – Schedules assignments at clock edges.
Different Levels of Abstraction

- The Register Transfer Level, (RTL), is a design level of abstraction. RTL refers to coding that uses a subset of the HDL language.

- RTL is the level of abstraction below behavioral and above structural. Events are defined in terms of clocks and certain behavioral constructs are not used.

- A RTL description implies that the data in the registers you are describing "transfers" in the manner specified by your clocks. RTL synthesis must preserve the transfer of data between registers as specified by your clocks.
Different Levels of Abstraction

• Gate
  – Interconnection of logic elements (or gates) to check functionality, performance, or the timing of design.

• Switch
  – Describes logic behavior of transistor circuits.
  – Evaluates conflicts caused by bidirectional pass transistors, signal strengths of multiple elements driving a net, and so on.
Design Methodologies

• Bottom-up design flow
  – Start with small functions
  – Build up a hierarchy, eventually creating the top level design.

• Top-down design flow
  – Start with an overall design specification.
  – Add detailed functions to make it work.
Verilog is Flexible

- There are many ways to code a Verilog project.
- Standards have developed over the years. Examples presented here frequently show variations on what CAN be done and do not always correspond to best industrial practices.
Always Use Synchronous Design

NEVER do anything like this!
Synchronous Count

CLK
Q0
Q1
Q2

0 1 2 3 4 5 6
Ripple Clock Behavior
Synchronous Designs

• Synchronous designs are more robust.
• Synchronous designs are testable.
• Synchronous designs are portable.
• Synchronous designs have better noise immunity.
• http://www.elecdesign.com/Articles/Index.cfm?AD=1&ArticleID=7596
The Verilog Simulator

- It is a logic-level simulator that you can use to:
  - Determine feasibility of new design ideas.
  - Try out more than one approach to a design problem.
  - Verify functionality.
  - Identify design errors.

- Verilog started out as a verification tool. Hardware design and synthesis came later.

- NC Verilog simulator now supports all Verilog 2001 enhancements.
Invoking the Simulator

• Starting in Fall 2007, all Verilog-based courses will use NC Verilog.
  – ncverilog filename.v

• The obsolete Cadence simulator Verilog XL is still running on the workstations.

• It is invoked by
  – verilog filename.v

• Make sure you always use NC Verilog and not Verilog XL
Designing with Primitives

• First labs use Verilog built-in primitives for design.
• This is not an improvement over schematic capture.
• While it is useful for learning the functioning of tools, it is NOT the way Verilog is used for circuit design.
Verilog Primitives

• Just like drawing schematics, circuits of any arbitrary size can be made with primitive components.

• Primitive logic operators are
  – AND
  – OR
  – XOR
  – NOT

• And the inverses of these four
  – NAND, NOR, XNOR, BUF
nand (out, in1, in2, in3, in4);


out: gate output is always the first port

in1, etc.: Verilog supports gates of arbitrary width. Gate instances do not need to correlate 1:1 to physical gates.

If not otherwise declared, variables (in1, in2, etc. default to single-bit wires.
Multi-bit wires

Not needed for Lab 1, but will be needed shortly.

    wire andbus;
    wire [1:0] bus;
    and gate_1(andbus, bus[0], bus[1]);
    //gate_1 is the NAME of one particular and gate. Gate
    //names are optional but generally advisable.
module MUX2_1 (out, a, b, sel);

output out;
input a, b, sel;
wire a, b, sel, sel_, out; //note sel_ is left out of manual
Lab 1 Netlist

//The netlist
not (sel_, sel);
and (a1, a, sel_);
and (b1, b, sel);
or (out, a1, b1);
endmodule
Warning

- Verilog IS case sensitive.
- All Verilog keywords are lower case.
Case Sensitivity

• Keywords are always lower case
  – Wire A, B; //ILLEGAL
  – WIRE A, B; //ILLEGAL
  – wire A, a; //Legal but bad technique
  – wire A, B; //The advised way
  – wire a, b; //OK but may be confusing
Case Sensitivity

• Verilog is case sensitive
  – reg [3:0] count;
  – reg [15:0] COUNT;
  – reg [31:0] Count;

• Logic synthesizer and other downstream tools are not.

• Above would simulate correctly, fail in synthesis.
Lab 1 Netlist

//The netlist

    not (sel_, sel);
    and (a1, a, sel_);
    and (b1, b, sel);
    or (out, a1, b1);

endmodule
Lab 1 Schematic
How Would This Change the Netlist?
/// The netlist

not (sel_, sel);
and (a1, a, sel_);
and (b1, b, sel);
and (ab, a, b);
or (out, a1, b1, ab);
endmodule
Test Bench

- Lab 1, like all labs, has a Verilog circuit description and a Verilog test program.
- Circuit description may contain ONLY synthesizable constructs. Many Verilog constructs are not synthesizable.
- Test fixtures (test benches, same thing) are not so limited. All of Verilog is legal for test programs.
Managing Files

• It is generally good practice to
  – Have exactly one UNIX file per Verilog module.
  – Have the Verilog module name be the same as the UNIX file name.
  – Give the UNIX file a .v extension.
  – Have one directory per lab.
Directory Organization

- Design Root
  - Source Code
  - Synthesis
    - Scripts
  - Simulation
    - Logs, Reports
    - Netlists
Lab 1 Test Bench

`timescale 1 ns / 1 ns

module test_mux;

    reg a, b;
    wire out;
    reg sel;

    MUX2_1   m1(out, a, b, sel);
    initial
        $monitorb( "%d out=%b a=%b b=%b sel=%b", $time, out, a, b, sel);
Lab 1 Test Stimulus

initial begin
  #10 a=0; b=1; sel=0;
  #20 sel=1;
  #20 a=1; b=1;
  #10 sel=0;
  #20 $stop;
  #20 $stop;
  #1 $finish;
end
endmodule
More Stimulus

- The stimulus shown is a start but is not complete.
- For this lab, ALL possible combinations must be tested.
- This means all logic values (000 → 111)
- It also means all legal Verilog values.
- What are the other values an input can have?
Design Errors and Faults

• Lab manual does not entirely correspond to industry-standard nomenclature (testing and verification, pg. 12).

• Faults are manufacturing defects. “Factory test vectors” are written to enable them to be detected once the design is suitably modified for test.
Verification

• In ECE 526, we are concerned with logic verification, NOT fault detection.
• These are two distinct procedures.
• You will write test fixtures to determine the logical functioning of your designs. You will not write test vectors to enable the foundry to sort out the good chips from the bad.
Lab Report

- Lab manual page 10 has an example of good technique and bad technique.
- Both have a 2:1 mux.
- Do they work identically? Answer this question in your lab report.
Lab 1

- Tools change as time goes on.
- Not all commands/procedures in the lab manual will work exactly as written.
- Lab printer generally won’t work. Do not depend on it.
Bottom-Up Design Flow

1. Lowest level components are modeled at gate level and tested
2. Intermediate level components are modeled at the gate level and tested
3. The complete design is modeled and tested
4. Design timing, performance, and testability are analyzed
Bottom-Up Design Flow

Since bottom-up design closely follows the steps involved in breadboarding and building a system using hardware only, bottom-up design is the traditional method.

• The advantages of bottom-up design are:
  – Many designers already have the software and hardware required to design this way.
  – Designers are traditionally trained to work this way.
  – Less time is needed to implement individual pieces of the circuit.

• The disadvantages of bottom-up design are:
  – Typically, there is a poor functional view of the entire system.
  – You need more time to implement the entire system, because you must complete individual pieces first. In other words, concurrent engineering is more difficult with bottom-up design methods.
Top-Down Design Flow

1. The top-level system is modeled for functionality and performance using a high-level behavioral description.

2. Each major component is modeled at the behavioral level and the design is simulated again for functionality and performance.

3. Each major component is modeled at the gate level and the design is simulated again for timing, functionality and performance.
Top-Down Design Flow

• Top-down design begins with a Verilog functional model of the top-level system. As the individual partitions are modeled in more details, they are plugged back into this top-level function description.

• The advantages of top-down design are:
  – System analysis is done at the beginning of the design cycle. Therefore, quality and performance trade-off decisions can be made at the earliest stage.
  – Concurrent engineering is facilitated. In other word, a number of engineers can work on different parts of the design at the same time.
  – Typically, fewer design iterations are required; so time to market is reduced.
  – Increasingly large designs are easier to handle.
Top-Down Design Flow

• The disadvantages of top-down design are:
  – There is a learning curve involved, because it is not as traditional as bottom-up design.
  – It requires software and hardware capable of performing mixed-level simulations.
Verilog HDL and NC Verilog

• Verilog HDL
  - Hardware description language that allows you to describe circuits at different levels of abstraction and to mix any level of abstraction in the design.
  - Verilog can be applied to all aspects of electronic design, including testing and verification. Therefore, only one language needs to be learned.

• NC Verilog Software
  High speed simulator that reads Verilog HDL and simulates the description to emulate the behavior of real hardware.
Key Language Features

Verilog Module

• Modules are the basic building blocks in the hierarchy.
• Verilog descriptions or models are placed inside modules.
• Modules may represent:
  - A physical block like an IC or ASIC cell.
  - A logical block like the ALU portion of a CPU design.
  - Smaller design components.
  - The complete system.
Verilog Module

module SN74LS74
    
endmodule

module DFF
    
endmodule

module ALU
    
endmodule
Module Ports

- Module ports are equivalent to the pins in hardware.
- Modules communicate with the outside world through ports.
- You list a module's ports in parentheses after the module name.
- You declare ports to be *input*, *output* or *inout* (bidirectional) in the module description.
module DFF (D, CLK, PRE, CLR, Q, QB);
input D, CLK, PRE, CLR;
output Q, QB;
endmodule
Things to Note

• Verilog keywords (module, endmodule, input, output) are all lower case.
• There is no defined order for ports.
• Input and output statement order does not need to correspond to port order.
• Ports declared together must be all the same type: all single bit in the example.
• Continuation character not needed (module declaration line).
Modifying a Module

How would the previous module change for a JK flipflop?
JK Flipflop Ports

module JKFF(J, K, CLK, PRE, CLR, Q, QB);
input J, K, CLK, PRE, CLR;
output Q, QB;
....
....
....
endmodule
Module Instances

• You can create a larger system or component by listing instances of other modules and connecting those instances by their ports.

• In the following example, REG4 contains four instances of the module DFF. Note that each module has its own instance name (d0, d1, d2 and d3). The instance name gives a unique name to every object, and is used so that the internals of the instance can be examined.

• Notice that the order of the ports in the instance follows the order in the module definition.

• Instantiating a module is not the same as calling a routine; each instance is a complete, independent, and concurrently active copy of the module.
module REG4(d, clk, clr, q, qb);
output [3:0] q, qb;
inout [3:0] d;
inout clk, clr;

endmodule
Simpler DFF

Registers used in data flow paths rarely include RST or QB ports.

```
module DFFS(CLK, D, Q);
input CLK, D;
output Q;
.....
endmodule
```
Hooking Up Instances

Data In → DFFS → DFFS → DFFS → DFFS → Data Out

Clock
module shift4(Clock, Data_In, Data_Out);

input Clock, Data_In;
output Data_Out;
wire Delay1, Delay2, Delay3;

DFF D0 (Clock, Data_In, Delay1);
DFF D1 (Clock, Delay1, Delay2);
DFF D2 (Clock, Delay2, Delay3);
DFF D3 (Clock, Delay3, Data_Out);
endmodule
module tb_shift4;
reg clk, data_in;
wire data_out;
shift4 uut(clk, data_in, data_out);
initial clk = 1’b0;
always #10 clk = ~clk;
initial begin
    #5 data_in = 1’b1;
    forever #20 data_in = ~data_in;
end
endmodule
Basic Verilog Constructs

- White space (blank, tab, new line) is ignored.

- Comments follow “C” conventions:
  - /*Multi line comments can be done with star-slash.*/
  - //Single line comments are double slashes.
  - /*This does not work./*Why would you want to do it,*/ anyhow?*/
System Tasks for Simulation

- $display
- $write
- $monitor
- $strobe
- $time
- $stop
- $finish
$display

- Displays the value of signals or variables in a design or test fixture.
- Usage: $display(s1, s2, s3,….sn);
- s1, s2, etc. are signals in the uut or variables in a test fixture.
- Uses string formatting similar to C.
- Includes new line character by default.
$display string formatting

- %d  display a variable in decimal
- %b  display a variable in binary
- %s  display a string
- %c  display an ASCII character
- %h  display a variable in hex
- %g  display a real number in scientific notation or decimal, whichever is shorter.
$display Examples

• $display(“I want a cookie.”);
  – I want a cookie.
• $display(“Count = %d”, count);
  – Count = 6
• $display(“Count = %b”, count);
  – Count = 0110
• $display($time, “ count = %b, state = %h”, count, state);
  – 80 count = 0110, state = fa
$time

- $time calls out the simulation time.
- It has nothing to do with the time of day.
- Internally, it is represented by a 64-bit number, so it can keep track of a long simulation.
- Example: $display($time);
  - 80
- 80 time units have passed since the simulation started.
$write

• $write is exactly the same as $display except that it does not implicitly include a new line character.

• Example:
  – $write(“I want a cookie. ”);
  – $write(“I want a cookie! \n”);

• Output:
  – I want a cookie. I want a cookie!
$monitor

- Monitors a signal when its value changes.
- Usage: $monitor(p1,p2,p3…pn);
- p1, p2, etc. are signals in the uut or variables in a test fixture.
- $monitor displays the values of all objects in its list whenever any one of them changes.
Using $monitor

- Same formatting as $display
- Only ONE $monitor can be active at a time. Thus
  - $monitor("clock = %b reset = %b", clk, rst);
  - $monitor("state = %h", state);
- will result in only "state" being monitored.
Difference Between $monitor and $display

- $monitor is continuous.
- $display only runs once.
- Possible usage: display header, monitor changing data:
  - $display("time\tstate\tcound");
  - $monitor("$time,%h \t%h", state, count);
$strobe

- Very similar to $display.
- The difference is that $strobe is always the last task to be executed at any time specification whereas $display may not be.
$strobe$ and $display$ Example

always @(posedge clk) begin

    a = b;
    c = d;
end

always @(posedge clk) begin

    $display("a = %b, c = %b", a, c);
    $strobe("a = %b, c = %b", a, c);
end

They COULD, but probably won’t, give different results.
Scope of Variables

• All system task examples so far have not had any scope operator.
• They would all operate on variables in the test fixture.
• Internal variables can also be displayed, monitored, etc. by scoping down to where they are.
Monitoring an Internal Variable

• Change scope by stringing together hierarchical names with periods.

• Example: UUT is instance of design DESNAME
  – DESNAME UUT(CLK, X, Y, Z);
  – $display ("a = %b, X = %b", UUT.a, X);

• The value of the variable “a” that is in the uut will be shown, as will the variable X in the test fixture.
Looking Inside the UUT

Monitor this point

$monitor("AND1 Output = %b", UUT.C);
Unlimited Scoping

• Big designs can have many levels of hierarchy.
• Each instance of a sub design will have an instance name.
• So you can monitor signals at any level of hierarchy by using the scoping operator and hierarchical names.

$\text{monitor(\text{TOP.CONTROLLER.SM.STATE});}$
$stop and $finish

- $stop halts simulation and puts the simulator into interactive mode. Resume simulation by typing “." at the prompt.
- $finish ends the simulation.
When to use $stop and $finish

• $stop is used to get into interactive mode.
• This makes it good for debugging in the lab.
• $finish ends the session.
• Larger projects are typically simulated in “Batch Mode.” Then they need to be terminated with a $finish. Leaving $stop in the test bench would not allow the simulator process to terminate.
Verilog Primitives

- Verilog provides basic logical functions as predefined primitives. You do not have to define this basic functionality.
- Most ASIC libraries are developed using primitives.
- They form an integral part of the bottom-up design methodology but are not used in behavioral code.
- Verilog simulates the predefined primitives with built-in algorithms.
- Behavioral simulation is far faster than gate-level simulation.
## Verilog Primitives

<table>
<thead>
<tr>
<th>Primitive Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>Logical AND</td>
</tr>
<tr>
<td>or</td>
<td>Logical OR</td>
</tr>
<tr>
<td>not</td>
<td>Inverter</td>
</tr>
<tr>
<td>buf</td>
<td>Buffer</td>
</tr>
<tr>
<td>xor</td>
<td>Logical Exclusive OR</td>
</tr>
<tr>
<td>nand</td>
<td>Logical AND Inverted</td>
</tr>
<tr>
<td>nor</td>
<td>Logical OR Inverted</td>
</tr>
<tr>
<td>xnor</td>
<td>Logical Exclusive OR Inverted</td>
</tr>
</tbody>
</table>
Primitive Pins Are Expandable

• The number of pins for a primitive gate is defined by the number of nets connected to it. Therefore, you do not need to redefine a new logical function whenever the number of inputs or outputs to that function change.

• All gates except *not* and *buf* can have a variable number of inputs, but only one output.

• The *not* and *buf* gates can have a variable number of outputs, but only one input.
Verilog Primitives

\[
\text{nand (out, in1, in2);}
\]

\[
\text{nand (out, in1, in2, in3);}
\]

\[
\text{nand (out, in1, in2, in3, in4);}
\]
Multiple-input Primitives

- AND
- OR
- XOR
- NOR
- XNOR

<table>
<thead>
<tr>
<th>Input 1</th>
<th>0</th>
<th>1</th>
<th>X</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Z</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
### More Primitives

- **AND**
- **OR**
- **XOR**
- **NOR**
- **XNOR**

Note that gates are **NOT** limited to two inputs.

<table>
<thead>
<tr>
<th>Input 1</th>
<th>0</th>
<th>1</th>
<th>X</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Another Multiple-input Primitive

- AND
- OR
- XOR
- NOR
- XNOR

Any XOR/XNOR will output an X (unknown) if any input is unknown or Hi Z.
Single-input Primitives

• Buf

• Not

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Z</td>
<td>X</td>
</tr>
</tbody>
</table>
Primitives with Control Inputs

- Bufif1
- Bufif0
- Notif1
- Notif0

Output is Z (high impedance) if control is not asserted.

Output is X if control IS asserted and input is X or Z.
Netlists of Primitives

and #gate_delay ANDGATE1 (out, in1, in2, in3);

Gate Type

Lumped Intrinsic and Fanout Delay

Gate Name

Port list, output always first

109
Time Delays

- Gates can have an associated delay
  - and #gate_delay ANDGATE1 (out, in1, in2, in3);
- The simulator can be instructed to wait before executing a line of code
  - #10 in1 <= 1'b1;
- Note the syntactical difference. In the first case, there is a delay associated with the gate instance. In the second, the simulator is instructed to wait before proceeding.
Time Delays

• Time delays are specified using \#n, where n is the amount of time in units as specified in a timescale directive. This will default to nanoseconds.

• Example: \#10 rst <= 1’b1;

• Wait for 10 ns, then set signal rst to logic 1.
Setting bits

• Lab manual is sloppy on this syntax. Don’t be that way in your code.

• Proper syntax includes the number of bits and the radix.

• Example: rst <= 1’b0;

• rst is assigned a one-bit value of logic 0.
More on Setting Bits

• As shown in the lab manual, bit variables can be set to logic 0 or logic 1 without proper formatting:
  – Example: control = 0;

• This is poor practice and DOES NOT WORK with x and z values.

• The right way:
  – control = 1’b0;
  – tribit = 1’bz;
Assignment Operators

• Both = and <= have been used in these examples.

• They are not identical in function. The difference will be covered later.

• Rule of thumb: use = with combinational code, <= with sequential.
  – Sequential is code that infers flops.
What’s This?