Pre-systhesis Simulation

- Simulation for design validation
  - Done before a design is synthesized
  - Referred to as RT level, or Pre-synthesis Simulation
  - The Required Test data: generated graphically using waveform editors, or through a testbench
- RTL level Simulation is clock accurate
  - The advantage is speed over gates/transistor-level
- Outputs of simulators:
  - Waveforms (for visual inspection)
  - Text for large designs for machine processing

Testbench in Verilog

- Generates test data and observes simulation results
  - Tests of a designed system functionality
  - Detects incompatibility of design component
- Testbench is a Verilog module that uses Verilog high-level constructs for:
    - Data Generation
    - Response Monitoring
    - Handshaking with the design
  - Each testbench instantiates a design module

Pre-systhesis Simulation

- **Actual hardware behaves differently than RTL model**.
  - Timing and delays of the parts used,
    - Nonzero delay between clock edges.
    - Output is unpredictable if the clock frequency applied is too fast for propagation delays.
- **This is typical of a pre-synthesis or high-level behavioral simulation.**
  - Potential timing problems of the hardware that are due to gate delays cannot be detected.

Design Validation

- An important task in any digital system design
- The process to check the design for any design flaws

- A design flaw due to:
  - Ambiguous Problem Specifications
  - Designer Errors
  - Incorrect Use of Parts in the Design

- Can be done by:
  - Simulation
  - Assertion Verification
  - Formal Verification

Structural vs Behavioral Models

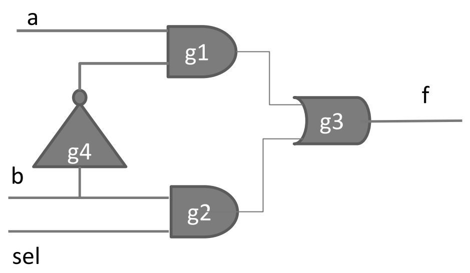
- Structural model
  - Just specifies primitive gates and wires
  - i.e., the structure of a logical netlist
- Behavioral model
  - More like a procedure in a programming language
  - Still specify a module in Verilog with inputs and outputs…
  - ...but inside the module you write code to tell what you want to have happen, NOT what gates to connect to make it happen
  - i.e., you specify the behavior you want, not the structure to do it
- Why use behavioral models
  - For testbench modules to test structural designs
  - For high-level specs to drive logic synthesis tools

Assertion Verification

- Assertion Monitors: Used to continuously check for design properties during simulation
  - Assertion Monitors developed to assert that the Design Properties are not violated
  - Firing of an assertion verification: alerts the malfunctioning of design according to the designer's expectation
  - Instead of having to inspect simulation results manually or by developing sophisticated test benches.
- Design Properties: Certain conditions have to be met for the design to function correctly
- Open verification library (OVL): provides a set of assertion monitors for monitoring common design properties

Representation: Structural Models

- Structural models
  - Are built from gate primitives and/or other modules
  - They describe the circuit using logic gates — much as you would see in an implementation of a circuit.
    - You could describe your lab1 circuit this way
- Identify:
  - Gate instances, wire names, delay from a or b to f.



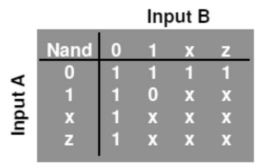
```
module mux (f, a, b, sel);
output f;
Input a, b, sel;
            and #5 g1 (f1, a, nsel),
                        g2 (f2, b, sel);
            or #5 g3 (f, f1, f2);
            not g4 (nsel, sel);
endmodule
```

Formal Verification

- Formal verification: The process of checking a design against certain properties
  - Examining the design to make sure that the described properties by the designer to reflect correct behavior of the design hold under all conditions
  - Property's Counter Examples: Input combinations that causes a property to fail
  - Property coverage: Measures how much the design is exercised by the property

Four-Valued Logic

- Logic with multi-level logic values
  - Logic with these four values make sense
    - Nand anything with a 0, and you get a 1. This includes having an x or z on the other input. That’s the nature of the nand gate
    - Nand two x’s and you get an x
- Note: z treated as an x on input. Their rows and columns are the same
- If you forget to connect an input ... it will be seen as an z.
- At the start of simulation, everything is an x.



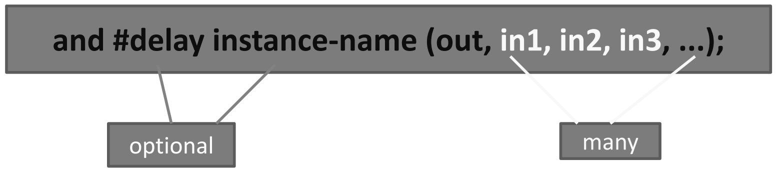
Input A / Input B

Nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

A 4-valued truth table for a Nand gate with two inputs

Representation: Gate-Level Models

- Need to model the gate’s:
  - Function
  - Delay
- Function
  - Generally, HDLs have built-in gate-level primitives
  - Verilog has NAND, NOR, AND, OR, XOR, XNOR, BUF, NOT, and some others
  - The gates operate on input values producing an output value
    - typical Verilog gate instantiation is:



and #delay instance-name (out, in1, in2, in3, ...);

How to build and test a module

- Construct a “test bench” for your design
  - Develop your hierarchical system within a module that has input and output ports (called “design” here)
- Develop a separate module to generate tests for the module (“test”)
- Connect these together within another module (“testbench”)

```
module testbench ();
    wire     l, m, n;

    design   d (l, m, n);
    test     t (l, m);

    initial begin
        //monitor and display
        ...
```

```
module design (a, b, c);
    input    a, b;
    output   c;
    ...
```

```
module test (q, r);
    output  q, r;

    initial begin
        //drive the outputs with signals
        ...
```

Four-Valued Logic

- **Verilog Logic Values**
  - The underlying data representation allows for any bit to have one of four values
  - 1, 0, x (unknown), z (high impedance)
  - x — one of: 1, 0, z, or in the state of change
  - z — the high impedance output of a tri-state gate.
- **What basis do these have in reality?**
  - 0, 1 ... no question
  - z ... A tri-state gate drives either a zero or one on its output. If it’s not doing that, its output is high impedance (z). Tri-state gates are real devices and z is a real electrical affect.
  - x ... not a real value. There is no real gate that drives an x on to a x is used as a debugging aid. x means the simulator can’t determine the answer and so maybe you should worry!
- **BTW ...**
  - some simulators keep track of more values than these. Verilog will in some situations.

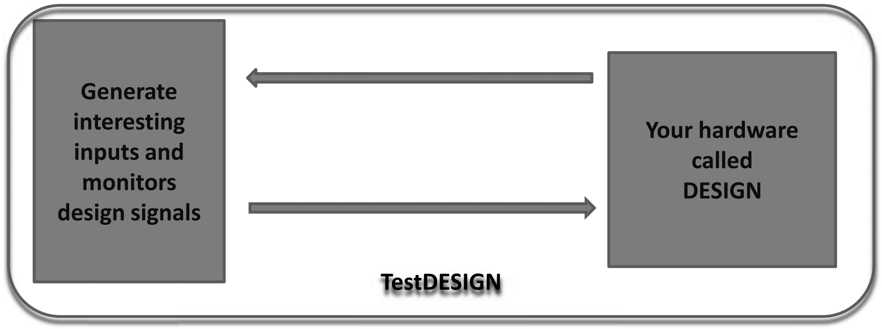
The test module (Test generator)

- $monitor
  - prints its string when executed.
  - after that, the string is printed when one of the listed values changes.
  - only one monitor can be active at any time
  - prints at end of current Simulation time
- Function of this tester
  - at time zero, print values and set a=b=0
  - after 10 time units, set b=1
  - after another 10, set a=1
  - after another 10 set b=0
  - then another 10 and finish

```
module  testHalfAdd(a, b, sum, cOut);
Input     sum, cOut;
output  a, b;
reg        a, b;
initial begin
       $monitor ($time,, “a=%b, b=%b,
               sum=%b, cOut=%b”, a, b, sum,
               cOut);
       a = 0; b = 0;
      #10 b = 1;
      #10 a = 1;
      #10 b = 0;
      #10 $finish;
end
```

Verilog Design

- Three Verilog modules


Generate interesting inputs and monitors design signals
Your hardware called DESIGN
TestDESIGN

- More than modeling hardware
  - $monitor — give it a list of variables. When one of them changes, it prints the information. Can only have one of these active at a time. e.g. ...

```
$monitor ($time,,, “a=%b, b=%b, sum=%b, cOut=%b”,a, b, sum, cOut);
```

Extra commas prints spaces

%b is binary, %h hex, %d for decimal

```
$display ($time,,, “a=%b, b=%b, sum=%b, cOut=%b”,a, b, sum, cOut);
```

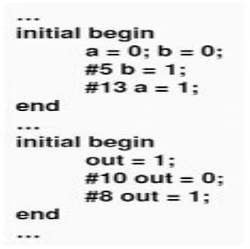
$display() — sort of like printf()

```
module testbench;
Wire su, co, a, b;
             halfAdd        U1 (su,co,a,b);
             testHalfAdd  U2 (a,b,su,co)
Endmodule
```

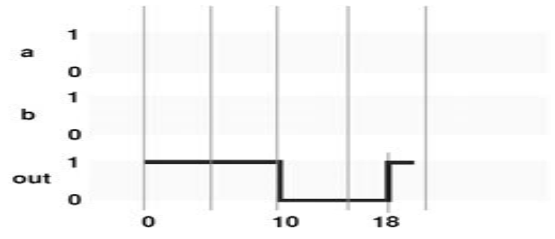
```
module  testHalfAdd(a, b, sum, cOut);
Input     sum, cOut;
output  a, b;
reg        a, b;
initial begin
       $monitor ($time,, “a=%b, b=%b, sum=%b, cOut=%b”,
                         a, b, sum, cOut);
       a = 0; b = 0;
      #10 b = 1;
      #10 a = 1;
      #10 b = 0;
      #10 $finish;
end
endmodule
```

```
module halfAdd (sum, cOut, a, b);
output sum, cOut;
Input a, b;
             xor #2  G1(sum,a,b);
             and #2 G2(Count,a,b);
endmodule
```

Two initial statements?



```
...
initial begin
        a = 0; b = 0;
        #5 b = 1;
        #13 a = 1;
end
...
initial begin
        out = 1;
        #10 out = 0;
        #8 out = 1;
end
...
```



- Things to note
  - Which initial statement starts first?
  - What are the values of a, b, and out when the simulation starts?
  - These appear to be executing concurrently(at the same time). Are they?

Structural vs Behavioral Models

- Structural model
  - Just specifies primitive gates and wires
  - i.e., the structure of a logical netlist
- Behavioral model
  - More like a procedure in a programming language
  - Still specify a module in Verilog with inputs and outputs…
  - ...but inside the module you write code to tell what you want to have happen, NOT what gates to connect to make it happen
  - i.e., you specify the behavior you want, not the structure to do it
- Why use behavioral models
  - For testbench modules to test structural designs
  - For high-level specs to drive logic synthesis tools

How do behavioral models fit in?

- How do they work with the event list and scheduler?
  - Initial (and always) begin executing at time 0 in arbitrary order
  - They execute until they come to a “#delay” operator
  - They then suspend, putting themselves in the event list 10 time units in the future (for the case at the right)
  - At 10 time units in the future, they resume executing where they left off.

```
module  testHalfAdd(a, b, sum, cOut);
Input     sum, cOut;
output  a, b;
reg        a, b;
initial begin
       $monitor ($time,, “a=%b, b=%b,
sum=%b, cOut=%b”, a, b,
                sum, cOut);
        a = 0; b = 0;
       #10 b = 1;
       #10 a = 1;
       #10 b = 0;
       #10 $finish;
end
```