Outline
1) Tip: WS_FTP
2) Behavioral Verilog Discussion
3) Timescale Discussion
4) Problem Set #3 Questions

Tip: WS_FTP
- A graphical way to ftp files securely between accounts
- In Windows, go to CAE Applications > Internet > WS_FTP.
- Get a copy for your PC at (may not work anymore)
  http://shelf.doit.wisc.edu/PC/WiscWorld/Client_Updates/WS_FTP/

Behavioral Verilog Discussion
- Behavioral Verilog consists of procedures, procedural statements, and flow control.
  - Procedures
    - initial (executed once)
    - always (repetitively executed)
  - Procedural Statements
    - Blocking (=)
    - Nonblocking (<=)
    - Procedural continuous assignments (assign, force)
  - Flow Control
    - Conditional operator (?:)
    - Case statements (case, casex, casez)
    - Branching (if-else)
    - Loops (forever, repeat, while, for)
- Each procedure construct starts a separate activity flow (thread), which are run in parallel (just like hardware).
- By default, procedures only include the next statement. An arbitrary number of statements maybe included by use of begin and end.
```verilog
module behave;
    reg [1:0] a, b;

    initial begin
        a = 'b1;
        b = 'b0;
    end

    always begin
        #50 a = ~a;
    end

    always
        #100 b = ~b;
endmodule
```

- The `always` procedure can be executed conditionally using the `@` symbol.
  - Signal's value changes - `always@(signal)`
  - Transition of signal
    - Positive edge - `always@(posedge clk)`
    - Negative edge - `always@(negedge clk)`
  - Multiple conditions - `always@(signalA or signalB)`
    - `always@(posedge clk or signalA)`
    - `always@(posedge clk or posedge reset)`
  - `posedge` and `negedge` CANNOT be mixed in the conditional.

- Procedural statements update register variables (reg, integer, etc.). In other words, only register variables can be on LHS. However, there are several ways in which they can be assigned. Assume `a` and `b` are 8-bit registers.
  - Whole variable (`a = ...`)
  - Partial variable
    - Subrange (`a[6:0] = ...`)
    - Bit (`a[7] = ...`)
  - Concatenated variables (`{a,b} = ...`)

- Blocking and nonblocking procedural assignments determine different flows through `begin-end` blocks.
  - A blocking statement (=) evaluates its RHS and updates its LHS before flow continues to the next statement within a `begin-end` block.
  - A nonblocking statement (=) allows for assignment scheduling without interrupting activity flow. Simply put, all RHS of consecutive nonblocking statements evaluate before all their LHS update.

<table>
<thead>
<tr>
<th>Procedural Assignment</th>
<th>Flow</th>
<th>Design Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking</td>
<td>Sequential</td>
<td>Sequential</td>
</tr>
<tr>
<td>Nonblocking</td>
<td>Parallel</td>
<td>Combinational</td>
</tr>
</tbody>
</table>
Example 1:

```verilog
testbench module evaluates2 (out);
output out;
reg a, b, c;
initial begin
  a = 0;
b = 1;
c = 0;
end
always c = #5 ~c;
always @(posedge c) begin
  a <= b; // evaluates, schedules,
b <= a; // and executes in two steps
end
endmodule
```

At posedge c, the simulator evaluates the right-hand sides of the non-blocking assignments and schedules the assignments of the new values at the end of the non-blocking assign update events (see 5.4).

**Step 1:**

```
<table>
<thead>
<tr>
<th>Values at Time 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 0</td>
</tr>
<tr>
<td>b = 1</td>
</tr>
</tbody>
</table>
```

When the simulator activates the non-blocking assign update events, the simulator updates the left-hand side of each non-blocking assignment statement.

**Step 2:**

```
<table>
<thead>
<tr>
<th>Values at Time 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 1</td>
</tr>
<tr>
<td>b = 0</td>
</tr>
</tbody>
</table>
```

Assignment values are:

```verilog```

Example 2:

```verilog
//non_block1.v
module non_block1;
reg a, b, c, d, e, f;

//blocking assignments
initial begin
  a = #10 1; // a will be assigned 1 at time 10
  b = #2 0; // b will be assigned 0 at time 12
  c = #4 1; // c will be assigned 1 at time 16
end

//non-blocking assignments
initial begin
  d <= #10 1; // d will be assigned 1 at time 10
  e <= #2 0; // e will be assigned 0 at time 2
  f <= #4 1; // f will be assigned 1 at time 4
end
endmodule
```

scheduled changes at time 2

```
<table>
<thead>
<tr>
<th>Values at Time 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>e = 0</td>
</tr>
</tbody>
</table>
```

scheduled changes at time 4

```
<table>
<thead>
<tr>
<th>Values at Time 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>f = 1</td>
</tr>
</tbody>
</table>
```

scheduled changes at time 10

```
<table>
<thead>
<tr>
<th>Values at Time 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>d = 1</td>
</tr>
</tbody>
</table>
```
Procedural continuous assignments (assign) override all procedural assignments to a register while active. This assignment can only be changed by reassignment (another assign) or de-assignment (deassign).
**Timescale Discussion**

- `timescale - a compiler directive that sets the simulation’s time unit size and precision
- `timescale <time_unit> / <time_precision>
  - time_unit – constant multiplier of time values
  - time_precision – minimum step size during simulation, which determines rounding of numerical values
- Allowed unit/precision values: {1 | 10 | 100, s | ms | us | ns | ps}
- Different timescales can be used for different sequences of modules.
- Smallest time_precision determines the precision of the simulation.
- **Must be used in synthesis designs.**

Example:
```
`timescale 10ps / 1ps

module sampleDesign (z,x1,x2);
  input x1, x2;
  output z;

  nor #3.57 (z, x1, x2);
endmodule
```

The nor gate’s delay is 36 ps (3.57 x 10 = 35.7 ps rounded to 36).