System Verilog (SV) is a parallel, hardware description language. SV differs from procedural languages such as C in that it models concurrency in digital logic. Logic gates operate in parallel, but software programs execute serially, one statement at a time. System Verilog looks/feels a lot like C...but behaves very differently! Remember, when writing SV code, you are describing hardware structures, not creating a binary executable.
System Verilog allows us to do two things with the same piece of code:

- Infer the physical structure of the logic to be created.
- Simulate the behavior of the logic prior to building it.

We must write code to simultaneously satisfy two tools. In order of importance...

- Logic synthesizer - Our code must clearly infer the structure (including errors!) we want from logic synthesis.
- Logic simulator - Our code, when simulated, must exhibit the exact behavior of the synthesized gates that are yet to be built.

Much of the power of HDLs comes from the ability to quickly simulate a design with the confidence that it represents the behavior of the gates that will be built. A simulation of an RTL-level design will run several orders of magnitude faster than a synthesis-generated netlist of the structural gate-level design.
Verilog Execution Semantics

- Don’t keep hacking code till the simulator is happy... Its working gates you want!
- Well structured RTL code written with an eye towards logic synthesis almost always simulates correctly. The synthesizer is way pickier than the simulator.
  - This is a big reason for our *structure first* approach.

*We write HDL code that expresses structure while making sure the simulator understands what structure we are creating. Writing SV code that creates correct behavior in the simulator is easy but is often incomprehensible to the synthesizer as the synthesizible subset of SV is far smaller than that understood by the simulator.*
Verilog Execution Semantics

- System Verilog models concurrency with two basic constructs:
  - Continuous assignment
  - Procedural Blocks

- Continuous assignment statements execute when a variable on the RHS changes. When the change occurs, the LHS is updated immediately. This behavior is just like a logic gate. An example of a continuous assignment statement is below.

```verilog
generate d = a + b - c; //d gets updated whenever a, b or c change
```
Verilog Execution Semantics

- A procedural block is made up of procedural statements which are executed sequentially just like any C program whenever the block is awakened for execution.
- Procedural blocks are awakened when a signal in the sensitivity list changes.
- Two procedural blocks describing some combinatorial logic are shown below.

```
// combo logic - V95 style, explicit sensitivity list
always @ (a, b, c, d) begin
  out_0 = a + b;
  out_1 = c + d;
end

// combo logic - SV style, implied sensitivity list
always_comb begin
  out_0 = a + b;
  out_1 = c + d;
end
```
To describe both clocked (sequential) and combinatorial logic, and to avoid problems with non-determinism, Verilog uses two different assignment operators within procedural blocks.

- **Blocking assignments**, "\=\” evaluate the RHS and update the LHS immediately before any further instructions are executed.

- **Non-Blocking assignments**, "<=" evaluate the RHS and postpone updating LHS until the end of present simulation timestep.
Verilog Execution Semantics

Blocking and Non-Blocking Assignment

- In `always_comb` blocks we describe purely combinatorial logic.
- To describe combinatorial logic, use the blocking assignment: ";=

```verilog
always_comb begin
    x = a & b;
    y = x | c;
end
```

```verilog
always_comb begin
    x <= a & b;
    y <= x | c;
end
```

Which piece of code correctly models the logic below?

![Logic Diagram]

U1

U2

a

b

c

x

y
Verilog Execution Semantics

```verilog
always_comb begin
    x = a & b;
    y = x | c;
end
```

<table>
<thead>
<tr>
<th>Blocking Behavior</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial condition</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>a changes</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x = a &amp; b;</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>y = x</td>
<td>c;</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Correct behavior of the combinatorial gates is observed.
Verilog Execution Semantics

```verilog
always_comb begin
  x <= a & b;
  y <= x | c;
end
```

### Non-Blocking Behavior

<table>
<thead>
<tr>
<th>Non-Blocking Behavior</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>x</th>
<th>y</th>
<th>Deferred assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial condition</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>a changes</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>x = a &amp; b;</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x &lt;= 0</td>
</tr>
<tr>
<td>y = x</td>
<td>c;</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>assignment completion</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x &lt;= 0, y &lt;= 1</td>
</tr>
</tbody>
</table>

- Non-blocking assignments do not embody the behavior of the gate structure desired. Incorrect behavior is the result.
- Use blocking assignments for combinatorial logic.
Verilog Execution Semantics

- In `always_ff` blocks we describe clocked logic such as flip-flops using the nonblocking assignment "`<=`".
- Think about how the following circuit operates after reset has deasserted. It's a serial shift register.

![Serial Shift Register Diagram]

- The behavior we wish to describe is that at each clock edge, `q1`, `q2` and `out` simultaneously take on the values of `in`, `q1` and `q2` respectively.
- How would the operation differ using blocking and nonblocking assignments?
Non-blocking assignments defer all assignments to the end of the simulation cycle. This is how real flip-flops would act in this circuit.

Blocking assignments ripple the value of `in` all the way to `out` by completing each assignment in order. At the end of the simulation cycle, `out` holds the value of `in`, essentially removing the 2nd and 3rd flip-flops.

Use nonblocking assignments for clocked logic.
Verilog Execution Semantics

Since SV is a parallel language, and models concurrent execution, it is logical to ask what the order of execution is outside procedural statements.

In Verilog, when multiple procedural blocks or continuous assignments are triggered to run, the order in which they are evaluated is not specified and is therefore non-deterministic.

For example, these two procedural blocks may be executed in any order, with different results for the variable z.

```verilog
always @ (a)
    z = b ^ c;
always @ (a)
    z = b ^ d;
```

Hardware-wise, this is a nonsensical case. Two XOR gates would be needed to compute the results and their outputs could never be connected together without creating a buffer-fight.
Verilog Execution Semantics

- Since **always** blocks can execute in any order, blocking assignments can cause trouble.

```verilog
module fbosc1(
    output logic y1, y2;
    input    clk, rst);

    //"A" procedural block
    always_ff(posedge clk, posedge rst)
        if (rst) y1 = 0; // reset
        else         y1 = y2;

    //"B" procedural block
    always_ff(posedge clk, posedge rst)
        if (rst) y2 = 1; // preset
        else         y2 = y1;

endmodule
```

"A" then "B" execution

"B" then "A" execution
Using nonblocking assignments, we can live with non-deterministic scheduling.

```verilog
module fbosc1(
    output logic y1, y2,
    input   clk, rst);

//"A" procedural block
always_ff(posedge clk, posedge rst) 
    if (rst) y1 <= '0; //reset
    else       y1 <= y2;

//"B" procedural block
always_ff(posedge clk, posedge rst) 
    if (rst) y2 <= '1; //preset
    else       y2 <= y1;
endmodule
```
Verilog Execution Semantics

- Understanding all the nuances of the blocking and nonblocking assignment is beyond the scope of our class.

- However, adherence to the following guidelines will keep your simulator and synthesis tools in sync.

1. When modeling sequential logic, use `always_ff` and "\(<=\)."
2. When modeling latches, use `always_latch` and "\(<=\)."
3. When modeling combinatorial logic use `always_comb` and "\(=\)."
4. When modeling both sequential and combinatorial logic within the same always block (like a counter), use `always_ff` and "\(<=\)."
5. Don’t mix "\(=\)" and "\(<=\)" assignments in the same always block.
6. Don’t make assignments to the same variable from more than one always block or continuous assignment.