# 8 Hardware Description Languages (cont.)

## Verilog

### Delays

We have a unit of 'ticks', which we can have delayed logic through the circuit.
We can do, for example
assign #1 {ab,bb,cb} = ~{a,b,c}; will assign the result after one tick instead of immediatly.

### Sequential Logic

We could have synchronous reset:

```
always_ff @(posedge clk)
  if (reset) q <= 4'b0;
  else       q <= d;
```

or asynchronous reset:

```
always_ff @(posedge clk, posedge reset)
  if (reset) q <= 4'b0;
  else       q <= d;
```

We could enable a value to be passed:

```
always_ff @(posedge clk, posedge reset)
  if      (reset) q <= 4'b0;
  else if (en)    q <= d;
```

or have a value that changes any time that the clock is high.

```
always_latch
   if (clk) q <= d;
```

We have a nice structure always_comb where any time any of the inputs change, it will reevaluate:

```
always_comb        // need begin/end because there is
  begin            // more than one statement in always
    y1 = a & b;    // AND
    y2 = a | b;    // OR
    y3 = a ^ b;    // XOR
    y4 = ~(a & b); // NAND
    y5 = ~(a | b); // NOR
  end
```

We have a great example:

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);
  always_comb
    case (data)
      //                    abc_defg
      0: segments =       7'b111_1110;
      1: segments =       7'b011_0000;
      2: segments =       7'b110_1101;
      3: segments =       7'b111_1001;
      4: segments =       7'b011_0011;
      5: segments =       7'b101_1011;
      6: segments =       7'b101_1111;
      7: segments =       7'b111_0000;
      8: segments =       7'b111_1111;
      9: segments =       7'b111_0011;
      default: segments = 7'b000_0000; // required
    endcase
endmodule
```

We can also ignore some parts of our input:

```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);
  always_comb
    casez(a)
      4'b1???: y = 4'b1000; // ? = don't care
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
   endcase
endmodule
```

## Blocking and Nonblocking

In sequential logic areas, <= is a non blocking assignment. These happen at the same time. If we care about a value from the previous

input, then we would make it blocking, by assigning with the sign =.

However, what if we wrote something like:

always_comb begin
a<=b&c;
d<=a&w;
e<=d|p;
end
If we update b, then a will change. The others will evaluate with the old a. Then, those values will update, and the always_comb will update them. Then the others will change again. This will keep going until we have the answer, but it takes 3 cycles to simulate. We could do this in just one. Using equals, then we wait for each step and update the rest.

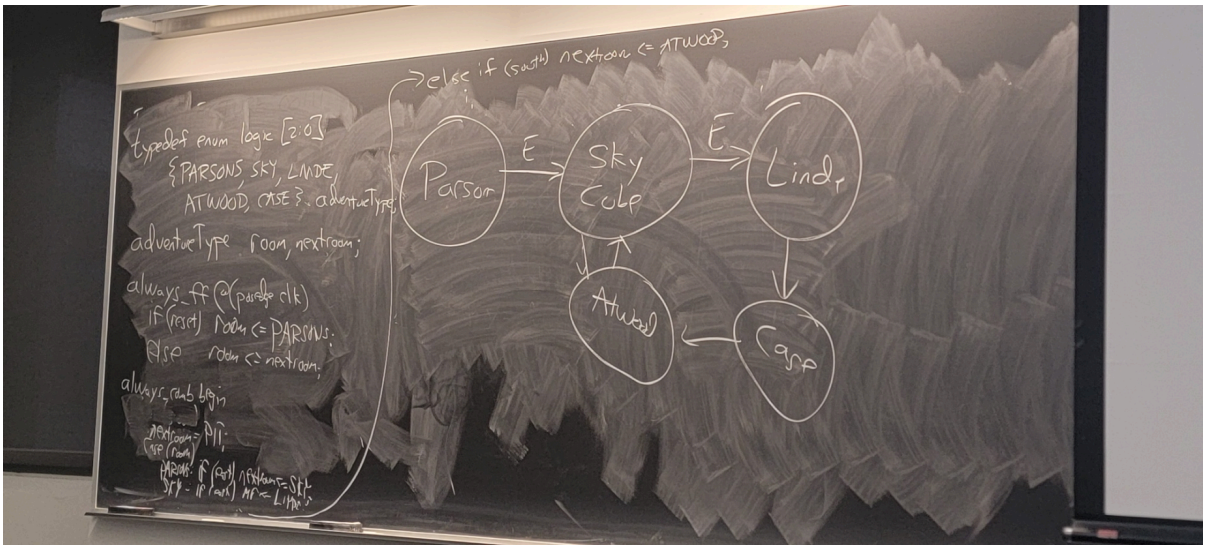
We now have some rules:
Synchronous sequential logic should use always_ff @(posedge clk) and non-blocking assignments, so that our flip-flops will work.

Simple combinational logic should use continuous statements,
assign y = a&b

More complicated combinational logic, use =.

In an always block, use gets (<=) and you will be safe.

## FSMs




I stopped taking live notes here:

# Parameterized Modules

We can make modules that automatically have a parameter as one value, but which can be overwritten.

**2:1 mux:**

```
module mux2
  #(parameter width = 8)  // name and default value
   (input  logic [width-1:0] d0, d1,
    input  logic             s,
    output logic [width-1:0] y);
  assign y = s ? d1 : d0;
endmodule
```

**Instance with 8-bit bus width (uses default):**

```
  mux2 myMux(d0, d1, s, out);
```

**Instance with 12-bit bus width:**

```
   mux2 #(12) lowmux(d0, d1, s, out);
```

# Test Benches

Big overview notes: Check that the number of bits in your test vector matches the number of bits for your input file, and also set that bit length at the end for terminating the test vector file when it becomes nulls.

Generate the clock signal

```
module testbench3();
  logic        clk, reset;
  logic        a, b, c, yexpected;
  logic        y;
  logic [31:0] vectornum, errors;    // bookkeeping variables
  logic [3:0]  testvectors[10000:0]; // array of testvectors

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // generate clock
  always     // no sensitivity list, so it always executes
    begin
      clk = 1; #5; clk = 0; #5;
    end
```

Read your testvector file:

```
// at start of test, load vectors and pulse reset

initial
   begin
     $readmemb("example.tv", testvectors);
     vectornum = 0; errors = 0;
     reset = 1; #22; reset = 0;
   end


// Note: $readmemh reads testvector files written in
// hexadecimal
```

Set your input values and the expected outputs

```
// apply test vectors on rising edge of clk
always @(posedge clk)
  begin
    #1; {a, b, c, yexpected} = testvectors[vectornum];
  end
```

Check your outputs:

```
// check results on falling edge of clk
   always @(negedge clk)
    if (~reset) begin // skip during reset
      if (y !== yexpected) begin
        $display("Error: inputs = %b", {a, b, c});
        $display("  outputs = %b (%b expected)",y,yexpected);
        errors = errors + 1;
      end

// Note: to print in hexadecimal, use %h. For example,
//       $display("Error: inputs = %h", {a, b, c});

// increment array index and read next testvector
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 4'bx) begin
          $display("%d tests completed with %d errors",
                vectornum, errors);
        $stop;
      end
    end
endmodule

// === and !== can compare values that are 1, 0, x, or z.
```

**This bit width needs to be the same as vector size!**