More on Verilog

Sign extension: Example 1

```
wire [3:0] c, d
reg [4:0] sum;
always @ (c or d) begin
sum= {c[3],c} + {d[3],d};
end
```

Sign extension: Example 2

```
input [2:0] in;
reg    [3:0] d;
wire  [3:0] c;
output [4:0] sum;
reg [4:0] sum;

assign c = {in{2},in} + 4'h1;

always @(posedge clk)
    d <= a;

always@(c or d)
    sum <= {c[3],c} + {d[3],d};
```

Blocking example

- It is sequential in the sense that everything is updated one statement at a time within one always block just as C/C++ does.
- Example:

```
wire a;
assign a = 1;
always @(posedge clk) begin
    b = a;
    c = b;
end
```

- Results:

  At time = 0: clk = 0, a = 1, b = X, and c = X

  At time = 1: clk = 1, a = 1, b = 1, and c = 1

Nonblocking Example

- The nonblocking statements do not wait for the previous statements. They execute right away thus taking the old values of signals.
- Example

```
wire a;
assign a = 1;
always @(posedge clk) begin
    b <= a;
    c <= b;
end
```

- Results

  At time = 0: clk = 0, a = 1, b = X, and c = X
  At time = 1: clk = 1, a = 1, b = 1, and c = X
  At time = 2: clk = 0, a = 1, b = 1, and c = X
  At time = 3: clk = 1, a = 1, b = 1, and c = 1

Never update one Reg using multiple always blocks

- Example

```
reg a;
always @(posedge clk)
    a = b & c;
always @(posedge clk or negedge reset)
   if (~reset)
        a = 0;
```

This maybe fine in simulation but confuses the synthesis. For very complex state machines this can also increase chances of creating bugs.

“Wildcard” support by Verilog-2001

- As of Verilog-2001, the language supports "wildcard" sensitivity lists.
- Example:
  ```
  reg a;
  always @(b or c)
     a = b & c;
  ```
- Can now be done like this:
  ```
  always @(*)
     a = b & c;
  ```
  * = any input inside the always block. So now the only time you don't use * is when you do posedge/negedge or intentionally make a latch.

  @(*) are very useful when you have a huge number of inputs since it eliminates the chance of you forgetting something.

Arithmetic shift

Arithmetic right shift is this: >>> . Unlike regular right shift it extends the MSB instead of simply putting '0'.

Example:

5'b11111 >> 4 = 00001

5'b11111 >>> 4 = 00001

$signed(5'b11111) >>> 4 = 11111

Arithmetic Shift Example

```
reg signed [15:0] my_number;
wire [15:0] new_number;
assign new_number = my_number >>> 4;
```

So "new_number" didn't need to be "wire signed" but "my_number" did so that the ">>>" operation worked properly.

Alternative example:

```
reg [15:0] my_number;
wire [15:0] new_number;
assign new_number = $signed(my_number) >>> 4;
```

In this case I casted "my_number" to a "signed format" before doing the arithmetic shift.