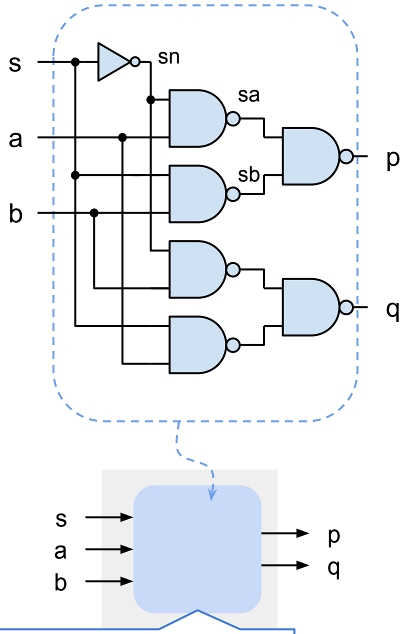
Digital Techniques 2

L3: Combinational Logic Design 1

UNIVERSITY OF OULU
CIRCUITS and SYSTEMS

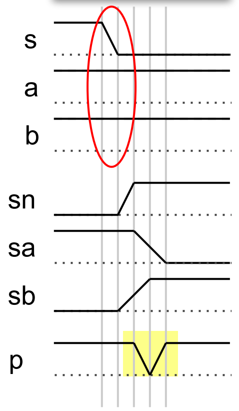
Combinational Logic Circuit Properties

- In combinational circuits, outputs' states depend only on current state of the inputs
- Change in any input can cause a change in outputs (circuit is *sensitive* to all inputs)
- To define a circuit completely, output values must be defined for all combinations of input values
- Combinational logic operations are implemented with **logic gates** in ICs or **RAM look-up tables** in FPGAs
- Because of **propagation delays** of logic components, outputs of combinational circuits often show wrong values before they settle ("hazard", "glitch")
- Delays are not considered in RTL models



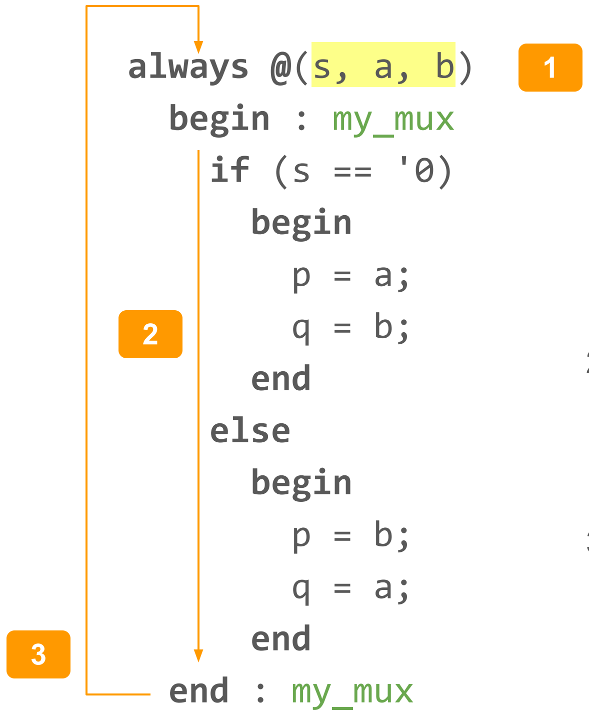
Blue symbols denote combo logic in this course!

s	a	b	p	q
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	0	0
1	0	1	1	0
1	1	0	0	1
1	1	1	1	1



Processes in Hardware Description Languages

- A process is a block of *sequential* programming language statements that is **executed continuously**
- A module can contain many processes that operate **concurrently**
- An ***always* procedure** is this kind of process in SystemVerilog[1]
- An **always** procedure consists of
  - An (optional) **event list** (a.k.a. **sensitivity list**) that contains a list of expressions whose change starts the procedure
  - A list of **sequential statements** that are executed in order



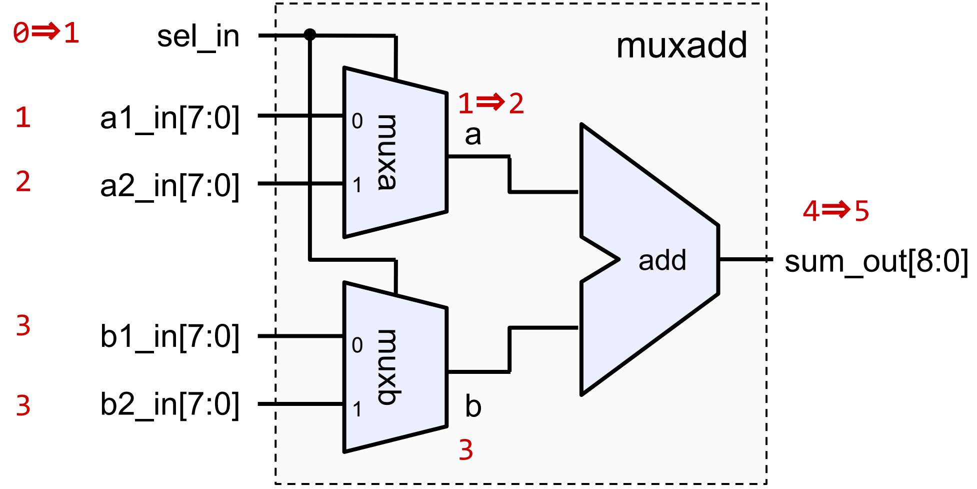
```
always @(s, a, b)
  begin : my_mux
    if (s == '0)
      begin
        p = a;
        q = b;
      end
    else
      begin
        p = b;
        q = a;
      end
  end : my_mux
```

1. **always** procedure waits here for a change in **s**, **a** or **b**.
2. When a change occurs, the sequential statements are executed
3. When the end is reached, the process returns to the beginning to wait for the next event

1 **process** statement in VHDL.

UNIVERSITY OF OULU
CIRCUITS and SYSTEMS

Process Example



Assume that in the beginning **sel_in**=0, **a1_in**=1, **a2_in**=2, **b1_in**=3, **b2_in**=3, and thus **sum_out**=4.

If **sel_in** rises to 1 ⇒ **muxa** and **muxb** are triggered ⇒ **a** changes to 2, **b** remains unchanged ⇒ **add** is triggered because of **a** ⇒ **sum_out** changes to 5.

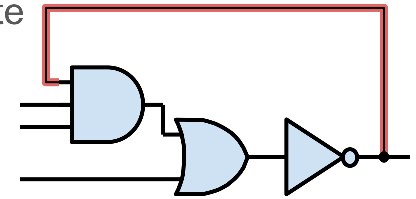
```
module muxadd
  (input logic sel_in,
   input logic [7:0] a1_in, a2_in, b1_in, b2_in,
   output logic [8:0] sum_out);

   logic [7:0] a, b;

   always @(sel_in, a1_in, a2_in)
     begin : muxa
       if (sel_in == '0)
         a = a1_in;
       else
         a = a2_in;
     end : muxa

   always @(sel_in, b1_in, b2_in)
     begin : muxb
       if (sel_in == '0)
         b = b1_in;
       else
         b = b2_in;
     end : muxb

  always @(a, b)
     begin : add
       sum_out = a + b;
     end : add
endmodule
```

Coding Rules Combinational Processes

1. Process must be **sensitive to all of its inputs** (to variables and ports whose values are read in the process)
2. Process must ***always* assign a value to *all* of its outputs** (variables and ports that are assigned to *somewhere* in the process)
3. Internal variables must be given values before they are read (**"no memory"** rule)
4. It contains **no feedback** (variable cannot be used as input and output, and it must written before it is read in the process)
5. Variables must be assigned using the **blocking assignment operator =**, which takes effect immediately

If you break these rules, the following can happen:

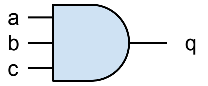
- Synthesis generates level-sensitive **latches** to implement an *implied* memory,
- Simulated and synthesized functionality can be different ("simulation-synthesis mismatch")
- A **combinational feedback** loop ("timing loop") can be created, potentially causing circuit to oscillate



Common Modeling Errors 1

- Incomplete sensitivity list

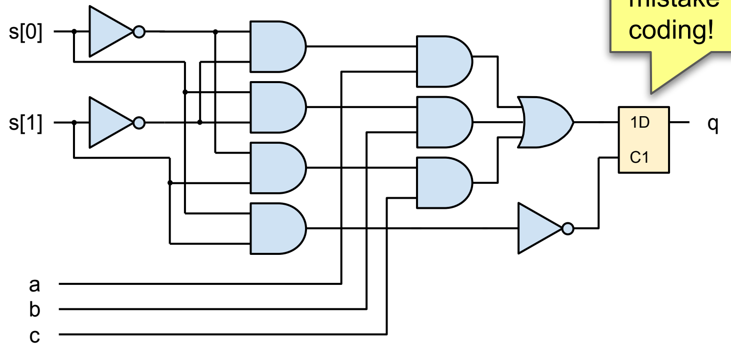
```
always @(a, b)
  q = a & b & c;
```



Synthesis tool creates an AND gate, but in RTL simulation changes in **c** do not affect output ⇒ RTL simulation results differ from real circuit behavior.

- Latch inferred

```
logic a, b, c, q;
logic [1:0] s;

always @(s, a, b, c)
 begin : mux3
   if (s == 2'b00)
     q = a;
    else if (s == 2'b01)
     q = b;
    else if (s == 2'b10)
     q = c;
 end : mux3
```



This is the most common "rookie mistake" in RTL coding!

Code does not assign a value to **q** when **s** == 2'b11. A **latch** must be inserted at the output to make **q** hold its previous state in that case.

A latch is a 1-bit memory whose output follows its D-input when its clock input C is '1. When C is '0, the output preserves its state.

UNIVERSITY OF OULU
CIRCUITS and SYSTEMS

Common Modeling Errors 2

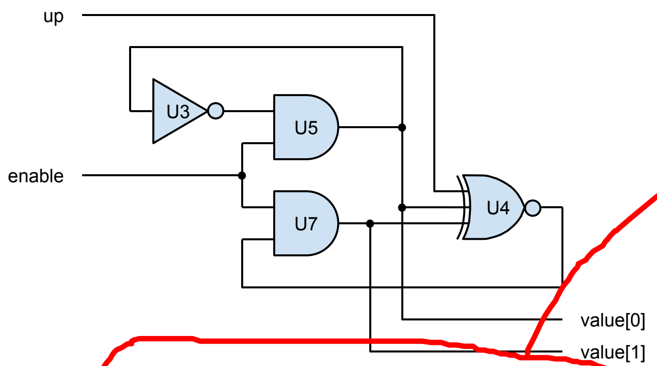
- Combinational feedback loop

```
module badcombo
  (input logic      enable, up,
   output logic [1:0]  value);

  logic [1:0] count;

  always @(enable, up)
    begin
       if (enable)
         if (up)
           count = count + 1;
         else
           count = count - 1;
       else
         count = 0;
    end

   assign value = count;

endmodule
```

An up/down counter is NOT modelled this way!

If a process reads and writes the same variable, care must be taken that the variable is always assigned a value in the process before it is read. Failing to do that will cause a **latch** to be inferred or a **combinational feedback loop** to be created.

VVI



Synthesis tool warnings:

```
Information: Timing loop detected. (OPT-150)
        U3/A U3/Y U5/A2 U5/Y
Information: Timing loop detected. (OPT-150)
        U4/A2 U4/Y U7/A1 U7/Y
```

always_comb Procedure in SystemVerilog

- **always** is a "general-purpose" process in Verilog and SystemVerilog
- SystemVerilog defines the **always_comb** procedure that is exclusively meant for modeling combinational logic for RTL synthesis
- always_comb **does not require or allow a sensitivity list** to be defined (it is by default sensitive to all of its inputs)
- always_comb **models combinational behaviour more accurately** in simulation
- The always_comb keyword **declares the designer's *intent*** for synthesis tools

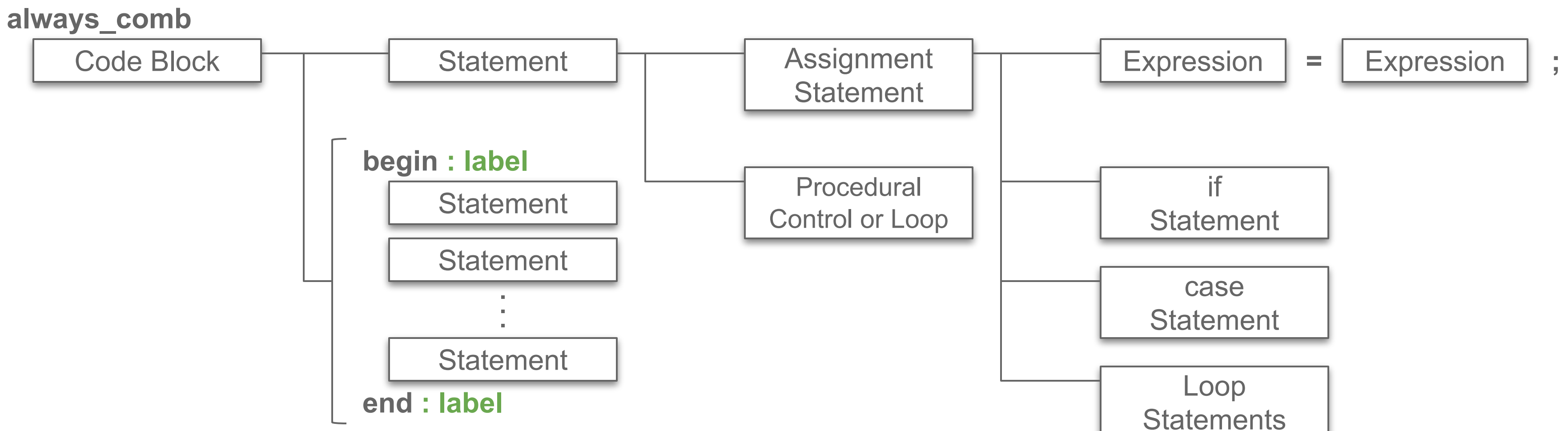
```
always_comb
  begin
    if (s == 2'b00)
      q = a;
     else if (s == 2'b01)
      q = b;
     else if (s == 2'b10)
      q = c;
  end
```

Now that the synthesis tool *knows* that the code tries to model combinational logic, it can issue a warning if sequential components are created:

```
Warning: badcombo.sv:35: Netlist for
always_comb block contains a latch. (ELAB-974)
```

Conclusion: Always use only **always_comb** to model combinational logic!

Procedural Code Structure (Simplified!)



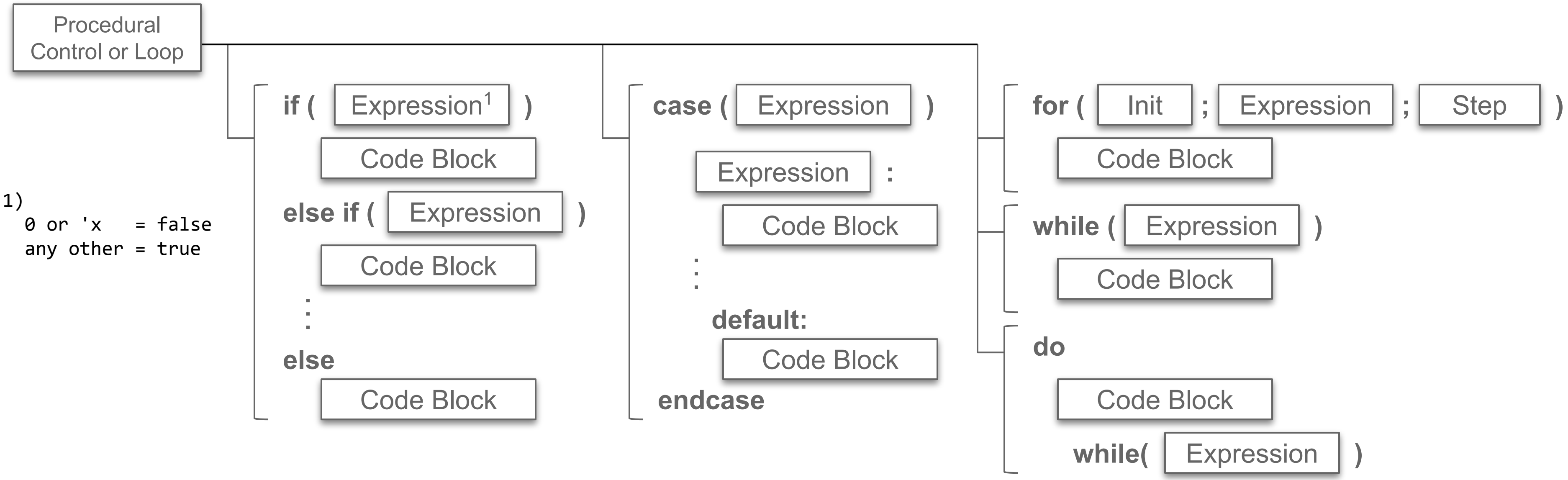
```
always_comb
  q_out = a_in + b_in;
```

```
always_comb
  q1_out = a_in + b_in;
  q2_out = a_in - b_in;
```

Indentation has no effect as in Python. always_comb only "affects" q1_out = …

```
always_comb
  begin
    q1_out = a_in + b_in;
    q2_out = a_in - b_in;
  end
```

Procedural Control Statements and Loops



```
1)
  0 or 'x    = false
  any other = true
```

Check out also:

```
unique if
unique0 if
priority if
```

```
unique case
unique0 case
priority case
```

See 12.4, 12.5 and 12.7 in SystemVerilog Standard

Examples: **case** Statement and **for** Loop

```
case ( sel_in )
 3'b000:
   case_out = a_in;
 3'b001:
   case_out = b_in;
 3'b010, 3'b011:
   case_out = c_in;
 default:
   case_out = 4'b0000;
endcase
```

Case expression whose value selects one case item.

Case item expressions matched with case expression

default item is selected if none of the case item expressions match.

The code block of a case item is executed if the value of its **case item expression** == **case expression**.

```
        a  sum
00000000 1000
00000001 0111
00000010 0111
00000011 0110
........ ....
11111111 0000
```

```
module count_zeros
  (input logic  [7:0] a,
   output logic [3:0] sum);

   always_comb
    begin
      logic [3:0] nzeros;

      nzeros = 0;
      for (int i = 0; i < 8; i = i+1)
       begin
         if (a[i] == '0)
           nzeros = nzeros + 1;
       end
       sum = nzeros;
    end
endmodule
```

Loop expression: if TRUE, loop code is executed

Loop variable **initialization**

Loop step: Executed after every loop iteration.

This code loops through the elements of vector **a** and counts the number of zero bits in **a**.

Exercise: Check that all combinational logic coding rules were obeyed!

Coding Style Examples: Multiplexers

Notice! This coding style only works if N is power of two!
In other cases, add an **if .. else** to filter out too big select values.

```
module mux_4x8
  (input logic [1:0] sel_in,
   input logic [7:0] a_in, b_in, c_in, d_in,
   output logic[7:0] mux_out);

  always_comb
    begin : mux_logic
       case (sel_in)
         2'b00: mux_out = a_in;
         2'b01: mux_out = b_in;
         2'b10: mux_out = c_in;
         2'b11: mux_out = d_in;
       endcase
    end : mux_logic

endmodule
```

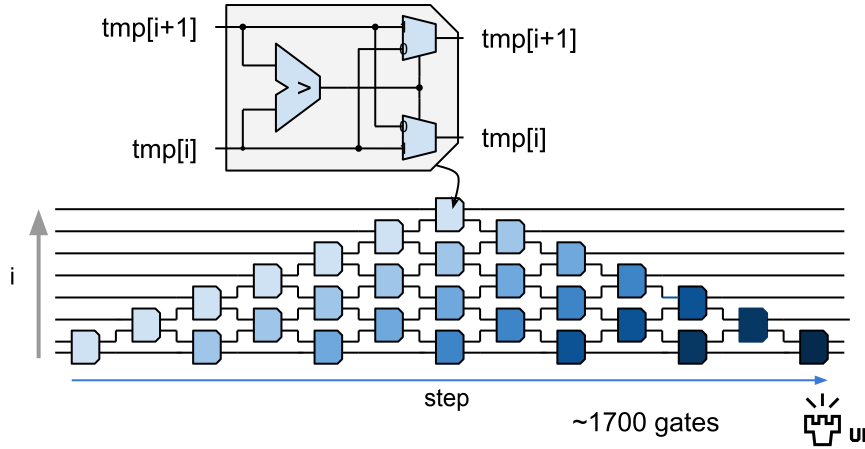
Notice! If number of inputs is less than the addressable range of sel_in, a **default** case item must be added

```
module mux_Nx8
  #(parameter N = 4)
  (input logic [$clog2(N)-1:0] sel_in,
   input logic [N-1:0][7:0]    data_in,
   output logic[7:0]           mux_out);

  always_comb
     begin : mux_logic
          mux_out= data_in[sel_in];
     end : mux_logic
endmodule
```

System function **$clog2**() returns the log base 2 of x argument rounded up to an integer value. Very useful in hardware design!

Coding Style Examples : Algorithmic Models

- There are no theoretical limits to the complexity of algorithmic models of combinational logic
- However, logic synthesis programs do not optimize the RTL architecture; complex algorithms may therefore yield complex hardware



```
module bubble_sort8
  (input logic  [7:0][7:0] data_in,
   output logic [7:0][7:0] data_out);

   always_comb
     begin : sort_logic
        logic [7:0][7:0] tmp;
        tmp = data_in;

        for (int step = 0; step < 7; step = step + 1)
          for (int i = 0; i < 7-step; i = i + 1)
            if (tmp[i] > tmp[i+1])
              { tmp[i+1], tmp[i] } = { tmp[i], tmp[i+1] };
        data_out = tmp;
    end : sort_logic
endmodule
```

For each iteration of the inner loop, a comparator and two muxes are synthesized to implement a swap operation.

Exercise: Check that all combinational logic coding rules were obeyed!

Additional Coding Rules

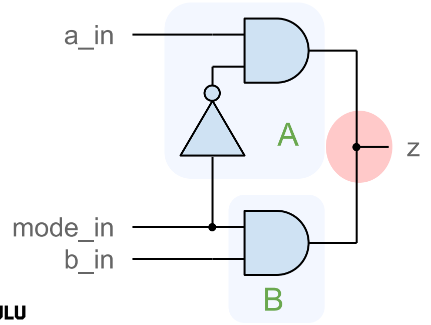
- Initial assignments of variables must **not** be used

```
logic x = '0, y = '0; // Wrong!
always_comb
  begin
    x = a_in | b_in;
    if (a_in & b_in & c_in)
       y = '1;
  end
```

- Initial assignment can hide the bug in the code as **y** should remain in unknown state in simulation until (**a_in** & **b_in** & **c_in**) becomes true
- **Synthesis tools ignore initial assignments** as they cannot be implemented in hardware, which can cause a **simulation-synthesis mismatches**

- Variable cannot be driven from two processes

```
always_comb
  begin : A
    z = a_in & ~mode_in; // Wrong!
  end : A
always_comb
  begin : B
    z = b_in & mode_in; // Wrong!
  end : B
```



Short circuit created!

always_comb will report an error from this but **always** does not.

References

1. IEEE Std 1800-2012 IEEE STANDARD FOR SYSTEMVERILOG
   Ch 9. Processes, esp. always (9.2.2.1) and always_comb (see 9.2.2.2)
2. IEEE Standard for Verilog® Register Transfer Level Synthesis
   5.1 Modeling combinational logic