
PUFacademy
A PUFsecurity Alliance

Digital Logic Design
- Lecture 2
- Combinational Logic

2025 Spring

PUFacademy
A PUFsecurity Alliance

Agenda

1. Design intent
2. Code Syntax
3. Advance Syntax

Combinational Logic

- Combinational logic circuit
  - A type of digital logic design
  - The output depends only on the combination of current input values and not on any previous input or output values.

- Consist of logic gates
  - AND, OR, NOT, XOR gates …
  - Output is the Boolean expression of all inputs.

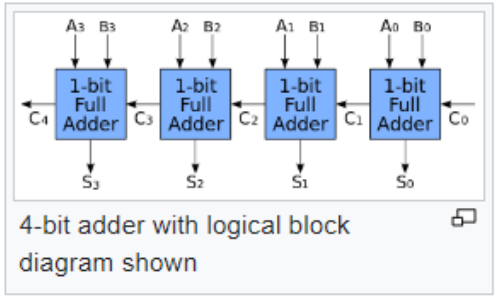
Combinational Logic

- HDL like Verilog allows designers to write higher-level, abstract syntax without specifying logic gate details.

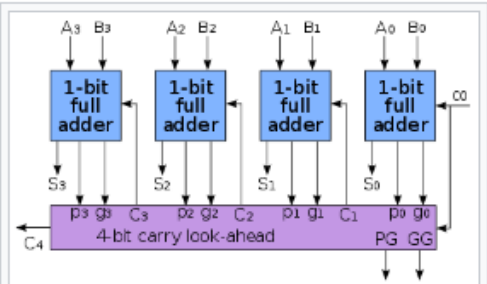
- For example, designer can write "+",
the synthesis tool will automatically select a timing-matching circuit such as a ripple or carry-lookahead adder during backend synthesis.

```
1 wire [3:0] a;
2 wire [3:0] b;
3 wire [3:0] s;
4
5 //    4b   4b   4b
6 assign s = a + b;//no carry-in and truncate carry-out
```

synthesis



4-bit adder with logical block diagram shown



4-bit adder with carry lookahead

Agenda

1. Design intent
2. Code Syntax
3. Advance Syntax

Data Types

- There are different data types in Verilog, some of them can be used in design and some can be used in testbench

Type	Purpose	Used in
wire	Node of combinational circuit	design
	Signal connection	testbench
reg	Node of combinational circuit or sequential element	design
	Variable with a specific bit number	testbench
integer	32-bit signed integer	testbench
time	32-bit unsigned integer	testbench
real	Real number	testbench

Procedural Blocks

- In design
  - **always** block
    - Can have if-else, switch-case inside
    - Do not use for-loop inside always block
      - Although this may be seen in some designs, it is not a good coding style.

- In testbench
  - **initial** block, **always** block
  - Basically, you can write whatever you want
    - It is recommended to use **initial** + **forever** instead of **always**
    - Unless you want to write a behavioral model using the syntax of writing a circuit

Value Assignments

- Both of these descriptions can be used to express combinational circuits.

- "reg" type for always block
  - always@*
  - blocking assignment "="

```
20  reg  [3:0] a;
21  always@*begin
22     a = ...
23  end
24
25
26  reg  [3:0] b;
27  always@*begin
28     b[3:2] = ...
29  end
30  always@*begin
31     b[1:0] = ...
32  end
33
34  reg  [3:0] c;
35  reg  [3:0] d;
36  always@*begin
37     {c,d} = ...
38  end
```

- "wire" type for continuous assignment
  - assign
  - blocking assignment "="

```
 1 wire [3:0] a;
 2 assign a = ...
 3
 4
 5
 6
 7 wire [3:0] b;
 8 assign b[3:2] = ...
 9 assign b[1:0] = ...
10
11
12
13
14
15 wire [3:0] c;
16 wire [3:0] d;
17 assign {c,d} = ...
18
19
```

Value Expression

- Verilog value expression
  - <sign><width>’<base><value>
- Sign
  - + (can be omitted)
  - -
- Width
  - An integer number
- Base
  - b: binary, o: octal, d: decimal, h: hexadecimal
- Value
  - Number expression depends on its base
  - Be careful of values exceeding the range allowed by the signal width.

```
 1 1'b0 //1-bit 0
 2 1'b1 //1-bit 1
 3 1'bz //1-bit z,
 4 1'bx //1-bit x, do not use in design
 5
 6 5'b0 //5'b00000, exact value
 7 5'b1 //5'b00001, exact value
 8 5'bz //5'bzzzzz, will extension
 9 5'bx //5'bxxxxx, will extension
10
11 8'b10010010  //8-bit
12 8'b1001_0010 //can use underline
13
14 12'b100100101101 //12-bit with base binary
15 12'o4455         //12-bit with base octal
16 12'd2349         //12-bit with base decimal
17 12'h92d          //12-bit with base hexadecimal
18
19 -4'd6 //signed value, is 4'b1010
20
21 4'd16 //exceeding the range
22       //do not write this
```

Declaration of Parameter

- The constants declared in the module usually use the keyword parameter.
  - parameter (global)
    - Can be overwritten when the module instance
  - localparam (local)
    - Can not be overwritten

- Naming rule
  - SCREAMING_SNAKE_CASE
  - Avoid case-only differences with variables the module, that might cause problems in the back-end flow.

- Value with unspecified base (is 32-bit unsigned integer)
  - Use for width or index `WIDTH = 4`
- Value with specific width and base
  - Use for circuit value `INIT  = 4'b1111`

```
1  module my_design#(
2     //parameter declaration
3     parameter WIDTH = 4,
4     parameter INIT  = 4'b1111
5     )(
6     input      [WIDTH  -1:0] in   ,
7     output reg [WIDTH  -1:0] out  ,
8     input                    clk  ,
9     input                    rst_n
10    );
11
12    //local parameter declaration
13    localparam STATE_IDLE = 2'd0;
14    localparam STATE_IN   = 2'd1;
15    localparam STATE_CALC = 2'd2;
16    localparam STATE_OUT  = 2'd3;
17    localparam CALC_LOOP  = 3;
18
19    //
20    //...
21    //
22
23 endmodule
```

Declaration of Signal

- Type
  - wire or reg,
  - Depending on the syntax that you used to describe the circuit
- Signed
  - unsigned (can be omitted) or signed (2's complement)
- Width
  - [MSB:LSB]
  - Can be omitted if signal is 1-bit
  - The conventionally accepted coding style is
    - MSB is WIDTH-1
    - LSB is 0
- Naming rule
  - case sensitive
    snake_case is recommended

```
1 wire        [WIDTH-1:0] a1_us;
2
3 wire signed [WIDTH-1:0] a2_s;
4
5 reg         [WIDTH-1:0] a3_us;
6
7 reg  signed [WIDTH-1:0] a4_s;
```

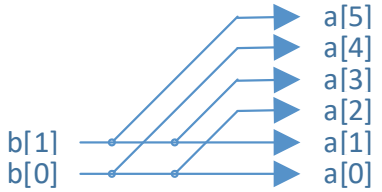
Operation

- Concatenation
  - a = {b,c};

```
1 wire [4:0]a;
2 wire [1:0]b;
3 wire [2:0]c;
4 assign a = {b,c};
```

- Repetition by constant
  - a = {N{b}};

```
7 wire [5:0]a;
8 wire [1:0]b;
9 assign a = {3{b}};
```

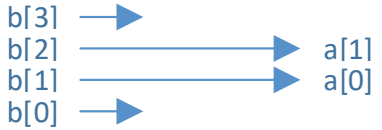


- Bit selection by constant
  - a = b[INDEX];

```
12 wire      a;
13 wire [2:0]b;
14 assign a = b[1];
```

- Partial selection by constant
  - a = b[MSB:LSB];
  - a = b[LSB+:WIDTH];
  - a = b[MSB-:WIDTH];

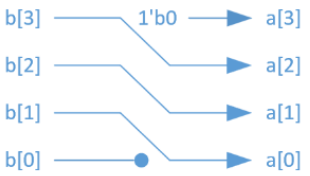
```
17 wire [1:0]a;
18 wire [3:0]b;
19 assign a = b[2:1];
20 assign a = b[1+:2];
21 assign a = b[2-:2];
```



Operation

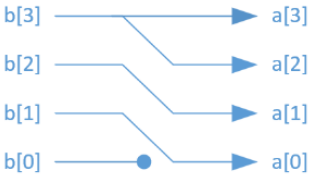
- Bit-wise shift by constant
  - a = b >> N;
    - Divided by 2^N (Add 0 to the MSB)
  - a = b << N;
    - Multiplied by 2^N (Add 0 to the LSB)

```
1 wire          [3:0]a;
2 wire          [3:0]b;
3 assign a = b >>  1;
```

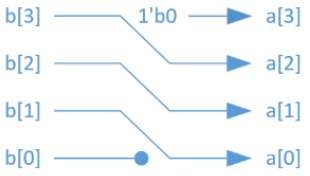


- Arithmetic shift by constant
  - a = b >>> N;
    - If b is signed, divided by 2^N (Extend MSB's signed bit)
    - If b is unsigned, shifted right (Add 0 to the MSB)
  - a = b <<< N;
    - Multiplied by 2^N (Add 0 to the LSB)

```
5 wire signed [3:0]a;
6 wire signed [3:0]b;
7 assign a = b >>> 1;
```




```
 9 wire          [3:0]a;
10 wire          [3:0]b;
11 assign a = b >>> 1;
```



Operation

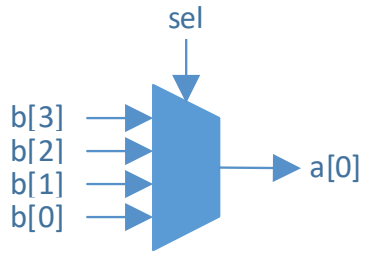
```
232     reg       [3:0]ua;
233     reg       [3:0]ub;
234     reg signed[3:0]sa;
235     reg signed[3:0]sb;
236
237     initial begin
238        ub = 4'b1000;
239        for(int i=0 ; i<5 ; i++)begin
240           ua = ub >> i;
241           $display("ua = b%4b(d%0d) >> %0d = b%4b(d%0d), is /%0d",ub,ub,i,ua,ua,2**i);
242        end
243        $display("");
244
245        ub = 4'b0001;
246        for(int i=0 ; i<5 ; i++)begin
247           ua = ub << i;
248           $display("ua = b%4b(d%0d) << %0d = b%4b(d%0d), is *%0d",ub,ub,i,ua,ua,2**i);
249        end
250        $display("");
251
252        sb = 4'b1000;
253        for(int i=0 ; i<5 ; i++)begin
254           sa = sb >>> i;
255           $display("sa = b%4b(d%0d) >>> %0d = b%4b(d%0d), is /%0d",sb,sb,i,sa,sa,2**i);
256        end
257        $display("");
258
259        sb = 4'b0001;
260        for(int i=0 ; i<5 ; i++)begin
261           sa = sb <<< i;
262           $display("sa = b%4b(d%0d) <<< %0d = b%4b(d%0d), is *%0d",sb,sb,i,sa,sa,2**i);
263        end
264        $display("");
265     end
```

```
ua = 4'b1000(8) >> 0 = 4'b1000(8), is /1
ua = 4'b1000(8) >> 1 = 4'b0100(4), is /2
ua = 4'b1000(8) >> 2 = 4'b0010(2), is /4
ua = 4'b1000(8) >> 3 = 4'b0001(1), is /8
ua = 4'b1000(8) >> 4 = 4'b0000(0), is /16    ←out of range

ua = 4'b0001(1) << 0 = 4'b0001(1), is *1
ua = 4'b0001(1) << 1 = 4'b0010(2), is *2
ua = 4'b0001(1) << 2 = 4'b0100(4), is *4
ua = 4'b0001(1) << 3 = 4'b1000(8), is *8
ua = 4'b0001(1) << 4 = 4'b0000(0), is *16    ←out of range

sa = 4'b1000(-8) >>> 0 = 4'b1000(-8), is /1
sa = 4'b1000(-8) >>> 1 = 4'b1100(-4), is /2
sa = 4'b1000(-8) >>> 2 = 4'b1110(-2), is /4
sa = 4'b1000(-8) >>> 3 = 4'b1111(-1), is /8
sa = 4'b1000(-8) >>> 4 = 4'b1111(-1), is /16  ←out of range

sa = 4'b0001(1) <<< 0 = 4'b0001(1), is *1
sa = 4'b0001(1) <<< 1 = 4'b0010(2), is *2
sa = 4'b0001(1) <<< 2 = 4'b0100(4), is *4
sa = 4'b0001(1) <<< 3 = 4'b1000(-8), is *8    ←out of range
sa = 4'b0001(1) <<< 4 = 4'b0000(0), is *16    ←out of range
```

Operation

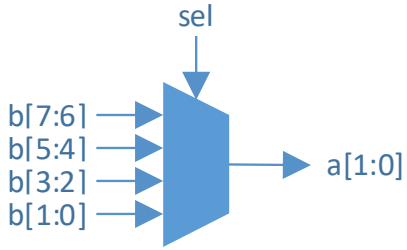
- Bit selection by signal
  - a = b[index];

```
27 wire [1:0]sel;
28 wire      a;
29 wire [3:0]b;
30 assign a = b[sel];
```



- Partial selection by signal
  - a = b[lsb+:WIDTH];
  - a = b[msb-:WIDTH];

```
33 wire [1:0]sel;
34 wire [1:0]a;
35 wire [7:0]b;
36 assign a = b[(sel*2)+:2];
```



Make sure the *representation range* of *index, lsb, msb* are inside the *width of b*

- Otherwise, strange circuits may appear.

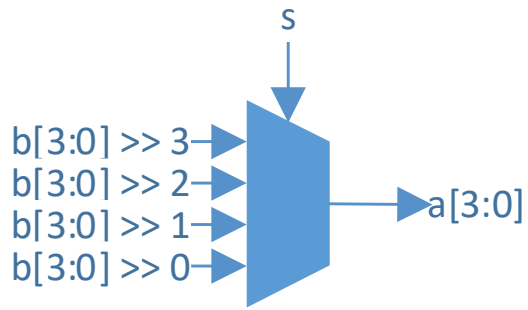
```
12 wire [2:0]sel;
13 wire [1:0]a;
14 wire [7:0]b;
15 assign a = b[(sel*2)+:2];
```

It is safer to use if-else, switch-case to implement multiplexer.

Operation

- Shift by signal
  - a = b >> s;

```
11 wire         [1:0]s;
12 wire         [3:0]a;
13 wire         [3:0]b;
14 assign a = b >>  s;
```



Make sure the *representation range* of *s* is inside the *width of b*

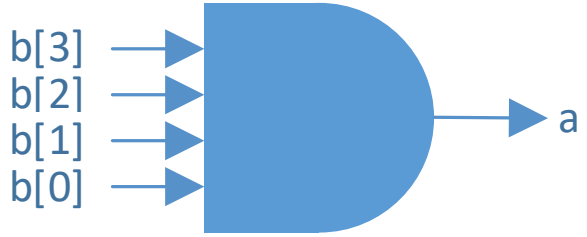
- Otherwise, strange circuits may appear.

It is safer to use if-else, switch-case to implement multiplexer.

Operation

- Reduction operator (return 1-bit)
  - AND a = &b;
  - NAND a = ~&b;
  - OR a = |b;
  - NOR a = ~|b;
  - XOR a = ^b;
  - XNOR a = ~^b;

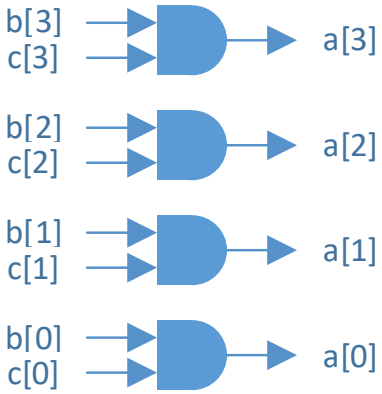
```
1 wire      a;
2 wire [3:0]b;
3 assign a = &b;
```



Operation

- Bit-wise operator for value calculation (can be multi-bit)
  - NOT a = ~b;
  - AND a = b & c;
  - OR a = b | c;
  - XOR a = b ^ c;

```
 7 wire [3:0]a;
 8 wire [3:0]b;
 9 wire [3:0]c;
10 assign a = b & c;
```



- Logical operator for condition expression (return 1-bit)
  - NOT a = !b;
  - AND a = b && c;
  - OR a = b || c;

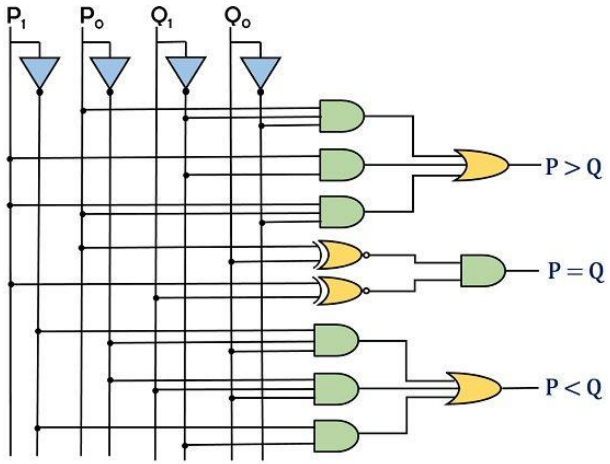
```
14 wire      a;
15 wire      b;
16 wire      c;
17 assign a = b && c;
```

It is recommended that the signals for logical operations are 1-bit.

What if b, c are multi-bit?

Operation

- Relational operator
  - <
  - <=
  - >
  - >=
  - ==
  - !=



Logic Circuit of 2-bit Magnitude Comparator

Electronics Coach

What is Digital Comparator? Magnitude and Identity Comparator - Electronics Coach

```
3 initial begin
4  b = 2'd1;
5  c = 2'd2;
6 end
7
8 always @(posedge clk) begin
9    a <= b <= c;
10 end
```

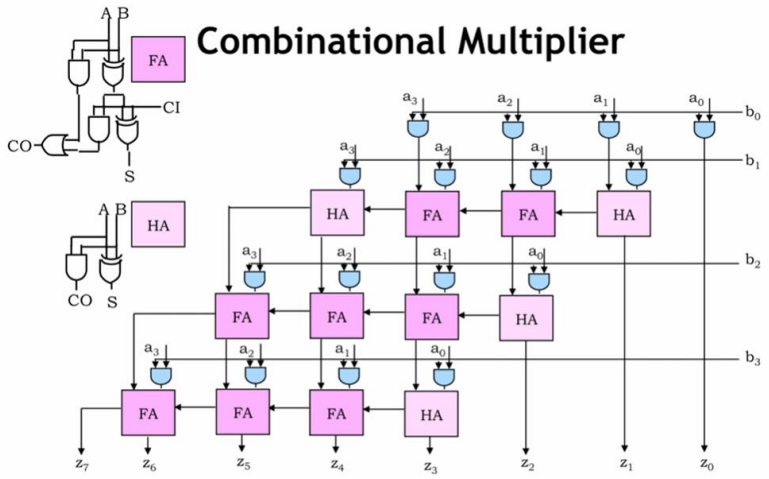
- Exact equality
  - ===
  - check also x and z,
    not synthesizable, only use in testbench
  - Highly recommended for use in testbench when compare values.

```
a = 4'b01zx
b = 4'b01zx
(a ==  b) is x
(a === b) is 1
x = 1'bx
z = 1'bz
(1'b0 ==  x) is x
(1'b1 ==  x) is x
(1'b0 ==  z) is x
(1'b1 ==  z) is x
(1'b0 === x) is 0
(1'b1 === x) is 0
(1'b0 === z) is 0
(1'b1 === z) is 0
```

Operation

- Arithmetic operator
  - a = b + c;
  - a = b – c;
  - a = b * c;

- Attention to the increase of LHS signal width.
  - LHS width of +,- is max_width(b,c)+1 → keep the carry bit
  - LHS width of +,- is max_width(b,c) → truncate the carry bit
  - LHS width of * is width(b)+width(c)



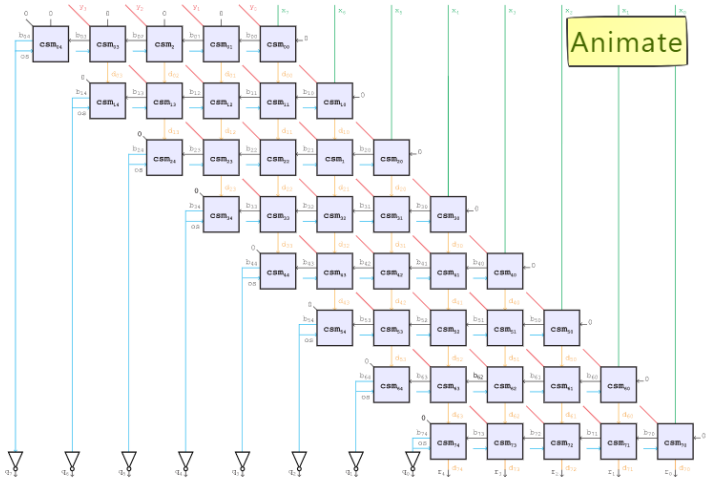
MIT OpenCourseWare - YouTube

Operation

- Arithmetic operator
  - a = b / c;
  - a = b % c;

- These two calculations are more complicated,
  it is recommended not to use them directly in the design
  to avoid unexpected circuits after the synthesis.
  - Will have precision problem or timing violation
  - It is better to describe more details for design intention.

Attempt-subtraction divider



Divider using logic gates - Coert Vonk

Operation Precedence

IEEE Std 1800™-2017 Table 11-2—Operator precedence and associativity

Operator	Associativity	Precedence
() [] :: .	Left	Highest
+ - ! ~ & ~& | ~| ^ ~^ ^~ ++ -- (unary)		
**	Left	
* / %	Left	
+ - (binary)	Left	
<< >> <<< >>>	Left	
< <= > >= inside dist	Left	
== != === !== ==? !=?	Left	
& (binary)	Left	
^ ~^ ^~ (binary)	Left	
| (binary)	Left	
&&	Left	
||	Left	
?: (conditional operator)	Right	
-> <->	Right	
= += -= *= /= %= &= ^= |= <<= >>= <<<= >>>= := :/ <=	None	
{} {{}}	Concatenation	Lowest

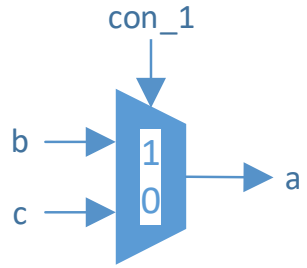
- For complicated descriptions, it is preferred to use parentheses () to indicate the design intent.
- a = |b>>3~^c&d;

Conditional Statement

- If-else

```
1 assign a = (con_1)? b: c;
```

```
16 always@*begin
17    if(con_1)begin
18       a = b;
19    end
20    else begin
21       a = c;
22    end
23 end
```

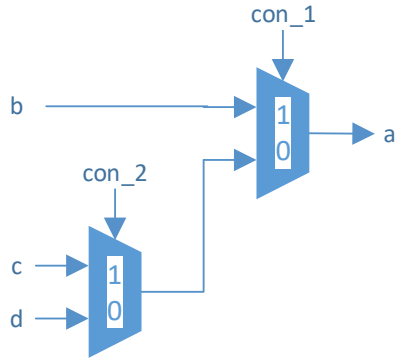


- Nested if-else

```
11 assign a = (con_1)? b:
12            (con_2)? c:
13                     d;
```

```
26 always@*begin
27    if(con_1)begin
28       a = b;
29    end
30    else begin
31       if(con_2)begin
32          a = c;
33       end
34       else begin
35          a = d;
36       end
37    end
38 end
```

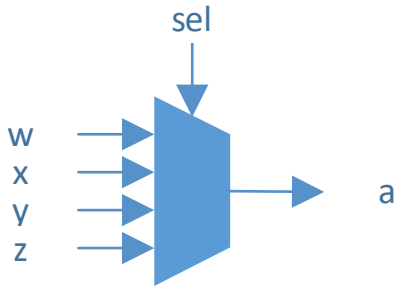
```
40 always@*begin
41    if(con_1)begin
42       a = b;
43    end
44    else if(con_2)begin
45       a = c;
46    end
47    else begin
48       a = d;
49    end
50 end
```



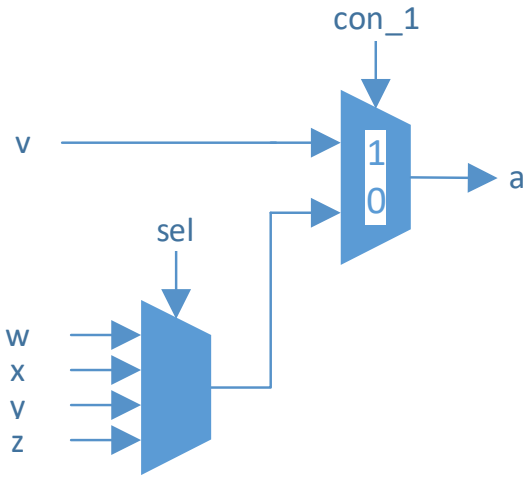
Conditional Statement

- Switch-case
  - Add default value if it is not a full-case mux.
  - It is recommended to write default even if it is a full-case mux in the current design flow.

```
1  always@*begin
2     case(sel)
3        2'b00   : a = w;
4        2'b01   : a = x;
5        2'b10   : a = y;
6        default: a = z;
7     endcase
8  end
9
10
11 always@*begin
12    case(sel)
13       SEL_W   : a = w;
14       SEL_X   : a = x;
15       SEL_Y   : a = y;
16       SEL_Z   : a = z;
17    endcase
18 end
```




```
21 always@*begin
22    if(con_1)begin
23       a = v;
24    end
25    else begin
26       case(sel)
27          SEL_W   : a = w;
28          SEL_X   : a = x;
29          SEL_Y   : a = y;
30          SEL_Z   : a = z;
31       endcase
32    end
33 end
```



Width of Signal

- To prevent the synthesizer from generating unexpected circuits, pay attention to the width of each signal.

- Logical operation or condition expression
  - Operant must be 1-bit

- Arithmetic or bit-wise operation
  - RHS signals maintain the same width as possible
    - Extend to the same width as other RHS signals.
  - LHS signals must have sufficient width to ensure precision.
    - Same width as RHS: truncate
    - RHS width+1: with carry for +,- operation
    - RHS width*2 : for * operation

User Defined function

- Function can represent a combinational circuit in design for reuse. (***synthesizable***)
  - The function name is the return variable
  - Other internal signals are declared as reg type

- Naming rule
  - snake_case
  - The name of the variable in the function should be different from the variable in the module otherwise, might cause problems in the back-end flow.

```
 1 module my_design //...
 2
 3    //...
 4    assign a1 = add_inv(x1,y1);
 5    assign a2 = add_inv(x2,y2);
 6
 7
 8    //...
 9    function [3:0] add_inv;
10       input [3:0] fx;
11       input [3:0] fy;
12       reg   [3:0] fz;
13       begin
14          fz = fx + fy;
15          add_inv = ~fz;
16       end
17    endfunction
18
19 endmodule
```

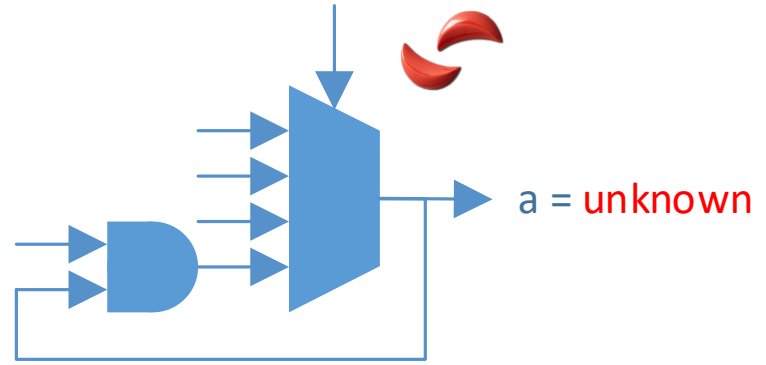
User Defined task

- Task can represent a control sequence in testbench for reuse. (***un-synthesizable***)
  - Can have timing control
  - Can have multiple arguments
    - Inputs (specific values for each call)
    - Outputs (return values for each call)
  - Can have no argument
    - Direct control variables within the module

```
 1 module my_testbench //...
 2
 3    //...
 4    reg      clk;
 5    reg [3:0]a;
 6
 7    //...
 8    initial begin
 9       repeat(5)begin
10          drive_input($random);
11       end
12    end
13
14    //...
15    task drive_input;
16       input [3:0]in;
17       begin
18          @(posedge clk);
19          a <= in;
20       end
21    endtask
22
23 endmodule
```

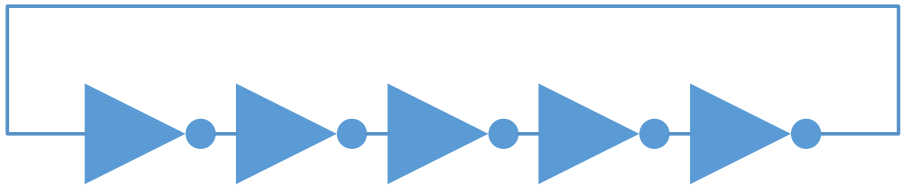
Coding style of Combinational Circuits

- Avoid combinational loop
  - Should not exist in the general cycle-based design
  - Results in unknown values when simulation with 0 gate delay



  - Common loops such as oscillators have special handling during simulation and synthesis.

```
 1 module ring_oscillator#(
 2    parameter INV_NUM = 5
 3    )(
 4    output      ro
 5    );
 6    `ifdef SYNTHESIS
 7       (*DONT_TOUCH = "true" *) wire [INV_NUM-1:0] n;
 8       assign  n[INV_NUM-1:1] = ~n[INV_NUM-2:0];
 9       assign  n[0]           = ~n[INV_NUM-1);
10       assign  ro             =  n[INV_NUM-1];
11    `else
12       reg n;
13       initial begin
14          n = 1'b1;
15          forever begin
16             #100 n = ~n;
17          end
18       end
19       assign ro = n;
20    `endif
21 endmodule
```



Coding style of Combinational Circuits

- Avoid latch
  - level-triggered components are sensitive to noise interference.
  - Difficult to analyze timing in the synthesis stage.
  - Only some special purposes will intentionally use latch.

```
4 reg  c1;//cmb_1
5 always@*begin
6    if(sel) c1 = x;
7    else    c1 = y;//complete statement
8 end              //becomes combinational
```

```
29 reg  c2;//cmb_2
30 always@*begin
31    c2 = y;//add default value at beginning
32    if(sel) c2 = x;
33 end
```

```
22 reg  l0;//latch_0
23 always@*begin
24    if(en) l0 = x;//incomplete statement
25 end             //becomes latch
```

Coding style of Combinational Circuits

- Be aware of the orders of blocking assignment.
  - May lead to inconsistent simulation and synthesis results.
  - It is recommended that only one LHS variable is in an always block.

```
1 always@*begin
2     b = d & e;
3     a = b | c;
4 end
```

- simulation / synthesis match

```
6 always@*begin
7     a = b | c;
8     b = d & e;
9 end
```

- simulation / synthesis mismatch

```
15 always@*begin
16     a = b | c;
17 end
18
19 always@*begin
20     b = d & e;
21 end
```

Design Intent

- Designer clearly express the design intent with high-level description.
  - The low-level implementation details can be determined by the synthesizer

- Designer need to have an awareness of the area and timing delay of the description.
  - If-else, switch-case
    - Multiplexer
  - Operation
    - Multiplier has larger area and timing delay.
  - Shift or concatenation
    - Does not cost logic gate
    - However, too complex assignments can affect the difficulty of physical routing

Agenda

1. Design intent
2. Code Syntax
3. Advance Syntax

Multi-dimensional Array

- Multi-dimensional array
  - Size
    - [FIRST:LAST]
    - The conventionally accepted coding style is FIRST is 0, LAST is SIZE-1

```
1 wire [7:0]byte_array[0:3];
2
3 assign byte_array[0] = ...
4 assign byte_array[1] = ...
5 assign byte_array[2] = ...
6 assign byte_array[3] = ...
```

```
10 reg  [7:0]byte_array[0:3];
11
12 always@*begin
13    byte_array[0] = ...
14 end
15 always@*begin
16    byte_array[1] = ...
17 end
18 always@*begin
19    byte_array[2] = ...
20 end
21 always@*begin
22    byte_array[3] = ...
23 end
```

Generate

- Generate
  - Used to create circuits based on some parameterized conditions or rules.
    - Sub-module instance
    - Assignment or Always block
      - One-dimensional signal segments expression
      - Multi-dimensional array expression

```
1  module my_design#(
2     parameter OUT_INV = 1;
3     )(
4     //...
5     output out,
6     //...
7     );
8
9     wire out_w;//wire before out
10
11    generate
12       if(|OUT_INV)begin:G_OUT_INV
13          assign out = ~out_w;
14       end
15       else begin:G_OUT
16          assign out = out_w;
17       end
18    endgenerate
19
20 endmodule
```

```
24 genvar i;
25 generate
26    for( i=0 ; i<DATA_WIDTH/8 ; i=i+1)begin:G_NUM
27       //
28       assign byte_swap[(i*8) +:8] = data[(DATA_WIDTH-1-(i*8) -:8];
29       //
30       always@*begin
31          if(|data[(DATA_WIDTH-1-(i*8) -:8])begin
32             byte_true[i] = 1'b1;
33          end
34          else begin
35             byte_true[i] = 1'b0;
36          end
37       end
38       //
39       byte_op I_BYTE_OP (a.(byte_swap[(i*8) +:8]), .b(byte_true[i]), c.(byte_out[i]));
40    end
41 endgenerate
```

For Loop vs. Generate

- for loop
  - loop inside always block
  - multiple regs in one block
  - old coding style

```
18 always @(posedge clk or negedge rst_n) begin
19    if (!rst_n) begin
20       for(i=0,i<255,i=i+1) begin
21          ram[i] <= 32'b0;
22       end
23    end
24    else if (wr_en_i) begin
25       ram[addr_i] <= data_i;
26    end
27 end
```

- generate
  - loop outside always block
  - only one reg in one block

```
29 genvar i;
30 generate
31    for(i=0,i<255,i=i+1)begin
32       always @(posedge clk or negedge rst_n) begin
33          if(!rst_n)
34             ram[i] <= 32'b0;
35          else if((addr_i == i) & wr_en_i)
36             ram[i] <= data_i;
37       end
38    end
39 endgenerate
```

Feedback to us

https://forms.office.com/r/DYDu8vLaWN

Thank you!

PUFacademy
A PUFsecurity Alliance

Visit our website: pufacademy.com

Contact us: PUFacademy@pufsecurity.com