Combination logic function can be expressed as:

\[ \text{logic\_output}(t) = f(\text{logic\_inputs}(t)) \]

- Avoid technology dependent modeling; i.e. implement functionality, not timing.
- The combinational logic must not have feedback.
- Specify the output of a combinational behavior for all possible cases of its inputs.
- Logic that is not combinational will be synthesized as sequential.
Synthesizable combinational can have following styles

- Netlist of gate instances and Verilog primitives (Fully structural)
- Combinational UDP (Some tools)
- Functions
- Continuous Assignments
- Behavioral statements
- Tasks without event or delay control
- Interconnected modules of the above
Synthesis tools further optimize a gate netlist specified in terms of Verilog primitives.

Example:

```verilog
class or_nand_1 (enable, x1, x2, x3, x4, y);
    input enable, x1, x2, x3, x4;
    output y;
    wire w1, w2, w3;
    or (w1, x1, x2);
    or (w2, x3, x4);
    or (w3, x3, x4); // redundant
    nand (y, w1, w2, w3, enable);
endmodule
```
General Steps:

- Logic gates are translated to Boolean equations.
- The Boolean equations are optimized.
- Optimized Boolean equations are covered by library gates.
- Complex behavior that is modeled by gates is not mapped to complex library cells (e.g. adder, multiplier).
- The user interface allows gate-level models to be preserved in synthesis.
Example:

```verbatim
module or_nand_2 (enable, x1, x2, x3, x4, y);
    input enable, x1, x2, x3, x4;
    output y;
    assign y = !(enable & (x1 | x2) & (x3 | x4));
endmodule
```

Pre-synthesis

Post-synthesis
Synthesis of Combinational Logic – Behavioral Style

Example:

```verbatim
module or_nand_3 (enable, x1, x2, x3, x4, y);

input enable, x1, x2, x3, x4;
output y;
reg y;

always @ (enable or x1 or x2 or x3 or x4)
  if (enable)
    y = !((x1 | x2) & (x3 | x4));
  else
    y = 1; // operand is a constant.
endmodule
```

*Note:* Inputs to the behavior must be included in the event control expression, otherwise a latch will be inferred.
Example:

```verilog
module or_nand_4 (enable, x1, x2, x3, x4, y);
    input enable, x1, x2, x3, x4;
    output y;
    assign y = or_nand(enable, x1, x2, x3, x4);

function or_nand;
    input enable, x1, x2, x3, x4;
    begin
        or_nand = ~(enable & (x1 | x2) & (x3 | x4));
    end
endfunction
endmodule
```
Example:

```vhdl
module or_nand_5 (enable, x1, x2, x3, x4, y);
  input enable, x1, x2, x3, x4;
  output y;
  reg y;
  always @ (enable or x1 or x2 or x3 or x4)
    or_nand (enable, x1, x2, x3c, x4);

task or_nand;
  input enable, x1, x2, x3, x4;
  output y;
  begin
    y = !(enable & (x1 | x2) & (x3 | x4));
  end
endtask
endmodule
```
Construct to Avoid for Combinational Synthesis

- Edge-dependent event control
- Multiple event controls within the same behavior
- Named events
- Feedback loops
- Procedural-continuous assignment containing event or delay control
- **fork** ... **join** blocks
- **wait** statements
- External **disable** statements
- Procedural loops with timing
- Data dependent loops
- Tasks with timing controls
- Sequential UDPs
Conditional Operator

```verbatim
classical
module mux_4bits(y, a, b, c, d, sel);
  input [3:0] a, b, c, d;
  input [1:0] sel;
  output [3:0] y;
  assign y =
    (sel == 0) ? a :
    (sel == 1) ? b :
    (sel == 2) ? c :
    (sel == 3) ? d : 4'bX;
endmodule
```
CASE Statement

module mux_4bits (y, a, b, c, d, sel);
    input [3:0] a, b, c, d;
    input [1:0] sel;
    output [3:0] y;
    reg [3:0] y;
    always @ (a or b or c or d or sel)
        case (sel)
            0: y = a;
            1: y = b;
            2: y = c;
            3: y = d;
            default: y = 4'bx;
        endcase
    endmodule
if .. else Statement

```verilog
module mux_4bits (y, a, b, c, d, sel);
    input [3:0] a, b, c, d;
    input [1:0] sel;
    output [3:0] y;
    reg [3:0] y;
    always @ (a or b or c or d or sel)
        if (sel == 0) y = a; else
        if (sel == 1) y = b; else
        if (sel == 2) y = c; else
        if (sel == 3) y = d;
    else y = 4'bx;
endmodule
```

Note: *CASE* statement and *if/else* statements are more preferred and recommended styles for inferring MUX
Unwanted Latches

- Unintentional latches generally result from incomplete case statement or conditional branch

- Example: case statement

  ```vhdl
  always @ (sel_a or sel_b or data_a or data_b)
  case ({sel_a, sel_b})
  2'b10: y_out = data_a;
  2'b01: y_out = data_b;
  endcase
  ```

The latch is enabled by the "event or" of the cases under which assignment is explicitly made. e.g. ({sel_a, sel_b} == 2'b10) or ({sel_a, sel_b} == 2'b01)
Example: if .. else statement

```verbatim
class always @ (sel_a or sel_b or data_a or data_b)
  if ({sel_a, sel_b} == 2'b10)
      y_out = data_a;
  else if ({sel_a, sel_b} == 2'b01)
      y_out = data_b;
```
When the branching of a conditional (if) is not mutually exclusive, or when the branches of a case statement are not mutually exclusive, the synthesis tool will create a priority structure.

Example:

```verilog
module mux_4pri (y, a, b, c, d, sel_a, sel_b, sel_c);
    input a, b, c, d, sel_a, sel_b, sel_c;
    output y;
    reg y;
    always @ (sel_a or sel_b or sel_c or a or b or c or d)
    begin
        if (sel_a == 1) y = a; else
        if (sel_b == 0) y = b; else
        if (sel_c == 1) y = c; else
            y = d;
    end
endmodule
```
General Rule: A variable will be synthesized as a flip-flop when its value is assigned synchronously with an edge of a signal.

Example:

```verilog
module D_reg4a (Data_in, clock, reset, Data_out);
    input [3:0] Data_in;
    input clock, reset;
    output [3:0] Data_out;
    reg [3:0] Data_out;
    always @(posedge reset or posedge clock)
        if (reset == 1'b1) Data_out <= 4'b0;
        else Data_out <= Data_in;
endmodule
```
Registered Combinational Logic

- Combinational logic that is included in a synchronous behavior will be synthesized with registered output.

- Example:

```verilog
module mux_reg (a, b, c, d, y, select, clock);
    input [7:0] a, b, c, d;
    output [7:0] y;
    input [1:0] select;
    reg [7:0] y;
    always @(posedge clock)
        case (select)
            0: y <= a; // non-blocking
            1: y <= b; // same result with =
            2: y <= c;
            3: y <= d;
            default y <= 8'bx;
        endcase
    endmodule
```
Shift register can be implemented knowing how the flip-flops are connected

```verilog
always @(posedge clock) begin
    if (reset == 1'b1) begin
        reg_a <= 1'b0;
        reg_b <= 1'b0;
        reg_c <= 1'b0;
        reg_d <= 1'b0;
    end
    else begin
        reg_a <= Shift_in;
        reg_b <= reg_a;
        reg_c <= reg_b;
        reg_d <= reg_c;
    end
end
```

![Shift Register Diagram](image)
Shift register can be implemented using concatenation operation referencing the register outputs

```verilog
module Shift_reg4 (Data_out, Data_in, clock, reset);
  input Data_in, clock, reset;
  output Data_out;
  reg [3:0] Data_reg;
  assign Data_out = Data_reg[0];
  always @ (negedge reset or posedge clock) begin
    if (reset == 1'b0) Data_reg <= 4'b0;
    else Data_reg <= {Data_in, Data_reg[3:1]};
  end
endmodule
```
Verilog Ripple Counter

4-bit Ripple Counter
module ripple_counter (clock, toggle, reset, count);

input clock, toggle, reset;
output [3:0] count;
reg [3:0] count;
wire c0, c1, c2;
assign c0 = count[0], c1 = count[1], c2 = count[2];

always@(posedge reset or posedge clock)
if (reset == 1'b1) count[0] <= 1'b0;
else if (toggle == 1'b1) count[0] <= ~count[0];

always@(posedge reset or negedge c0)
if (reset == 1'b1) count[1] <= 1'b0;
else if (toggle == 1'b1) count[1] <= ~count[1];

always@(posedge reset or negedge c1)
if (reset == 1'b1) count[2] <= 1'b0;
else if (toggle == 1'b1) count[2] <= ~count[2];

always@(posedge reset or negedge c2)
if (reset == 1'b1) count[3] <= 1'b0;
else if (toggle == 1'b1) count[3] <= ~count[3];
endmodule
Verilog Up/Down Counter

Functional Specs.

- Load counter with Data_in when load = 1
- Counter counts when counter_on = 1
  - counts-up when count_up = 1
  - Counts-down when count_up = 0
module up_down_counter (clk, reset, load, count_up, counter_on, Data_in, Count);

input clk, reset, load, count_up, counter_on;
input [2:0] Data_in;
output [2:0] Count;
reg [2:0] Count;

always @(posedge reset or posedge clk)
  if (reset == 1'b1)
    Count = 3'b0;
  else if (load == 1'b1)
    Count = Data_in;
  else if (counter_on == 1'b1) begin
    if (count_up == 1'b1)
      Count = Count +1;
    else
      Count = Count -1;
  end
endmodule