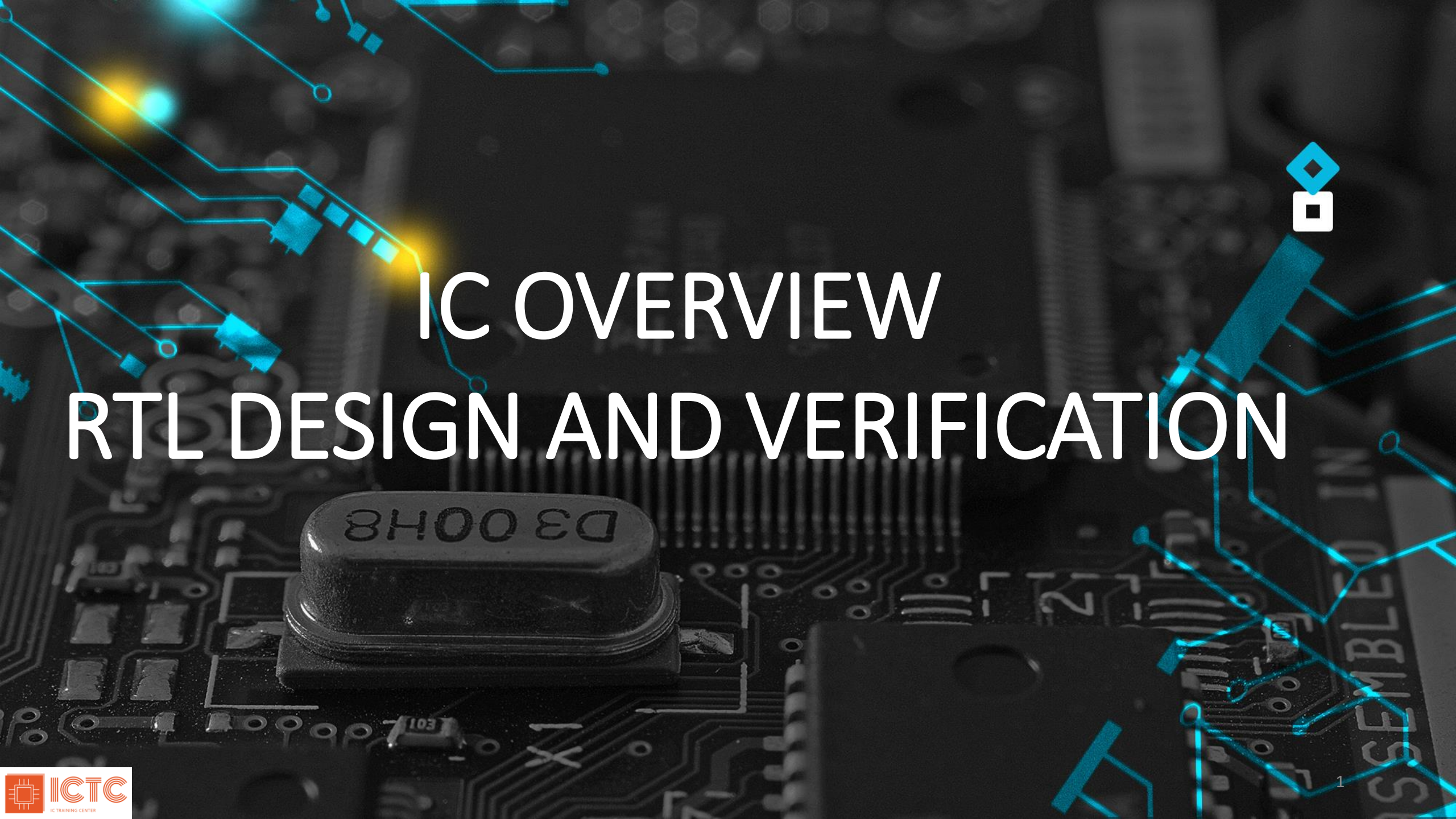
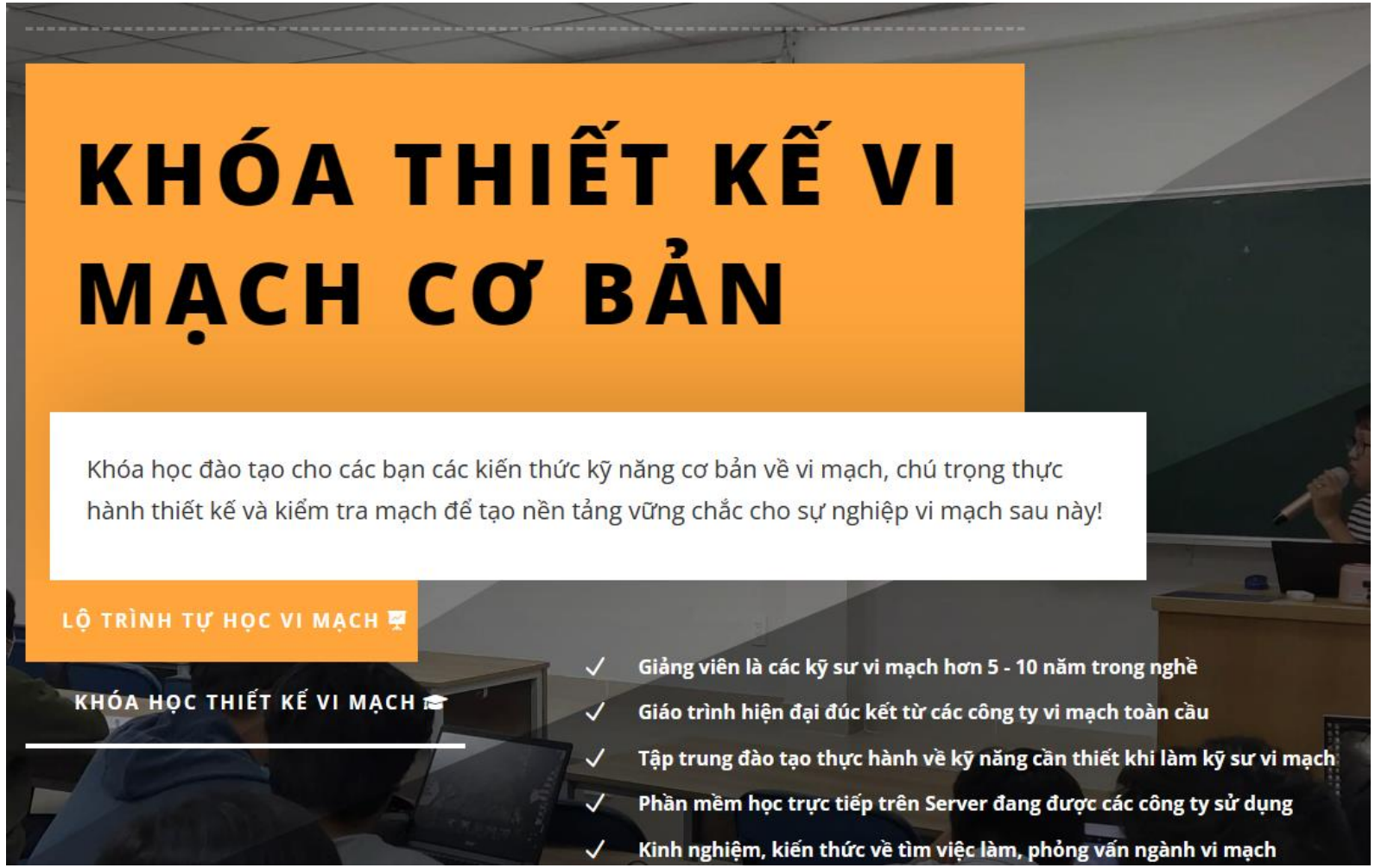


# IC OVERVIEW
# RTL DESIGN AND VERIFICATION

# COURSE INTRODUCTION

[Khóa Học Thiết Kế Vi Mạch Cơ Bản - Trung Tâm Đào Tạo Thiết Kế Vi Mạch ICTC]



# COURSE INTRODUCTION

SUMMARY

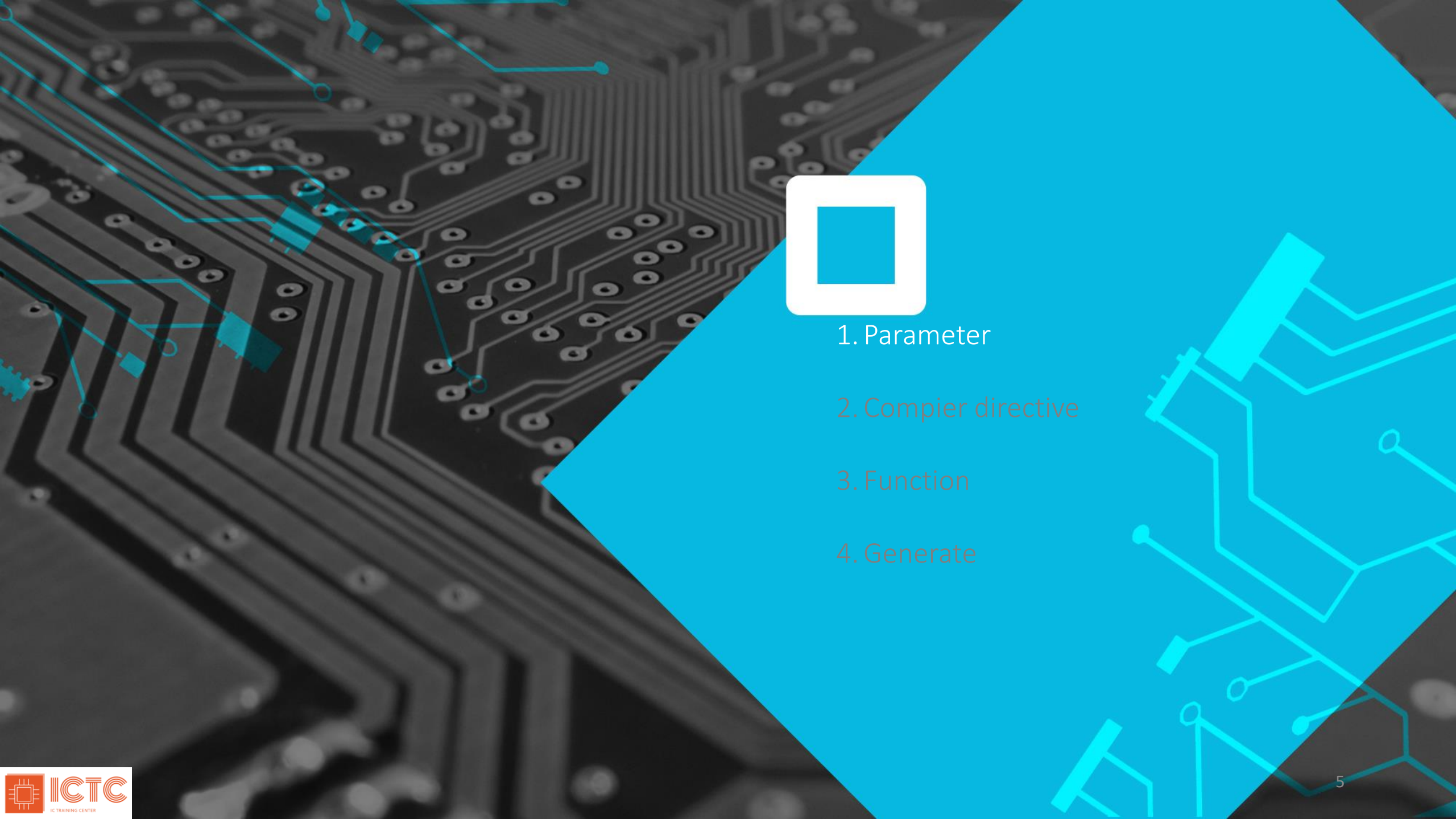
HOMEWORK

QUESTION

SELF-LEARNING

# Session 10: Verilog Fundamental – Part 5 - Parameter

1. Parameter
2. Compiler directive
3. Function
4. Generate

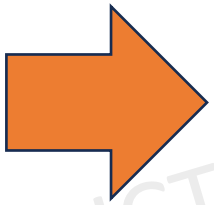


1. Parameter
2. Compier directive
3. Function
4. Generate

# PARAMETER

- Below is our counter example. What if we want to change to 16-bit?

```
module counter(
   input    wire clk,
   input    wire rst_n,
   output  wire overflow,
   output reg [7:0] count
)

   //Combination logic
   always @ (posedge clk or negedge rst_n) begin
      if( !rst_n)
         count <= 8'h00;
      else
         count <= count + 1'b1;
   end

   assign overflow = (count == 8'hff);

endmodule
```



```
module counter(
   input    wire clk,
   input    wire rst_n,
   output  wire overflow,
   output reg [15:0] count
)

   //Combination logic
   always @ (posedge clk or negedge rst_n) begin
      if( !rst_n)
         count <= 16'h00;
      else
         count <= count + 1'b1;
   end

   assign overflow = (count == 16'hff_ff);

endmodule
```

# PARAMETER

- Parameter can be used to improve the readability and reusability.
- **Syntax**: parameter <P_NAME> = <VALUE>;
- 2 most popular usages of parameter

```
//signal declaraion
wire [7:0] cnt_pre;
reg  [7:0] cnt;
```

```
parameter CNT_W = 8;
//signal declaration
wire [CNT_W-1:0] cnt_pre;
reg  [CNT_W-1:0] cnt;
```

```
//constant
if( cnt == 8'hff ) begin
  ...
end
```

```
parameter CNT_MAX = 8'hff;

//constant
if( cnt == CNT_MAX) begin
  ...
end
```

Later on, just need to change the parameter if any design changes (bit-width changes or max value changes



TRY TO USE PARAMETER FOR SENSITIVE BIT-WIDTHS OR CONSTANTS !!!

# PARAMETER

- Parameter must be declared inside module.
- Other modules can have same parameter name, but has different value.
- The parameter only takes effect inside its module

cnt is 8 bit-width in aaa module

```
module aaa ;
  parameter CNT_W = 8;
  …
  reg [CNT_W-1:0] cnt;
endmodule
```

cnt is 16 bit-width in bbb module

```
module bbb ;
   parameter CNT_W = 16;
   ….
   reg [CNT_W-1:0] cnt;

endmodule
```

These 2 modules has same parameter name but different value.

# PARAMETER DECLARATION

- Parameter can be declared as below

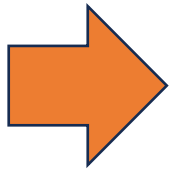
```
module aaa ( <port_list> );
  //parameter list
  parameter PAR1 = VAL_PAR1;
  parameter PAR2 = VAL_PAR2;
  …
  //ports & data type declaration
  //logic description
endmodule
```

```
module aaa  #(parameter PAR1 = VAL_PAR1,
              parameter PAR2 = VAL_PAR2)
  (<port_list>);

  //port & data type declaration
  //logic description
endmodule
```

- The second way is useful if the port declaration is inside the port list

Compiler can not understand the CNT_W

```
module counter
( input   wire   clk ,
  input   wire   rst_n,
  output reg [CNT_W-1:0] cnt
);
  parameter CNT_W = 8;
endmodule
```



```
module counter #(parameter CNT_W=8)
( input   wire   clk ,
  input   wire   rst_n,
  output reg [CNT_W-1:0] cnt
);
  …
endmodule
```

Compiler can understand the CNT_W

# PARAMETER OVERRIDE

- Parameters can be overridden with new values during module instantiation

```
module top #(parameter ADDR_W=8,
             parameter DATA_W=16)
(
   input wire   [ADDR_W-1:0] addr,
   output wire [DATA_W-1:0] data
);
   …


endmodule
```

```
module tb;
   //module instantiation overide
   top #(ADDR_W = 16, DATA_W = 32) u_dut ( <port_list>);
endmodule
```

This way is recommended

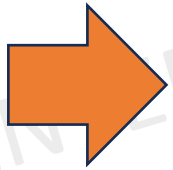
```
module tb;
   //module instantiation
   top u_dut ( <port_list>);
   //Override using defparam
   defparam dut.ADDR_W = 16;
   defparam dut.DATA_W = 32;
endmodule
```

If parameters are not overriden the during module instantiation, they will keep the default values declared the module.

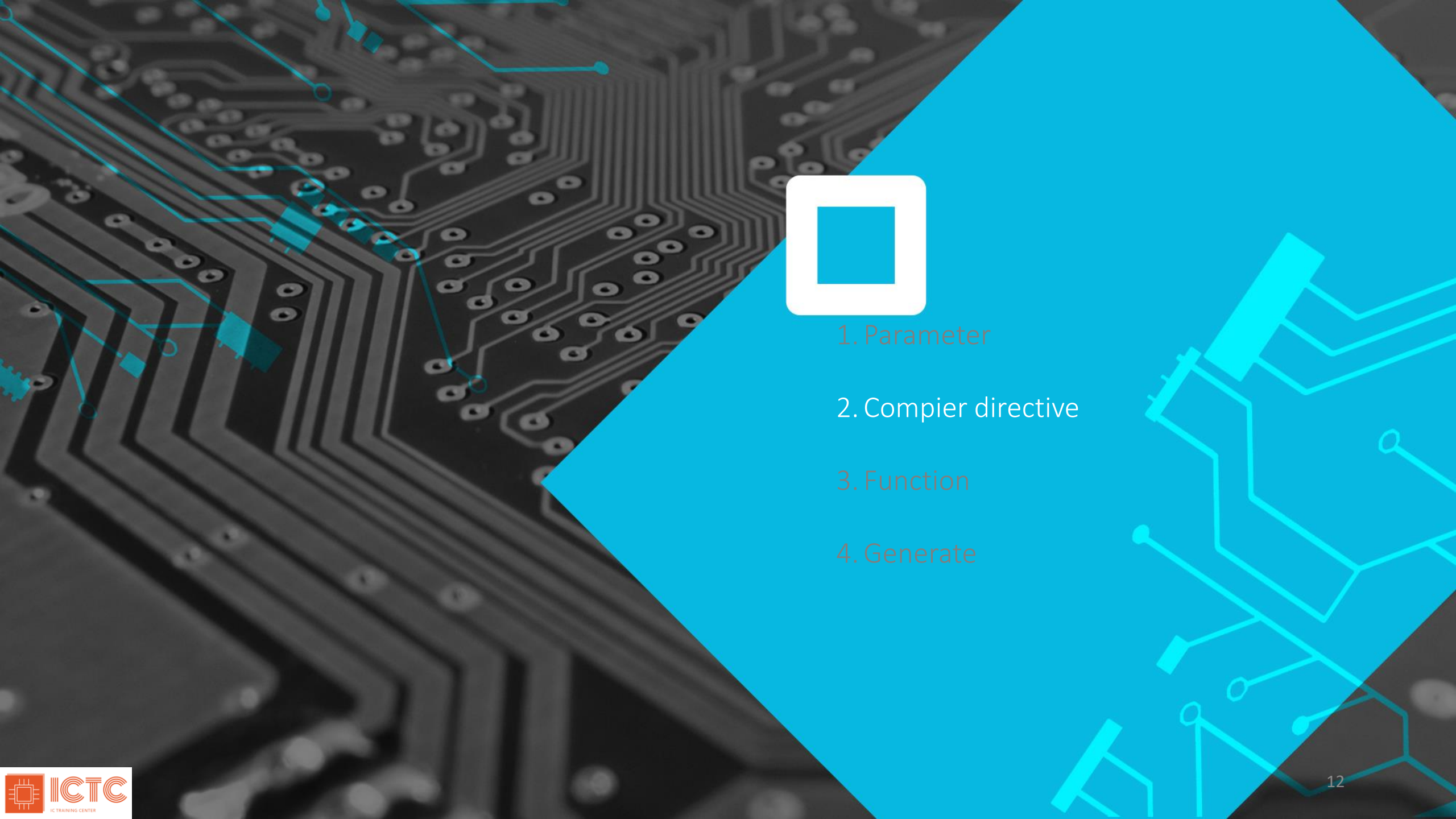
# PARAMETER PASSING

- Parameters can be passed from upper hierarchy.

```
module top #(parameter ADDR_W=8,
                       parameter DATA_W=16)
(
   input wire   [ADDR_W-1:0] addr,
   output wire [DATA_W-1:0] data
);
   …


endmodule
```



```
module tb;
    parameter TB_ADDR_W = 16;
    parameter DATA_W = 32;
   //passing the parameter to module
   top #(.ADDR_W (TB_ADDR_W, .DATA_W(DATA_W)) u_dut ( <port_list>);
   …
endmodule
```



1. Parameter

2. Compier directive

3. Function

4. Generate

# DEFINE

- “define” is a compiler directive used to define macros, which are essentially text substitutions.
- “define” can be used similar to parameter, but it is global, means all the module in the file will be affected by this define

```
`define ADDR_W  8
`define DATA_W  16
module top (
   input wire [`ADDR_W-1:0] addr,
   input wire [`DATA_W-1:0] data
);
   ...
endmodule

module sub (
   input wire [`ADDR_W-1:0] sub_addr,
   input wire [`DATA_W-1:0] sub_data
);
   ...
endmodule
```

Both top and sub
module use same define

# DEFINE

- When using “define”, it is recommended to create a file as shown below

def.v

```
`define ADDR_A  8’h00
`define ADDR_B  8’h04
```

```
module top;
   `include “def.v”
   …
endmodule
```

- Since “define” is a global compiler directive, it’s recommended to use `undef to limit the effective scope.

```
module top;
   `include “def.v”
   …
   `undef ADDR_A
   `undef ADDR_B
endmodule
```

# DEFINE

- “define” can be used to replace text to simplified common logic

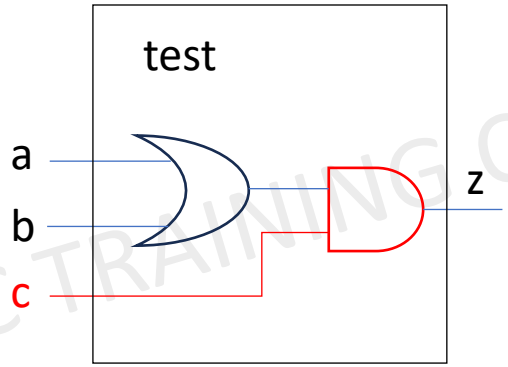
```
`define ff_nrst  always @(posedge clk or negedge rst_n) begin

module counter(
   input    wire clk,
   input    wire rst_n,
   output  wire overflow,
   output reg [7:0] count
)
   `ff_nrst
      if( !rst_n)
         count <= 8'h00;
      else
         count <= count + 1'b1;
   end

   assign overflow = (count == 8'hff);

endmodule
```

```
always @ (posedge clk or negedge rst_n) begin
      if( !rst_n)
         count <= 8'h00;
      else
         count <= count + 1'b1;
end
```

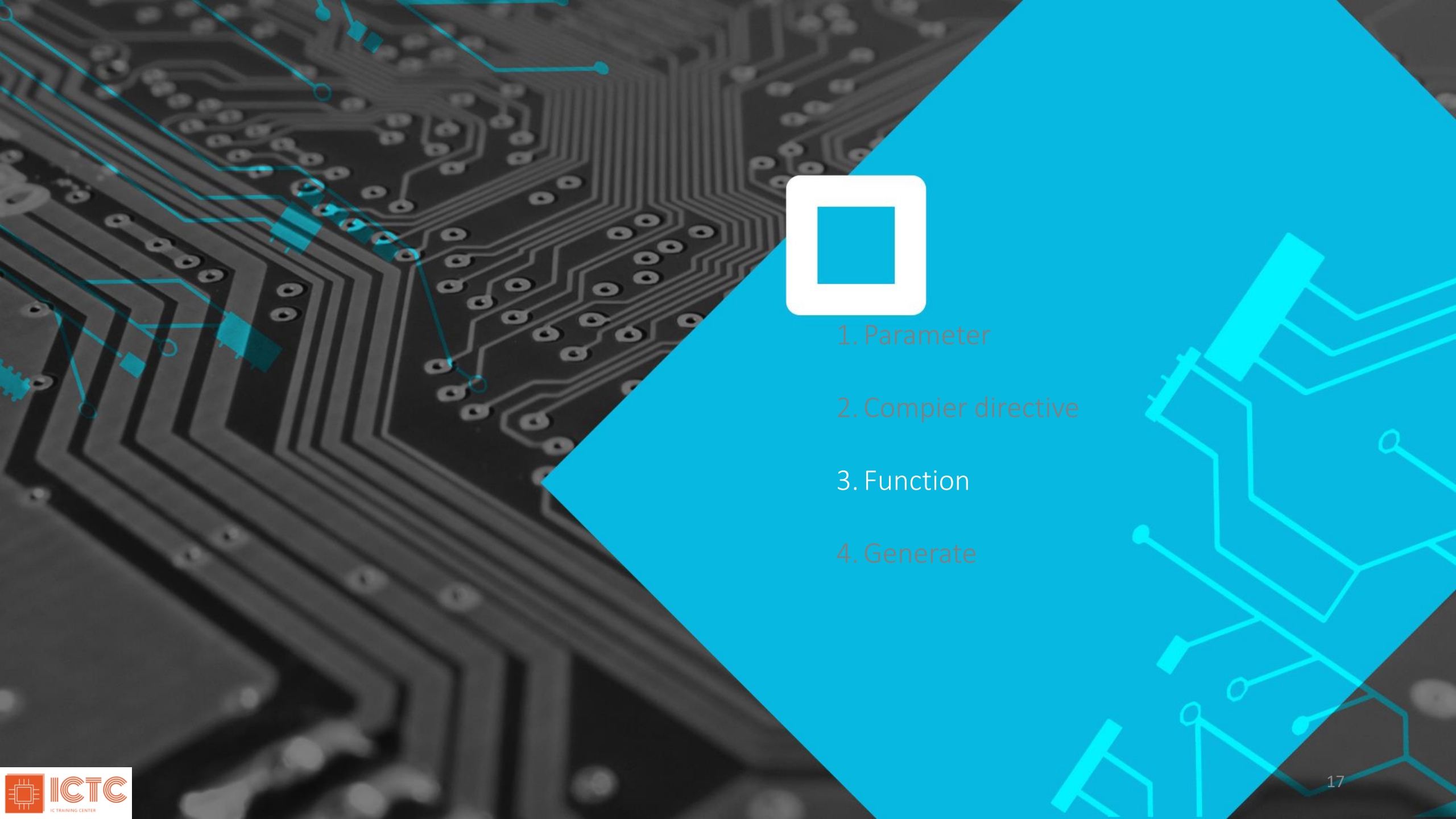
# Conditional compilation

- Compiler directive `ifdef and `endif can be used for conditional compilation

```
module test (
   input wire a ,
   input wire b ,
`ifdef EXTD
   input wire c,
`endif
   output wire z
);

`ifndef EXTD   //EXTD is not defined
   assign z = a | b;
`else
   assign z = (a | b) & c;
`endif
endmodule
```




The red highlighted only available when macro “EXTD” is defined by `ifdef directive



1. Parameter

2. Compier directive

3. Function

4. Generate

# FUNCTION

- Function is a procedural block to create combinational logic.
- Function can be used to split the code to smaller parts that can be reused.

At least 1 input.

Must not have output or inout type

Blocking assignment must be used

```
//How to define a function with 2 arguments
function <range> function_name;
   input   <range>   argument1;
   input   <range>   argument2;

   //internal variable declaration
   begin
      function_name = expression;
   end
endfunction
```

Calling a function:

LHS = function_name( argument1, argument2)

Note: this can be a continuous assignment or procedural assignment

# FUNCTION RULE

- Function can be synthesizable.
- No delay or timing control (*# or @*) can be used inside function.
- Non-blocking assignment is not allowed inside function.
- Function can not have multiple outputs. The return value of function is returned via function_name.
- Function must be written inside module.

Comparison between function and task are described in session 11

# FUNCTION EXAMPLE

Practice: Below is the example in session 7. Now let's try to use function instead of module instance.

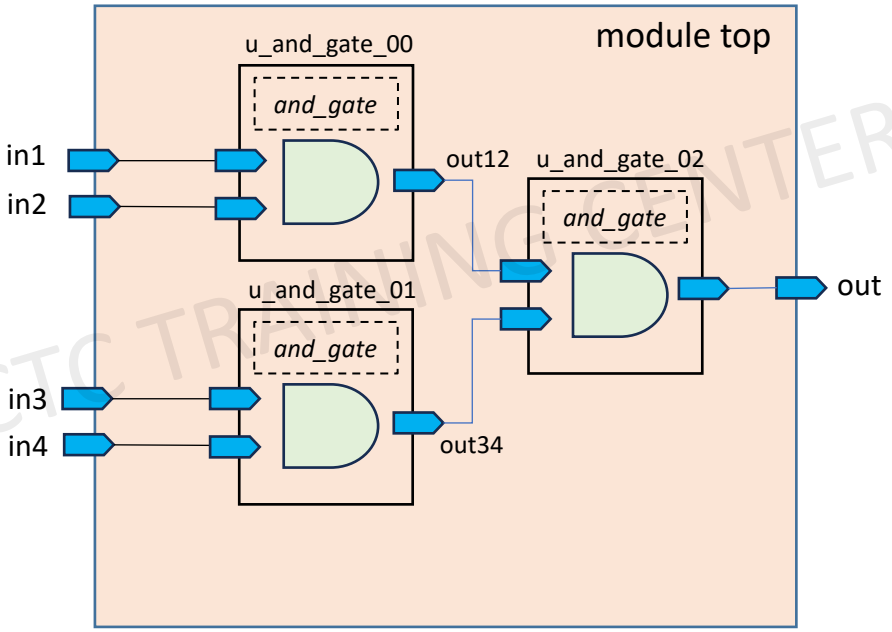
1.Create 10_ss10 folder in your home directory

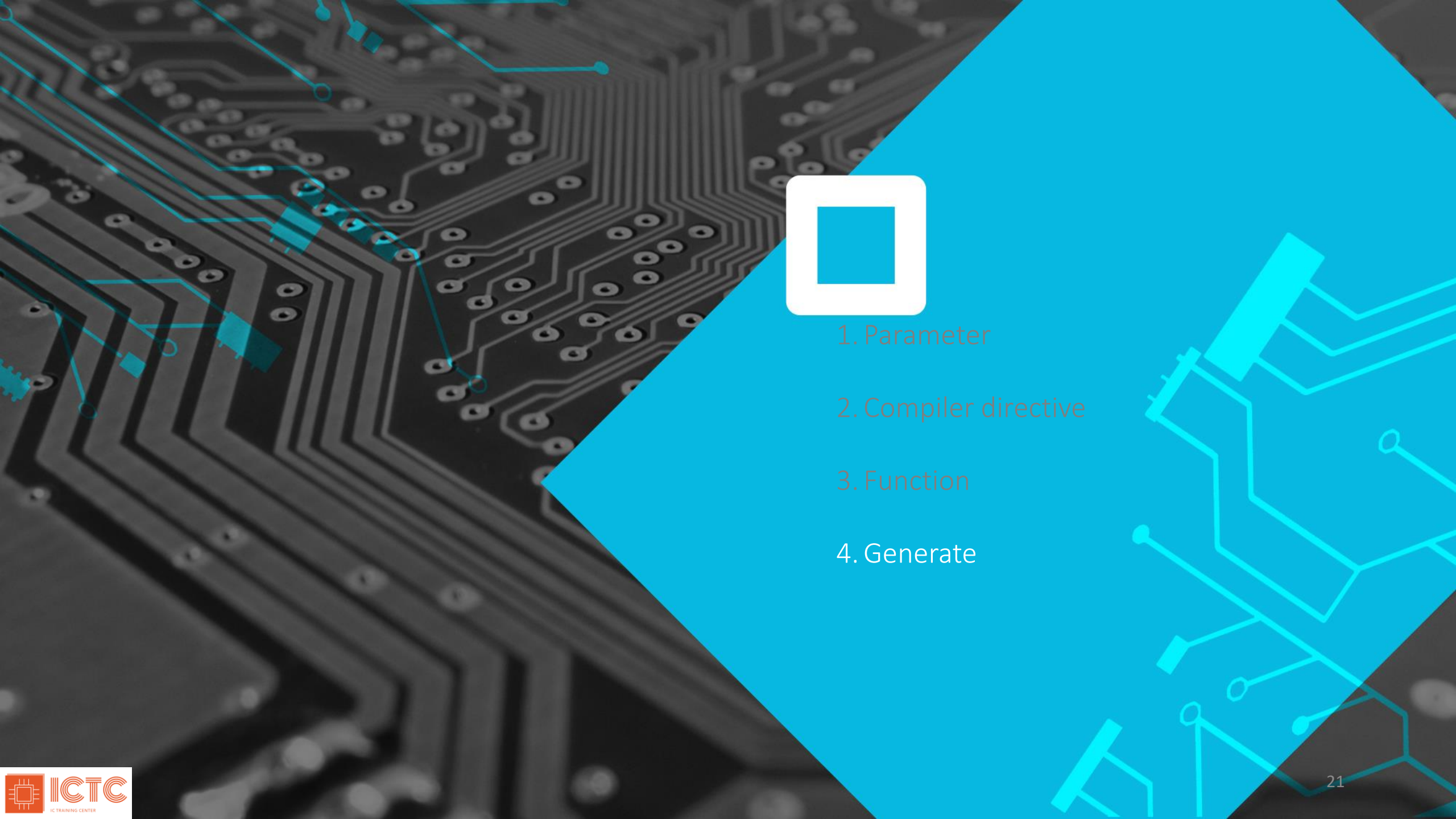
2.Copy /ictc/student-data/share/teacher/10_ss10/and_gate_func to your 10_ss10 folder

3.Create and_gate_func.v and put it under rtl directory

4.Run simulation and check the result.

```
module and_gate
(
   input wire a,
   input wire b,
   output wire c
);
   assign c = a & b;

endmodule
```





1. Parameter
2. Compiler directive
3. Function
4. Generate

# GENERATE

Loop generate constructs allow for moudules with repetitive structure to be described in more simple way.



```
assign c[0] =  a[0] & b[0];
assign c[1] =  a[1] & b[1];
assign c[2] =  a[2] & b[2];
assign c[3] =  a[3] & b[3];
assign c[4] =  a[4] & b[4];
```

These are equivalent

```
genvar i;

generate
   for( i=0; i<5; i++) begin: <name> (optional)
      assign c[i] = a[i] & b[i];
   end

endgenerate
```

The loop index must be declared by "genvar". This index is just for the synthesis tool to know this is generate loop index. It should not be used outside the loop generate

# GENERATE

- Coditional generate construct can be use to select which code is active based on parameter.
- The simulation and synthesis tool only process with the codes corresponding to the parameter value.

```
parameter BIT_DEPTH = 8;

generate
   if( BIT_DEPTH == 8 ) begin
      process_8b u_proc (...);
   end else if( BIT_DEPTH == 10 ) begin
      process_10b u_proc (...);
   end else begin
      process_12b u_proc (...);
   end
endgenerate
```

Only process_8b logic is active, no logic for others function (process_10b, process_12b)

These are equivalent

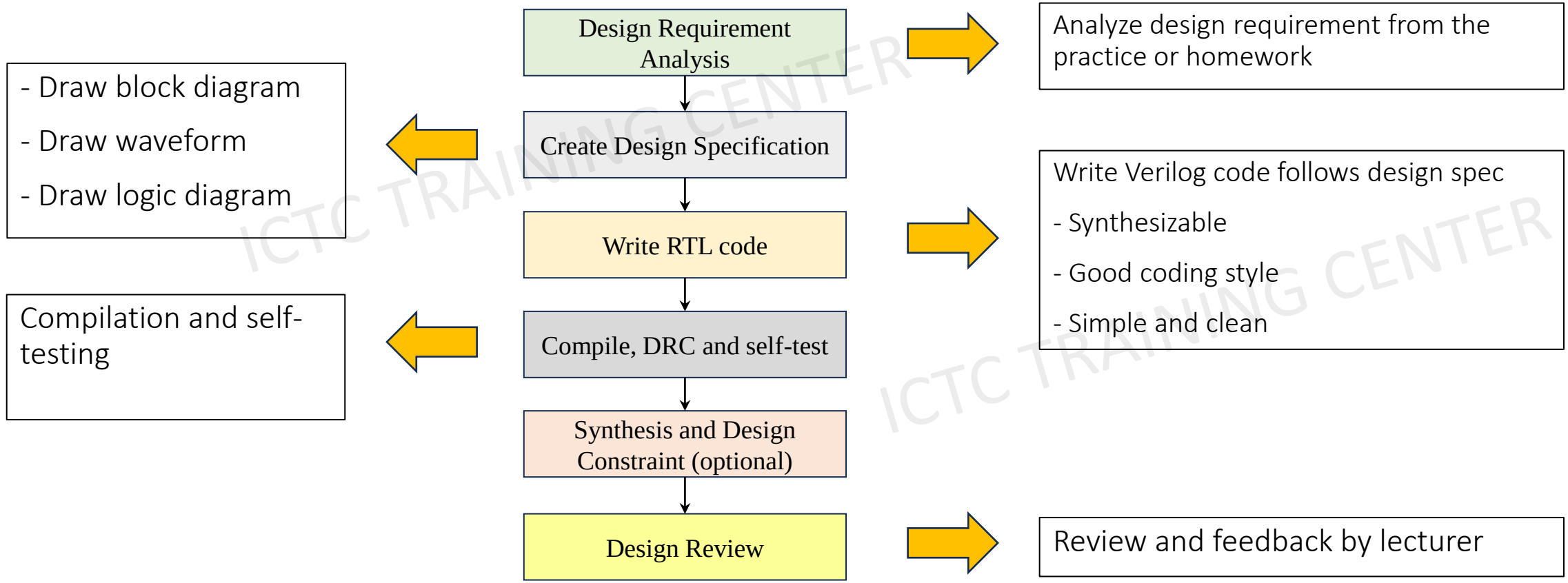
```
parameter BIT_DEPTH = 8;

generate
   case ( BIT_DEPTH )
      8:
        begin
          process_8b u_proc (...);
        end
      10:
        begin
          process_10b u_proc (...);
        end
      default:
        begin
          process_12b u_proc (...);
        end
   endcase
endgenerate
```

CONGRATULATIONS !!!

WE FINISHED ALL THE BASIC VERILOG FOR RTL DESIGN !!!

# RTL DESIGN FLOW RECAP

Let’s review again the RTL Design Flow and see how we can apply into this course



# DESIGN SPECIFICATION EXAMPLE

Let's see again our previous example of counter module.
The later projects in this course need to follow this style.

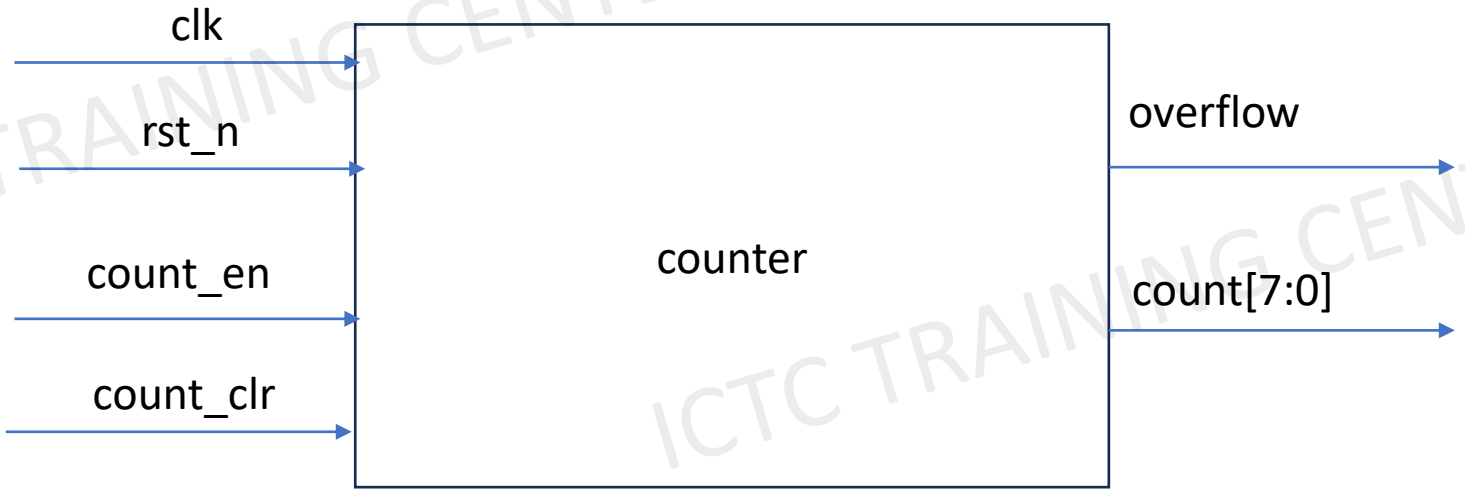


**Part 1: describe feature and requirement:**

- 8-bit counter using D-FF, low active async reset.
- When counter reach max value, it overflowed (overflow = 1) and count again.
- "overflow" is assert only when counter is overflowed and negate after that
- Counter only start counting when input "count_en" is High. Otherwise, keep current value.
- Counter's value is initialized when "count_clr" is High regardless of count_en.

# DESIGN SPECIFICATION EXAMPLE

Part 2: describe block diagram




Block diagram

# DESIGN SPECIFICATION EXAMPLE

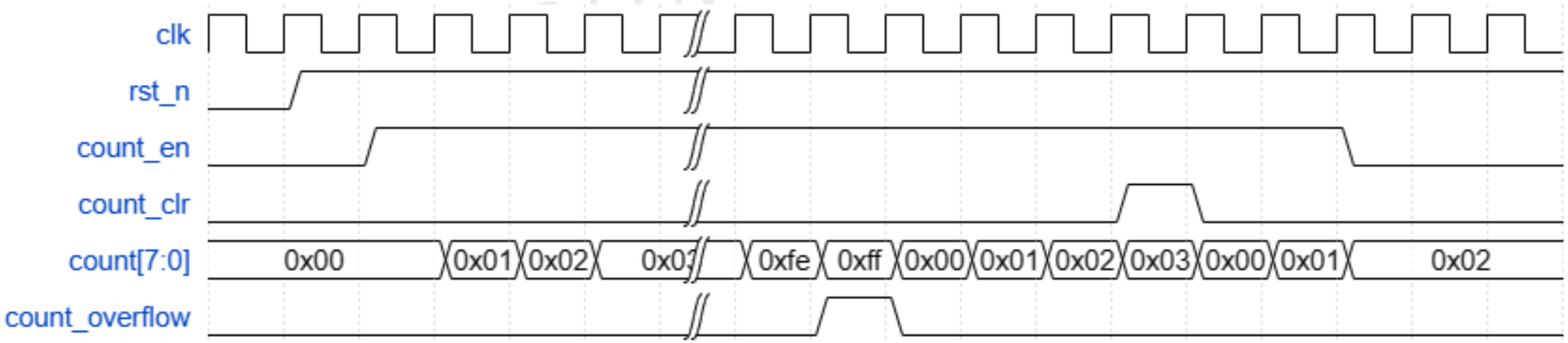
## Part 3: describe IO table

| Signal | Direction | Bit-width | Description |
|---|---|---|---|
| clk | Input | 1 | Input clock |
| rst_n | Input | 1 | Active low asynchronous reset<br>0: counter is in reset state<br>1: counter is in non-reset state |
| count_en | Input | 1 | Counter enable<br>0: counter does not count<br>1: counter counts |
| count_clr | Input | 1 | Counter clear<br>0: counter value is not reset<br>1: counter value is reset |
| overflow | Output | 1 | Counter overflow<br>0: counter is not overflowed<br>1: counter is overflowed |
| count | Output | 8 | Counter value output 0..255 |

Input-Output description

# DESIGN SPECIFICATION EXAMPLE

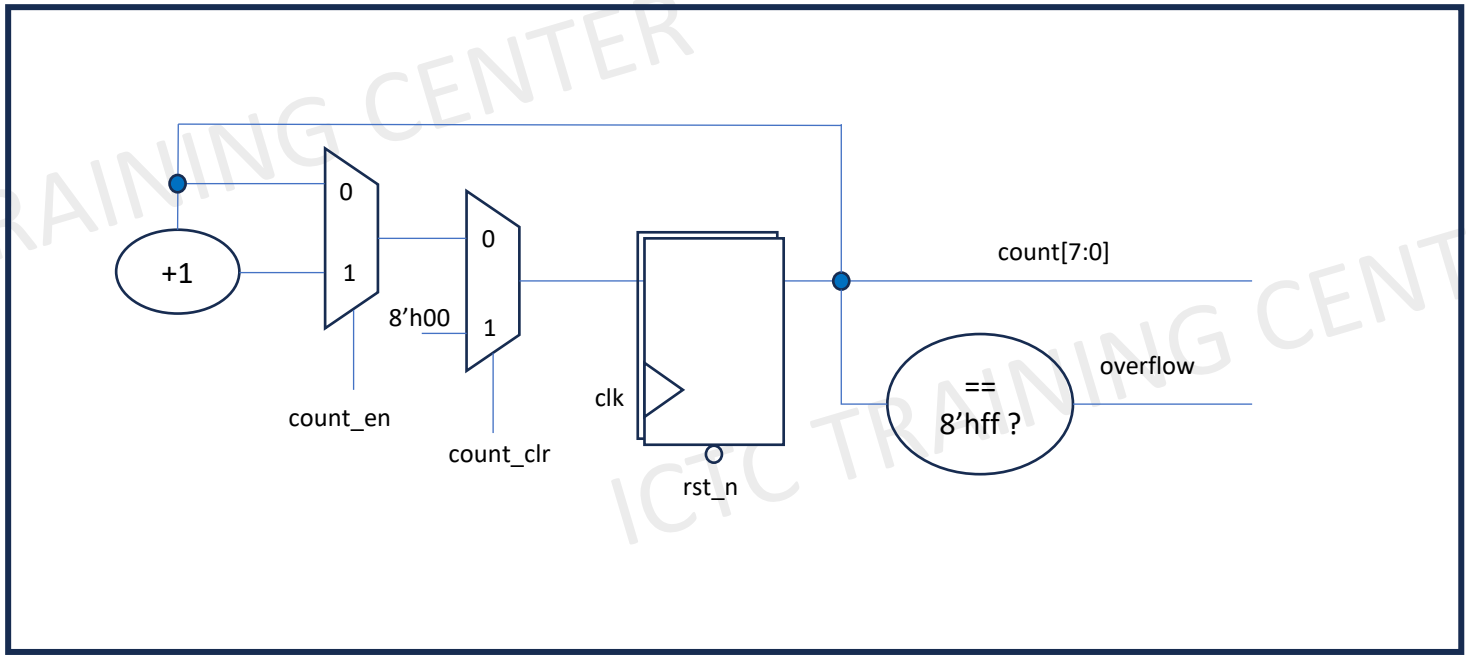
Part 4: describe waveform (use excel, powerpoint, wavedrom)




Waveform (by wavedrom)

# DESIGN SPECIFICATION EXAMPLE

Part 5: describe logic diagram (can use excel, power point, xcircuit)




Logic diagram (by powerpoint)

# SESSION 10
## SUMMARY

SUMMARY:

- ❑ Parameter can be used to improve the readability and reusability.
- ❑ “define” can be used as macro or to replace text to simplified common logic.
- ❑ Function is a procedural block to create combinational logic and can be used to split the code to smaller parts that can be reused.
- ❑ There are 2 types of generate: loop generate of conditional generate.
- ❑ Need to follow the RTL design flow, do specification making before RTL coding !!!

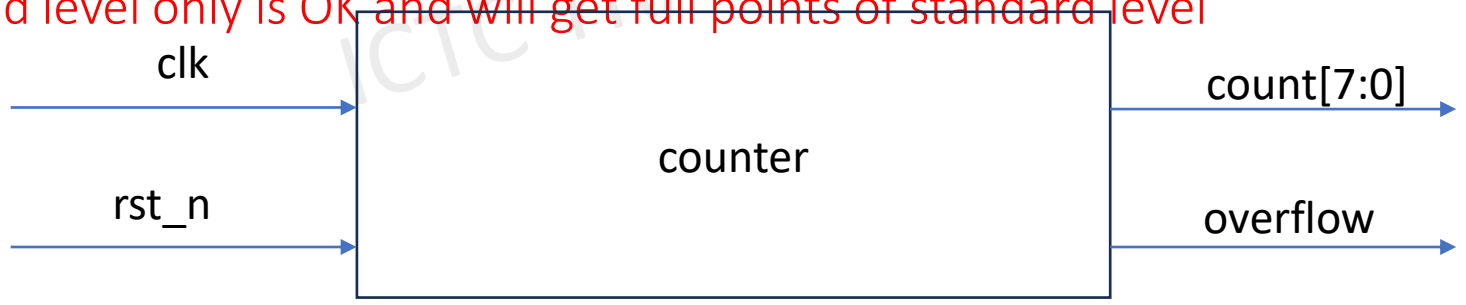
# Session 10

**Homework1**: Make the counter module in ss9 become configurable

- Copy 09_ss9/counter (the simple one without count_en & count_clr) to your 10_ss10
- Make it configurable using parameter CNT_W: 1<= CNT_W <= 16 (counter bit-width)
- Standard level:
  - This design should run OK with current counter environment when CNT_W = 8
  - This design should run OK with 16-bit counter tb put in /ictc/student-data/share/teacher/10_ss10/counter_16b
- Advanced level(*): modify testbench so that it can check with full range of CNT_W. Use 1 tb to check all the design configuration. Only need to change parameter in the tb.

  Example
  - CNT_W=1, cnt is configured as 1-bit counter and tb can check it.
  - CNT_W=4, cnt is configured as 4-bit counter and tb can check it.
  - …

Do only advanced level only is OK ~~and will get full points of standard level~~



# Session 10

**Homework2**: Make the full_adder become configurable

- The homework is placed under 10_ss10 folder in your home directory
- Full adder is configured by parameter N: 2 <= N <= 32
- Standard level: this full_adder can be run successfully in
  - full_adder_2b environment when configure N = 2 (need to change module name to full_adder_2b)
  - full_adder_16b environment when configure N=16 (need to change module name to full_adder_16b)
  - **Note**: those 2 environments need to be stored in your 10_ss10
- Advanced level: modify testbench so that it can check any N configuration. Use 1 tb to check all the design configuration. Only need to change parameter in the tb.

  Example
  - N=2: full_adder is configured as 2-bit full adder and tb can check it
  - N=5: full_adder is configured as 5-bit full adder and tb can check it
  - …

Do only advanced level only is OK and will get full points of standard level