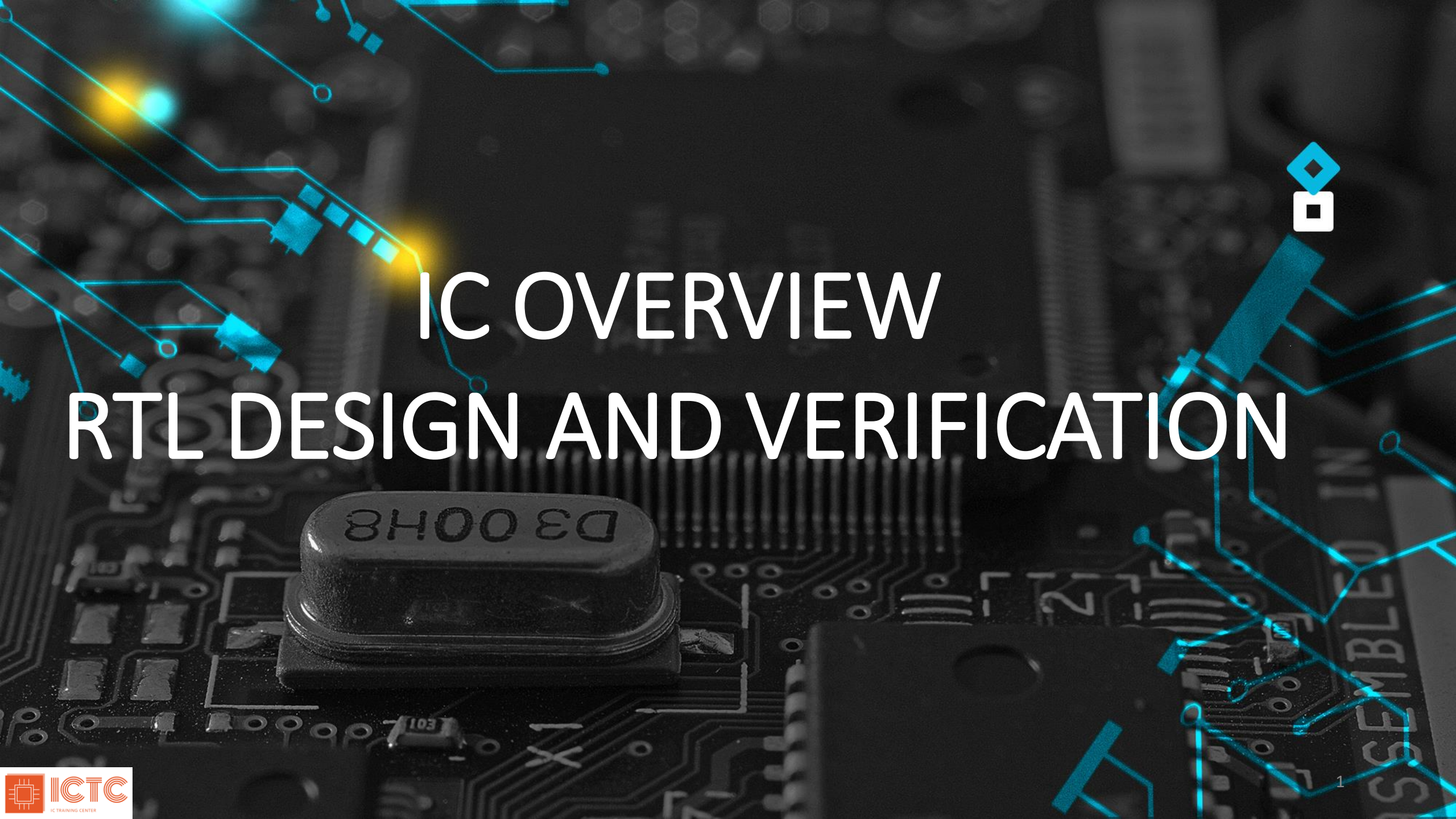
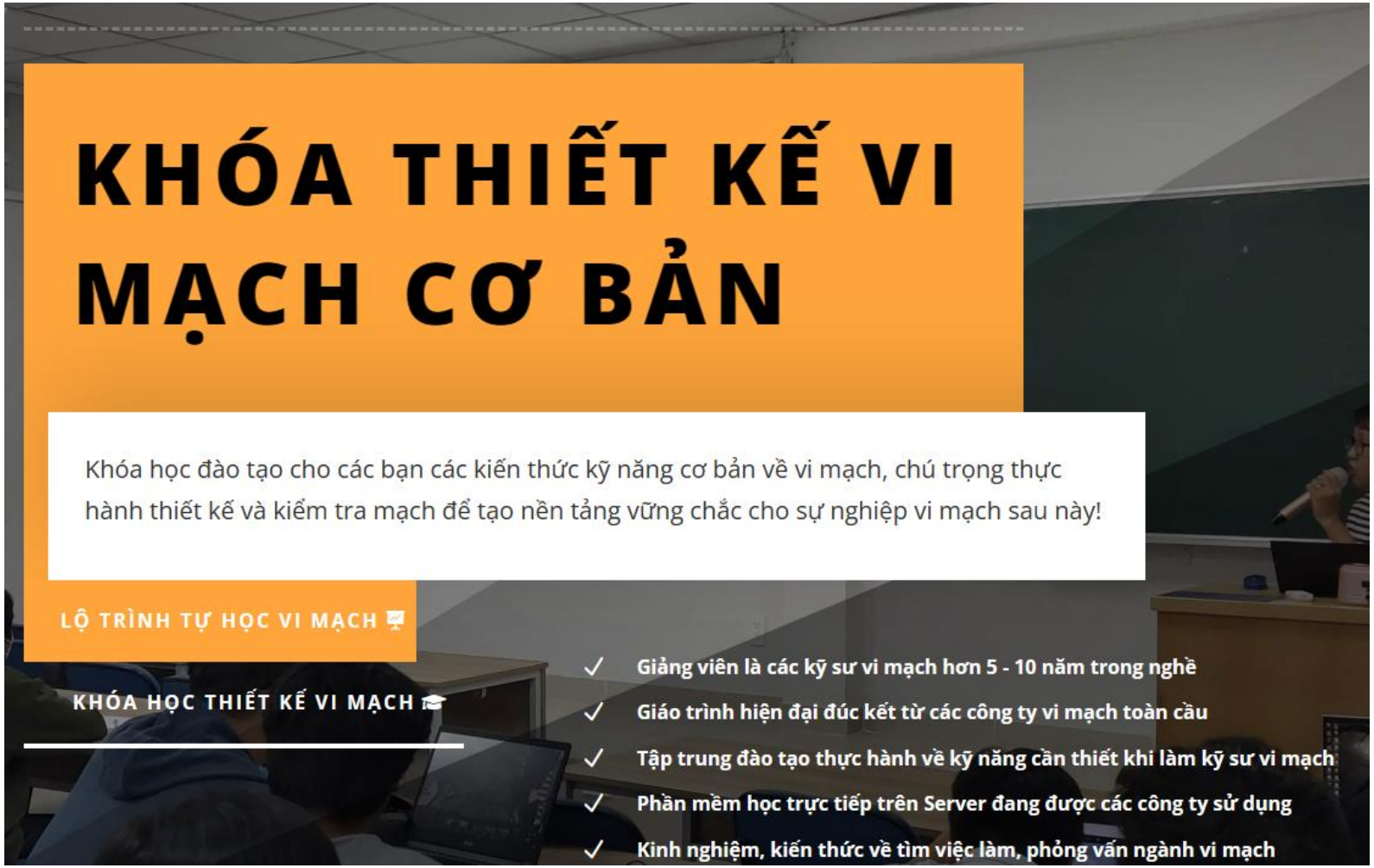


IC OVERVIEW
RTL DESIGN AND VERIFICATION

COURSE INTRODUCTION

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



COURSE INTRODUCTION

SUMMARY

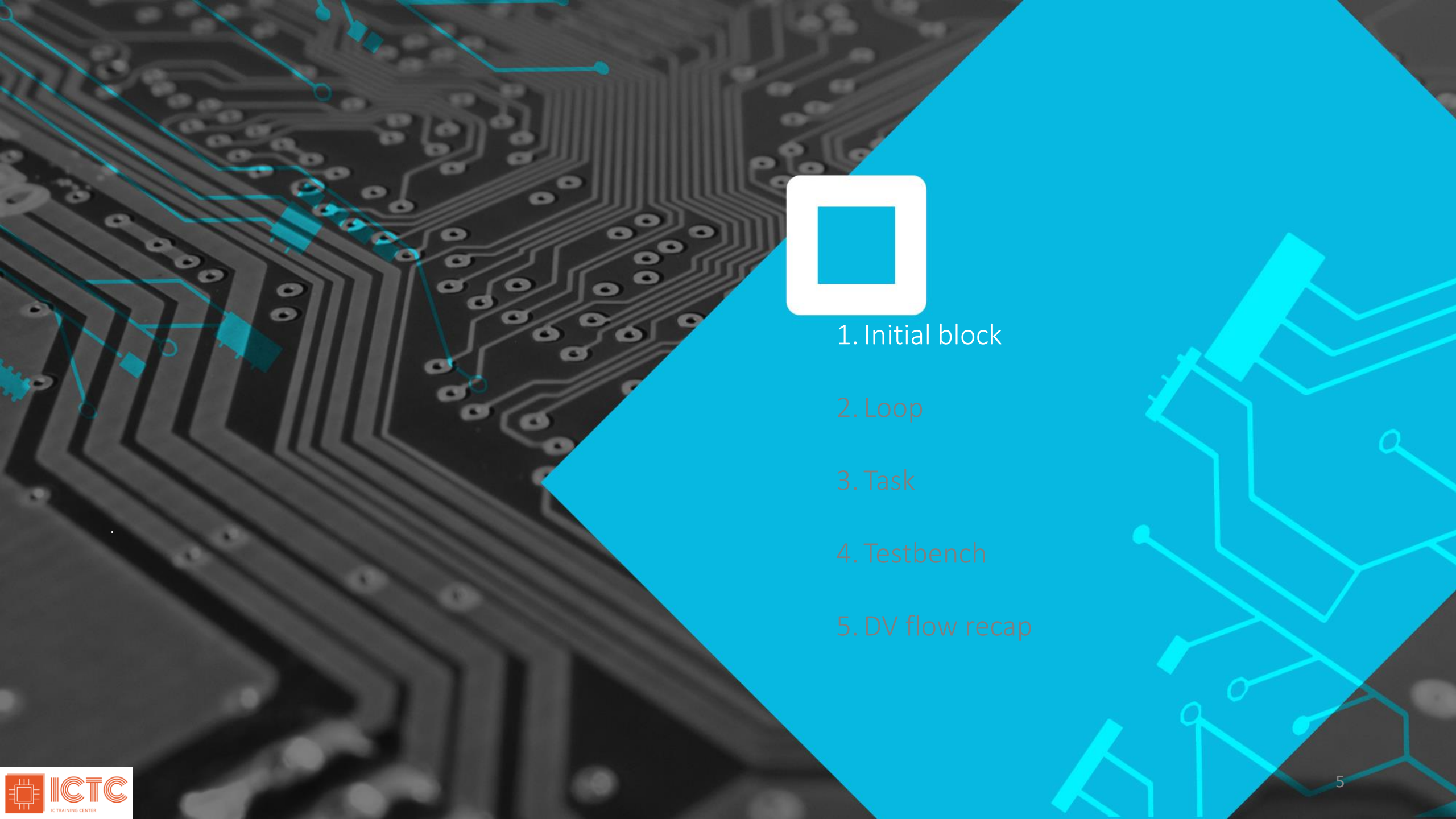
HOMEWORK

QUESTION

SELF-LEARNING

Session 11: Verilog Fundamental – Part 5 – Verilog For Verification

1. Initial block
2. Loop
3. Task
4. Testbench
5. DV flow recap



1. Initial block
2. Loop
3. Task
4. Testbench
5. DV flow recap

Initial Block

- Initial block is procedural block.
- Initial block is enabled at time 0 and is executed only once until all statements are finished.

```
initial
   statement0

initial begin
   statement1
   statement2
   statement3
end
```

Statement1 is executed at time 0. Statements are executed in-order.

- Statement0 and statement1 are executed at time 0. Which one is executed first is based on tool vendor.
- Initial block is un-synthesizable. It is often used for simulation purpose.
- Initial block MUST be written inside module.

Initial Block And Delay Statement

- Use delay statement (#) in initial block to generate timing dependencies.
- Below 2 initial block run parallel at time 0.
- The first initial block finishes first. The second initial block finishes after 90ns and terminate the simulation

```
module tb

  reg a;
  initial begin
    a = 0;
    #10 a = 1;
    #30 a= 0;
  end

  initial begin
    #90 $finish;
  end
endmodule
```

[*0ns*]: a gets value 0

[*10ns*]: a gets value 1

[*40ns*]: a gets value 0

[*90ns*]: end simulation

What if the second initial block as the delay of 30ns only?

$finish is a Verilog system task that tells the simulator to end the current simulation.

Initial Block And Delay Statement

Questions: what's the value of a[1:0] at 0ns, 10ns, 20ns, 30ns?

```
reg [1:0] a;
initial begin
   a <= 2'b00;
   a <= #20 2'b01;
   a <= #10 2'b10;
end
```

Initial Block And Delay Statement

Questions: what's the value of a[1:0] at 0ns, 10ns, 20ns, 30ns?

```
reg [1:0] a;
initial begin
   a <= 2'b00;
   a <= #20 2'b01;
   a <= #10 2'b10;
end
```

[*0ns*]: a[1:0] gets value 2'b00

[*10ns*]: a[1:0] gets value 2'b10

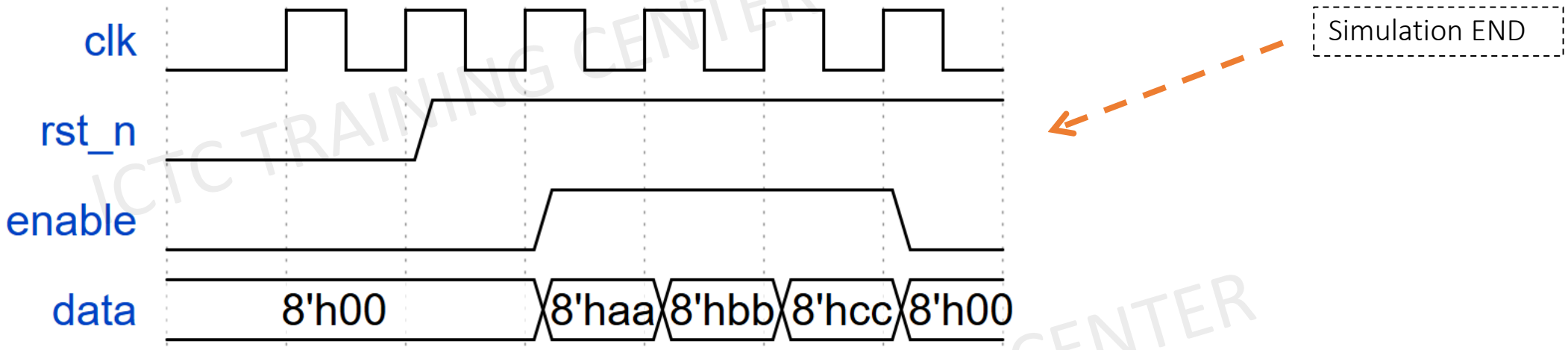
[*20ns*]: a[1:0] gets value 2'b01

[*30ns*]: a[1:0] keeps value 2'b01

Do not use non-blocking assignment in the initial block

Initial Block And Delay Statement

Practice: generate below waveform in testbench using initial block.
Clock period is 100ns



INITIAL VS ALWAYS

Initial	Always
In this, each block assignment executes in the 0 simulation time and continues for the next specified sequence	In this, each block assignments continues to execute in simulation time 0 and repeats forever depending on the sensitivity list event
This block is executed only once	The simulation in this block continues forever. If wait construct is there then it will be held during simulation session
It is non-synthesizable construct	It is synthesizable construct

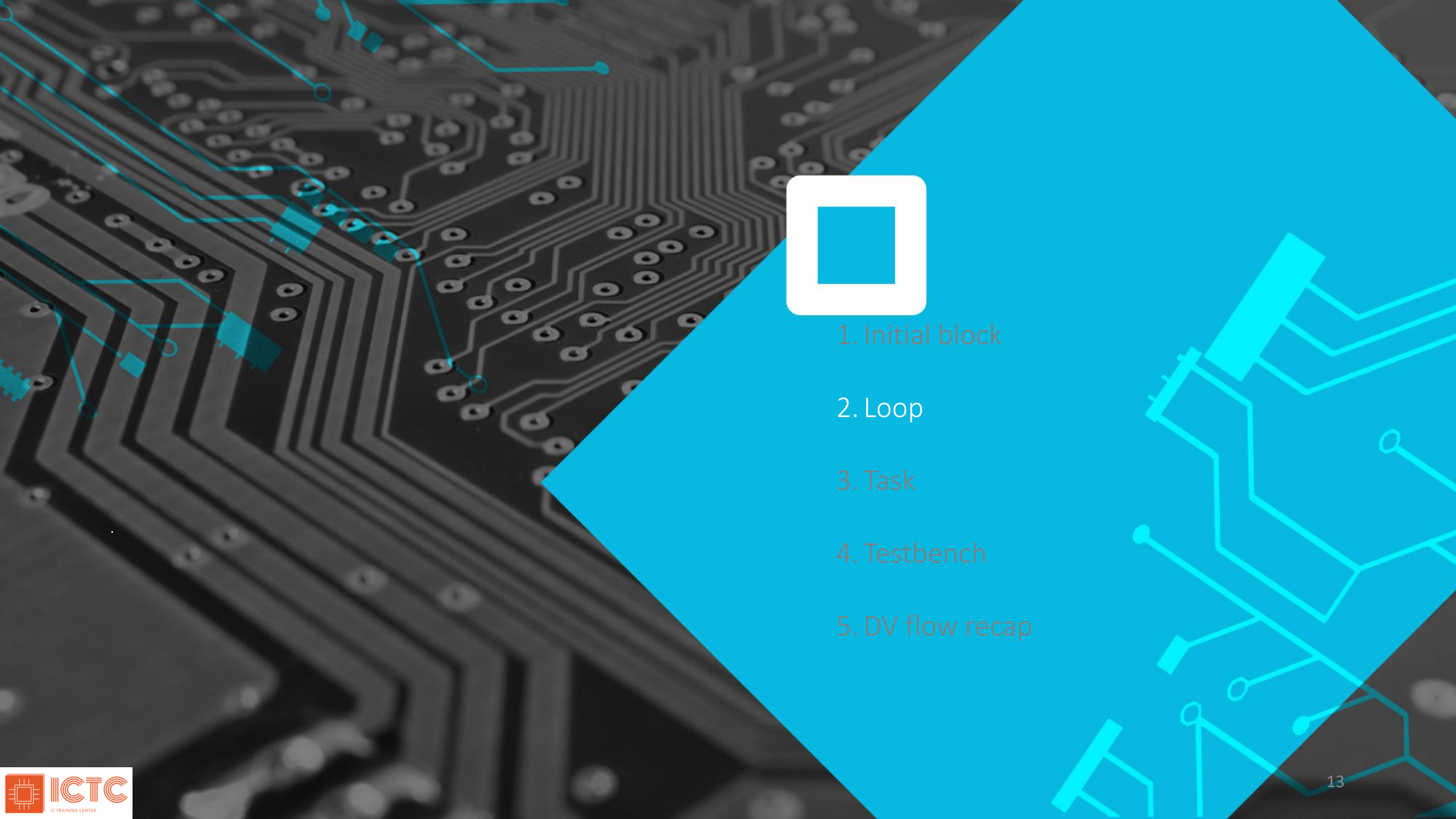
Wait and @

Beside "#", we have other timing control method: "wait" or "@"

- wait statement:
  - **Syntax**: wait (condition)
  - Wait statement pause the execution of current procedure until the condition is true.
- @: event control operator
  - **Syntax**: @(event expression)
  - Used to wait for specific events occur, such as changes in signal values.

```
initial begin
   …
   wait( ready == 1'b1);  //wait ready become 1 before sending data
   send_data(…);
end
```

```
initial begin
   …
   @( posedge clk); //wait clk rising edge
   …
   @(data);    //wait for any change in data
end
```



1. Initial block
2. Loop
3. Task
4. Testbench
5. DV flow recap

FOR LOOP

- For loop can be used in initial block.
- Syntax:

for (initial_condition; condition; step_assignment) begin

..statement..

end

```
module test_bench;
  integer   i;

  initial begin
    //Note that i++ operator does not exist in Verilog,
    //Only support in Systemverilog
    for(i = 0; i<10; i=i+1) begin
      $display("Current loop %0d",i);
    end
  end

endmodule
```

```
Current loop 0
Current loop 1
Current loop 2
...
Current loop 9
```

REPEAT

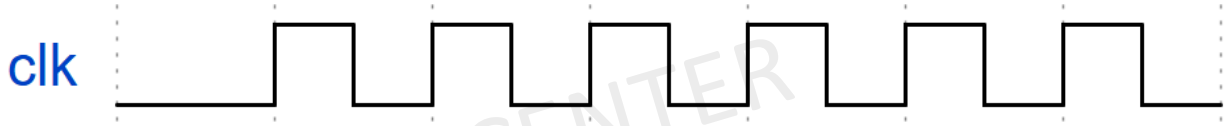
- Repeat can be used in initial block, to execute a set of statemens N times.
- Syntax:

repeat (number)
    Single statement

repeat (number) begin
    Multiple statements
end

```
initial begin
    clk = 0;
    #50;
    repeat (6) #50 clk = ~clk;
end
```

Clock of above practice can be generated using repeat




```
initial begin
    …
    repeat (5) @(posedge clk);
    …
end
```

Wait 5 posedge clk

Then do something

WHILE

- While are looping constructs that execute the given set of statements as long as the condition is true
- Syntax:

while (condition) begin

    [statements]

end

```
initial begin
   while( cnt < 10) begin
      $display("cnt = %d", cnt);
      cnt = cnt+1;
   end
   $display("End loop");
end
```

```
cnt = 0
cnt = 1
cnt = 2
...
cnt = 9
End loop
```

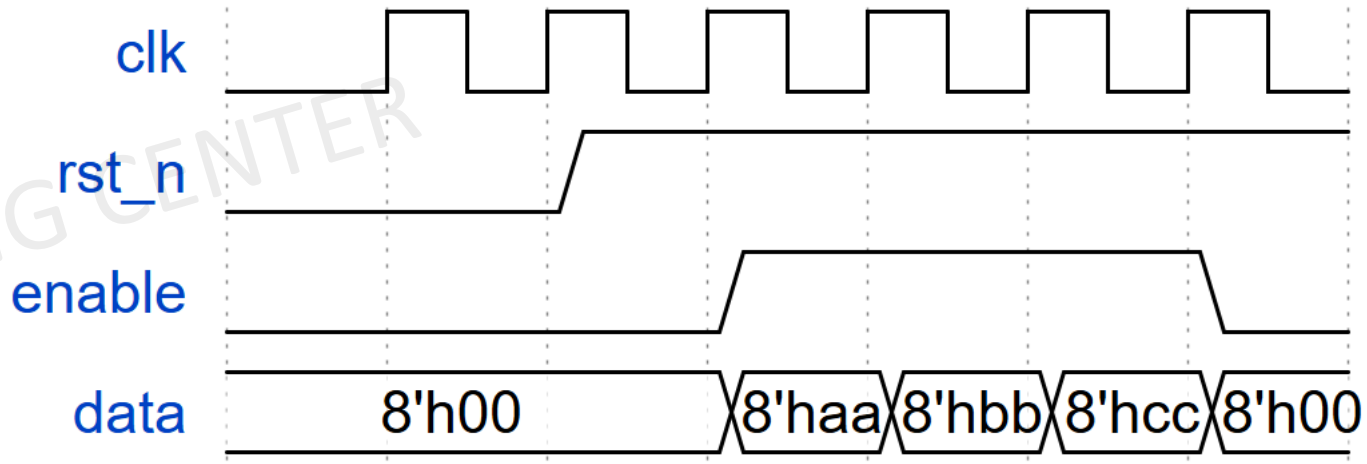
```
initial begin
   while( data_valid == 1'b0);

   $display("data is valid");
end
```

```
Wait data_valid signal = 1'b1
Then continue
```

Initial Block And Delay Statement

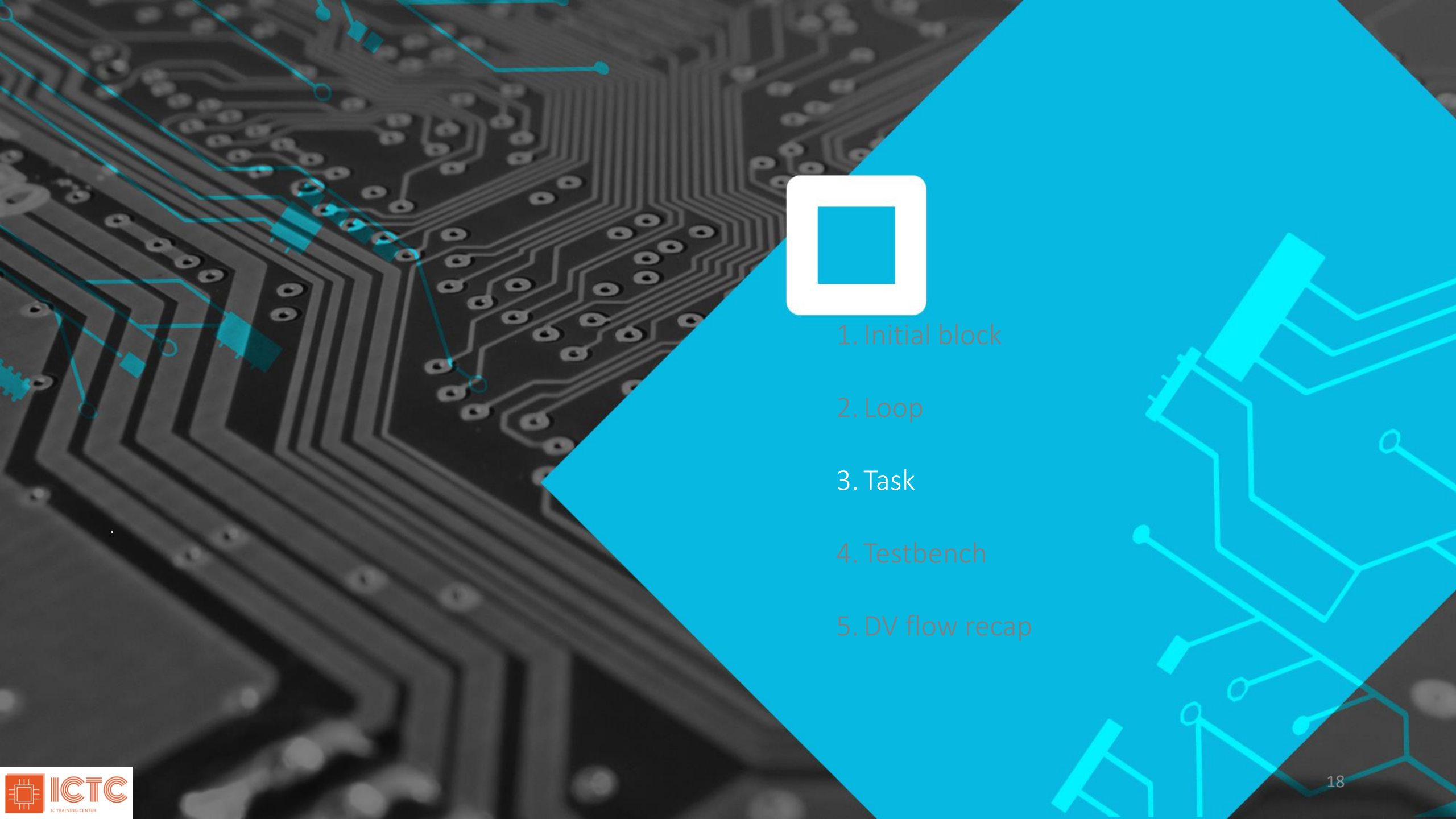
Example: The above practice can be re-written as below. This is the common way for testbench writing.

```
module tb;
  reg clk, rst_n, enable;
  reg [7:0] data;

  initial begin
    rst_n = 0;
    enable = 0;
    data = 8'h0;

    repeat (2) @posedge clk;
    #1 rst_n = 1'b1;
    @posedge clk;
    #1 enable = 1;
    @posedge clk;
     #1 data = 8'haa;
    @posedge clk;
     #1 data = 8'hbb;
    @posedge clk;
     #1 data = 8'hcc;
     @posedge clk;
     #1 enable = 0;
     data = 8'h00;

      #100 $finish;
  end
```




```
  initial begin
    clk = 0;
    #50;
    //forever keyword is used to describe
       forever loop until simulation is finished
    forever clk = #50 ~clk;
  end
endmodule
```

In previous practice, if changing the clock frequency, need to revise the whole testbench. Now the whole simulation is depended to the timing of clock only. Changing clock frequency does not affect the relationship between signals.



1. Initial block
2. Loop
3. Task
4. Testbench
5. DV flow recap

TASK

- Function is meant to do some processing on the input and return a single value, while a task can have multiple output.
- Can define tasks in two ways as below

```
task task_name;
   input_declaration;
   output_declaration;
   inout_declaration;

   begin
      [statements]
   end
endtask
```

These are equivalent

```
task task_name (
   input <range> arg1,
   output <range> arg2,
   ....
);
   begin
      [statements]
   end
endtask
```

TASK

- Different with function, task can have timing control statements (“#” or “@” or “wait”) inside.

```
task delay_100ns;
   input a;
   output y;

   begin
      #100 y = a;
   end
endtask
```

Output y is delayed 100ns than input a

TASK INVOKE

```
module tb;
  reg in;
  reg out;

  initial begin
    in = 0;
    out = 0;
    #100 in = 1;
    delay_100ns( in, out); //task invoked
  end

  task delay_100ns;
    input a;
    output y;

    begin
      #100 y = a;
    end
  endtask
endmodule
```

Task need to be called inside procedure.

Task can be called as many times as needed.

Output of tasks are given value when the task completed. Then if you read “out” between 100ns and 200ns, it will return 0.

Task need to be declared inside module

TASK VS FUNCTION

Function	Task
Can not have time, executes in the same simulation time unit	Consume simulation time
Can not enable a task	Can enable others task and functions
Should have at least one input argument and can not have output or inout arguments	Can have zero or more arguments of any type
Can return only a single value	Can not return a value but can achieve the same effect using output arguments

TASK PRACTICE

Practice: Create a task to perform swapping between two 8-bit reg variables .
Hint: need to use "inout" type so that value can be swapped directly.

For example:

- a = 5
- b = 10
- swap(a , b) → call task
- $display("a = %d, b = %d", a,b) → a = 10, b = 5 → value of a & b is swapped.

VERILOG SYSTEM TASK

The system tasks are used to perform some operations like displaying the messages, terminating simulation, generating random numbers, etc

System task	Description
$display	To display strings, variables, and expressions immediately in the active region.
$monitor	To monitor signal values upon its changes and executes in the postpone region.
$strobe	To display strings, variables, and expressions at the end of the current time slot i.e. in the postpone region.
$finish	End simulation
$random	Return a random 32-bit integer

VERILOG SYSTEM TASK FORMAT

The display system tasks use various format specifiers to print the values

Format specifiers	Description
%c	To display ASCII character
%s	To display string
%t	To display the current time
%f	To display real numbers in decimal format. (Ex. 3.14)
%e	To display real numbers in scientific format. (Ex. 2e20)
%x	To display hexa number
%o	To display octal number
%d	To display decimal number
%b	To display binary number

COMPILER DIRECTIVE

We already learn `define, `include, `ifdef in previous session

Compiler directives	Description
`define	To define text macros (Similar #define in C language
`include	To include entire content from another Verilog file into existing file during compilation
`ifdef ...`endif `ifdef ...`else ...`endif	Conditional compiler directives that behave as if…else conditional statement
`timescale	To specify time units and precision for the module

TIMESCALE

- Verilog simulation depends on how time is defined.
- The `timescale compiler directive specifies the time unit and precision for the simulation.
- **Syntax**: `timescale <time_unit>/<time_precision>

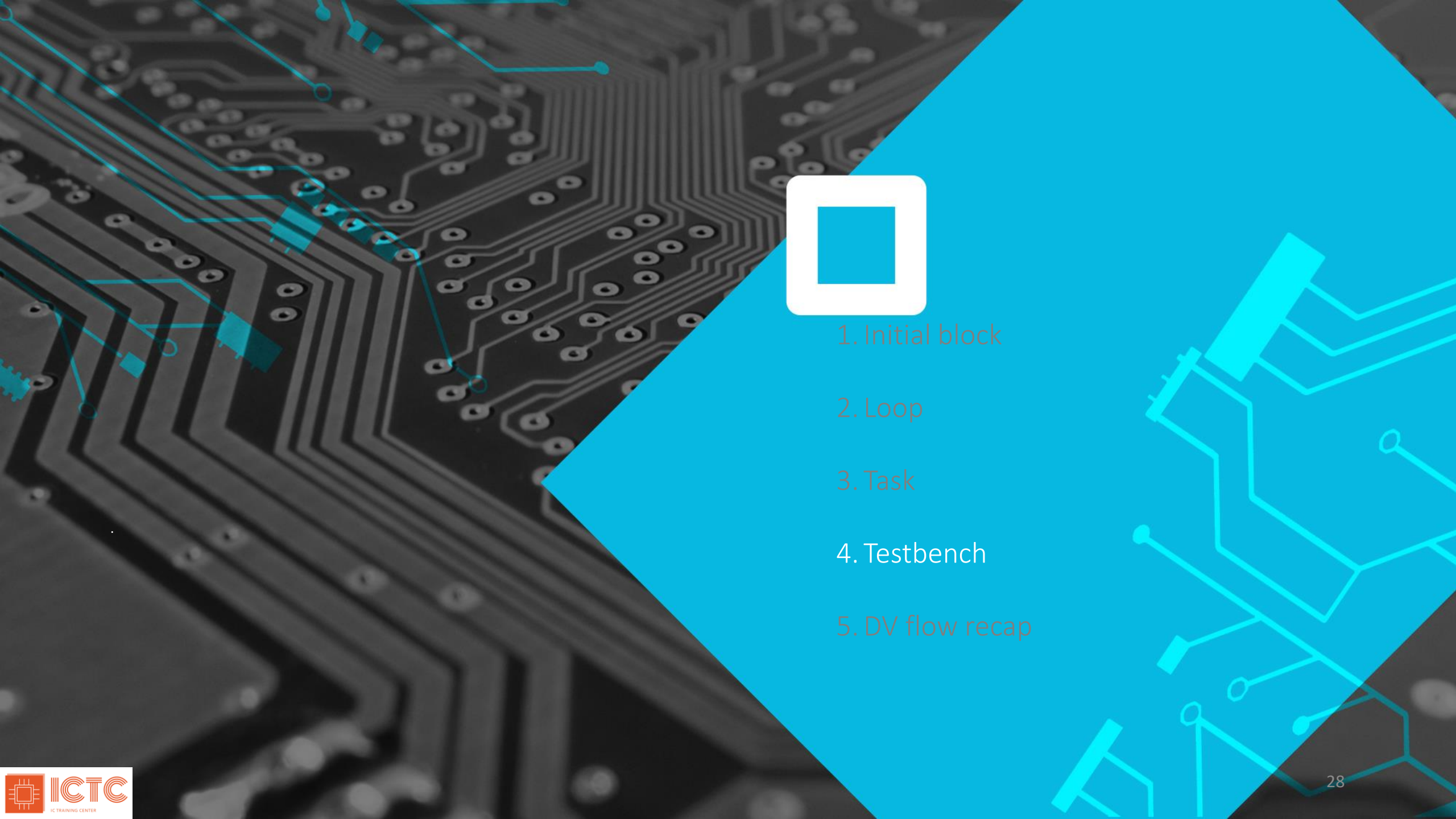
  `timescale 1ns/1ps

  `timescale 10us/100ns

- The time_unit is the measurement of delays and simulation time while the time_precision specifies how delay values are rounded before being used in the simulation.

```
`timescale 1ns/1ns
…
initial begin
   #1     $display(“T=%t at time #1”, $realtime);  //result is 1
   #0.49 $display(“T=%t at time #1”, $realtime);  //result is 1
   #0.5  $display(“T=%t at time #1”, $realtime);  //result is 2
   #0.51 $display(“T=%t at time #1”, $realtime);  //result is 3
end
…
```

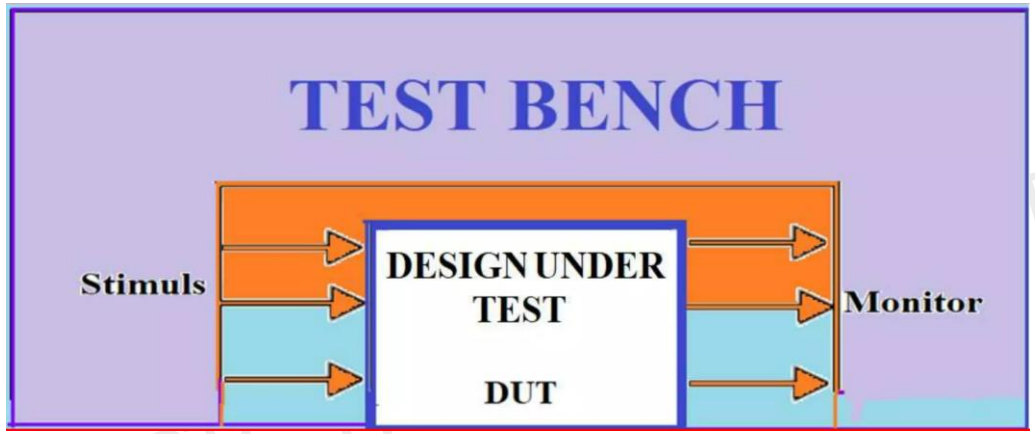
```
`timescale 1ns/1ps
…
initial begin
   #1     $display(“T=%t at time #1”, $realtime);  //result is 1000
   #0.49 $display(“T=%t at time #1”, $realtime);  //result is 1490
   #0.5  $display(“T=%t at time #1”, $realtime);  //result is 1990
   #0.51 $display(“T=%t at time #1”, $realtime);  //result is 2500
end
…
```



1. Initial block
2. Loop
3. Task
4. Testbench
5. DV flow recap

TESTBENCH

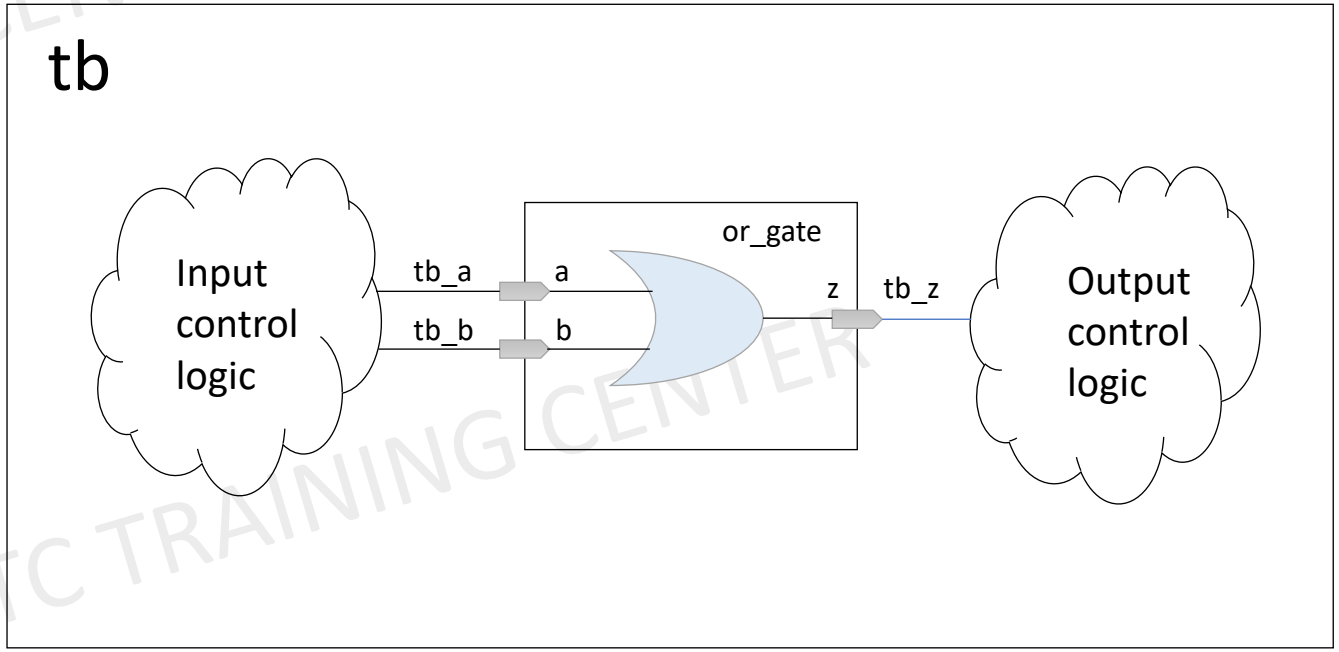
- A test bench is a program which generate inputs to DUT and observe the output.
- A test bench does not have to be synthesized. Hence, we can use all features of Verilog programming on the test bench development.



TESTBENCH

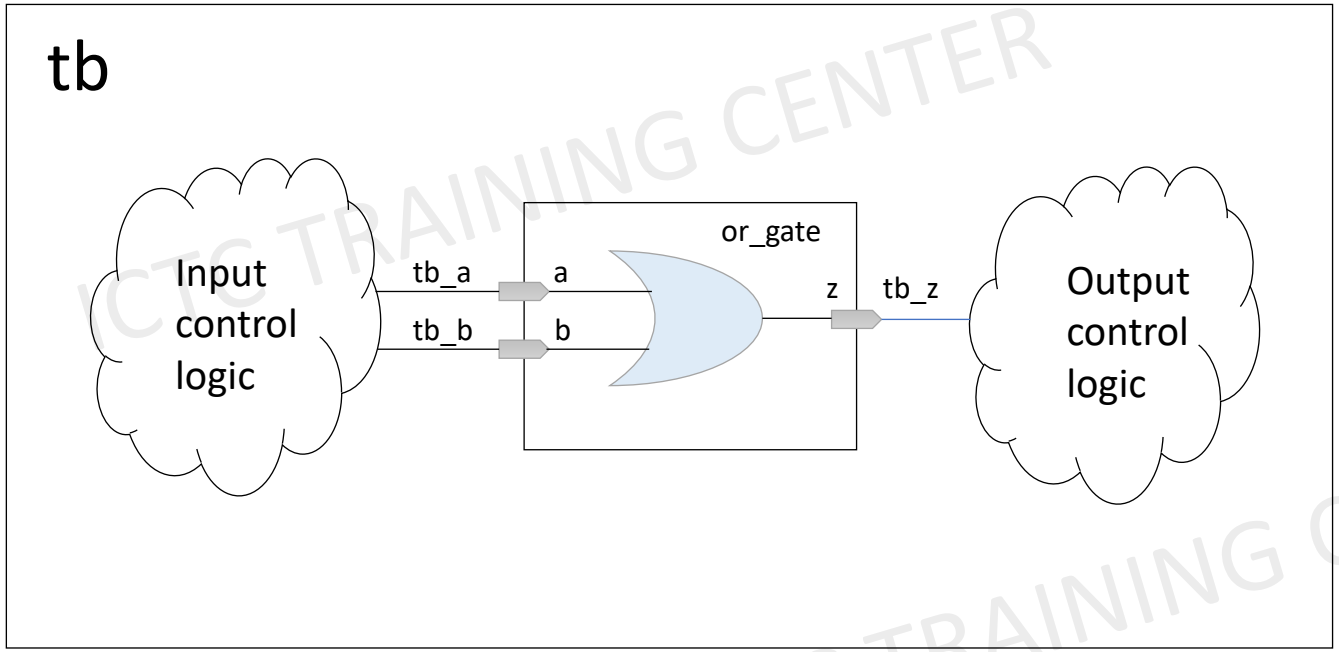
- Testbench diagram for an OR gate module

```
module or_gate (a,b,z);
   input   wire a;
   input   wire b;
   output wire z;

   assign z = a | b;
endmodule
```



TESTBENCH

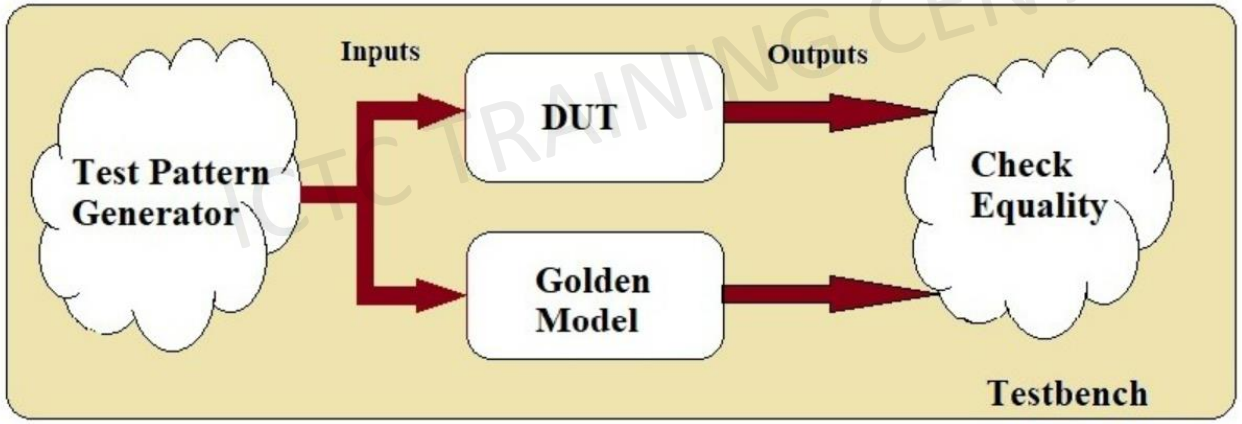
- Example of test bench for an OR gate module




```
module tb ;
  wire tb_a, tb_b, tb_z;

  or_gate u_dut (.a( tb_a ), .b( tb_b ), .z ( tb_z ));

  initial begin
    tb_a = 0;
    tb_b = 0;
    #10;
    $display("Case 1: a=%b b=%b",tb_a, tb_b);
    if( tb_z === 0 ) begin
       $display("PASSED);
    end else begin
      $display("FAILED. Exp: 0 Actual: %b", tb_z);
      $finish;
    end

    #10;
    tb_a = 0;
    tb_b = 1;
    $display("Case 2: a=%b b=%b",tb_a, tb_b);
    ....
  end


endmodule
```

TESTBENCH USING GOLDEN MODEL

Golden model is a no-timing program, which generates the expected result




```
1  module test_bench;
2
3      reg [1:0]  a,b;
4      wire [2:0] sum;
5      reg [2:0] res;
6
7      full_adder_2b dut(.a(a), .b(b), .sum(sum));
8
9      initial
10     begin
11         for( integer i = 0; i < 50; i++ ) begin
12             a = $urandom_range(0,3);
13             b = $urandom_range(0,3);
14         #5;
15         $display("================================");
16         $display("==== Case:%2d a = %d, b = %d =======",i,a,b);
17         $display("================================");
18         res = a+b;
19         $display("Exp: %d Actual: %d", res, sum);
20         if( res === sum) begin
21             $display(">>>>>>>>>>>>  PASS <<<<<<<<<<<<\n");
22         end else begin
23             $display(">>>>>>>>>>>>  FAIL <<<<<<<<<<<<\n");
24             $finish;
25         end
26       end
27
28     #100; $finish;
29     end
30
31
32 endmodule
```

DUT

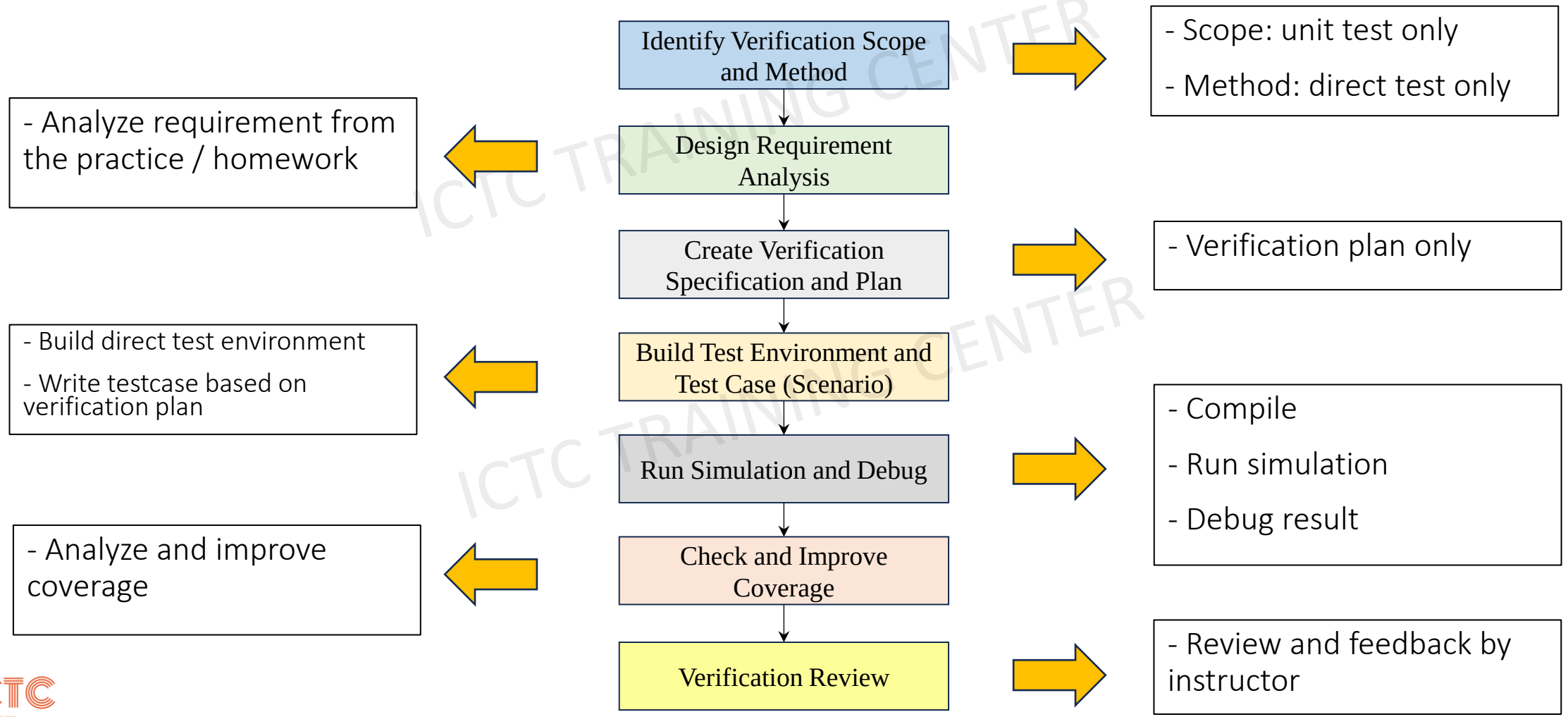
Pattern generator

Golden model

Checker

REVIEW DV DESIGN FLOW

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

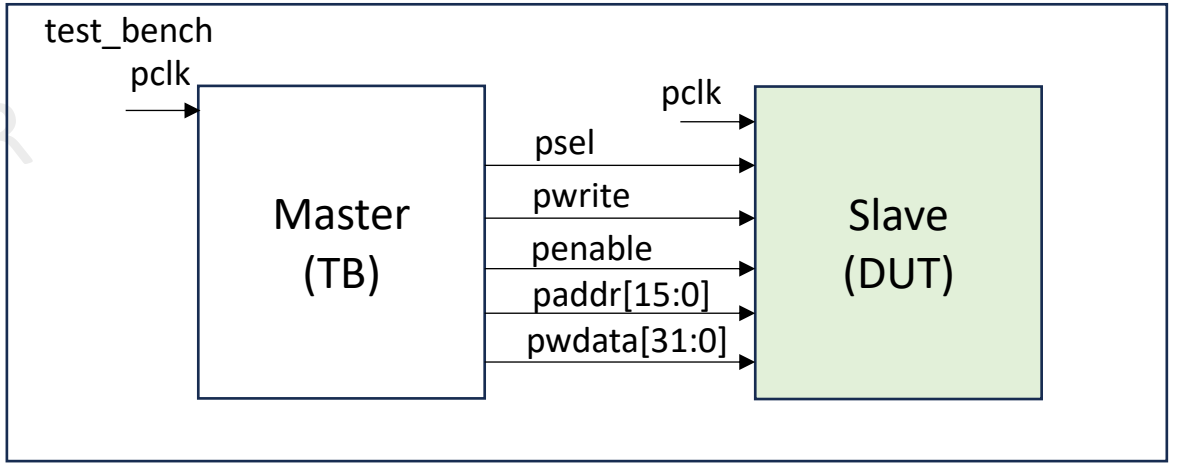


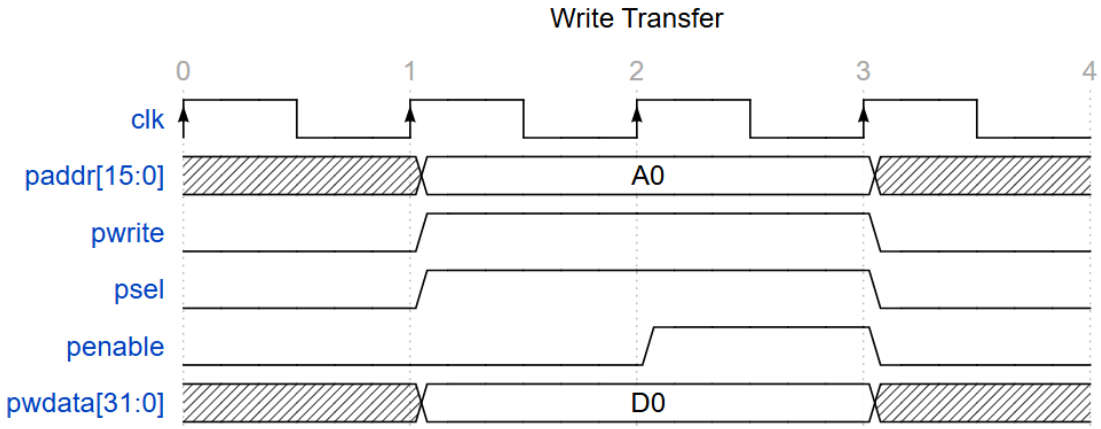
Session 11

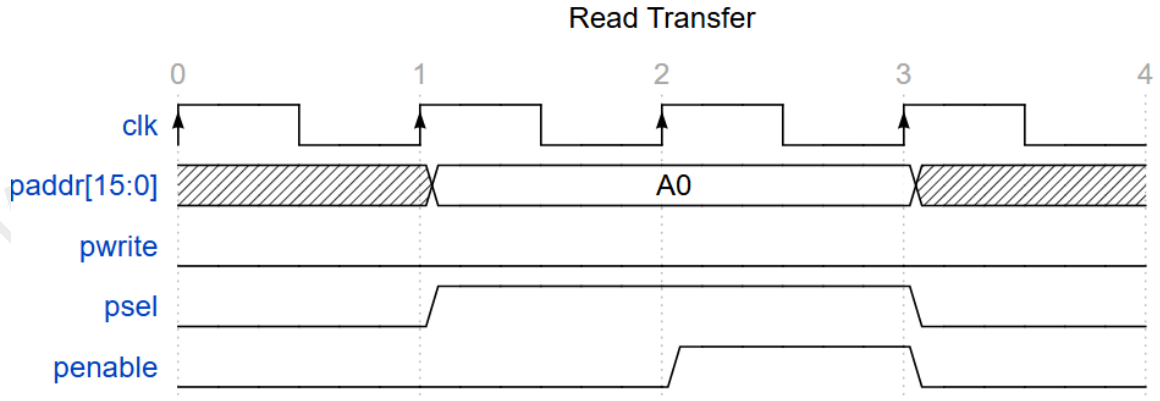
Homework1: A BUS protocol include master and slave. Master send read/write request, slave receive the request.

Write master read/write task to realize below waveform.

- All signals need to be generated based on clock using "@" operator.
- paddr[15:0], pwdata[31:0] can be assigned to any value.
- When calling master write task, the timing of write transfer is realized. The same for calling master read task.
- Task can be called as many times as needed.
- The "slave" on the block diagram is just to illustrate the design idea. It can be ignored in the test_bench



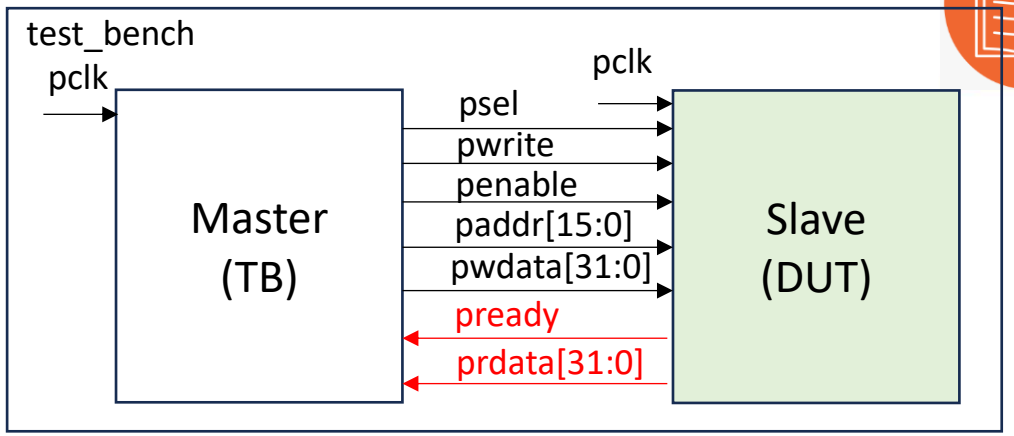


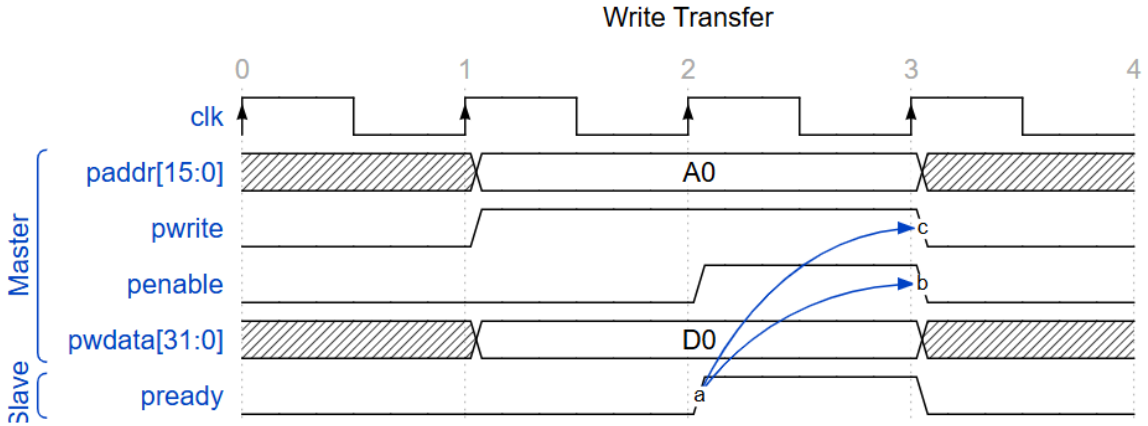


Session 11

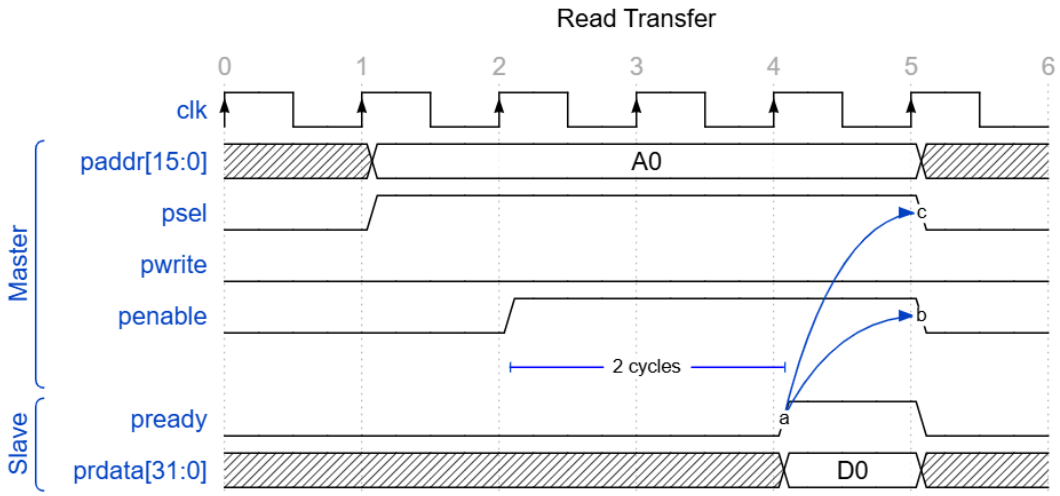
Homework2(*): add slave logic

- Slave has "pready" signal indicates the transfer is completed.
- Pready is 1 immediately or some cycles after both psel and penable is High. The number of cycles can be randomized between 0 and 5.
- Write: only when pready is 1, master's signals can be negated as the waveform
- Read: only when pready is 1, master's signals can be negated as the waveform
- Slave generate prdata[31:0] (any value) and the master read task print out the value when it detects pready is High.
- There is no DUT in this testbench, the "slave logic" needs to be driven b





Example: write transfer with pready returns immediately



Example: read transfer with pready returns in 2 cyle

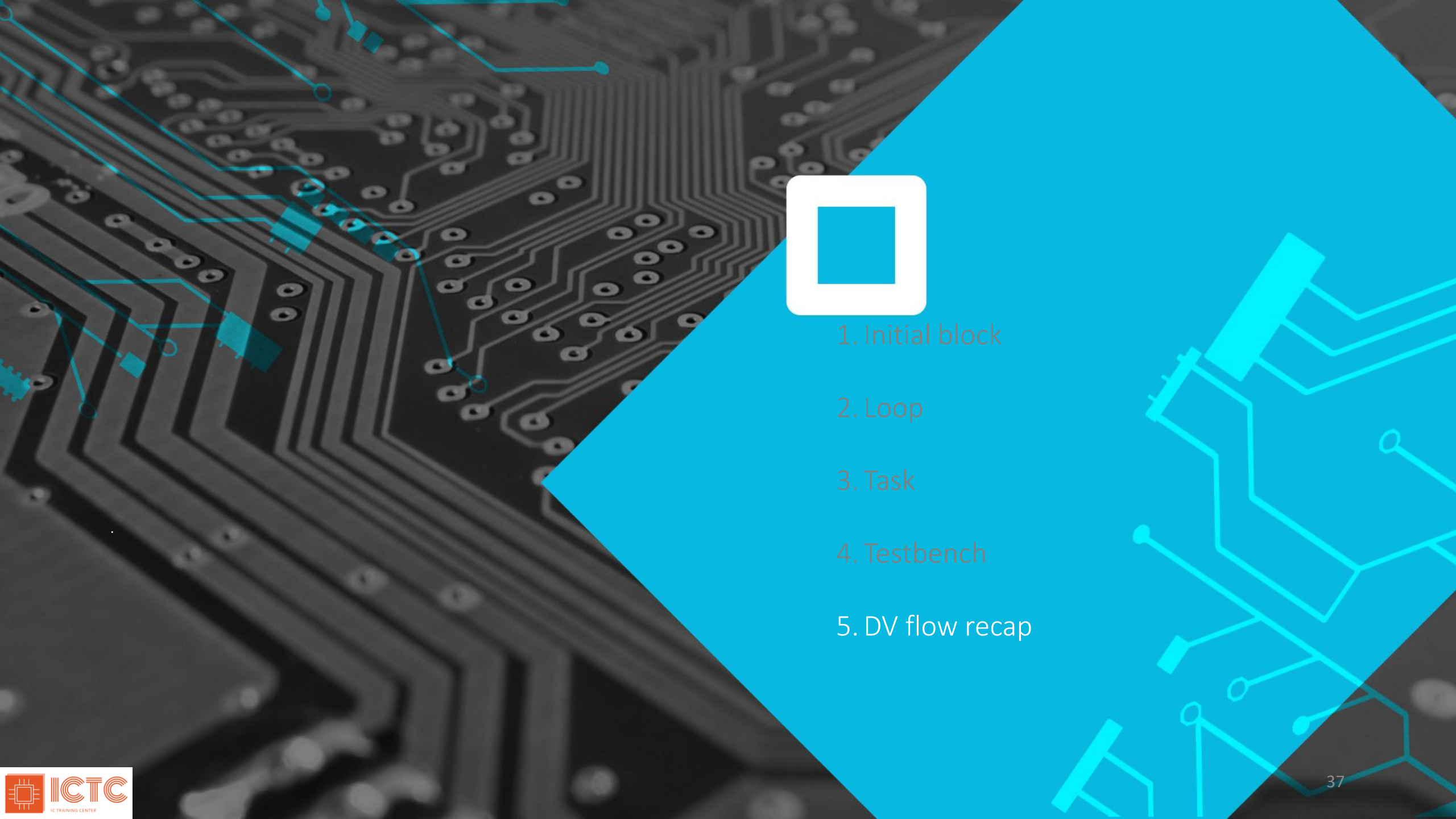
Session 11

```
{signal: [
  {name: 'clk', wave: 'P...'},
  {name: 'paddr[15:0]', wave:
'x=.x',data:["A0","A0"]},
  {name: 'pwrite', wave: '01.0',},
  {name: 'psel', wave: '01.0',},
  {name: 'penable', wave: '0.10',},
  {name: 'pwdata[31:0]', wave:
'x=.x',data:["D0"]},
],
  head:{
   text:'Write Transfer',
   tick:0,
   every:1
  },
  config: { hscale: 3 }
}
```

```
{signal: [
  {name: 'clk', wave: 'P...'},
  {name: 'paddr[15:0]', wave:
'x=.x',data:["A0","A0"]},
  {name: 'pwrite', wave: '0...',},
  {name: 'psel', wave: '01.0',},
  {name: 'penable', wave: '0.10',},
],
  head:{
   text:'Read Transfer',
   tick:0,
   every:1
  },
  config: { hscale: 3 }
}
```

```
{signal: [
  {name: 'clk', wave: 'P...'},
  ["Master",
  {name: 'paddr[15:0]', wave: 'x=.x',data:["A0","A0"]},
  {name: 'pwrite', wave: '01.0',node: '...c'},
  {name: 'penable', wave: '0.10',node: '...b'},

  {name: 'pwdata[31:0]', wave: 'x=.x',data:["D0"]},
  ],
  ["Slave",
  {name: 'pready', wave: '0.10',node: '..a'},
  ],
],
  head:{
   text:'Write Transfer',
   tick:0,
   every:1
  },
  edge: [
    'a~->b','a~->c','I+J 1 cycle'
  ],
  config: { hscale: 3 }
}
```

```
{signal: [
  {name: 'clk', wave: 'P.....'},
  ["Master",
  {name: 'paddr[15:0]', wave: 'x=...x',data:["A0","A0"
  {name: 'psel', wave: '01...0',node: '.....c'},
  {name: 'pwrite', wave: '0.....',},
  {name: 'penable', wave: '0.1..0',node: '.....b'},
  {                    node: '..I.J', },
  ],
  ["Slave",
  {name: 'pready', wave: '0...10',node: '....a'},
  {name: 'prdata[31:0]', wave: 'x...=x',data:["D0"]},
  ],
],
  head:{
   text:'Read Transfer',
   tick:0,
   every:1
  },
  edge: [
    'a~->b','a~->c','I+J 2 cycles'
  ],
  config: { hscale: 2 }
}
```

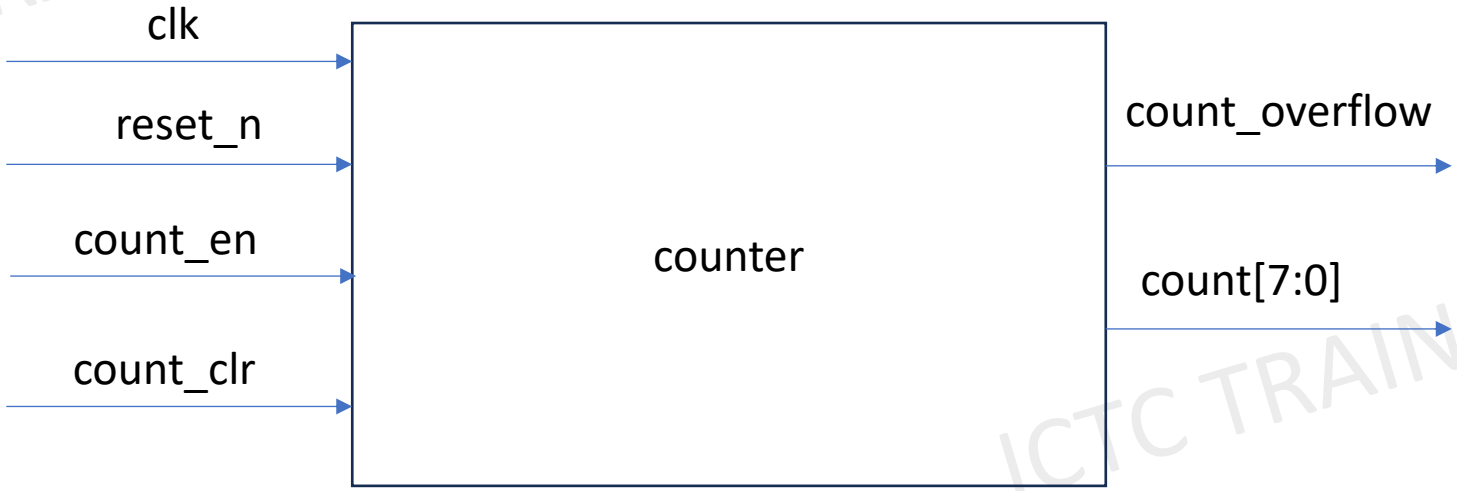


1. Initial block
2. Loop
3. Task
4. Testbench
5. DV flow recap

VERIFICATION PLAN EXAMPLE

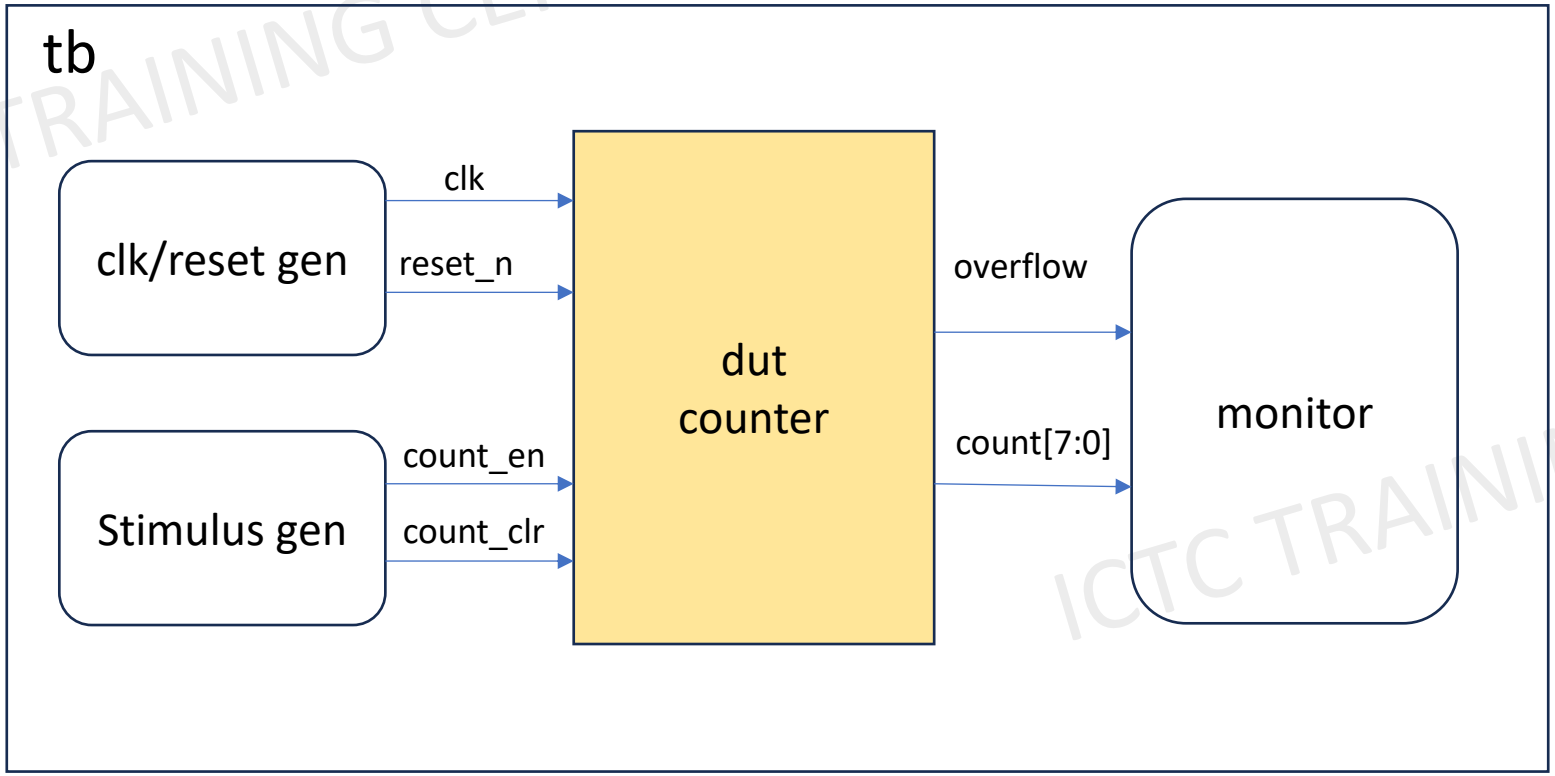
Let’s see again our previous counter example.

The later projects in this course need to follow this style.



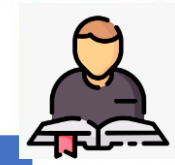
VERIFICATION PLAN EXAMPLE

- Test bench hierrachy



VERIFICATION PLAN EXAMPLE

- Verification item list example. Later on it should be written in excel



ID	Item name	Description
1	cnt_init	After reset is released, the counter initial value is 8'h00 and remain the same while counter_en is 0.
2	cnt_up	When "counter_en" is High, counter start counting up.
3	cnt_stop	When "counter_en" is negated from High to Low, the counter stops and remain the same value. When "counter_en" is resumed from Low to High, the counter can resume counting from the stop value
4	cnt_clr	When "counter_clr" is High, counter is cleared to 8'h00, regardless of "counter_en" value (counter_clear has higher priority).
5	cnt_reset	When rst_n is asserted during operating, the counter is initialized to 8'h00.
6	cnt_overflow	The overflow flag is asserted when counter reached 8'hff and negated to Low in 1 cycle. The counter is initialized to 8'h00 and count up normally after overflow (if counter_en is still High).

VERIFICATION PLAN EXAMPLE

The recommended verification item list.

ID	Item	Sub item 1	Sub item 2	Method	Class	Test sequence	Formal assertion	Pass condition	Plan Start	Plan End	Actual Start	Actual End	PIC
1	Address map	Address is 4KB	-	Direct	A	Toggle each bit of 12 bit address - R/W at address offset 0 - R/W at address offset 1 - R/W at address offset 2 .... - R/W at address offset 0x800	-	RW is OK Psel always toggle	2-Feb	3-Feb			
2		Boundary address	Address inside address map 0x0000_1000 - 0x0000_1FFF	Direct	A	Write/Read to 0x0000_1000 Then Write/Read to 0x0000_1FFF	-	RW is OK Psel is asserted	4-Feb	5-Feb			
3			Address outside address map 0x0000_1000 - 0x0000_1FFF	Direct	A	Write/Read to 0x0000_0FFF Write/Read to 0x0000_2000		RW is not hang-up Psel is not asserted	4-Feb	5-Feb			
		Reserved address	-	Formal	B	-	Use jasper gold CSR to check all the reserved addresses	Jasper gold reserved assertion is PASSED	2-Feb	10-Feb			

The more detail of the verification item list, the more chance to find bugs of the design