
Verilog
SystemVerilog
training
RTL approach

Agenda

Verilog Systemverilog training course for RTL

Students are expected to have basic knowledge on Boolean algebra, some logic gates and K-map.

Step 1

Basic

RTL designer introduction.
Basic logic component
Basic syntaxes

Step 2

Advance

More languages' syntaxes.
Design training and coached small project.

Pros & Cons of

Digital system

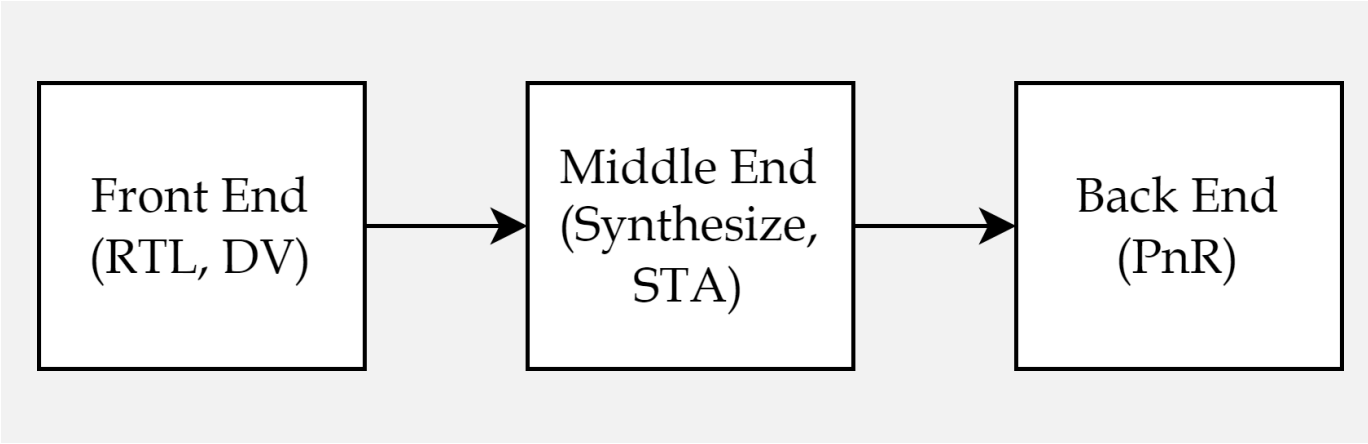
"Systems which process discrete values are called digital systems. In digital representation the quantities are represented not by proportional quantities but by symbols called digits."
--- Deepak Kumar Tala, asic-world

- Easy to design
- Digital system only cares about value (HIGH or LOW), not the exact voltage.
- Easy to store information.
- Programmable.
- Less affected by noise. Size is smaller (more logic can be fabricated in to 1 IC

- Most signals in real world are analog. Thus, there is a need to convert the signals to digital.

Simple flow

Digital design



1. Front End (FE)

Requirements from user and designer are written in documents.
The design is implemented (e.g., RTL design)
The functionality of the designed is verified (Design Verification)

2. Middle End (ME)

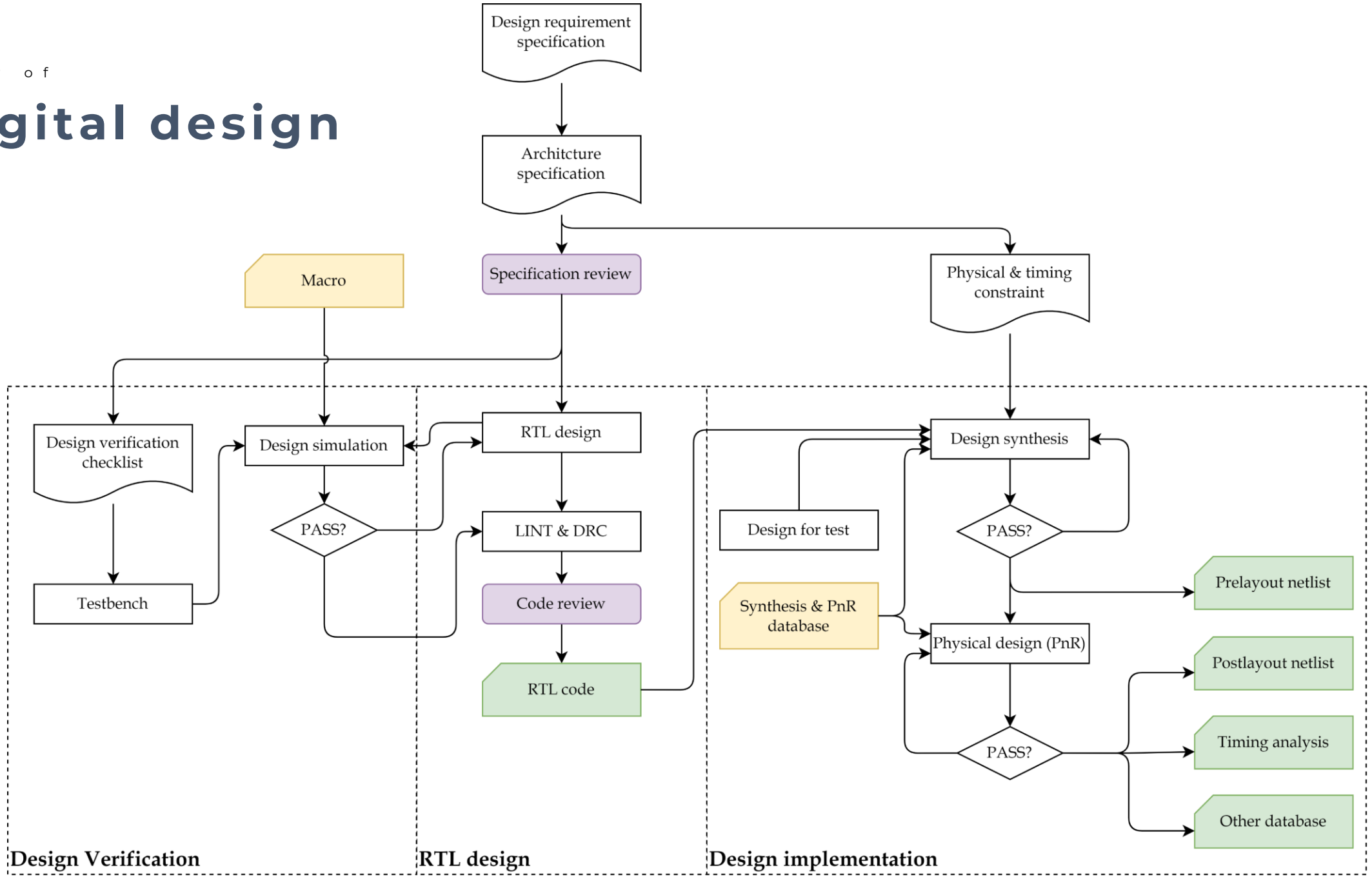
Convert (Synthesize) the design to logic gate cells (netlist).
The speed (performance) of the netlist is verified (STA)

3. Back End (BE)

Place and Route (PnR) the logic gate cells.
The netlist is put in physical world (layout) with real cells and metal connection for manufacturing.

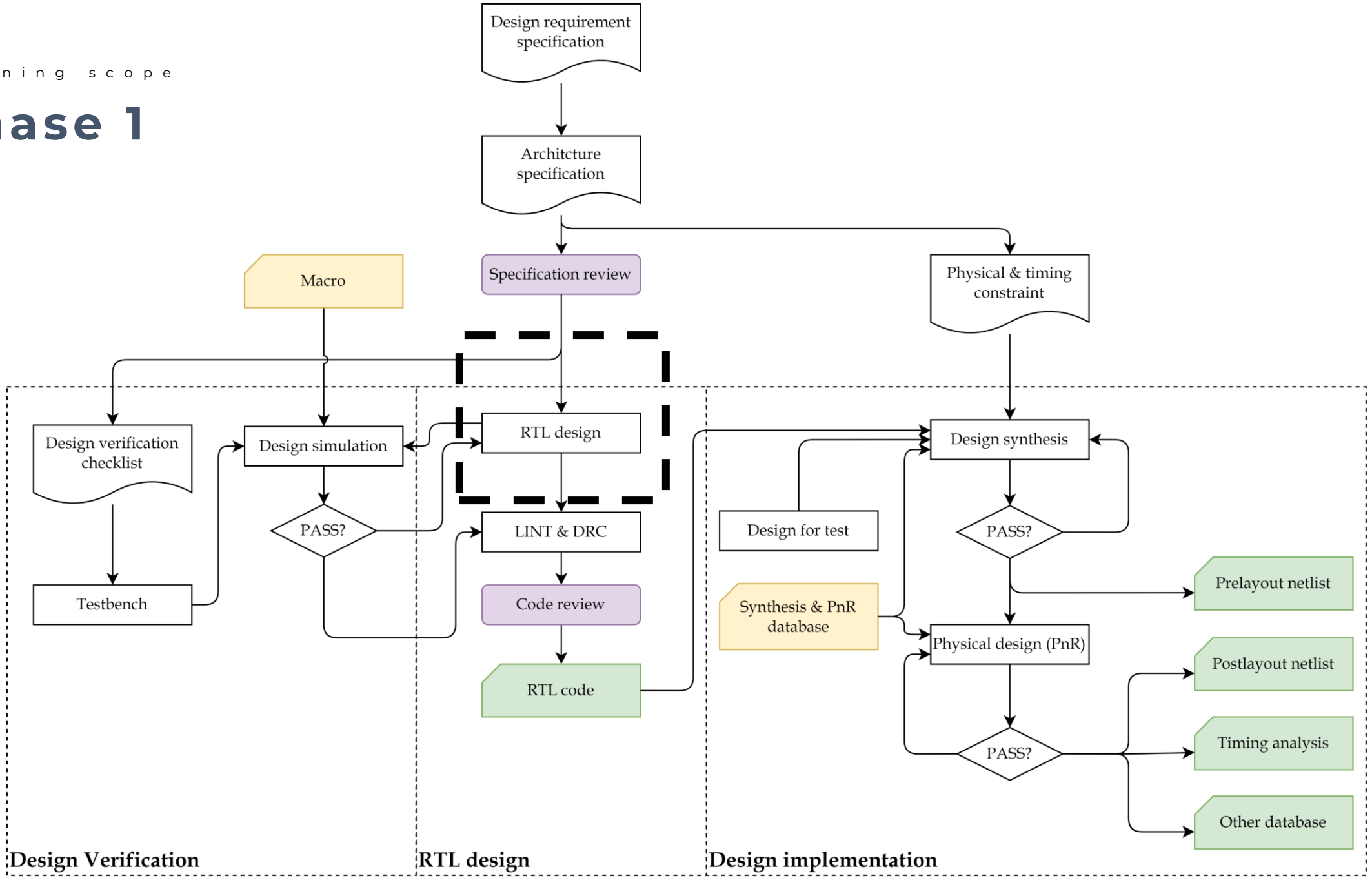
Flow of

Digital design



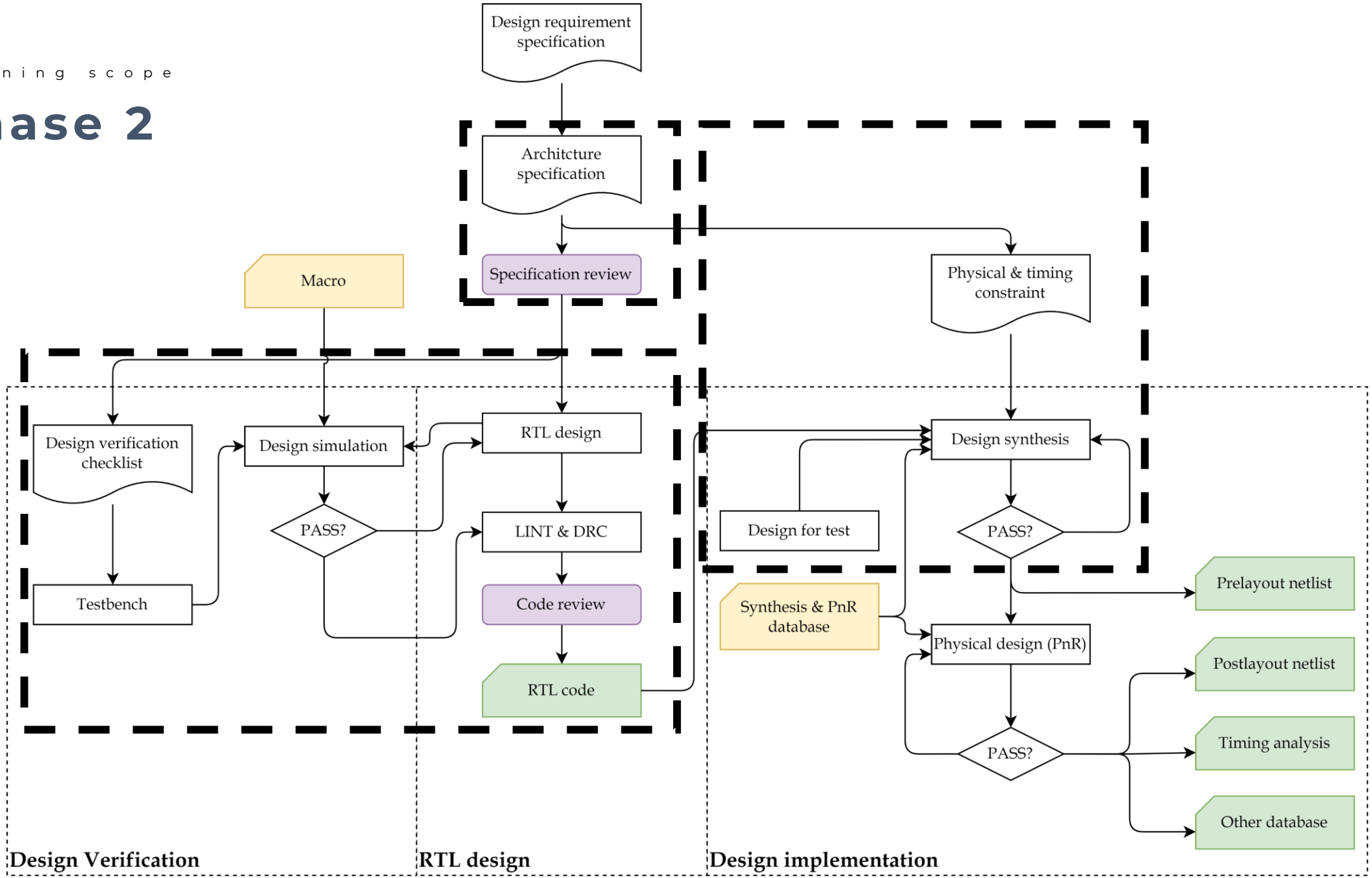
Training scope

Phase 1



Training scope

Phase 2



Expected

Training outcome

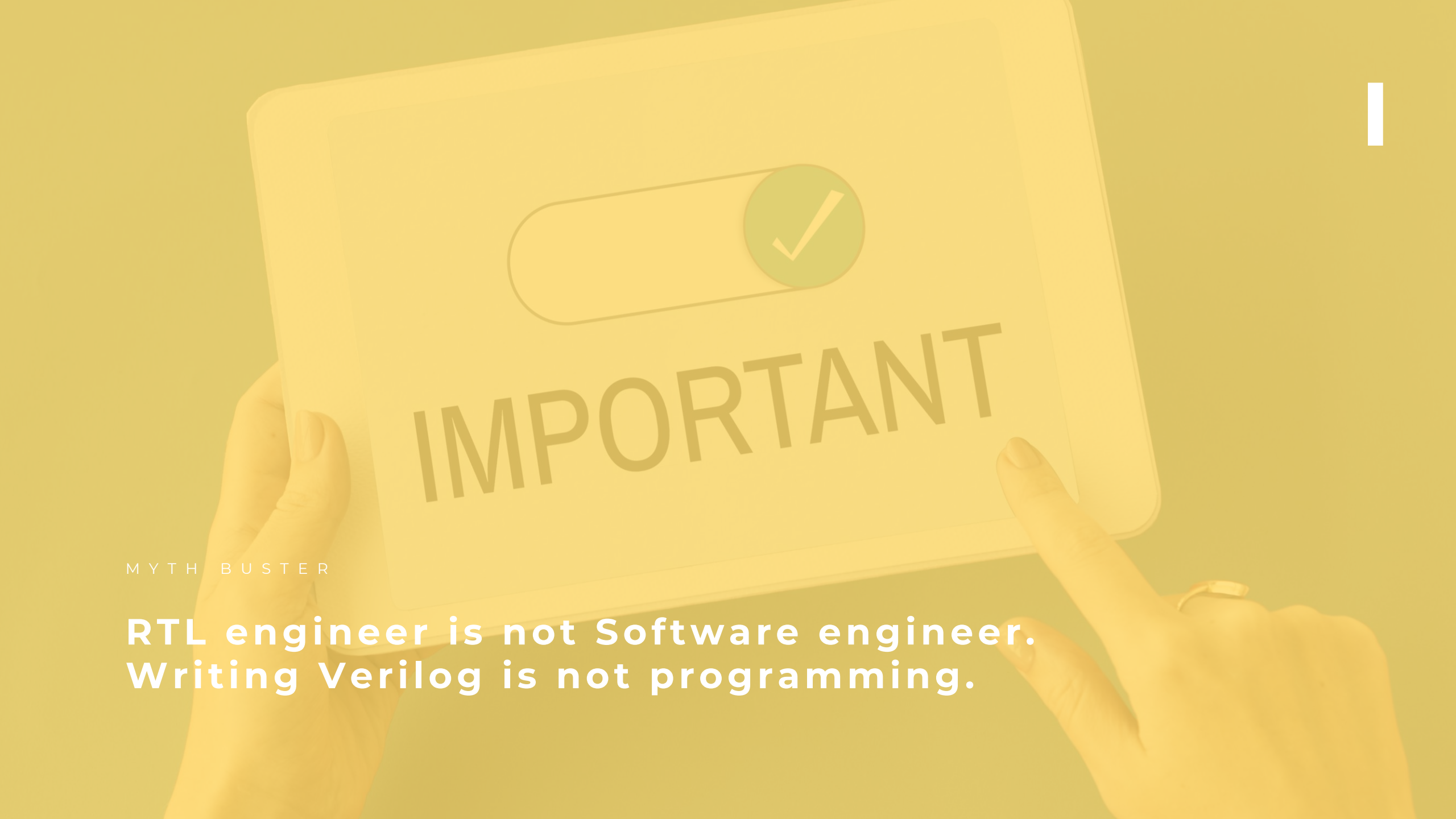
As Verilog language was first created to describe hardware (HDL), thus this program focuses more on RTL part.

What it provides:

- Understand RTL design concept: From specification, to design and writing code.
- Basic syntaxes of Verilog/SystemVerilog to work in daily life tasks.

Expected outcome:

- Understand simple snippets of RTL code. Has ability to write RTL for simple blocks.
- Has ability to debug syntax problems of RTL codes.
- Has ability to create simple test bench & run text-based simulation.



MYTH BUSTER

RTL engineer is not Software engineer. Writing Verilog is not programming.

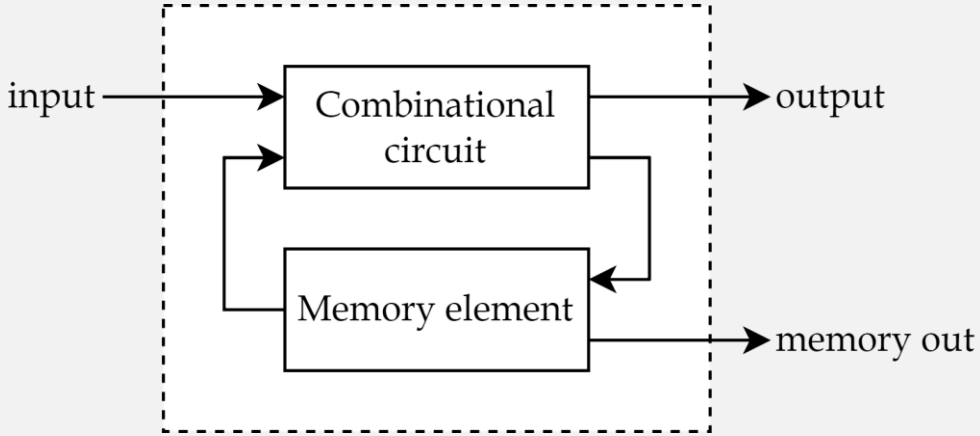
COMMON LOGIC CELLS

Please review functionality of combinational cells:
INV, AND, OR, XOR, MUX.
Following slides are about sequential logic cells.

Digital system

Sequential circuit

Most systems in practice include memory elements, which is one of sequential logic type.



There are 2 types of sequential logic cells:

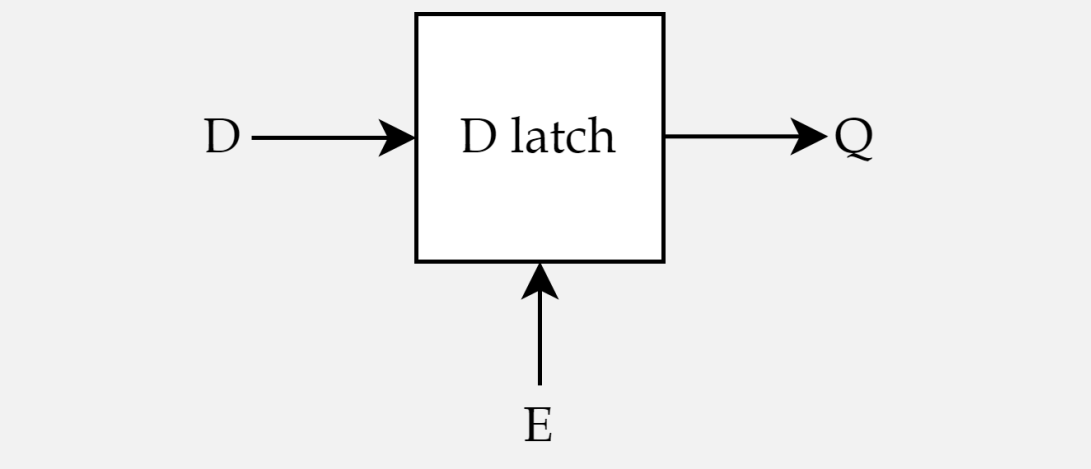
- Latch: The output senses the level of the inputs to capture data.
- Flipflop: The output sense the edge of the inputs to capture data.

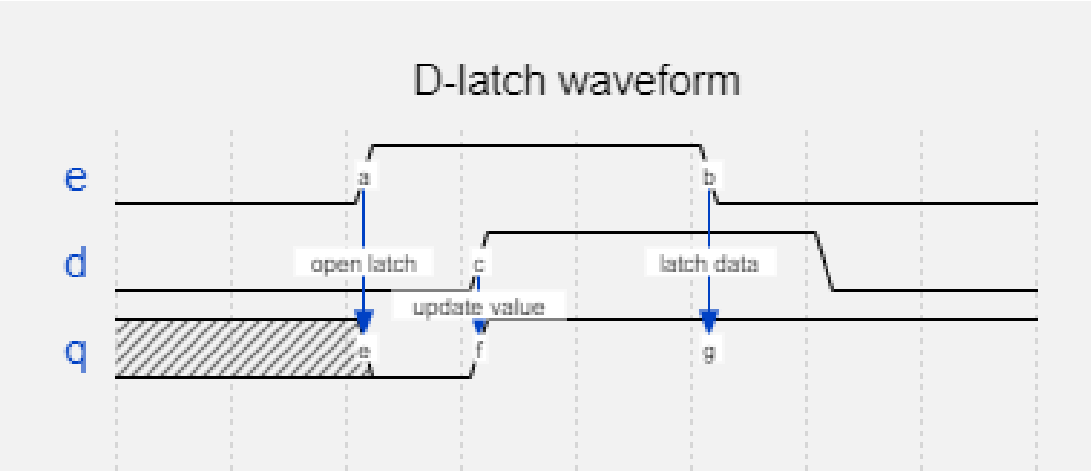
Commonly use are D-latch and D-flipflop

Data is captured by input's level

Pin E enables data to goes through the latch.

E	D	Q
0	*	Q
1	0	0
1	1	1





Sequential logic cell

D-latch

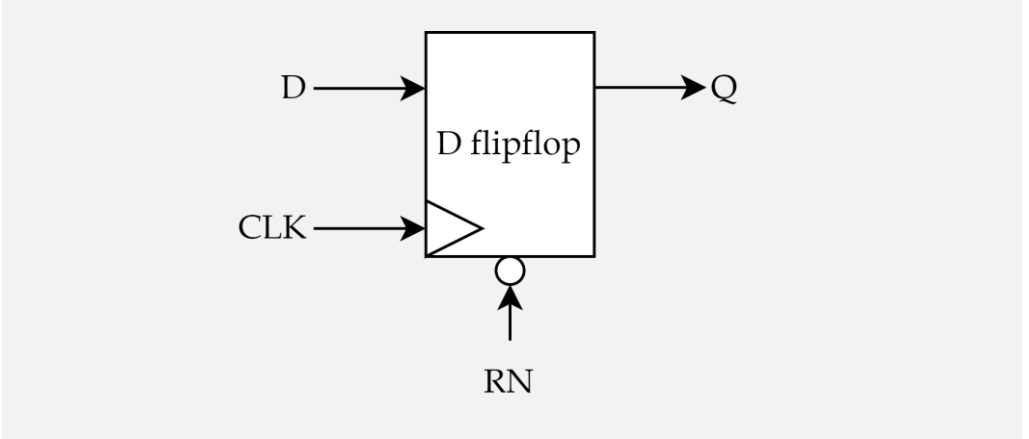
Sequential logic cell

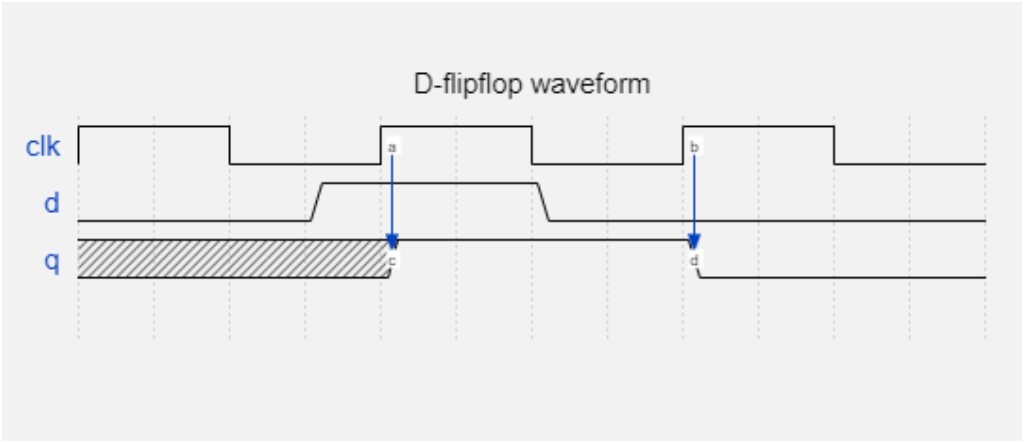
D-flipflop

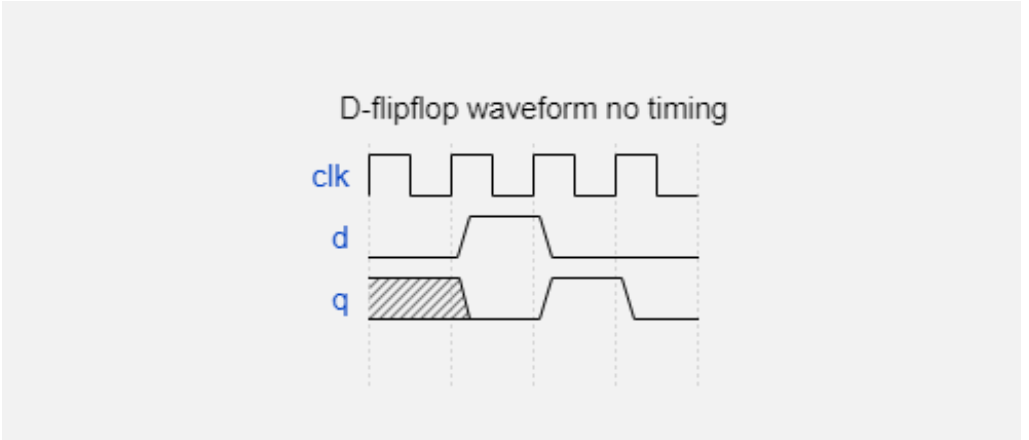
Data is captured by input's edge

Rising edge of the clock captures input data.

CLK	D	RN	Q
*	*	0	0
0	*	1	Q
1	*	1	Q
R	0	1	0
R	1	1	1
F	*	1	Q

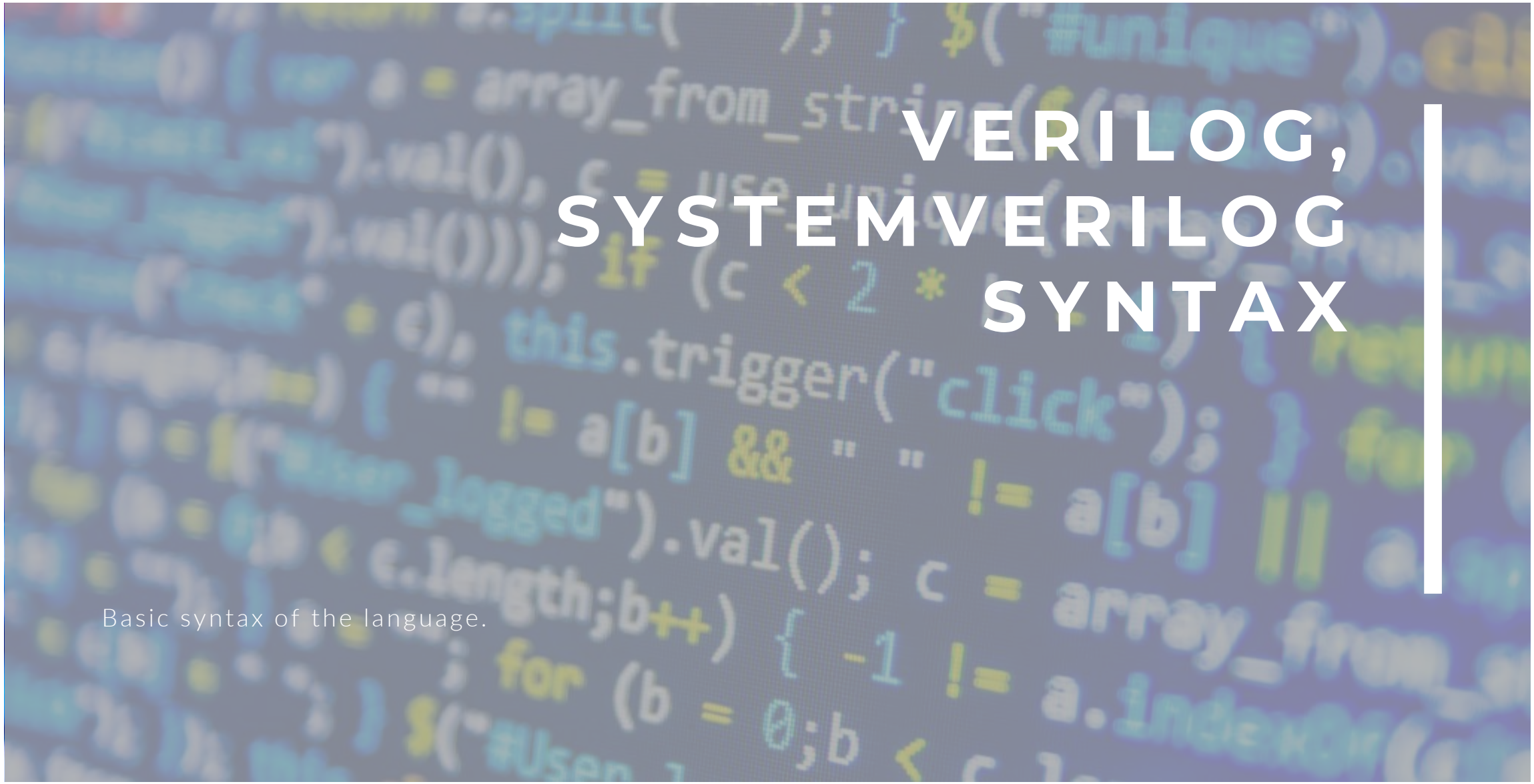






ACTIVITY

Check 3.1, 3.2

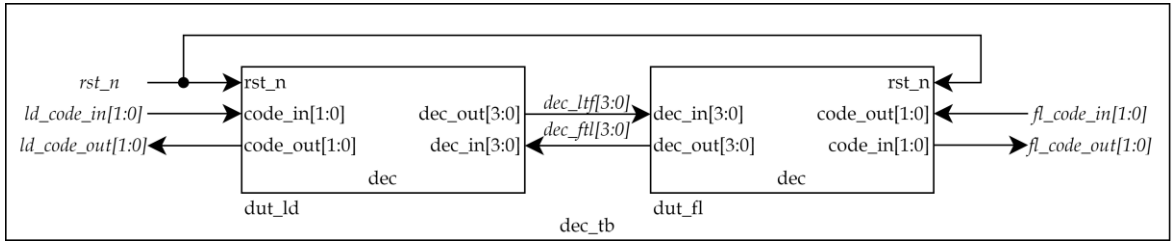


VERILOG, SYSTEMVERILOG SYNTAX

Basic syntax of the language.

Syntax

Module and Instance



Module: dec, dec_tb
Instance: dut_ld, dut_fl

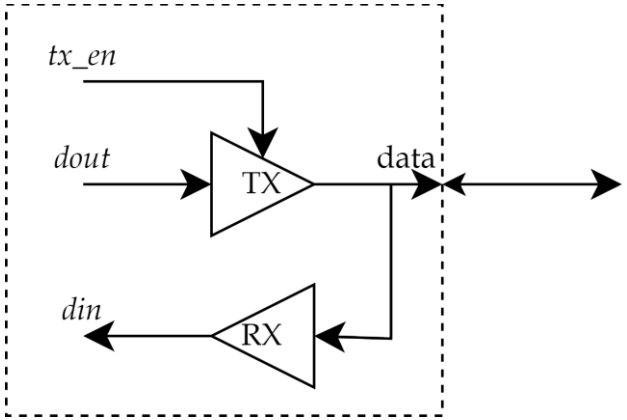
```v
// file: dec.v
module dec (
  // Ports of the DUT
    input  wire       rst_n
  , input  wire [3:0] code_in
  , output wire [3:0] code_out
  , input  wire [1:0] dec_in
  , output wire [1:0] dec_out
);

endmodule // dec
```

```v
// file: dec_tb.v
module dec_tb;

  // Signals of the testbench
  wire [1:0] dec_ltf;
  wire [1:0] dec_ftl;
  reg  [3:0] ld_code_in;
  wire [3:0] ld_code_out;
  reg  [3:0] fl_code_in;
  wire [3:0] fl_code_out;
  reg  rst_n;

  // Instances of the DUT
  dec dut_ld (
      .rst_n    ( rst_n       )
    , .code_in  ( ld_code_in  )
    , .code_out( ld_code_out )
    , .dec_in   ( dec_ftl     )
    , .dec_out  ( dec_ltf     )
  )
  dec dut_fl (
      .rst_n    ( rst_n       )
    , .code_in  ( fl_code_in  )
    , .code_out( fl_code_out )
    , .dec_in   ( dec_ltf     )
    , .dec_out  ( dec_ftl     )
  )
endmodule // dec_tb
```

A module is a block of Verilog code that implements a certain functionality.
An instance is an object of the module. Each instance is a complete, independent, and concurrently active copy of a module.

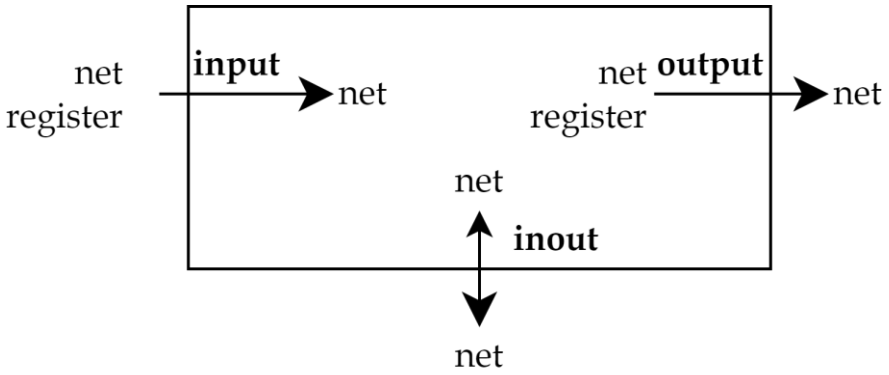
Syntax

Port declaration




```
// file: dec.v
module dec (
  // Ports of the DUT
    input  wire       rst_n
  , input  wire [3:0] code_in
  , output wire [3:0] code_out
  , input  wire [1:0] dec_in
  , output wire [1:0] dec_out
);

endmodule // dec
```

input: tx_en, dout
output: din
inout: data



Inout ports are ports that can be either input or output depend on the configuration.
In Verilog, there are rules to define data type & connect data types to the ports. The data types are explained later.

Syntax

Data type

Type	Commonly use	Description
Net	wire	Structural connection between componnents. It does not hold value.
Register	reg	Represent variables that are used to store data.

```
module and4 (
    input  wire [3:0] din
  , output wire       dout
);

assign dout = din[0]&din[1]&din[2]&din[3];

endmodule //and4
```

```
module dff (
    input  wire rst_n
  , input  wire clk
  , input  wire din
  , output reg  dout
)

always @(posedge clk or negedge rst_n) begin
  if (!rst_n) dout <= 1'b0;
  else dout <= din;
end

endmodule //dff
```

A wire get its' value through assign

A reg get its' value in a procedural block such as always

Common mistake

wire and reg

When converting between code and schematic, a wire can be understood as real connection between components, it is a true wire to connect ports.
However, reg does not always represent sequential components as shown in DFF example.

MUX cell

dout is reg type, the variable stores data but the functionality is not. There is no case that dout can keep its' data.

```
module mux2(
    input  wire sel
  , input  wire din_0
  , input  wire din_1
  , output reg  dout
)

always @(sel or din_0 or din_1) begin
  if (sel==1'b0) dout = din_0;
  else dout = din_1;
end

endmodule //mux2
```

Latch cell

dout is reg type, the variable stores data and the function does to. It stores data when en is 0.

```
module latch(
    input  wire en
  , input  wire din
  , output reg  dout
);

always (en or din) begin
  if (en) dout = din;
end

endmodule //latch
```

ACTIVITY

Check 4.1, 4.2

FLEXIBLE DATA SIZE

When designing hardware, we can declare variables with a range (normally we call them bus). The language does support fixed size data type such as integer.
Unlike most programming languages, Verilog allows us to interact with each bit.

IMPORTANT NOTE

Data size

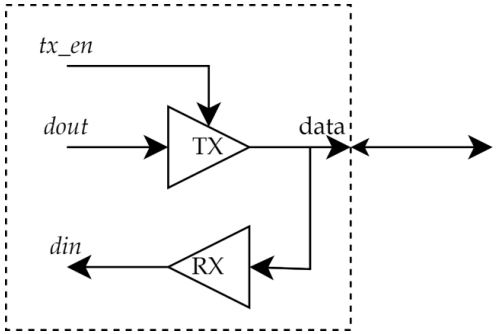
```
wire [1:0] a;
wire [3:0] b;
assign a[1] = 1'b1; // Assign different value for a[0] and a[1]
assign a[0] = 1'b0;
assign b[3:2] = a;  // Assign value of 'a' to a range of bus b
assign b[1] = a[0]; // Assign bit by bit for bus b
assign b[0] = a[1];
```

Syntax

Data value

Value	Description
0	Logic zero or false
1	Logic one or true
X	Unknown logical value
Z	The high impedance of the tri-state gate

inout port is HiZ if the driver is disabled.




```
assign data = dout1;
assign data = dout2;
```

dout1	dout2	data
1	1	1
0	0	0
Z	*	dout2
*	Z	dout1
1	0	X
0	1	X

Unknown value is where the value can't be identified as the other 3 values.

Syntax

Arithmetic Operator

Operator Type	Operator Symbol	Operation Performed
Arithmetic	*	Multiply
	/	Division
	+	Add
	-	Subtract
	%	Modulus
	+	Unary plus
	-	Unary minus

Same as C programming language.

Syntax

Logical Operator

Operator Type	Operator Symbol	Operation Performed
Logical	!	Logical negation
	&&	Logical and
	||	Logical or

```
if (a) // True if a is not 0
if (!a) // True if a is 0
if (a&&b) // True if both a and b are not 0
if (a||b) // True if a or b is 0
```

The result is true or false as in most programming language

Syntax

Relational Operator

Operator Type	Operator Symbol	Operation Performed
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal
	<=	Less than or equal

Same as other programming language

Syntax

Equality Operator

Operator Type	Operator Symbol	Operation Performed
Equality	==	Equality
	!=	Inequality
	===	Equality, compare X and Z
	!==	Inequality, compare X and Z

Same as other programming language

Unlike programming language, Verilog has value X and Z, thus it need “===“ and “!==“ to compare those value. Otherwise, it ignores the value.

Syntax

Reduction & bitwise Operator

Operator Type	Operator Symbol	Operation Performed
Reduction & bitwise	~	Bitwise negation
	&	and
	~&	nand
	|	or
	~|	nor
	^	xor
	^~	xnor
	~^	xnor

```
logic [2:0] a, b, c;
logic e;

initial begin
  a = 3'b101;
  b = 3'b011;
  c = ~a; // c = 3'b010
  e = |a; // e = 1'b1, reduction operation always result 1 bit
  c = a&b; // c = 3'b001;
  c = a|b; // c = 3'b111;
end
```

Operators work on bit scale instead of the whole signal.

The operators & and | are different with the logical operator && and ||.

Syntax

Shift Operator

Operator Type	Operator Symbol	Operation Performed
Shift	>>	Right shift
	<<	Left shift

```
logic [3:0] a;

initial begin
  a = 4'b0111;
  a = (a<<2); // a = 4'b1100;
  a = (a>>2); // a = 4'b0011;
end
```

MSB in Right shift and LSB in Left shift are filled with 0.

Similar to C programming language.

Syntax

Concatenation Operator

Operator Type	Operator Symbol	Operation Performed
Concatenation	{ }	Concatenation

```
logic [3:0] a, b;
logic [1:0] c;
logic [15:0] d;

assign d = {a, c, b, {2{c}}, 2'b10};
```

Assign portion values of 1 bus.

In above example, {2{c}} mean signal c is duplicated 2 times.

Syntax

Conditional Operator

Operator Type	Operator Symbol	Operation Performed
Conditional	?	Conditional

```
wire s0, s1;
wire i0, i1, i2, i3;
wire o;

// Select i* depends on s*
// - {s1, s0} =  2'b11, o = i3
// - {s1, s0} =  2'b10, o = i2
// - {s1, s0} =  2'b01, o = i1
// - {s1, s0} =  2'b00, o = i0
assign o = s1 ? (s0 ? i3 : i2) : (s0 ? i1 : i0);
```

Similar to if…else… clause.

Mainly used for wire data type, where if…else clause is not permitted.

Syntax

Procedural block

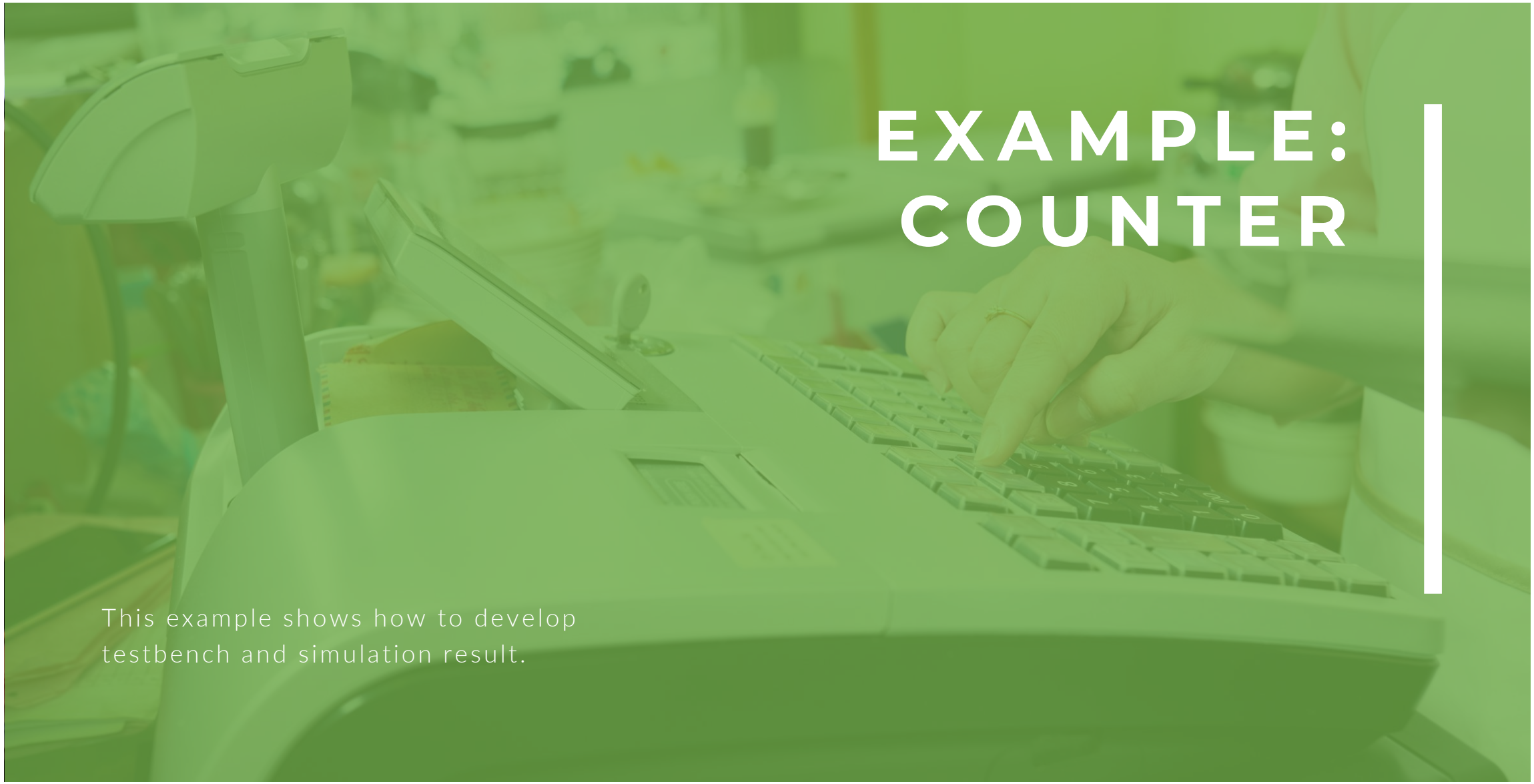
```
// begin...end is required if 1 block has more than 1 assignment
initial begin
  rst_n = 1'b0;
  ld_code_in = 4'b0000;
  fl_code_in = 4'b0000;
end
```

```
reg clk;

// This block does not use begin...end because it only has 1 assignment
initial clk = 0;
always #50ns clk = ~clk;
```

Procedural blocks are used to define behavior of a module. Register data types can get value from these blocks.

There are 2 types of procedural blocks:

- initial: Run once at the beginning of the simulation process. Don't use in RTL code because this does not reflect any hardware.
- always: Always runs since the simulation starts.



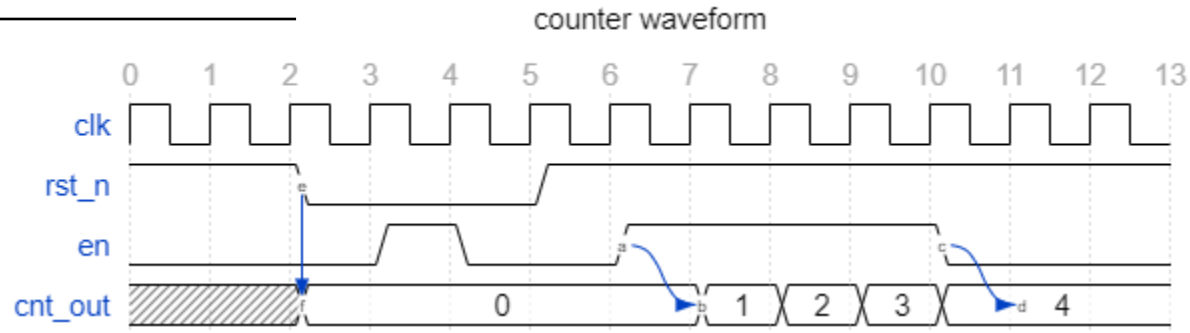
EXAMPLE: COUNTER

This example shows how to develop testbench and simulation result.

Example

Counter

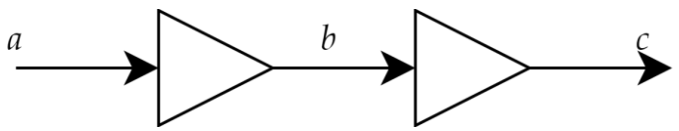
Port name	Size	Description
clk	1	Clock, active at risign edge
rst_n	1	Asynchronous reset, counter is reset to 0 whenever this signal is 0
en	1	Synchronous enable, counter can only count up when this signal is 1
cnt_out	4	Counter output



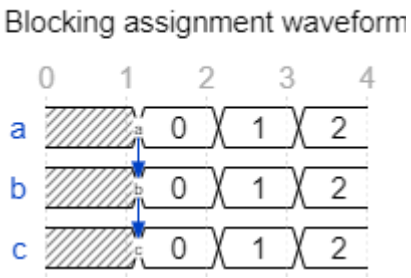
- At T0, cnt_out is undefined since it is starting point of simulation.
- T2: cnt_out is reset to 0 when rst_n = 0
- T4: en = 1 when rst_n = 0 so there is no change on cnt_out
- T6: en = 1 when rst_n = 1 so cnt_out starts counting up.
- T11: en = 0 so cnt_out is unchanged

Syntax

Blocking assignment




```
always @(a) begin
  b = a;
  c = b;
end
```

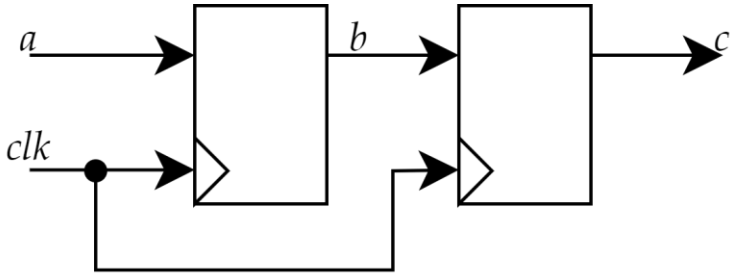


Use to represent combinational logic & latch

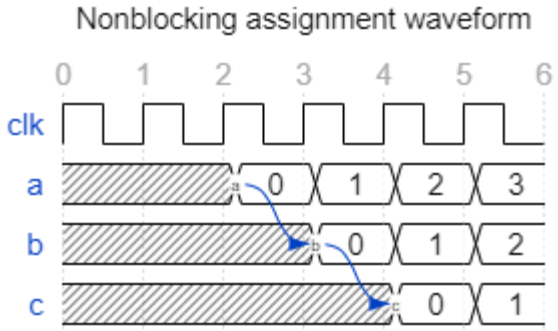
Blocking assignments are executed in order as they are coded. Each assignment blocks the next one until it is executed, thus the assignment is called *blocking*. They are made with "=", e.g. a=b

Syntax

Nonblocking assignment

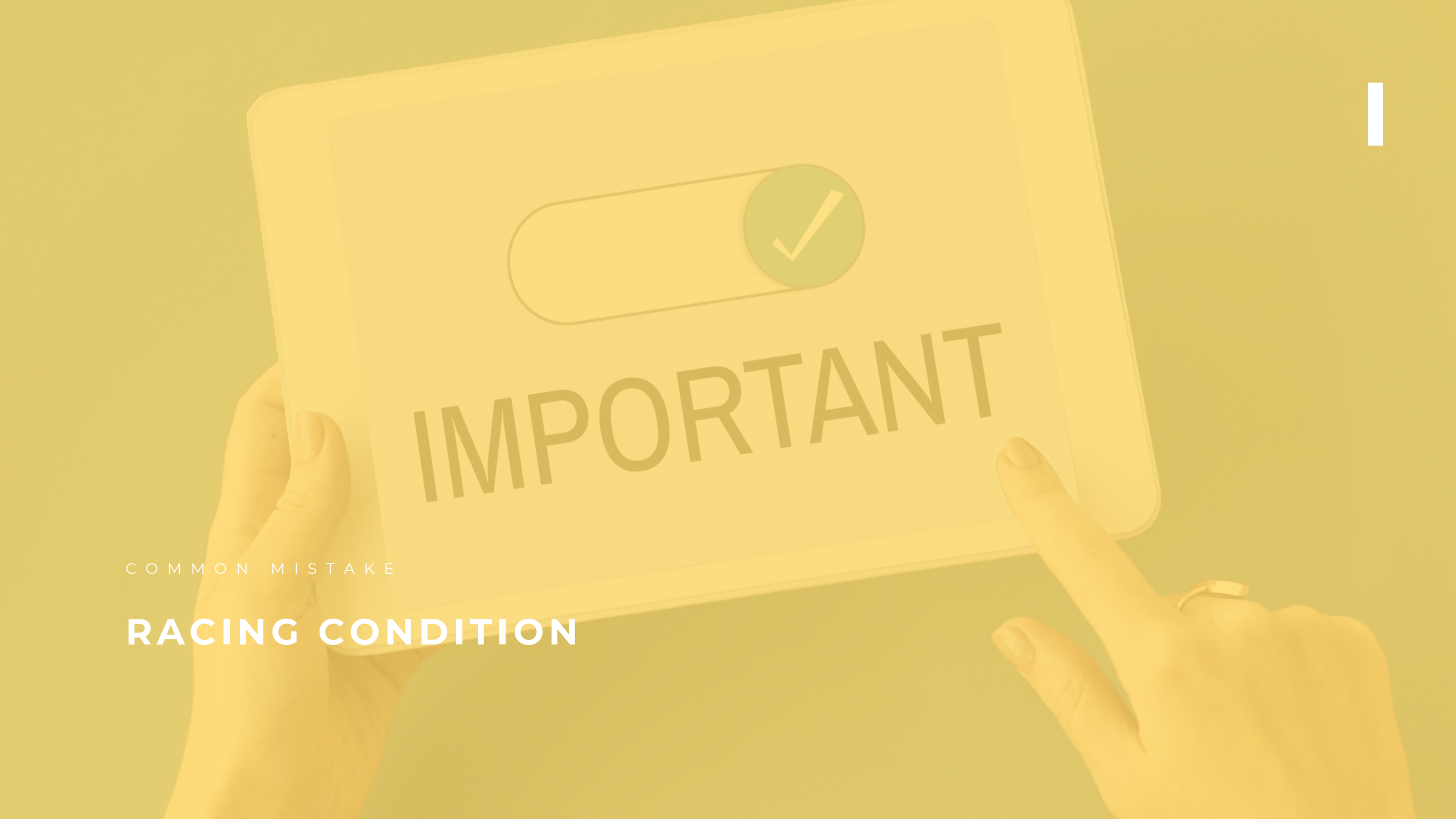



```
always @(posedge clk) begin
  b <= a;
  c <= b;
end
```



Use to represent flip flop

Nonblocking assignments are executed in parallel. They do not block each other thus they are called *nonblocking*. They are made with "<=", e.g., a<=b



COMMON MISTAKE

RACING CONDITION

Common mistake

Racing condition

```
always @(a) b = a;
always @(a) c = b;
```

Racing, c is unpredictable when a changes value

```
initial b = 0;
initial b = 1;
```

Racing, b is unpredictable at the beginning of simulation

Mostly happens with non blocking assignment

```
always @(posedge clk) b <= a;
always @(posedge clk) c <= b;
```

Not racing, this follows RTL coding rules and represents 2 serial flipflop

When there are 2 or more assignments happen at the same time, simulation tools can't define which assignment happens first and racing happens. It depends on tools' algorithm and the code to make order and it is unpredictable.

ACTIVITY

Check 4.3

Syntax

if…else statement

Why do we need first assignment for out & hit?

```
module encoder (
    input  wire [3:0] in
  , output reg  [1:0] out
  , output reg        hit
);

  always @(in) begin
    out = 2'd0;
    hit = 1'b0;
    if (in==4'b0001) begin
      out = 2'd0;
      hit = 1'b1;
    end else if (in==4'b0010) begin
      out = 2'd1;
      hit = 1'b1;
    end else if (in==4'b0100) begin
      out = 2'd2;
      hit = 1'b1;
    end else if (in==4'b1000) begin
      out = 2'd3;
      hit = 1'b1;
    end
  end
endmodule
```

The syntax is not different with programming languages.

Syntax

case statement

```
module decoder (
    input  wire [1:0] in
  , output reg  [3:0] out
);

  always @(in) begin
    case (in)
      2'b00 : out = 4'b0001;
      2'b01 : out = 4'b0010;
      2'b10 : out = 4'b0100;
      2'b11 : out = 4'b1000;
    endcase
  end

endmodule
```

case statement is used when there are too many branches.

If we don't care about other bits, case can be written as:

```
casez (a)
  2'b?1: // True if bit [0] is 1, regardless value of bit[1]
  2'b10: // True if bit [0] is 0 and bit [1] is 1
  default: // Remaining cases
endcase
```

However, casez and casex are not recommended in RTL code because they allow Z and X values.

Syntax

for loop statement

for loop is the same as other languages.

```
module decoder_tb;
  reg [1:0] in;
  wire [3:0] out;

  int idx;

  initial begin
    $display("Initial the design");
    $monitor("Monitor: %t in: %d, out: %b", $realtime, in, out);
    #5ns in = 2'd0;
    for (idx=0; idx<4; idx=idx+1) begin
      #5ns in = idx;
    end
    $display("Reach final value");
  end

endmodule
```

The example is used in test bench

ACTIVITY

Check 4.4

Verification Syntax

repeat & wait statements

```
initial begin
  idx = 0;
  repeat (4) begin
    #5ns in = idx;
    idx = idx+1;
  end
end

initial begin
  $display("Initial the design");
  $monitor("Monitor: %t in: %d, out: %b", $realtime, in, out);
  wait (idx==4) $display("Reach final value");
end
```

repeat runs a statement in a defined number of times.
wait is a level condition that hold execution until it is true.

The example is used in test bench

Verification Syntax

System tasks & function

They are predefined by the language. Some of them can be also use for RTL code.

- $display and $strobe display once every time they are executed.
- $monitor displays every time one of its parameter changes.
- $finish exits the simulator back to the operating system.
- $random generates a random integer every time it is called.
- $realtime returns the current simulation time as a real number

begin…end: Sequential block
fork…join: Parallel block

```
// file: sequential_tb
module sequential_tb;
  reg [1:0] idx;

  initial begin
    $monitor("Monitor: %t idx: %d", $realtime, idx);
    // Each assignment blocks the latter ones
    #5ns idx = 0;
    #10ns idx = 1;
    #15ns idx = 2;
  end

endmodule
```

```
Monitor:                    0 idx: x
Monitor:                 5000 idx: 0
Monitor:                15000 idx: 1
Monitor:                30000 idx: 2
```

```
// file: parallel_tb
module parallel_tb;
  reg [1:0] idx;

  initial begin
    $monitor("Monitor: %t idx: %d", $realtime, idx);
    // All assignments run at once
    fork
      #5ns idx = 0;
      #10ns idx = 1;
      #15ns idx = 2;
    join
  end

endmodule
```

```
Monitor:                    0 idx: x
Monitor:                 5000 idx: 0
Monitor:                10000 idx: 1
Monitor:                15000 idx: 2
```

Verification Syntax

Procedural assignment group

Syntax

Procedural timing control

```
#<time> statement;
```

```
<signal 1> = #<time> <signal 2>;
```

```
#5ps a = 1; // a = 1 at 5ps
b = 0;      // b = 0 at 5ps
#10ps;
a = #5ps 0; // a = 0 at 20ps.
b = 1;      // b = 1 at 15ps
```

```
@([<posedge>|<negedge>] <signal>) <statement>;
```

```
wait (<expression>) <statement>;
```

Timing for procedural blocks is introduced across the document.

- Delay controls.
- Edge-Sensitive Event controls.
- Level-Sensitive Event controls-Wait statements.

Syntax

Directive

```
`define <MACRO>
```

```
`define display_info(message) $display("%m [INFO]    %t %s %d: %s", $realtime, `__FILE__, `__LINE__, message)

...
initial #1200 `display_info("Test message");
```

Directives are guidance for tool to compile the source code.

```
`ifdef <NAME>
`else
`endif
```

```
initial begin
`ifdef FIRST
  $display("FIRST");
`else
`ifdef SECOND
  $display("SECOND");
`else
  $display("NONE");
`endif
`endif
end
```

Simulation tools do at least 2 steps:

- Compile code into a database
- Simulate the code

Syntax

Parameter

```
module counter_param #(
  pCOUNTER_WIDTH = 4; // Default value of the counter
) (
    input  wire                       clk
  , input  wire                       rst_n
  , input  wire                       cnt_en
  , output reg  [pCOUNTER_WIDTH-1:0] cnt_out
);
  always @(posedge clk or negedge rst_n) begin
    if (!rst_n) cnt_out <= 0;
    else if (cnt_en) cnt_out <= cnt_out + 1'b1;
  end
endmodule
```

```
counter_param #(
  .pCOUNTER_WIDTH(8)
) u_dut (
    clk    ( clk     )
  , rst_n  ( rst_n   )
  , cnt_en ( cnt_en  )
  , cnt_out( cnt_out )
);
```

A module with parameters allow the instances to be different.

The simplest example for parameter is capacitors have the same function, but they have different capacitance values.

Syntax

Task

```
task compare;
input  [1:0] drv_in;
input  [3:0] exp_out;
output       error;
begin
  error = 1'b0;
  #5ns in = drv_in; // Task can drives the global variable
  #5ns if (out!==exp_out) begin
    $display("Data is unexpected out:%b exp:%b", out, exp_out);
    error = 1'b1;
  end
end
endtask
```

A task does a common set of steps that other parts of the code can call it

Task is only used in testbench

Syntax

Function

```
assign a[0] = c[0] ? b[0] : b[1];
assign a[1] = c[1] ? b[1] : b[2];
assign a[2] = c[2] ? b[2] : b[3];
```

```
function [2:0] muxing;
  input [2:0] sel;
  input [3:0] din;
  integer idx;
  begin
    for (idx=0;idx<3;idx=idx+1)
      muxing[idx] = sel[idx] ? din[idx] : din[idx+1];
  end
endfunction
```

A function is a replacement for a combinational logic (can be used in RTL code), while task replaces for a routine (like in testbench).

- A function definition cannot contain any time-controlled statements—that is, any statements introduced with #, @, or wait.
- Functions cannot enable tasks.
- A function definition must contain at least one input argument.
- A function definition must include an assignment of the function result value to the internal variable that has the same name as the function.
- A function definition can not contain an inout declaration or an output declaration

FINITE STATE MACHINE (FSM)

Explanation & example

F S M

Definition

“FSM [...] is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some inputs; the change from one state to another is called a transition.”

--- Wikipedia

FSM is not syntax; it is a digital design.

Safe

State: Locked, Unlocked

Transition:

- Correct combinations, state transfer to Unlocked
- Wrong: Reset to initial Locked

Traffic light

State: Green, Yellow, Red

Transition: State changes after a given time.

FSM types

Mealy & Moore

Only use Moore version unless it is undoable.

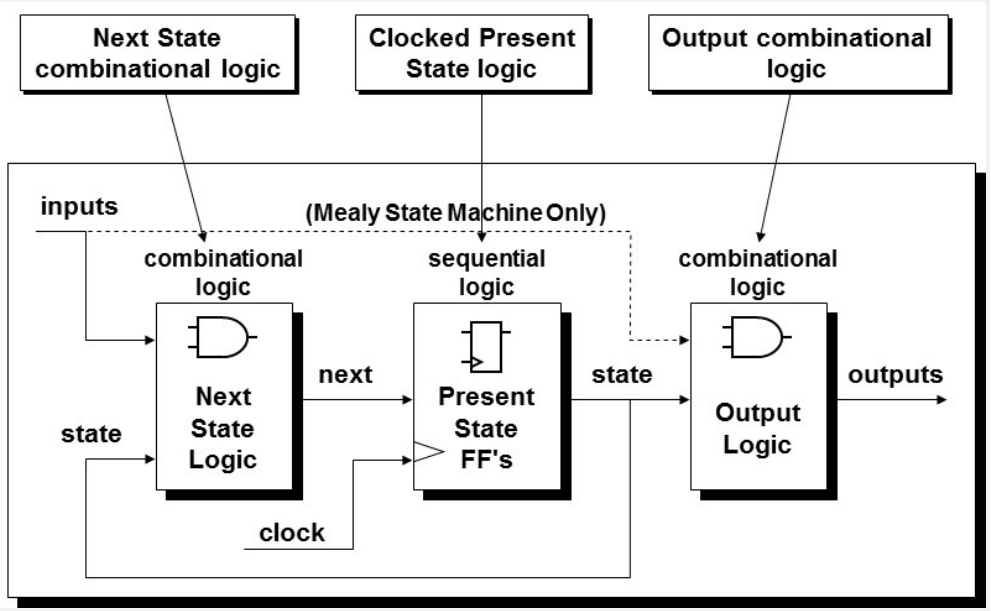


Figure credit: Clifford E. Cummings, posted in SNUG

Moore state machine: outputs are only a function of the present state.
Mealy state machine: have one or more outputs that are a function of the present state and one or more inputs.

Example

Binary bit stream

Port name	Size	Description
clk	1	Clock, active at risign edge
rst_n	1	Asynchronous reset, counter is reset to 0 whenever this signal is 0
din	1	input
lock	1	output

The output lock asserts when din data matches with the sequence 1001, it keeps at high if the data repeat the sequence, for example 1001100110.... It is low if din has unexpected value

SystemVerilog

SystemVerilog is an expansion of Verilog. It supports both RTL and Verification teams to solve a lot of problems when working with Verilog.

ACTIVITY

Check 4.6

Syntax

always blocks

```
always_ff @(posedge clk or negedge rst_n)
  <code to describe flipflop function>
```

```
always_latch
  <code to describe latch function>
```

```
always_comb
  <code to describe combinational logic function>
```

3 types of always block for 3 different functions.

- always_ff is used for flipflop.
- always_latch is used for latch.
- always_comb is used for combinational logic.

Syntax

Logic datatype

```
logic a;
initial #5 a = 1;
initial #5 a = 0;
```

logic can replace reg and most of wire cases.

- A *logic* variable can't be driven by multiple sources at the same time. (Example assert an error message)
- *logic* data type can't be assign to "inout" port.

Syntax

Assigning all bits

```
logic [2:0] a;

initial begin
  a = '1; // a = 3'b111;
  a = 'x; // a = 3'bxxx;
end
```

There are many cases that engineer want to assign all bits to the same value.

- '0 : Set all bits to 0
- '1: Set all bits to 1
- 'X or 'x : Set all bits to x
- 'Z or 'z : Set all bits to z

Syntax

Packed array

```
wire a [2:0]; // 3 elements array, each has 1 bit
wire [3:0] b [5:0]; // 6 elements array, each has 4 bits
```

```
logic [5:0][3:0] b;

initial begin
  b = 18'ha_b_c_d_e_f; //b[5]=4'ha, b[4]=4'hb, etc.
  b[5] = 4'h0; // b = 18'h0_b_c_d_e_f
  b = (b<<1); // b = 18'hb_c_d_e_f_0
end
```

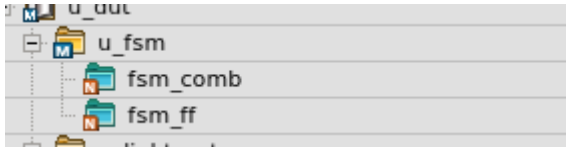
Verilog only has 1 type of array.
SystemVerilog has packed and unpacked array.

Packed array works the same as bus in Verilog with additional dimensions.

Syntax

Naming block

```
module mdl_name();
...
endmodule : mdl_name

task tsk_name();
...
endtask : tsk_name

initial begin : blk_name
...
end : blk_name
```



Block naming helps reading & debugging the code easier.

If ending name does not match beginning name, it is an error.

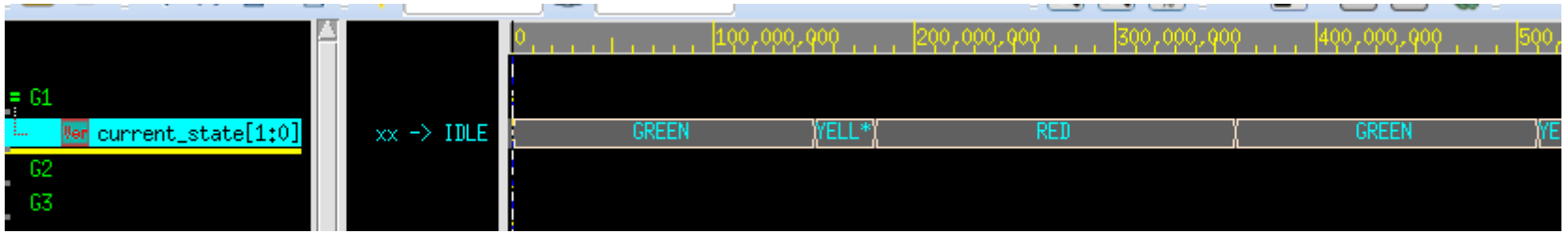
Syntax

Enumeration data type

```
enum integer {IDLE=0, GNT0=1, GNT1=2} state; // define base type as integer, and initialize each state's value
enum integer {RED,GREEN,ORANGE} color; // Automatically assign value. RED=0, GREEN=1, ORAGE=2
enum integer {BRONZE=4, SILVER, GOLD} medal; // The first value is 4, thus the next ones are 5 & 6
```

```
enum {BRONZE=4, SILVER, GOLD} medal;
enum {GOLDEN} medal_ex;

initial medal = GOLDEN; // This will result in an error
```

enum is useful in FSM



enum is better than parameter for FSM:

- We can't assign an enum to names that are not defined in the declaration.
- Tools display enum as name, not just value thus it is much easier to debug.
- enum in EDA tool is shown as name instead of value as default.

Syntax

Struct data type

struct is used to group signals that have related function.

It can be used for ports too.

```
struct packed {
  logic a;
  logic [1:0] b;
  logic c;
} struct_eg;

initial begin
  struct_eg = '{a:1'b1, default:0}; // all bits except a are 0
  struct_eg.b = 2'b11; // assign b to 1
  struct_eg[0] = 1'b1; // assign c to 1
end
```

```
// To reuse a set of enum and struct, we have to define
// them first ussing typedef
typedef logic [1:0] enum {
  GREEN
  , YELLOW
  , RED
} state_t;
typedef struct packed {
  logic init;
  logic [2:0] light;
} signal_t;

state_t current_state, next_state;
signal_t current_sig, next_sig;
```

Syntax

Enhanced task and function

function now can have struct data type.

```
typedef struct packed {
  logic hit;
  logic [1:0] code;
} pe_t;

function automatic pe_t pe (
  input [4:0] in
);
  integer idx;

  pe = '{default:0};
  for (idx=0; idx<5; idx=idx+1) begin
    if (in[idx]==1'b1) begin
      pe.hit = 1'b1;
      pe.code = idx;
      return pe;
    end
  end

  return pe
endfunction : pe
```

Some enhancements:

- automatic is for dynamic memory. In Verilog, they use static memory and cause troubles when there are more than one code call them at one time.
- Remove begin…end
- struct can be used for function, so now function can return more than 1 output.
- function now has return keyword to escape it before the last line.

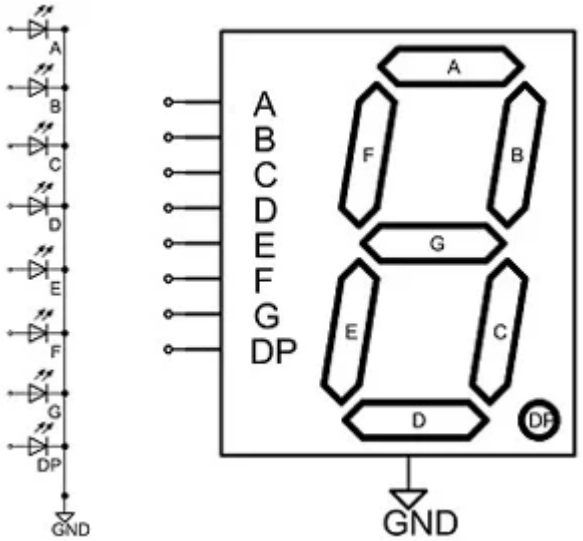
Guide project

A traffic light controller using FSM.

Example

Traffic light

Port	Direction	Bus size	Description
clk	in	1	10MHz clock
rst_n	in	1	Active low asynchronous reset
en	in	1	Active high enable
red_light	out	1	Enable red light
yellow_light	out	1	Enable yellow light
green_light	out	1	Enable green light
display_led	out	16	7 segment display, 2 digits



Specification:

- Clock frequency: 10MHz
- Control red (18s), yellow (3s), green light (15s)
- Control 7 segment light

ACTIVITY

Check 5.1, 5.2