TOC

- TOC
- 1. Overview
  - 1.1. Prerequisites
  - 1.2. Analog system vs. Digital system
  - 1.3. Digital design flow
  - 1.4. This program's target
- 2. RTL designer
- 3. Common logic cells
  - 3.1. Combinational logic cells
  - 3.2. Sequential logic cells
    - 3.2.1. D-latch
    - 3.2.1. D-flipflop
- 4. Verilog/SystemVerilog syntax
  - 4.1. Module and instance
  - 4.2. Port
  - 4.3. Data type
    - 4.3.1. Net and Register
    - 4.3.2. Data value
  - 4.4. Verilog operator
    - 4.4.1 Arithmetic
    - 4.4.2. Logical
    - 4.4.3. Relational
    - 4.4.4. Equality
    - 4.4.5. Reduction and bitwise
    - 4.4.6. Shift
    - 4.4.7. Concatenation
    - 4.4.8. Conditional
  - 4.5 Procedural block
    - 4.5.1 initial and always blocks
    - 4.5.2. Example: Counter
  - 4.6. Blocking & nonblocking assignment
  - 4.7. Statement
    - 4.7.1. if...else statement
    - 4.7.2. case statement
    - 4.7.3. for loop statement
  - 4.8. Verification useful syntax
    - 4.8.1. repeat, wait

        - 4.8.2. System tasks & functions
        - 4.8.3. Procedural assignment group
        - 4.8.4. Procedural timing control
    - 4.9. Directive & parameter
        - 4.9.1 Directive
        - 4.9.2. Parameter
    - 4.10. Task and function
        - 4.10.1. Task
        - 4.10.2. Function
    - 4.11. FSM coding
        - 4.11.1. Definition
        - 4.11.2. Types of FSM: Mealy & Moore
        - 4.11.3. Example: Binary bit stream
    - 4.12. SystemVerilog
        - 4.12.1. always_ff, always_comb, always_latch
        - 4.12.2. Data type `logic`
        - 4.12.3. Assigning constant value
        - 4.12.4. Packed array
        - 4.12.5. Naming block
        - 4.12.6. Data type enumerations
        - 4.12.7. Data type `struct`
        - 4.12.8. Enhanced for task and function
- 5. Guide project
    - 5.1. Brief designing the blocks
    - 5.2. Detail designing the blocks
        - 5.2.1. Second clock counter
        - 5.2.2. Light control counter
        - 5.2.3. FSM block
        - 5.2.4. Counter decoder
        - 5.2.5. Connect altogether
    - 5.3. Testbench
- 6. Reference
- A. Appendix A: Using debugging EDA tool
    - A.1. Running simulation
    - A.2. Running simulation
- B. Appendix B: Ending course activity
    - B.1. Signal toggling detector
    - B.2. Hello world display
    - B.3. Bit streaming
    - B.4. Morse code
    - B.5. Data alignment

    - B.6. Data aligment with additional signal
- C. Revision

1. Overview

1.1. Prerequisites

In order to catch up with the speed of this document, student need to have following knowledge:

- Digital number
- Combinational logic gates' function and truth tables
- Sequential logic cells' (D-Flipflop, Latch) function and truth tables
- Boolean algebra
- Karnaugh map

1.2. Analog system vs. Digital system

Analog system is defined as:

> "Systems which are capable of processing a continuous range of values varying with respect to time are called analog systems. In analog representation a quantity is represented by a voltage, current, or meter movement that is proportional to the value of that quantity. Analog quantities such as those cited above have an important characteristic: they can vary over a continuous range of values."
>
> --- Deepak Kumar Tala, asic-world

Digital system is defined as:

> "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

Advantages of a digital system:

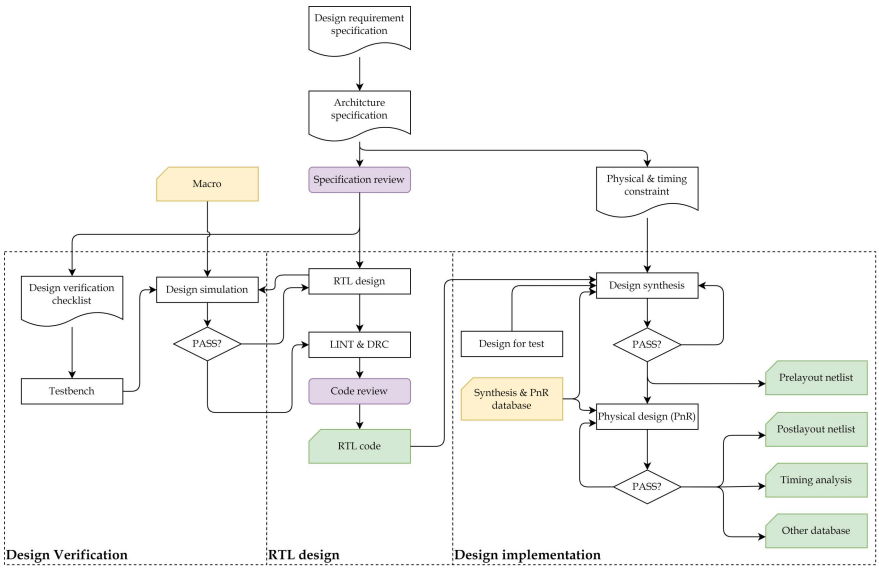
- Easy to design
- Digital system only cares about value (HIGH or LOW), not the exact voltage.
- Easy to store information.
- Programmable.
- Less affected by noise. The system only be affected if noises are big enough to reverse the value since it only cares about HIGH or LOW.
- Size is smaller (more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

Disadvantages of a digital system:

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

1.3. Digital design flow

Figure below shows a typical design flow for a digital product.



- Design requirement specification: This is requirement from customer or other teams, it describes the functionality and characteristic (speed, area, etc.) of the product.
- Architecture specification: Design the architect of the product, separate it into smaller blocks and functionality for each of them.
- RTL design: Convert the architecture from text based to Hardware Description Language (HDL) such as Verilog or SystemVerilog. This step may require to have smaller scale of architecture design, called block design or micro architecture design.
    - Lint: There are common mistakes in RTL code that lead to wrong behavior between RTL and the final data. Lint check makes sure the code is clean and prevent such issues.
    - DRC: Design rule check. There are rules that a design has to follow to make sure there is no issue in real world. This is similar to Lint check, but it verify the design itself, not the coding style.
- Design verification: Verify if the RTL code matches with the specification or not. This step also use Verilog/SystemVerilog with simulation tools to verify the design.
    - Checklist: Include items to verify the design.
    - Testbench: The environment to verify the design
    - Design simulation: Put the macro & RTL code into the environment and run simulation to verify the design.
- Design implementation: Convert the RTL code to logic cells and create physical database to manufacture it.
    - Synthesis: Convert the RTL code to logic cells. The cells' connection and function are still written in Verilog syntax, and called Gate level netlist (GLN). After this phase, there are tests to make sure the netlist and RTL code match together, or the design can meet the performance requirement, etc.
    - Design for test (DFT): In order to verify the design in physical world (when it is manufactured), it must included testing functions beside normal functions. This step

adds the functions into the design.
    - Physical design (PD): This step uses logical GLN and design physical version of the design. The result of this step can be used to send to manufacturer.

In the flow above, all "verification" steps returns to previous ones if the criteria does not match. For example Prelayout STA can push data back to synthesis or RTL coding step, depends on the problem.

1.4. This program's target

As shown in previous section, Verilog/SystemVerilog is just a tool to describe/verify a digital design. Learning the language syntax must be combined with learning the design concept (RTL code) or verification concept (DV). As Verilog language was first created to describe hardware (HDL), thus this program focuses more on RTL part.

This program 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.

> ⚠ **ATTENTION:** This program does not provide either deep explanation of syntaxes or cover all syntaxes of the language. Students are expected to do further investigation on suggested topics.

2. RTL designer

To start, we have to understand RTL engineer is not a software engineer. Even though both seems to write some code on computer, RTL engineer is a hardware designer, while a software engineer writes application that run on the hardware.
Verilog/SystemVerilog is hardware description languages (HDL), they are not programming language. Using HDL to write RTL code thus is not a programming process. In this process, in fact, we convert a hardware design (in textes, diagrams) into a language that machines can understand. So when writing RTL code, we have to think about how that code is converted into the hardware.

Through out this program, we will learn the syntaxes with diagrams to show how some code blocks are converted to hardware.

3. Common logic cells

In this section, we will review some common logic gates and sequential elements.

3.1. Combinational logic cells

Please review the functionality and truth tables of the following logic cells: INV, AND, OR, XOR, MUX

3.2. Sequential logic cells

There are 2 types of sequential logic cells:

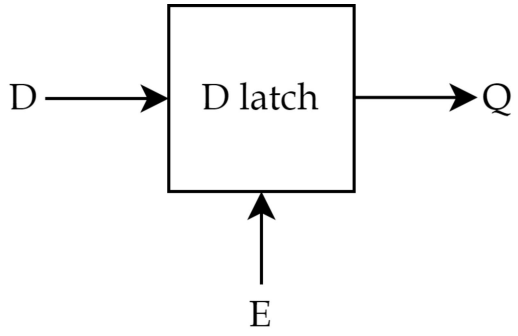
- Latch: The output senses the level of the inputs to capture data.
- Flipflop: The output sense the edge of the inputs to capture data.

Commonly use is D-latch and D-flipflop

3.2.1. D-latch

D-latch normally has 3 pins:

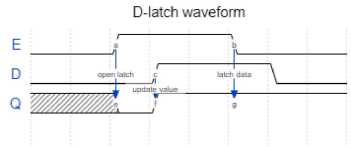
- D: data input
- E: Enable for data goes through the latch
- Q: data output In next sections, we simply call it is latch.



When E is 1, data from D goes to Q. When E is 0, data on Q is latched and kept as is.

Truth table:

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



3.2.1. D-flipflop

D-flipflop normally has 4 pins:

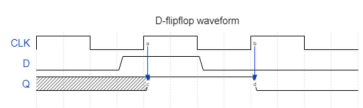
- D: data input
- CLK: clock signal
- RN: asynchronous reset
- Q: Data out In next sections, we simply call it is flipflop.

When CLK has rising edge, data from D goes to Q. Otherwise, Q keeps its data. If RN is 0, Q is reset to 0 regardless values of D or CLK.

Truth table:

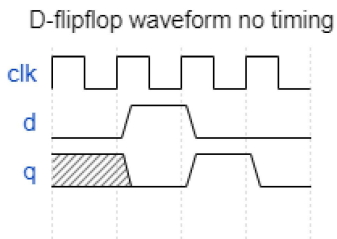
CLK	D	RN	Q
*	*	0	0
0	*	1	Q
1	*	1	Q
R	0	1	0
R	1	1	1
F	*	1	Q

The waveform below does not include RN.



There are some other variant of the flipflop (e.g. CLKN where clock is active at falling edge, SN where Q is set to 1 instead of 0, etc.)

> **NOTE:** The waveform for flipflop in digital field normally doesn't have delay between D and CLK. In this kind of waveform, CLK captures value of D right before the edge and sends to Q.



ACTIVITY 3.1: Can you design a 2 to 4 decoder with function as below:
Input: i0, i1
Output: o0, o1, o2, o3
The outputs are all 0, except the index that matches with inputs' values is 1:

- o0 = 1 when i1 = 0, i0 = 0 ({i1,i0} = 2'd0)
- o1 = 1 when i1 = 0, i0 = 1 ({i1, i0} = 2'd1)
- o2 = 1 when i1 = 1, i0 = 0 ({i1,i0} = 2'd2)
- o3 = 1 when i1 = 1, i0 = 1 ({i1, i0} = 2'd3)

This can be solved using K-map table.

ACTIVITY 3.2: Can you design a 2 to 4 encoder with function as below:
Input: i0, i1, i2, i3 Output: o0, o1, hit There is only 1 bit of input is 1 at a time, the output indicates the index of input to be 1

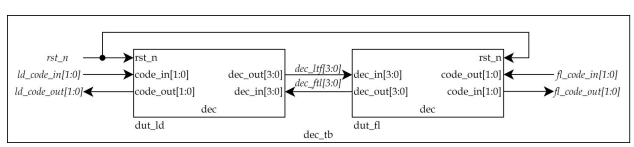
- o0 = 0, o1 = 0 ({o1,o0} = 2'd0), hit=1 when i0 = 1
- o0 = 0, o1 = 1 ({o1,o0} = 2'd1), hit=1 when i1 = 1
- o0 = 1, o1 = 0 ({o1,o0} = 2'd2), hit=1 when i2 = 1
- o0 = 1, o1 = 1 ({o1,o0} = 2'd3), hit=1 when i3 = 1
- All outputs are 0 if none of above conditions matches

4. Verilog/SystemVerilog syntax

4.1. Module and instance

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.

Figure below about a module `dec_tb` that contains 2 instances of the same module `dec`.

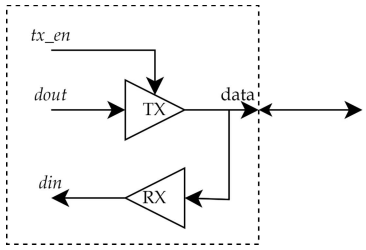


Verilog code for the diagram:

```
// 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

// 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
```

4.2. Port

Input and Output declaration can be found in the Example: Module and instance. A design also include port call inout, which can act both input or output, depends on configuration.
Figure below shows a typical schematic for this kind an inout port.



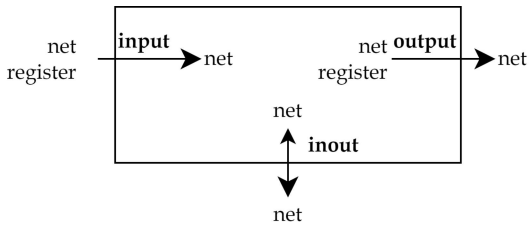
In the figure above, TX is a buffer with control signal. It only drives data when being enabled.

- *tx_en*=1, the buffer TX is actived thus *data*=*dout* and *din*=*dout*.
- *tx_en*=0, TX is not actived thus *data* and *din* depends on the outside to drive the value.

This kind of port is very popular in FPGA and Microcontroller, where generic ports can be defined as "in" or "out" depends on application.

Port connection rule:

- Input: Internally always net type, externally the port can be connected to either net or register type.
- Output: Internaly are net or register type, externally it always connect to net type.
- Inout: Internally and externally are all net type. The data type is explained in next section.



A signal can be defined as 1 bit or a bus (multiple bit) as shown in previous examples.

4.3. Data type

4.3.1. Net and Register

There are 2 group of data types in Verilog

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

Example below shows an AND gate logic with 4 inputs.

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

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

endmodule //and4
```

In the example above, `dout` does not hold data. It is a result of 4 bits of `din`, whenever `din` changes value, `dout` will reflect the changes on the output.

`assign` syntax is used to assign value for net type variables.

> **NOTE:** Above code can be reduced by using reduction operation.
>
> ```
> assign dout = &din;
> ```
>
> Operation is explained in 4.4. Verilog operator.

Example below shows a typical flipflop.

```
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
```

In the example above, `dout` changes value if 1 of the 2 events happen

- Rising edge of `clk`
- Falling edge of `rst_n`

If there both events don't happen, `dout` **holds** it current value. This means no matter what the value of `din` is, `dout` only updates value under certain condition.

> **ATTENTION:** COMMON MISTAKE
> 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 above.
> This leads to common mistaken when we write code, that the code behavior does not match with what we think.

Example below shows a MUX written by register type signals.

```
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
```

In example above, eventhough `dout` is defined as `reg`, it still represents a MUX. `dout` changes value whenever there is an event of `sel` or `din_0` or `din_1` changes value. And `dout` is assigned to either `din_0` or `din_1` depends on `sel`, it can't reallly "hold" any data. This kind of assignment is called *full-case*, where all cases of `sel` are defined.

> **ACTIVITY 4.1:** Can you rewrite the code where `dout` is `wire`?

Example below shows a LATCH code.

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

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

endmodule //latch
```

The LATCH code is quite similar to MUX version, but because the condition is not *full case*, `dout` can still store data. In this case, `dout` also changes whenever there is an event of `en` or `din` changes value. However, `dout` only really catch new data if `en` is 1, it still **holds** data when `en` is 0.

> **ACTIVITY 4.2:** Further reading about other data types such as `string`, `int`, etc. Mostly used when doing verification.

> **NOTE:** As shown in previous samples, ports, `wire`, `reg` can be defined with a range of bit as called bus. But there are other data types that have fixed size, for example `integer` has 32 bits just like `C` programming language.
> Unlike most programming language, Verilog allow us to work with each bit of the variable.
> Example:
>
> ```
> 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];
> ```

> **ATTENTION:** When working with digital design, people also use the term "register". This term is a replacement for flipflop in most cases.

4.3.2. Data value

There are 4 basic value in Verilog:

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

The inout port in section Port is an example of Z value. When `tx_en` is 0, signal `data` becomes high-Z and depends on outside to drive value. If this signal doesn't connect to anywhere (or only connects to inputs of other blocks), the signal's value is Z.

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

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

Table below shows the value of `data` depends on `dout1` and `dout2`.

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

4.4. Verilog operator

List of Verilog's operator:

Operator Type	Operator Symbol	Operation Performed
Arithmetic	*	Multiply
	/	Division
	+	Add
	-	Subtract
	%	Modulus
	+	Unary plus
	-	Unary minus
Logical	!	Logical negation
	&&	Logical and
	||	Logical or
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal
	<=	Less than or equal

Operator Type	Operator Symbol	Operation Performed
Equality	==	Equality
	!=	Inequality
	===	Equality, compare X and Z
	!==	Inequality, compare X and Z
Reduction & bitwise	~	Bitwise negation
	&	and
	~&	nand
	|	or
	~|	nor
	^	xor
	^~	xnor
	~^	xnor
Shift	>>	Right shift
	<<	Left shift
Concatenation	{ }	Concatenation
Conditional	?	Conditional

4.4.1 Arithmetic

It is the same as normal operators like `C` programming language

4.4.2. Logical

The result of this operator group is 1 bit, like `true` and `false` in most programming language.

```
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
```

4.4.3. Relational

This group is the same as other languages.

4.4.4. Equality

Same as other languages, with note of `===` and `!==` are used to compare signals contain `X` and `Z`

4.4.5. Reduction and bitwise

As we know, Verilog represents hardware thus it work with bit of data, the operators in this section work in bitwise.

```
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
```

4.4.6. Shift

The bits are "shift" left or right with this operation, the LSB or MSB are fill with 0, depends on the direction.

```
logic [3:0] a;

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

4.4.7. Concatenation

It is used to assign portion value of 1 bus.

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

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

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

4.4.8. Conditional

It is used as an (if..else...) clause for `wire` data type.

```
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);
```

4.5 Procedural block

4.5.1 initial and always 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.

In the file `dec_tb.v` in Example: Module and instance, all `reg` type signals have unknown (X) value since nothing drives them. To initialize the value, we can use `initial` 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
```

`initial` block is only used for verification, while `always` can be used for RTL code. `always` can work with or without condition. In no condition case, it repeat the code again and again (this kind of code is only used in verification).

```
reg clk;

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

In the snippet above, `clk` is initialized with value 0 at the beginning of simulation. `clk` switches its' value in each 500ns, which creates a 10MHz clock.

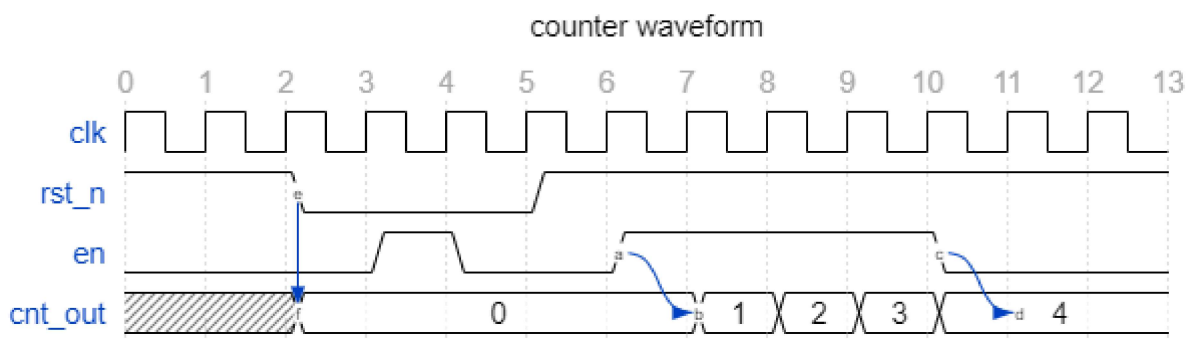
Mux example code shows example for always block that condition is the changing value of signals. DFF example code shows example for always block that condition is the edges (positive and negative) of signals.

4.5.2. Example: Counter

This example shows how to develop testbench and simulation result. To use waveform viewer tool or run the simulation package, please refer to 7. Appendix: Using debugging EDA tool. Design a 4 bits counter to with asynchronous reset and synchronous emable.

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

Waveform of the design:



- 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

```
// file: counter.v
module counter (
    input  wire      clk
  , input  wire      rst_n
  , input  wire      en
  , output reg [3:0] cnt_out
);

always @(posedge clk or negedge rst_n) begin
  if (!rst_n) cnt_out <= 4'b0;
  else if (en) cnt_out <= cnt_out + 1'b1;
end

endmodule // counter
```

```
// file: counter_tb.v
module counter_tb;
  reg        clk;
  reg        rst_n;
  reg        en;
  wire [3:0] cnt_out;

  always begin
    #25ns clk = ~clk;
    #50ns clk = ~clk;
    #25ns;
  end

  initial begin
    // display the first log
    $display("Initial the design");
    // Monitor check all signals, it will rewrite to log file if any of them ch
    $monitor("Monitor: %t rst_n: %b, en: %b, cnt_out: %d", $realtime, rst_n, er
    clk = 1'b0;
    rst_n = 1'b0;
    en = 1'b0;

    // Release rst_n after 5 clk cycles)
    #(5*100ns) rst_n = 1'b1;
    $display("Release reset");
    // Enable counter after 2 clk cycles)
    #(2*100ns) en = 1'b1;
    $display("Enable counter");
    // Disable counter after 10 clk cycles)
    #(10*100ns) en = 1'b0;
    $display("Disable counter");

    #(10*100ns) $finish;
  end

  counter dut (
      .clk    ( clk     )
    , .rst_n  ( rst_n   )
    , .en     ( en      )
    , .cnt_out( cnt_out )
  );

endmodule // counter_tb
```

In simulation script, do the following commands:

```
module load vcs
module load verdi
run.csh counter
```

The script load required files and run simulation. The simulation script can be found in `compile.log` and `sim.log`.

- `compile.log`: Log file in compile phase. In this phse VCS convert Verilog codes to data base. We use this log to debug if there is any syntax errors and warnings.
- `sim.log`: Simulation result. This log displays any message we want to write out during simulation phase.

The result of the simulation:

```
Initial the design
Monitor:                    0 rst_n: 0, en: 0, cnt_out:  0
Release reset
Monitor:               500000 rst_n: 1, en: 0, cnt_out:  0
Enable counter
Monitor:               700000 rst_n: 1, en: 1, cnt_out:  0
Monitor:               725000 rst_n: 1, en: 1, cnt_out:  1
Monitor:               825000 rst_n: 1, en: 1, cnt_out:  2
Monitor:               925000 rst_n: 1, en: 1, cnt_out:  3
Monitor:              1025000 rst_n: 1, en: 1, cnt_out:  4
Monitor:              1125000 rst_n: 1, en: 1, cnt_out:  5
Monitor:              1225000 rst_n: 1, en: 1, cnt_out:  6
Monitor:              1325000 rst_n: 1, en: 1, cnt_out:  7
Monitor:              1425000 rst_n: 1, en: 1, cnt_out:  8
Monitor:              1525000 rst_n: 1, en: 1, cnt_out:  9
Monitor:              1625000 rst_n: 1, en: 1, cnt_out: 10
Disable counter
Monitor:              1700000 rst_n: 1, en: 0, cnt_out: 10
```

NOTE:

- Unlike programming language execution, a simulation in Verilog has factor time. It reflects the function of a hardware in physical world.
- The location of "Enable counter" and "Disable counter" messages seem not to be in correct time. We expect the message "Enable counter" should be displayed before the first "Monitor" message and "Disable counter" is after the last "Monitor" message. This is called race condition, we will learn about it later.

ACTIVITY 4.3:

- Can you change testbench to make the log file writes out all 16 values of the counter?

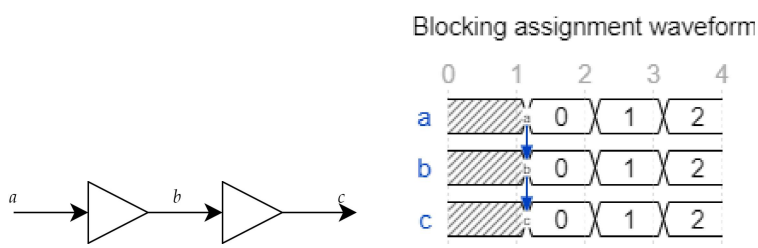
- Can you change this counter and testbench to 16 bits?
- **(HARD)** Can you do the same thing, but use 4 instances of counter 4 bits?

4.6. Blocking & nonblocking assignment

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

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

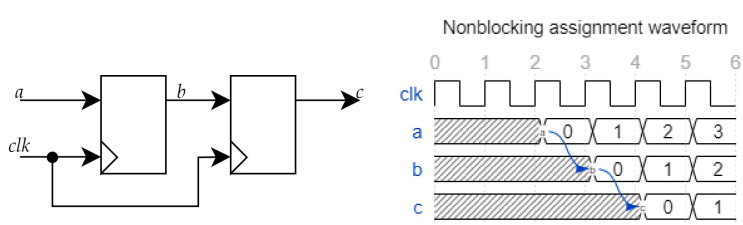
In the example above, the assignment blocks each other, which mean when a changes value, b will change first and c change later. This results in c is captured the latest value of a. The above example can be translated into following schematic and waveform.




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

In the example above, when there is a positive edge of `clk`, both b and c update data, where b captures the current value of a and c captures the current value of b. This mean c only capture the value of a after 2 positive edges of `clk`. The example above can be translated into following schematic and waveform.

The "Waveform" section has tools to analyze the waveform like zoom in/out, change radix, etc.



ATTENTION: RACING CONDITION

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.

For example, there are 2 assignments

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

In above assignment, when a changes value, the value of c is unknown, it can be either the old value of b or the new one.

Another example can be found at Example log file, where the `$display` command executes at the same times of `$monitor` command, thus the log file seems not to be in correct order.

Racing condition also happens when you assign 1 signal in 2 locations at the same time.

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

In above code, value of b at time 0 is unpredictable, it can be either 0 or 1.

If we use nonblocking assignments correctly, the tool can understand they are flipflop and thus there is no such racing happen. Take example of 2 flipflops in previous code snippet, it can be changed into 2 blocks:

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

The code is still converted into 2 serial flipflops and it is predictable.

NOTE: Common guideline to avoid racing condition:

- Do not assign 1 signals at the same time in 2 blocks.
- Do not assign 1 signals in 2 always block.
- Do not mix nonblocking and blocking assignment in 1 block.
- Use nonblocking assignment for flipflop.
- Use blocking assignment for combinational logic.
- In testbench, don't change input signals at clock's active edge, unless the signal represent a flipflop.

4.7. Statement

4.7.1. if...else statement

Syntax `if`:

```
if (<condition>)
 <statement>
```

Syntax `if...else`:

```
if (<condition>)
  <statement 1>
else
  <statement 2>
```

Syntax nested `if`:

```
if (<condition 2>)
  <statement 1>
else if (<condition 2>)
  <statement 2>
...
else
  <statement n>
```

Example code for decoder & encoder blocks using `if...else`

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

  always @(in) begin
    if (in==2'b00) out = 4'b0001;
    else if (in==2'b01) out = 4'b0010;
    else if (in==2'b10) out = 4'b0100;
    else out = 4'b1000;
  end
endmodule
```

```
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
```

> **NOTE:** Encoder code does not have last `else` state, but it still is a combinational logic (not latch). The reason is we have assigned default value of `out` and `hit` as 0 before the `if...else` statement.

4.7.2. case statement

Syntax:

```
case (<signal>)
  <condition 1> : <statement 1>
  <condition 2> : <statement 2>
  ...
  default : <statement 0>
endcase
```

Example code for decoder block using `case`

```
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



module encoder (
    input  wire [4:0] in
  , output reg  [1:0] out
  , output reg        hit
);

  always @(in) begin
    case (in)
      4'b0001 : begin
        out = 2'd0;
        hit = 1'b1;
      end
      4'b0010 : begin
        out = 2'd1;
        hit = 1'b1;
      end
      4'b0100 : begin
        out = 2'd2;
        hit = 1'b1;
      end
      4'b1000 : begin
        out = 2'd3;
        hit = 1'b1;
      end
      default : begin
        out = 2'd0;
        hit = 1'b0;
      end
    endcase
  end

endmodule
```

4.7.3. for loop statement

Syntax:

```
for (<initial assignment>; <expression>; <step>)
```

Example code for the decoder testbench:

```
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
```

ACTIVITY 4.4:

- Can you write testbench for encoder?
- Can you design a priority encoder? The working condition is slightly different with an encoder.
  - The output shows the lowest index that is 1.
  - Example: out = 2'd0, hit = 1'b1 if in = 4'b0001 or 4'b0011 or 4'b1001, etc. In this case in[0] is 1 so out is always 2'd0.

If using `case` statement for priority encoder, this syntax is useful:

```
casez (in):
  4'b???1 : // Run if bit [0] = 1, ignore other bits
  4'b??10 : // Run if bit [1:0] = 2'b10, ignore other bits
...
```

(Note: `casez` and `casex` are not really recommended in RTL code, we should use other method such as using if clause, for loop or function that is introduced later)

4.8. Verification useful syntax

4.8.1. repeat, wait

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

Syntax:

```
repeat (<number>) <statement>
wait (<expression>) <statement>
```

Encoder testbench can be changed to following code to serve the same function:

```
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
```

4.8.2. System tasks & functions

Some useful tasks and functions for testbench:

- `$display` and `$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.

4.8.3. Procedural assignment group

There are 2 types:

- `begin...end` Sequential block
- `fork...join` Parallel block

Sequential block executes command serially.

```
// 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
```

```
run.csh sequential
```

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

Parallel block executes all assignment at once.

```
// 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
```

```
run.csh parallel
```

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

4.8.4. Procedural timing control

Timing for procedural blocks is introduced across the document.

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

Delay control delay to execute an assigment by a specific simulation time

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

Intra-assigment timing control evaluates the assignment first, but the value is assigned after the specified delay time.

```
<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
```

In above examples, command `a= #5ps 0;` runs immediately after the waiting time `10ps`. Thus, the command `b = 1;` runs right at that time. However, the value `0` is only assigned to signal `a` after `5ps` more.

Edge-sensitive event controls are normally used in `always` block.

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

Level-sensitive event controls is used to hold execution until a condition is true.

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

Please read previous code snippets for more examples of the syntaxes.

4.9. Directive & parameter

4.9.1 Directive

Directives are guidance for tool to compile the source code. Some of the most used directives are introduced below.

```
`define <MACRO>
```

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

`define directive create a macro that can be used across the design. For example:

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

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

The result can be like below

```
vector.display_data [INFO]                    1200 tx.sv          40: Test mess
```

`ifdef includes a line of code if a macro is defined. Example

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

The macro can be defined in verilog code using `define or in the compiler command using option +define+<MACRO>

4.9.2. Parameter

There are designs with different configuration but have similar functions, we can use the same code with parameters instead of creating multiple modules.

For example, we can create counters with different bits.

```
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
```

The module's instances can be different with different parameter values.

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

The function can be changed with parameters too.

```
module counter_param #(
    pCOUNTER_WIDTH = 4; // Default value of the counter
  , pCOUNT_UP = 1; // 1 - count up, 0 - count down
) (
    input  wire                      clk
  , input  wire                      rst_n
  , input  wire                      cnt_en
  , output reg  [pCOUNTER_WIDTH-1:0] cnt_out
);

  generate
    if (pCOUNT_UP) begin : count_up
      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
    end else begin : count_down
      always @(posedge clk or negedge rst_n) begin
        if (!rst_n) cnt_out <= {pCOUNTER_WIDTH{1'b1}};
        else if (cnt_en) cnt_out <= cnt_out - 1'b1;
      end
    end
  endgenerate
endmodule
```

NOTE:

- Above code uses `generate` keywords to select code block to use. `generate` block can be used to create multiple code blocks with `for` loop. Check reference for further explanation.
- There are cases we use `parameter` as local, mainly to maintain and fix the code easier.

4.10. Task and function

4.10.1. Task

The same as in any programming language, a task does a common set of steps that other parts of the code can call it.

Example of the decoder testbench:

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

  int idx;
  int error_tol;
  int error_tmp;

  initial begin
    $display("Initial the design");
    $monitor("Monitor: %t in: %d, out: %b", $realtime, in, out);
    #5ns in = 2'd0;

    error_tol = 0;
    for (idx=0; idx<4; idx=idx+1) begin
      compare(.drv_in(idx), .exp_out(1<<idx), .error(error_tmp));
      error_tol = error_tol + error_tmp;
    end
    $display("Reach final value");

    if (error_tol) $display("FAIL: error count %d", error_tol);
    else ("PASS");
  end

  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

endmodule
```

4.10.2. Function

Eventhough the syntaxes between 2 types are comparable, `function` is totally different with `task`. Differences in syntax view:

- 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

In terms of code replacement, we can understand `function` is replacement for a combinational logic (can be used in RTL code), while `task` replaces for a routine (like in testbench).

Example below code:

```
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];
```

Can be replaced with function:

```
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
```

Example code, priority encoder

```
module priority_encoder (
    input  wire [4:0] in
  , output wire [1:0] out
  , output wire       hit
);

  assign {hit, out} = pe(.in(in));

  function [2:0] pe; // MSB is "hit"
    input [4:0] in;
    integer idx;
    begin
      pe = 3'b0;
      for (idx=0;idx<5;idx=idx+1) begin
        if ((in[idx]==1)&&(pe[2]==0)) begin // Only true in first "hit"
          pe[2] = 1'b1;
          pe[1:0] = idx;
        end
      end

      // By the end of above loop, pe[2] contain "hit" information
      // and p[1:0] contain the index of first "hit"
    end
  endfunction
endmodule
```

4.11. FSM coding

4.11.1. 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

Some examples of state machine:

- A Safe:
  - State: Locked, Unlocked
  - Transition:
    - Correct combinations move the safe from Locked to Unlocked.
    - While being Locked, wrong combinational reset the whole unlocking process and return to the initial Locked.
- Traffic lights:
  - State: Green, Yellow, Red
  - Transition: States change after a given time.

4.11.2. Types of FSM: Mealy & Moore

A Moore state machine is classified as an FSM where the outputs are only a function of the present state, while Mealy state machines have one or more outputs that are a function of the present state and one or more FSM inputs.

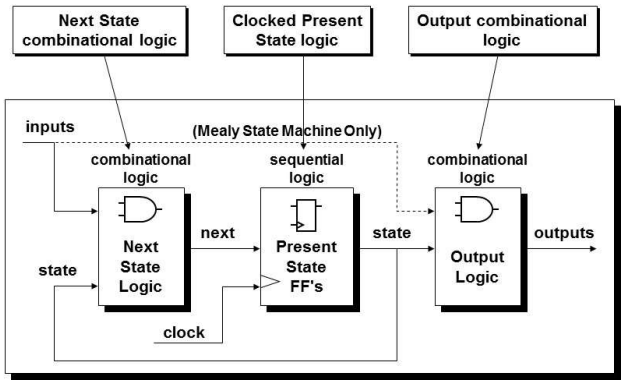


Figure credit: Clifford E. Cummings, posted in SNUG

> **NOTE:** Common guideline to design FSM:
> Always use Moore version unless Mealy is unavoidable. Mealy depends on inputs to drive outputs, which mean the timing optimization on the outputs depends on the inputs too. While Moore's version only depends on internal logic and therefore easier to control.
> Timing optimization is out of this document's scope to explain further.

There are many styles to write code for FSM, followed is just 1 method, readers can apply any style as they feel it suits their needs the best (coding management, performance, area of result netlist).

Further reading about FSM coding style and results: http://www.sunburst-design.com/papers/CummingsSNUG2019SV_FSM1.pdf

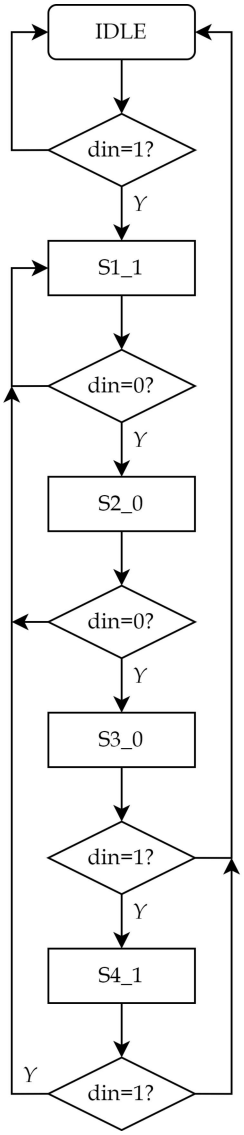
4.11.3. Example: Binary bit stream

Design a binary bit stream with following interface.

Port	Size	Direction	Description
clk	1	input	Clock
rst_n	1	input	Asynchronous reset, active low
din	1	input	Data in
lock	1	output	Lock signal, indicates the data stream on din match defined sequence

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.

From the sequence, we can easily design the state machine should have at least 4 stages, each stage represent 1 matching bit, namely `S1_1`, `S2_0`, `S3_0`, `S4_1`. We also need a stage when the first bit does not match, this stage is also status of the machine in reset mode, we can name it `IDLE`. The condition to change stage is shown in the below figure.

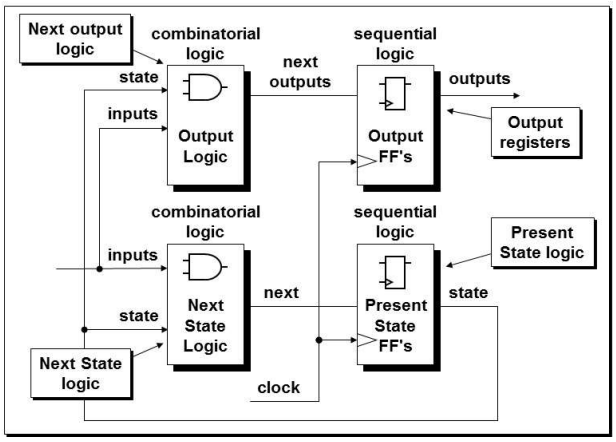


The signal `lock` keeps at 0 until stage `S4_1`. After that, it keeps at 1 if the stream continues to be corrected.

```
module bit_streaming (
    input  wire clk
  , input  wire rst_n
  , input  wire din
  , output wire lock
);

  localparam pSTS_WIDTH = 3;
  localparam pSTS_IDLE = 3'd0,
             pSTS_S1_1 = 3'd1,
             pSTS_S2_0 = 3'd2,
             pSTS_S3_0 = 3'd3,
             pSTS_S4_1 = 3'd4;

  reg [pSTS_WIDTH-1:0] current_state, next_state;
  reg current_lock, next_lock;

  assign lock = current_lock;

  // Register part of FSM
  always @(posedge clk or negedge rst_n) begin // fsm_ff
    if (!rst_n) begin
      current_state <= pSTS_IDLE;
      current_lock <= 1'b0;
    end else begin
      current_state <= next_state;
      current_lock <= next_lock;
    end
  end // fsm_ff

  // Combinational part of FSM
  // * mean any signals change will activate this block
  always (*) begin // fsm_comb
    // Assign default value
    next_state = pSTS_IDLE;
    next_lock = 1'b0;

    case (current_state)
      pSTS_IDLE : begin
        if (din) begin
          next_state = pSTS_S1_1;
        end
      end
      pSTS_S1_1 : begin
        // If the flow is correct, jump to next state
        // The lock signal keep the same as before
        // Otherwise the state reset and lock signal deasserts
        if (!din) begin
          next_state = pSTS_S2_0;
          next_lock = current_lock;
        end else begin
          next_state = pSTS_S1_1;
```

```
            next_lock = 1'b0;
          end
        end
        pSTS_S2_0 : begin
          if (!din) begin
            next_state = pSTS_S3_0;
            next_lock = current_lock;
          end else begin
            next_state = pSTS_S1_1;
            next_lock = 1'b0;
          end
        end
        pSTS_S3_0 : begin
          if (din) begin
            next_state = pSTS_S4_1;
            next_lock = 1'b1;
          end else begin
            next_state = pSTS_S1_1;
            next_lock = 1'b0;
          end
        end
        pSTS_S4_1 : begin
          if (din) begin
            next_state = pSTS_S1_1;
            next_lock = current_lock;
          end else begin
            next_state = pSTS_IDLE;
            next_lock = 1'b0;
          end
        end
      endcase
    end // fsm_comb
  endmodule
```

The hardware structure for above coding style can be understand as the figure below.



ACTIVITY 4.5:

- Can you write testbench to create a bit stream and monitor the lock? The testbench should know when the lock signal should assert and deassert.
- Can you fix the design, if it is required the bit stream must be correct 4 times before `lock` asserts to 1?
- The state machine is encoded with binary code. Can you change it to one hot and gray code? What is the pros and cons of each encoding style?

4.12. SystemVerilog

While there are some problems when develop RTL as well as testbench, SystemVerilog is created to enhance the Verilog language. The new language inherits all syntaxes from Verilog, thus there is no problem when moving to this language.

The following syntaxes are recommended to use as replacement for Verilog.

ACTIVITY 4.6: Can you replace all previous code to the SystemVerilog syntaxes?

4.12.1. always_ff, always_comb, always_latch

As described, `always` block can be used to model a flipflop, latch or a combinational logic. SystemVerilog adds 3 new syntaxes for them.

`always_ff` is used for flipflop. The syntax is similar to the old one.

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

`always_latch` is used for latch, we don't have to define condition for it.

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

`always_comb` is used for combinational logic, we don't have to define condition for it either

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

These syntaxes help to eliminate common RTL coding mistakes when writing code. For example `always_latch` syntax *recommends* EDA tools to check if the code represent a latch or not; thus, engineer *may* recevie messages if the block does not work as intended.

4.12.2. Data type `logic`

`logic` is a datatype that can totally replace `reg` and mostly replace `wire`. Engineer don't have to think what type to declare a signal any more, all can be `logic` type (except some rare cases for `wire`).
This syntax ensures we don't drive the same signal at a time from multiple source and create hard debugged problem.

For example, below code will create an error.

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

Even though both `initial` blocks assign the same value to `a`, they drive that signal at the same time and create error. This also mean we will not be able to connect the same `logic` signal into 2 outputs (which is not valid in digital design either).

The rare case is we can't assign `logic` to 2 inout ports, this case is a valid usage but since the ports are inout, there can be cases that both sides are output and violate `logic` rule. In this case we have to use `wire`. (Review section 4.2. Port for inout port)

4.12.3. Assigning constant value

To assign all bits a constant 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

```
logic [2:0] a;

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

4.12.4. Packed array

Verilog does support (unpacked) array with syntax:

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

The unpacked array behaves different with bus (different syntaxes, operations). There are many cases engineer want to work with a so call multi-dimentions bus than an array. Packed array in SystemVerilog can serve this purpose.

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

initial begin
  b = 24'ha_b_c_d_e_f; //b[5]=4'ha, b[4]=4'hb, etc.
  b[5] = 4'h0; // b = 24'h0_b_c_d_e_f
  b = (b<<1); // b = 24'hb_c_d_e_f_0
end
```

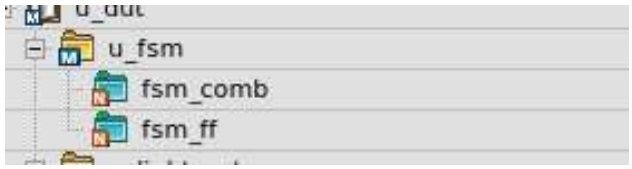
4.12.5. Naming block

Code blocks now can have name for engineer to trace back the code easier as well as using EDA tool better, while they supports to trace code with block name.

The block name can be used in `module`, `task`, `function`, `begin...end`, etc.

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

task tsk_name();
...
endtask : tsk_name

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

Using EDA tool, we can trace code like below



4.12.6. Data type enumerations

Enumeration data type is very useful in FSM, where state can be referred by name instead of number.

```
enum integer {IDLE=0, GNT0=1, GNT1=2} state; // define base type as integer, an
enum integer {RED,GREEN,ORANGE} color; // Automatically assign value. RED=0, GR
enum integer {BRONZE=4, SILVER, GOLD} medal; // The first value is 4, thus the
```

Like C programming language, enumeration has methods (e.g. `medal.name()` returns the current name of the variable). Enumeration is better parameter in FSM because:

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

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

In above example, there will be no error if we define 4 parameters and assign `medal` to them. This leads to unexpected behavior if we accidently assign `medal` to `GOLDEN`. Moreover, `enum` makes sure there are no 2 names have the same value, which prevent incorrect function. This can't be prevented if we use `parameter`

In debugging tool, enum is shown by its' name as default. While for parameter, it is shown by value.

4.12.7. Data type `struct`

`struct` is the same as C programming language, with exception that a packed struct "compresses" all components together, thus there is no memory gap and it works just like a bus.

```
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
```

This syntax is very useful to work with signal in FSM. With it, we don't have to define all signals twice for current state and next state.

```
// 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;
```

> **NOTE:** `struct` data type also have unpacked version, by not defining the keyword `packed`.

4.12.8. Enhanced for task and function

`automatic` is keyword to make `task` & `function` get dynamic memory. In Verilog, they use static memory and cause troubles (please check reference sources for further explanation)

Engineer now doesn't have to add `begin..end` in `task` & `function` anymore.

`struct` can be used for `function`, which solve the limitation that there is only 1 output for `function`.

`function` now has `return` keyword to escape it before the last line.

The priority encoder function can be changed to below code.

```
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
```

5. Guide project

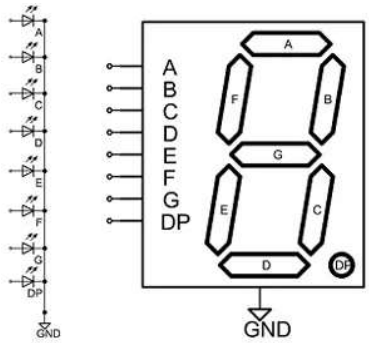
Design a traffic light controller using FSM and following specification:

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

To simplify the project, only control 1 direction of the road.

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

The 7 segments led display code can be found in the image below.



The bit mapping of signal `display_led` to the segments is: {A, B, C, D, E, F, G, DP}
(This mean MSB is segment A and LSB is DP)

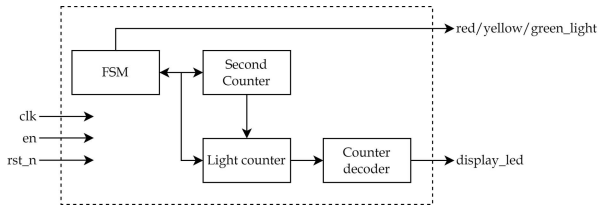
NOTE: Before reading further, readers are encouraged to think about solution first.

5.1. Brief designing the blocks

From the problem, we will need following blocks:

- FSM to control state of the design. We need at least 3 states RED, YELLOW, GREEN. We also need IDLE state when the system is not enabled.
- Because clock frequency is 10MHz, we need 1 counter to count the clock cycles and notify when 1 second is reached.
- We also need another counter to count the number of seconds has passed to switch lights.
- A 7 segment decoder to convert the value on the counter to leds.

An overall block diagram can be found below



5.2. Detail designing the blocks

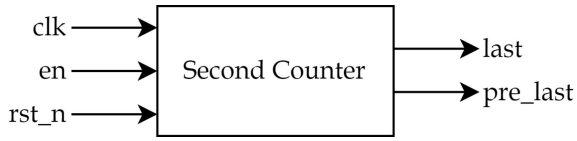
From the previous figure, we can go further to design the blocks' functions and interfaces.

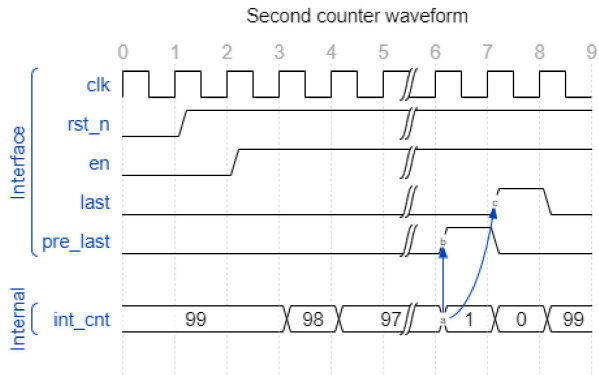
5.2.1. Second clock counter

From the `clk` signal's frequency, we understand 1 second is equal to 100 cycles of this clock, thus this clock counter needs to count from 99 to 0 then reset automatically.

At every last value, it sends out signal `last` to indicates one second has passed. It also send out `pre_last` that is 1 cycle sooner than `last`, this signal help the FSM to catch up with the last cycle (because the FSM has register on the output)

The interface diagrams and waveform can be found in below figures





From the interface and waveform, we can write the code as below.

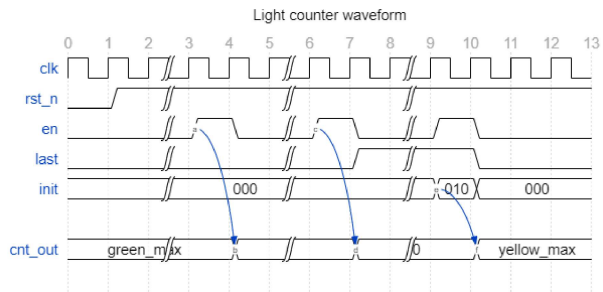
```
module second_cnt # (
  parameter pINIT_VAL = 99
) (
  input  logic clk
  , input  logic rst_n
  , input  logic en
  , output logic cnt_last
  , output logic pre_last
);

  `ifndef SYNTHESIS
    timeunit 1ps;
    timeprecision 1ps;
  `endif

  //--------------------
  // LOCAL PARAMETERS
  //--------------------
  // Below command make sure pCNT_WIDTH is always enough to
  // reach value of pINIT_VAL
  localparam pCNT_WIDTH = $clog2(pINIT_VAL+1);
  // Alternative: We can write hard code value
  //              But this value needs to be changed if pINIT_VAL changes
  // localparam pCNT_WIDTH = 7; // Max value is 127 > pINIT_VAL

  //--------------------
  // VARIABLES
  //--------------------
  logic [pCNT_WIDTH-1:0] int_cnt;

  //--------------------
  // FUNCITON
  //--------------------
  always_ff @(posedge clk or negedge rst_n) begin : cnt_proc
    if (!rst_n) int_cnt <= pINIT_VAL;
    else if (en) begin
      if (|int_cnt) int_cnt <= int_cnt - 1'b1;
      else int_cnt <= pINIT_VAL;
    end else int_cnt <= pINIT_VAL;
  end : cnt_proc

  assign pre_last = (int_cnt==1'b1) ? 1'b1 : 1'b0;
  assign cnt_last = ~(|int_cnt);
endmodule : second_cnt
```

NOTE: The directive `SYNTHESIS` make sure the code does not exist in synthesize step because the keywords `timeunit` and `timeprecision` are only meaningful in simulation step.

5.2.2. Light control counter

This light control need to be initialized with different value (red, green, blue), so it has `init` port with bus size is 3, each bit is for 1 light. This counter only counts down in each second (not each clock), it is enabled by `last` value of the *second counter*.

The interface diagrams and waveform can be found in below figures



In this design, `init` has higher priority than `enable` so it won't count down if `init` is not zero.

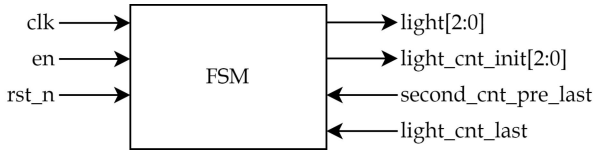
From the interface and waveform, we can write the code as below.

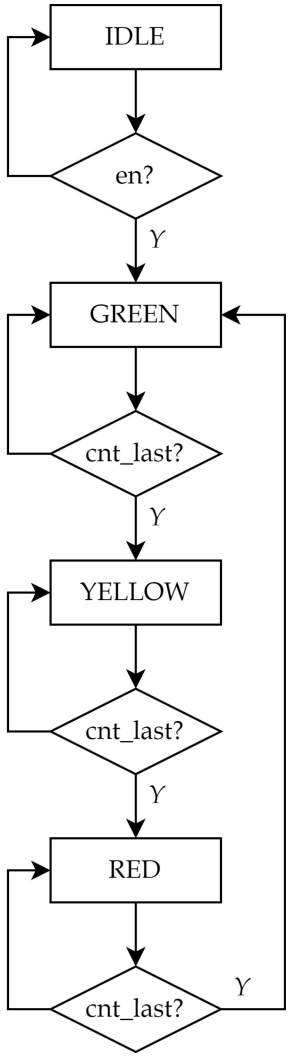
```
module light_cnt # (
    parameter pGREEN_INIT_VAL  = 14
  , parameter pYELLOW_INIT_VAL = 2
  , parameter pRED_INIT_VAL    = 17
  , parameter pCNT_WIDTH       = $clog2(pRED_INIT_VAL+1)
  , parameter pINIT_WIDTH      = 3
) (
    input  logic                   clk
  , input  logic                   rst_n
  , input  logic                   en
  , input  logic [pINIT_WIDTH-1:0] init
  , output logic                   cnt_last
  , output logic [pCNT_WIDTH -1:0] cnt_out
);

  `ifndef SYNTHESIS
    timeunit 1ps;
    timeprecision 1ps;
  `endif

  //--------------------
  // LOCAL PARAMETERS
  //--------------------
  // This index must be the same between FSM and this counter
  // We can do so by using "package" syntax, which is not in this document's sc
  localparam pGREEN_IDX  = 0,
             pYELLOW_IDX = 1,
             pRED_IDX    = 2;

  //--------------------
  // VARIABLES
  //--------------------

  //--------------------
  // FUNCITON
  //--------------------
  always_ff @(posedge clk or negedge rst_n) begin : cnt_proc
    if (!rst_n) cnt_out <= pGREEN_INIT_VAL;
    else if (init[pGREEN_IDX]) cnt_out <= pGREEN_INIT_VAL;
    else if (init[pYELLOW_IDX]) cnt_out <= pYELLOW_INIT_VAL;
    else if (init[pRED_IDX]) cnt_out <= pRED_INIT_VAL;
    else if (en) cnt_out <= cnt_out - 1'b1;
  end : cnt_proc

  assign cnt_last = ~(|cnt_out);
endmodule : light_cnt
```

5.2.3. FSM block

This block connects with the 2 mentioned counters and control the red/yello/green lights. The indexes of the light control signals must match with the indexes of the light counter block.

The interface and state of this block is shown in below figures.





`cnt_last` signal in the diagram above is a function of `second_cnt_pre_last` and `light_cnt_last`

NOTE: Eventhough not shown, the state will return to `IDLE` if the en signal is zero.

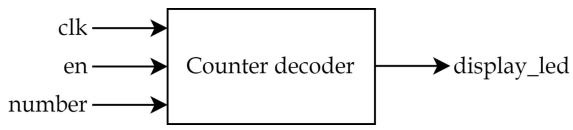
```
module fsm #(
  parameter pLIGHT_WIDTH = 3
) (
    input  logic                    clk
  , input  logic                    rst_n
  , input  logic                    en

  // Counter connection
  , input  logic                    second_cnt_pre_last
  , input  logic                    light_cnt_last

  , output logic [pLIGHT_WIDTH-1:0] light
  , output logic [pLIGHT_WIDTH-1:0] light_cnt_init
);

  `ifndef SYNTHESIS
    timeunit 1ps;
    timeprecision 1ps;
  `endif

  //--------------------
  // LOCAL PARAMETERS
  //--------------------
  // This index must be the same between FSM and light counter blocks
  // We can do so by using "package" syntax, which is not in this document's sc
  localparam pGREEN_IDX  = 0,
             pYELLOW_IDX = 1,
             pRED_IDX    = 2;

  localparam pST_WIDTH = 2;

  //--------------------
  // VARIABLES
  //--------------------
  typedef enum logic [pST_WIDTH-1:0] {
    IDLE
    , GREEN
    , YELLOW
    , RED
  } st_t;
  st_t current_state, next_state;

  typedef struct packed {
    logic [pLIGHT_WIDTH-1:0] light;
    logic [pLIGHT_WIDTH-1:0] light_cnt_init;
  } sig_t;
  sig_t current_sig, next_sig;

  logic cnt_last;
  //--------------------
  // FUNCITON
  //--------------------
```

```
assign cnt_last = light_cnt_last & second_cnt_pre_last;
assign light = current_sig.light;
assign light_cnt_init = current_sig.light_cnt_init;
always_ff @(posedge clk or negedge rst_n) begin : fsm_ff
  if (!rst_n) begin
    current_state <= IDLE;
    current_sig <= '{default:0};
  end else if (en) begin
    current_state <= next_state;
    current_sig <= next_sig;
  end else begin
    current_state <= IDLE;
    current_sig <= '{default:0};
  end
end :fsm_ff

always_comb begin : fsm_comb
  next_state = IDLE;
  next_sig = '{default:0};
  case (current_state)
    IDLE : begin
      next_state = GREEN;
      next_sig.light[pGREEN_IDX] = 1'b1;
    end
    GREEN : begin
      if (cnt_last) begin
        next_state = YELLOW;
        next_sig.light_cnt_init[pYELLOW_IDX] = 1'b1;
        next_sig.light[pYELLOW_IDX] = 1'b1;
      end else begin
        next_state = GREEN;
        next_sig.light[pGREEN_IDX] = 1'b1;
      end
    end
    YELLOW : begin
      if (cnt_last) begin
        next_state = RED;
        next_sig.light_cnt_init[pRED_IDX] = 1'b1;
        next_sig.light[pRED_IDX] = 1'b1;
      end else begin
        next_state = YELLOW;
        next_sig.light[pYELLOW_IDX] = 1'b1;
      end
    end
    RED : begin
      if (cnt_last) begin
        next_state = GREEN;
        next_sig.light_cnt_init[pGREEN_IDX] = 1'b1;
        next_sig.light[pGREEN_IDX] = 1'b1;
      end else begin
        next_state = RED;
        next_sig.light[pRED_IDX] = 1'b1;
```

```
            next_sig.light[pRED_IDX] = 1'b1;
          end
        end
      endcase
      // There is no "default" case because the default values are assigned
      // in the first 2 lines
    end : fsm_comb
  endmodule : fsm
```

5.2.4. Counter decoder

The counter decoder doesn't have any register. It only convert value from light counter block to 7-segments LEDs.



ACTIVITY 5.1: Can you write code for this block?

5.2.5. Connect altogether

From the overall diagram and block designs, we connect all block together as following code.

```
module traffic_light # (
    parameter pSECOND_CNT_INIT_VAL = 99
  , parameter pGREEN_INIT_VAL      = 14
  , parameter pYELLOW_INIT_VAL     = 2
  , parameter pRED_INIT_VAL        = 17
  , parameter pLED_NO              = 2
  , parameter pLED_WIDTH           = 8

) (
    input  logic                    clk
  , input  logic                    rst_n
  , input  logic                    en

  , output logic                                green_light
  , output logic                                yellow_light
  , output logic                                red_light
  , output logic [pLED_NO-1:0][pLED_WIDTH-1:0] display_led

);
  `ifndef SYNTHESIS
    timeunit 1ps;
    timeprecision 1ps;
  `endif

  //--------------------
  // LOCAL PARAMETERS
  //--------------------
  localparam pLIGHT_WIDTH     = 3;
  localparam pLIGHT_CNT_WIDTH = $clog2(pRED_INIT_VAL+1);

  localparam pGREEN_IDX  = 0,
             pYELLOW_IDX = 1,
             pRED_IDX    = 2;

  //--------------------
  // VARIABLES
  //--------------------
  logic second_cnt_pre_last;
  logic second_cnt_last;
  logic light_cnt_en;
  logic light_cnt_last;
  logic [pLIGHT_WIDTH    -1:0] light;
  logic [pLIGHT_WIDTH    -1:0] light_cnt_init;
  logic [pLIGHT_CNT_WIDTH-1:0] light_cnt_out;

  //--------------------
  // FUNCITON
  //--------------------
  second_cnt # (
    .pINIT_VAL(pSECOND_CNT_INIT_VAL)
  ) u_second_cnt (
      .clk     ( clk                 )
```

```
    , .rst_n   ( rst_n              )
    , .en      ( en                 )
    , .cnt_last( second_cnt_last    )
    , .pre_last( second_cnt_pre_last )
  );
  assign light_cnt_en = en & second_cnt_last;
  light_cnt # (
      .pGREEN_INIT_VAL ( pGREEN_INIT_VAL  )
    , .pYELLOW_INIT_VAL( pYELLOW_INIT_VAL )
    , .pRED_INIT_VAL   ( pRED_INIT_VAL    )
  ) u_light_cnt (
      .clk     ( clk            )
    , .rst_n   ( rst_n          )
    , .en      ( light_cnt_en   )
    , .init    ( light_cnt_init )
    , .cnt_last( light_cnt_last )
    , .cnt_out ( light_cnt_out  )
  );

  light_decode # (
      .pLED_NO      ( pLED_NO )
    , .pNUMBER_WIDTH( pLIGHT_CNT_WIDTH )
  ) u_light_decode (
      .number ( light_cnt_out )
    , .led    ( display_led   )
    , .en   ( en    )
  );

  fsm u_fsm (
      .clk  ( clk   )
    , .rst_n( rst_n )
    , .en   ( en    )

    // Counter connection
    , .second_cnt_pre_last( second_cnt_pre_last )
    , .light_cnt_last     ( light_cnt_last      )

    , .light         ( light          )
    , .light_cnt_init( light_cnt_init )
  );
  assign red_light    = light[pRED_IDX];
  assign yellow_light = light[pYELLOW_IDX];
  assign green_light  = light[pGREEN_IDX];

endmodule : traffic_light
```

5.3. Testbench

The testbench include the design with display to shows the activity of light and led counter.

```
module traffic_light_tb;
  timeunit 1ps;
  timeprecision 1ps;

  //--------------------
  // PARAMETERS
  //--------------------
  parameter pSECOND_CNT_INIT_VAL = 99;
  parameter pGREEN_INIT_VAL      = 14;
  parameter pYELLOW_INIT_VAL     = 2;
  parameter pRED_INIT_VAL        = 17;
  parameter pLED_NO              = 2;
  parameter pLED_WIDTH           = 8;

  //--------------------
  // VARIABLES
  //--------------------
  logic                    clk;
  logic                    rst_n;
  logic                    en;

  logic                              green_light;
  logic                              yellow_light;
  logic                              red_light;
  logic [pLED_NO-1:0][pLED_WIDTH-1:0] display_led;

  logic [pLED_NO-1:0][3:0] led;
  genvar led_idx;

  //--------------------
  // CLOCK & INIT
  //--------------------
  //--- Clock
  parameter CLK_DELAY = 25*1000;
  parameter CLK_WIDTH = 50*1000;
  parameter CLK_PERIOD = 100*1000;
  always begin
    #(CLK_DELAY) clk = 1'b1;
    #(CLK_WIDTH) clk = 1'b0;
    #(CLK_PERIOD-CLK_DELAY-CLK_WIDTH);
  end

  initial begin
    clk = '0;
    rst_n = '0;
    en = '0;

    #(5*CLK_PERIOD) rst_n = '1;
    en = '1;

    #(100000*CLK_PERIOD) $finish;
  end
```

```
  always @(*) begin
    $display("%t RED: %b, YELLOW: %b, GREEN: %b", $realtime, red_light, yellow_
  end
  always @(*) begin
    $display("%t Counter: %d%d", $realtime, led[1], led[0]);
  end

  generate
  for (led_idx=0;led_idx<pLED_NO;led_idx++) begin : led_cvt_proc
    always_comb begin
      case (display_led[led_idx])
        8'b1111_1100: led[led_idx] = 4'd0;
        8'b0110_0000: led[led_idx] = 4'd1;
        8'b1101_1010: led[led_idx] = 4'd2;
        8'b1111_0010: led[led_idx] = 4'd3;
        8'b0110_0110: led[led_idx] = 4'd4;
        8'b1011_0110: led[led_idx] = 4'd5;
        8'b1011_1110: led[led_idx] = 4'd6;
        8'b1110_0000: led[led_idx] = 4'd7;
        8'b1111_1110: led[led_idx] = 4'd8;
        8'b1111_0110: led[led_idx] = 4'd9;
        default: led[led_idx] = 'x;
      endcase
    end
  end
  endgenerate


  //--------------------
  // DUT
  //--------------------
  traffic_light # (
      .pSECOND_CNT_INIT_VAL( pSECOND_CNT_INIT_VAL )
    , .pGREEN_INIT_VAL     ( pGREEN_INIT_VAL      )
    , .pYELLOW_INIT_VAL    ( pYELLOW_INIT_VAL     )
    , .pRED_INIT_VAL       ( pRED_INIT_VAL        )
    , .pLED_NO             ( pLED_NO              )
    , .pLED_WIDTH          ( pLED_WIDTH           )

  ) u_dut (
      .clk  ( clk   )
    , .rst_n( rst_n )
    , .en   ( en    )

    , .green_light ( green_light  )
    , .yellow_light( yellow_light )
    , .red_light   ( red_light    )
    , .display_led ( display_led  )

  );
endmodule
```

```
endmodule
```

In the simulation path, run with commnad

```
.\run.csh traffic_light
```

The result looks like

```
1220425000 Counter:  0 0
1230325000 RED: 0, YELLOW: 1, GREEN: 0
1230425000 Counter:  0 2
1240425000 Counter:  0 1
1250425000 Counter:  0 0
1260325000 RED: 1, YELLOW: 0, GREEN: 0
1260425000 Counter:  1 7
1270425000 Counter:  1 6
1280425000 Counter:  1 5
```

> **ACTIVITY 5.2:** Write code for the other side of traffic light, where 2 sides have different timeset (RED light time in one direction is equal to YELLOW + GREEN on the other side)

> **ACTIVITY 5.3:** A simpler for the blocks light counter + counter decoder is a BCD light counter (counter that counts to 9 per digit). Design and write code that replaces those blocks.

6. Reference

Webpage:

- https://www.asic-world.com/
- http://www.sunburst-design.com/papers/
- https://sutherland-hdl.com/papers.html

Book:

- Verilog HDL: A Guide to Digital Design and Synthesis By Samir Palnitkar
- SystemVerilog for Verification: A Guide to Learning the Testbench Language Features
- SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling

A. Appendix A: Using debugging EDA tool

A.1. Running simulation

To use the simulation and debugging tools, following command load the required modules

```
module load vsc
module load verdi
```

In the simulation package, there is `run.csh` file that helps you to run the sample testbench. For example:

```
./run.csh traffic_light
```

The content of that file can be reduced as below:

```
# Below variables depend on your design
setenv TOP traffic_light
setenv SIM_TB traffic_light_tb
setenv WAVE_FILE traffic_light.fsdb

# Step 1: Compile code to data base
vcs +vcs+lic+wait -full64 -lca -debug_access+all -l compile.log -timescale=1ns/
+vpdports -sverilog \
-kdb -debug_acc+all \
+lint=all \
-f filelist \
$TOP\_tb.sv \
-top $TOP\_tb

# Step 2: Run simulation
simv -ucli -i wave.tcl -l sim.log
```

In step 1 above, there are 3 options we want to check:

```
-f filelist
```

This option define a file that collects all required files for compliation. Current package is all RTL codes.

```
$TOP\_tb.sv
```

This option calls any file directly for compliation, in this case it call the testbench.

```
-top $TOP\_tb
```

This option defines the top module to do simulation, in this example it is the testbench.

Other options can be keep the same, you can investiage more with command:

```
vcs -help
```

The environment variables are required because in step 2, we use the file `wave.tcl` to create waveform for further debuggin. In that file, we define the scope for the tool to generate waveform.

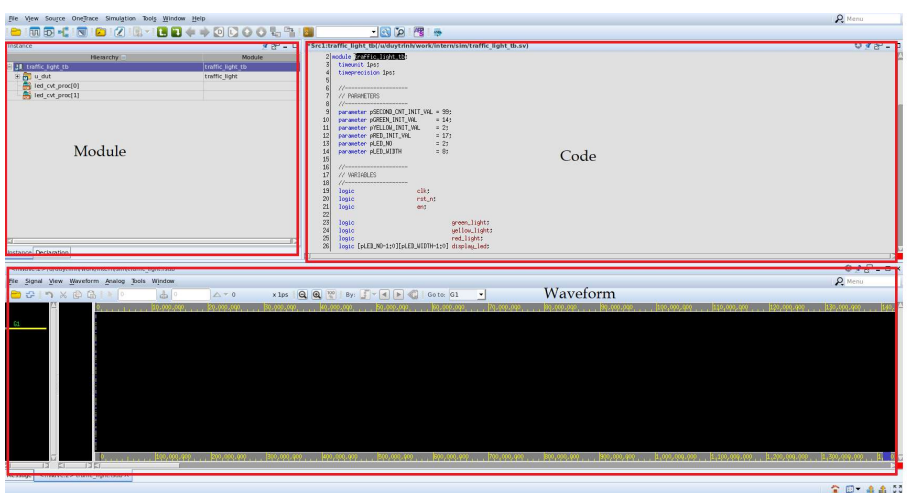
A.2. Running simulation

With the environment variables, waveform from above example is `traffic_light.fsdb`. To open the debugging tool, use command:

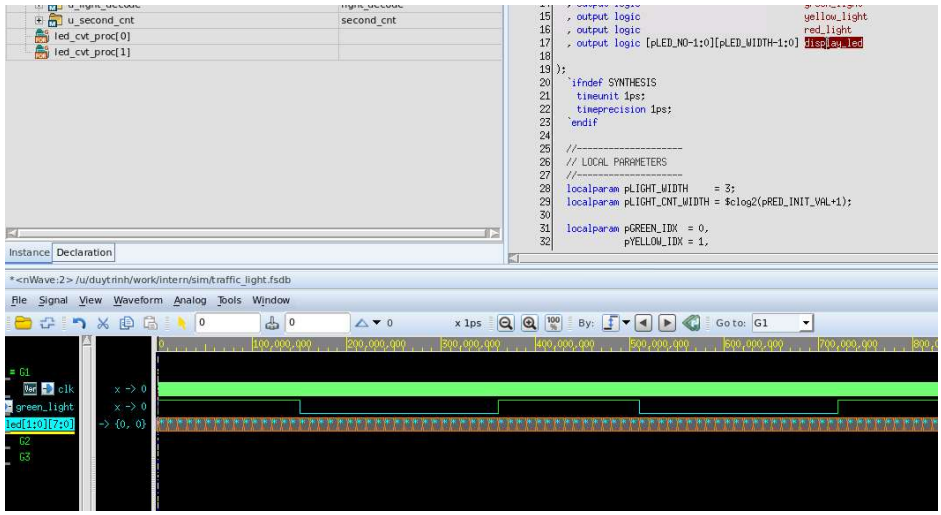
```
verdi
```

On the tool, choose `File->Open Waveform file` and select the waveform.

The layout is divided into 3 sections: Module/instance, Code of the selected instance and Waveform for the signal.



To probe a signal, double click on an instance in "Module" and drag the signal from "Code" to "Waveform"



The "Waveform" section has tools to analyze the waveform like zoom in/out, change radix, etc.

B. Appendix B: Ending course activity

Student can choose 1 (or many) of below activities to work and present for the inteview by the end of this training course. In each activities:

- Draw diagram to design the functionality of the block.
- Write RTL code for the block.
- Write testbench to test the block.
- Capture waveform.
- FSM is not a must.

B.1. Signal toggling detector

Design a block that detect the changing in value of an input:

Port	Size	Direction	Description
clk	1	input	50MHz Clock
rst_n	1	input	Asynchronous reset
din	1	input	Enable
redge	1	output	Assert 1 clk edge if din change from 0 to 1
fedge	1	output	Assert 1 clk edge if din change from 1 to 0

B.2. Hello world display

Make a set of eight 7-segments led to scrool "HELLO " string (3 space after the HELLO string) with specification:

Port	Size	Direction	Description
clk	1	input	50MHz clock
rst_n	1	input	Asynchronous reset
en	1	input	Enable
L7	8	output	Control LED7
L6	8	output	Control LED6
L5	8	output	Control LED5
L4	8	output	Control LED4
L3	8	output	Control LED3
L2	8	output	Control LED2
L1	8	output	Control LED1
L0	8	output	Control LED0

If en=1, the set scroll from right to left in each 1ms. See sample (character "_" is replaced by white spacing):

Cycle	Display
0	HELLO___
1	ELLO___H
2	LLO___HE
3	LO___HEL
4	O___HELL
5	___HELLO

B.3. Bit streaming

Make a bit streaming checker with specification:

Port	Size	Direction	Description
clk	1	input	50MHz Clock
rst_n	1	input	Asynchronous reset
en	1	input	Enable

Port	Size	Direction	Description
din	1	input	Data in
lock	1	output	Active if there is "Lock" event
value	1	output	The value of "Lock" event

A "Lock" event occurs when the block has found din is all "1" or all "0" in 5 consecutive edges. *value* signal indicates the lock is all "1" or all "0".

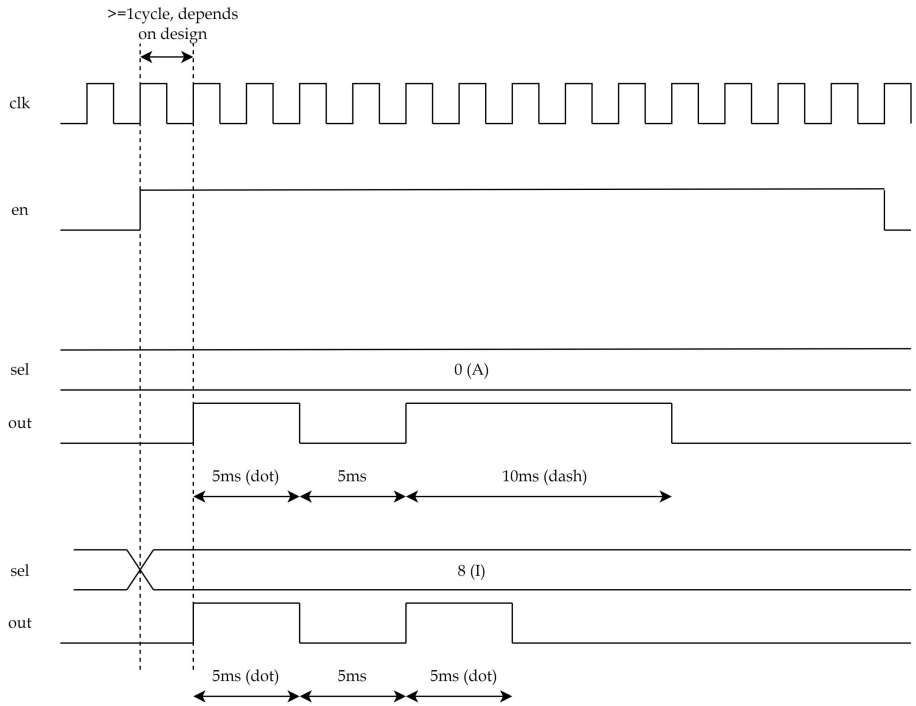
B.4. Morse code

Make a Morse code display with specification:

Port	Size	Direction	Description
clk	1	input	50MHz Clock
rst_n	1	input	Asynchronous reset
en	1	input	Enable
sel	5	input	Character select
out	1	output	Morse code display

When *en* asserts, the signal *out* display the Morse code of the character selected by *sel* (0b00000 for A, 0b00001 for B, etc.). The Morse code is displayed as following format:

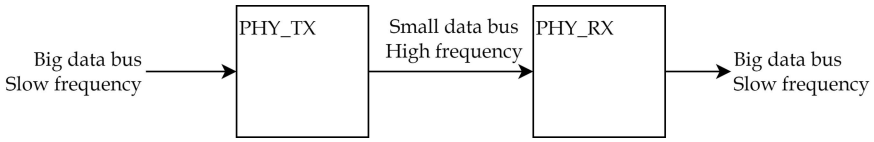
- Pulse: 0.5ms if it is a dot, 1ms if it is a dash.
- 0.5ms wait time between 2 pulses.



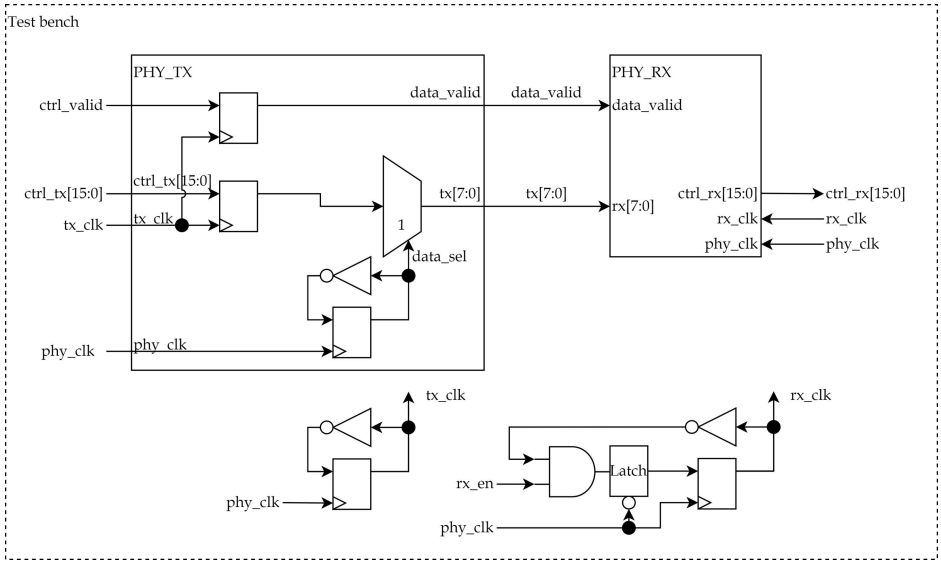
B.5. Data alignment

NOTE This assignment and below are only for students who have finished all others.

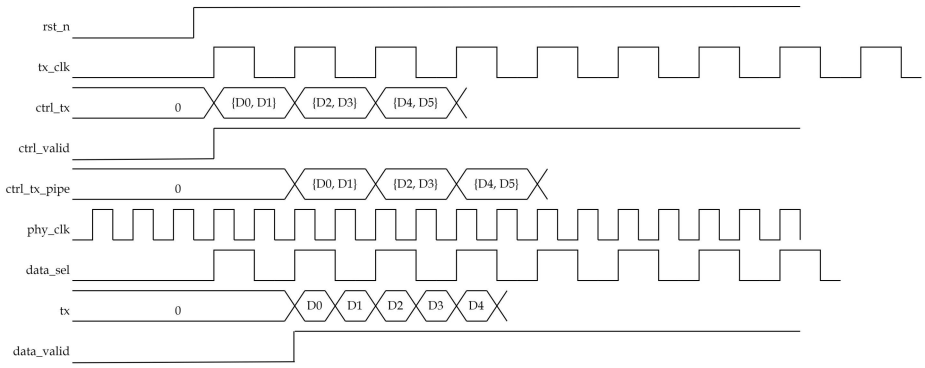
A PHY is a bridge to send data from one side to other one, sometimes they are in different die or chip. In those cases, there are times that the controller can't run with high speed as the PHY.



Below figure shows a sample case of testbench to connect the 2 PHYs, where the clock ratio is 2:1

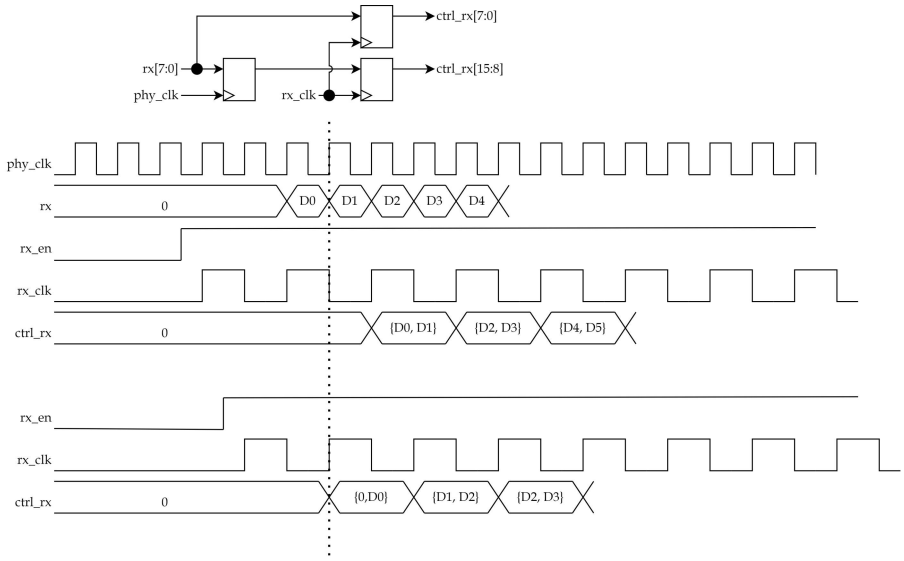


In the figure above, the controller has bigger data bus than the PHY (2X) to keep up with the clock speed of the PHY. The TX has a MUX to select which data segment to send to RX side.



*TX doesn't have to send valid data (D0, D1, D2, etc.) right after reset. It can wait for RX to be stable (after rx_en = 1), then send valid data. Before that, it keep the bus 0).
The example is used to shorten the waveform.

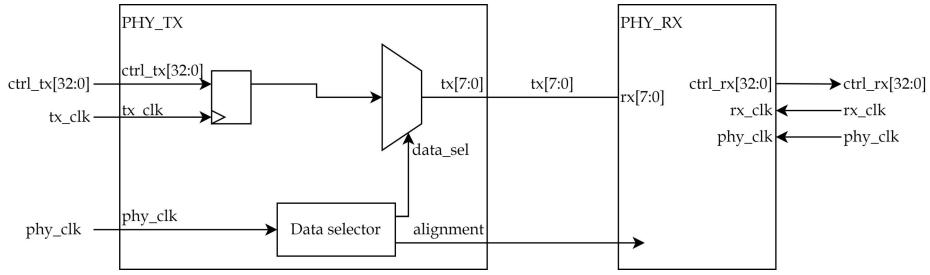
But how do we implement the RX? The problem is in the RX side, we don't know the clock phase to enable RX clock. Example below is a wrong implementation, it will causes issue if *rx_en* turns on in incorrectly phase.



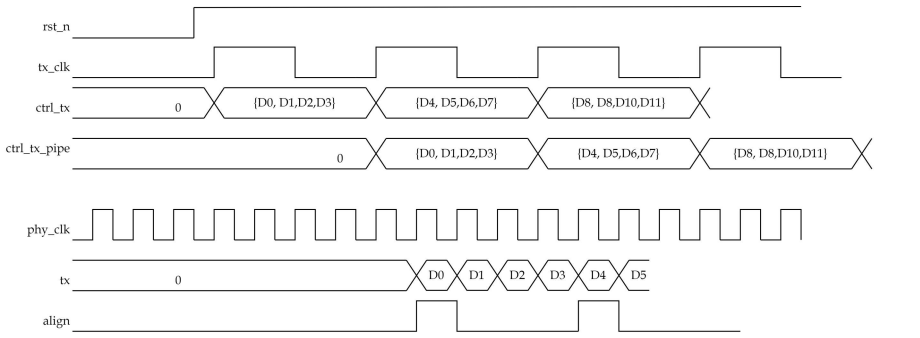
Design & write testbench for the RX block.

B.6. Data aligment with additional signal

One design solution is using an "align" signal to mark the data bursts. Below figure shows a 4:1 system with alignment signal.



The TX must mark the first segment with 1, and the others with 0. RX side bases on that signal to align the correct data regardless the phase of *rx_clk*



*TX doesn't have to send valid data (D0, D1, D2, etc.) right after reset. It can wait for RX to be stable (after rx_en = 1), then send valid data. Before that, it keep the bus 0).
The example is used to shorten the waveform.

Design both TX and RX side.

C. Revision

Version	Date	Description
v0p1	8-22-22	Initial version
v0p2	8-30-22	Fix various typos. Add Morse code waveform. Add 2 advanced problems.