PUFacademy

A PUFsecurity Alliance

Digital Logic Design
- Lecture 1
- Introduction

2025 Spring

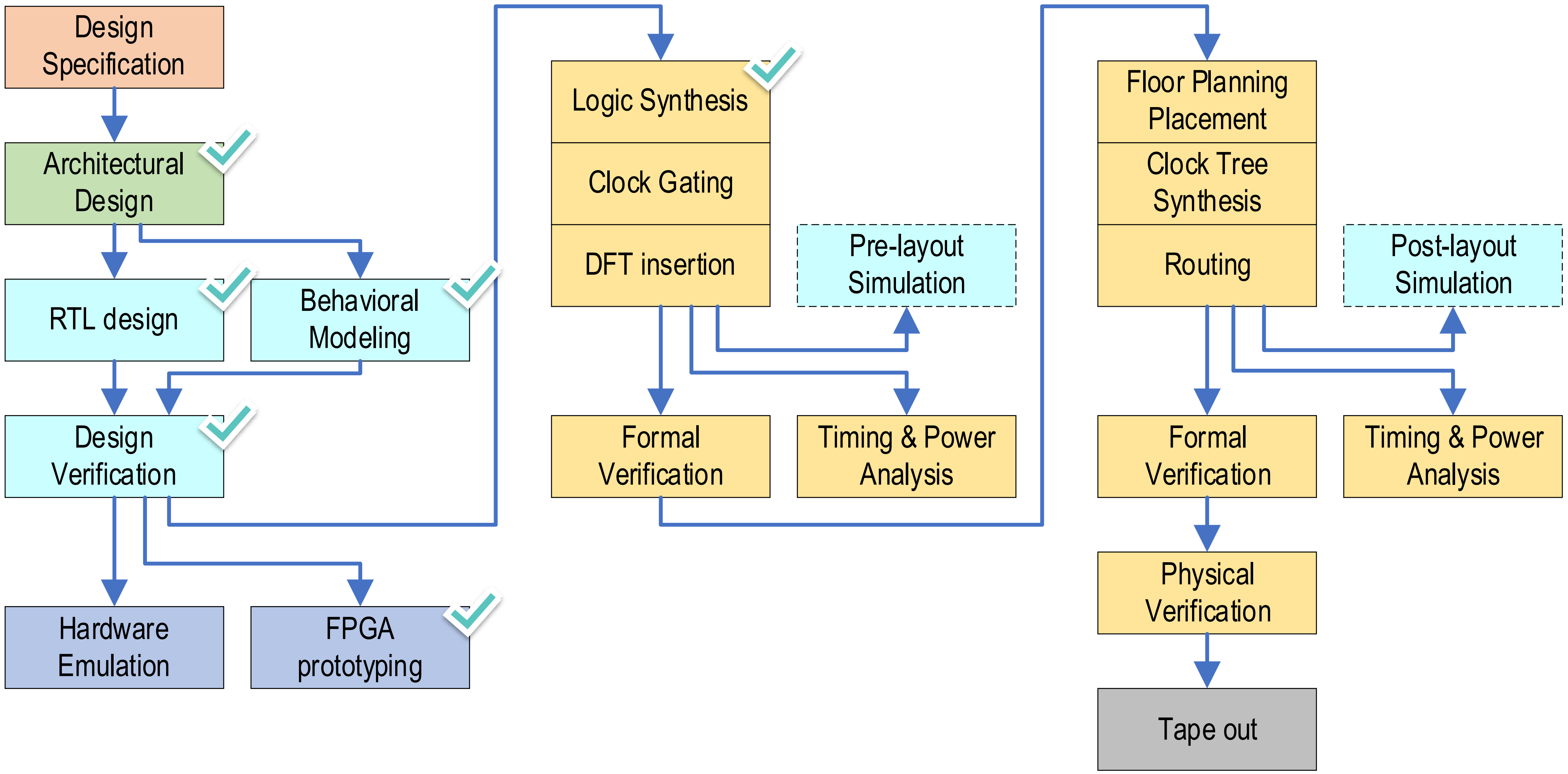
PUFacademy

A PUFsecurity Alliance

Agenda

1. Introduction
2. RTL basic
3. Testbench basic

From Code to Chip .



Hardware Description Language

- Hardware Description Language (HDL)
  - Describe the structure and behavior of electronic circuits
  - Writing HDL is just like drawing a circuit.

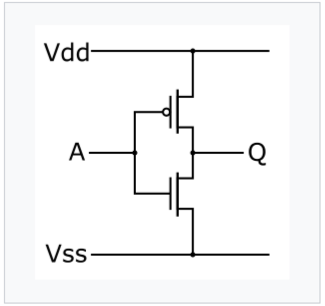
- Important Note
  - HDL is not a software code

Software language	HDL
Describe program, Execute in order	Describe hardware, Execute in parallel
Can be run on specific platforms like CPU or MCU	Can be general purpose IC (CPU, MCU) or application specific IC (ASIC)
Software code Binary code Machine code	Register-Transfer Level Gate level Transistor level

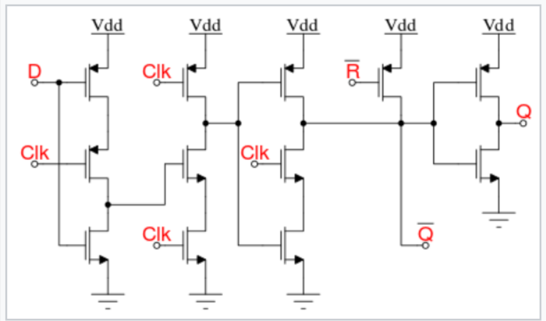
Hardware Description Language

- Common HDL
  - Transistor Level
    - HSPICE
    - PSPICE
  - Gate Level
  - Register-Transfer Level (RTL)
    - VHDL
    - Verilog

This Lecture



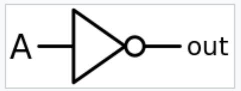
Static CMOS logic inverter



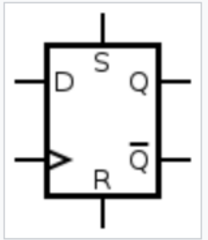
A CMOS IC implementation of a dynamic edge-triggered flip-flop with reset

Inverter (logic gate) - Wikipedia

Flip-flop (electronics) - Wikipedia



Traditional NOT gate (inverter) symbol



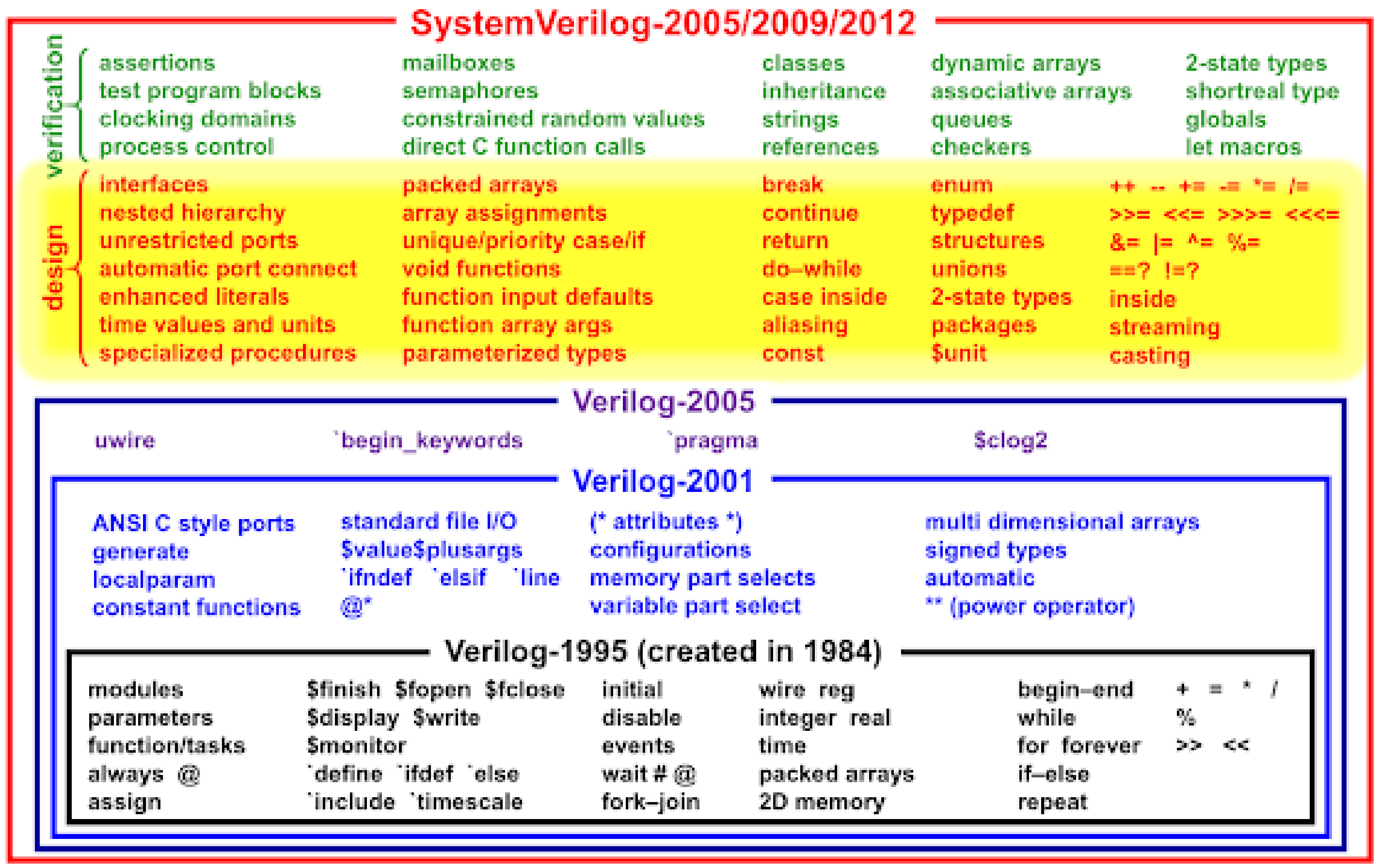
D flip-flop symbol

History of Verilog

- First created in 1983
- IEEE 1364-1995: Verilog-95 (.v)
  - Fundamental syntax
- IEEE 1364-2001: Verilog-2001
  - Improve readability and functionality
- IEEE 1364-2005: Verilog-2005
  - Minor improvements

This Lecture

- IEEE 1800-2009: SystemVerilog (.sv)
  - More strict and abstract way for describing hardware
  - Powerful testbench syntax



https://nguyenquanicd.blogspot.com/2017/08/verilog-nao-la-verilog-hoac-system.html

Agenda

1. Introduction
2. RTL basic
3. Testbench basic

Verilog Execution Order

- In Verilog, all code blocks are executed in parallel

- design
  - All hardware (assignment or procedural blocks) were executed in parallel

```verilog
assign a = e;

always@(*)begin
    b = |f;
end
```

```verilog
assign c = g;

always@(*)begin
    d = &h;
end
```

- testbench
  - All procedural blocks of test code were start simultaneously

```verilog
initial begin
    a = 1'b0;
end
```

```verilog
initial begin
    b = 1'b1;
end
```

Verilog Execution Order Inside “begin…end”

- Blocking assignment “=”
  - Like software code
  - Executed in order

```
//////////////////////////
//blocking assignment
integer ba;
integer bb;

initial begin
   #1;
   //init value
   ba = 1;
   bb = 2;

   #10;//delay 10ns
   ba = bb;
   bb = ba;

   $display("-----"   );
   $display("ba:%d",ba); //2
   $display("bb:%d",bb); //2
   $display("-----"   );
end
```

- Non-blocking assignment “<=”
  - Describe the circuit behavior
  - Executed in parallel

```
//////////////////////////
//non-blocking assignment
integer na;
integer nb;

initial begin
   @(negedge rst_n);
   //init value
   na <= 1;
   nb <= 2;

   @(posedge clk);
   na <= nb;
   nb <= na;
   #1;//print value a little later
   $display("-----"   );
   $display("na:%d",na); //2
   $display("nb:%d",nb); //1
   $display("-----"   );
end
```

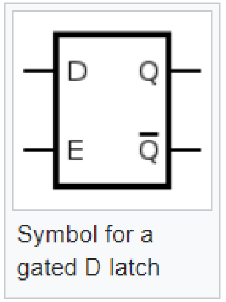
Basic Components of a Digital Circuit

- Combinational circuits
  - Logic gates
  - The output depends directly on the input, and be independent on time event signal

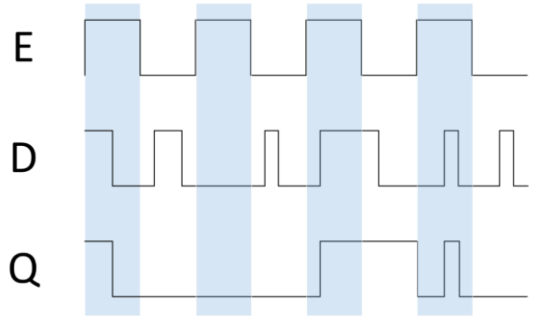
- Sequential circuits
  - The output depends not only on the current input, but also on time event signal
  - It has ability to store previous state.

Sequential circuits

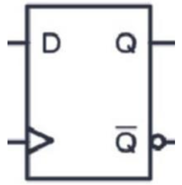
- Latch, level-triggered
  - Output will only change when the enable signal occurs

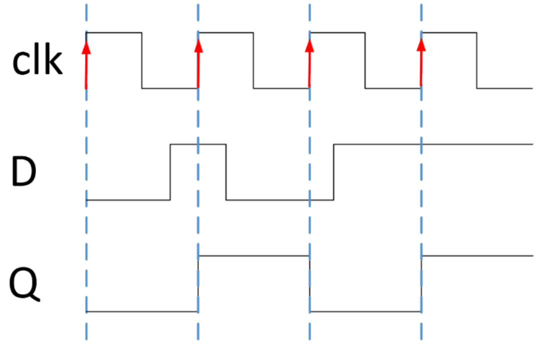


Symbol for a gated D latch



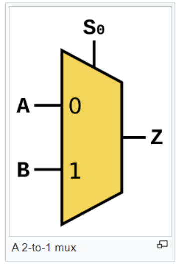
- Flip-Flop (also called register), edge-triggered
  - Output will only change when the edge of clock/reset occurs





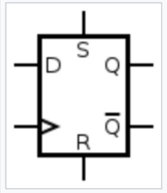
Signal Declaration vs Design Intent

- Combinational circuits
  - "wire" type for continuous assignment
  - "reg" type for procedural block



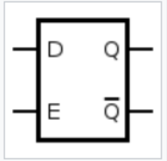
A 2-to-1 mux

- Sequential circuit
  - "reg" type for procedural block
  - Flipflop



D flip-flop symbol

  - "reg" type for procedural block
  - Latch



Symbol for a gated D latch

```
wire c0;//cmb_0
assign c0 = (sel)? x: y;

reg  c1;//cmb_1
always@*begin
   if(sel) c1 = x;
   else    c1 = y;//complete statement
end                //becomes combinational




reg  f0;//flipflop_0
always@(posedge clk or negedge rst_n)begin
   if(!rst_n) f0 <= 1'b0;
   else if(..)f0 <= ...;
end




reg  l0;//latch_0
always@*begin
   if(en) l0 = x;//incomplete statement
end              //becomes latch
```

- Describe the logical behavior of the circuit,
  in fact, there will be a time delay in each circuit after synthesis to different technology node.

4-state Variable in Verilog

- To describe the circuit behavior, the variable (each bits) has 4 states in Verilog
  - "0" : logical 0 (Physically connected to VSS/GND)
  - "1" : logical 1 (Physically connected to VDD)
  - "z" : high-impedance (Physically not connected or tri-state)
  - "x" : unknown (Simulator cannot determine the value)

- Value after declaration
  - wire : "z"
  - reg : "x"

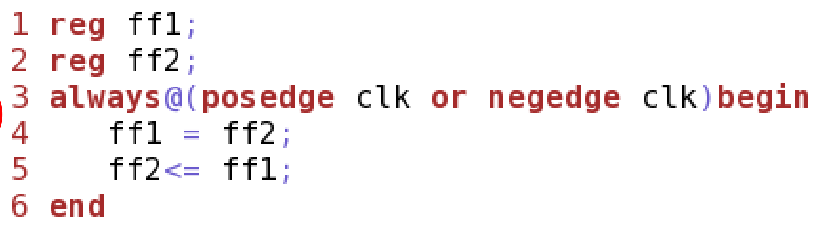
- Value propagation
  - wire : become "x" if RHS has "x" or "z", and the simulator cannot determine the value.
  - reg : become "x" if RHS has "x" or "z", and the simulator cannot determine the value.

(RHS: Right Hand Side)

Avoid "x" in design after reset signal

Coding Style

- Verilog syntax is very loose and can write a variety of circuit behavior, but the circuit of such behavior may not exist.
- Bad coding styles can lead to inconsistent results in different EDA tools.
  - Simulator vs Simulator (ex: NC-verilog, VCS)
    - In design or testbench
    - Weird waveform (blocking, non-blocking racing issue)
  - Simulator vs Synthesizer (ex: NC-verilog, Design Compiler)
    - In design
    - Unexpected circuit (latch or other strange circuit)

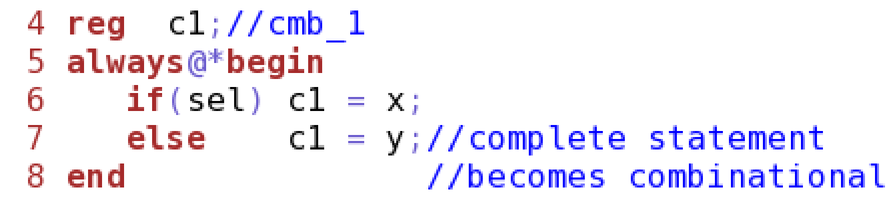


```
reg ff1;
reg ff2;
always@(posedge clk or negedge clk)begin
   ff1 = ff2;
   ff2<= ff1;
end
```

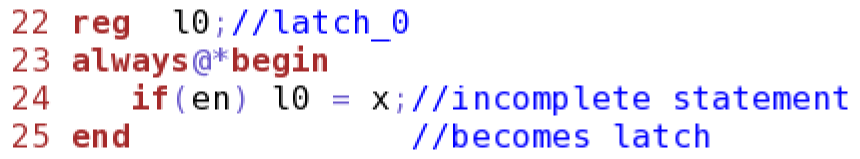
Avoid Latches in Design

- Problem of latch
  - level-triggered components are sensitive to noise interference.
  - Difficult to analyze timing in the synthesis stage.
  - Only some special purposes will intentionally use latch.

- Important Note
  - Most of the time, avoid latches in design stage



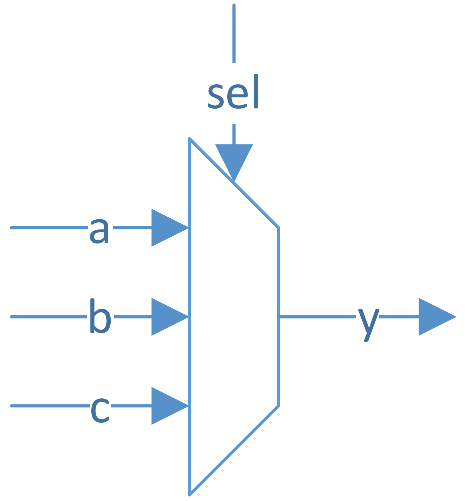
```verilog
reg  c1;//cmb_1
always@*begin
   if(sel) c1 = x;
   else    c1 = y;//complete statement
end               //becomes combinational
```



```verilog
reg  l0;//latch_0
always@*begin
   if(en) l0 = x;//incomplete statement
end              //becomes latch
```

Coding Style for Combinational Circuit

- Design intent: 3-to-1 mux



```
//bad coding style,
//incomplete statement
//y will hold it's value when sel=2'b11, it becomes latch
always@*begin
   case(sel)
      2'b00:   y=a;
      2'b01:   y=b;
      2'b10:   y=c;
   endcase
end
```

```
//bad coding style,
//incomplete statement, with EDA tools specific comment
//y is 3-to-1 mux only when using synopsys tool
always@*begin
   case(sel)//synopsys full_case
      2'b00:   y=a;
      2'b01:   y=b;
      2'b10:   y=c;
   endcase
end
```

```
//correct coding style
//use default (or else) statement
//y is 3-to1 mux in all situations
always@*begin
   case(sel)
      2'b00:   y=a;
      2'b01:   y=b;
      default: y=c;
   endcase
end
```

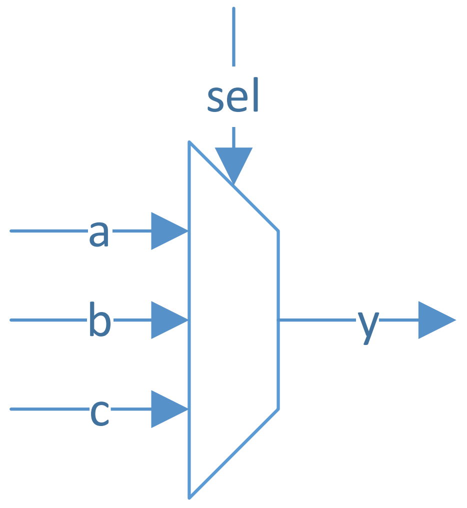
Coding Style for Combinational Circuit

- Design intent: 3-to-1 mux
  - Verilog-1995

```verilog
//Verilog-1995
//Sensitive lists should be specifically filled out
reg y;
always@(a or b or c or sel)begin
   if(sel==2'b00)        y=a;
   else if(sel==2'b01)   y=b;
   else                  y=c;
end
```

  - Verilog-2001

```verilog
//Verilog-2001
//Sensitive lists can be *
reg y;
always@*begin
   if(sel==2'b00)        y=a;
   else if(sel==2'b01)   y=b;
   else                  y=c;
end
```



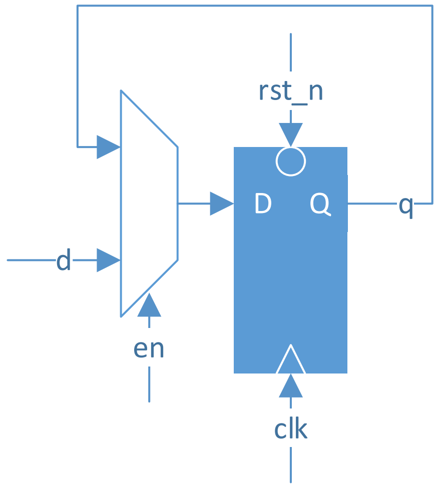
Coding Style for Sequential Circuit

- Design intent: D-flipflop with selection
  - What will happen if rst_n == 0 and en == 1?

```verilog
//bad coding style,
//control a variable in different blocks
reg q;
always@(posedge clk or negedge rst_n)begin
   if(!rst_n) q<=1'b0;
end
always@(posedge clk or negedge rst_n)begin
   if(en)     q<=d;
end
```

  - Describe the signal behavior in one always block

```verilog
//correct coding style
//control a variable in only-1 block
reg q;
always@(posedge clk or negedge rst_n)begin
   if(!rst_n) q<=1'b0;
   else if(en)q<=d;
end
```



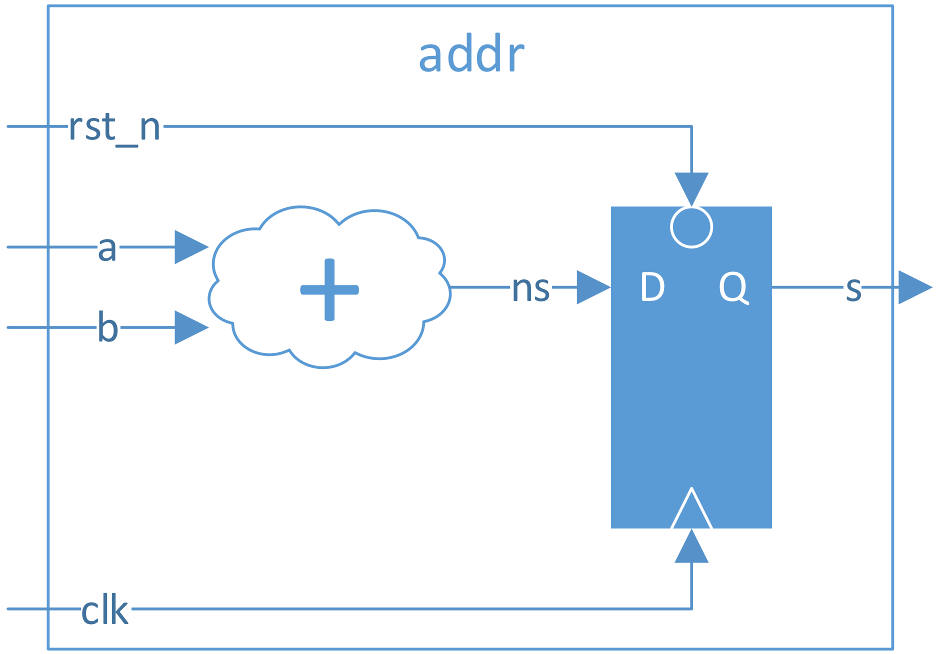
Module

- "module" represents a group of related circuits in Verilog
- Can be either combinational or sequential circuits or both

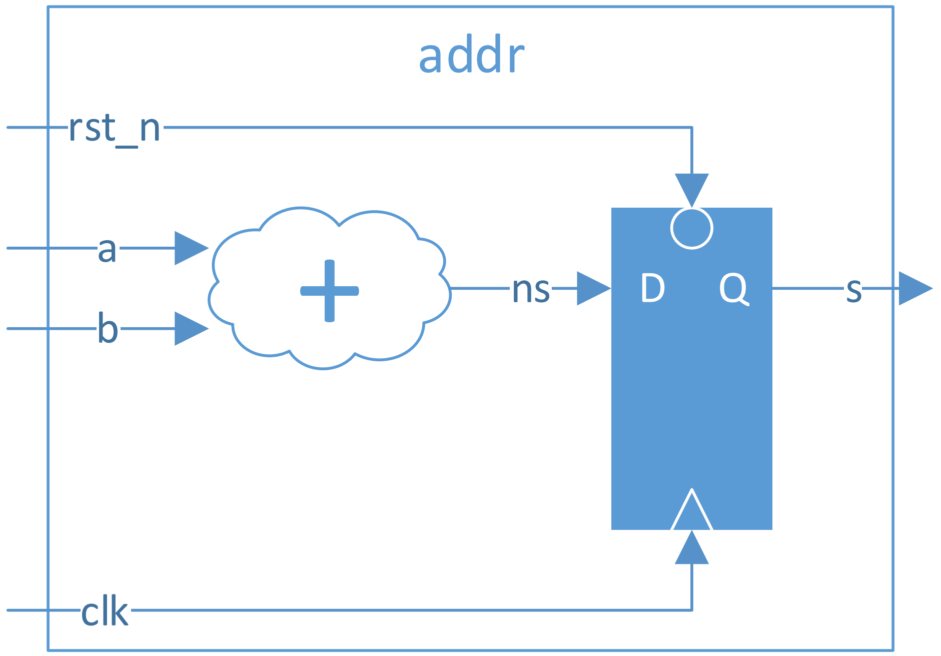
```
module module_name
```

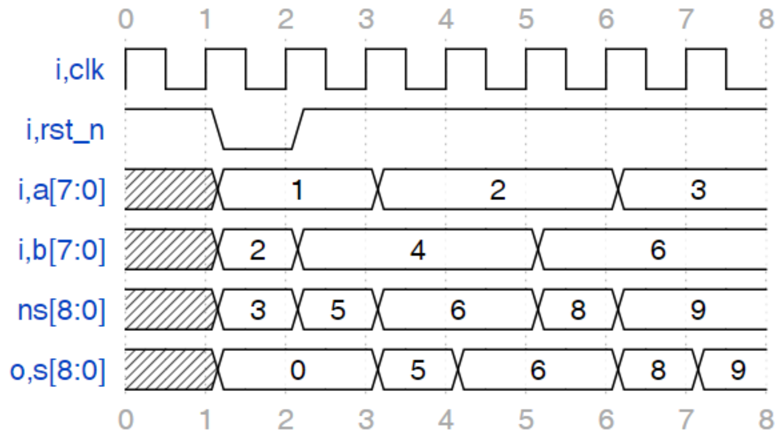
- Parameter list
  - Configurable when using this module
  - Constant at compile time
- Port list
  - input (Usually have)
  - output (Usually have)
  - inout (special circuits)
- Internal signal declaration
- Circuit implementation

```
endmodule
```



Module

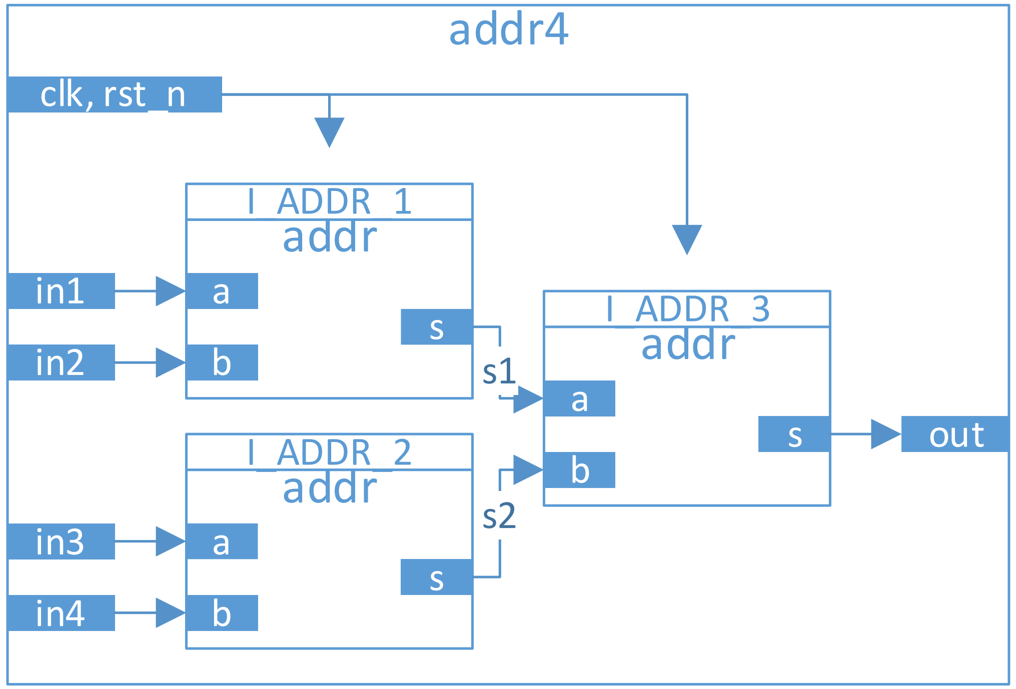




```verilog
module addr#( //name of this module
   //parameter declaration
   parameter WIDTH = 8
   )(
   //port declaration
   input        [WIDTH-1:0] a    ,//input a
   input        [WIDTH-1:0] b    ,//input b
   output reg [WIDTH   :0] s     ,//seq output of (a + b)
   input                    clk  ,
   input                    rst_n
   );

   //internal signal declaration
   wire [WIDTH  :0] ns;//cmb value of (a+b)

   //circuit implementation
   //cmb
   assign ns = a + b;
   //seq
   always@(posedge clk or negedge rst_n)begin
      if(!rst_n)  s <= {WIDTH+1{1'b0}};
      else        s <= ns;
   end

endmodule //end of this module
```

Module Instance and Connection

```
module addr4#(
    //parameter declaration
    parameter IN_WIDTH = 8
    )(
    //port declaration
    input  [IN_WIDTH-1:0] in1  ,
    input  [IN_WIDTH-1:0] in2  ,
    input  [IN_WIDTH-1:0] in3  ,
    input  [IN_WIDTH-1:0] in4  ,
    output [IN_WIDTH+1:0] out  ,
    input                 clk  ,
    input                 rst_n
    );

    //internal signal declaration
    wire   [IN_WIDTH  :0] s1;
    wire   [IN_WIDTH  :0] s2;

    //sub-module instancec
    addr#(.WIDTH(IN_WIDTH  ))I_ADDR_1(.a(in1), .b(in2), .s(s1 ), .clk(clk), .rst_n(rst_n));
    addr#(.WIDTH(IN_WIDTH  ))I_ADDR_2(.a(in3), .b(in4), .s(s2 ), .clk(clk), .rst_n(rst_n));
    addr#(.WIDTH(IN_WIDTH+1))I_ADDR_3(.a(s1 ), .b(s2 ), .s(out), .clk(clk), .rst_n(rst_n));

endmodule
```



Signal Declaration, wire? reg?

- Module
  - Internal signal
    - Flipflop – reg
    - Combinational – reg or wire
  - IO
    - Input – wire (leave it blank)
    - Output – reg or wire (leave it blank)

What does the circuit look like?

- Sub-module instance
  - Connect to input – wire or reg
  - Connect to output – wire

- Having the circuit in your mind,
  It is obvious what type to declare.

Coding Style for Port Declaration

- Module port declaration
  - Verilog-1995
  - Many repetitive declarations

```
//Verilog-1995 port declaration
module addr(a, b, s, clk, rst_n);
   parameter WIDTH = 8;

   input      [WIDTH-1:0] a    ;
   input      [WIDTH-1:0] b    ;
   output     [WIDTH  :0] s    ;
   input                  clk  ;
   input                  rst_n;

   reg        [WIDTH  :0] s    ;

   //...
   //...

endmodule
```

  - Verilog-2001
  - Simplified declarations
  - add local parameter

```
//Verilog-2001 port declaration
module addr#(
   parameter WIDTH = 8
   )(
   input      [WIDTH-1:0] a    ,
   input      [WIDTH-1:0] b    ,
   output reg [WIDTH  :0] s    ,
   input                  clk  ,
   input                  rst_n
   );

   localparam PIPENUM = 1;
   //...
   //...

endmodule
```

Coding Style for Port Connection

- Sub-module instance port connection
  - by order (implicit)

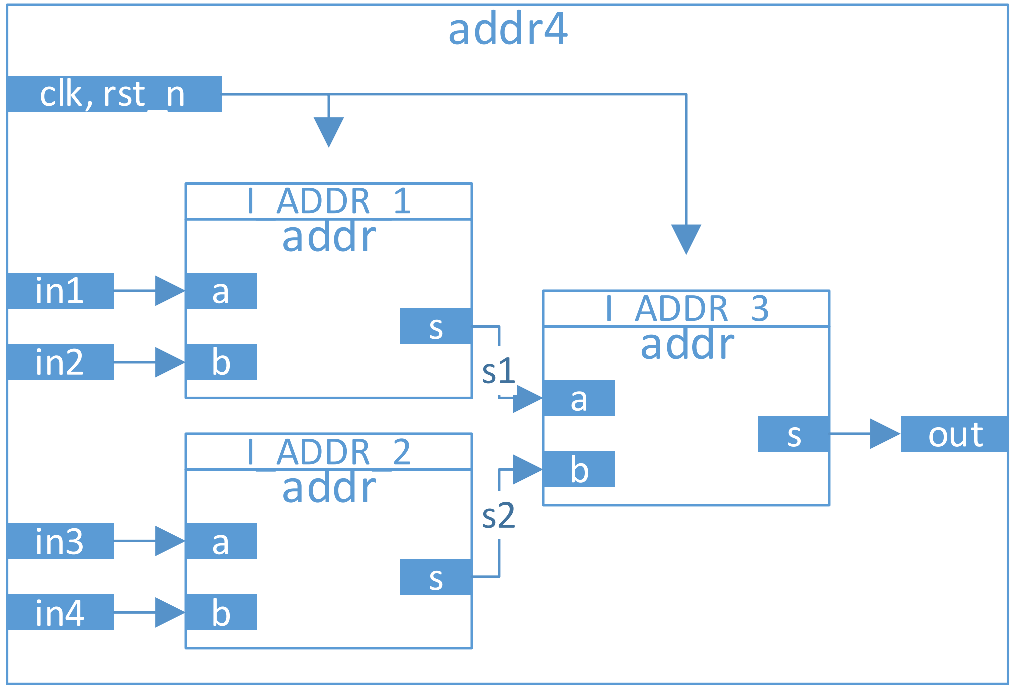
```
//Verilog-1995 port connection, by order
addr#(IN_WIDTH) I_ADDR_1(in1, in2, s1, clk, rst_n);
```

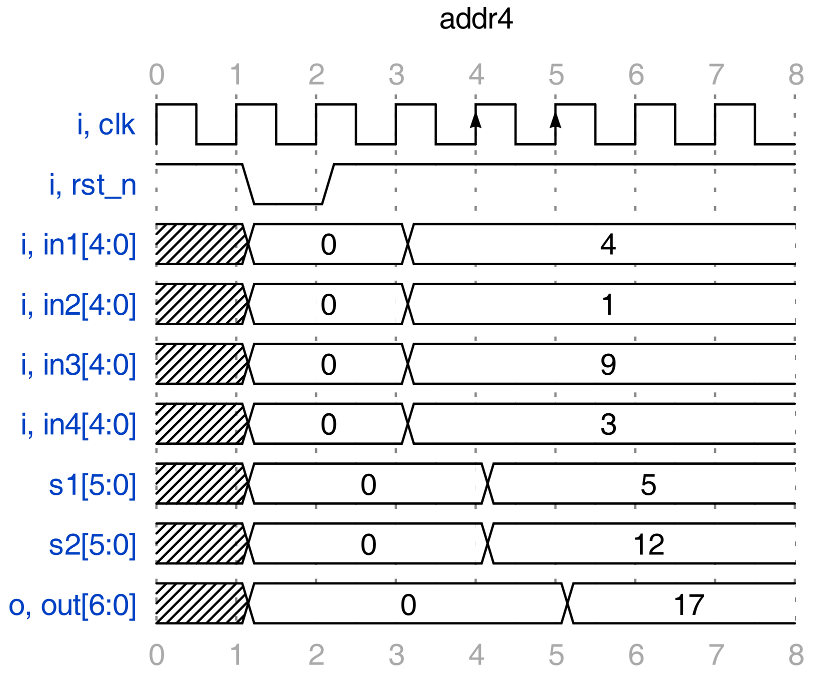
  - by name (explicit)

```
//Verilog-2001 port connection, by name
addr#(.WIDTH(IN_WIDTH))I_ADDR_1(.a(in1), .b(in2), .s(s1 ), .clk(clk), .rst_n(rst_n));
```

Module IO Behavior

- When is the correct “out” value available?
- In order to easy integration of different modules, the IO waveform must be specified.



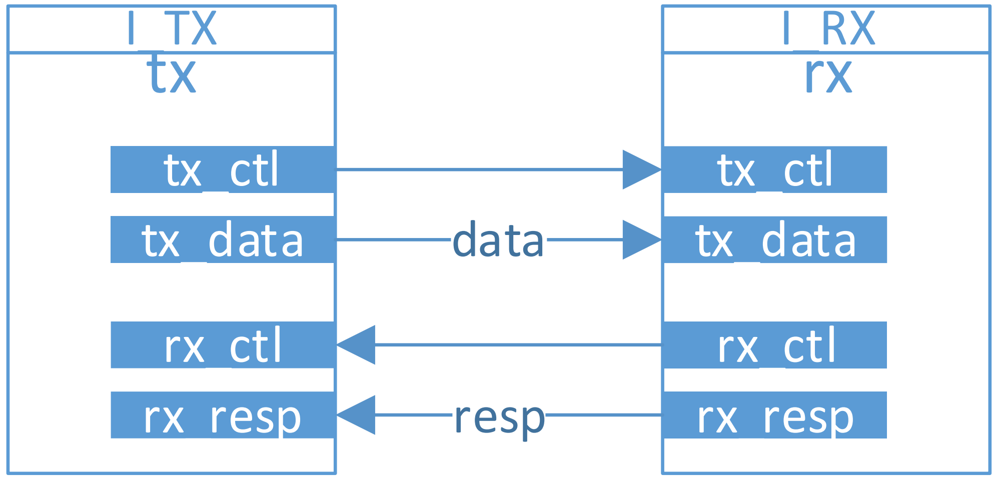


Common Types of IO Protocol

- Add control signals next to data signal.
  - The transmitter (tx) module
    - sends a control signal with data (data is optional)
  - The receiver (rx) module
    - replies a control signal with response (response is optional).

- Control signal types
  - Level (high / low)
  - Pulse (high within a clock period)

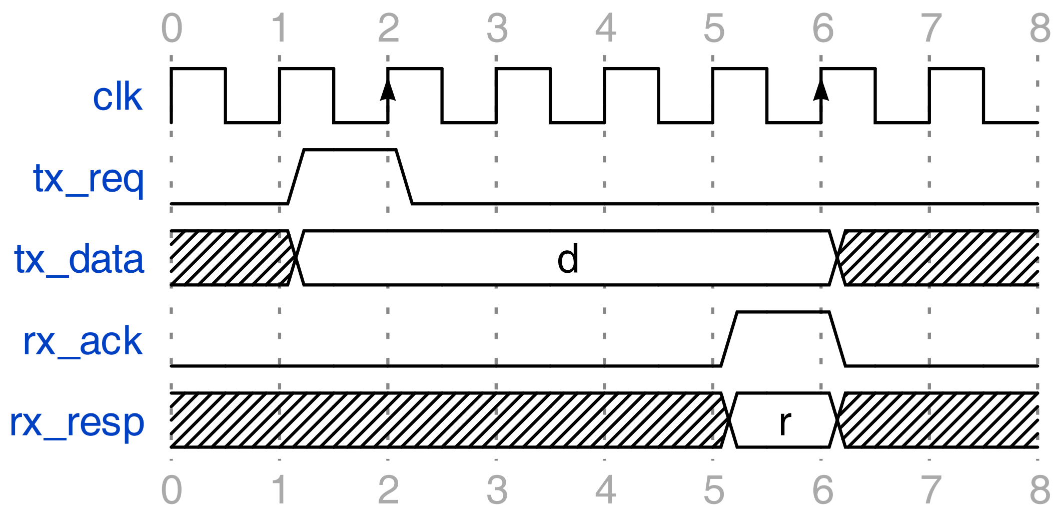
- Common types of IO protocol
  - Pulse, Pulse
  - Level, Pulse
  - Level, Level
  - Pulse, Level



Common Types of IO Protocol

- Request (Pulse), Acknowledgement (Pulse)
  - TX sends a request (req) with optional data (d)
  - RX replies an acknowledgement (ack) with an optional response signal (r)

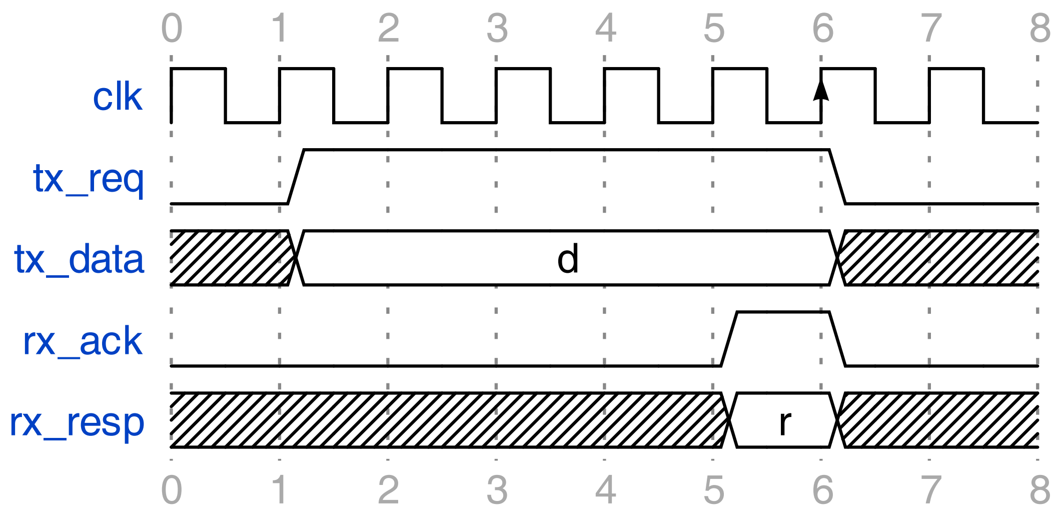
pulse to pulse



Common Types of IO Protocol

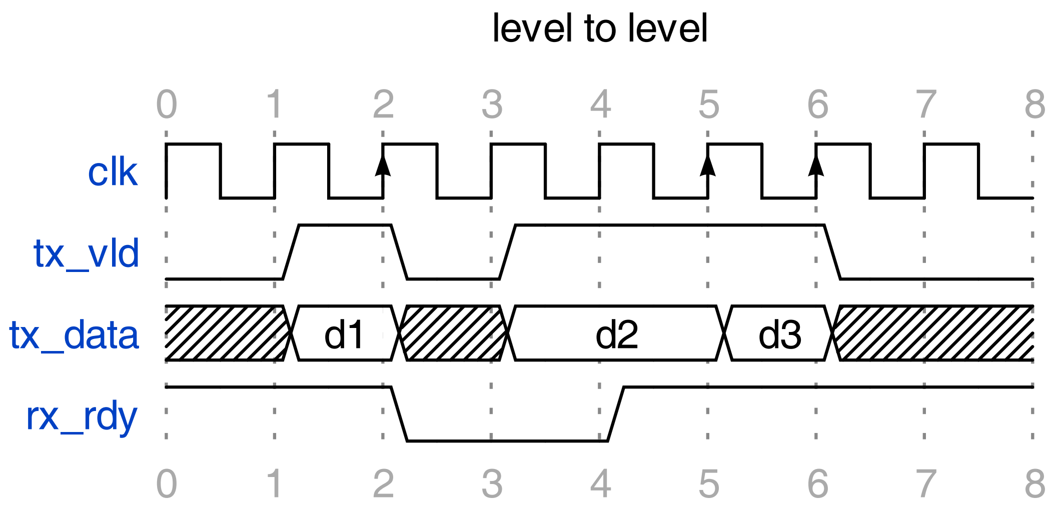
- Request (Level), Acknowledgement (Pulse)
  - TX sends a request (req) with optional data (d)
  - RX replies an acknowledgement (ack) with an optional response signal (r)

level to pulse



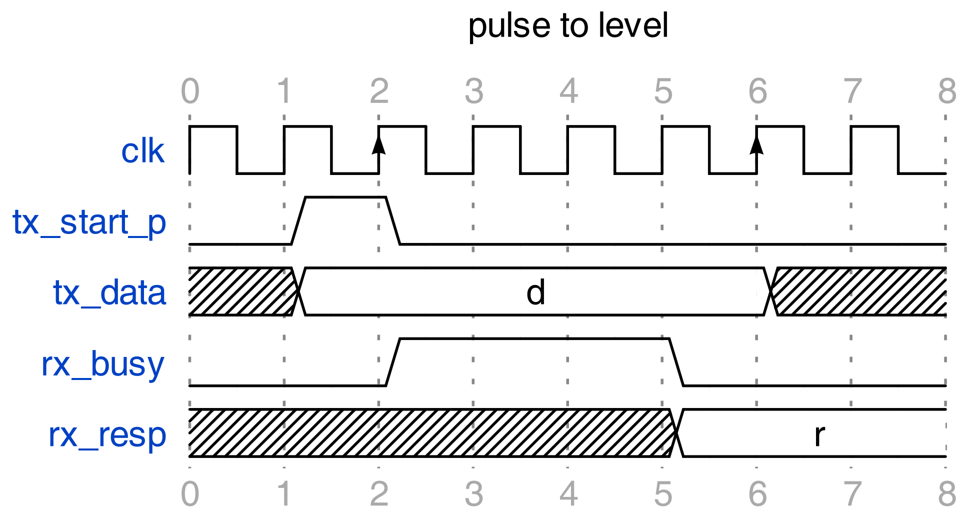
Common Types of IO Protocol

- Valid (Level), Ready (Level)
  - TX pulls up a valid signal (vld)
    to indicate that the output data (d) is valid.
  - RX pulls up a ready signal (rdy)
    to indicate that it is ready to receive data (d).
  - When both valid and ready are high
    it means the data has been successfully transferred.
  - This protocol is often used to send continuous data
  - The valid and ready signals must be independent from each other to avoid deadlock.



Common Types of IO Protocol

- Start (pulse), Busy (Level)
  - TX sends a start signal (start_p) with optional data (d)
  - RX pulls up the busy signal (busy) and starts the internal operation when the operation is complete puts down the busy along with the response (r).
  - This protocol is usually used for the controller module to operate the calculation module.



Agenda

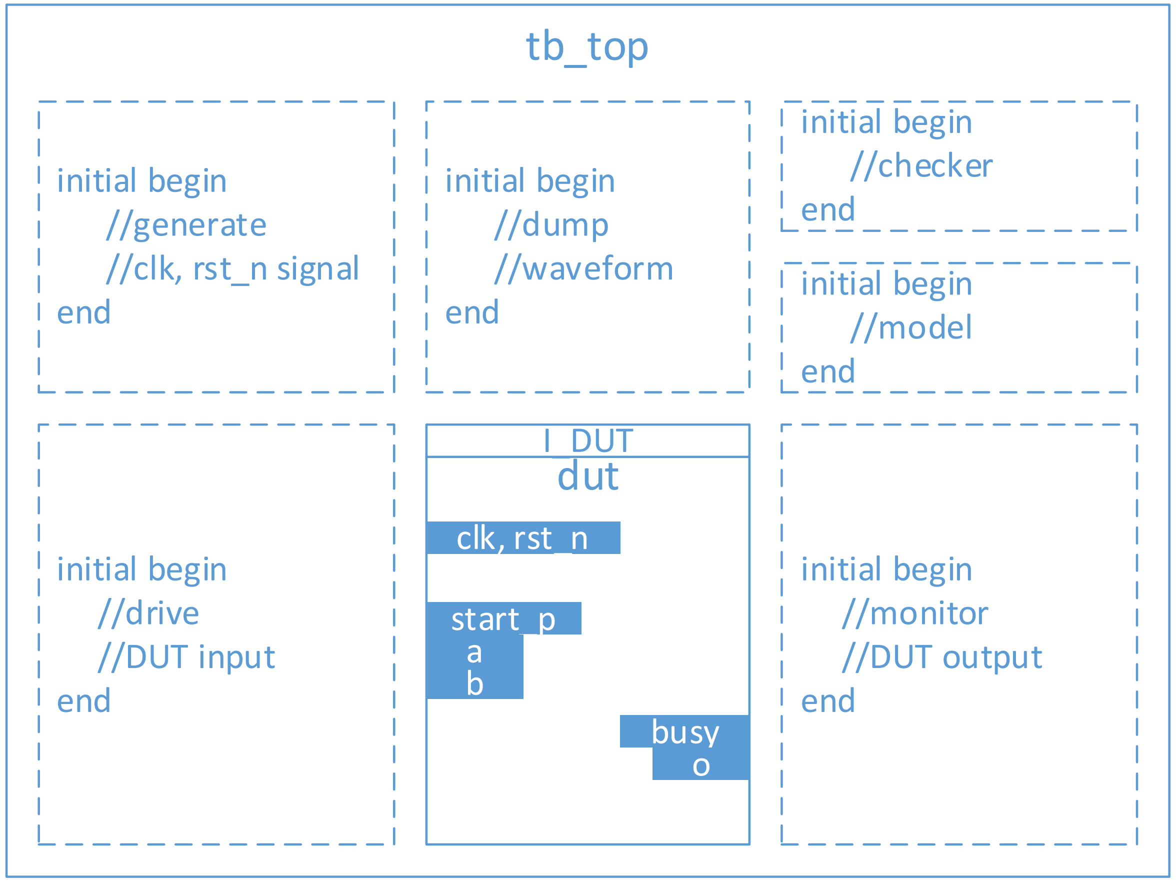
Basic Components of Testbench

- Testbench (tb_top.sv)

```
module tb_top;
```

  - Dump waveform
  - DUT (Design Under Test) instance
  - Generate clock, reset signal
  - Drive DUT input (test pattern)
  - Monitor DUT output
  - Behavioral model of DUT
  - Checker to compare answer
  - …

```
endmodule
```

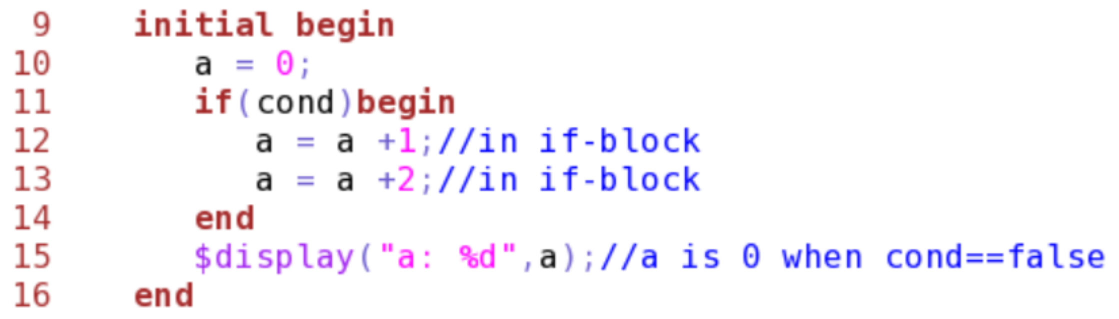


tb_top
initial begin //generate //clk, rst_n signal end
initial begin //dump //waveform end
initial begin //checker end
initial begin //model end
I_DUT
dut
clk, rst_n
start_p
a
b
busy
o
initial begin //drive //DUT input end
initial begin //monitor //DUT output end

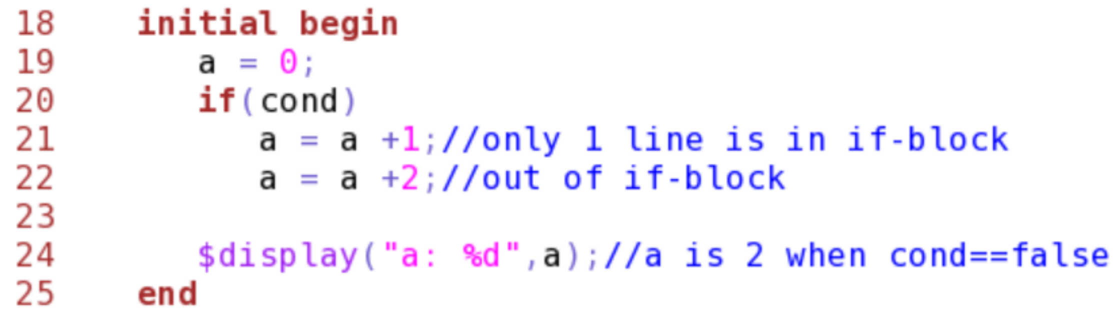
Basic Syntax of Testbench

- Procedural block (initial)
  - System task ($)
    - $random
    - $display
    - $finish
  - Time related
    - delay a specific time (#)
    - wait for event (@)
    - wait for condition (wait)
  - Branch
    - if-else
    - switch-case
  - Loop
    - for, while, repeat, forever

- begin … end
  - A block of code in a branch or a loop



```
initial begin
    a = 0;
    if(cond)begin
        a = a +1;//in if-block
        a = a +2;//in if-block
    end
    $display("a: %d",a);//a is 0 when cond==false
end
```



```
initial begin
    a = 0;
    if(cond)
        a = a +1;//only 1 line is in if-block
        a = a +2;//out of if-block

    $display("a: %d",a);//a is 2 when cond==false
end
```

Dump Waveform

```verilog
4   //////////////////////////////////////////////////////
5   // DUMPWAVE
6   //////////////////////////////////////////////////////
7   `ifdef USE_FSDB
8   initial begin
9      if($test$plusargs("FSDB")) begin//+FSDB to enable
10        $fsdbDumpfile("wave.fsdb");
11        // $fsdbDumpvars(depth, instance);
12        $fsdbDumpvars();
13     end
14  end
15  `endif
```

- `define is a preprocessor directive
  Can enable or disable some part of code,
  Not recommended for use in design.

- Because $fsdbDumpXXX is a Synopsys specific command,
  if you use other simulator, it will cause compile error, so use `ifdef … `endif to separate it,
  use +define+USE_FSDB to open this code when simulation.

- Also +FSDB to trigger $test$plusargs(“FSDB”) to enter if statement in line 9.

Timeout Control

- In the simulation process,
  there may be some reasons causing the simulation cannot be finished,
  so, it is better to set a timeout value in each testbench.

- In this example, the default timeout is 1000ns,
  if you need a longer timeout, you can use +TO=XXXX to overwrite.

- The # sign
  means delay
  for some unit length of time.

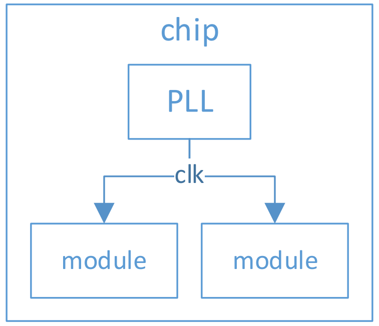
```
//        unit/precision
`timescale 1ns/1ps
```

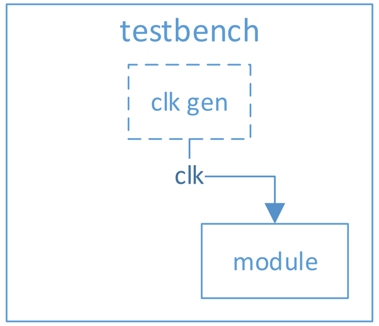
```
//////////////////////////////////////////////////////
// TIMEOUT
//////////////////////////////////////////////////////
integer timeout;
initial begin
   if(!$value$plusargs("TO=%d",timeout)) begin//+TO=XXXX to overwrite
      timeout = 1000;//default
   end
   #timeout;
   $display("*****************");
   $display("simulation timeout");
   $display("*****************");
   $finish;
end
```

Generate Clock and Reset

- In a large chip, the clock and reset signals are usually generated internally,
but in the case of a small module simulation,
the clock and reset signals are generated by the testbench.

```
///////////////////////////////////////////////////////
// CLK RESET
///////////////////////////////////////////////////////
reg clk  ;
reg rst_n;

initial begin
   clk = 1'b1;
   forever begin
      #(5.0/2.0) clk = ~clk;
   end
end
initial begin
   rst_n = 1'b1;
   #7;
   rst_n = 1'b0;
   #7;
   rst_n = 1'b1;
end
```





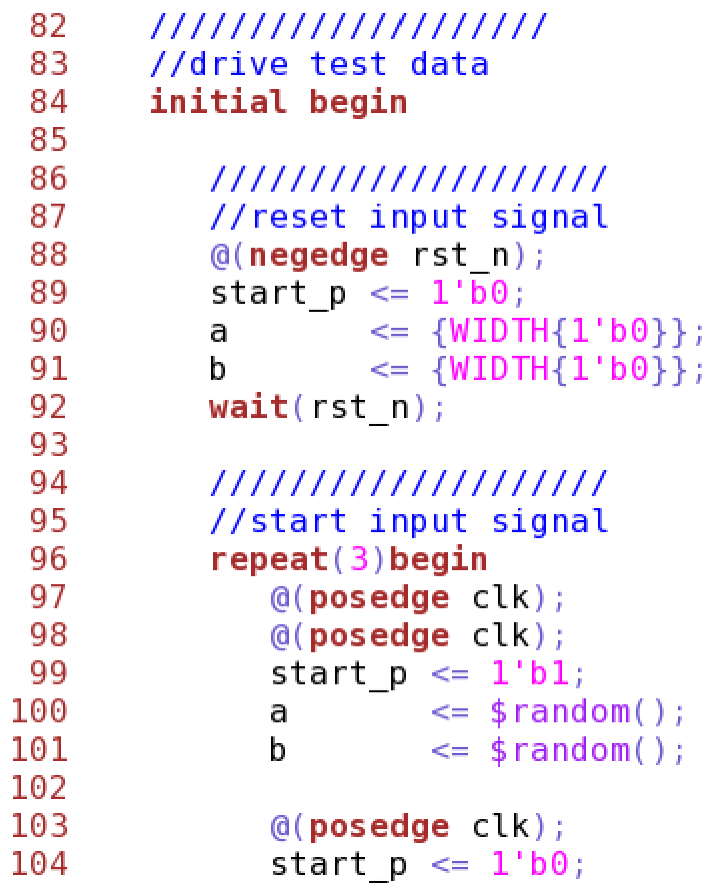
Instance of DUT

- There must be a module instance to be tested in the testbench, called DUT (Design Under Test)
- Set the parameter
  and connect the io when instance the module.
- Usually, the input signal of DUT
  will be operated in the initial block of testbench,
  so, it will be declared as reg type
- Testbench will check the output of the DUT
  and only wire types can be connected
  to the output of the module instance.

```
//////////////////////////////////////////////////////
// DUT
//////////////////////////////////////////////////////
parameter WIDTH = 5;
reg                start_p;
reg [WIDTH  -1:0] a;
reg [WIDTH  -1:0] b;
wire               busy;
wire[WIDTH*2-1:0] o;

mult2#(
    .WIDTH(WIDTH)
)I_MULT2(
    .start_p(start_p ),
    .a      (a       ),
    .b      (b       ),
    .busy   (busy    ),
    .o      (o       ),
    .clk    (clk     ),
    .rst_n  (rst_n   )
);
```

Drive DUT Input

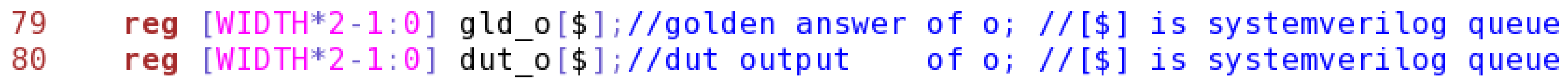
- Control the input signals to generate test patterns to DUT
- The testbench acts as another module to interact with the DUT
- In order to describe the correct circuit interaction behavior
  - it is recommended to use
    edge trigger and non-blocking assignment
    to control these inputs.



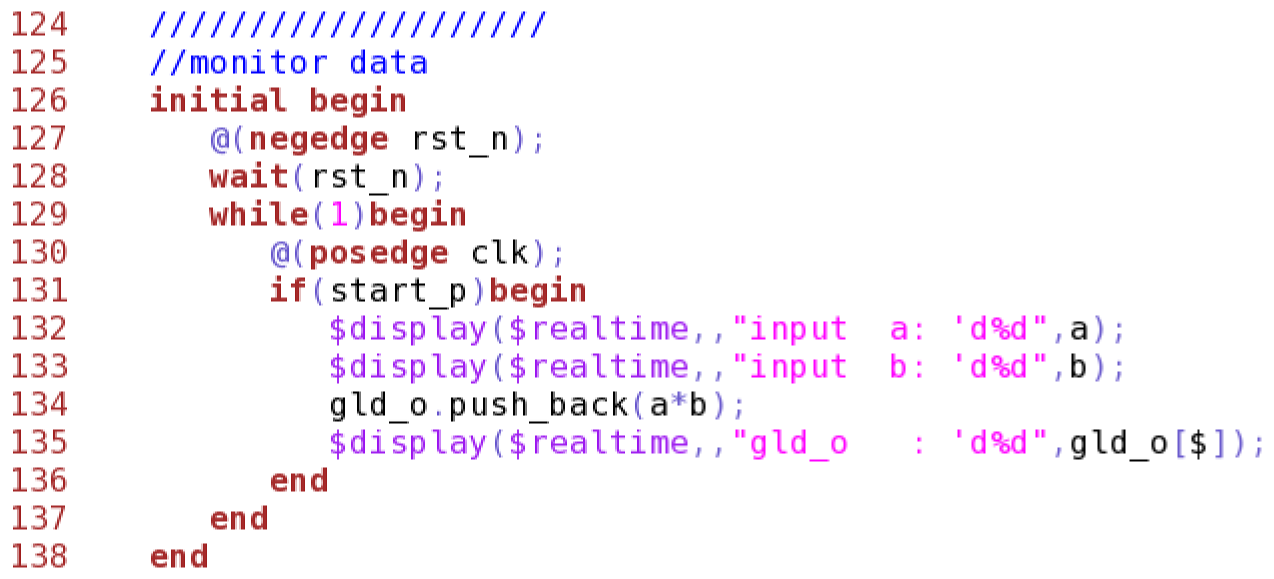
```
////////////////////
//drive test data
initial begin

   ////////////////////
   //reset input signal
   @(negedge rst_n);
   start_p <= 1'b0;
   a       <= {WIDTH{1'b0}};
   b       <= {WIDTH{1'b0}};
   wait(rst_n);

   ////////////////////
   //start input signal
   repeat(3)begin
      @(posedge clk);
      @(posedge clk);
      start_p <= 1'b1;
      a       <= $random();
      b       <= $random();

      @(posedge clk);
      start_p <= 1'b0;
```

Monitor DUT Input and Output

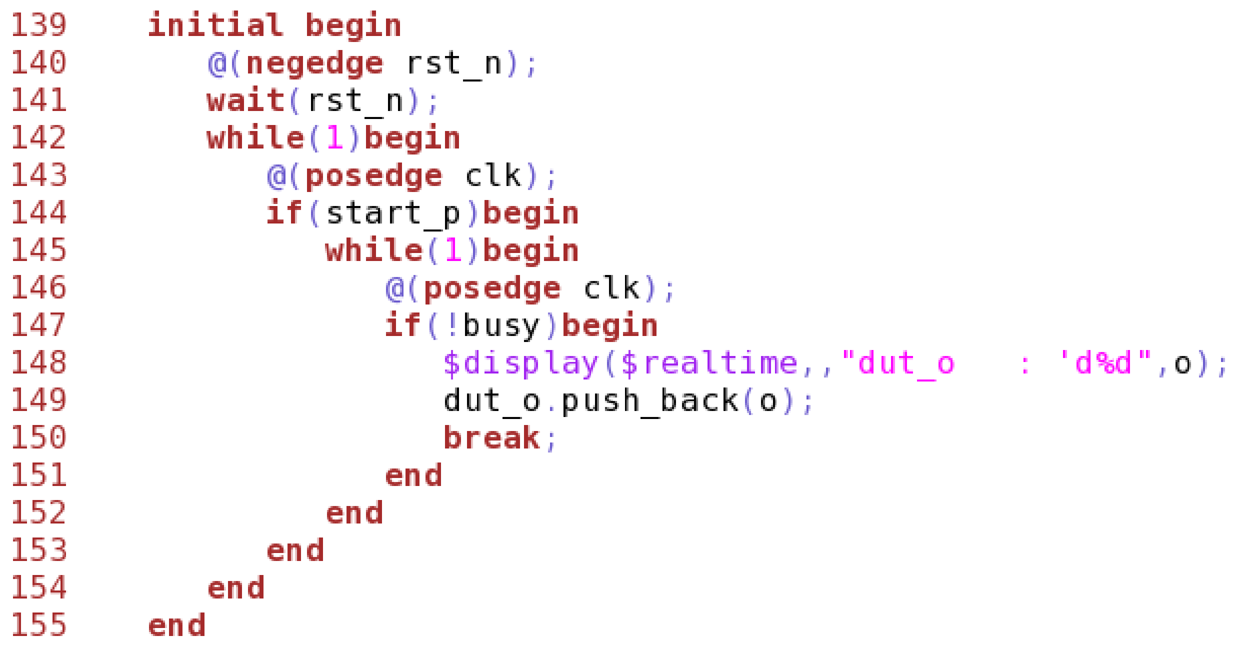
- An independent monitor block to capture and store the input and output of the DUT.
- Same as driver block, you can use edge-trigger control to capture the correct data.



```verilog
reg [WIDTH*2-1:0] gld_o[$];//golden answer of o; //[$] is systemverilog queue
reg [WIDTH*2-1:0] dut_o[$];//dut output    of o; //[$] is systemverilog queue
```



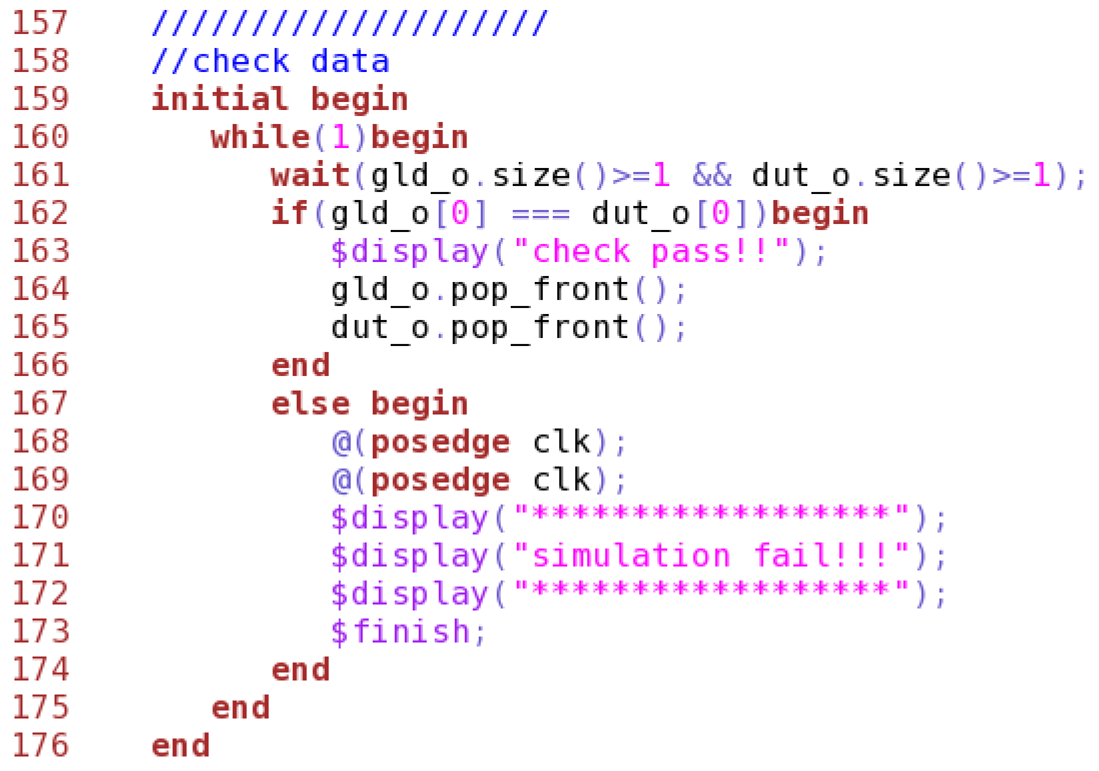
```verilog
////////////////////
//monitor data
initial begin
   @(negedge rst_n);
   wait(rst_n);
   while(1)begin
      @(posedge clk);
      if(start_p)begin
         $display($realtime,,"input  a: 'd%d",a);
         $display($realtime,,"input  b: 'd%d",b);
         gld_o.push_back(a*b);
         $display($realtime,,"gld_o   : 'd%d",gld_o[$]);
      end
   end
end
```



```verilog
initial begin
   @(negedge rst_n);
   wait(rst_n);
   while(1)begin
      @(posedge clk);
      if(start_p)begin
         while(1)begin
            @(posedge clk);
            if(!busy)begin
               $display($realtime,,"dut_o   : 'd%d",o);
               dut_o.push_back(o);
               break;
            end
         end
      end
   end
end
```

Check with Golden

- An independent checker block to compare the golden output with DUT output.
- If compare error, terminate the simulation.



```
////////////////////
//check data
initial begin
   while(1)begin
      wait(gld_o.size()>=1 && dut_o.size()>=1);
      if(gld_o[0] === dut_o[0])begin
         $display("check pass!!");
         gld_o.pop_front();
         dut_o.pop_front();
      end
      else begin
         @(posedge clk);
         @(posedge clk);
         $display("*****************");
         $display("simulation fail!!!");
         $display("*****************");
         $finish;
      end
   end
end
```

Reference

- IEEE Std 1800™-2017
- https://en.wikipedia.org/wiki/Verilog
- https://blog.csdn.net/l471094842/article/details/109714550

Feedback to us

NTHU 晶片安全設計 回饋單

https://forms.office.com/r/DYDu8vLaWN

Thank you!

PUFacademy

A PUFsecurity Alliance

Visit our website: pufacademy.com

Contact us: PUFacademy@pufsecurity.com