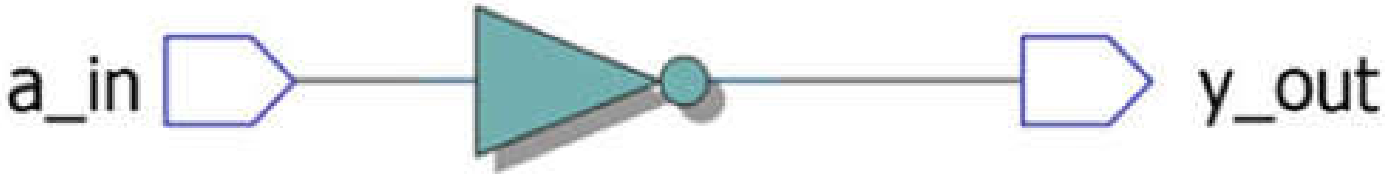
Digital / Logic Design Through Verilog HDL

Tuan Nguyen-viet

Part 1

LOGIC GATES

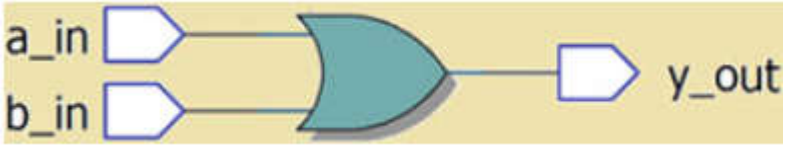
Synthesized NOT logic



Synthesized NOT logic

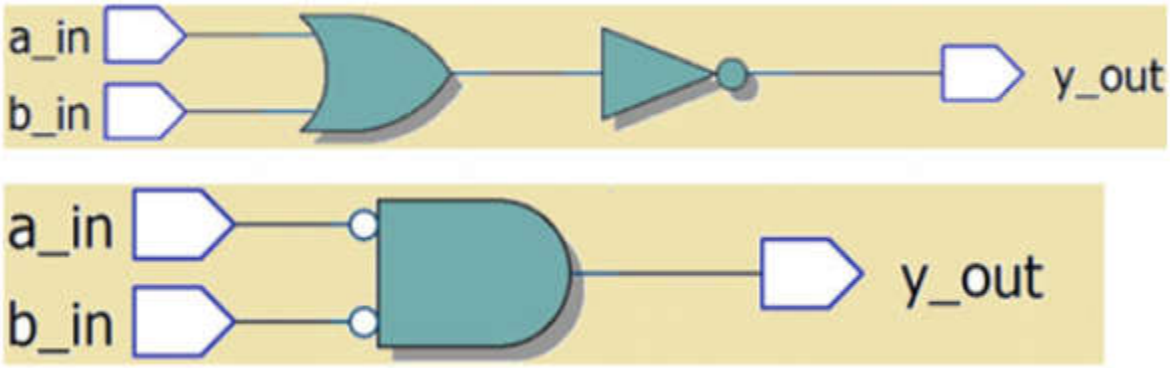
a_in	y_out
0	1
1	0

Synthesized two-input OR logic



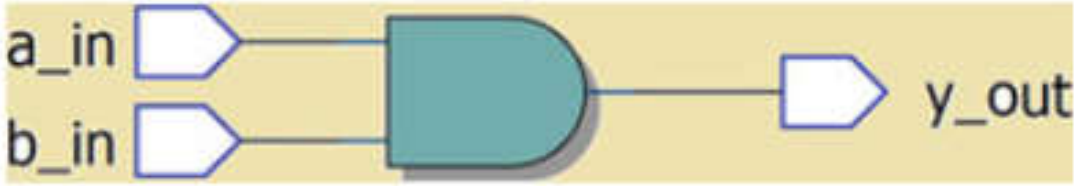
a_in	b_in	y_out
0	0	0
0	1	1
1	0	1
1	1	1

Synthesized two-input NOR logic



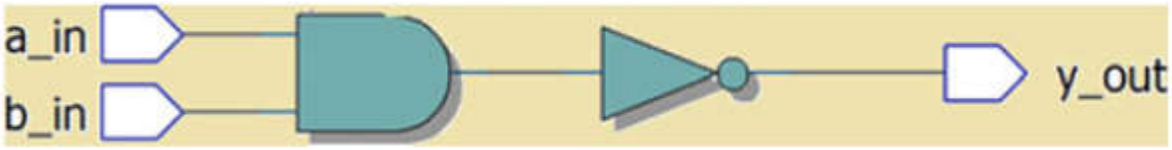
a_in	b_in	y_out
0	0	1
0	1	0
1	0	0
1	1	0

Synthesized two-input AND logic



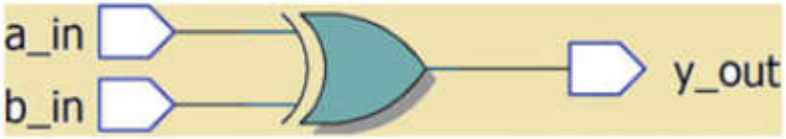
a_in	b_in	y_out
0	0	0
0	1	0
1	0	0
1	1	1

Synthesized two-input NAND logic



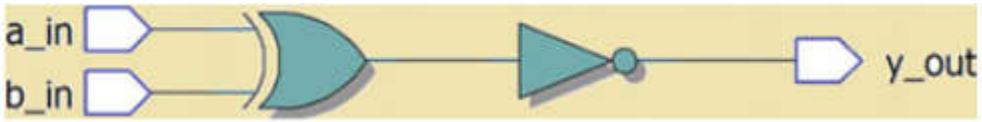
a_in	b_in	y_out
0	0	1
0	1	1
1	0	1
1	1	0

Synthesized two-input XOR logic



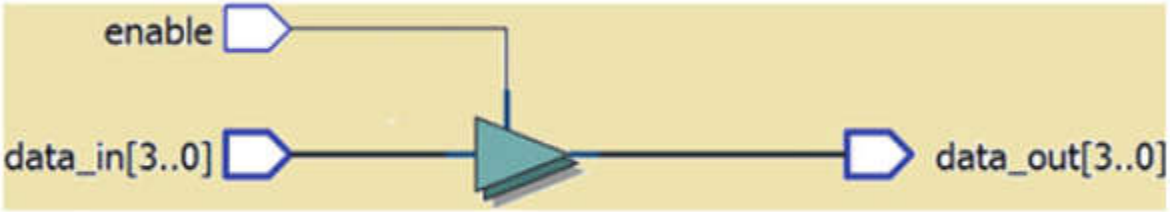
a_in	b_in	y_out
0	0	0
0	1	1
1	0	1
1	1	0

Synthesized XNOR logic



a_in	b_in	y_out
0	0	1
0	1	0
1	0	0
1	1	1

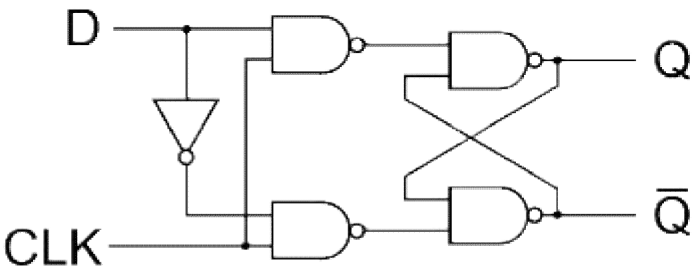
Synthesized tri-state NOT logic



enable	data_in	data_out
1	0000	0000
1	1111	1111
0	xxxx	zzzz

D Flip-Flop

- D flip-flop is very popular with digital electronics.
- They are commonly used for
  - counters,
  - shift registers,
  - and input synchronization.



- In the D flip-flops, the output can only be changed at the clock edge, and if the input changes at other times, the output will be unaffected.
  - The change of state of the output is dependent on the rising edge of the clock.
  - The output (Q) is the same as the input and can only change at the rising edge of the clock.

Truth Table:

Clock	D	Q	Q′
↓ » 0	0	0	1
↑ » 1	0	0	1
↓ » 0	1	0	1
↑ » 1	1	1	0

REF: https://www.electronicsforu.com/technology-trends/learn-electronics/flip-flop-rs-jk-t-d#d-flip-flop

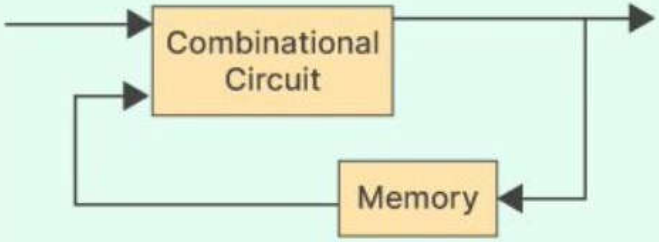
D Flip-Flop Applications

Some of the applications of D flip flop in real-world includes:

- **Shift registers:** D flip-flops can be cascaded together to create shift registers, which are used to store and shift data in digital systems.
  - Shift registers are commonly used in serial communication protocols
    - such as UART, **SPI**, and **I2C**.
- **State machines:** D flip-flops can be used to implement state machines, which are used in digital systems to control sequences of events.
  - State machines are commonly used in control systems, automotive applications, and industrial automation.
- **Counters:** D flip-flops can be used in conjunction with other digital logic gates to create binary counters that can count up or down depending on the design.
  - This makes them useful in real-time applications such as **timers** and **clocks**.
- **Data storage:** D flip-flops can be used to store temporary data in digital systems.
  - They are often used in conjunction with other memory elements to create more complex storage systems.

REF: https://www.electronicsforu.com/technology-trends/learn-electronics/d-flip-flop-circuit-truth-table-limitations-applications

Combinational vs Sequential Circuits

Combinational Circuit	Sequential Circuit
Output only depends on the present input	Output depends on present input & past output
Memory element is absent	Memory element is present
No clock signal is applied	Clock signal is required
	
Example - Half Adder, Full Adder, Multiplexer	Example - Flipflop, Counters, Registers

unstop

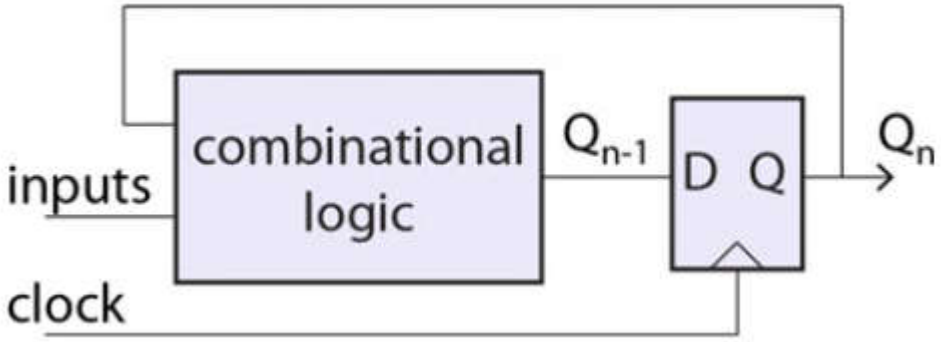
REF: https://unstop.com/blog/difference-between-combinational-and-sequential-circuit

Combinational Logic Design

- *Combinational logic* is implemented by using the **logic gates**
  - and in the *combinational logic*,
    - output is the function of
      - present input.

Sequential Logic Design

- Sequential logic is defined as the digital logic
  - whose output is a function of
    - present input
    - and past output.



Part 2

VERILOG HDL - OVERVIEW

Verilog HDL (and VHDL) => Hardware

- Verilog **HDL** is a **H**ardware **D**escription **L**anguage
  - Verilog HDL describes **hardware**
    - Things happen simultaneously or in parallel
      - whereas **software** is sequential
- So,
  - Verilog HDL is **not** a computer programming language

Signal Values in Verilog HDL

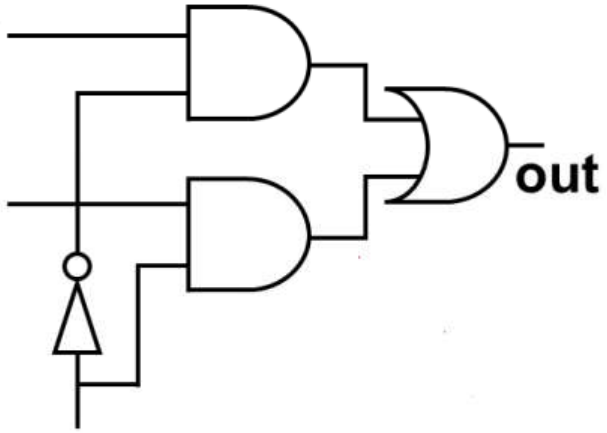
- Verilog signals can have 1 of 4 values: 0/1/x/z
- ***Actual hardware has levels 0 (điện áp thấp) and 1 (điện áp cao).***
- ***X and Z values might arise in simulation tools***
  - x is Unknown logical value
    - May be a 0, 1, z, in transition, or don't cares.
  - z means High impedance (trở kháng cao/không dẫn điện), floating (đầu dây không nối)

Combinational Logic

Continuous Assignment

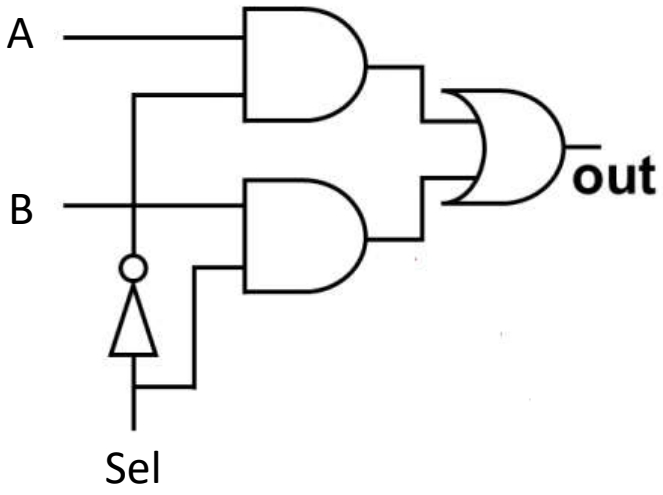
- An ***assign*** statement represents
  - continuously executing **combinational logic**.

- **assign out = … ;**



Combinational Logic with Continuous Assignment

```
1  module MUX2_1 (A, B, sel, out);
2
3  input A, B;
4  input sel;
5  output out;
6
7  assign out = (~sel & A) | (sel & B);
8
9  endmodule
```



***inputs* and *output* are *wire* variables *by default*.**

***assign* is used to set values for *wire* variables:**

- Left hand side (LHS) must be a ***wire*** type: '**out**'
- Right hand side (RHS) is recomputed when a value in the RHS changes
- The new value of the RHS is assigned to the LHS

Combinational Logic with Always Blocks

```
module MUX2_1 (A, B, sel, out);

input A, B;
input sel;
output reg out;

always @(A, B, sel)
    begin
        out = (~sel & A) | (sel & B);
    end
endmodule
```

Register type

```
module MUX2_1 (A, B, sel, out);

input A, B;
input sel;
output reg out;

always @(A, B, sel)
    begin
        if (sel == 0)
            out = A;
        else
            out = B;
    end
endmodule
```

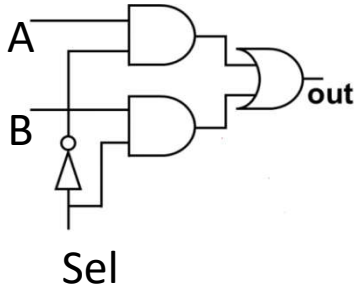
Conditional Statement

begin…end: Procedural Statement

Left hand side (**out**) inside the **always** block must be **reg** variable type

begin…end: Procedural Statement => similar to *conventional programming language* statements

Synthesis tool is able to infer a **MUX** based on Verilog code above



Logical Operators for Conditionals

&&	logical AND
||	OR
!	logical NOT
==	logical equality
!=	logical inequality
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal

Operator	Description
a && b	evaluates to true if a *and* b are true
a || b	evaluates to true if a *or* b are true
!a	Converts non-zero value to zero, and vice versa

Bitwise Boolean Operators

&	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

|	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

Concurrent Blocks and Other Elements in a Module

```
module name_of_module0 (…..);

input …;
output …;

assign … = ...;

always @(…)          <- Sensitivity List
    begin …
    end

always @(…)
    begin …
    end

name_of_module1 nom1 (...);

endmodule
```

- An ***always*** block executes **concurrently** with
  - other ***always*** blocks,
  - instance statements (other gates/modules), and
  - continuous assignment (***assign***) statements

  in a module.

Sensitivity List in Combinational Logic

- Includes every variable in an ***always*** block sensitivity list

  always @ (signal list)

- Another simple alternative

  always @ (*)

Sequential Logic

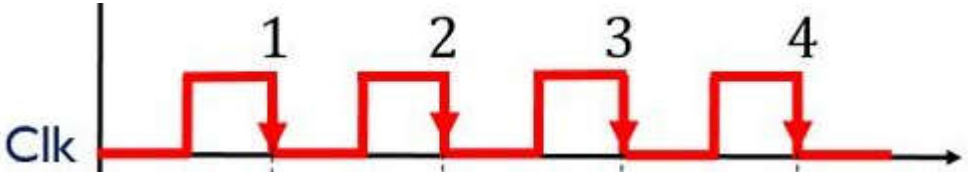
Sensitivity List in Sequential Logic

- Edge-triggered behavior

  always @ (posedge/negedge SIGNAL_name)

- Clock

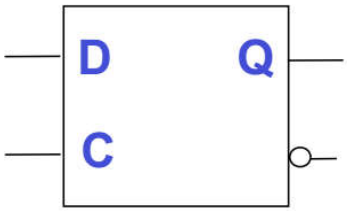
  always @ (posedge/negedge CLOCK_name)



Always Blocks

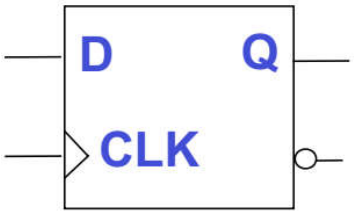
- **Sequential logic** can only be modeled using **always** blocks.
- Output **must** be declared as a "**reg**" variable type.

```
reg Q;

always @( clk, D )
    begin
        if ( clk )
            Q <= D;
    end
```



D latch

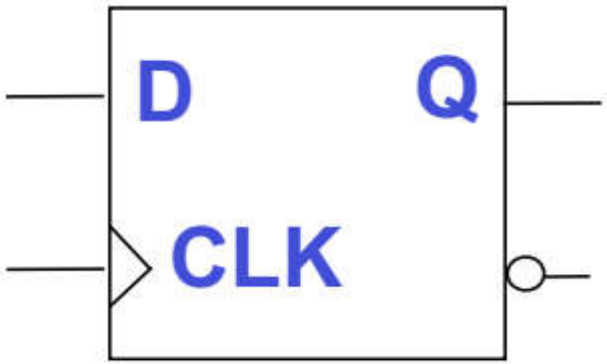
```
always @( posedge clk )
    begin
        Q <= D;
    end
```



DFF

Asynchronous Reset and Synchronous Reset

```
reg q;

always @ (posedge clk or posedge reset)
    if (reset==1)
        q <= 1'b0;
    else
        q <= d;
```



DFF

q is flip-flop in actual hardware

```
reg q;

always @ (posedge clk)
    if (reset==1)
        q <= 1'b0;
    else
        q <= d;
```

Blocking Assignment (=)

- Simulation tool behavior:
  - RHS is evaluated sequentially;
  - assignment to LHS is immediate.

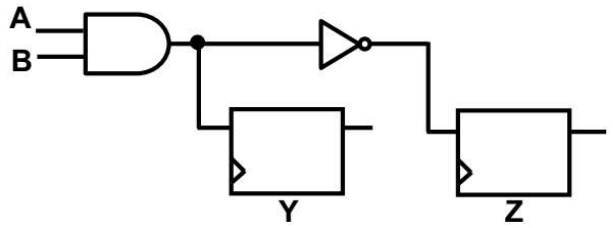
```
reg Y, Z;

always @ (posedge clk)
    begin
        Y = A & B;
        Z = ~Y;
    end
```

Simulator interpretation

$Y_{next} = A \& B$
$Z_{next} = \sim(A \& B)$

Y and Z are flip-flops in actual hardware

Resulting circuit (post synthesis)



Non-blocking Assignment (<=)

- Simulation tool behavior:
  - RHS evaluated in parallel (the order does not matter);
  - Assignment to LHS is delayed until end of **always** block.

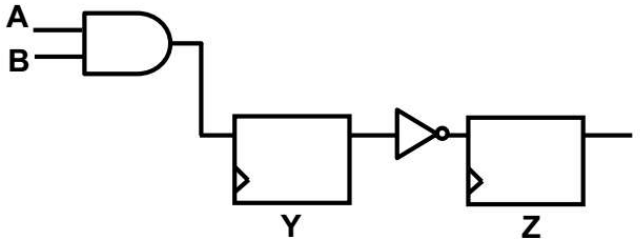
```
reg Y, Z;

always @ (posedge clk)
    begin
        Y <= A & B;
        Z <= ~Y;
    end
```

Simulator interpretation

Z_{next} = ~Y // reading the old Y

Y_{next} = A & B

Resulting circuit (post synthesis)



Assignment Note

- **Continuous** assignments apply to **Combinational** logic only
- **Always** blocks contain a set of procedural assignments (*blocking* or *non-blocking*)
  - Can be used to model
    - Either **Combinational** logic
    - Or **Sequential** logic.

Procedural Statement Note

- Procedural statement is similar to ***conventional programming language*** statements
  - **begin-end** blocks

    begin

        procedural-statement …

        procedural-statement

    end
  - **If**

    if (*condition*)

        procedural-statement

    else

        procedural-statement
  - **Case**

    case (*sel-expr*)

        choice : *procedural-statement* …

    endcase
  - **Blocking** assignment

    variable name **=** *expression* ;
  - **Non-blocking** assignment

    variable name **<=** *expression* ;

Combinational and Sequential Logic Note

- How to implement **Combinational** logic
  - output as a function of input
  - determined using logic gates

- **Sequential** logic
  - storage elements like latches, flip-flops

Part 3

VERILOG DESIGN DESCRIPTION

Coding Styles

- Structure
- **Behavioral**
- RTL

Structural Design of Half Adder

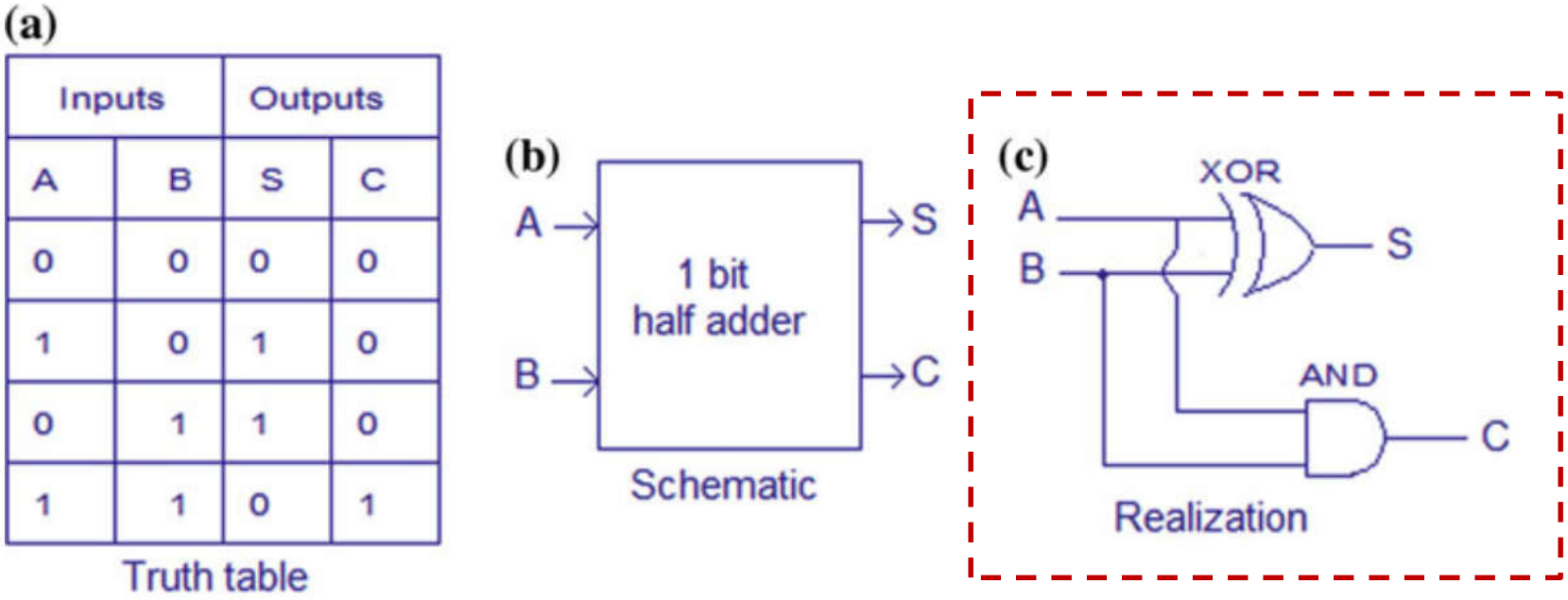


Fig. 1.4 Logic structure for "basic_Verilog"

Structural Code Style for "basic_verilog" module

```
// Verilog structural code style

module basic_verilog  (A,B,S,C) ;

input  A;
input  B;
output S;
output  C;
wire  A, B;
wire  S, C;

// Functionality of design

xor_gate  U1 ( .A(A), .B(B), .S(S));
and_gate  U2 (.A(A), .B(B), .C(C));

endmodule
```

List of inputs and outputs

Declare Verilog module 'basic_verilog' with input ports 'A', 'B' and output ports 'S', 'C'

Declarations

Instantiation of the components xor gate and and_gate" . It is assumed that the precompiled xor_gate and and_gate is available in the library.

Statements

Behavior Code Style

```
// Verilog behavior code style

module basic_verilog  (A,B,S,C) ;

input  A, B;

output  S, C;

reg  S , C;

// Functionality of design

always@(A or B)

if (A==B)

S= 1'b0;

else

S=1'b1;

always@(A or B)

if (A & B)

C=1'b1;

else

C=1'b0;

endmodule
```

Declare verilog module 'basic_verilog' with input port s 'A', 'B' and output ports 'S', 'C'

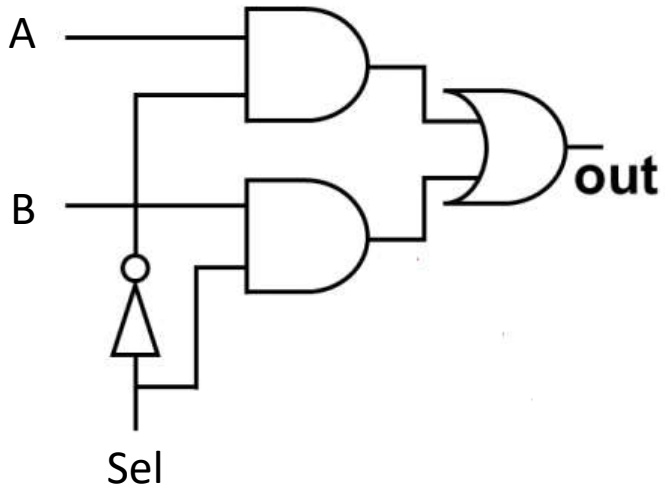
Sensitive to 'a' or 'b' and described in sensitivity list.

The description using 'if-else' statement describes the output assignment to 'S'. When both inputs 'A', 'B' are at same logic level output assigned to 'S' is logical '0' else output assignment to 'S' is logical '1'

The description using 'if-else' statement describes the output assignments to 'C'. When both 'A', 'B' are at logic '1' level then output assigned to 'C' is logic '1' else output assigned to 'C' is logic '0'.

Behavior Code Style (2): Continuous Assignment

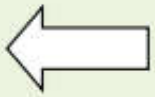
```
1  module MUX2_1 (A, B, sel, out);
2
3  input A, B;
4  input sel;
5  output out;
6
7  assign out = (~sel & A) | (sel & B);
8
9  endmodule
```



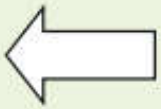
Combinational Logic

RTL Code Style

```
// Verilog synthesizable RTL code style

module basic_verilog  (A,B,S,C) ;

input  A;

input B;

output S;

output C;

reg   S;

reg  C;

// Functionality of design

always s@ (A or B)

begin

S= A ^ B;

C= A & B;

end

endmodule
```



Declare Verilog module 'basic_verilog' with input ports 'A', 'B' and output ports 'S', 'C'.



Description of outputs 'S', 'C' in the form of logical expressions and used for less complex designs!

Declarations

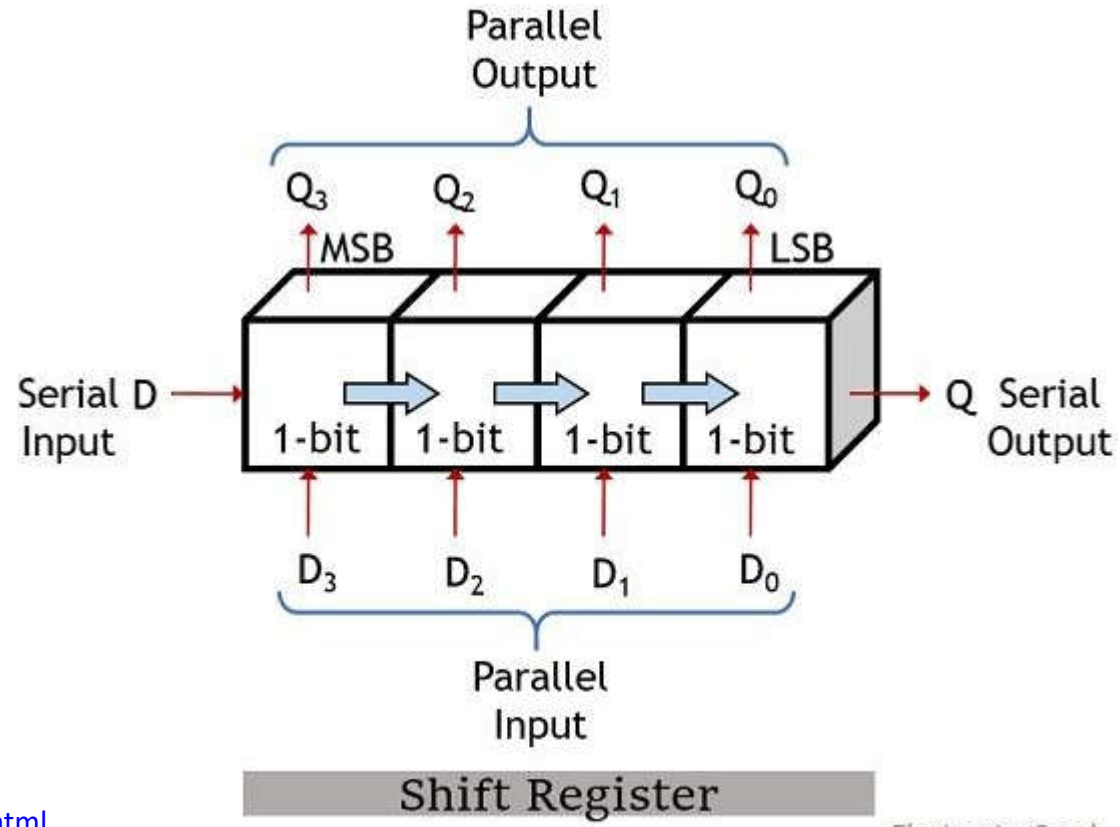
- **E.g., input [3:0] Bus;**
  - This would be a 4-bit bus,
    - with individual **wire** names
      - Bus[3] (MSB),
      - Bus[2],
      - Bus[1],
      - Bus[0] (LSB)

Part 4

SHIFT REGISTER

Shift Register

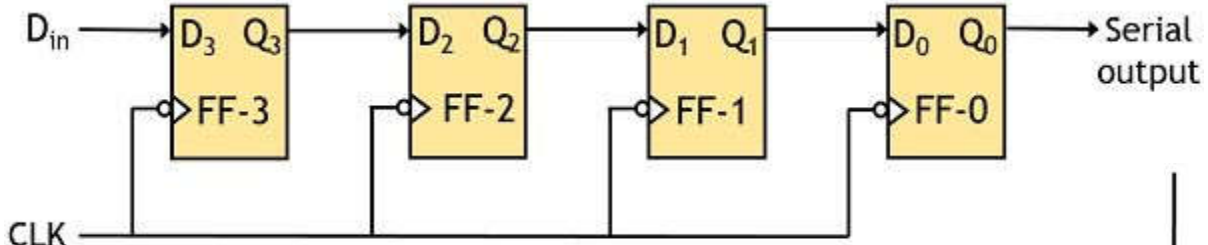
- A shift register is a **sequential logic circuit**
  - that acts as a unit to store and transfer binary data.
- Basically shift registers are **bidirectional FIFO circuit**,
  - that shifts every single bit of the data present in its input
    - towards its output on each clock pulse.



REF: https://electronicscoach.com/shift-register.html

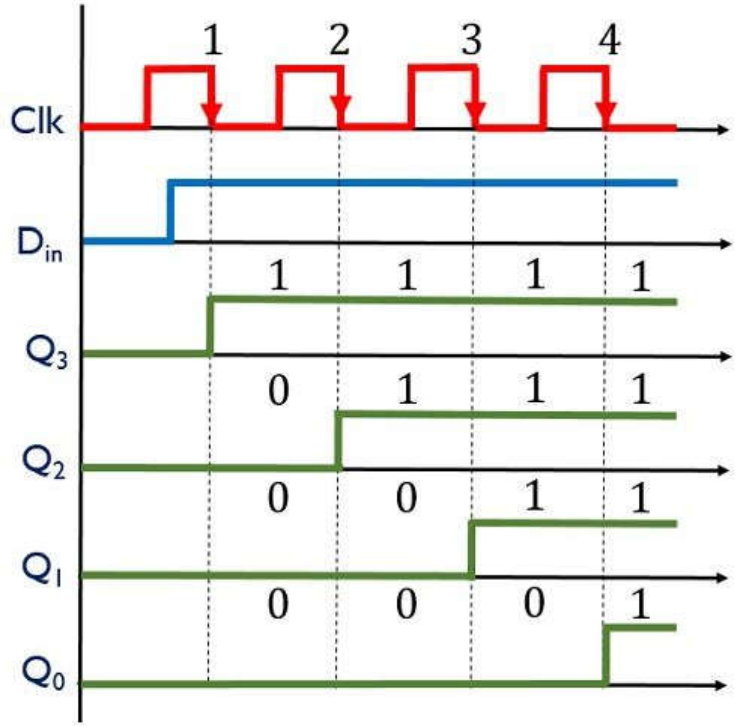
Operation of SISO Shift Register

- REF: https://electronicscoach.com/shift-register.html



4-bit SISO Shift Register

CLK	Q_3	Q_2	Q_1	Q_0
Initially	0	0	0	0
1st falling edge	1	0	0	0
2nd falling edge	1	1	0	0
3rd falling edge	1	1	1	0
4th falling edge	1	1	1	1



Waveform Representation of SISO Shift Register

Electronics Coach

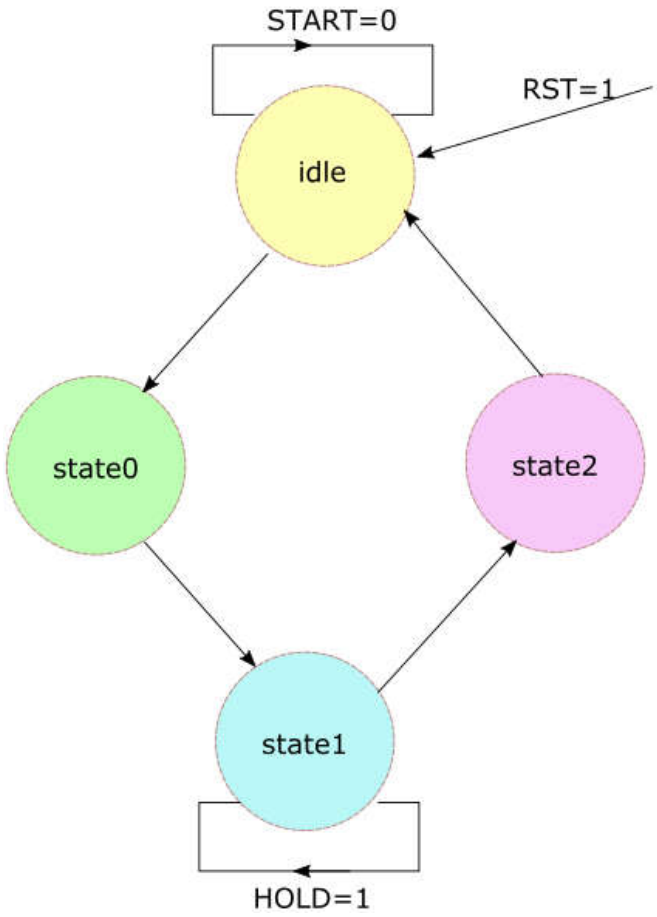
Part 5

FINITE STATE MACHINES (FSM)

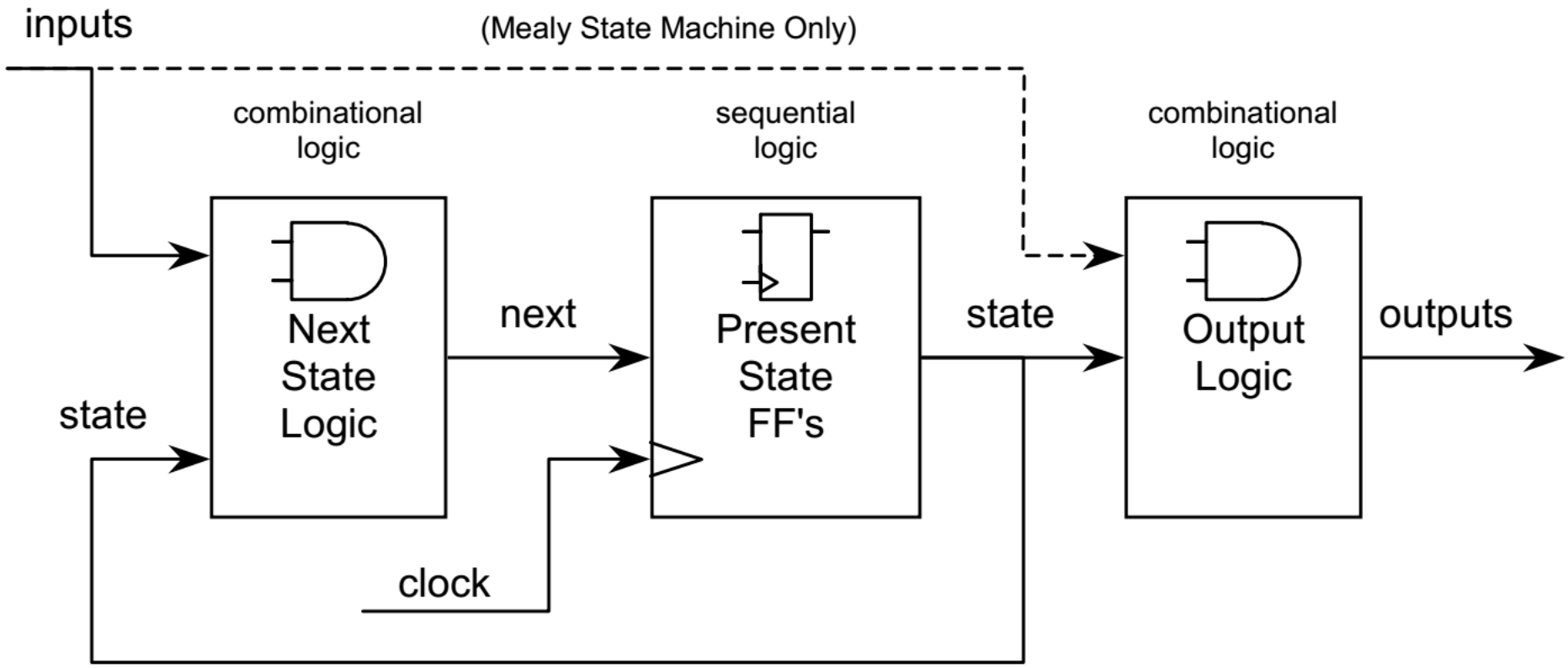
FSM Applications

- Most of the RTL design needs accurate **timing and controlling algorithms**.
- Finite state machines (FSM) are used to implement the control and timing algorithms.
- Finite state machines can be coded by using different encoding styles.
  - These encoding styles are
    1. **binary**,
    2. **gray**
    3. and **one-hot** encoding.

State	Encoding
idle	00
state0	01
state1	10
state2	11



Mealy and Moore State Machine



Difference b/w Moore and Mealy

Table 8.1 Differences between Moore and Mealy machines

Moore machine	Mealy machine
Outputs are function of current state only	Outputs are function of the current state and inputs also
As output is the function of current state it is stable for one clock cycle	Output is the function of current state and inputs so it may change during the state and hence may or may not be stable for one clock cycle
Output is stable for one clock cycle and not prone to glitches or spikes	Output may change multiple times depending on changes in the input and hence prone to glitches or hazards
It requires more number of states compared to Mealy machine	Mealy machine needs at least one state less compared to Moore machine
STA is easy as combinational paths between the registers are shorter	STA is complex as combinational paths are relatively larger area compared to Moore machine
Higher operating frequency compared to Mealy machine	Less operating frequency compared to Moore machine

Moore and Mealy

- Both Moore and Mealy FSMs have been successfully implemented in digital designs.
- **How the outputs are generated** for these state machines is an interesting topic.
  - Outputs are sometimes generated by combinational logic based on comparisons with a set of states,
  - and sometimes outputs can be derived directly from individual state bits.

Two-Always-Block FSM Style

- One of the Verilog coding styles is to code the FSM design
  - using ***two always blocks***,
    - one for the ***sequential state register***
    - and one for the ***combinational next-state*** and *combinational output logic*.

```
3   module fsm1a (ds, rd, go, ws, clk, rst_n);
4
5   output ds, rd;
6   input go, ws;
7   input clk, rst_n;
8
9   parameter [1:0] IDLE = 2'b00,
10                  READ = 2'b01,
11                  DLY = 2'b10,
12                  DONE = 2'b11;
13  reg [1:0] state, next;
14
15  always @(posedge clk or negedge rst_n)
16      if (!rst_n) state <= IDLE;
17      else state <= next;
18
19  always @(state or go or ws) begin
20      next = 2'bx;
21      case (state)
22          IDLE:   if (go) next = READ;
23                  else next = IDLE;
24          READ: next = DLY;
25          DLY:    if (ws) next = READ;
26                  else next = DONE;
27          DONE: next = IDLE;
28      endcase
29  end
30
31  assign rd = (state==READ || state==DLY);
32  assign ds = (state==DONE);
33
34  endmodule
```

State register, sequential always block

Next state, combinational always block

Continuous assignment outputs

```
1  module fsm1 (ds, rd, go, ws, clk, rst_n);
2
3  output ds, rd;
4  input go, ws;
5  input clk, rst_n;
6
7  reg ds, rd;
8  parameter [1:0] IDLE = 2'b00,
9                  READ = 2'b01,
10                 DLY = 2'b10,
11                 DONE = 2'b11;
12 reg [1:0] state, next;
13
14 always @(posedge clk or negedge rst_n)
15     if (!rst_n) state <= IDLE;
16     else state <= next;
17
18 always @(state or go or ws) begin
19     next = 2'bx;
20     ds = 1'b0;
21     rd = 1'b0;
22     case (state)
23         IDLE:   if (go) next = READ;
24                 else next = IDLE;
25         READ: begin
26                   rd = 1'b1;
27                   next = DLY;
28               end
29         DLY:  begin
30                 rd = 1'b1;
31                 if (ws) next = READ;
32                 else next = DONE;
33               end
34         DONE: begin
35                 ds = 1'b1;
36                 next = IDLE;
37               end
38     endcase
39 end
40
41 endmodule
```

State register, sequential always block

Next state & outputs, combinational always block

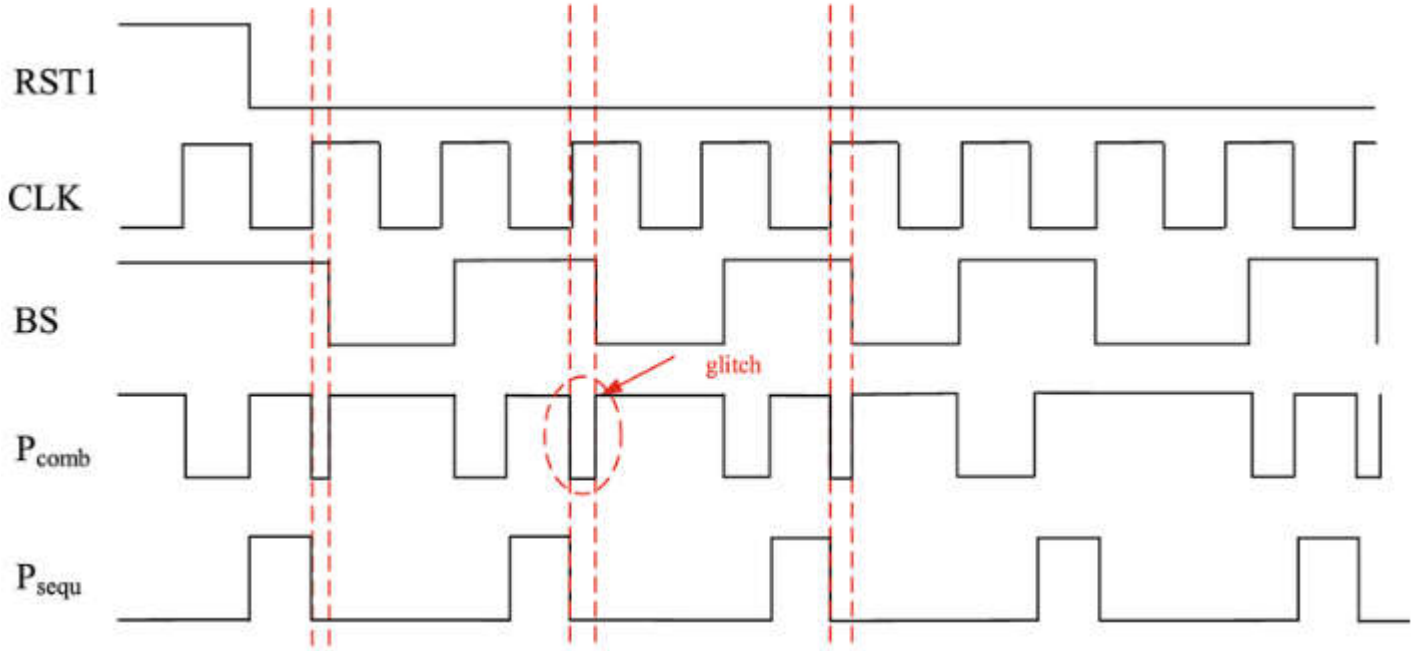
```
1  module fsm_4states (output reg gnt,
2                      input dly, done, req, clk, rst_n);
3
4  parameter [1:0] IDLE = 2'b00,
5                  BBUSY = 2'b01,
6                  BWAIT = 2'b10,
7                  BFREE = 2'b11;
8  reg [1:0] state, next;
9
10 always @(posedge clk or negedge rst_n)
11     if (!rst_n) state <= IDLE;
12     else state <= next;
13
14 always @(state or dly or done or req)
15     begin
16         next = 2'bx;
17         gnt = 1'b0;
18         case (state)
19             IDLE :  if (req) next = BBUSY;
20                     else next = IDLE;
21             BBUSY:  begin
22                         gnt = 1'b1;
23                         if (!done) next = BBUSY;
24                         else if (dly) next = BWAIT;
25                         else next = BFREE;
26                     end
27             BWAIT:  begin
28                         gnt = 1'b1;
29                         if (!dly) next = BFREE;
30                         else next = BWAIT;
31                     end
32             BFREE:  if (req) next = BBUSY;
33                     else next = IDLE;
34         endcase
35     end
36
37 endmodule
```

State register, sequential always block

Next state & outputs, combinational always block

Note (2)

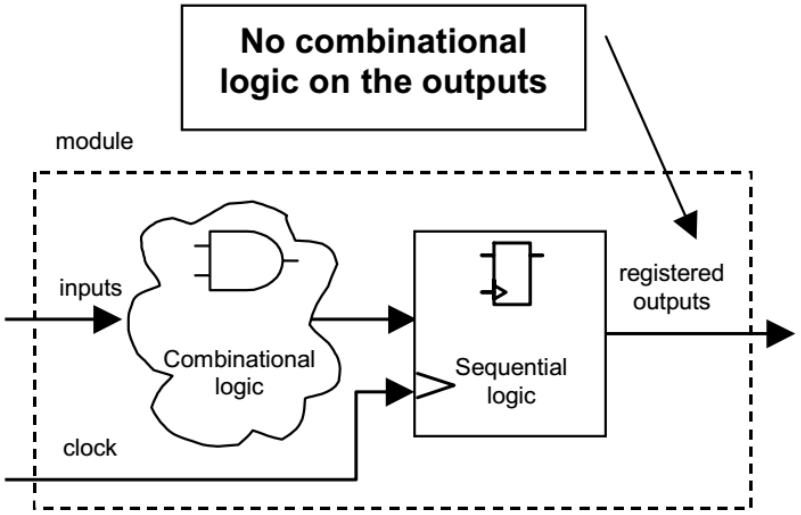
- The combinational outputs generated by the two coding styles (last slides) suffer two principal disadvantages:
  - 1. Combinational outputs can **glitch** between states.
  - 2. Combinational outputs consume **part** of the overall clock cycle
    - that would have been available to the block of logic that is driven by the FSM outputs.

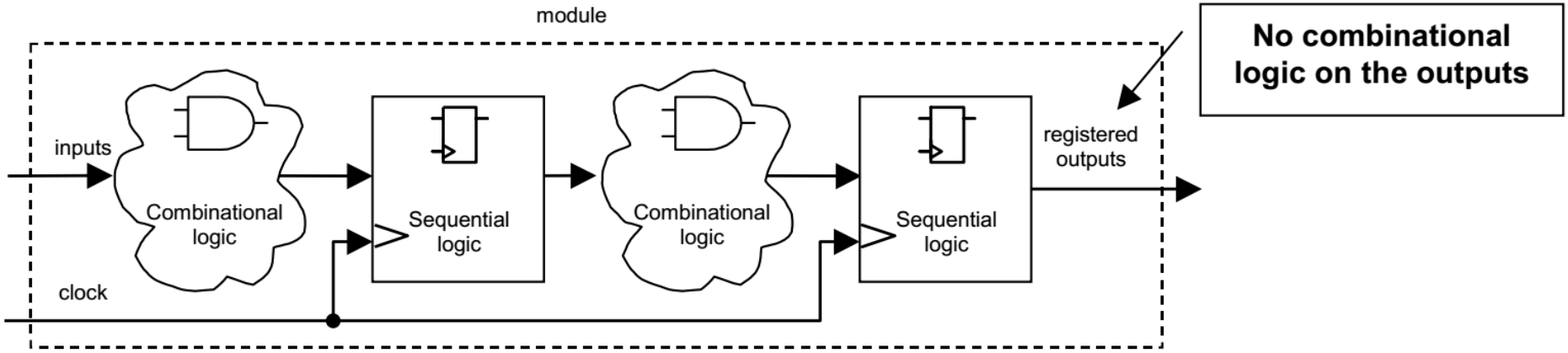


Note

- In addition to disadvantages of FSM mentioned on the last slide:
  - When module outputs are generated using combinational logic,
    - there is less time for the receiving module
      - to pass signals through inputs and additional combinational logic
        - » before they must be clocked.

- **Solution**:





Registering FSM Outputs

```
1  module fsm1b (ds, rd, go, ws, clk, rst_n);
2  output ds, rd;
3  input go, ws;
4  input clk, rst_n;
5  reg ds, rd;
6  parameter [1:0] IDLE = 2'b00,
7                  READ = 2'b01,
8                  DLY = 2'b10,
9                  DONE = 2'b11;
10 reg [1:0] state, next;
11
12 always @(posedge clk or negedge rst_n)
13     if (!rst_n) state <= IDLE;
14     else state <= next;
15
16 always @(state or go or ws) begin
17     next = 2'bx;
18     case (state)
19         IDLE:   if (go) next = READ;
20                 else next = IDLE;
21         READ: next = DLY;
22         DLY:    if (ws) next = READ;
23                 else next = DONE;
24         DONE: next = IDLE;
25     endcase
26 end
27
28 always @(posedge clk or negedge rst_n)
29     if (!rst_n) begin
30         ds <= 1'b0;
31         rd <= 1'b0;
32     end
33     else begin
34         ds <= 1'b0;
35         rd <= 1'b0;
36         case (state)
37             IDLE: if (go) rd <= 1'b1;
38             READ: rd <= 1'b1;
39             DLY:    if (ws) rd <= 1'b1;
40                     else ds <= 1'b1;
41         endcase
42     end
43 endmodule
```

State register, sequential always block

Next state, combinational always block

Registered outputs, sequential always block

Part 6

CODING NOTES

Note for UUT and Test Bench

Rules

- Use ***blocking*** *assignments* (=) to model *combinational logic* within an **always** block.
  - or just implement combinational logic
    - without an **always** block,
    - using assign statements
- Use ***non-blocking*** *assignments* (<=) to implement *sequential logic*.
- Do not mix ***blocking*** and ***non-blocking*** *assignments*
  - in the same **always** block.
- Do not make *assignments* to the same variable
  - from more than one **always** block

Main Variable Types

- ***wire***: represents a physical connection (net) between hardware elements
  - Used for *structural style* and *continuous assignments*
  - ***Default*** for input/output ports (if you do not specify a type as ***wire/*reg**)
  - Can only be used to model **combinational logic**
  - Cannot be used in the left-hand side (LHS) in an ***always*** block

- ***reg***: a variable that can be used to store state (registers)
  - Used in the procedural code (***always*** blocks)
  - May also be used to express combinational logic
  - Can be used to model both **combinational & sequential logic**
  - Cannot be used in the left-hand side of a continuous assignment statement

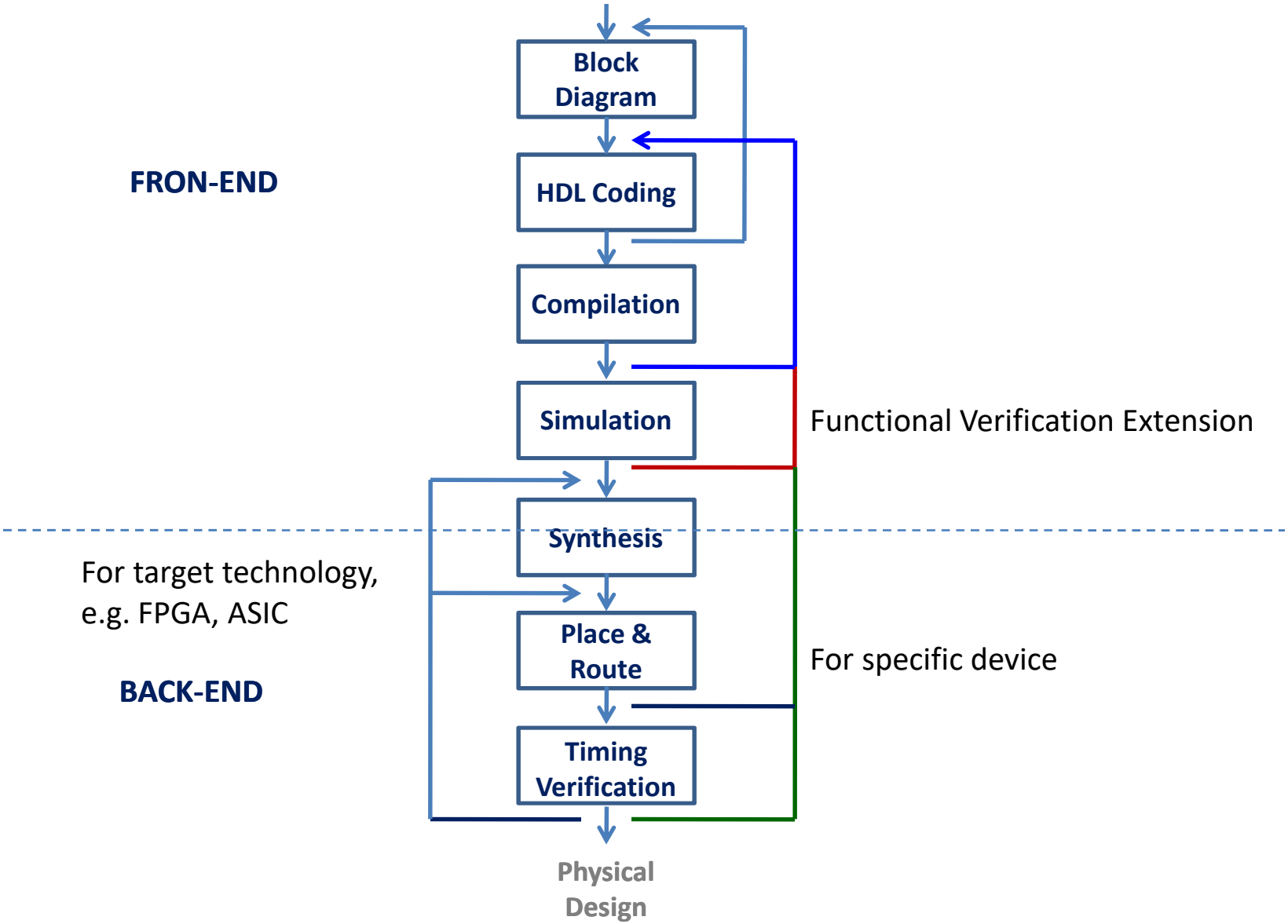
Full_case / parallel_case

- case (case_expression (with 2^n possible combinations))
  case_item1 : <action #1>;
  case_item2 : <action #2>;
  case_item3 : <action #3>;
  ...
  case_item2n-1: <action #2n-1>;
  case_item2n: <action #2n>;
  default: <default action>;
  endcase

Part 7

AN EXAMPLE OF DESIGN AND TESTBENCH/FUNCTIONAL VERIFICATION

Logical Design Flow



Simple Design Flow (for beginning) and EDA Tools

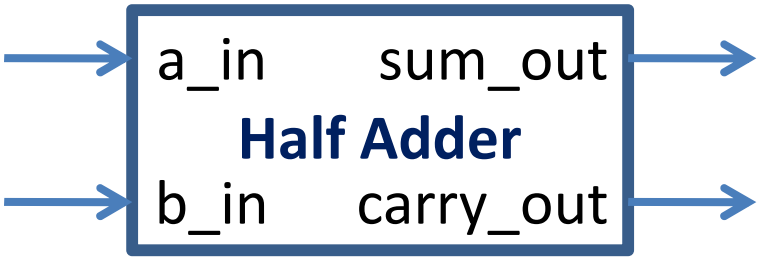
- The design process (e.g., starts from RTL) as follows:
  - **RTL** (Register Transfer Level) Verilog HDL Coding
  - **Simulation** (RTL): needed to develop a test bench (Verilog).
    - **Modelsim/Questasim,** Cadence **VerilogXL/NCSim/Xcelium,** Synopsys **VCS, Open-sources EDA tools**
  - Synthesis (Gate level netlist)
    - Although there are a variety of software tools available for synthesis (such as **LeonardoSpectrum**, **Synplify,** Cadence **Genus,** Synopsys **Design Compiler**),
      - they all have generally
        - » similar approaches and design flows.
  - Post-synthesis Simulation
  - Timing Simulation (Cadence **Tempus**, Synopsys **PrimeTime**) to check timing
  - etc.

Cadence and Synopsys

- Both of them are used
- A lot of companies have mixed flows
- E.g.,
  - Synopsys Design Compiler
  - Then, Cadence Innovus for all the place and route.
  - Then, Primetime for STA signoff.
  - **Siemens/Mentor** Calibre is almost always the LvS/DRC signoff tool.

Test Bench

- Verilog test benches **DO NOT** describe hardware
- Verilog test benches look like a **software program**



UUT – Unit Under Test

(DUT/EUT)

Half Adder - UUT

```
//Verilog RTL code for half adder

module half_adder ( a_in, b_in, sum_out,carry_out);

input a_in;

input b_in;

output sum_out;

output carry_out;

wire sum_out;

wire carry_out;

assign sum_out = a_in ^ b_in;

assign carry_out = a_in & b_in;

endmodule
```

Output sum_out is assigned as XOR of 'a_in', 'b_in'. XOR is summation operation.

Output carry_out is assigned as AND of 'a_in', 'b_in'. Carry logic is AND operation.

a_in	b_in	sum_out	carry_out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

General Items of a Test Bench

- Timescale
  - Specifies the simulation granularity
    - Syntax

```
`timescale time_unit base / precision base
```

unit of delays *for fractional delay*

```
`timescale 1 ps / 1 ps
```

- Module definition
  - A testbench is a module, but without inputs or outputs
    - E.g., **module cnt4bit_tb();**
- UUT instantiation
- Stimulus generation
- Monitoring output

General Items of a Test Bench: Connection Declaration

- Declare input and output signals that will link/connect to the UUT:

```
// feeding signals connected at inputs of UUT
        reg a;
        reg b;
// connecting to UUT outputs to monitoring
        wire sum;
        wire carry;
```

UUT Instantiation

```
half_adder UUT ( // module instantiation
                 .a_in(a),
                 .b_in(b),
                 .sum_out(sum),
                 .carry_out(carry)
                 );
```

Stimulus Generation: Clock

```
`timescale 1 ps / 1 ps
```

- **// e.g., generate a 50MHz clock**

```
always
    begin
        CLK50 = 1'b0;
        CLK50 = #10000 1'b1;
        #10000;
    end
```

- *#10000 means a delay i.e. wait for 10000 time units before changing the value of CLK50. The total time period is 20000 time units*

Test Bench using 'initial' block

```
`timescale 1 ns /10 ps // time unit = 1 ns, precision = 10 ps

module half_adder_tb;

reg a, b;
wire sum, carry;

// duration for each bit = 20 * timescale = 20 * 1 ns = 20ns
localparam period = 20;

half_adder UUT (.a_in(a), .b_in(b), .sum_out(sum), .carry_out(carry)); // module instantiation

initial // initial block executes only once
    begin
        // values for a and b
        a = 0;
        b = 0;
        #period; // wait for period

        a = 0;
        b = 1;
        #period;

        a = 1;
        b = 0;
        #period;

        a = 1;
        b = 1;
        #period;
    end
endmodule
```

```
half_adder UUT ( // module instantiation
                 .a_in(a),
                 .b_in(b),
                 .sum_out(sum),
                 .carry_out(carry)
                 );
```

Test Bench with ‘always’ block

```
1  // half_adder_procedural_tb.v
2
3  `timescale 1 ns/10 ps // time-unit = 1 ns, precision = 10 ps
4
5  module half_adder_procedural_tb;
6
7  reg a, b;
8  wire sum, carry;
9
10 // duration for each bit = 20 * timescale = 20 * 1 ns = 20ns
11 localparam period = 20;
12
13 half_adder UUT (.a(a), .b(b), .sum(sum), .carry(carry));
14 reg clk;
15
16 // note that sensitive list is omitted in always block
17 // therefore always-block run forever
18 // clock period = 2 ns
19 always
20     begin
21         clk = 1'b1;
22         #20; // high for 20 * timescale = 20 ns
23
24         clk = 1'b0;
25         #20; // low for 20 * timescale = 20 ns
26     end
```

Test Bench with 'always' block (2)

```
28  always @(posedge clk)
29      begin
30          // values for a and b
31          a = 0;
32          b = 0;
33          #period; // wait for period
34          // display message if output not matched
35          if(sum != 0 || carry != 0)
36          $display("test failed for input combination 00");
37
38          a = 0;
39          b = 1;
40          #period; // wait for period
41          if(sum != 1 || carry != 0)
42          $display("test failed for input combination 01");
43
44          a = 1;
45          b = 0;
46          #period; // wait for period
47          if(sum != 1 || carry != 0)
48          $display("test failed for input combination 10");
49
50          a = 1;
51          b = 1;
52          #period; // wait for period
53          if(sum != 0 || carry != 1)
54          $display("test failed for input combination 11");
55
56          a = 0;
57          b = 1;
58          #period; // wait for period
59          if(sum != 1 || carry != 1)
60          $display("test failed for input combination 01");
61
62          $stop; // end of simulation
63      end
64  endmodule
```

BACKUP

Thank You