ECE 270 : Embedded Logic Design ⇒ CO + DC

Resource for Quiz : hdlbits.01xz.net/wiki/Main_Page
Lab videos : youtube

Programming : 1st half → Verilog
2nd half → Embedded C

Theory : FPGA and SOC

Vivado 2019.1 (including SDK)
arch user repository ⇐ AUR ✓

* GRADES:

mid sem	30 %
end sem	30 %
surprise quiz	25 %
lab hw	15 %

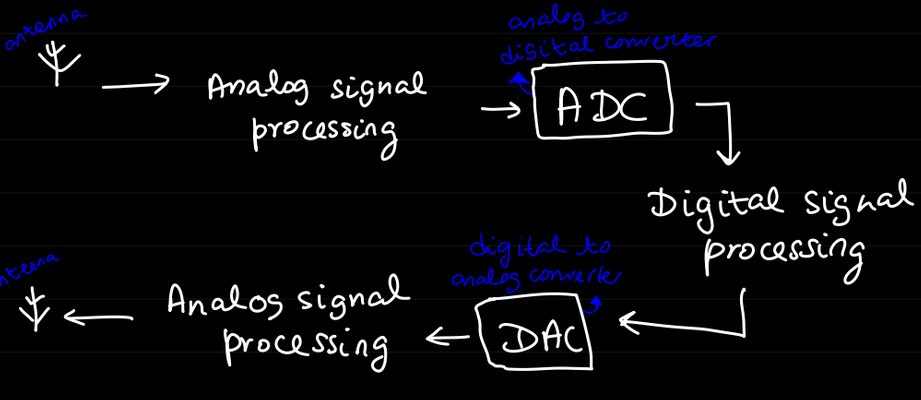
LECTURE : 1

* Which is faster : Analog vs digital ?

⇒ Depends on the use case

* No product is purely digital / analog ?

⇒ Analog is present in nature however digital can be processed easily and has more use cases.



HDL → Hardware Description language
↳ eg ⇒ Verilog

LECTURE : 2

* combinational circuit : The output depends upon the present input (same clock cycle)

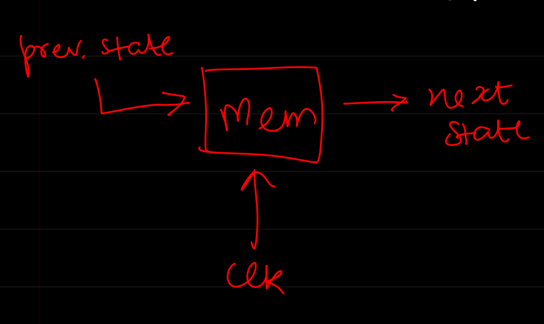
* sequential circuit : output depends upon the current input and the current state of the circuit

you need current state of circuit
need to store state
hence we use flip flops
Note: we need not store current input

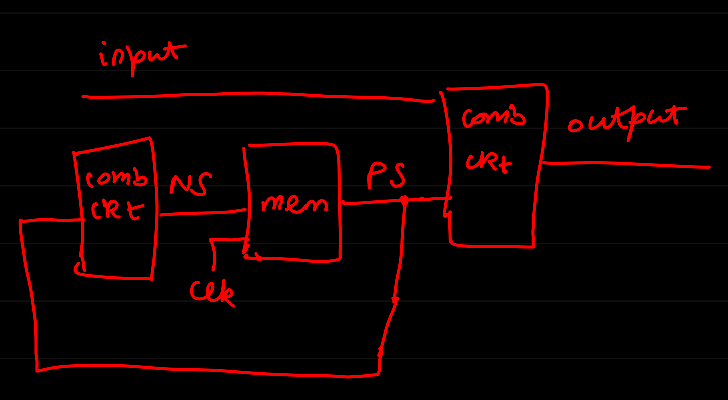
what we get → output + next state
because we go from one state to another

⇒ Note: combinational circuits use clock as its an input as well.

* D flip flop : input is stored at falling or rising edge of the clock
(edge triggered)
next state becomes current state

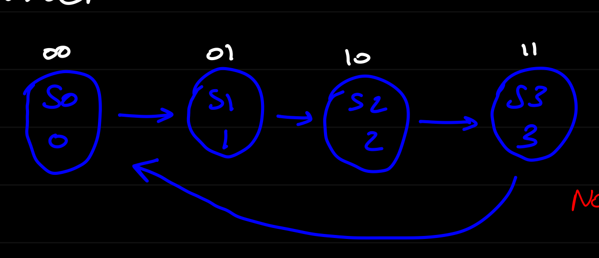


* Sequential circuit using combinational ckt



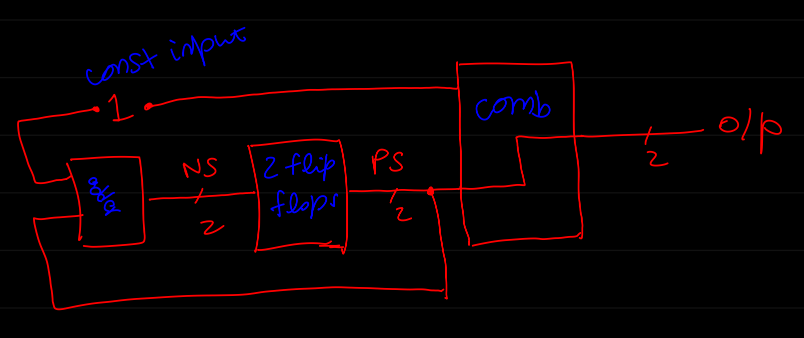
* FSM (finite state machine)

⇒ up counter

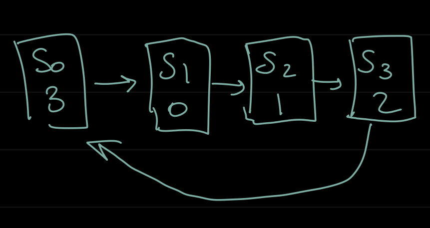


Note: if curr state = S_n, the output is n

2 bits required to store in mem

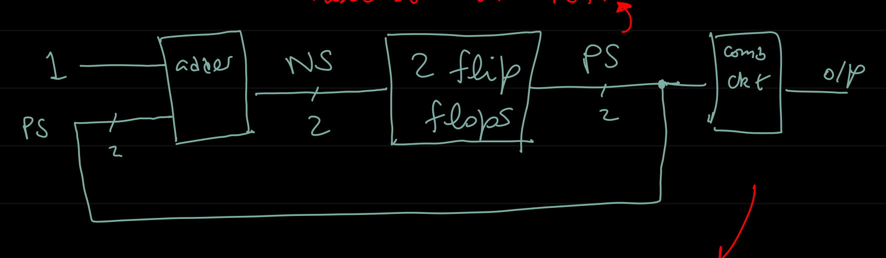


* Another example



S_0 = 00
S_1 = 01
S_2 = 10
S_3 = 11

relation btw present state and next state ⇒ NS = PS + 1



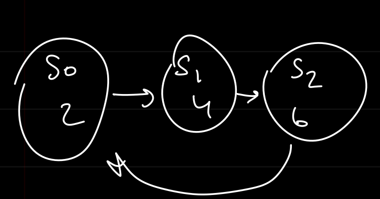
Here we should use 2 not gates for the 2 bits

eg: if PS = S_0 (00) output is 3 (11)
i.e. NOT(0), NOT(0) ⇒ (11)

So, we are storing current state + processing input (using comb ckt)

* Example : 3

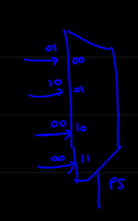
counter : 2 → 4 → 6



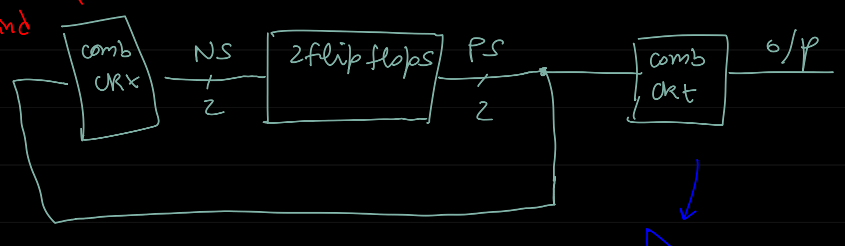
S_0 00
S_1 01
S_2 10

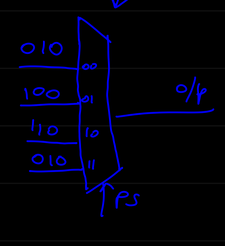
we are not storing 2/4/6 {output}
we are storing S_0/S_1/S_2 {state}

since we represent states using 2 bits, we need 2 flip flops



use either mux or K-map to find suitable ckt





NOTE: no. of states is equal to no. of flip flops

* Any combinational circuit can be implemented using memory **LUT**
↳ lookup table

In DC, we learnt that we can implement any comb. ckt using gates.

However implementing it using memory is more versatile since we can always change memory (i.e. it is dynamic)

conclusion,

Combinational ckts can be implemented using...

① Pure gates
② Lookup Table

* **FPGA** Field Programmable Gate Array
⇒ doesn't has gates
we use lookup tables to implement logic

→ Array of generic logic gates

→ Gates where logic function can be programmed

→ Programmable interconnection btw gates

→ Field : system can be programmed in the field (after fabrication)

CLB : configurable logic blocks
BUFG : global buffer
BUFR : regional "
BUFIO : input/output "

* comb CKts on FPGA
≡ Generic FPGA Design Flow

Requirements
↓
Design specifications
↓
Design Formulation
↓
Design Entry { includes: block diagram* — specification / pseudo code, hand drawn block diagram
Hand-drawn schematics, VHDL, verilog
↓ " CAD ⇒ computer aided design
Behavioural Simulation : done using test benches / waveforms — ensures that the design is functionally correct } doesn't guarantee the code will work on the hardware (or self checking test bench, test bench)
↓
Logic Synthesis ⇒ input VHDL description → output circuit netlist (gate level circuit representation) — automatically converts code to visual representation (instead of manual k-map)
↓
Post Synthesis Simulation ⇒ we use the same test bench and test the netlist for its functionality — the delay in the circuit is also calculated at this step
↓ output ⇒ NCD file (native circuit desc)
mapping, placement, routing — Implementation: maps the logic defined by an NGD file into FPGA elements; compares gate level circuit & FPGA circuit — state of all the blocks in the FPGA, which and how I gave use; connecting different blocks (eg. CLB) so that data flows from one to another depending on the logic ⇒ how the components are connected with each other — ↳ location of diff components in the circuit in consideration with its placement requirements (eg. component temp)
↙ ↘
ASIC / SoC FPGA

NGD : Native Generic Database
NCD : Native circuit description
delay : Logic delay + Routing delay
= combinational ckt delay
hence we need to run through the simulation again after Routing in order to verify functionality and timing constraints
post-place and route static timing

If you identify problems in the timing report
- increase the placer effort level
- using re-entrant routing
- using multi-pass place and route
- redesign the logic paths to use fewer levels of logic, moving to a faster device, or allocate more time for the paths.

Final Step : generating the bitstream (*.bit)

downloaded directly to the FPGA OR converted to a PROM file which contains programming information.

* Lecture : 3

Sat, Aug 17 3→4:30pm

- **ASIC design flow:** routed design is used to generate photomask for producing integrated circuits

Application specific integrated chip

FPGA	ASIC
The application can be configured and changed even after fabrication	fabricate the chip for one specific application
VHDL code can be converted to .bit and downloaded directly to FPGA	we have to send our VHDL code to the fabrication organization
	90% devices are ASIC as of this moment

- **APPLICATION SPECIFIC IC ⇒ ASIC**

High non-recurring engineering cost (NRE)
{the cost required for fabrication of the FIRST product}
one time cost to research, design, develop and test a new product

High cost for engineering change orders
hence testing is critical

lowest price for high volume production

fastest clock performance (high performance)
because ASIC is intended for specific application

unlimited size and low power consumption

design and test tools are expensive

expensive IPs

steep learning curve

- **Field Programmable Gates Array: FPGA**

lowest cost for low to medium volume

No NRE (non recurring engineering) cost and fastest time to market

Field reconfigurable and partial reconfigurable
≡ upgradable ✓

Slower performance than ASIC

Limited size and steep learning curve

Digital only

Industry often use FPGAs to prototype their chips before creating them
(FPGA → then ASIC)

* MICRO-CONTROLLERS

simple computer placed inside a single chip with all the necessary components like memory, timers etc., embedded inside and performs a specific task

↳ sequential execution
commands: one by one
one task at a time

Cannot carry out parallel operation

A micro controller designed for parallel operations ≡ GPU

consumes less power than FPGA and suitable for edge cases

* MICRO-PROCESSORS

ICs that come with a computer or CPU inside and are equipped with processing power.

No peripherals like Rom, memory

CPU GPU ASIC

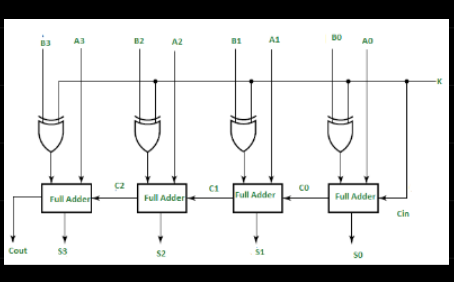
flexibility ⟷ efficiency

solution for the near future ≡ ARM + FPGA + GPU

microcontroller : time limited
FPGA : space limited

8:30 → 11

* LAB : 1 Design of Full Adder 20/08/24



In Hardware, execution occurs parallely whereas in software, we write programs that run sequentially

Two major HDLs : Verilog & VHDL

Verilog
- more popular
- syntax close to C
- easy to master
- more prominent in Indian VLSI industry
- 1983 : introduced by Gateway Design System
- Invented as a SIMULATION language. SYNTHESIS was an afterthought.
- 1987 : Verilog Synthesizer by Synopsis
- 1989 : Gateway Design System acquired by Cadence
- Latest verilog version = System Verilog
  - ↳ much simpler than initial versions

VHDL
- 1981 – 1983 : US Dept of defence developed VHDL (VHSIC HDL)
  - VHSIC : very high speed integrated circuit hardware description language
- open source unlike Verilog (closed src)
- Afraid of losing market share, Cadence made Verilog open sourced (1990)
- 1995 : became IEEE Standard 1364

↓ implementation of placement, routing of generating bitstream

Hardware : parallel processing
Software : sequential processing

In verilog, all lines execute parallely unlike languages like, C, Hs etc.

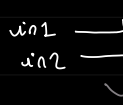
Verilog looks like C but describes hardware

Understand the circuit and specifications then figure out the code

* VERILOG

- Verilog HDL is case sensitive
- all keywords are in lower case
- statements terminated by semi-colon ;
- Two data types :
  - Net (wire) → default data type
  - variable (Reg, Integer, real, time, realtime)
- Primitive Logic Gates and Switch-Level gates are built in

* EXAMPLES



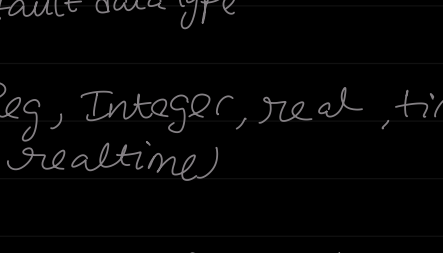
in 1	in 2	out
0	0	0
0	1	0
1	0	0
1	1	1

```
// <module_name> <ports>
module AND (out, in1, in2);
    input in1, in2;
    output wire out;
    // in1 and in2 are also
    // wire data type since
    // it is default type
    assign out = in1 & in2;
    // data flow - continuous assignment
endmodule
```

* 4 BIT FULL-ADDER

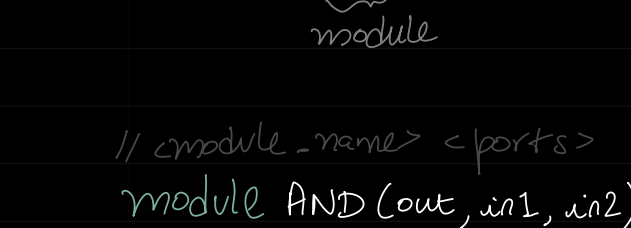
⇒ Half Adder

x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



⇒ Full adder

C_i	x_i	y_i	C_{i+1}	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

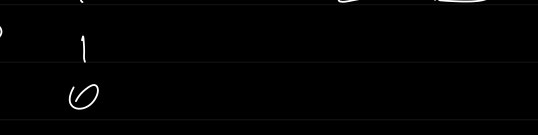



```
module full_adder_1bit (
    input FA1_InA,
    input FA1_InB,
    input FA1_InC,
    output FA1_OutSum,
    output FA1_OutC,
);

    assign FA1_OutSum = FA1_InA ^ FA1_InB ^
                        FA1_InC;
    assign FA1_OutC = (FA1_InA ^ FA1_InB) |
                      (FA1_InA & FA1_InB)

endmodule
```

Homework : currently the output = 5bits
now → make the output → 4bit with 1bit for overflow



* Lecture : 4

2020 8/24
3 → 4:30pm

* Verilog

- diff btw wire and register
- behavioural modelling
- blocking / non-blocking

eg: AND gate implimentation

```
                  interface
module  AND(in1, in2, out);
        input in1, in2;
        output out;
        assign out = in1 & in2;
endmodule
```

* Identifiers

name given to an object, such as register or a function or a module

must begin with alpha character or underscore
cannot start with number or "$"

can be upto 1024 characters

```
eg:  reg sum;   // this is 1 bit by
                //      default

// for defining 3 bits:
   reg foo [2:0];
```

* MODULE

- verilog describes a digital system or a set of modules.
- each module has an interface and contents desc.
- Modules communicate externally with
  - input
  - output
  - inout (bidirectional ports)
- modules consists of a list of statements declaring relationships between a module and its environment, and between signals within a module.

* Alternative to the AND example code

```
module AND(out, in1, in2);
       input in1, in2;
       output out;
          and (out, in1, in2);   // similar to
                                 // func. call
                                 // present in C
endmodule
```

This is called gate-level modelling.

Whereas, when we use the assign keyword, it is called Data flow modelling.

* Behavioural Modelling

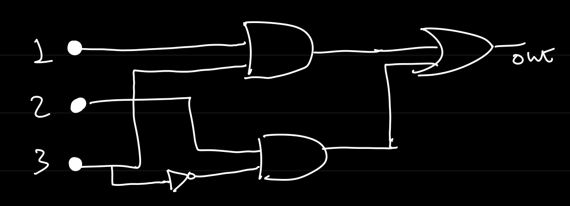
```
module AND (out, in1, in2);
       input in1, in2;
       output reg out;

       always @ (in1 or in2)
            out = in1 & in2;
endmodule
```

- must be reg type when used as LHS in an always block
- always @ means only when in1 or in2 changes, do whatever is inside the block. When they don't change, the variable with reg datatype retains its previous value.
- register is a data type in Verilog
- different than the register in the memory in the hardware
- why did we define the output as a reg datatype and not as wire datatype
- output is not updating everytime. Wherever in1 or in2 change, output changes here but when they don't, the output retains its last held value.
- if some variable is an output of the always block, it must be of the reg datatype.

eg: exam_1

Q = in1 & in3
P = $\overline{in3}$
R = in2 & $\overline{in3}$
O1 = $(1 \cdot 3) + (2 \cdot \overline{3})$



1	2	3	out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

2:1 multiplexer

in3 : select line
if in3 = 0 ⇒ out = in2
else ⇒ out = in1

* NOTE: We need to define wire datatype of size one bit

eg: 2 ⇒ 2 → 4 decoder

en	①	②	o0	o1	o2	o3
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

```
module decoder(
       input in_1,
       input in_2,
       input en,
       output out_0, out_1, out_2, out_3
       );
       assign out_0 = (en & ~in_1 & ~in_2);
       assign out_1 = (en &  in_1 & ~in_2);
       assign out_2 = (en & ~in_1 &  in_2);
       assign out_3 = (en &  in_1 &  in_2);
endmodule
```

eg3   eq = p0 + p1
      p0 = ~i0 x ~i1
      p1 = i0 x i1

since ⇒ parallel processing

⇒ XNOR $ab + \overline{ab}$
used to see if both inputs are same or not

eg3   Note: 1'b0 means 1 bit which is zero

2:1 multiplexer
using always @

```
module test_1 (
    input [2:0] in_1,
    input [2:0] in_2,
    input sel,
    output reg [2:0] out_1
    );

    always @(*) begin
        if (sel == 1'b0)
            out_1 = in_1;
        else
            out_1 = in_2;
    end
```

eg3   4:1 Mux

```
module mux(
     input [1:0] in_1, in_2, in_3, in_4,
     input [1:0] sel,
     output reg out_1
     );

     always @(*) begin

          if (sel == 2'b00)
               out_1 = in_1;
          else if (sel == 2'b01)
               out_1 = in_2;
          else if (sel == 2'b10)
               out_1 = in_3;
          else if (sel == 2'b11)
               out_1 = in_4;
     end

endmodule
```

HW: gate level representation of 4:1 mux

eg3   D latch [active high clear]
              positive edge triggered

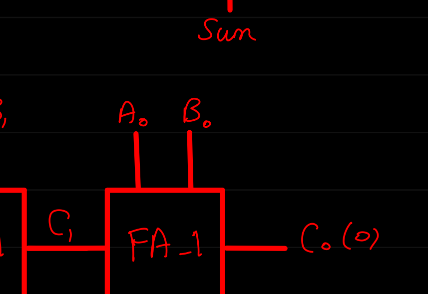
```
always @(*) begin
   if (clr)
       Q_out = 1b'0;
   else if (clk)
       Q_out = D_in;
end
```

- In D-latch, if clk = 1, output = data_in
  if clk = 0, latch the previous state

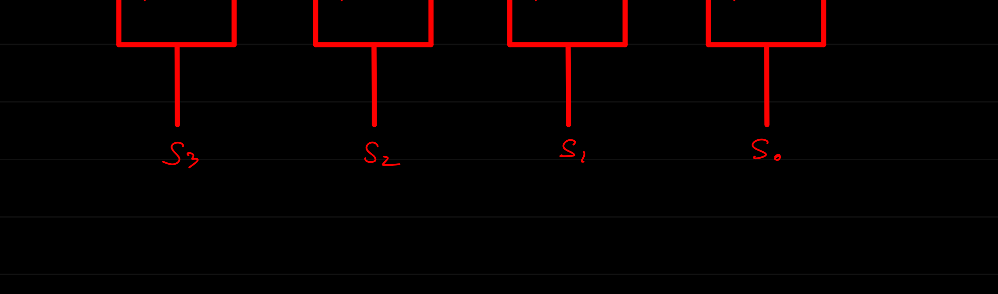
HW: HDL Bits
    Q1–12   Lab HW2
                 due: sept 2
    submit report (PDF)

Lab HW1: 4bit adder.

full_adder_1_bit (A, B, Cin, Sum, Cout)



full_adder : 4bit



* Lecture: 5

8:1 multiplexer

```
module mux(
    input a,b,c,d,e,f,g,h,
    input [2:0] sel,
    output reg out
    );
    always @(*) begin
        if (sel ==3'b000)
            out = a;
        else if ....
            ....
    ...
    end
endmodule
```

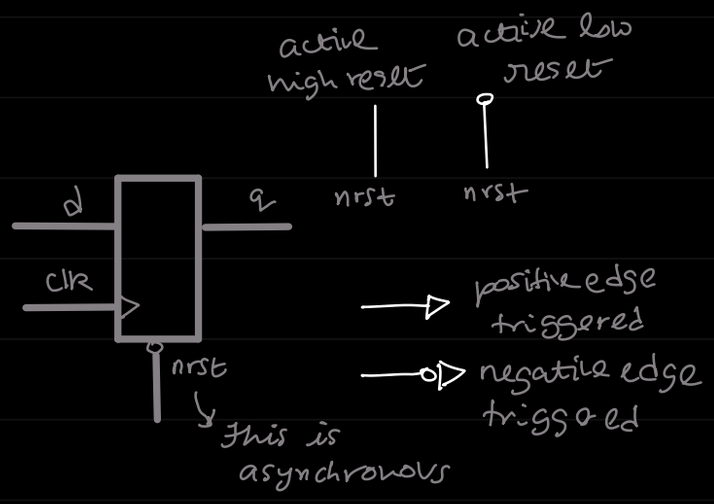
⇒ Note: These both are same

```
module foo(in1,in2,out);
    input in1 ,in2;
    output out;
endmodule

module bar(
    input in1, in2;
```

```
output out;
endmodule
```

* Flip flops:



- Reset : setting value to zero
- Preset : setting value to one

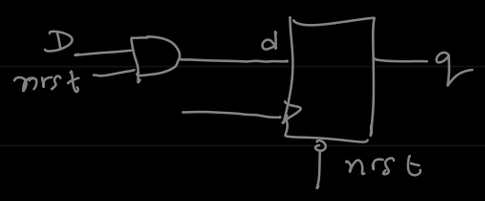
- Active high reset / clear / preset
  ↳ operation happens when input signal is active high

- Active low reset / clear / preset
  ↳ operation happens when input signal is active low

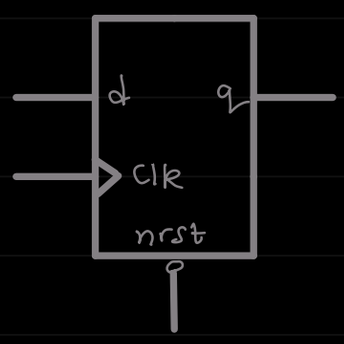
- Synchronous : operation will happen at the edge of the clock

- Asynchronous : operation can happen at any time (irrespective of the clock)

⇒ Flip flops are used to
- store something in memory
- pass a digital signal synchronously to make reset synchronous



* Verilog code for D flip flop




```
module D_FF(clk, nrst, d, q);
    input clk, nrst, d ;
    output reg q;

    always @ (posedge clk or negedge nrst)
        begin
            if (!nrst)
                q <= 0;
            else
                q <= d;

        end
endmodule
```

* NOTE: <= means blocking or non-blocking
we cannot assign inside always block
because assign uses data flow level approach

- Designing combinational ckt - use always@(*)

- Designing Flip Flops – Use always @(posedge clk)

* Parallel-In, Parallel-Out REGISTER

memory in the hardware, not the datatype here

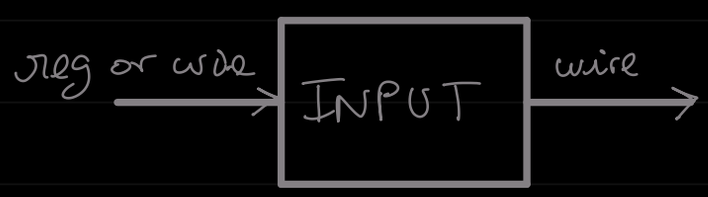
- Note: is reg datatype linked to a register in memory in the hardware?
  Not necessarily
  eg: not in mux code
  but yes in register code or d flip flop code

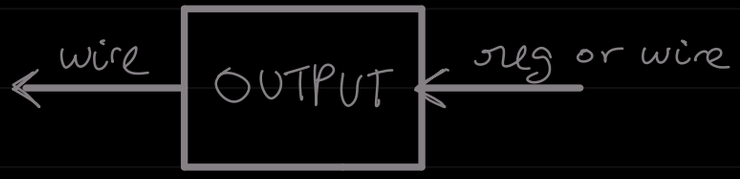
* MODULE PORTS : provide interface for the module to communicate with its environment

≡ interface
input, output, inout

- Declaration: <Port direction><width><port_name>;

- Port direction can be input/output/inout





- An Input Port
  driven by external entity
- An output Port
  driven by internal entity
- An Inout Port
  driven by both internal and external entities.

* Module Interconnections

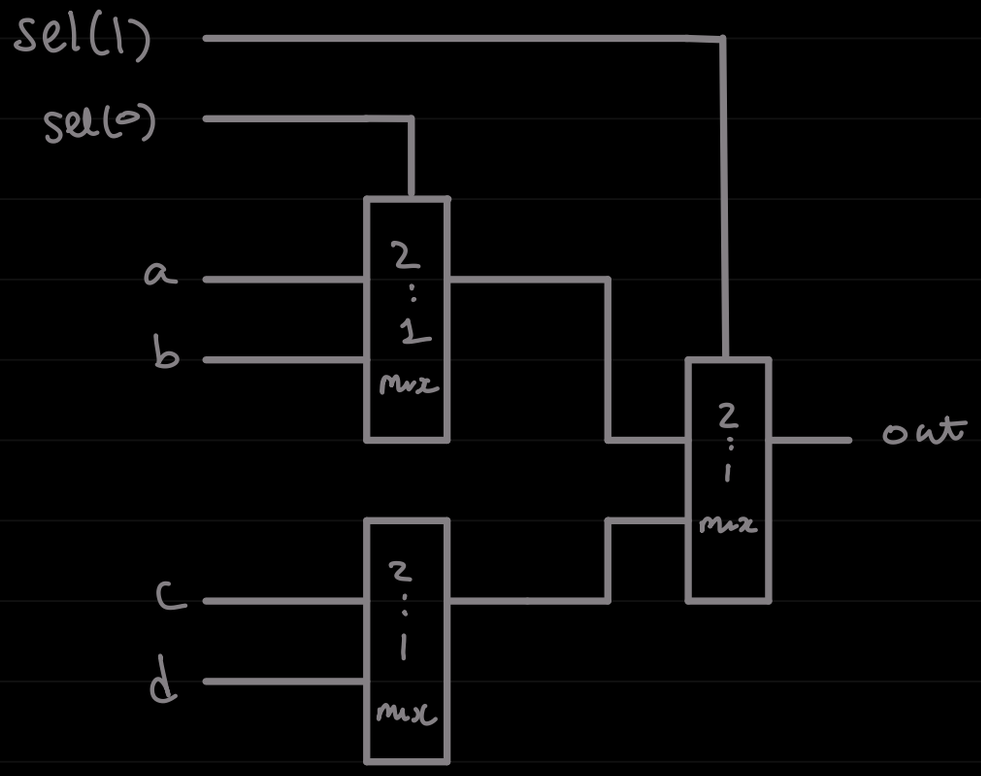
⇒ Named Association

```
FA9  fa_byname (.cout(COUT), .sum(SUM),
                .b(B), .a(A), .cin(Cin));
```

⇒ Order Association

```
FA4  fa_byorder (SUM, COUT, A, B, Cin) ;
```

* Coding a 4:1 Mux using 3x 2:1 Mux



* Design a 2:1 mux using data flow approach

```
module mux(a,b, sel,out);
    input a, b, sel;
    output out;

        assign out = a & sel | b & ~sel;
endmodule.
```

* VERILOG : Number Representation

- Verilog allows integer numbers to be specified as :

Sized	(dynamic size)
Unsized	(always 32 bits)

- In a radix of binary, hexa, octa, decimal (default)

Syntax:

549 ≡ 32'd549
'h8FF ≡ 32'h8FF
'o765 ≡ 32'o765
4'b11 ≡ 4'b11
8d9 × ⇒ 8'h9 ✓
1 ≡ 000.....00001
12'hx ≡ 12 bit unkown number

* Negative Number

– [number] Signed

default : positive number unsigned

in hardware : 2' compliment

eg: –4'b11

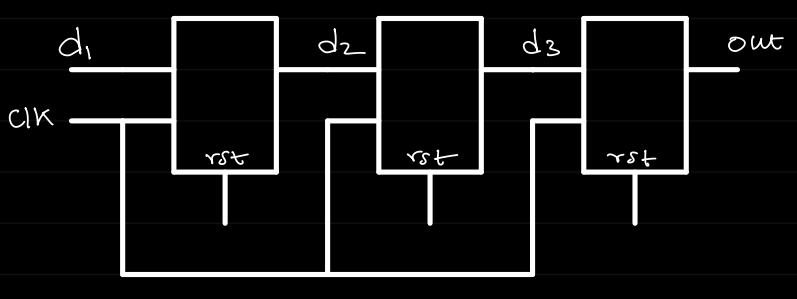
* LAB: ELD

27/08/24 8:30AM

- For counter to increment every second, Design 8 bit up counter (0→255) using behavioral modelling
- Design 1Hz clock from input 100MHz clock using clock divider
- Lab HW: Design up/down counter with maximum count of 85
- write a verilog code where output is delayed version of input by 1 clk cycle

Ans) just make a d flip flop

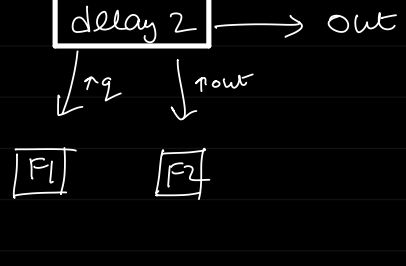
note: if we want 3 cycle delay, we pass the input through 3 d flip flops



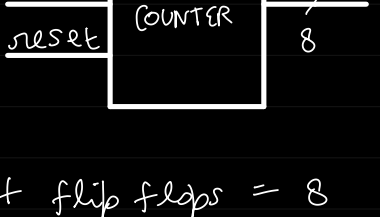

```
module delay_2(Din, clk, out);
    input Din, clk;
    output reg* out;

    D_FF F1(Din, clk, Q);
    D_FF F2(Q, clk, out);
endmodule
```

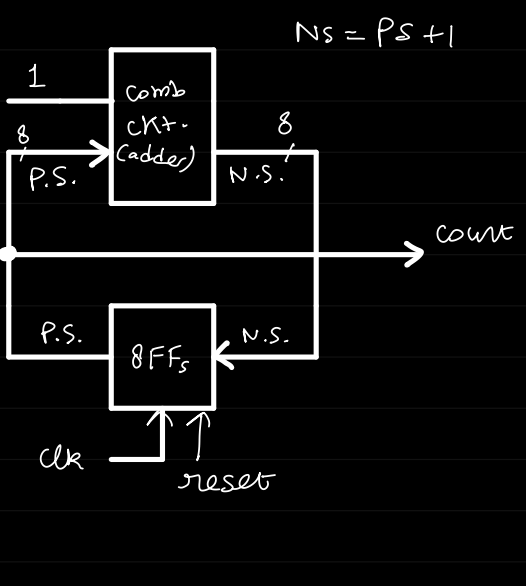
*Note: Q and out are reg inside each D_FF block but will be wire in this module because both are inputs from perspective of our module. Look up previous lecture for the same.



- 8 bit up counter
  (1) block diagram
  (2) define all signals
  (3) write code



number of flip flops = 8
because number of states = 256
note: in the flip flop, we just store the state of the circuit




```
module counter(
    input clk, reset,
    output [7:0] count  reg or wire ?
);

reg [7:0] NS, PS;

// flipflop
always @ (posedge clk)
begin
    if (reset)
        PS <= 8'b00000000
        // same as PS <= 8'd0
    else
        PS <= NS
end

// combinational ckt
always @(*)
begin
    NS = PS + 1;
end

// finally, assigning the output

assign count = PS; // if we take count as wire
                   // or * : same functionality
always @(PS)       // if we take count as reg
begin
    count = PS;
end

endmodule
```

- Note: we need to define NS and PS here because they should be 8bit each but by default size = 1bit

synchronous, active high reset } ⇒ D flip flop