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)
↖ AUR ✓ ↙ arch user repository

* GRADES:

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

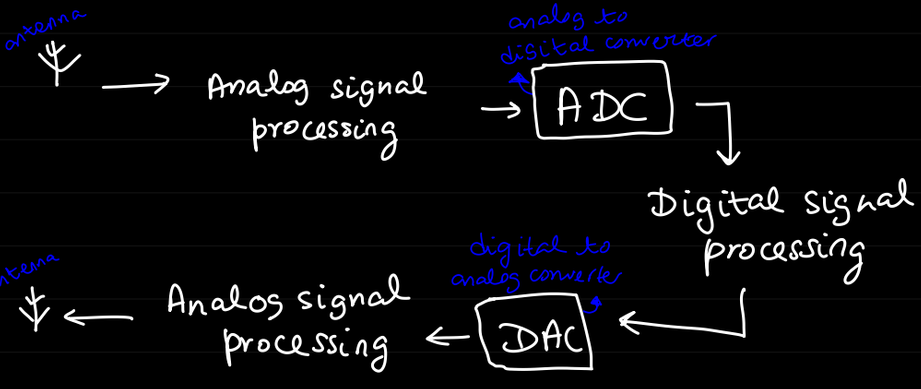
LECTURE : 1

* **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)

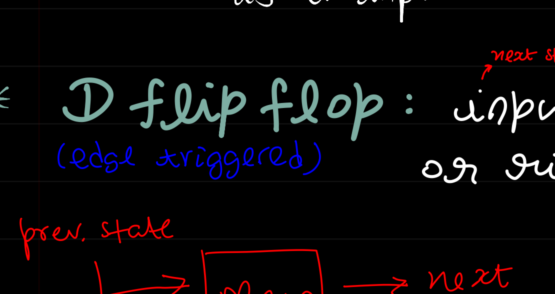
* **sequential circuit** : output depends upon the current input and the current state of the circuit

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

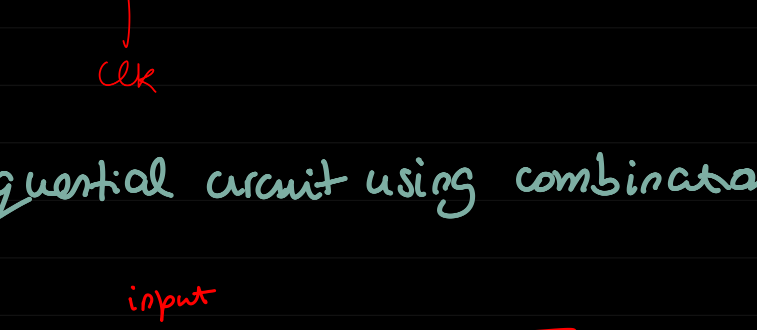
what we get → output + next state
(because we go from one state to another)

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

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

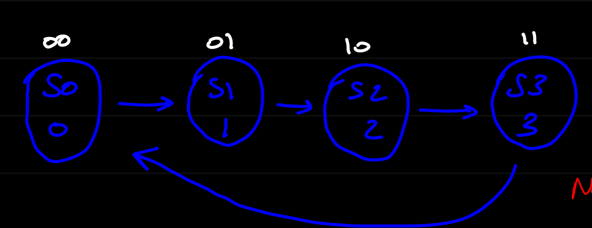


* **Sequential circuit using combinational ckt**



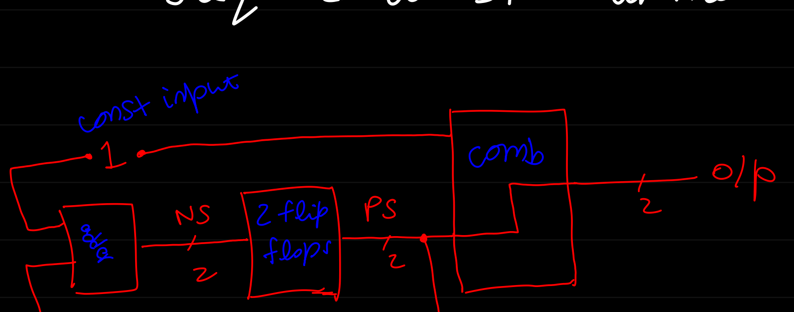
* **FSM (finite state machine)**

⇒ up counter

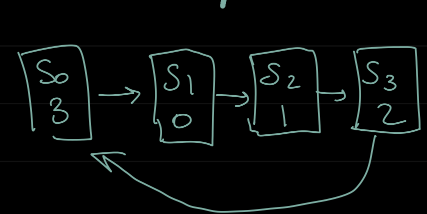


Note : if curr state = S_n, the output is n

2 bits required to store in mem

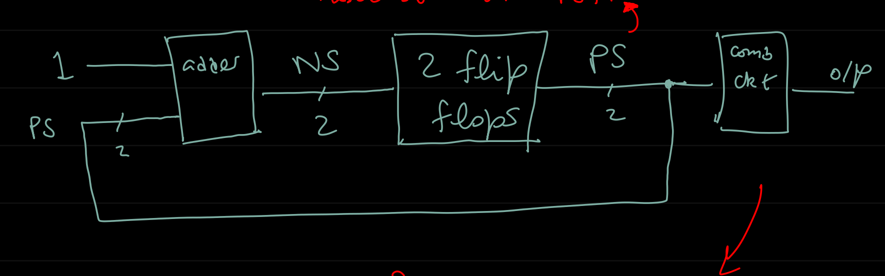


* **Another example**



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

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



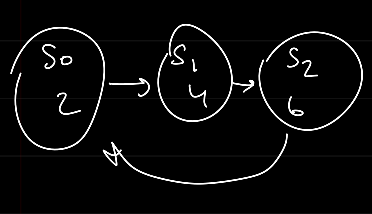
{ Here we should use 2 not gates for the 2 bits

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)

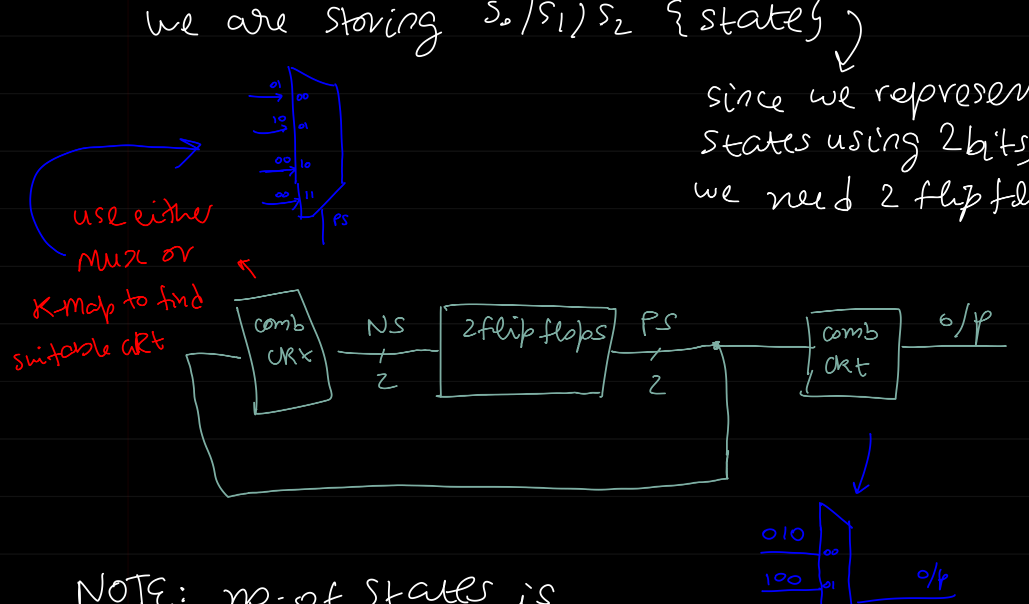
* **Example : 3**

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



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

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

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 (or self checking test bench) — ensures that the design is functionally correct } doesn't guarantee the code will work on the hardware
↓
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
(maps the logic defined by an NGD file into FPGA elements; output is NCD file (native circuit desc))
↓
Implementation ⇐ mapping, placement, routing
- mapping: compares gate level circuit & FPGA circuit
- placement: location of diff components in the circuit in consideration with its placement requirements (eg. component temp) — one of all the blocks in the FPGA, which one am I gonna use?
- routing: 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

→ 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

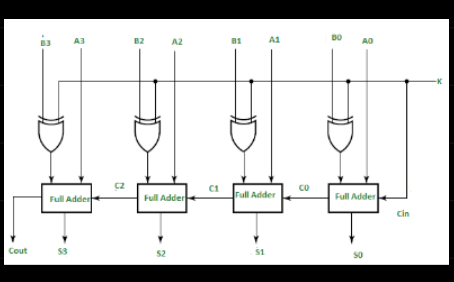
flexibility ⟷ efficiency

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.

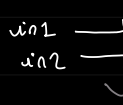
Verilog looks like C but describes hardware

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



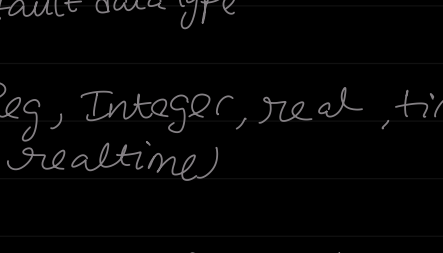
in 1	in 2	out
0	0	0
0	1	0
1	0	0
1	1	1

```
// <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

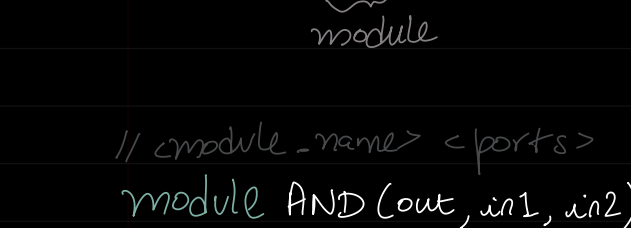
⇒ Half Adder

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

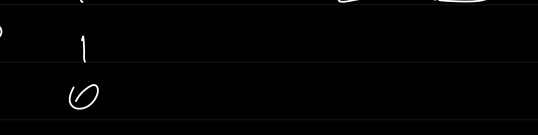



```
module full_adder_1bit (
    input FA1_InA,
    input FA1_InB,
    input FA1_InC,
    output FA1_OutSum,
    output FA1_OutC,
);

    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

2016/8/24
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 as 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

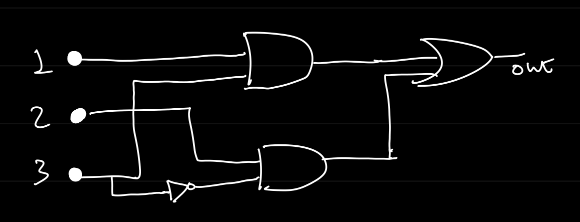
```
module AND (out, in1, in2);
       input in1, in2;
       output reg out;

       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	6
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

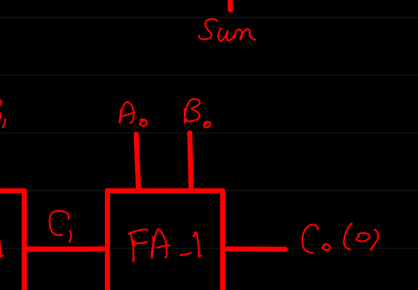
```
always @ (*) begin
    if (clr)
        Q_out = 1'b0;
    else if (clk)
        Q_out = D_in;
end
```

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

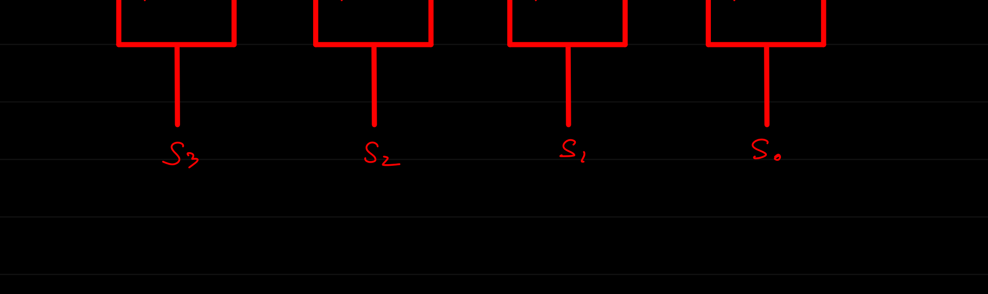
HW: HDL Bits
Q1-12   Lab HW2
due: Sept 2
submit report (PDF)

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
```

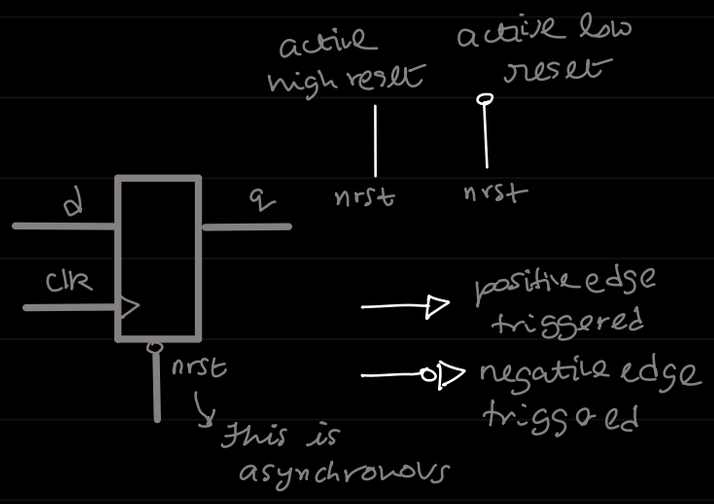
⇒ Note: These both are same

```
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

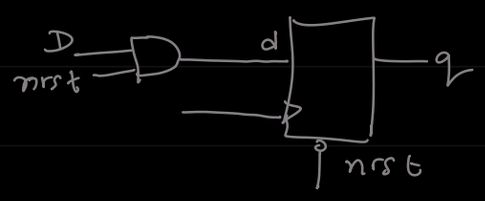
- Active high reset / clear / preset
  ↳ operation happens when input signal is active high

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

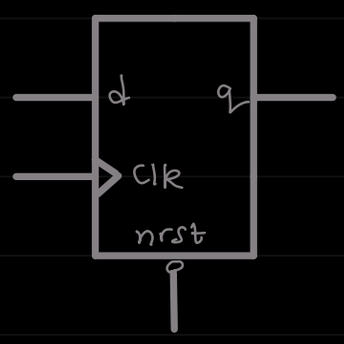
- Synchronous : operation will happen at the edge of the clock

- 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

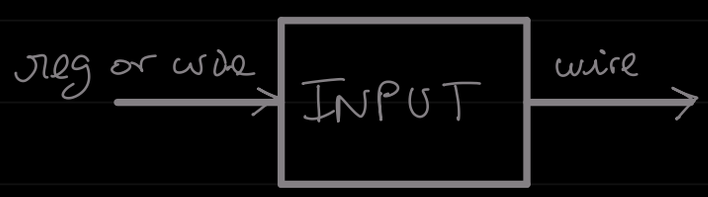
- Note: is reg datatype linked to a register in memory in the hardware?
  Not necessarily
  eg: not in mux code
  but yes in register code or d flip flop code

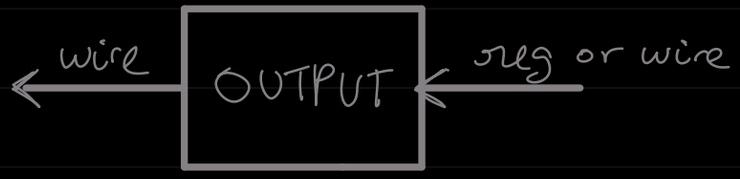
* MODULE PORTS : provide interface for the module to communicate with its environment

≡ 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

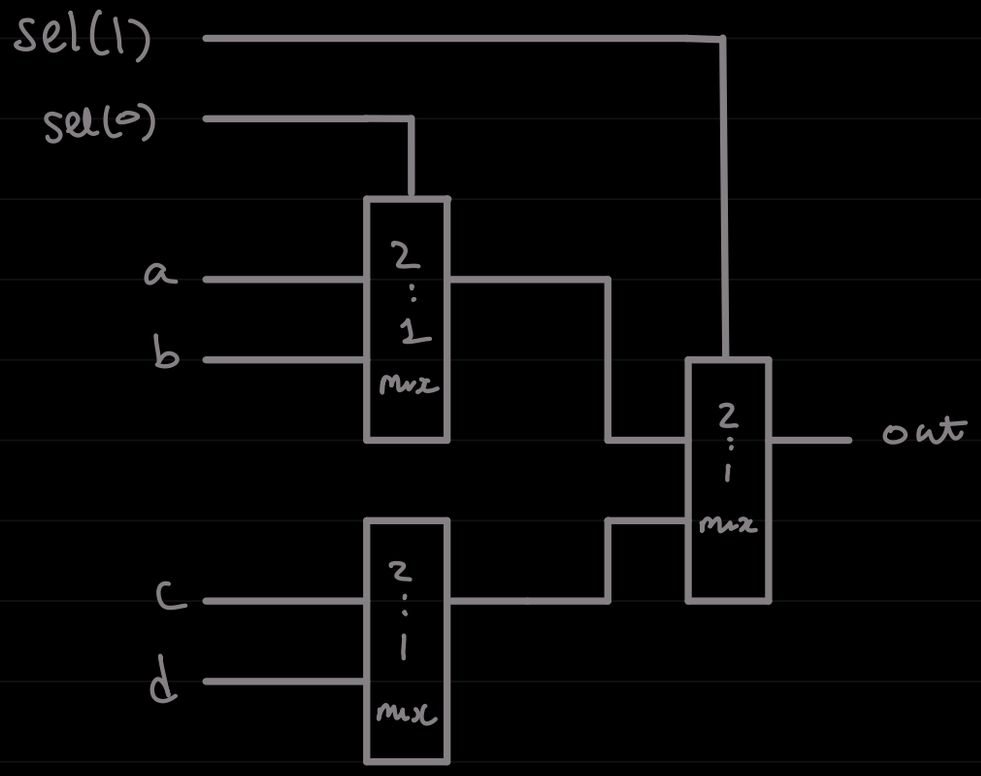
⇒ Named Association

```
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

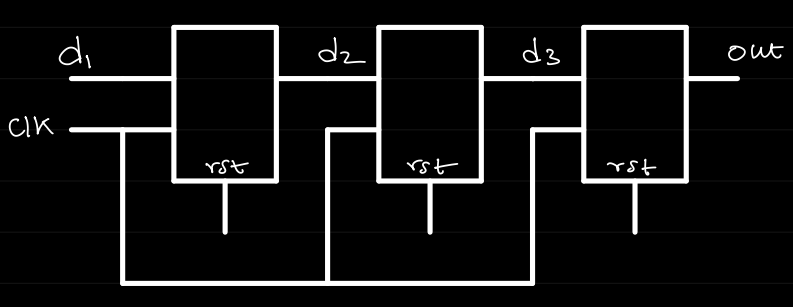
* LAB: ELD

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

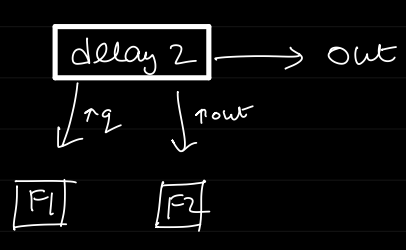
note: if we want 3 cycle delay, we pass the input through 3 d flip flops



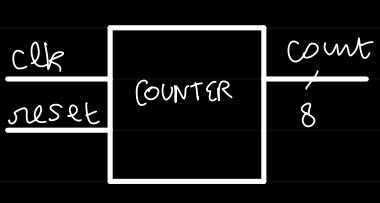

```
module delay_2(Din, clk, out);
    input Din, clk;
    output reg* out;

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

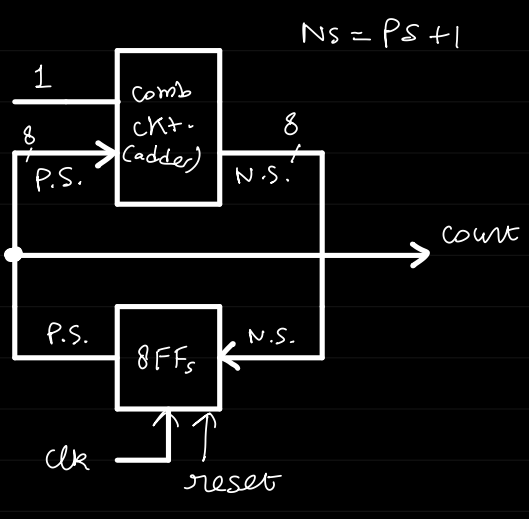
Note: Q and out are reg inside each D_FF block but will be wire in this module because both are inputs from perspective of our module. Look up previous lecture for the same.



- 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

- Testbench :

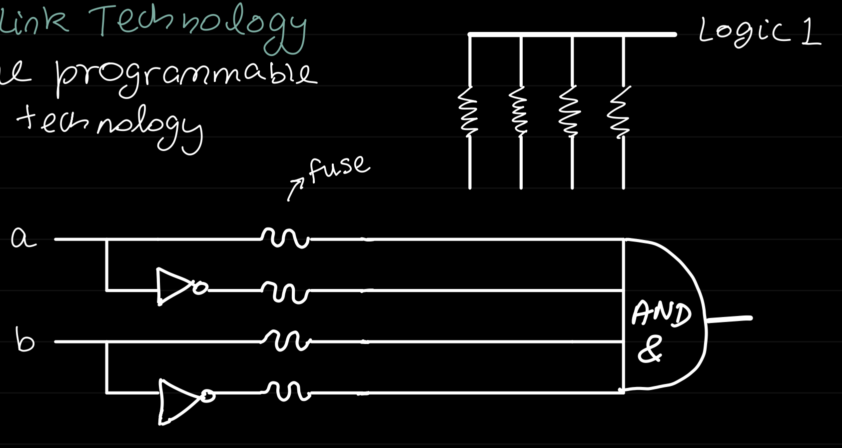
The testbench verilog file will be higher as compared to src file in context of hierarchy.

* LECTURE : 6 (Architecture) 27/08/24
3 – 4:30PM

- Programmable Logic Device (PLD)
  Devices whose...
  Internal Architecture is predefined by manufacturer but are created in a way so that they can be configured in the field to perform variety of functions.

- Programmability at the Software level
  eg: Arduino / RPI Pico
  But you cannot change the instruction set architecture of the CPU

* Fusible Link Technology
one time programmable
antifuse technology
fuses cannot be changed once blown



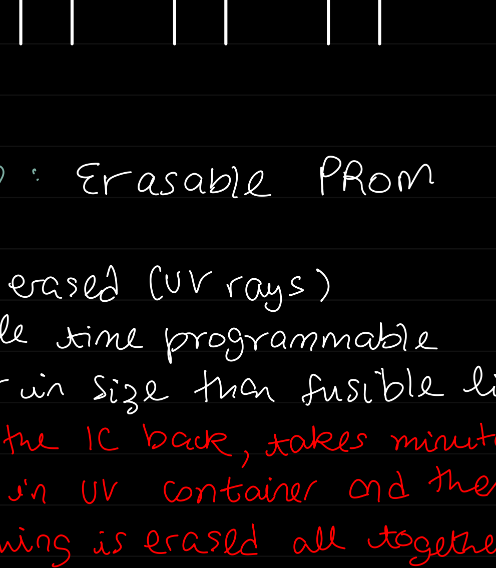
* PROM : progammable read-only memory (1970)

predefined link | programmable link (fuse)

- blow the fuses as per your logic
- one time programmable
- Single PROM instead of multiple chips
  smaller
  lighter
  cheaper
  less prone to errors (fewer solder joints)
  easy to identify errors / correct errors
- Designed for use as memories to store computer programs and constant data values
- Also useful for implementation for simple logic function such as LUT & State machines

  very high amount of fuses required for complex ckts
  programmable once only
  need to switch fuse once blown

(SOP ✓)



* EPROM : Erasable PROM (1971: by intel)

- can be erased (UV rays)
- multiple time programmable
- Smaller in size than fusible linked devices
- bring the IC back, takes minutes to erase by putting in UV container and then put it back
- whole thing is erased all together
  Cannot erase sections of the device.
- Expensive
- Erasing process becomes complex as density of transistors increases

* EEPROM : Electrically EPROM

* PLA : Programmable Logic Arrays (1975)

- High delay
- makes the left side of PROM programmable as well
- Did not get adopted because people were more comfortable with SoP form

* Programmable Logic Device

→ SPLD : Simple
→ CPLD : Complex

- Complex PLDs (CPLD) 1984

→ Need for bigger (functionally), smaller (size), faster and cheaper technology

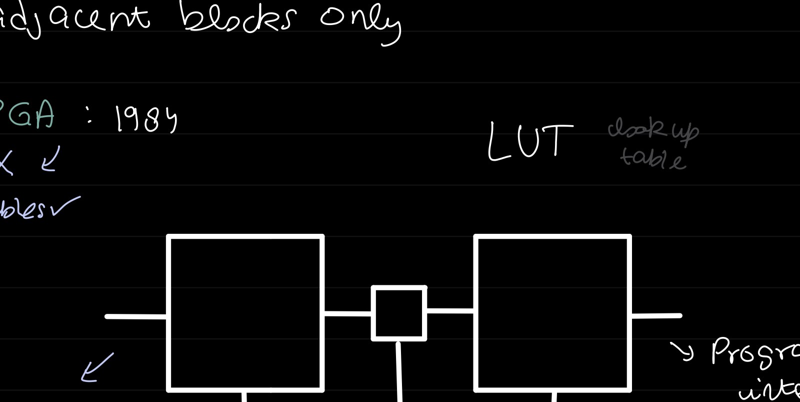
→ MegaPAL : interconnection of 4 PAL ↳ Programmable Array Logic
high power consumption

→ 1984: Altera introduced CPLD using CMOS (high density, low power) and EPROM / EEPROM (programmability)
$\equiv E^2PROM$

≡ multiple SPLDs

→ Added multiplexers to each SPLD so that only the necessary stuff is processed
≡ to combat higher power consumption

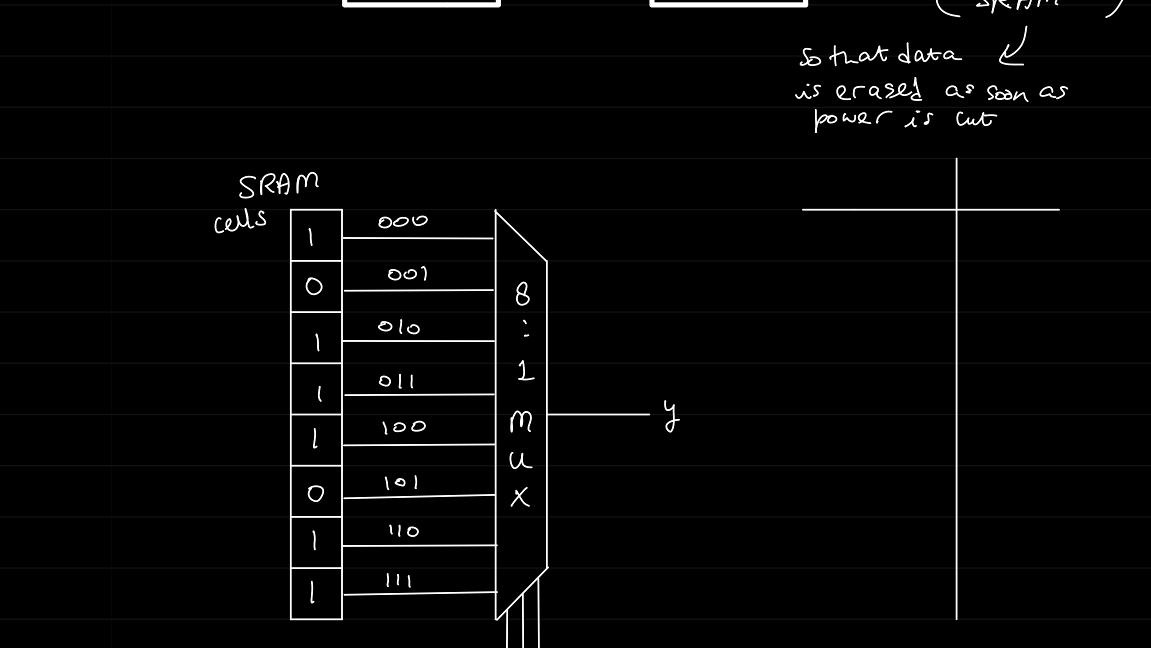
by ALTERA

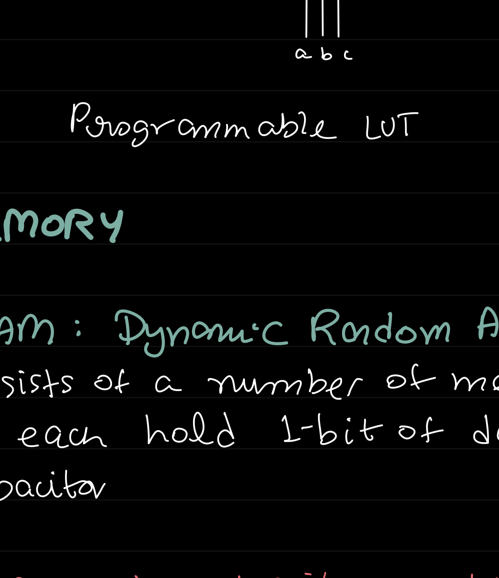


Signals from a logic block can travel (/communicate) through adjacent blocks only

* FPGA : 1984
gates ✗
Lookup tables ✓

LUT lookup table





Programmable LUT

* MEMORY

- DRAM : Dynamic Random Access Memory
  Consists of a number of memory cells which can each hold 1-bit of data, stored in a capacitor

  charge in capacitor eventually fades unless the data in each memory cell is refreshed periodically and we cannot read/write while it is being refreshed, hence, lower throughput

  The memory controller refreshes the memory cell by checking each one's state
  ≡ hence named dynamic

  used in all external memory since each bit uses only a single transistor to be stored.

* SRAM: Static RAM

- Uses latches to store the data

  very good for prototyping
  more space
  reprogrammable
  no refreshing needed
  overwritable unlike E^2PROM in CPLD
  high speed, low capacity memory chips (L1 cache) ← can be erased instantly on power down
  difficult to reverse engineer

  uses 6 transistors
  large size of configuration cell
  expensive
  data can be changed due to radiations
  volatile — will loose its data on powerdown unlike E^2PROM / Antifuse; must be programmed on power up

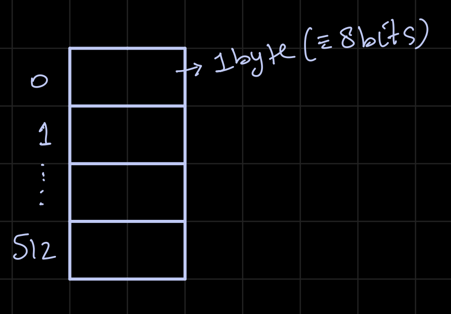
Note: Antifuse technology is used by military because it cannot be (or very difficult) reverse engineered and is safe from radiations

* Lecture : 7 28/08/24

• LUT as memory & LUT as ALU
↳ lookup table

⇒ Operations on memory : read or write

- read: we provide the address of the memory → 9 bytes here
- write: you provide the data and the address (1 byte, 9 bytes)



9 bits used to represent memory

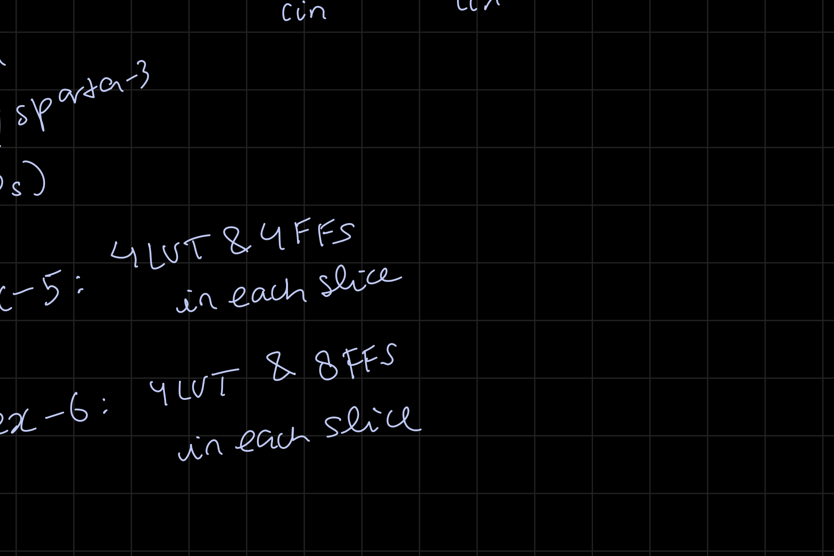
* FPGA

CLB : configuration logic block
BRAM : block RAM (used to store large amt of data)
input/output block
CMT : Clock management tile
FIFO Logic
BUFG : Global buffer
DSP : digital signal processing
BUF IO and BUFR : Input/output & Regional buffer
MGT : multi-gigabit transciever

→ we use one oscillator for generating one clock
→ we use CMT block for clocks other than the one generated by oscillator

* Configurable Logic Block [CLB]

slices can be 2/4/... in number inside CLB



2 LUT and 2 flip flops inside each slice in Virtex-2/Spartan-3 (1990s)

In Vertex-5 : 4 LUT & 4 FFs in each slice

In Vertex-6 : 4 LUT & 8 FFs in each slice

our flip flop now ?

Slices	LUT	Flip Flops	Arithmetic & carry chains
2	8	16	2
	(4 LUT/slice) ≡ 6 input LUT	(8 FF/slice)	

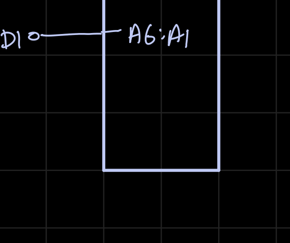
total : 64 bit of data stored in LUTs
8 bits stored in each LUT (6 bits input to LUT)

≡ 512 bit of data in one CLB ✗ see below

• CLB : SLICES

① SLICEM : Full slice } read and write only

- LUT can be used for logic (combinational circuit) and memory / SRL (shift register)



read address + write address

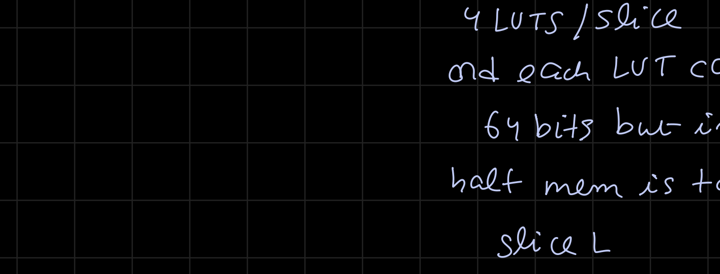
② SLICEL : logic and arithmetic only } read only

- LUT can only be used for logic (not memory / SRL)

read address only

• We store large amount of data in BRAM

→ in our FPGA (7-series), there are only 25% SLICEM and 75% SLICEL



So, we can store either 0 or 128 bits in one CLB depending on its type ↙ 4 LUTs / slice and each LUT can store 64 bits but in CLB_LM half mem is taken by slice L

A/B/C/D — is asynchronous
_Q is synchronous as it passes through FF

6 input LUT + 6 input LUT ⟶ 7 input LUT
(64+64 = 128 bit input)

FPGA → CLB → 2x slices → 4x LUT + MUX + carry chains + 4 Flip flops or latches + 4 additional flip flops

- slice L → LUT used logical / arithmetic
- slice M → LUT used for logic + memory

* Verilog : Vector and memory

- only **net** or **reg** datatypes can be declared as vectors (multiple bit width)
- specifying vectors for **integer**, **real**, **realtime** and **time** datatype is not allowed
- default : 1 bit (scalar)

eg: `wire [7:0] a_byte ; reg [31:0] a_word;`

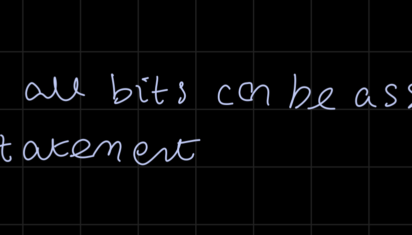
```
reg [11:0] counter;    12 bit
reg a;                 1 bit
reg [2:0] b;           3 bit

a = counter[7];        index 7
b = counter[4:2];      4,3,2
```

eg: `wire [-3:0];` 4 bits size

• memory

`reg [3:0] mem [255:0] , red ;` (red ↳ 4 bit vector)



256 locations with each = 4 bits

• memory vs vector

- vector and mem declarations are not same
- in a vector, all bits can be assigned a value in one statement
- in memory , assigned separately.

```
reg [7:0] vect = 8'b 10100011
reg array [7:0];   // 8 locations of 1 bit
array [7] = .... ;
array [6] = ...... ;
   ⋮
array [0] = ... ;
```

* LAB: 3 (Running on hardware) 03/09/24

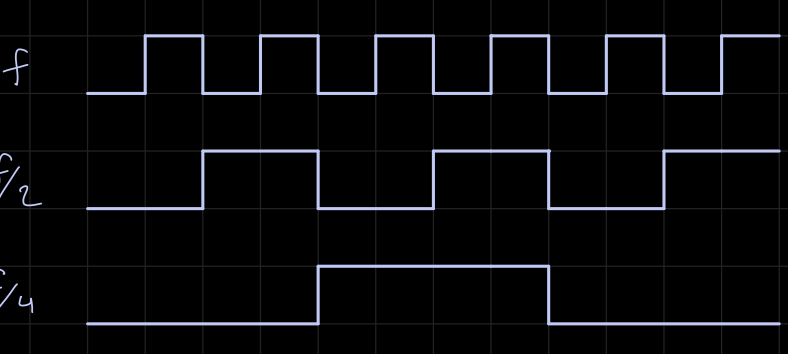
⇒ Lets say our program is an 8 bit upcounter the program will run on the hardware but how will you observe the output?

① one way could be using LEDs (8 of them for 8 bits) to represent the output physically on the board.

② Also, we need an additional circuit to convert 100MHz → 1Hz from oscillator ↙

③ Also, how will you implement reset signal? How about a physical button connected to the board.

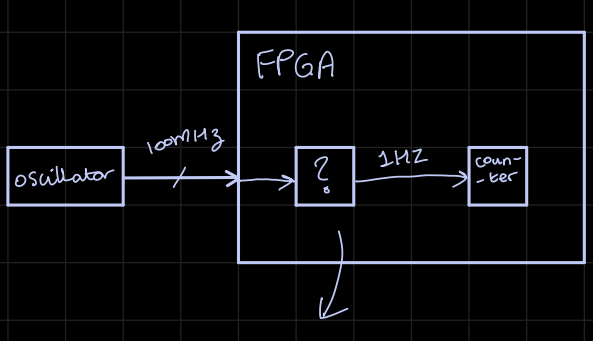
lets take example of 3 bit up counter

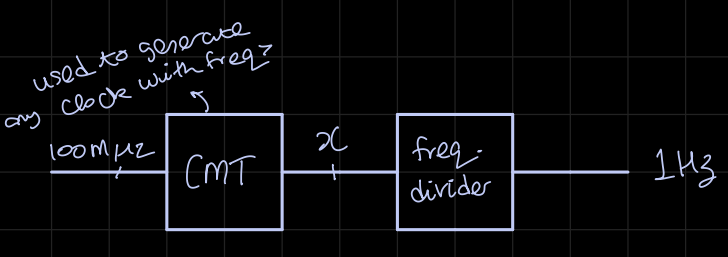


frequency dividing with order of 2

time period for which clock is high ≡ duty cycle

here, duty cycle is 50%





We need to find x such that $x > 4MHz$ and number of bits of freq. divider

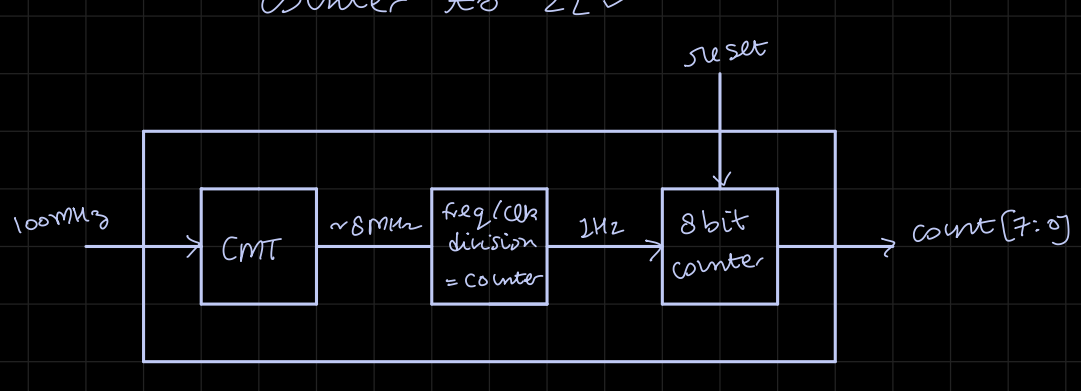
here $\frac{x}{2^{23}} = 1 \Rightarrow x = 8388608\ Hz = 8.388\ MHz$

(≡ clock division)

and hence the frequency divider / counter is 23 bits

to get 2Hz as output from above,

we can change no. of bit of counter to 22 ✓



we need 4 codes

→ CMT
→ freq_division
→ Counter
→ Top code to converge them all

Notes:

① How to get CMT on Vivado?

IP Catalog → Clocking wizard → Clocking options (clk_100m) ≡ 100MHz → output Clocks (clk_8m) ≡ 8.388MHz

note: the "locked" signal is high when the output signal reaches the intended frequency

after generating clock_wizard, we need to instantiate it

to instantiate Clocking_wizard:

SOURCES
↓
IP SOURCES
↓
clk_wiz_0
- ...
- ...
- Instantiation Template
  - clk_wiz_0.veo

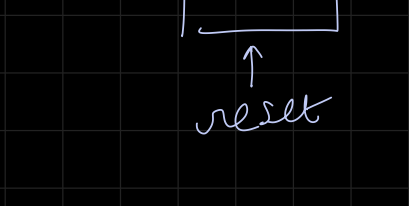
↙

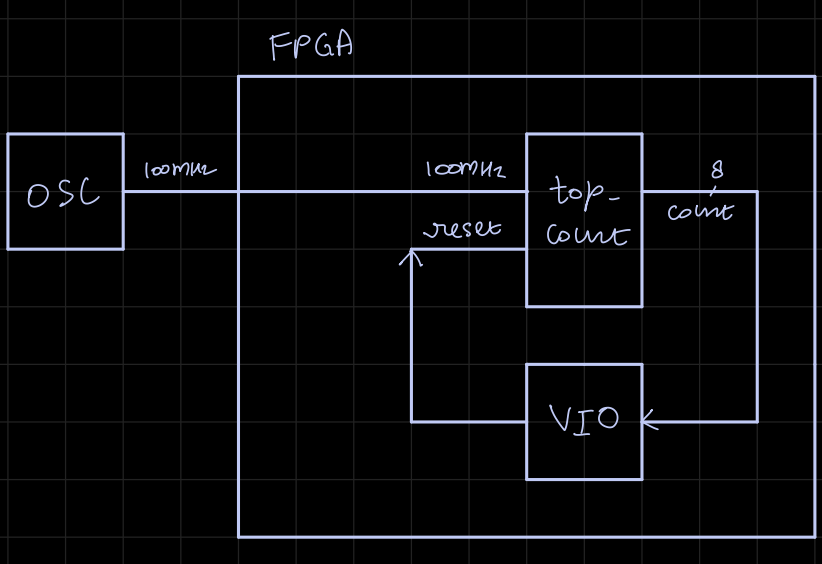
copy verilog code from here to top_count.v

⇒ How to run code on hardware now?

after instantiating all 3 modules in top_count ⇒

focus only on the i/p & o/p of whole block

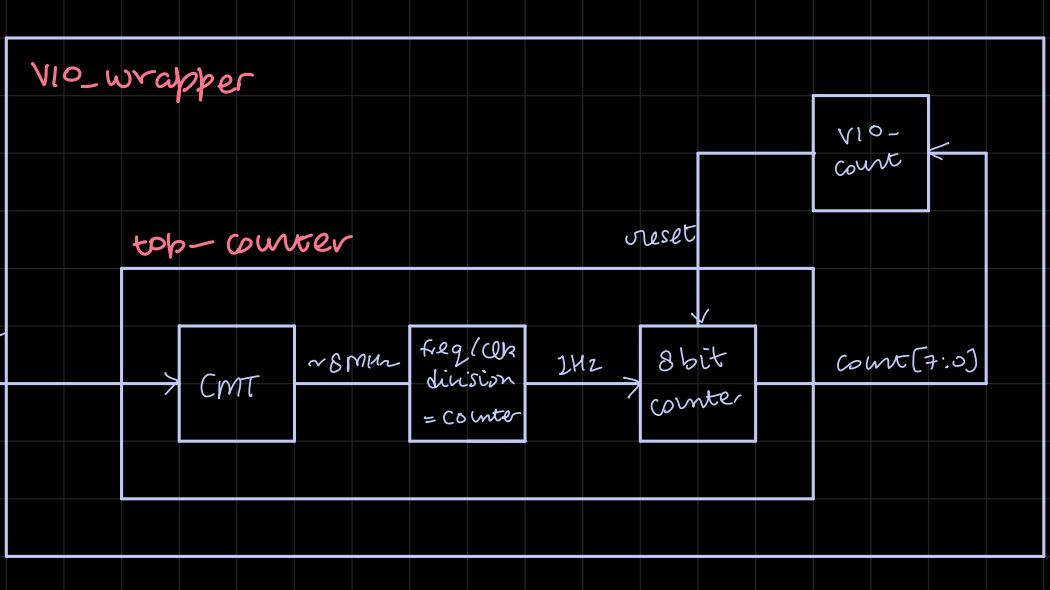




Virtual Input Output ≡ VIO

≡ debugger

≡ exact replica of testbencher but now it is running on hardware



* lecture : 8 03/09/24

To get stats about the project go to project summary tab on Vivado

→ vector & memory

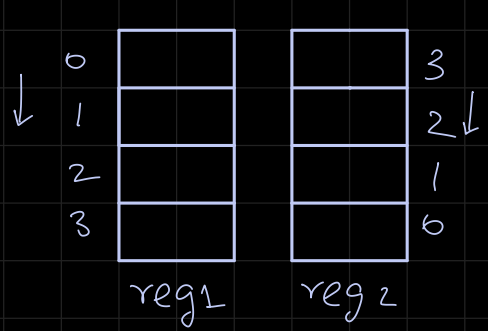
reg [7:0] my-reg [0:31];

↳ memory with 32 positions of 8bit size each

integer matrix [4:0] [0:31];

↳ 2dimensional memory

MSB LSB
wire [1:0] reg1 [0:3];
wire [1:0] reg2 [3:0];



array2 [100] [7] [31:24];

↳ 4th byte from 101th column and 8th row

reg [31:0] Data_RAM [0:255]

read 2nd byte from Address 11 (index)

Data_RAM [11] [15:8]

2nd and 3rd byte from address 77

Data_RAM [77] [23:8]

printf : C :: $display : verilog

* Vector indexing

reg [63:0] word;
reg [3:0] byte_num;
reg [7:0] byteN;

byteN = word [byte_num * 8 + : 8]

≡ 4x8 + : 8
≡ 32 + : 8 (forward direction)
≡ [39:32]

word [7+:16] ≡ [22:7]

a [31-:8] ≡ a [31:24]

8bits of data from index 31 in backwards direction

for (i=0; i<5; i=i+1)

$display ("%s", str [i*8:8]);

⇒ edcba

* Verilog : Register vs Integer

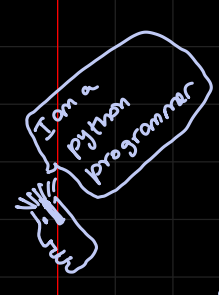
- Reg is by default 1bit wide data type. If more bits are required, we use range declaration.
- Integer is a 32 bit wide datatype.
- Integer cannot change its width. It is fixed.
- Not much utility as compared to Reg / Net
- Typically used for constants or loop variables
- Vivado automatically trims unused bits of Integers.
  eg: Integer i = 255;
  → then i ≡ 8bits

* OPERATORS

↳ unary
↳ Binary
↳ Ternary
} based on the number of operands

a = ~b;
a = a && b;
a = b ? c : d;

* BUS OPERATORS



[] Bit / Part select A[0] = 1'b1

{} Concatenation {A[5:2], A[7:6], 2'b01}

{x{}} Replication {3{A[7:6]}}
= 6'b101010

<< Shift left logical x 2^x

>> Shift right logical ÷ 2^x

shifting bits is very cheap (≡ signal renaming) used to perform multiplication and division powers of 2.

eg: 6 (≡ 4'b0110) <<→ 4'b1100 (≡ 8+4 = 12)
>>→ 4'b0011 (≡ 2+1 = 3)

works perfectly only for unsigned numbers

when working with signed numbers, towards LSB during right shift, you need to retain the signed bit

retain MSB when shifting towards LSB
10001011 >>>3
= 11110001

OR just use Shift right arithmetic works for signed 2's compliment

no such problems when shifting towards MSB (<<<)

assign {b[7:0], b[15:8]} = {a[15:8], a[7:0]}

↳ byte swap

eg: a = 8'h AB = 16'h 00AB

b = 16'h AB00

* LECTURE: 9 05/09/24

- Note: shift left logical (<<) = shift left arithmetic (<<<)

⇒ **Arithmetic operators:** +, −, *, /, %, ** (addition, subtract, multiply, division, modulus, power)

- For multiplication, FPGAs have DSP48 dedicated to fast math.
- If these are available, they will be used. However, if there are no free DSP48, CLB is used which is large and slow.
- Use shift operation for multiplying or dividing by power of 2.
- Subtraction/Addition of N bits numbers result in a max of (N+1) bits number
- For multiplication of two N bit numbers, result is max 2N bits number
- make sure to define your variables and their size explicitly.

⇒ **Bitwise operators:** operates on each bit individually

~	inverse	output can be multi-bit
&	And	
|	Or	
^	not	
~^	xNOR	

- if the two operands are of different lengths, the shorter one is padded with its MSB with signed bit
  - ↳ so number of gates required: max{len(A), len(B)}
- by default everything is unsigned
- this is how we can tell the tool that we want signed operation:

```
assign out = ($signed(a)) < ($signed(b))
```

or we can store the value in signed way using $signed and do operation normally.

⇒ **Logical operators:** output is one bit only (0, 1 or TRUE/FALSE)

!	NOT
&&	AND
||	OR
==	EQUAL
!=	NOT EQUAL
<, >, ≤, ≥	COMPARISON

a	b	a&b	a|b	a&&b	a||b
0	1	0	1	0/F	1/T
000	000	000	000	0/F	0/F
000	001	000	001	0/F	1/T
011	001	001	011	1/T	1/T

if any of the bits of the output of bitwise operator is 1 ⇒ then logical operator output = 1 (TRUE)
else: 0 (FALSE)

⇒ **Reduction operators:** output is also one bit

&	AND
~&	NAND
|	OR
~|	NOR
^	XOR
~^	XNOR

notation: <operator><operand>

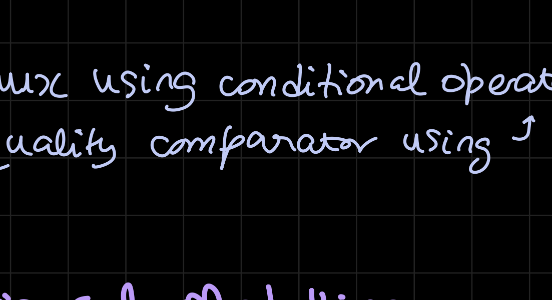
eg: ~&A $= \sum_{i=0}^{n} \sim\& A[i]$

⇒ **Conditional operators:** condition ? true_val : false_val

2:1 mux → sel ? a : b

* PRACTICE:

```
module max (
  input a,b,c,
  output out
)
  assign out = (a>b) ?
               ((a>c) ? a : c)
               : ((b>c) ? b : c)
```



HW: design 4:1 mux using conditional operator
design 1 bit equality comparator using ↑

2) Efficient Behavioural Modelling

- 2 basic blocks in behavioural modelling: always (used most in: synthesizable code) and initial (test benches)
  - both run in parallel (will not block the execution of other blocks)
- All of them start at simulation time 0 (#0)

- **INITIAL BLOCK**
  - starts at #0 and executes only once.
  - we cannot use "always" inside "initial" & vice versa
  - used for initializing / setting global constants

```
initial
  begin
    #5 a = 1'b1        after 5 units
    #25 b = 1'b0       after 30 units
    #70 $finish        after 100 units
  end
```

(wait 70 more units)

b = #50 c & d — calculated at t=0 but assigned at t=50

#50 b = c&d } calculated & assigned at t = 50

In the hardware, delay is created in context of clock cycles instead of seconds.

Hence, we use flip flops to create delay on hardware

note: "=" makes code run sequentially

- **ALWAYS BLOCK**
  - starts at t=0
  - executes statements continuously in a loop
  - statements inside always block are executed either sequentially (=) (blocking assignment) or parallely (<=) (non-blocking assignment)
  - describes the functionality of the circuit
  - used for clock declaration

always	always @(*) ...	always @(posedge....) ....
not synthesizable on hardware ✗	synthesizable on hardware ✓	synthesizable on hardware ✓

* Combinational circuits using always

- Common ERRORS
  1. variables updated in multiple always block → bad because of parallel execution, one var cannot be o/p of 2 blocks
  2. Incomplete sensitivity list
  3. Incomplete branch & incomplete output assignments → if/else either not present

* LAB : 4 10/09/24

⇒ ERROR : multi driver error
same variable driving two blocks
Synthesis error

eg:
```
always @(posedge clk)
    begin
        if(rst_n)
            Q <= D;
    end

always @(negedge rst_n)
    begin
        if(!rst_n)
            Q <= 1'b0;
    end
```

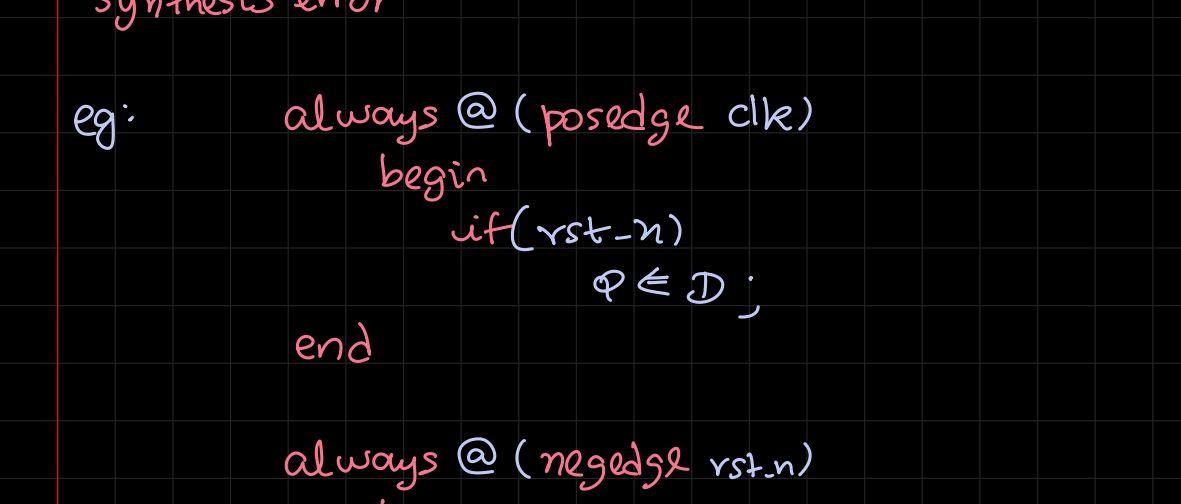
• Q is being updated by two blocks simultaneously

Parameters

```
module something(
    parameter foo = 1'b0
    )
```

* Digital clock (minute : seconds)
0 → 59 ↓ ↙ 0 → 59
(behavioural modelling) 6 bit up counter
always @ ↙

HW: modify the digital clk so that the output of CMT block is 16.777 MHz
Clock management time ↙
24 bit
size of the counter



sec_reg : output of sequential ckt (flip flop)
sec_next : output of combinational ckt
↓
hence we cannot initialise it
eg: we do not initialize output of AND ckt

if we initialize it, the tool might ignore the 2nd always block (the one whose o/p is sec_next)

ff: takes the next value and assigns to the current value

note: we are making the minutes logic and seconds logic in separate always blocks due to parallel processing

* Lecture: 10 10/09/24

- **Behavioural modelling for comb. ckt**
  (1) multi Driver error
  (2) Incomplete sensitivity list
  (3) Incomplete branch and incomplete o/p assignments

SOLUTIONS (≡ Guidelines)

① → have separate always block for each identifier

→ do not update multiple identifiers in one ALWAYS block

② designing AND gate

```
always @ (in1)
begin
    out = in1 & in2;
end
```

wrong. comb. ckts depend on all outputs (i.e. in1 & in2)

Note: we are just explaining our logic through code to the tool. Some tools might give warning but some might not. Hence resulting in incorrect logic

→ use always@(*) for combinational circuits.

* Note: leaving out an input trigger might result in a sequential circuit

③

```
always @ *
  if (a>b)
     gt = 1'b1
  else if (a==b)
     eq = 1'b1
```

→ intention: designing combinational circuit

* Problem 1: 2 outputs of one always block
* Problem 3: At each condition, only one of the two variables are getting updated and hence the other variable is stored in memory which we don't want because sequential
* Problem 3: code not considering what to do in the else case

→ assign values to all variables in each condition

→ deal with all cases in an if else block using else and in a switch block using default

→ OR we can initial vars to a value in the start of an always block

another example:

```
reg [1:0] s;

case (s)
   2'b00 : y = 1'b1;
   2'b10 : y = 1'b0;
   2'b11 : y = 1'b1;
endcase
```

not considering case where s = 2'b01

Solution: either define all cases or use default keyword
default: y = 1'b0;

CASE	IF-ELSE
All cases are checked simultaneously	Priority based

* CASE
  - full case: all possible outcomes are accounted
  - parallel case: all stated alternatives are mutually exclusive

eg:

```
case (sel)
   2'b11 : out <= a;
   2'b10 : out <= b;
   2'b01 : out <= c;
   default : out <= d;
endcase
```

full case ✓
parallel case ✓

eg2: combinational circuit

```
case (sel)
   2'b1? : out <= a;
   2'b?1 : out <= b;
   default : out <= c;
```

full case ✓
parallel ✗ because of ambiguity when sel is 2'b11

note: for sel = 2'b11 ⇒ out = a because of higher priority

summary

- If an always block executes and a variable is not assigned
  → variable has to be stored
    - not combinational ckt anymore
    - unnecessarily complex
    - might not be synthesizable

- Use blocking assignment for combinational circuits

* BLOCKING / NON-BLOCKING

→ Note: non blocking works only behavioural modelling i.e. always / initial block

① BLOCKING
statements are executed in the order they are specified in a sequential block

does not blocks execution in a parallel block

⇒ RULE

(1) Always @(*) : use blocking

(2) Always @ (posedge clk) : use non-blocking

eg)

```
always @ (posedge clk)
  begin
     reg1 <= #1 in1;
     reg2 <= @(negedge clk) in2^in3;
     reg3 <= reg1;
  end
```

note: the values in1, in2 and in3 are stored at posedge clk

hence reg3 will have the previous value of reg1 and not in1

also, it does not matter if in1 & in2 changed when clk hit neg edge, it will still take the value at initial pos edge to calculate reg2

note: this code isn't synthesizable

eg)

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

note: both always block execute at the same exact time since they are parallel blocks theoretically.

but on the hardware, it could happen that block (1) executes before (2) or vice versa

conditions: (1) both execute at same time
a = b
b = a

(2) ① then ②
a = b
b = b

(3) ② then ①
b = a
a = a

eg3)

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

case: (1) both at same time
a = b
b = a

(2) ① → ②

$a = b_{old}$
$b = a_{old}$

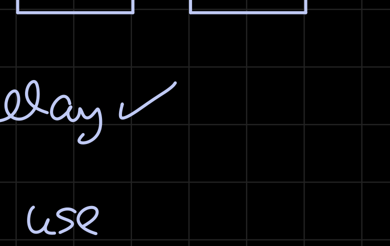
(3) ② → ①

$b = a_{old}$
$a = b_{old}$

eg)

blocking ✗

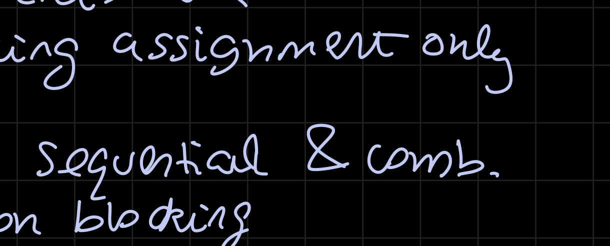
```
always @ (posedge clk)
  begin
     q1 = in;
     q2 = q1;
     out = q2;
  end
```



→ 1 clk cycle delay ✗

non blocking ✓

```
always @ (posedge clk)
  begin
     q1 <= in;
     q2 <= q1;
     out <= q2;
  end
```



3 clk cycles delay ✓

for sequential ckts use non-blocking assignment only

but for comb ckts use blocking assignment only

when doing both sequential & comb. we do non blocking

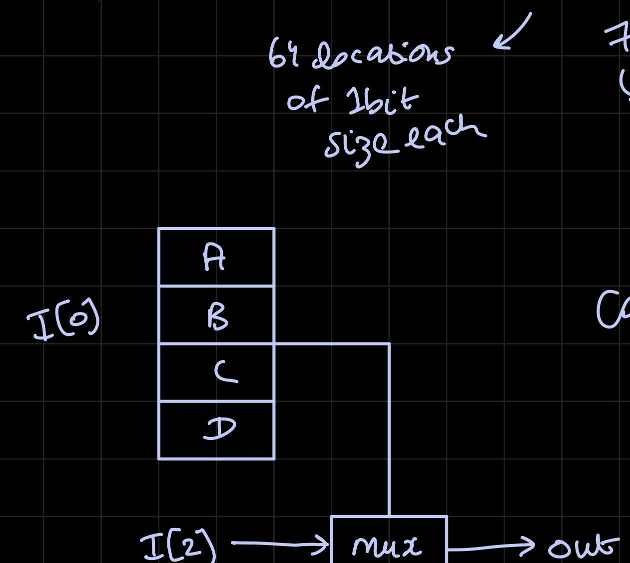
* LECTURE: 11 12/09/24

- **FPGA Architecture**

2 LUT outputs: O5, O6

overall outputs: _, _MUX, _Q
(MUX: through multiplexer; Q: passed through flip flop (synchronous) and delayed by 1 clk cycle)
eg: D, DMUX, DQ

if we combine 2 6bit LUTs: we get a 7bit LUT
(64 locations of 1bit size each → 128 locations)

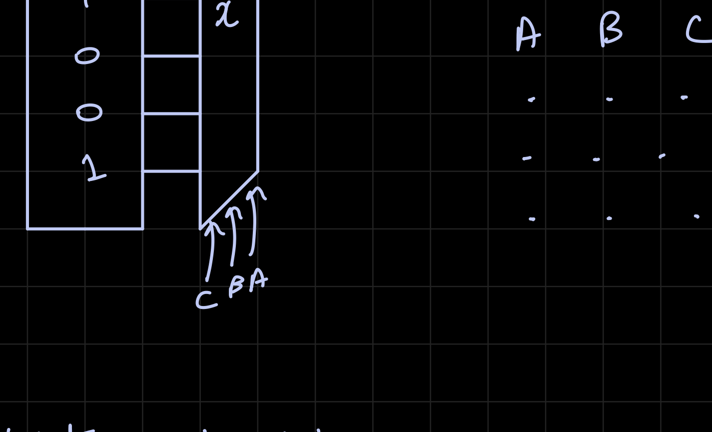


combining two 2bit LUTs to get a 3bit LUTs

eg: I = 100 ⇒ out = E
I = 001 ⇒ out = B

similarly, our 8-series FPGA, can combine all 4 6bit LUTs to get max one 8bit LUT

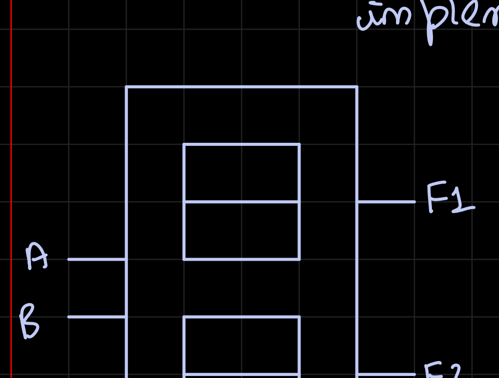
* 3input LUT



A	B	C	F1
.	.	.	0
.	.	.	1
.	.	.	0
			0
			1
			0
			0
			1

3input comb. ckt means 8 rows & 1 output col in truth table

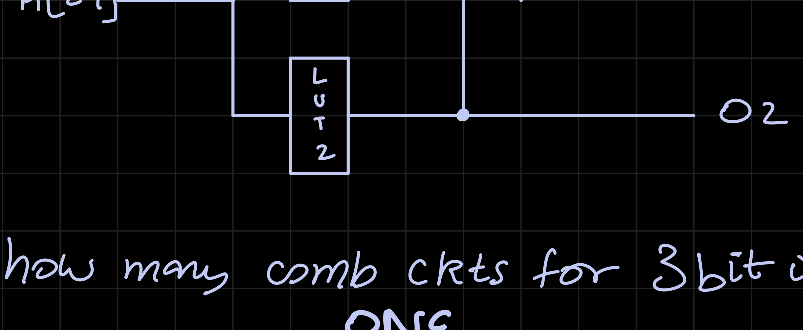
so, only 1 comb ckt can be implemented in one 3input LUT



this is 2input LUTs x2
how many comb ckt of 2 inputs can be implemented ?
1
because 2input ≡ 4 locations and so, either of the output can be used at once

how many comb ckt of 1 input can be implemented ?
2
($2^1 = 2$ location, $4/2 = 2$)

- **LUT 3 architecture**



how many comb ckts for 3bit input ?
ONE

how many comb ckts for 2bit input ?
TWO ??
set sel line to 0/1 const

limitation

2 x 2bit inputs: $f_1(A,B) = y_1$
$f_2(C,D) = y_2$

but we only have 3 inputs available at a time

but do note that no. of outputs match i.e. we have problem with input only

but, we can implement this:

$f_1(A,B)$
$f_2(B,C)$

3inputs ✓
2outputs ✓
8 locations ?
each LUT can store 4 locations

$f_1(A,B)$ OR
$f_1(A,C)$ AND
✗

but you need 8 locations in each LUT for implementing two comb ckts over LUT3 with 2bit input

$f_1(A,B)$ AND
$f_2(A,B)$ OR
} seems wrong but WORKS

since we now have 2inputs only, we need 4 locations only with each LUT which works here.

Conclusion: we can implement 2 comb ckts with 2bit input in LUT3 only if both the inputs are same

note: upper limit of no. of comb ckts is equal to the no. of outputs

max. no. of comb ckts that works with no constraint: 2

$f_1(A)$; $f_2(B)$

total locations required: $2^1 + 2^1 = 4$ ✓
inputs: 2 < 3 ✓
outputs: 2 ✓

Conclusion_2:

1 comb ckt with 3 inputs ✓

2 comb ckts with 2 inputs (with common inputs) ✓

2 comb ckts with 1 input ✓

* **Now what about LUT6 ?**

inputs	comb. ckts.	constraint
6	1	—
5	2	common input
3 & 2 (or less)	2	—

($f_1(A,B,C,D,E)$, $f_2(A,B,C,D,E)$ ← 5)

eg: $f_1(A,B,C)$
$f_2(D,E)$

eg: $f_1(A,B,C)$
$f_2(B,C,D)$
} works

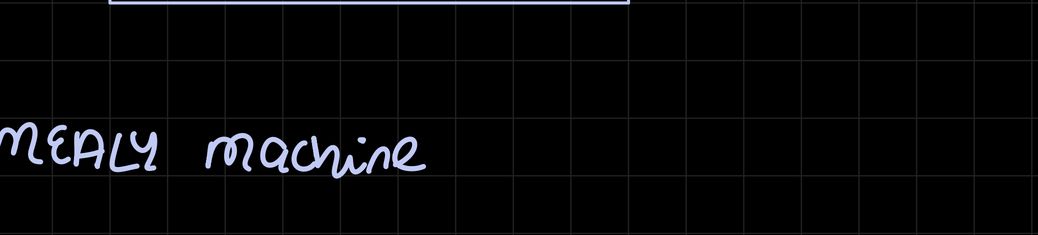
note: f_1, f_2, f_3 wont work because we only have 2 outputs but they give 3 o/ps

6inputs: 1 for sel
3 upper LUT
2 lower LUT

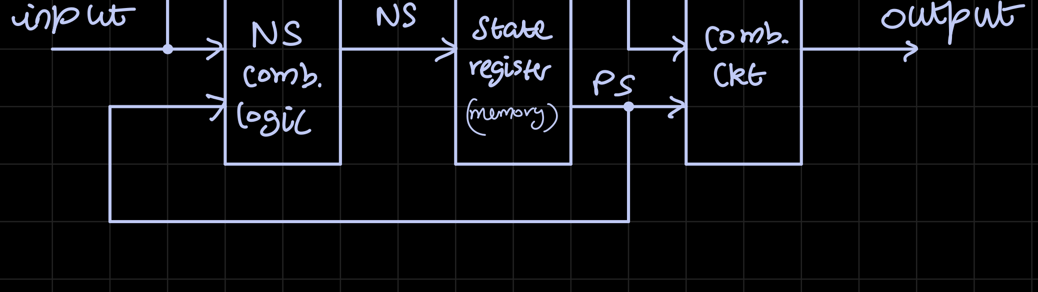
$f_1(A,B,C)$
$f_2(D,E,F)$
} ✗ because we have only 5 inputs (rest 1 is sel)

* **Finite state machine**

① MOORE machine



② MEALY machine



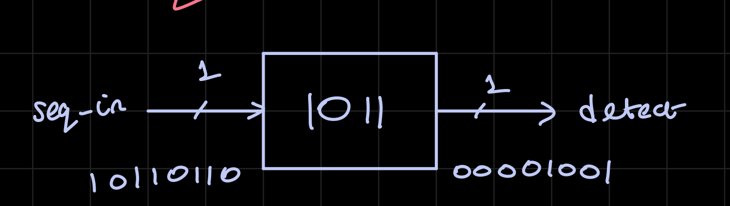
* LAB: 5 ⇒ design of sequence detector 17/09/24

FSM

midsem code: write a FSM code for something...

FSM : sequential circuit
|||
FFs + 2 comb ckts
↓
output + next state

1011 sequence detector

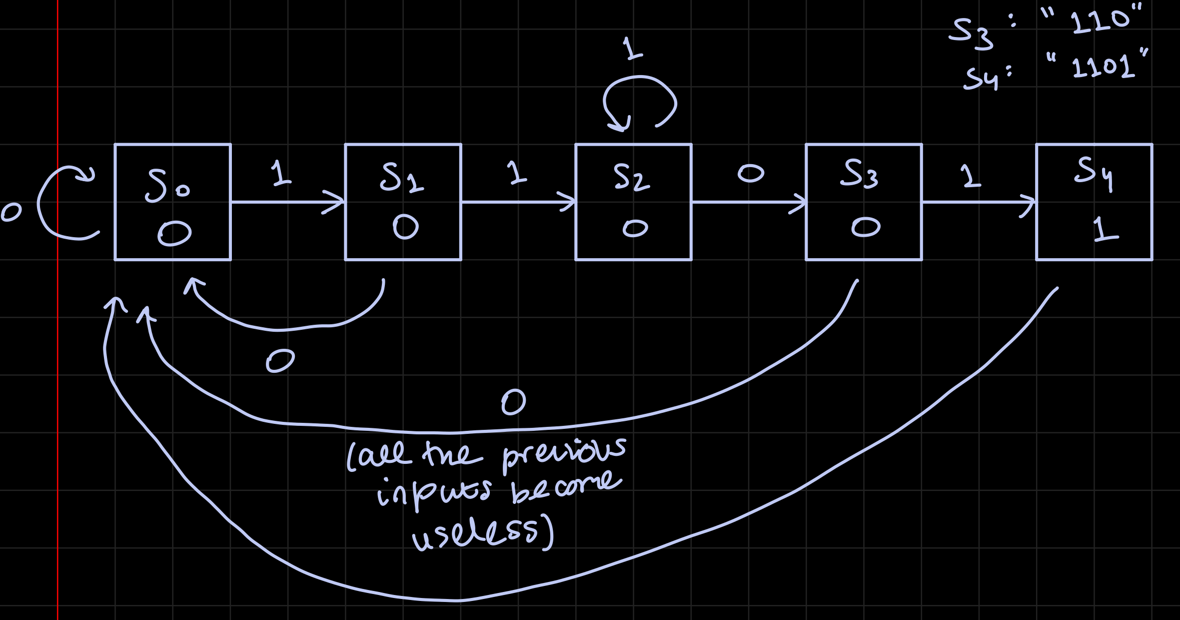


types: overlapping / non-overlapping

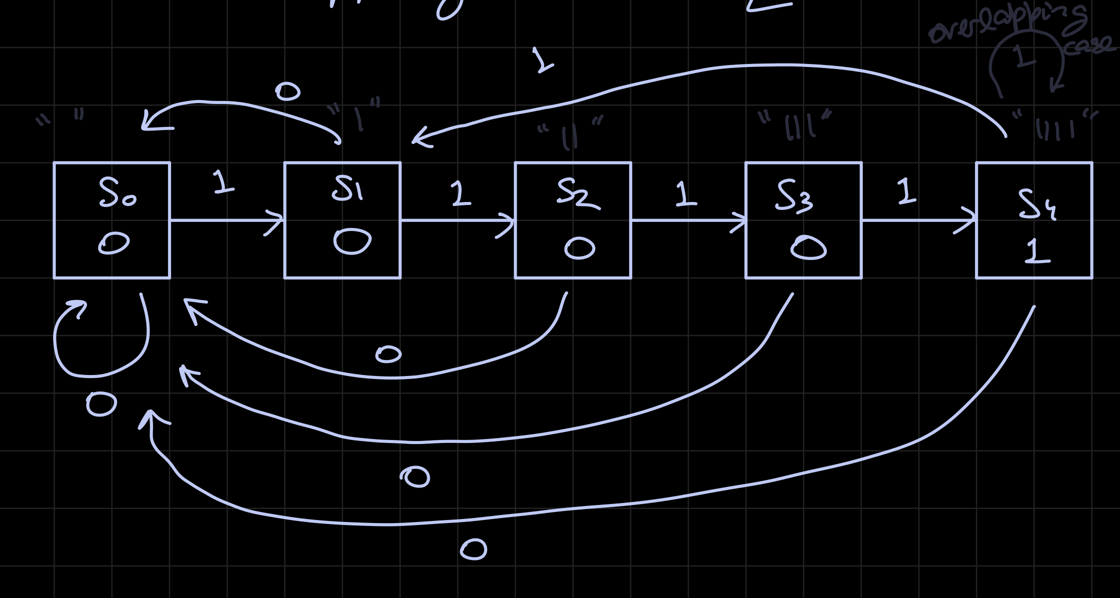
⇒ FSM: finite state machine
every state has a meaning

MOORE

S0 : " "
S1 : "1"
S2 : "11"
S3 : "110"
S4 : "1101"



non-overlapping 1111 seq detector



every state should be unique

5 steps

Verilog code for FSM (moore) (1101)

① module definition
2 comb ckt + 1 sequential ckt

② define variables for present and next state
size ≥ number of states

```
reg[2:0] present_state,
         next_state;

parameter S0 = 3'b000
          S1 = 3'b001
          S2 = 3'b010
          S3 = 3'b011
          S4 = 3'b100
```

* Three always blocks
because we have three blocks
1 seq + 2 comb

③ State Register always block

```
always @ (posedge clk or
          posedge clr)
  begin
    if (clr == 1)
        present_state <= S0;
    else
        present_state <= next_state;
  end
```

Note: non-blocking assignment for sequential ckts
and blocking assignment for comb. ckts

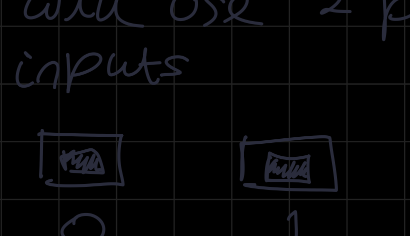
④ next state comb. ckt

```
always @ (*)
  begin
    case (present_state)
      S0 : if (din == 1)
             next_state = S1;
      ⋮        ⋮
```

⑤ output comb ckt

```
always @ (*)
  begin
    if (present_state == S4)
        dout = 1;
    else
        dout = 0;
  end
```

note: we will use 2 push buttons as inputs



Also, ILA not needed
(logic)
only counter 6bit will change from last lab
digital adder

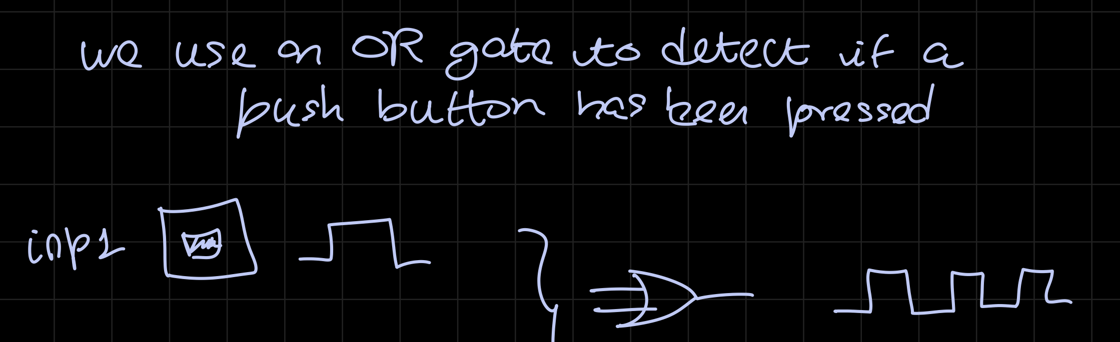
output of FSM: detect (1 bit)

for debugging: we will also output present state

input of FSM:

clear
1Hz clock
1 input or 2?
(when using push buttons)

we use an OR gate to detect if a push button has been pressed



but how do we detect which one was pressed

we can take either of the input lets say we take inp_1 as the other input

then at clk rising edge, if inp is high then inp1 is high else inp0 is high

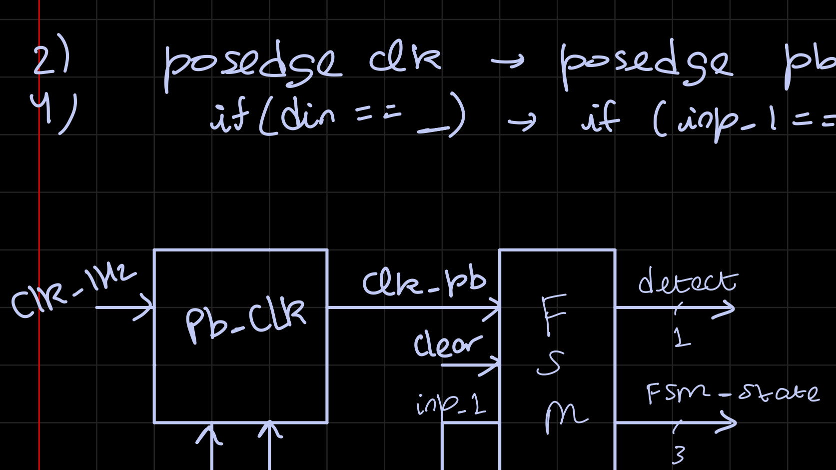
why did we not use a switch, because when we have pushed down, it assumes you are continuously sending 1 and vice versa

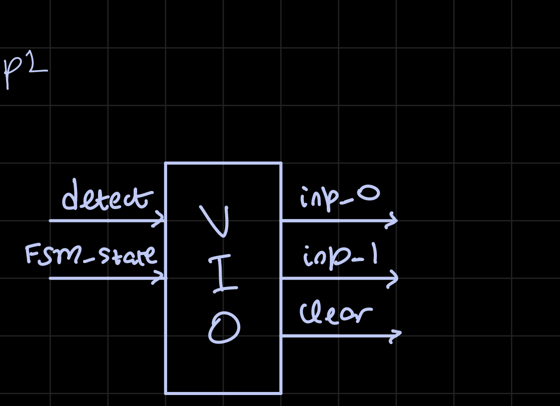
So, now inputs to FSM

clear
1Hz clock
pb clk
inp_1

Changes from above 5 steps

2) posedge clk → posedge pb_clk
4) if (din == _) → if (inp_1 == _)





* lecture: 12 17/09/24

⇒ SLICE ARCHITECTURE

Slice L	Slice M
comb. ckt ✓	comb. ckt ✓
memory X	memory ✓

note: momery
↳ LUT
↳ BRAM

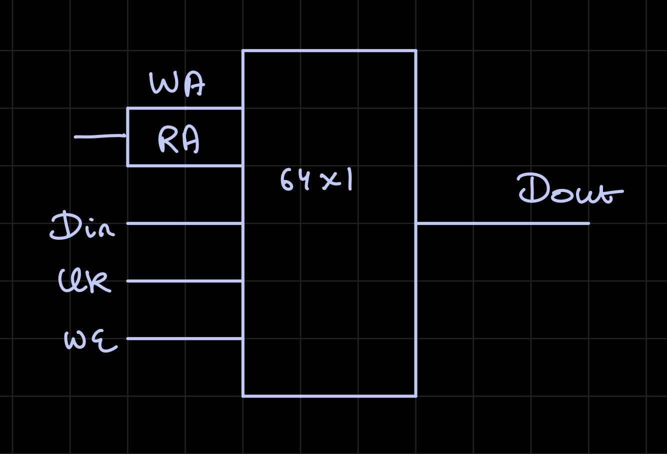
- **LUT**
  ↳ combinational ckt
  ↳ memory
  ↳ Shift register

① LUT as memory

- only in SLICEM
- Sychronous write operation, WEN (write enable) must be high
- asynchronous / sychronous read operation
  ↳ when passed through a FF

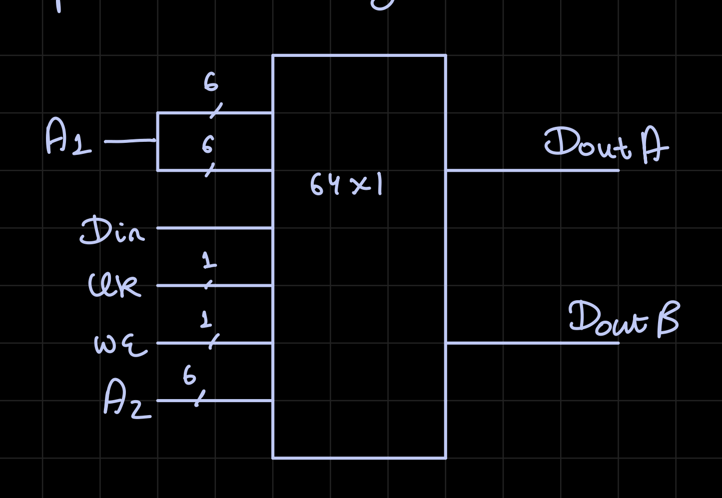
* Single Port

common address port for sychronous write and asynchronous read i.e. read and write addresses share the same address bus.



* Dual Port

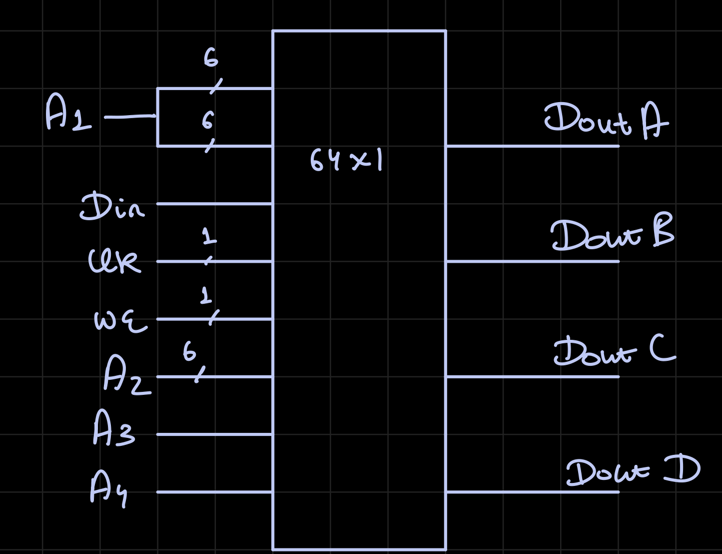
one port for async write + sync read
one port for async read



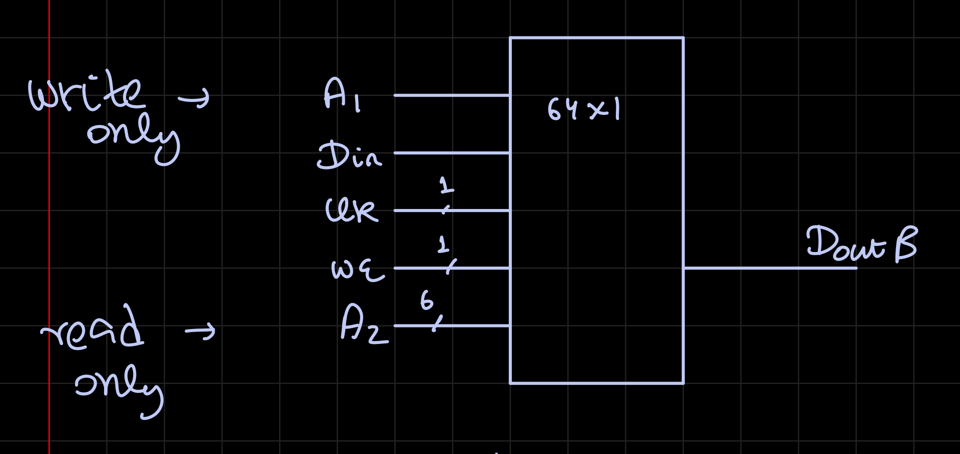
A1 can perform both read and write operation
where A2 can only do read operation

* Quad Port

one port for async write + sync read
three ports for async read



* Simple dual port



one port : sync write only
one port : async read only

notation: 64x4 quad port
↙ 64 locations
4bit : size of each location
size of din, dout = 4bits
quad port → 1 read/write + 3 read

Note: In LUT architecture,

the topmost block has its read and write address lines shorted whereas not the case with below 3 blocks.

below 3 blocks' read addresses are independent but write addresses are connected to the write address of topmost block

① Clock of all LUTs is common

② WEN (write enable) of all LUTs is common

③ Write addresses of all LUTs is common

So, when you perform write operation on any LUT, write operation will happen on all LUTs
(≡ observation ; might not be correct tho)

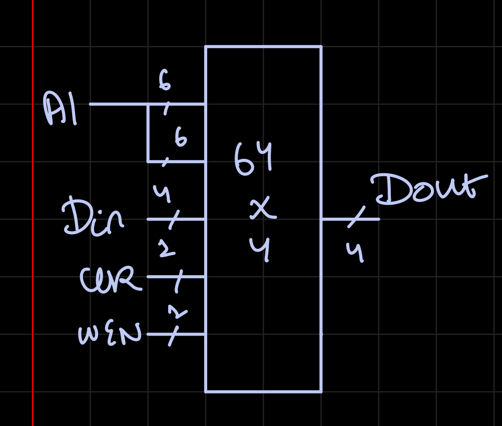
④ Read Addresses of all LUTs is different

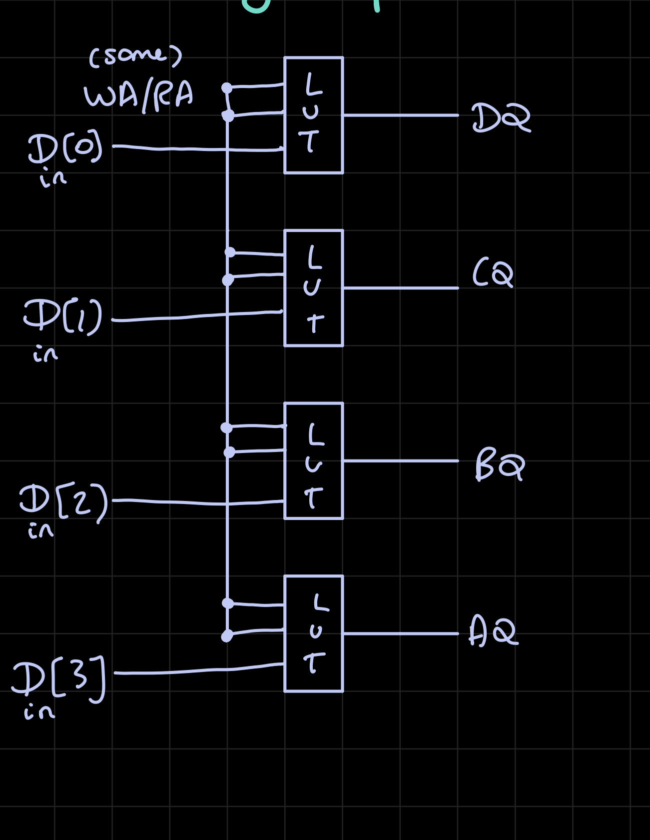
⑤ For first LUT, WA and RA are same

* LUT as MEMORY : 64x1 single port

How many LUTs needed to implement this?
≡ one 64x1 LUT ✓

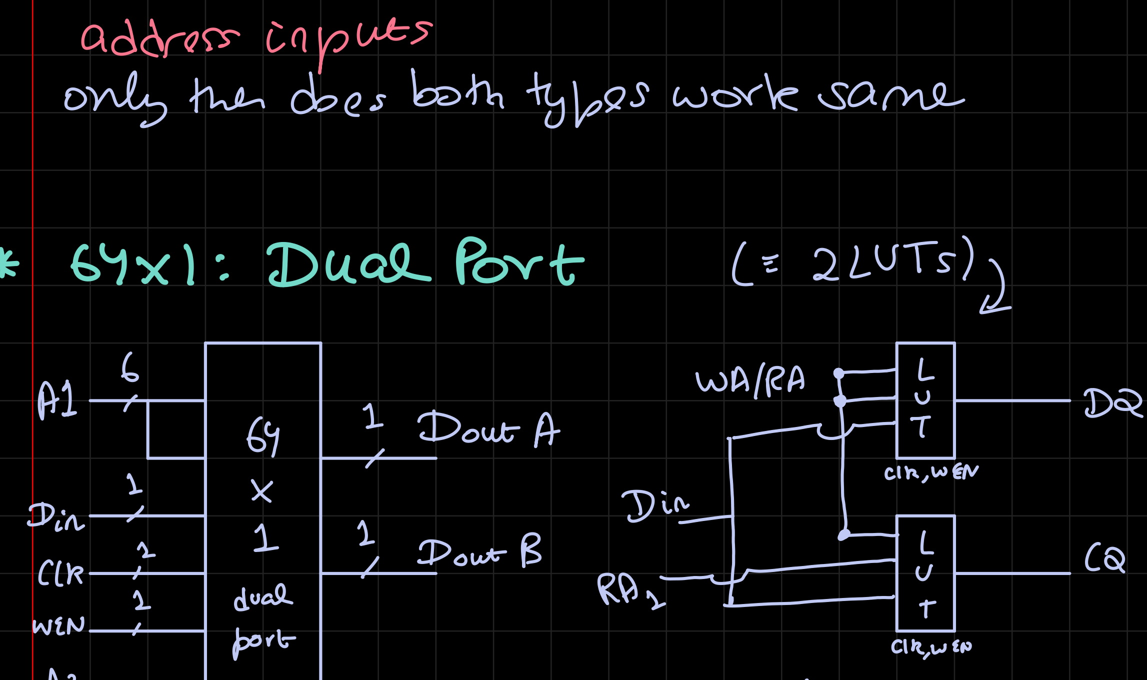
* LUT as MEMORY : 64x4 single port





using single SLICEM we can implement one 64x4 single port ✓
but we can implement 4 64x1 single port only if we have shared clock, write enable and shared read and write address inputs
only then does both types work same

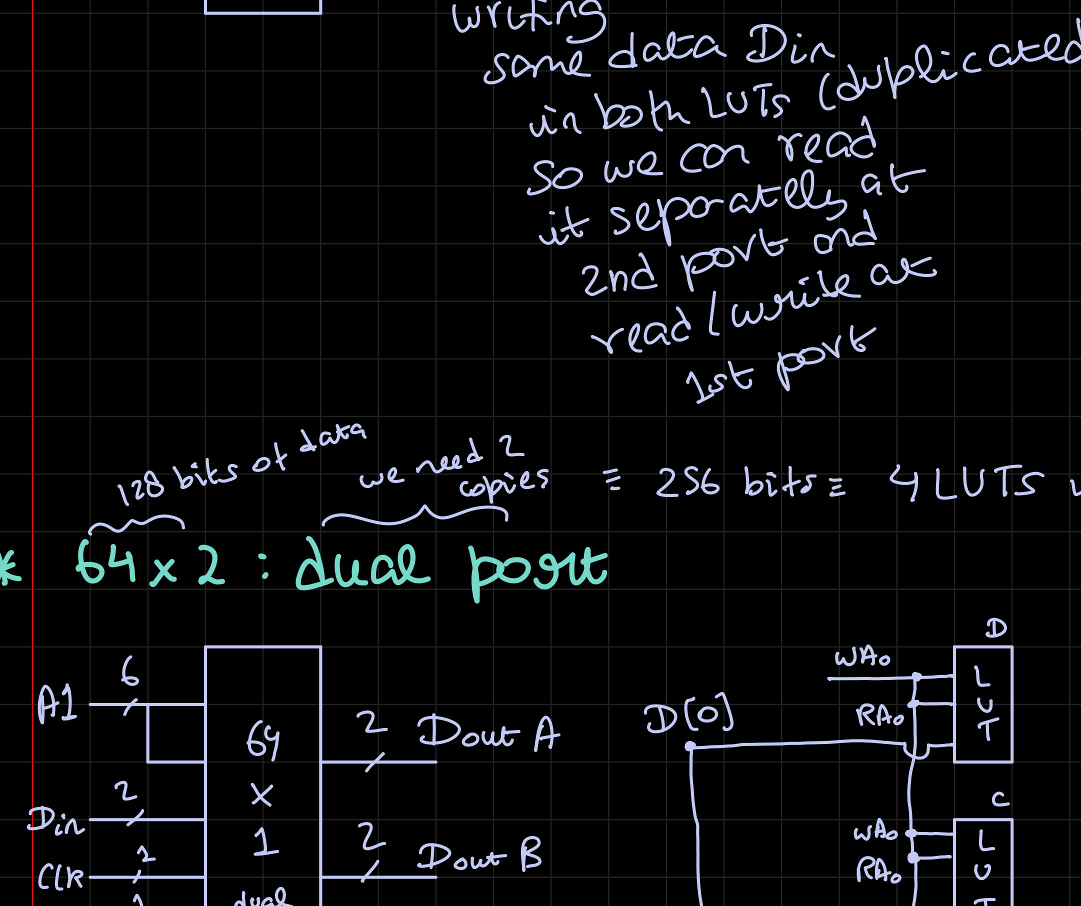
* 64x1 : Dual Port (≡ 2LUTs)



writing same data Din in both LUTs (duplicated) so we can read it separately at 2nd port and read/write at 1st port

128 bits of data, we need 2 copies ≡ 256 bits ≡ 4 LUTs ✓

* 64x2 : dual port



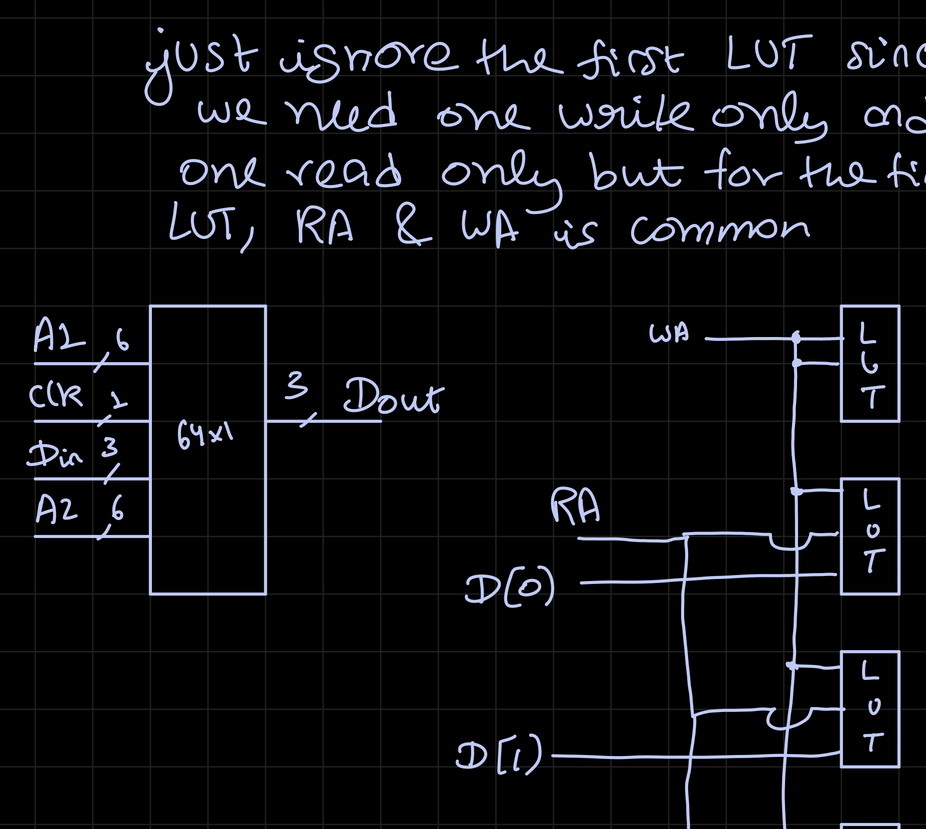
→ replicate the data three more times

* 64x1 : quad port ✓

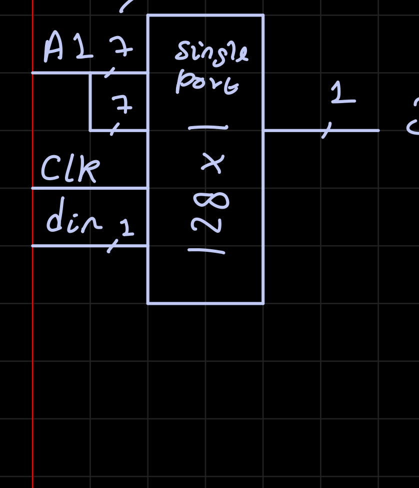
note: we cannot implement 64x2 quad port
∵ not enough LUTs

* 64x3 : Simple dual port

just ignore the first LUT since we need one write only and one read only but for the first LUT, RA & WA is common



* 128x1 : single port



→ but each LUT can have at max 6 bits size for addresses

So, we use 2 top and 2 bottom LUTs for first 64 locations and bottom 64 locations respectively

but WA is common all LUTs
Therefore, we have two "LUT select" bit W6 & W7 that cycle through LUTs

256 x 1	quad port X	
128 x 1	quad port X	
128 x 1	dual port ✓	⇒ required 128x2 ⇒ 4 LUTs ✓
32 x 16	single port X	⇒ input > 4 and 256 < 512

* lecture : 13

19/09/24

⇒ LUT as Shift Register (SRL)
(serial in serial out)

```
module SRL (
    input in,
    input clear,
    input clk,
    output QD;
    )

    wire QA, QB, QC, QD;

    always @ (posedge clk or posedge clear)
        if (clear)
            QA <= 0;
        else
            QA <= in;

    always @ (posedge clk or posedge clear)
        if (clear)
            QB <= 0;
        else
            QB <= QA;

    always @ (posedge clk or posedge clear)
        if (clear)
            QC <= 0;
        else
            QC <= QB;

    always @ (posedge clk or posedge clear)
        if (clear)
            QD <= 0;
        else
            QD <= QC;

endmodule

    D_FF(in, QA);
    D_FF(QA, QB);      } same as above
    D_FF(QB, QC);      } if D_FF defined
    D_FF(QC, QD);      } already
```

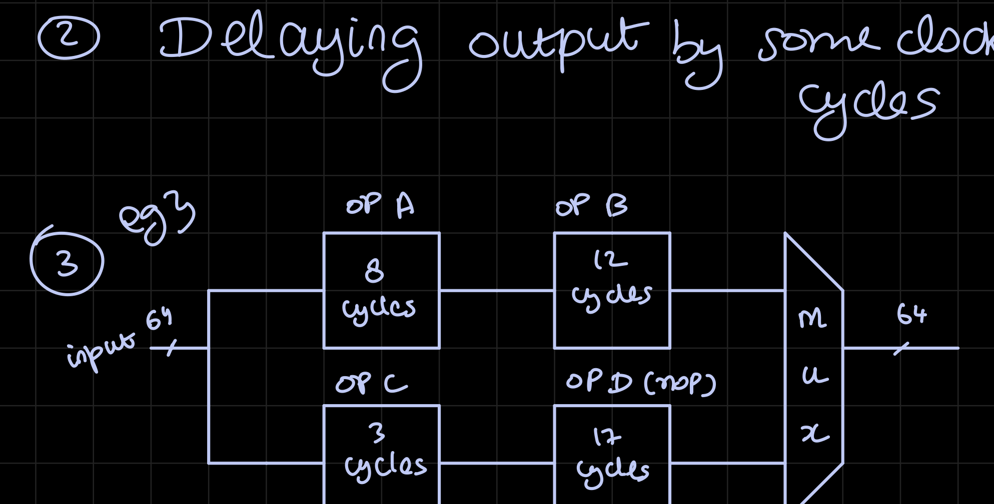
uses of shift registers

① multiplying / dividing by power of 2

* OR / by 2^x

② Delaying output by some clock cycles

③ eg3



64x17 flip flops required

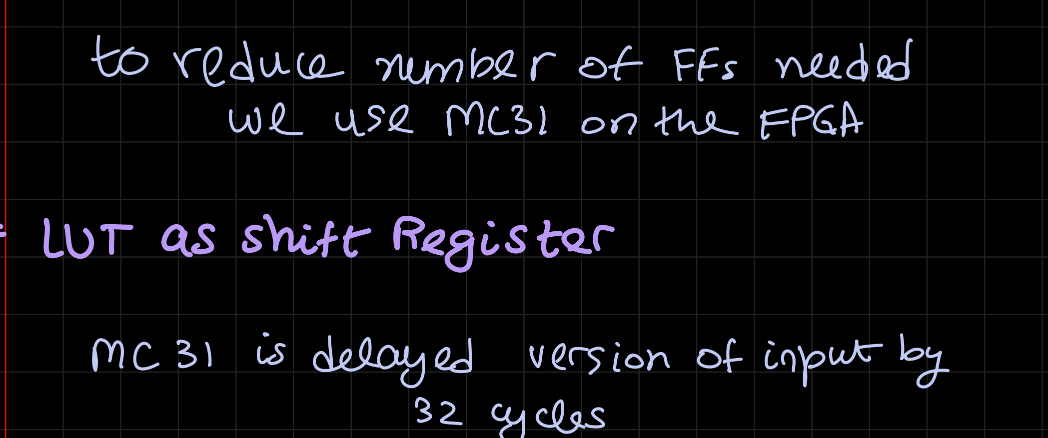
1 slice has 8 FFs
so, 8x17 slices required
to delay 64 bit signal by 17 cycles

OP A, B, C are some operations that take some clk cycles to process input but to hold the functionality of the given circuit, i.e. both pathways reach mux at same time, we need 17 clk cycle delay.

to reduce number of FFs needed we use MC31 on the FPGA

* LUT as shift Register

MC 31 is delayed version of input by 32 cycles



A[4:0] → controls how much cycles Q output is delayed by

However, SHIFTOUT is always delayed by 32 clk cycles

* note: in the FPGA, MC31 of an LUT is connected to data-in (DI1) of the LUT below it ($n-1^{th}$ LUT only)

- Sync write operation
- fixed read access to Q31 (≡ MC31)
- dynamic read access through 5bit address bus (LSB unused)
- We cannot have set/reset functionality
- Useful for smaller shift registers
- any of the 32 bits can be read out asynchronously by controlling the address.

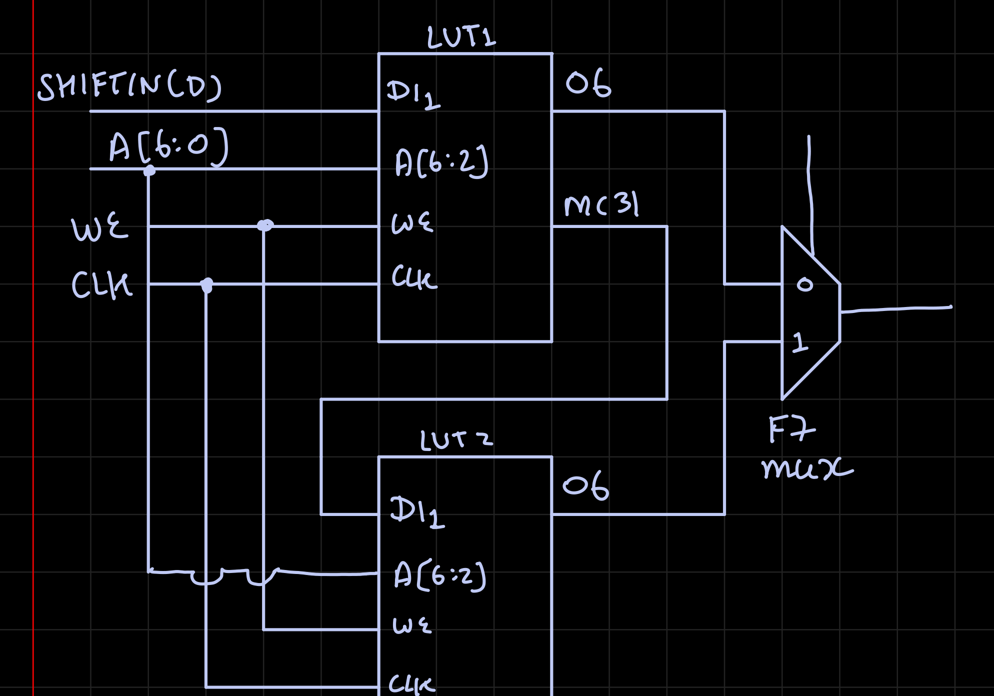
* If we write a verilog code without reset/clear, the tool can implement the code with either FFs OR LUTs whereas code with reset will be implemented using FF only.

when shifting, it just changes the address of the data in memory one bit to the right

One LUT can implement 32 bit shift register.

* 64 bit shift register using LUTs

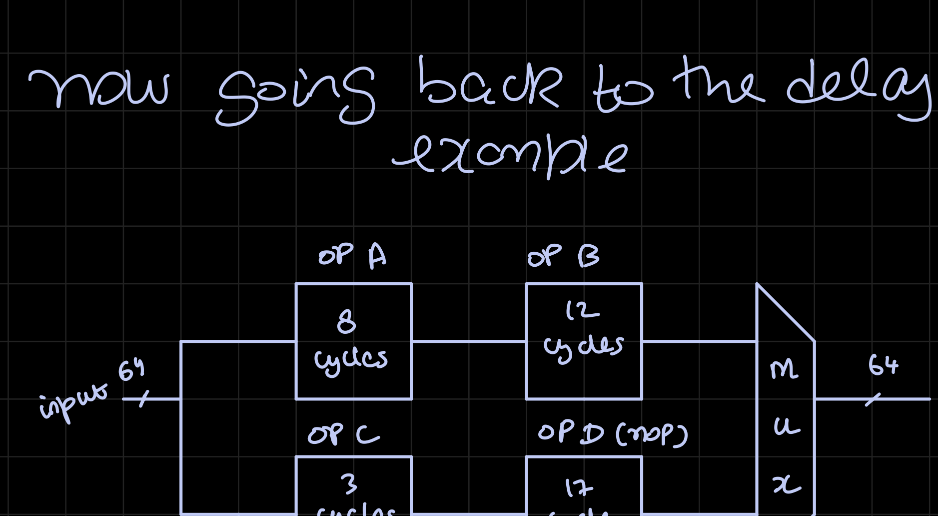
we use 2 LUTs



1 slice can only implement max 128 bit shift register because each slice has 4 LUTs and each LUT can implement 32 bit shift register

for 96 bit shift register = we need 3 LUTs

now going back to the delay example



64x17 flip flops required

1 slice has 8 FFs
so, 8x17 = 136 slices required
to delay 64 bit signal by 17 cycles

every slice has 4 LUT

64 bit input → 17 delay

max 32 bit delay by one LUT
but we need 17 bit delay so we use address bus

so, 1 slice can give 4x17 (bit input x delay)

but we need 64x17

So, 16 slices needed

136 → 16 slices needed
(FF) (LUT)

68 → 16 : CLBs needed

we just need to remove reset/clear functionality and we need to tell vivado/tool to prefer LUT

in one CLB we have one SLICEM

project settings → synthesis
↓
shift_min_size : 3
max_dsp : -1
auto

In one CLB, we have

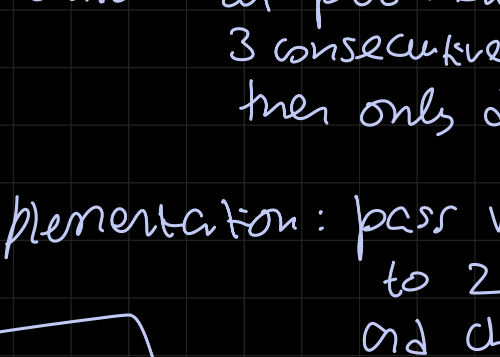
2 slices
8 LUTs
16 FFs
64x4 ⇐ 256 bit mem
128 bit shift register

* PUSH BUTTON 0 → 1 → 2 → 3 ....

```
module counter (
    input pb,
    output [2:0] out
    )
    always @ (posedge pb)
        begin
            out <= out + 1
        end
endmodule
```

but we get ① → ⑤ → ⑦

due to push button debounce

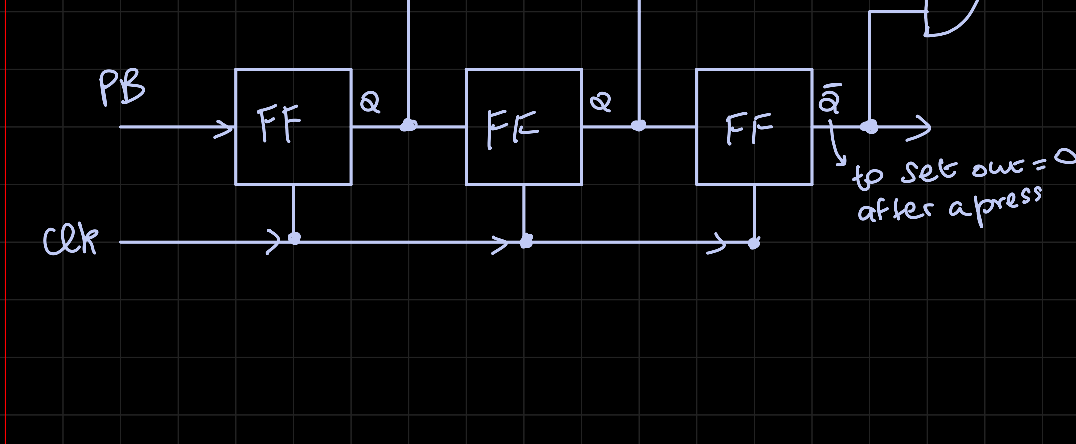


when we push once, it takes some time to settle

solution: if push button is high for 3 consecutive clock cycles, then only do the logic

implementation: pass values initially to 2 FFs and 1 FF and check with current value

Switch debounce



* Lab :6 ⇒ AXI Interface 24/09/24

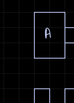
- basics of AXI Interface
- design of floating point arithmetic

$$y = \frac{1}{\ln(x)}$$

- HW: ACCELERATOR: $z = \sqrt{x} + \frac{1}{\log y} + 1.5$
- compare which is faster: processor (C/C++) vs FPGA (verilog)
  for communication between the two ⇒ AXI interface

* Advanced Extensible Interface (AXI)

data: A[30]
toggle each
10 bytes each

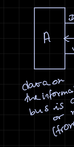


like I2C interface
data sent on clock edge

Problem:

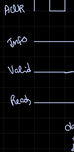
- B does not know when transmission should end and parse the data or how much data A should send
- No feedback mechanism from B→A if data received correctly or not
- A might not have all data ready at each clock rising edge. A should have control to pause transmission.

> Note: SPI solves all these problems



data or the information bus is correct or not (from A)

it B is ready to accept the data



data is valid since B not ready

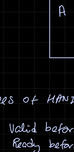
VALID before READY handshake

when does transfer occur?
when valid is high, ready is high and clock has rising edge

PROBLEM Now:

- Starting and ending point not known to B

& we add a "LAST" signal which goes high when the last byte of data is being sent. Whenever the last byte of data is being sent last bit goes high alongside valid signal independent of ready (B) signal



* Types of HANDSHAKES

- Valid before Ready
- Ready before Valid
- Valid with Ready

* PROBLEM: (SRC) (DEST)

- Valid should not wait until Ready becomes one else in dead lock!
- once valid is set to high, it CANNOT be reset until handshake happens
- DEST can set Ready independent of VALID it is allowed to go back to low after being set to high even before VALID is set to high

> Note: Clock is needed in order to know when transmission is occurring (indicated by the positive edge of the clock)

In today's lab ⇒ uni-sided data transfer from A to B

* AXI STREAM

$$y = \frac{1}{\ln(x)}$$

Data
VALID : by log IP
READY : by inverse IP
LAST

① LOG IP
② INVERSE IP
③ connect ① & ②
④ connect with testbench

SRC connected with log IP

DEST connected with inverse IP

* Verilog:

Latency = 23 ⇒ means it takes 23 clock cycles to carry out log function

Slave interface:	data	input
	Valid	input
	ready	output
Master interface:	data	output
	Valid	output
	ready	input

Latency for reciprocal : 30 cycles

total cycles delayed to find output:

53 cycles

clk cycle = 10ns per

So, total delay: 530ns

* Lecture: 14

24/09/24

- **BRAM**

in the FPGA, we can store data using:
(1) LUT — 64 bits
(2) FF
(3) BRAM — 36Kb

- for BRAM, we can use BRAM IP directly from vivado's IP catalog
OR we can code our logic for memory with specific conditions for synthesis with BRAM

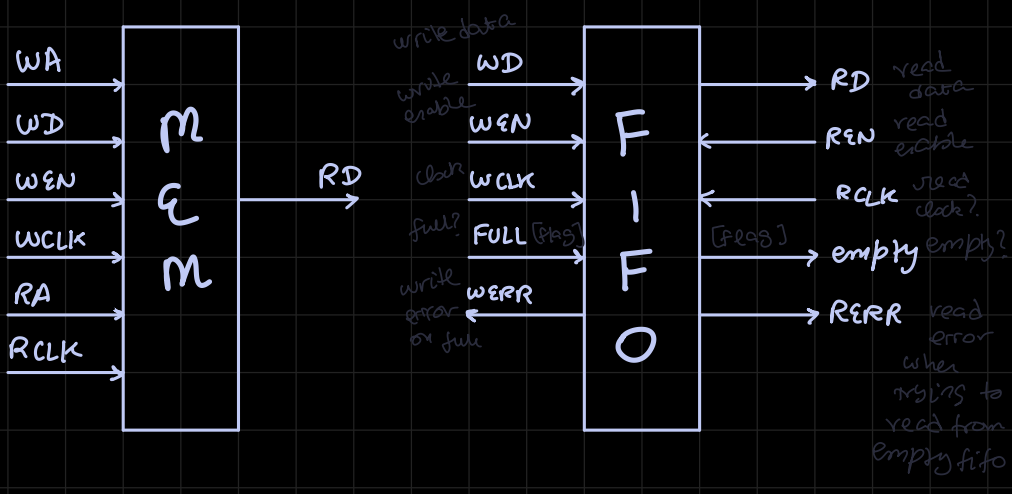
ultra RAM — 288Kb

in one LUT we can store 64 bits of memory
in one slice ⇒ 256 bits
in one CLB ⇒ 256 bits
in one BRAM ⇒ 36Kb = 36 x 1024 x 8 bits

also, in some boards, we have ultra RAM (can store max 288Kb mem) but ours doesn't have it.

- We had async read & sync write in LUT
- For BRAM, we have fully synchronous operations
- configurations
  - True dual port
  - Simple dual port
  - Single port

- Each BRAM can be segmented as:
(1) 36Kb BRAM : we can access data at any place by providing address
(2) 36Kb FIFO : sequentially access only
(3) 18Kb BRAM x2
(4) 18Kb FIFO + 18Kb BRAM



We can read the same data multiple times anytime

once the data is read, it is pushed out of the memory

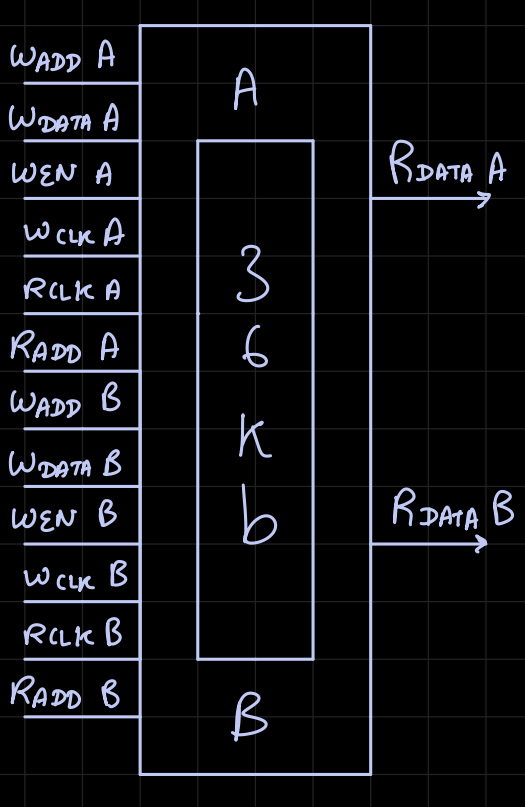
needs internal counter to get from where to read / write data

requires a separate controller alongside memory

* Integrate cascade logic to build larger memories eg: to get 72Kb BRAM

- **BRAM Configuration**

two ports: A and B
each port can do read / write operation
multi-bit read/write addresses ≡ configurable memory

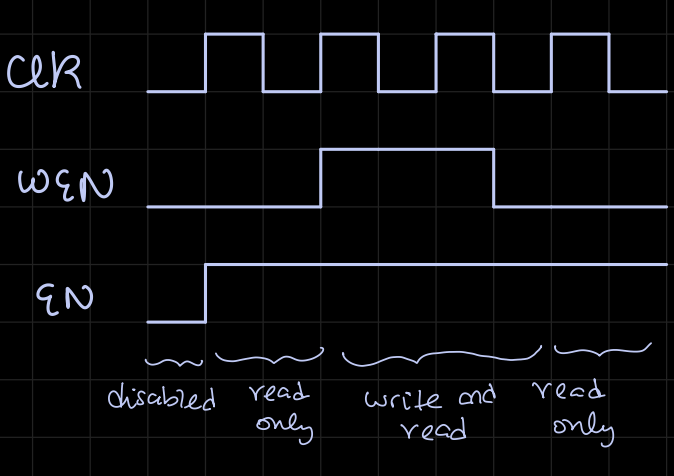


both ports are symmetrical and totally independent, sharing only the shared data.

each port can be configured in one of the available widths, independent of the other port

Read port width can be different from the write port width.

BRAM has a global enable signal
⇒ read/write operations are carried out only when EN ≡ high ⇒ else NOP

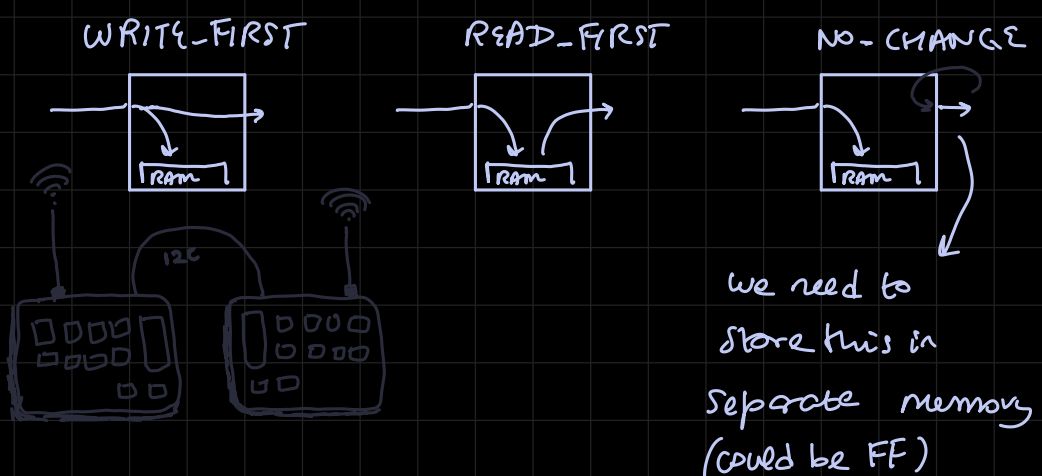


- **configurations (WRITE MODES)**

During write operation, output can have either newly written data / previous output or have no change

- WRITE_FIRST — perform the write operation and data written is passed as o/p to DOA
- READ_FIRST — perform write operation but old/prev data is available at output first
- NO_CHANGE — DOA (output) holds its previous values (power saver)

power efficient ✓

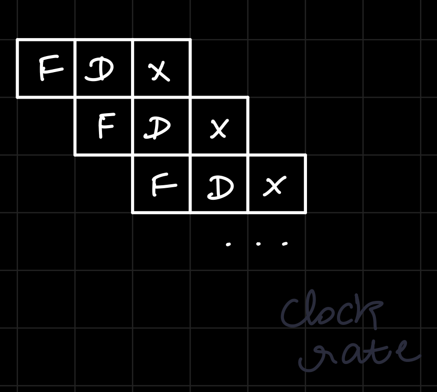


* lecture : 15 26/09/24

⇒ BRAM

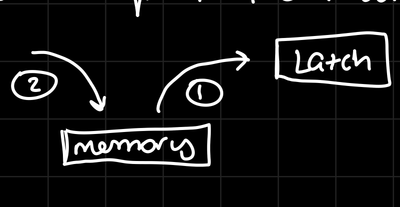
- fully synchronous operations
- memory access (read/write) is controlled by the clk
- to support read-first and no-change modes, we have a latch and register at the end.

• PIPELINING

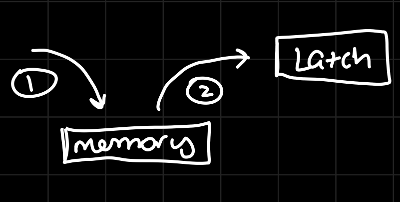


increase in HW cost?
↳ No
increases efficiency / throughput of existing resources ✓

- output latches will be loaded only when write mode = read-first and write-first
  not when mode = no-change
- first data gets loaded into output latch then write operation in READ_FIRST



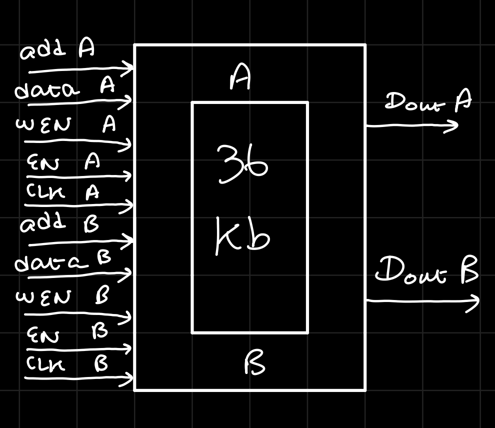
- first write operation to memory and then reads updated data to output latch in WRITE_FIRST



- When WE is inactive and EN is active, read operation happens and the data referenced by the address bus appear on the data_out bus regardless of write mode

• TYPES OF BRAM

① Dual Port block BRAM



eg: memory with 1K locations and each 8bits
⇒ we need 10bits for address bus and 8bits for data bus

now, for 2K → 4 bits
add bus = 11 bits
data bus = 4 bits

now, for 4K → 2 bits
add bus = 12bits
data bus = 2 bits

now, for 8K → 1 bits
add bus = 13bits
data bus = 1 bit

using parity bits we can detect for any errors in data

In our BRAM, it supports one parity for every byte of data

So, for 1K locations with each size 8 bit, we can add an additional parity bit

So,
1K, 8bit + 1
512, 16bit + 2
256, 32bit + 4 (parity bit)
⋮

for 36kb of memory, we can have these configs:

1K → 32 + 4 (36K/36)
2K → 16 + 2 (36K/18)
4K → 8 + 1 (36K/9)
8K → 4 + 0
16K → 2 + 0
32K → 1 + 0

maximum locations so, addr bus ⇒ 2^{15}
15 bits

max = 32 bit data input

allows 2 adjacent BRAMs to cascade hence ADDRA is 16 bits and not 15 bits

why do we have 4bits for WEN?
eg: WEN = 1000 i.e. enabled for one byte
in 1K → 32+4
we can write one byte at a time using this
eg: FXXX (garbage value)

for WEN = 0110
⇒ XABX

Saves time ✓

This is called **Byte-wide write Enable**
⇒ allows writing eight bit (one byte) portions of incoming data.

if last read data = ABCD
new data_in = XFXX
with mode = READ-first, output = ABCD
but in WRITE_FIRST, output = AFCD ↙
Combination of new and previous value

⇒ write enable is only for the data and hence WEN.size() ≠ 5
assumed that data-in will have parity bit included

Dual Port

- Two seperate write/read ports
- each BRAM (36Kb) can be divided into two individual 18Kb BRAM

configurations:

locations	size of each (bits)	
32K	1	+ 0
16K	2	+ 0
8K	4	+ 0
4K	8/9	+ 1
2K	16/8	+ 2
1K	32/36	+ 4

② SIMPLE DUAL PORT BRAM

- one read-only port and one write-only port
- independent clocks btw both ports
- each BRAM (36K) can be set to simple dual port mode (SDP) and locations can be doubled i.e. 72K locations max

not using 36K locations because addr bus should be power of 2 ←

locations	size of each (bits)	
32K	1	+ 0
16K	2	+ 0
8K	4	+ 0
4K	8/9	+ 1
2K	16/18	+ 2
1K	32/36	+ 4
512	64/72	+ 8

③ SINGLE PORT BLOCK RAM

- Just remove port B from dual port BRAM to achieve single port BRAM with port = A
- So, we have 18Kb single port BRAM

locations	size of each (bit)	
16K	1	+ 0
8K	2	+ 0
4K	4	+ 0
2K	8	+ 1
1K	16	+ 2

* cascading only supported in dual port memory

• CONFLICT

- 7 series FPGA has a true dual port BRAM
- No problem when both ports are reading (even if reading from same location)
- But both ports writing different data to the same memory location
  ≡ data corruption
  non-deterministic data
- When one port is writing and other is reading, write is successful but the resulting read value may vary depending on clock cycles and write mode

Clock Type	write mode PORT A	write mode PORT B	WEN A	WEN B	Dout A	Dout B	resulting mem value
common	RF WF NC	RF WF NC	0	0	old	old	no change
common	RF	RF WF NC	1	0	old	old	new (A)
common	WF	RF WF NC	1	0	new	ambiguous "X" we dont know if it will read before/after op	new (A)
common	NC	RF WF NC	1	0	no change	X	new (A)
common	RF WF NC	RF	0	1	old	old	new (B)
common	RF WF NC	WF	0	1	X	new	new (B)
common	RF WF NC	NC	0	1	X	no change	new (B)
common	RF WF NC	RF WF NC	1	1	X	X	X

↳ Data corruption case

both ports are symmetrical in context of this table i.e. when A is in read mode and B has one of the 3 write modes, the output will be similar as above but just reversed

Clock Type	write mode PORT A	write mode PORT B	WEN A	WEN B	Dout A	Dout B	resulting mem value
independent	RF WF NC	RF WF NC	0	0	old	old	no change
independent	RF	RF WF NC	1	0	old	X	new (A)
independent	WF	RF WF NC	1	0	new	X	new (A)
independent	NC	RF WF NC	1	0	no change	X	new (A)
independent	RF WF NC	RF	0	1	X	old	new (B)
independent	RF WF NC	WF	0	1	X	new	new (B)
independent	RF WF NC	NC	0	1	X	no change	new (B)
independent	RF WF NC	RF WF NC	1	1	X	X	X

↳ Data corruption case

- even reading should be avoided when the other port is writing

- **Lab 7 :** Axi (protocol) FFT
  (advanced extensible interface)

CPU → parallel processing
CPU + GPU → accelerators (specific use cases)
CPU + HW {ASIC, FPGA}
CPU + co-processor (ARM)
↳ serial processing

Fourier Transfer

Processor vs FPGA which will be faster?
↳ depends on which algorithm used.

{ implementation on FPGA: verilog
implementation on processor: C/C++ (:fire: ↓ ewww)

hardware/software co-design

"I am speed" - Dennis Ritchie, 1999 ≡ McQueen

Heterogeneous System on a Chip

- **Objective**

→ basics of fourier transform
→ design floating point FFT for input of size 8 elements

HW1: design f.p. FFT for input of size 32 elements

HW2: design $z = \frac{x}{y} + \sqrt{\frac{2\ln(t)}{y}}$

x, y, t : +ve int
$t > y > x$

- **Fourier Transform** (→ from one domain to another)

decompose a signal to sinusoidal terms

time domain → freq. domain

in 4G : ft of size 2048
in 5G : ft of size 4096

FS: for periodic signals
FT: for non-periodic signals

B.Tech kitne duration ka hai?
≡ fourier

$FT(\delta(t)) \rightleftharpoons 1$
constant → contains all frequencies

FT : time → freq
FT^{-1} = IFT : freq → time

FT
↳ Continuos
↳ Discrete : Sampled / present at only certain time instants
↓
We will work with this since we are using the ADC to get the digital signal

Discrete FT $X(K) = \sum_{n=0}^{N-1} x[n] e^{-jk\omega_0 n} = \sum_{n=0}^{N-1} x[n] e^{-jk\frac{2\pi}{N}n}$

X(K): freq. domain signal
x[n]: discrete time domain signal

Size of F.T. ⇒ N
4G ⇒ 2048
5G ⇒ 4096
Lab ⇒ 8
Lab HW ⇒ 32

if N = 4 ⇒ x[n] has 4 terms (discrete)
0 → 3

F.T. ⇒ matrix vector multiplication
no. of multiplications required : N^2
16 here

n : 0 → N-1
K : 0 → N-1

FFT: fast fourier transform ↓

N ∝ time to perform operation

no. of multiplications required : $N \log_2(N)$
eg: N = 8
⇒ $8 \cdot \log_2(8)$
$8 \times 3 = 24$
N = 32 ⇒ $32 \cdot 5$
⇒ 160

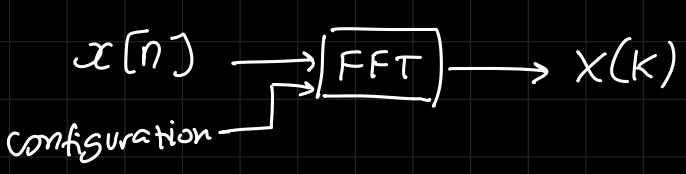
8 point FFT
fast in ARM
slow in FPGA

32 point FFT
fast in FPGA
slow in ARM

i.e. if n: small ⇒ ARM
else ⇒ FPGA

approach: same as lab 6

LTE ≡ 4G
long term evolution



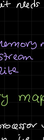
N ≡ transform length

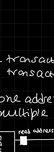
We have to send N data continuously
and similarly receive N data continuously

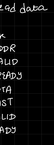
x[n] : complex number (64bit)
↓
real (32) + imaginary (32)
↓
Concatenation

to verify, use matlab :

```
x = (0 0 0 0 0 0 0 1);
fft(x);
```







* Lecture : 16

ARM and FPGA communication ?

↳ AXI Protocol

advanced
extensible
interface

AXI vs native

Some IP provide both
some mordern ones have
only AXI

Zynq ≡ ARM

ARM $\xleftrightarrow{AXI}$ FPGA

in order to have
efficient communication
between multiple blocks
of a SoC, we have
a standard protocol
eg: AXI

AXI is popular
because it is
the protocol which
is used by the famous
ARM processors

AXI : very popular and is implemented in
almost all IPs
hardware
block

Wishbone
protocol

famous in the past

- AMBA : advanced microcontroller bus architecture

AMBA
- ↳ APB : advanced peripheral bus
- ↳ AHB : advanced high performance bus
- ↳ AXI : advanced extensible interface
- ↳ ATB : advanced trace bus

processor accesses memory for
- persistent data storing
- instruction storing

AHB sed for processor ⟷ memory communication because accessing memory is slow

APB used for UART communication
↳ slower than AHB

ATB used by debugger to communicate with processor eg. when breakpoint is hit, it needs to fetch register values etc.

* AXI
  - ↳ memory mapped
  - ↳ stream
  - ↳ lite

memory map

ARM processor treats every other block as memory i.e. communication involves only read and write operation

no need to worry about seperate communication logic, say, between USB, UART

- **AXI :**

- every axi link has two parts
  - ↳ AXI master
  - ↳ AXI slave

- for a given link, there exists only one master and one slave

- master initializes transaction (r/w)

- slave can only respond to the transaction

- Read : slave → master
  write : master → slave

- between a master and the interconnect
  between a slave and the interconnect
  between a master and a slave

- AXI memory mapped → processor will know the start address and the end address and has to do only r/w operations read data →
← read address and control

Slave master

← write address and control
← write data
write response →

total channels: 5

write transaction : 3 channels
read transaction : 2 channels

for one address data, we can read multiple sets of data

⇒ Burst Transfer
↓
imp. when reading large amount of data from contiguous memory

BURST TRANSFER ↑

because in sending read addr and reading everytime, 50% of time is lost in just sending the read addr

Read Transaction

AR ≡ read address and control
R ≡ read data

} 2 channels of Read transaction

	source
ACLK	
ARADDR	master
ARVALID	master
ARREADY	slave
RDATA	slave
RLAST	slave
RVALID	slave
RREADY	master

read transaction is completed only after RLAST ≡ high

In order processing

↓ if we provide 2 read addresses from master: the 1st addr is read from until RLAST hits and 2nd is read from until the next RLAST

ARLAST : last signal for read address and control not needed

WRITE TRANSACTION {Burst write}

AW ≡ Write address and control
W ≡ Write data
B ≡ Write response

} Channels

	source
ACLK	
AWADDR	master
AWVALID	master
AWREADY	slave
WDATA	master
WLAST	master
WVALID	master
WREADY	slave
BRESP	slave
BVALID	slave
BREADY	master

note: clock is a reference signal and has clean edges

LAST signal is present only in read data & write data

* lecture 17

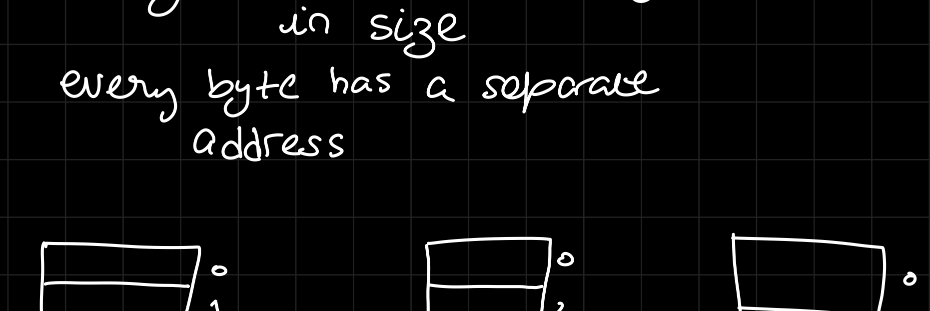
17/10/24

last lecture on AXI
next quiz → AXI

- **Byte addressable memory** ⤵

every location is one byte in size
every byte has a separate address

eg:



- each of the independent channels ⇒ we have a set of information signals
  - eg: AW/AR: address bus
  - W/R : data bus
  - B : response bus

- R/W include the LAST signal

- AXI provides simultaneous, bidirectional data transfer
- Both read and write operations can happen simultaneously

BURST TRANSFER: we do not need to send separate address and control data to get each separate byte.
we can send one address and fetch x amount of data from data.

BURSTS can vary from 1–256 data transfers per burst

- each BURST transfer size can be 1–128 bytes (must be of form 2^n ofc)

- max data per burst → 256 × 128 = 32kb
  (i.e. for one address)

what if i want 16kb : two options ⤵
no. of bursts : 256 → 128
OR each burst size : 128 → 64

how about 1 byte of data?
no. and each size = 1

4 byte? ⇒ 1×4 or 4×1

3 bytes? ⇒ burst : 3
transfer size : 1

258 bytes? ⇒ 129 × 2

259 bytes? ⇒ need NULL transfer

Byte lane strobe signal

for every 8 bits of data, indication which bytes of data are valid.
BLSS size ≡ burst transfer size

so, we have eg → 8 bits BLSS for 8 bytes of data

now for 259 bytes data transfer →

130 data transfers per bursts
2 burst transfer size
with BLSS = 01 for last transfer

for 9 bytes data transfer and with fixed 4 data transfers/burst

⇒ 4 burst transfer size with BLSS:
1111
1111
1000
0000

zeroes mean data : null ↓ not needed

AXI : Read

ACLK
ARID[3:0] identifier for when we have multiple slaves/ masters in interconnect
ARADDR
ARSIZE[2:0] Size of each transfer (1 → 128)
ARVALID
ARREADY

000	1
001	2
010	4
011	8
100	16
101	32
110	64
111	128

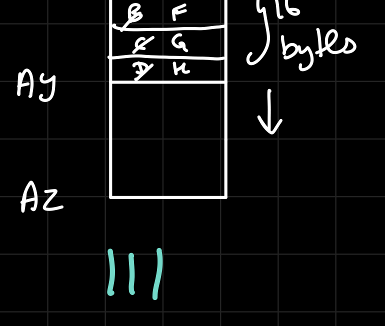
WSTRB [3:0]
↓
eg: F ≡ 111
all 3 bytes are valid

INCREMENT MODE

as we send data → it is written at the data NEXT to the current one

we send Ax as the AW ADDR

now lets say we have 4 locations of 4 bytes size each and we send 32 bytes say ABCDEFGH



then the initial data ABCD is overwritten and EFGH is written since we have limited memory

|||

WRAP MODE (and incremental mode)

coz way down we go

FIFO MODE / FIXED MODE

writing data to some address, say, Ax

≡ same as WRAP mode with size = 1

≡ fixed address burst

BURST TYPE SIGNAL

ARBURST[1:0]/
AWBURST[1:0]

00	FIXED
01	INCR
10	WRAP
11	RESERVED

A_LEN signal [7:0] 0 → 255/ 1→256

size of data transfer in one burst

if AWLEN : 011 → 3 ≡ 3+1 = 4 bytes transferred in one burst

Burst length : A_LEN[7:0] + 1

signals : AWLEN / ARLEN

eg: for 10 bytes of data :
if we have 4 bytes per burst ⇒
we can have any order like ⤵

bits valid (not NULL)	1 (0001)	4 (1111)	4 (1111)	1 (0001)
strobe signal →	1	F	F	1

new signals : ID
SIZE
STROBE
BURST TYPE

AXI memmory mapped

known for burst transfers
↳ efficiency ↑↑
↳ single address, multiple data

- **AXI : lite**
  ↳ single address, single data
  ↳ number of data transferred in one burst = one
  ↳ Bursting not supported
  ↳ subset of AXI : memory mapped
  ↳ useful for configuring hardware IP where only 1/2/3 bits of data required
  ↳ i.e. used for sending smaller sized data

  for LITE → mem mapped in timing diagrams
  ↳ add specific addresses in ARADDR for every transaction

- **AXI : Stream**

  - unidirectional
  - no address
  - unlimited data burst size
  - write data : master → slave

  complexity: STREAM < mem MAPPED

Direct Memory Access

- AXI Data mover
  ↓
  does what it sounds like

DMA : used to switch between STREAM ⇆ mem mapped

- CPU will config DMA to tell which photo will be stored where and hence we use AXI : lite there

so, AXI : Memory mapped for large data
AXI : lite for small data

next week lab : FFT in C on ARM processor