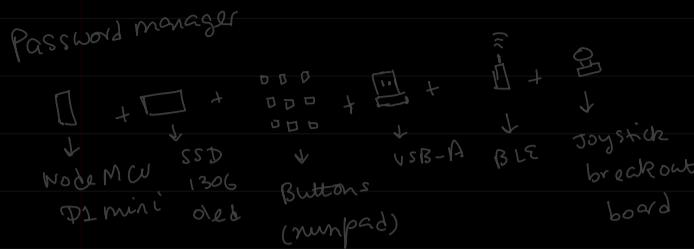


ECE 270 : Embedded Logic Design \Rightarrow Co + DC



Resource for Quiz: hobbits.0x3.net/wiky/Main_Page
Lab videos : youtube

Programming: 1st half \rightarrow Verilog
2nd half \rightarrow Embedded C

Theory: FPGA and SoC

Virado 2019.1 (including SDK)
arch user repository

* GRADES:	mid sem	30 %.
	end sem	30 %.
	Surprise quiz	28 %.
	lab hw	15 %.

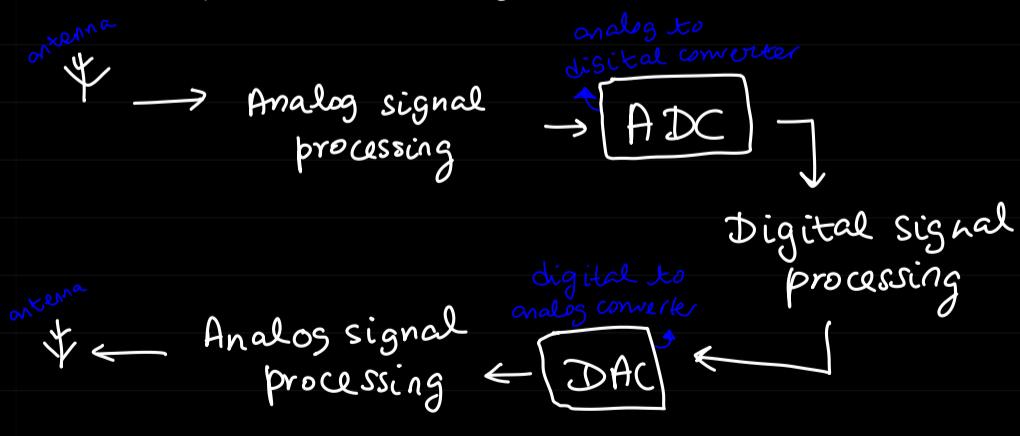
LECTURE: 1

* Which is faster : Analog vs digital ?

\Rightarrow Depends on the use case

* No product is purely digital/analog ?

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



HDL \rightarrow Hardware Description language
 \hookrightarrow eg \Rightarrow Verilog

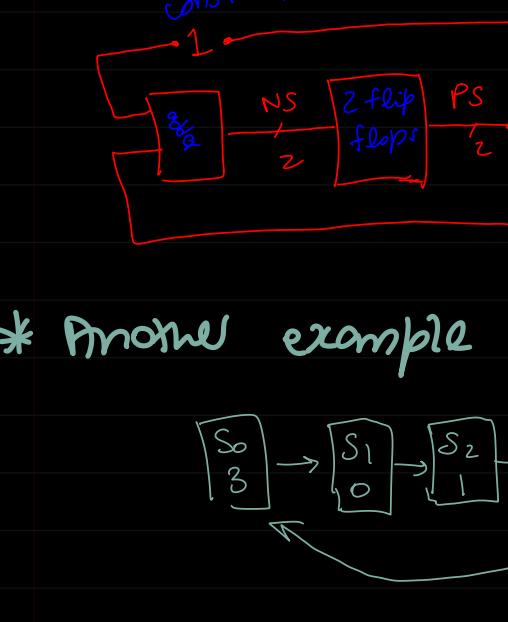
LECTURE : 2

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

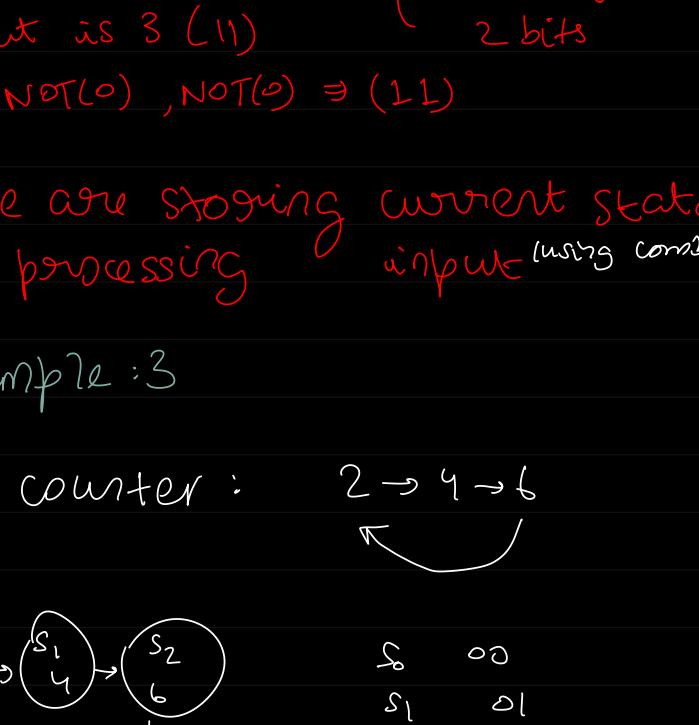
* **sequential circuit**: Output depends upon the current input and the current state of the circuit
what we get → output + next state
because we go from one state to another

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

* **D flip flop**: Input is stored at falling (edge triggered) or rising edge of the clock

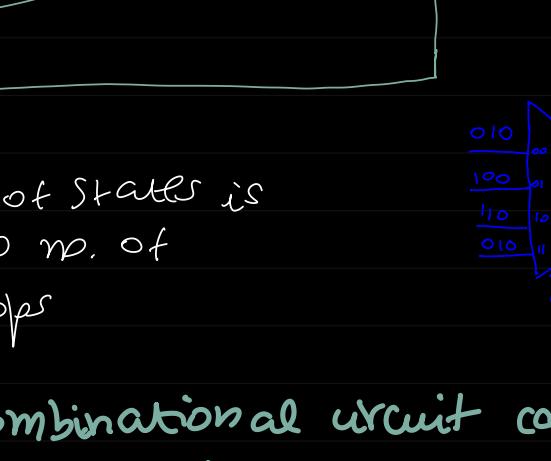


* **Sequential circuit using combinational ckt**



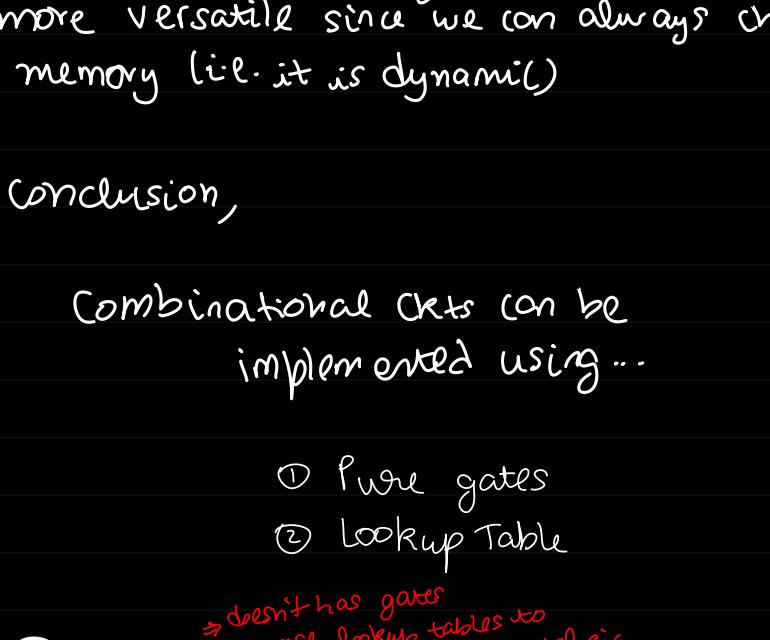
* **FSM (finite state machine)**

⇒ Up Counter

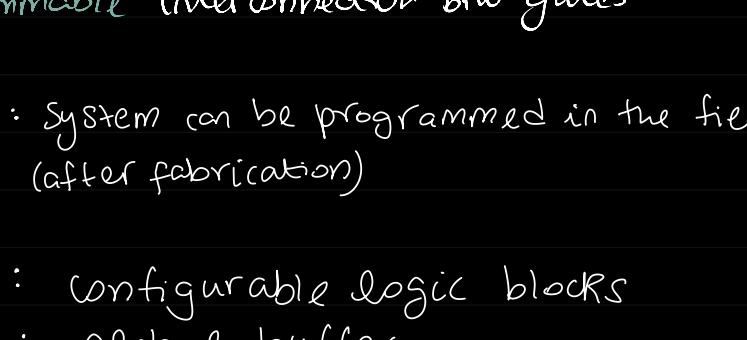


Note: if curr state = S_n , the output is n

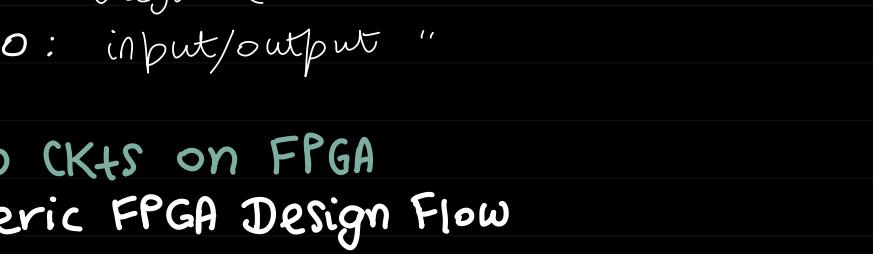
2 bits required to store in mem



* **Another example**



relation b/w present state & next state $\Rightarrow NS = PS + 1$



e.g. if $PS = S_0$ (00) \Rightarrow output is 3 (11)
i.e. $NOT(0), NOT(0) \Rightarrow (11)$

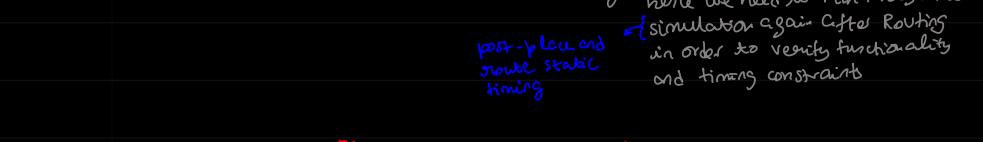
{ Here we should use 2 not gates for the 2 bits

so, we are storing current state + processing inputs (using comb. ckt)

* **Example : 3**

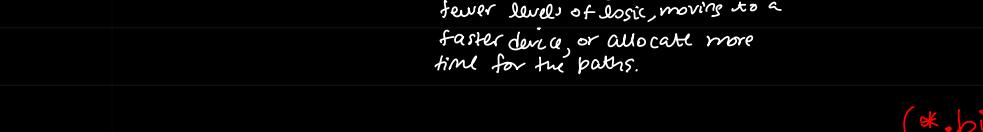
counter : $2 \rightarrow 4 \rightarrow 6$

use either MUX or K-map to find suitable ckt



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

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



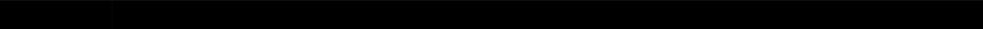
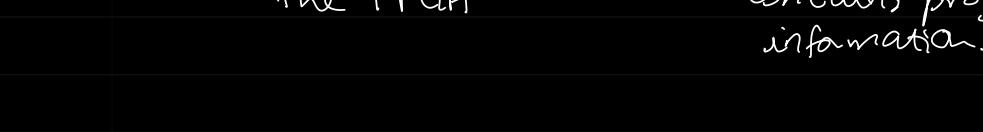
Final Step: generating the bitstream

downloaded directly to the FPGA

OR

converted to a PROM file which contains programming information.

(* .bit)



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

FPGA

The application can be configured and changed even after fabrication.

VHDL code can be converted to .bit and downloaded directly to FPGA

ASIC

fabricate the chip for one specific application we have to send our VHDL code to the fabrication organization

90% devices are ASIC as of this moment

• APPLICATION SPECIFIC IC \Rightarrow 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

* 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 microcontroller designed \equiv GPU for parallel operations

- # 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 \equiv ARM + FPGA + GPU

microcontroller : time limited

FPGA : space limited

* LAB : 1 Design of Full Adder

20/08/29



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

Two major HDLs: Verilog & VHDL

more popular
syntax close to C

easy to master
more prominent in Indian VLSI industry

1983 : introduced by Gateway Design System

Inverted as a SIMULATION language. SYNTHESIS was an afterthought.

1987 : Verilog

Synthesizable by Synopsys

1989 : Gateway DESIGN SYSTEM acquired by Cadence

Latest verilog version = system Verilog

↳ much simpler than initial versions

1981 - 1983 : US Dept of defence developed VHDL (VHSIC HDL)

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

Hardware : parallel processing
Software : sequential processing

In Verilog, all lines execute parallelly unlike languages like C, C++, 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 datatype
variable (Reg, Integer, real, time, realtime)
- Primitive Logic Gates and Switch-Level gates are built in

* EXAMPLES



// module_name <ports>

module AND (out, in1, in2);

input in1, in2;

output wire out;

// in1 and in2 are also

// wire datatype since it is default type

assign out = in1 & in2;

// data flow - continuous assignment

endmodule

in1	in2	out
0	0	0
0	1	0

in1	in2	out
1	0	0
1	1	1

in1	in2	out
1	0	0
1	1	1

in1	in2	out
1	0	1
1	1	0

in1	in2	out
1	0	0
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

* Lecture: 5

8:1 multiplexer

module mux1

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
```

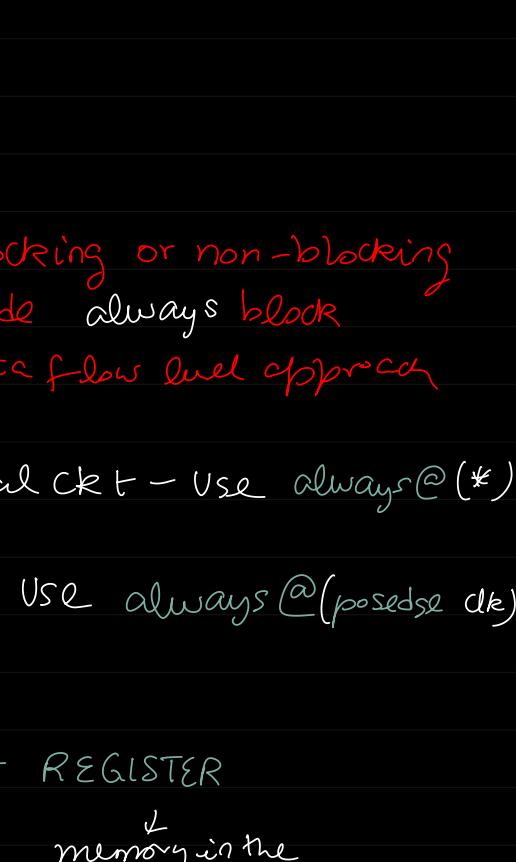
module bar(

input in1,in2;

output out;

endmodule

* Flip flops:



- Reset : setting value to zero

- Pre-set: setting value to one

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

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

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

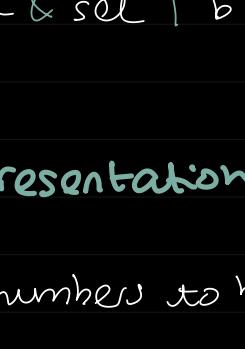
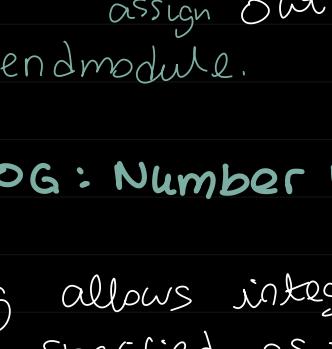
- Asynchronous: operation can happen at any time (irrespective of the clock)

⇒ Flip flops are used to

- Store something in memory

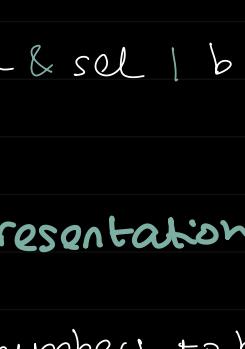
- pass a digital signal synchronously

to make reset synchronous



* Verilog code for D flip flop

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



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 ck + - 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

e.g.: not in Mux code

but yes in register code and flip flop code

* MODULE PORTS : provide interface for the module

≡ interface to communicate with its environment

input output inout

environment

- 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

F49 fa-byname (.cout(COUT), .sum(SUM),

.b(B), .a(A).cin(Cin));

⇒ Order Association

F49 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 ≡ 32'd.....0000

12'hX ≡ 12-bit unknown number

* Negative Number

- [number]

Signed

default : positive number unsigned

in hardware : 2' complement

e.g.: -4'b11

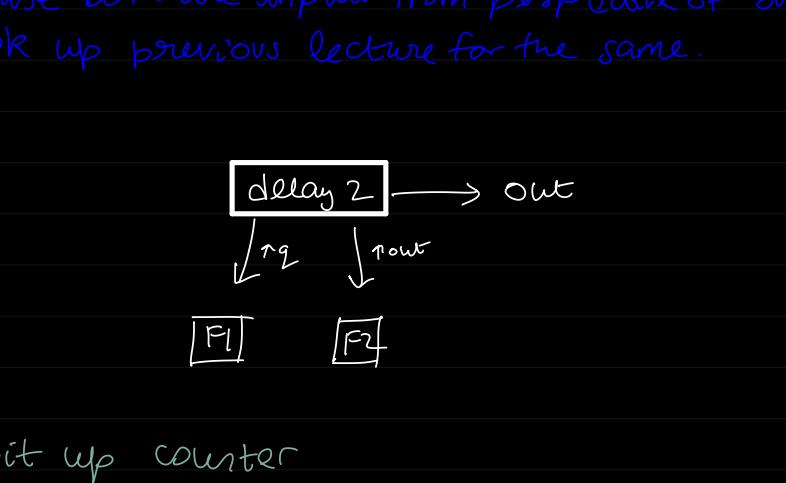
* LAB: ELD

27/08/24 8:30AM

- For counter to increment every second, Design 8 bit up counter ($0 \rightarrow 255$) using behavioural 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
```

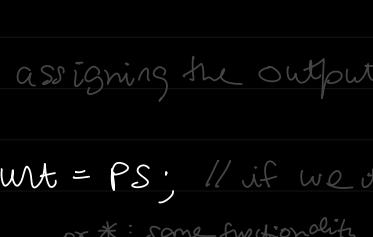
*Note: Q and out are reg inside each D-FF block but will be wire in this module because both are inputs from perspective of our module. Look up previous lecture for the same.

$\boxed{\text{delay 2}} \longrightarrow \text{out}$

$\downarrow \text{reg} \quad \downarrow \text{reg}$

$\boxed{F1} \quad \boxed{F2}$

- 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

$$NS = PS + 1$$

```
module counter(
    input CLK, reset,
    output [7:0] count
);

```

```
// flip flop
always @ (posedge CLK)
```

```
begin
```

```
if (reset)
```

```
PS <= 8'b00000000
```

```
// same as PS <= 8'd0
```

```
else
```

```
PS <= NS
```

```
end
```

```
// finally, assigning the output
```

```
assign count = PS; // if we take count as wire
```

or *: some 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 \Rightarrow D flip flop

- Testbench:

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

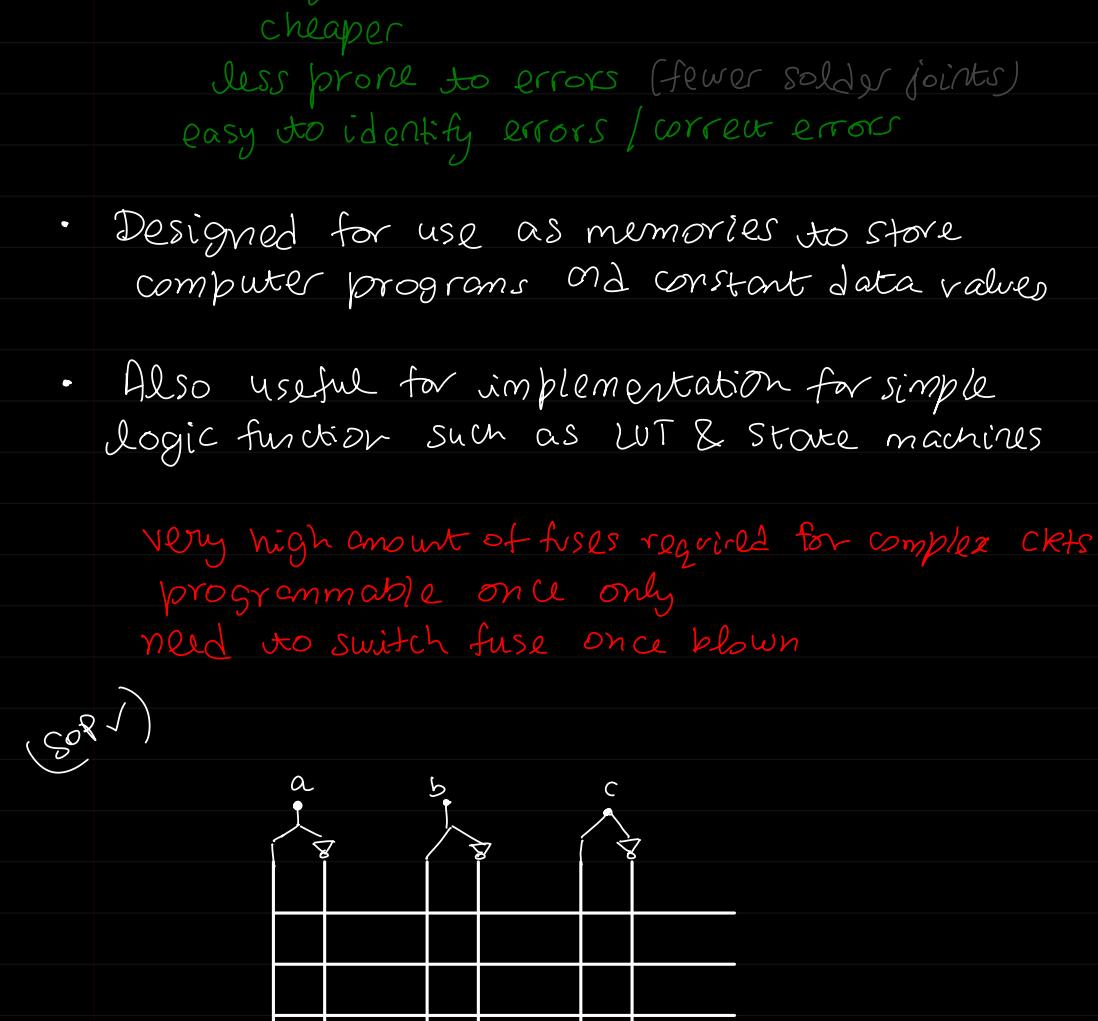
* LECTURE : 6 (Architecture)

27/08/24

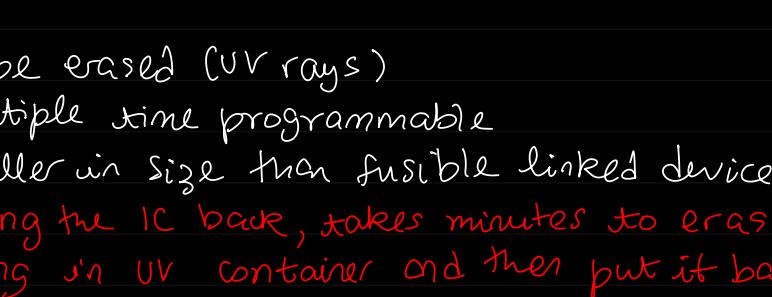
3 - 4:30pm

- 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



* PROM: programmable read-only memory (1970)



- 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

* 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 SOF 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

high power consumption

Programmable Array Logic

→ 1984: Altera introduced CPLD using CMOS (high density, low power) and EPROM / EEPROM (programmability)

E²PROM

= multiple SPLDs

→ Added multiplexers to each SPLD so that only the necessary stuff is processed

= to combat higher power consumption

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

(communicate)

Signals from a logic block can travel through adjacent blocks only

ALTERA

Programmable interconnect matrix

input / output pins

SPLD like blocks

<p

• LUT as memory & LUT as ALU

↳ lookup table

→ Operations on memory: read or write

we provide the address of the

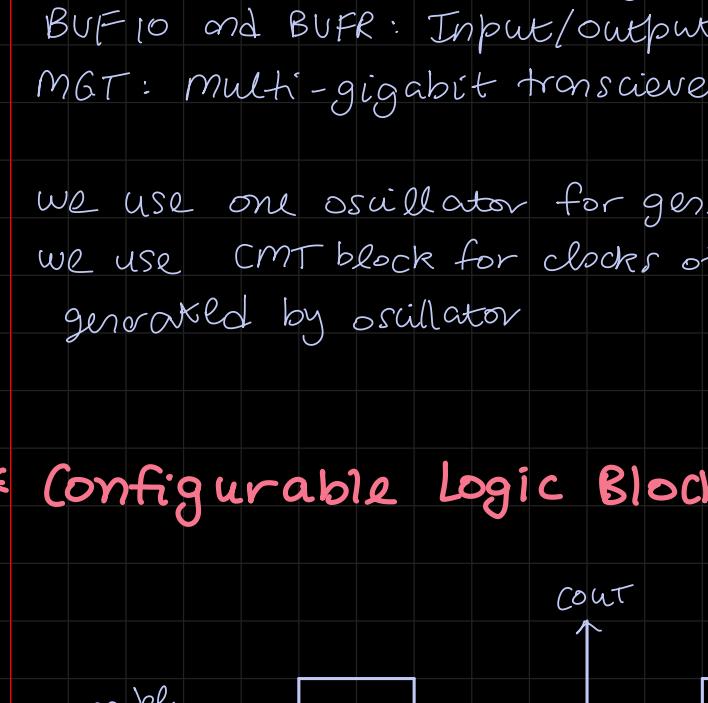
memory → 9 bytes

here

you provide the data and the

1 byte ← address

9 bytes ←



* 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

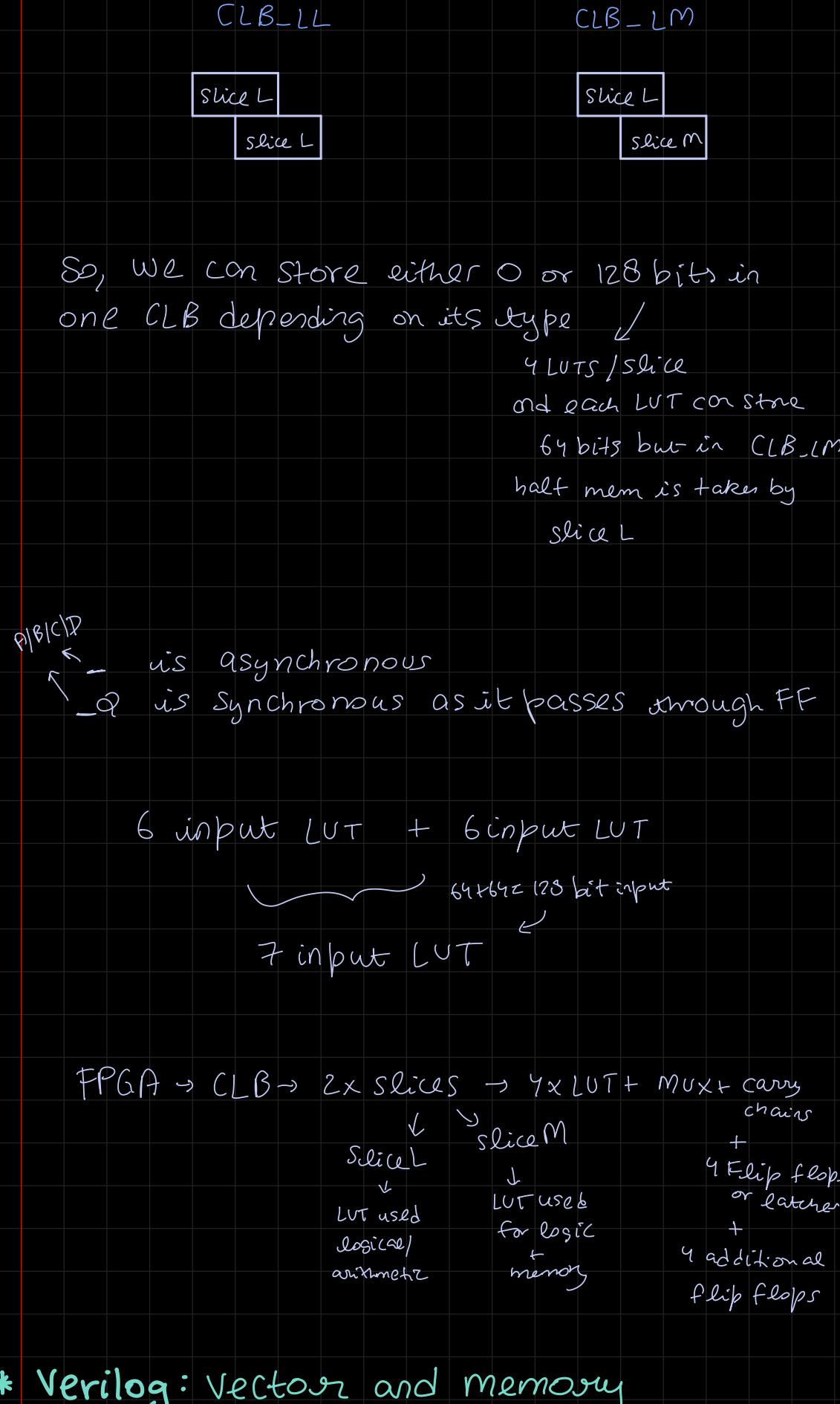
DSP: digital signal processing

BUFIO and BUFR: Input/output & Regional buffer

MGT: multi-gigabit transceiver

- 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]



Our flip flop now ↗

Slices	LUT	Flip Flops	Arithmetic & carry chains
2	8	16	2

(4 LUT/slice) (8 FF/slice)
= 6 input LUT

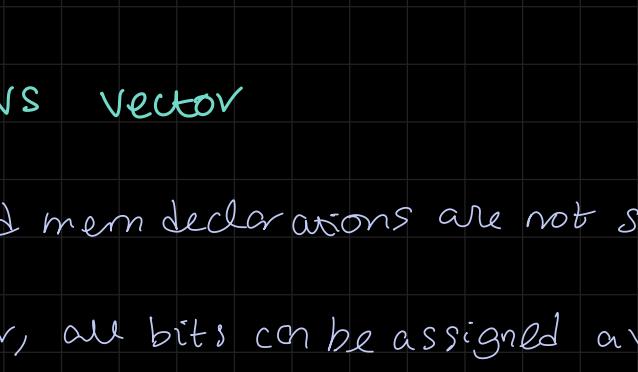
total: 64 bit of data stored in LUTs
8 bits stored in each LUT (6 bits input to LUT)

= 512 bit of data in one CLB X see below

• CLB : SLICES

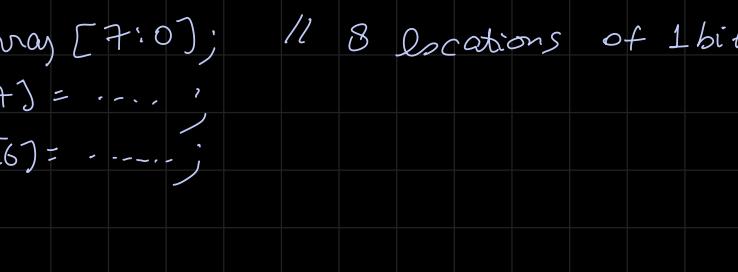
① SLICEM: Full slice } read and write only
↳ combinational circuit

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



② SLICEL: logic and arithmetic only } read only

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



• 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] = ...;

• Memory is slower than vector because it needs to access multiple locations sequentially

• Vector is faster than memory because it can access all bits simultaneously

• Memory is more flexible than vector because it can store different data types (e.g., integers, floating-point numbers, strings) in different locations

• Vector is more efficient than memory for certain operations (e.g., SIMD operations) because it can perform parallel calculations on multiple bits simultaneously

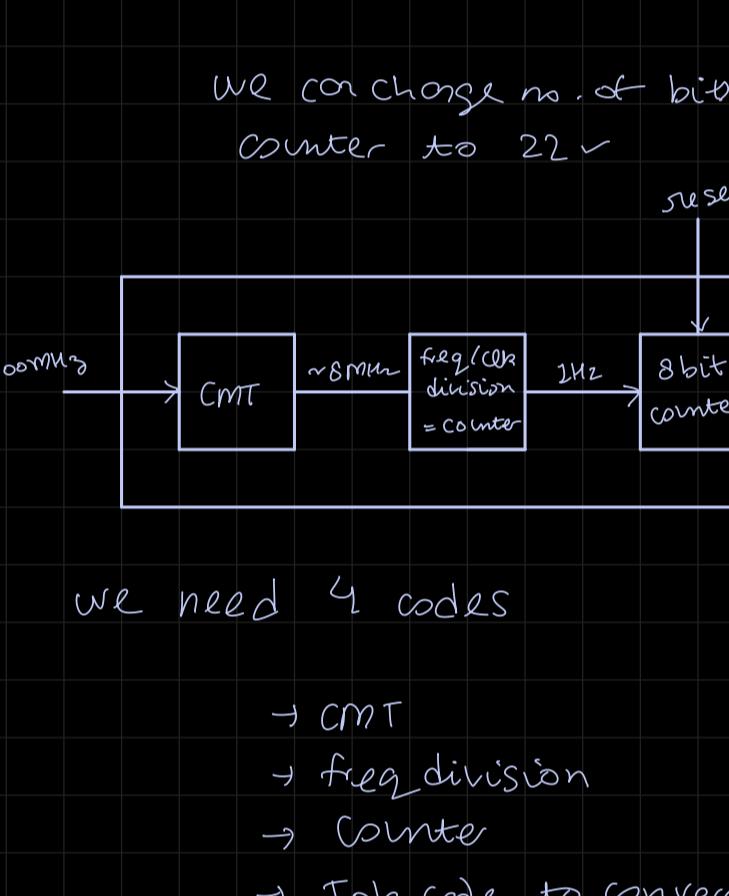
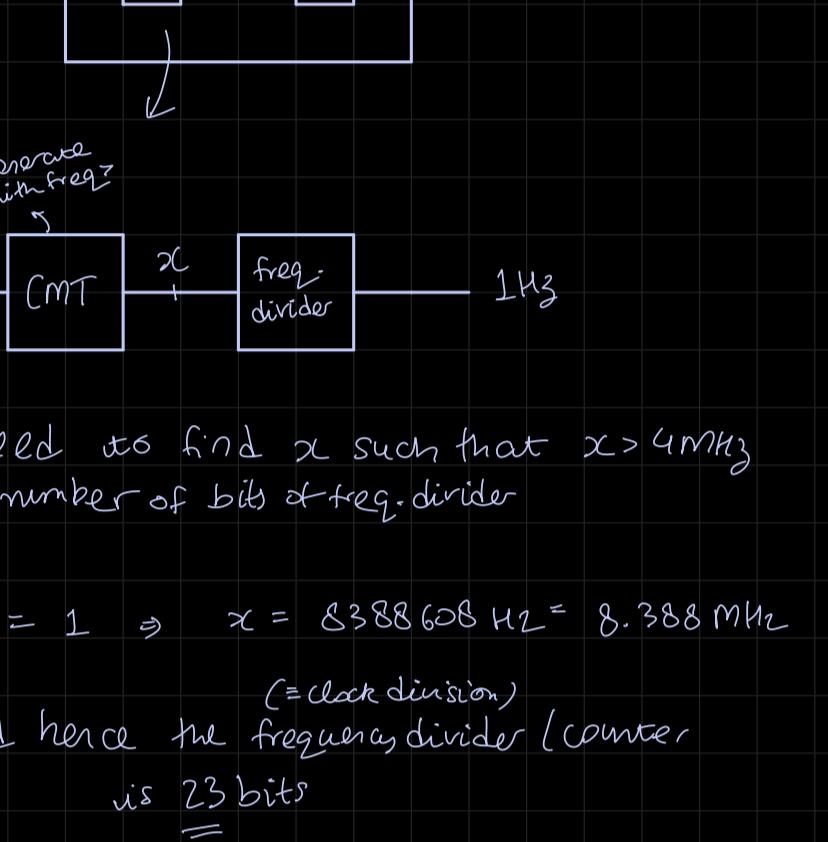
* LAB:3 (Running on hardware)

03/09/24

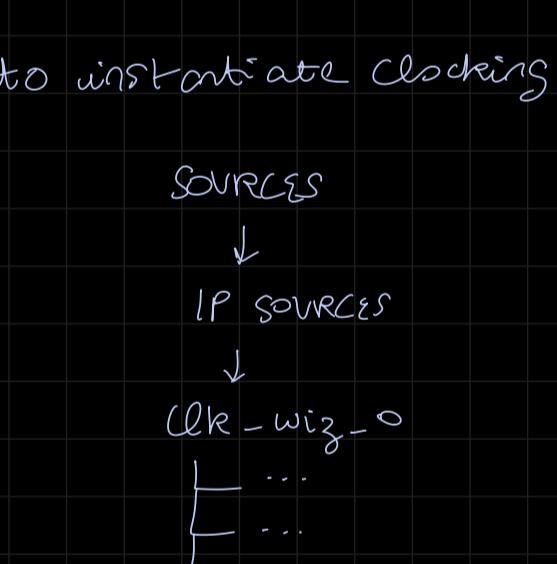
⇒ Let's say our program is an 8-bit upcounter
the program will run on the hardware
but how will you observe the output?

- 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.

Let's take example of 3 bit up counter



here, duty cycle is 50%



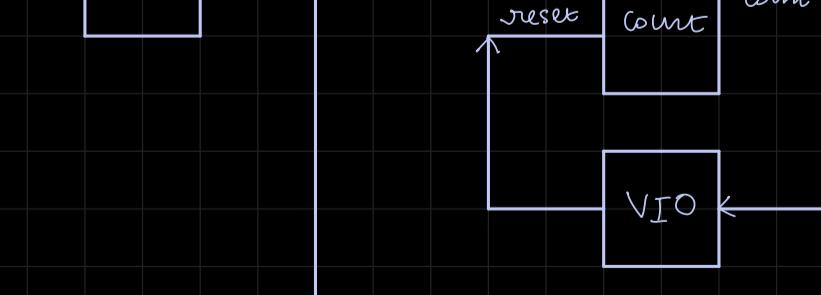
We need to find χ such that $\chi > 4 \text{MHz}$ and number of bits of freq. divider

$$\text{here } \frac{\chi}{2^3} = 1 \Rightarrow \chi = 8388608 \text{ Hz} = 8.388 \text{ MHz}$$

(= 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 → output options
clocks (clk-100m) $\equiv 100\text{MHz}$
 $\equiv 8.388\text{MHz}$

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.v

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

FPGA

Virtual Input Output = VIO

= debugger

= exact replica of test bench
but now it is running on hardware

* Lecture : 8

03/09/21

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 8 bit size each

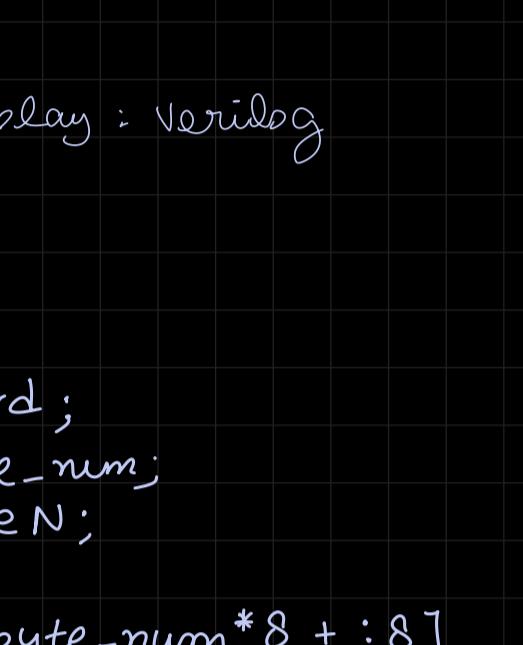
`integer matrix[4:0][0:31];`

↳ 2dimensional memory

`wire [1:0] regL [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]`

$$= 4 \times 8 + : 8$$

= 32 + : 8 (forward direction)

$$= [39:32]$$

$$\text{word}[7+16] = [22:7]$$

$$a[31-:8] = a[31:24]$$

8 bits 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/Port Select $A[0] = 1'b1$

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

{x}{y} Replication $\{3\{A[7:6]\}\}$

$= 6'b101010$

<< shift left logical $\times 2^x$

>> shift right logical $\div 2^x$

shifting bits is very cheap (= signal rerouting)

used to perform multiplication and division powers of 2.

eg: $6(=4'b0110) \xrightarrow{\ll} 4'b1100 (=8+4=12)$

$\xrightarrow{\gg} 4'b0011 (=2+1=3)$

works perfectly only for unsigned numbers

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

OR just use shift right arithmetic
works for signed 2's complement

no such problems when shifting towards msb (<<)

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

↳ byte swap

eg: $a = 8'hAB = 16'h00AB$

$b = 16'hAB00$

* OPERATORS

[] Bit/Port Select $A[0] = 1'b1$

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

{x}{y} Replication $\{3\{A[7:6]\}\}$

$= 6'b101010$

<< shift left logical $\times 2^x$

>> shift right logical $\div 2^x$

shifting bits is very cheap (= signal rerouting)

used to perform multiplication and division powers of 2.

eg: $6(=4'b0110) \xrightarrow{\ll} 4'b1100 (=8+4=12)$

$\xrightarrow{\gg} 4'b0011 (=2+1=3)$

works perfectly only for unsigned numbers

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

OR just use shift right arithmetic
works for signed 2's complement

no such problems when shifting towards msb (<<)

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

↳ byte swap

eg: $a = 8'hAB = 16'h00AB$

$b = 16'hAB00$

- Note:

left orith.
<<<)

- For multiplication, FPGAs have DSP48 dedicated to fast math.

However, if there are no free DSPs, used which is large and slow.

- For multiplication of two N bit numbers, result is max 2^N bits number
 - make sure to define your variables and their size explicitly.

Bitwise operators:
 operates on &
 each bit individually

\sim	unverse	Output
$\&$	And	can be
$ $	Or	multi-bit
\wedge	not	
$\sim\sim$	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: $\max\{\text{len}(A), \text{len}(B)\}$
 - by default everything is unsigned
 - this is how we can tell the tool that we want signed operation:
 $\text{assign out} = (\$signed(a)) < (\$signed(b))$
 or we can store the value in signed way using $\$signed$ and do operation normally,
 logical operators:

$!$	NOT	Output is
$\&\&$	AND	one bit only
$ $	OR	$0, 1 \leftarrow$
$==$	EQUAL	OR TRUE/FALSE
$!=$	NOT EQUAL	
$<, >, \leq, \geq$	COMPARISON	

a	b	$a \& b$	$a \oplus b$	$a \& \& b$	$a \oplus \oplus b$
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

- 2 basic blocks: always \rightarrow and initial \rightarrow
in behavioural modelling
 - both run in parallel
(will not block the execution of other blocks)

- INITIAL BLOCK
 - starts at #0 and executes only once.
 - we cannot use "always" inside "initial" & vice versa

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

$b = \#50$ $c \& d$

$\#50$ $b = c8d$ } calculated & assigned
at $x = 50$

*calculated at $x = x$
but assigned at $x = x$*

of clock cycles instead of seconds.
Hence, we use flip flops to create delay
on hardware

- ALWAYS BLOCK
 - starts at $t=0$
 - executes statements continuously in a loop
 - statements inside always block are executed

- describes the functionality of the circuit
- used for clock declaration

not Synthesizable on hardware	Synthesizable on hardware	Synthesizable on hardware
X	✓	✓

⇒ ERROR: Multi driver error
Some variable driving two blocks
Synthesis error

eg: always @ (posedge clk)

begin

if (rst-n)

$Q \leftarrow D$;

end

always @ (negedge rst-n)

begin

if (!rst-n)

$Q \leftarrow 1'b0$;

end

- Q is being updated by two blocks simultaneously

Parameters

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

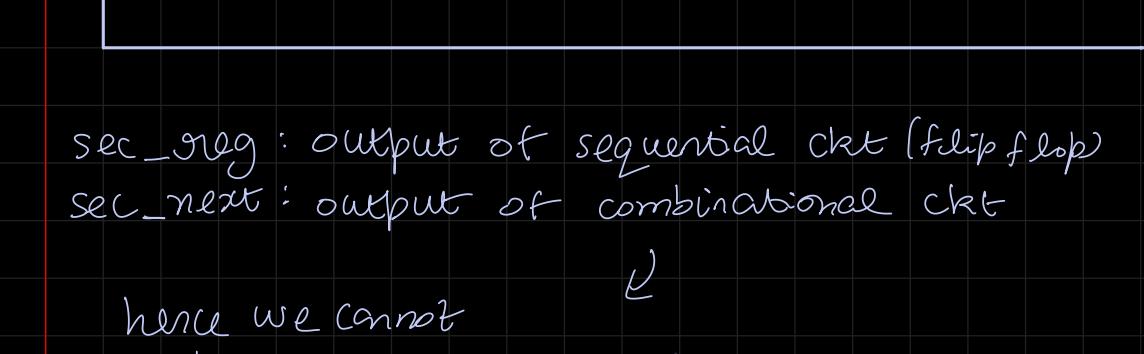
* Digital clock (minute : seconds)
(behavioural modelling)

$0 \rightarrow 59 \downarrow \downarrow 0 \rightarrow 59$

6 bit up counter

always @ \downarrow

HW: modify the digital clk so that the output of CMT block is 16.777 MHz
clock management time \downarrow
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 initialise 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

• Behavioral modelling for comb. ckt

- (1) multi-driver error
- (2) Incomplete sensitivity list
- (3) Incomplete branch and incomplete o/p assignments

SOLUTIONS (\equiv Guidelines)

① → have separate always block for each identifier

→ do not update multiple identifiers in one ALWAYS block

② designing AND gate wrong. comb. ckt's
always @ (in1, in2) begin
out = in1 & in2; end

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.

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

③ always @ * → intention: combinational circuit
if (a>b)
gt = 1'b1
else if (a=b)
eq = 1'b1

* Problem 1: 2 outputs of one always block
* Problem 2: for each condition, only one variable of the two variables are getting updated and hence the other variable is stored in memory which we don't want because sequential

→ 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

another example:

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

case (s)
2'b00 : y = 1'b1; } not considering
2'b10 : y = 1'b0; } case where
2'b11 : y = 1'b1; } s = 2'b01
endcase

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: case (sel)
2'b1? : out <= a; } parallel x
2'b?1 : out <= b; } because of ambiguity when sel
default: out <= c; }
note: for sel = 2'b11 \Rightarrow out = a is 2'b11
because of higher priority

summary
• If an always block executes and a variable is not assigned

→ variable has to be stored
↳ not combinational ckt
↳ unnecessary 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

eg¹) always @ (posedge clk)
begin

reg1 <= #1 in1;
reg2 <= @ (negedge clk) in2 ^ in3;
reg3 <= reg2;

end

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

hence reg3 will have the previous value of reg2 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

eg²) 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_{old}
b = a_{old}

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

eg³) always @ (posedge clk)
begin

q1 = in;
q2 = q1;
out = q2;

non-blocking
end

in FF q1, q2, out

always @ (posedge clk)
begin

q1 <= in;
q2 <= q1;
out <= q2;

end

in FF q1, q2, out

3 clk cycles delay ✓

for sequential ckt's use non-blocking assignment only

but for comb. ckt's use blocking assignment only

when doing both sequential & comb. we do non blocking

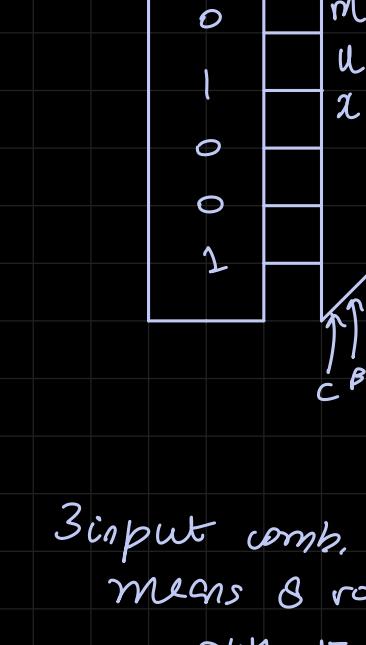
• FPGA Architecture

2 LUT outputs: O_5, O_6

overall outputs: $-, -MUX, -Q$
 eg: D, D_{MUX}, D_Q

through multiplexer
 passed through flip flop
 (synchronous)
 and delayed by 1 clock cycle

if we combine 2 6bit LUTs: we get a
 64 locations \leftarrow 7bit LUT
 of 1bit size each \hookrightarrow 128 locations

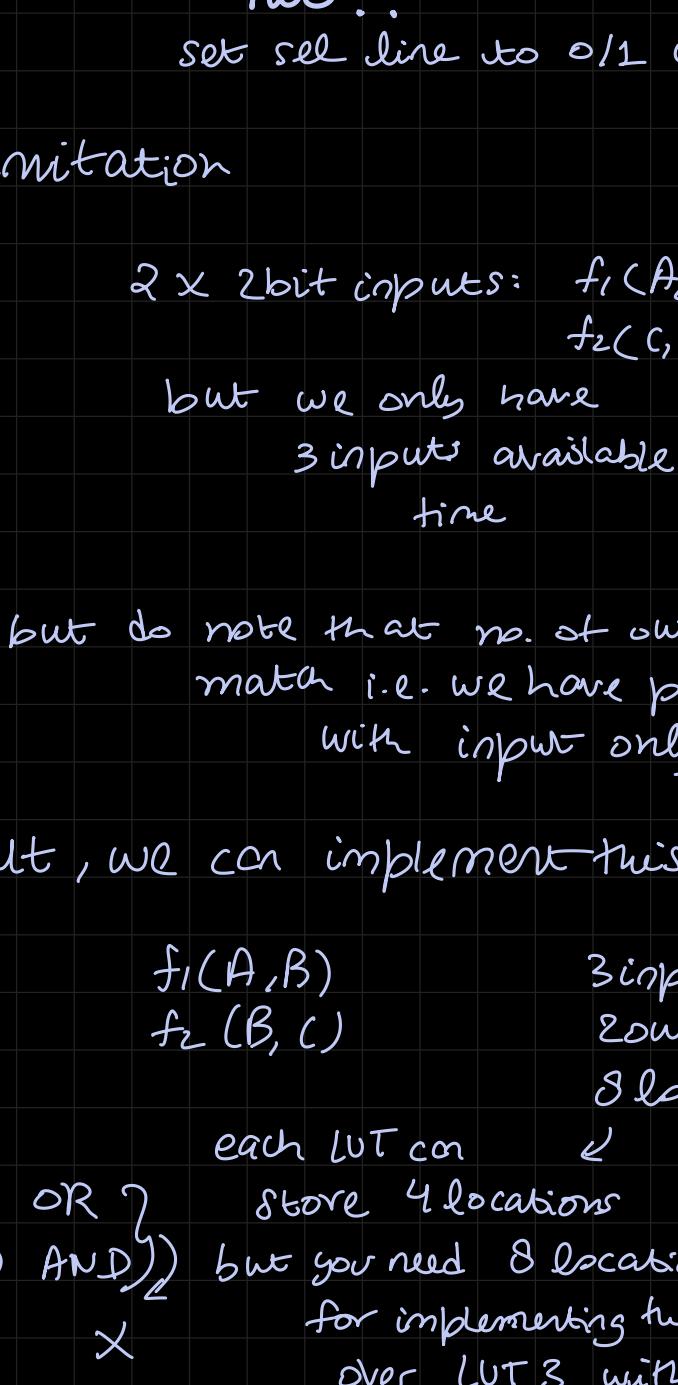


Combining two 2bit LUTs to get a 3bit LUTs

$$\text{eg: } I = 100 \Rightarrow \text{out} = E \\ I = 001 \Rightarrow \text{out} = F$$

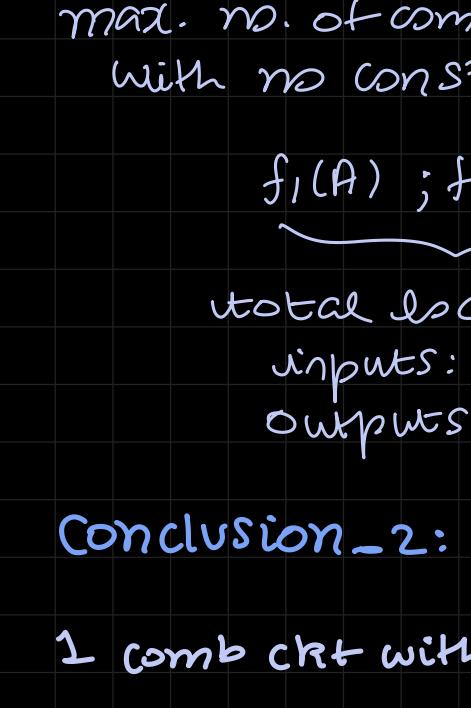
Similarly, our S-series FPGA, can combine all 4 6bit LUTs to get max one 8bit LUT

* 3input LUT



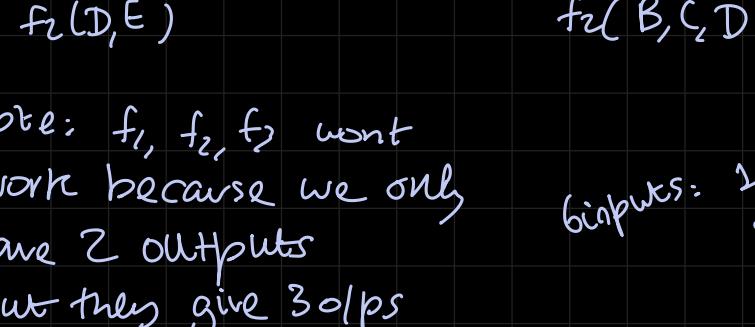
3input comb. ckt means 8 rows & 1 output cal in truth table

so, only 2 comb ckt can be implemented in one 3input LUT



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

• LUT 3 architecture



how many comb ckt for 3bit input? ONE

how many comb ckt for 2bit input? TWO?

set sel line to 0/1 const

limitation

$$2 \times 2\text{bit inputs: } f_1(A, B) = y_1 \\ f_2(C, D) = y_2$$

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:

$$f_1(A, B, C) \\ f_2(D, E, F)$$

each LUT can store 4 locations

but you need 8 locations in each LUT for implementing two comb ckt over LUT 3 with 2bit input

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

Conclusion: we can implement 2 comb ckt with 2bit input in LUT 3 only if both the inputs are same

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

max. no. of comb ckt 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

because we have only 3 inputs (rest 1 is sel)

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

Conclusion-2:

1 comb ckt with 3 inputs ✓

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

2 comb ckt with 1 input ✓

* Now what about LUT 6?

inputs	comb. ckt	constraint
$f_1(A, B, C, D, E)$	6	1
$f_2(F, G, H, I, J)$	5	2
382 (over less)	2	2

eg: $f_1(A, B, C)$

$f_2(D, E)$

eg: $f_1(A, B, C)$

$f_2(B, C, D)$

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

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

but, we can implement this:

$$f_1(A, B, C) \\ f_2(D, E, F)$$

each LUT can store 4 locations

but you need 8 locations in each LUT for implementing two comb ckt over LUT 3 with 2bit input

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

Conclusion-2:

1 comb ckt with 3 inputs ✓

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

2 comb ckt with 1 input ✓

* Finite State machine

① MOORE machine

② MEALY machine

