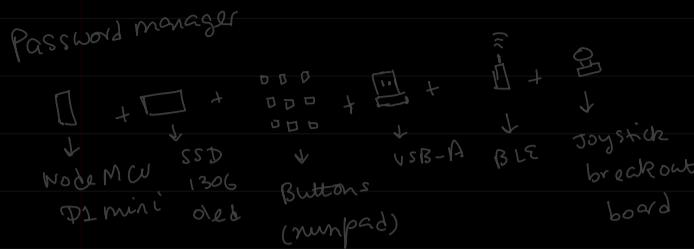


ECE 270 : Embedded Logic Design \Rightarrow Co + DC



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 %.

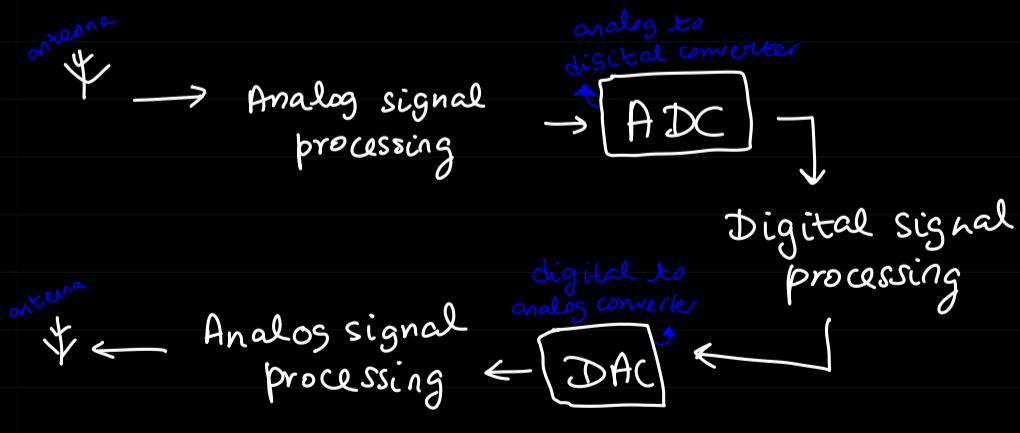
LECTURE: 1

* 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

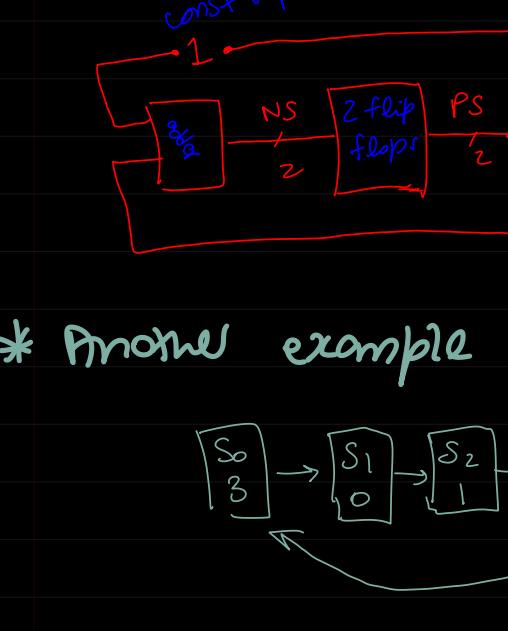
LECTURE : 2

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

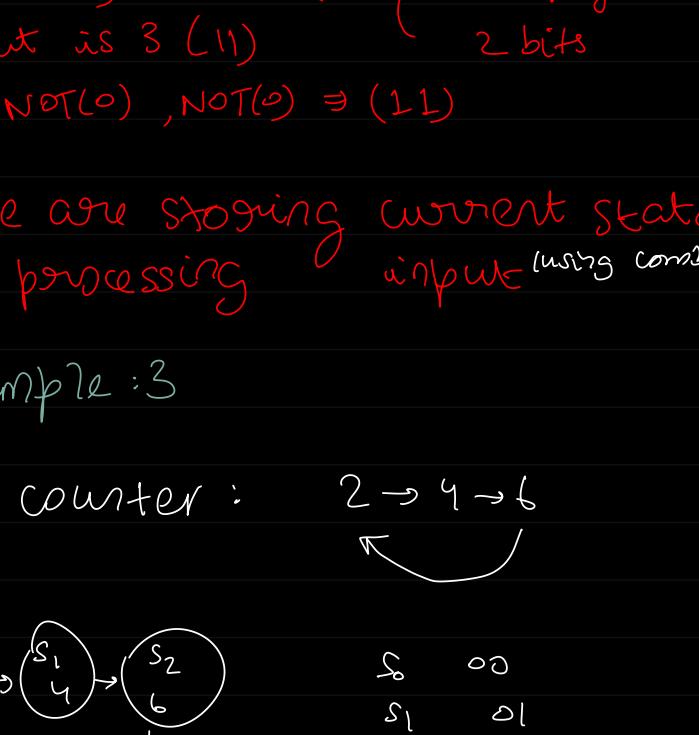
* **sequential circuit**: Output depends upon the current input and the current state of the circuit
what we get → output + next state
because we go from one state to another

⇒ 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

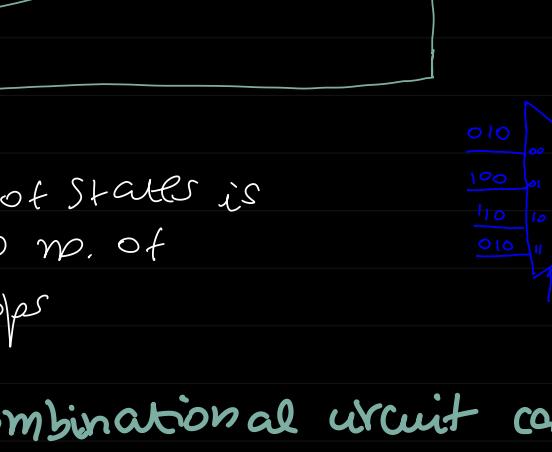


* **Sequential circuit using combinational ckt**



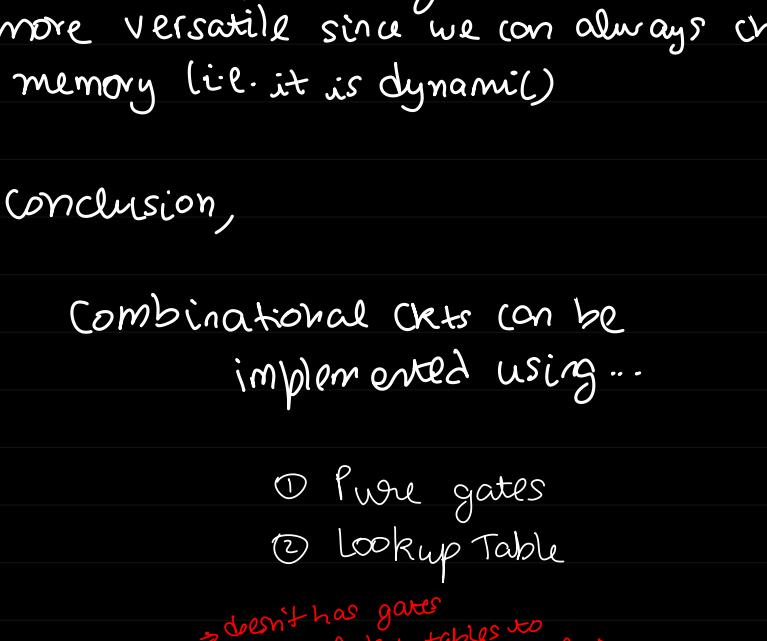
* **FSM (finite state machine)**

⇒ Up Counter

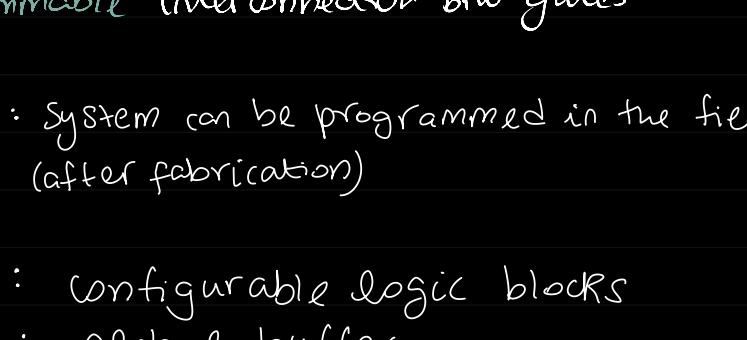


Note: if curr state = S_n , the output is n

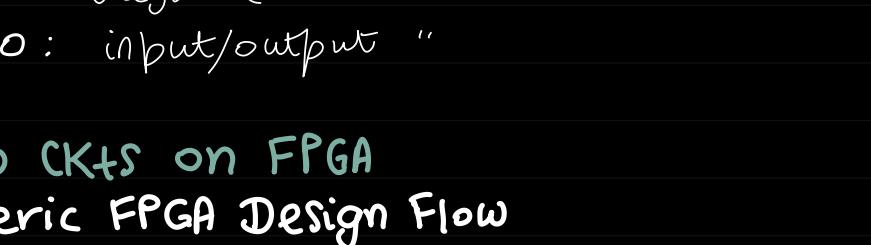
2 bits required to store in mem



* **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 dkt)

* **Example : 3**

counter : $2 \rightarrow 4 \rightarrow 6$

input: S_0 (0)

output: S_1 (1)

input: S_1 (1)

output: S_2 (0)

input: S_2 (0)

output: S_3 (1)

input: S_3 (1)

output: S_4 (0)

input: S_4 (0)

output: S_5 (1)

input: S_5 (1)

output: S_6 (0)

input: S_6 (0)

output: S_7 (1)

input: S_7 (1)

output: S_8 (0)

input: S_8 (0)

output: S_9 (1)

input: S_9 (1)

output: S_{10} (0)

input: S_{10} (0)

output: S_{11} (1)

input: S_{11} (1)

output: S_{12} (0)

input: S_{12} (0)

output: S_{13} (1)

input: S_{13} (1)

output: S_{14} (0)

input: S_{14} (0)

output: S_{15} (1)

input: S_{15} (1)

output: S_{16} (0)

input: S_{16} (0)

output: S_{17} (1)

input: S_{17} (1)

output: S_{18} (0)

input: S_{18} (0)

output: S_{19} (1)

input: S_{19} (1)

output: S_{20} (0)

input: S_{20} (0)

output: S_{21} (1)

input: S_{21} (1)

output: S_{22} (0)

input: S_{22} (0)

output: S_{23} (1)

input: S_{23} (1)

output: S_{24} (0)

input: S_{24} (0)

output: S_{25} (1)

input: S_{25} (1)

output: S_{26} (0)

input: S_{26} (0)

output: S_{27} (1)

input: S_{27} (1)

output: S_{28} (0)

input: S_{28} (0)

output: S_{29} (1)

input: S_{29} (1)

output: S_{30} (0)

input: S_{30} (0)

output: S_{31} (1)

input: S_{31} (1)

output: S_{32} (0)

input: S_{32} (0)

output: S_{33} (1)

input: S_{33} (1)

output: S_{34} (0)

input: S_{34} (0)

output: S_{35} (1)

input: S_{35} (1)

output: S_{36} (0)

input: S_{36} (0)

output: S_{37} (1)

input: S_{37} (1)

output: S_{38} (0)

input: S_{38} (0)

output: S_{39} (1)

input: S_{39} (1)

output: S_{40} (0)

input: S_{40} (0)

output: S_{41} (1)

input: S_{41} (1)

output: S_{42} (0)

input: S_{42} (0)

output: S_{43} (1)

input: S_{43} (1)

output: S_{44} (0)

input: S_{44} (0)

output: S_{45} (1)

input: S_{45} (1)

output: S_{46} (0)

input: S_{46} (0)

output: S_{47} (1)

input: S_{47} (1)

output: S_{48} (0)

input: S_{48} (0)

output: S_{49} (1)

input: S_{49} (1)

output: S_{50} (0)

input: S_{50} (0)

output: S_{51} (1)

input: S_{51} (1)

output: S_{52} (0)

input: S_{52} (0)

output: S_{53} (1)

input: S_{53} (1)

output: S_{54} (0)

input: S_{54} (0)

output: S_{55} (1)

input: S_{55} (1)

output: S_{56} (0)

input: S_{56} (0)

output: S_{57} (1)

input: S_{57} (1)

output: S_{58} (0)

input: S_{58} (0)

output: S_{59} (1)

input: S_{59} (1)

output: S_{60} (0)

input: S_{60} (0)

output: S_{61} (1)

input: S_{61} (1)

output: S_{62} (0)

input: S_{62} (0)

output: S_{63} (1)

input: S_{63} (1)

output: S_{64} (0)

input: S_{64} (0)

output: S_{65} (1)

input: S_{65} (1)

output: S_{66} (0)

input: S_{66} (0)

output: $S_{67}</$

- **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

• Field Programmable Gates Array: FPGA

- # lowest cost for low to medium volume
- # non recurring engineering
- # No NRE 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 $\xrightarrow{\text{then}}$ 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 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

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



in1	in2	out
0	0	0
0	1	0

// module_name <ports>
module AND (out, in1, in2);

input in1, in2;

// in1 and in2 are also

// wire datatype since it is default type

assign out = in1 & in2;

// data flow - continuous assignment

endmodule

⇒ Half Adder

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

C _i	X _i	Y _i	C _{i+1}	S _i
0	0	0	0	0
0	1	0	0	1
1	0	0	1	0
1	0	1	0	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
```

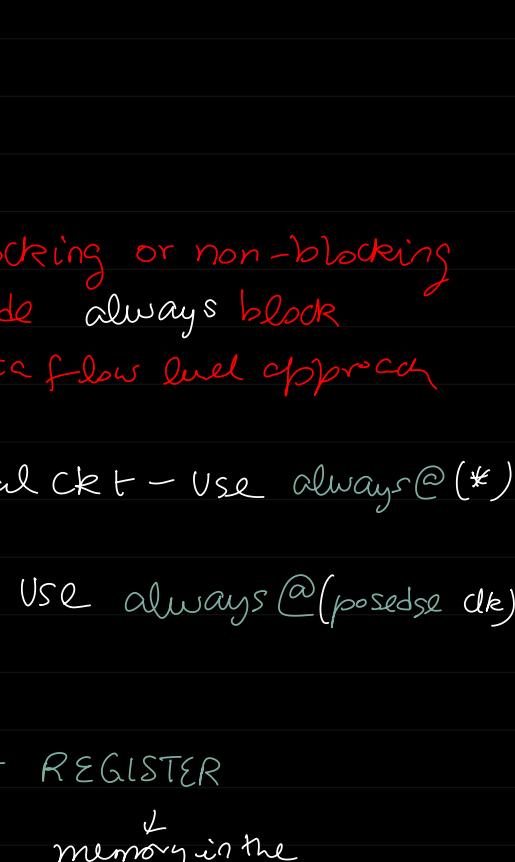
module bar(

input in1,in2;

output out;

endmodule

* Flip flops:



- Reset : setting value to zero

- Presest: 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

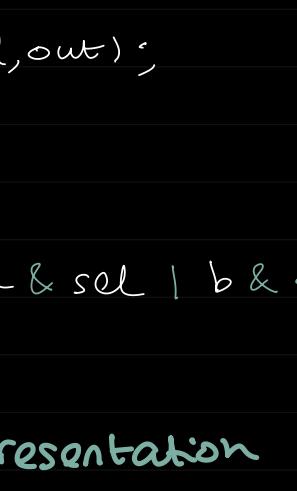
to make reset synchronous



\downarrow
memory in the hardware, not the datatype here

* 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

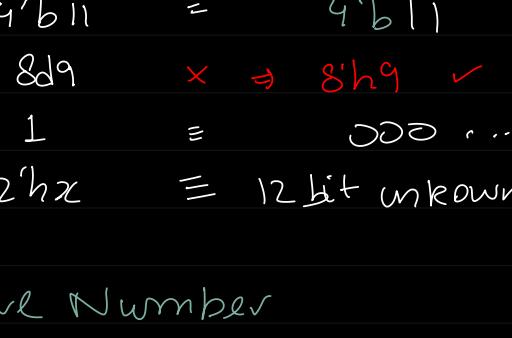
endmodule

* 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



* 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;
endmodule
```

assign out = a & sel | b & ~sel;

endmodule.

* VEROLOG: 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

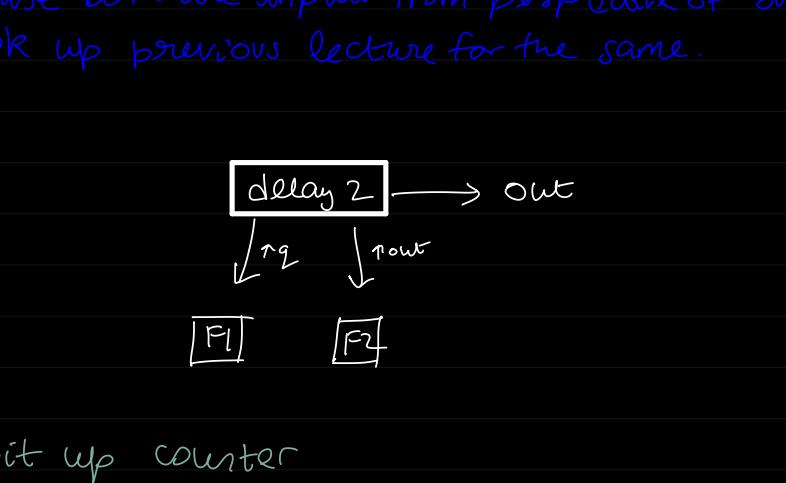
* LAB: ELD

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

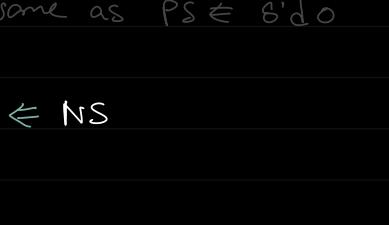


```
module delay_2(Din, CLK, out);
    input Din, CLK;
    output reg* out;

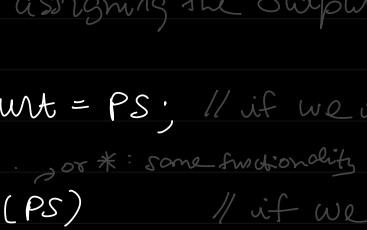
```

```
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.



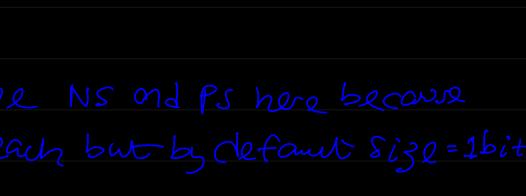
- 8 bit up counter
 - (1) block diagram
 - (2) define all signals
 - (3) write code



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
);
    reg [7:0] PS;
    reg [7:0] NS;
    // flipflop
    always @ (posedge CLK)
        begin
            if (reset)
                PS <= 8'b00000000;
            else
                PS <= NS;
        end
    // finally, assigning the output
    assign count = PS; // if we take count as wire
    // or *: some functionality
    always @ (PS)
        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.

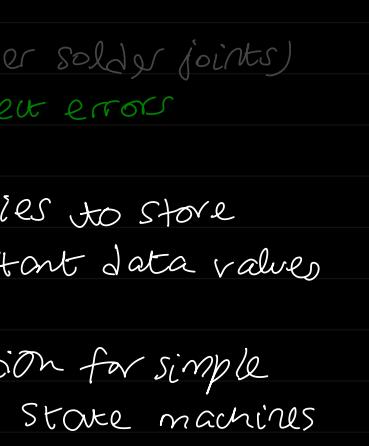
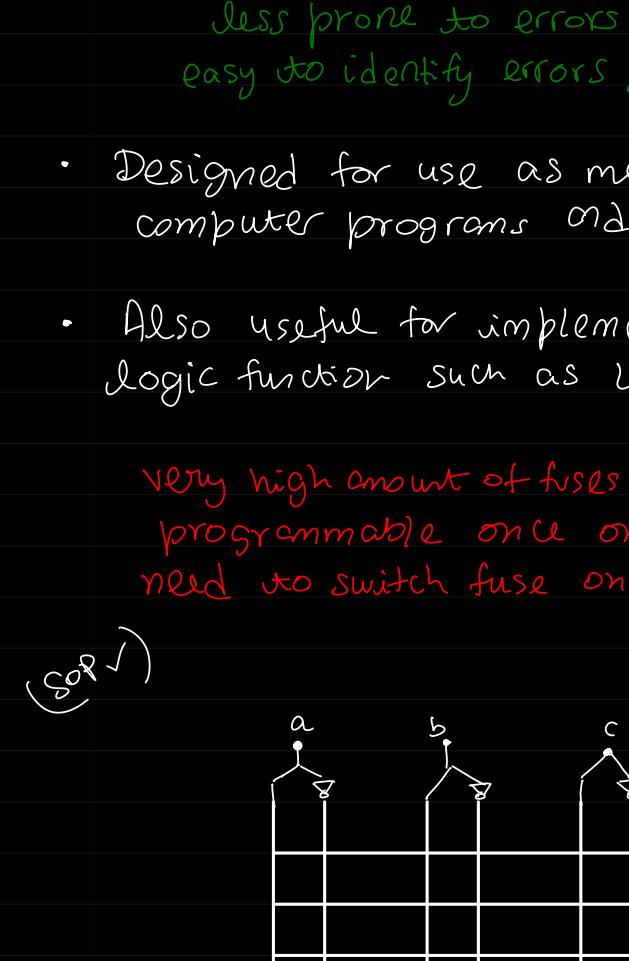
* LECTURE : 6 (Architecture)

27/08/24

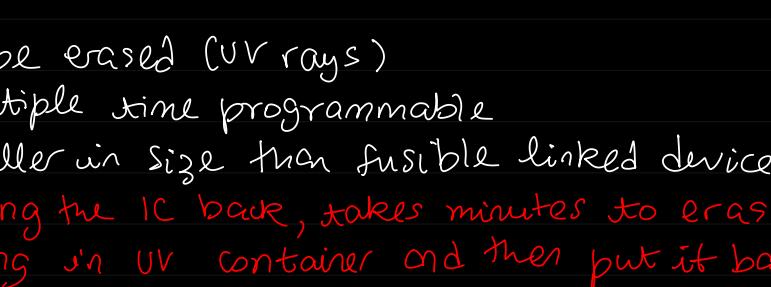
3 - 4:30pm

- 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
- burning 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

• LUT as memory & LUT as ALU

↳ lookup table

→ Operations on memory: read or write

we provide the address of the

memory → 9 bytes

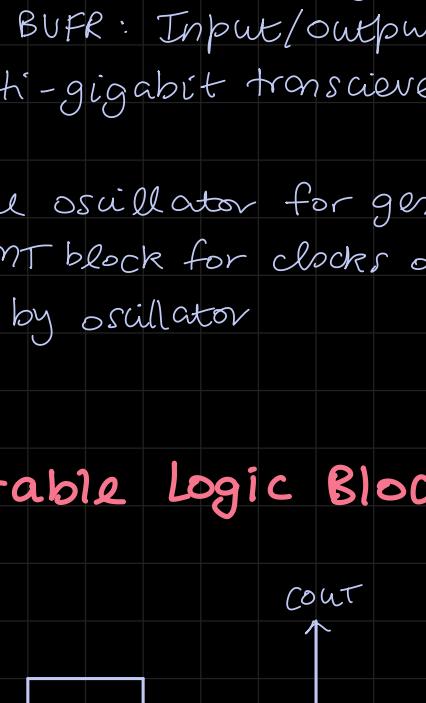
here

you provide the data and the

1 byte ← address

9 bytes ←

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

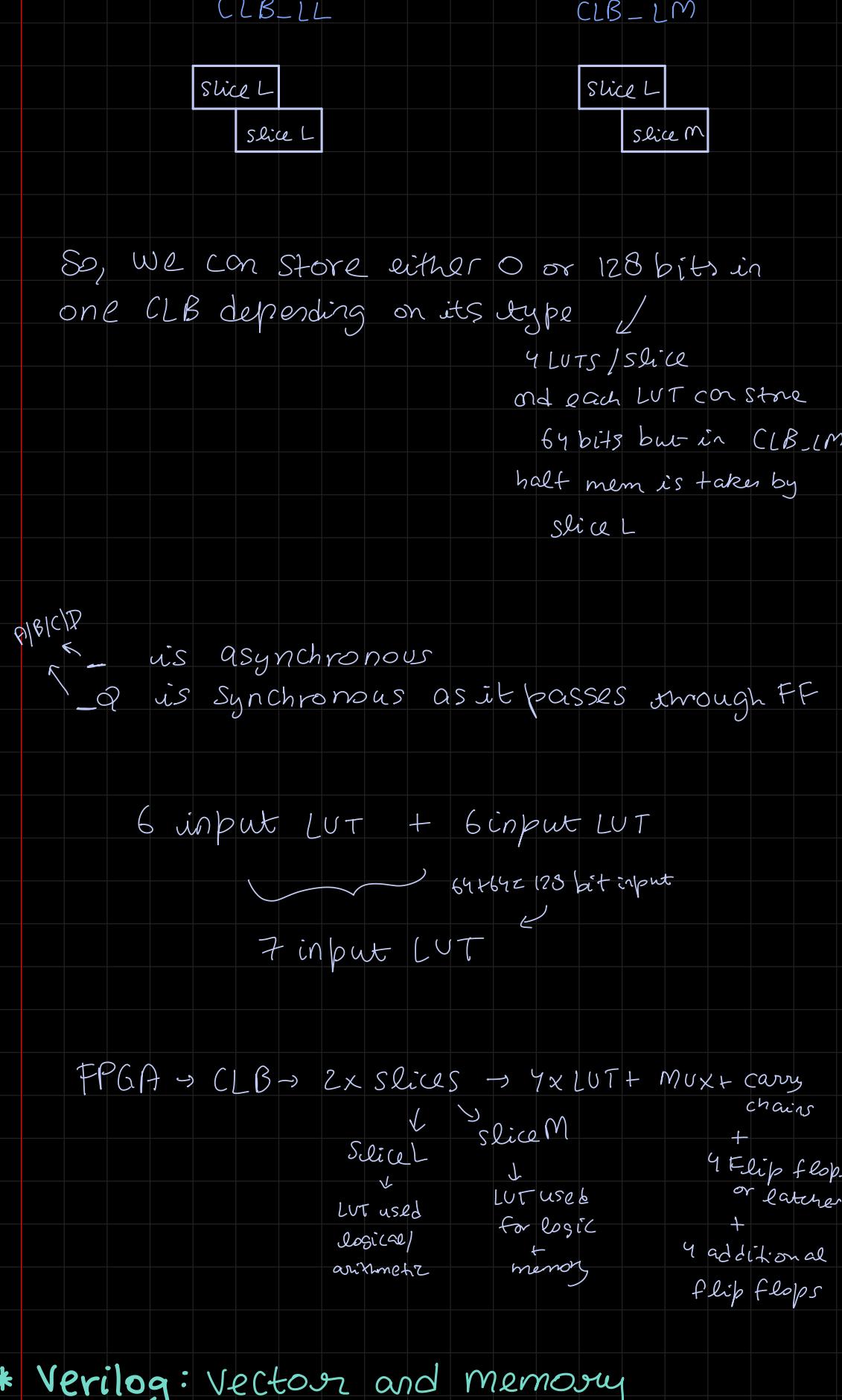
DSP: digital signal processing

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)

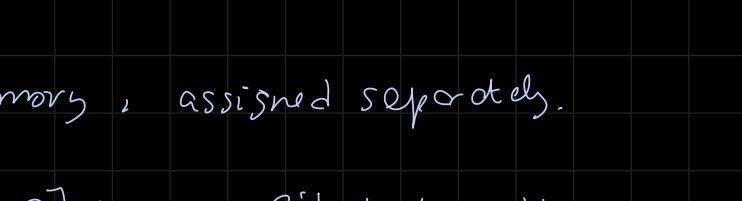
= 512 bit of data in one CLB X see below

• CLB : SLICES

① SLICEM: Full slice } read and write only
↳ 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)



• We store large amount of data in BRAM

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

CLB_LL CLB_LM

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

P/B/C/D
↳ is asynchronous
↳ is synchronous as it passes through FF

6 input LUT + 6 input LUT ↗ 64+64=128 bit input

7 input LUT ↗

FPGA → CLB → 2x slices → 4x LUT + MUX + carry chain

↓ ↓

Slice L ↗

LUT used

for logic

and memory

slice M ↗

LUT used

for logic

and memory

+ 4 flip flops or latches

+ 4 additional flip flops

* 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: 1bit (scalar)

eg: wire [7:0] a-byte ; reg [31:0] a-word;

reg [11:0] counter;

reg a;

reg [2:0] b;

a = counter[7]; index 7

b = counter[4:2]; 4, 3, 2

eg: wire [-3:0]; 4 bits sign

• memory

reg [3:0] mem[255:0], red;

↳ 4-bit vector

256 locations { 255 : 0 } ↗ each size: 4bits

with each 4bits

• 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 1bit

array [7] = ...;

array [6] = ...;

:

array [0] = ...;

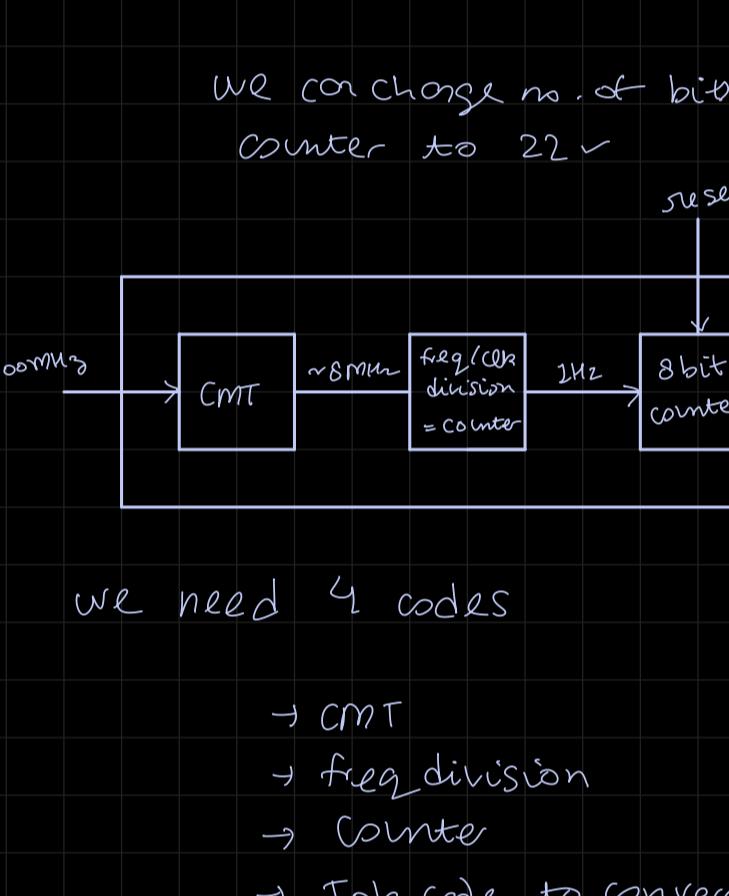
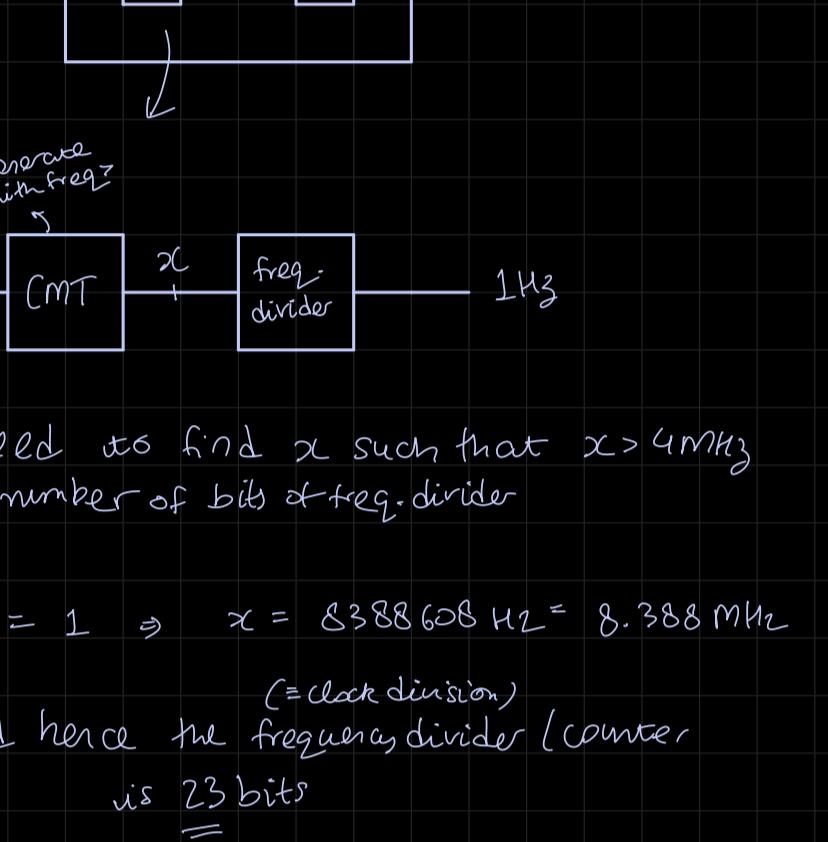
* LAB:3 (Running on hardware)

03/09/24

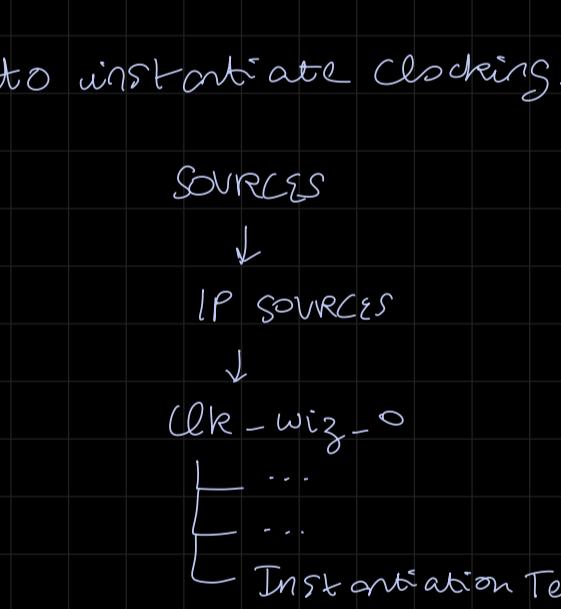
⇒ Let's say our program is an 8-bit upcounter
the program will run on the hardware
but how will you observe the output?

- 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%



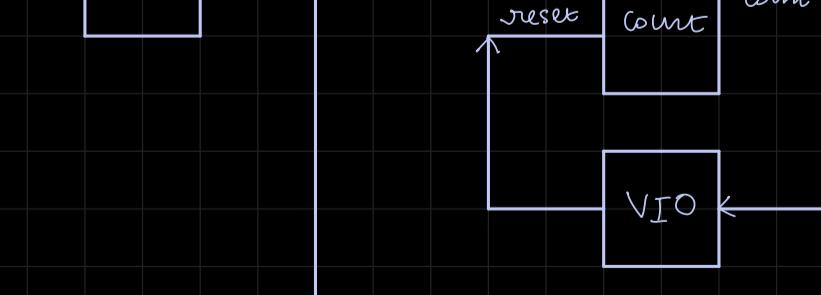
We need to find χ such that $\chi > 4\text{MHz}$ and number of bits of freq. divider

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

χ = 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

L clk-wiz-0.v

↙

copy verilog code from here to top-count.v

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

FPGA

OSC → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO → 100MHz

count[7:0] → 100MHz

reset → 100MHz

top-count → 100MHz

VIO →

* 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];`

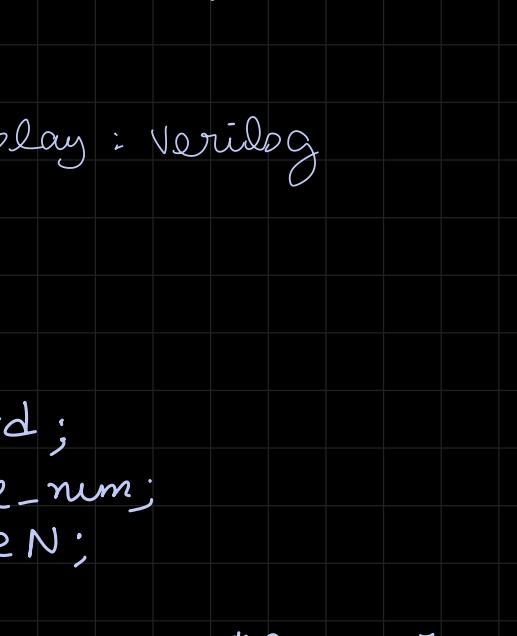
↳ memory with 32 positions of 8bit size each

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

↳ 2dimensional memory

`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]`

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

= 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

* Bus OPERATORS



[]

Bit/ Port select

`A[0]=1'b1`

{ }

Concatenation

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

{ } { }

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) << 4'b1100 (= 8+4 = 12)`

\Rightarrow

`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 <

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

↳ byte swap

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

`b=16'hAB00`