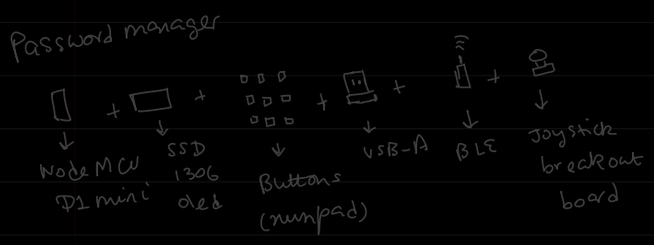
ECE 270 : Embedded Logic Design ⇒ CO + DC



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

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

Theory : FPGA and SOC

AUR ✓ ← Vivado 2019.1 (including SDK)
aur: user repository

* GRADES:

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

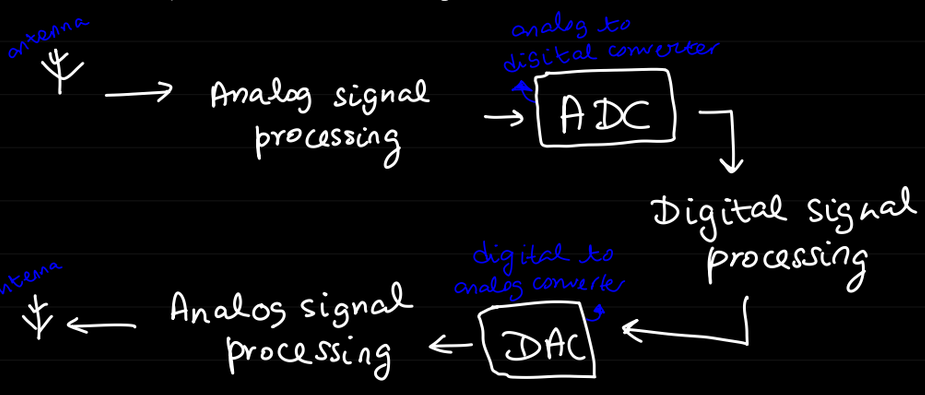
LECTURE : 1

* 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

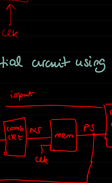
  (output depends upon current input and present state)

  what out put → output + next state (output depends upon current state & input)

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

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

* **Sequential circuit using combinational ckt**

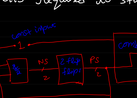


* **FSM (finite state machine)**

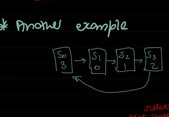
⇒ up Counter

Note : if cons state = 3, the output is 0

2 bits required to store in mem



* **Problem example**



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

relation b/w present state & next state is not fixed

eg. if $PS = S_0$ (00)
output is 3 (11)
i.e. NOT(0), NOT(0) ⇒ (1,1)

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

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

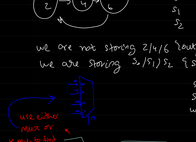
* **Example : 3**

counter : 2 → 4 → 6

S_0 00
S_1 01
S_2 10

we are not storing 2/4/6 (output)
we are storing 0/1/2 (states)

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** **LUT**
  ↳ lookup table

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

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

Conclusion,

Combinational ckts can be implemented using --

① Pure gates
② Lookup Table

* **FPGA** Field Programmable Gate Array
  (↑ bandwidth power vs. VLSI lookup tables are reprogrammable)

→ Array of generic logic gates

→ Gates where logic function can be programmed

→ Programmable interconnection b/w 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 (schematic, Verilog code, block diagram) (Verilog, Schematic, VHDL, Simulink)
↓
Behavioural Simulation (verify the design at functionality level) (ISE Simulator, ModelSim, Simulink) (Verilog, VHDL, Simulink)
↓
Logic Synthesis (translate the design into a netlist) (XST, Synplify, Synplify Pro, Precision RTL, Leonardo Spectrum) (Verilog, VHDL, Simulink)
↓
Post Synthesis Simulation (ISE Simulator, ModelSim, Simulink) (Verilog, VHDL, Simulink)
↓
mapping, placement, routing (ISE Implementation Tools, Synplify Pro, Precision RTL, Leonardo Spectrum) (Verilog, VHDL, Simulink)

ASIC/SoC | FPGA

NGD : Native Generic Database
NCD : Native circuit description
delay : Logic delay + Routing delay

- propagation delay
  - time taken for signal to pass through the gates
  - propagation delay depends upon the number of gates
- routing delay
  - time taken for signal to travel through the wires
  - routing delay depends upon the length of the wire, fanout, load on wire, capacitance and resistance of the wire

If you (Xilinx) provide tools, they typically perform

- remove the phase delay - break
- using re-timing shifting
- using multi-threaded jobs and more
- optimization has logic synthesis use their tools to look ahead at a future state or sequence more and for the future

(* .bit)

Final Step : generating the bitstream

downloaded directly to the FPGA ⇌ 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 cost and fastest time to market
non recurring engineering

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 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

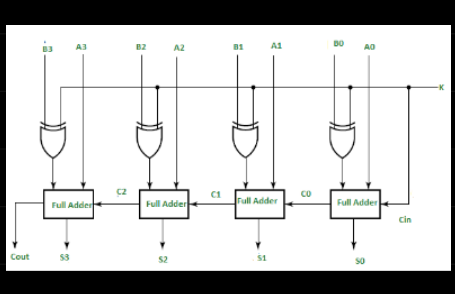
solution for the near future ≡ ARM + FPGA + GPU

micro controller : 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

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.
↓
implementation of placement, routing of generating bitstream

1987 : Verilog synthesizer by Synopsis

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 parallely unlike languages like, C, Hs etc.

verilog looks like C but describes hardware

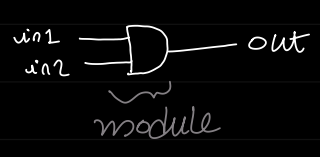
Understand the circuit and specifications then figure out the code

* 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



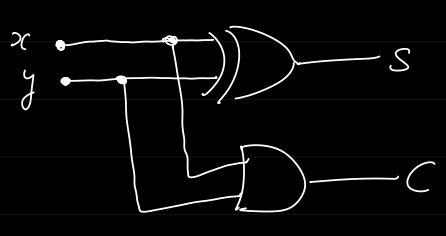
in1	in2	out
0	0	0
0	1	0
1	0	0
1	1	1

```
// <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
```

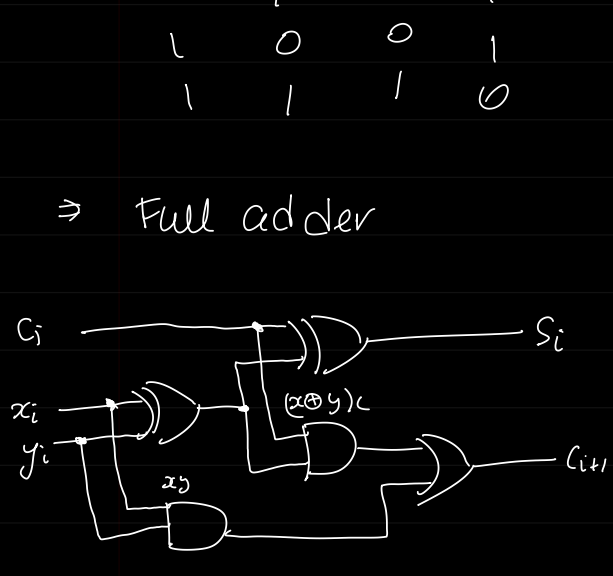
* 4BIT FULL-ADDER

⇒ Half Adder

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



⇒ Full adder

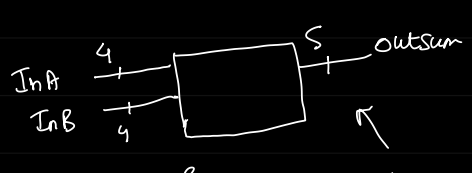


C_i	x_i	y_i	C_{i+1}	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

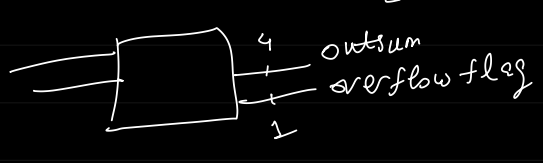
```
module full_adder_1bit (
    input FA1_InA,
    input FA1_InB,
    input FA1_InC,
    output FA1_OutSum,
    output FA1_OutC,
);

    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
HW → make the output → 4bit with 1bit for overflow



* Lecture - 4 — 2nd Oct 2020, 3:47 PM

* Verilog
  - diff b/w wire and register
  - Behavioural modelling
  - blocking / non-blocking

eg: AND gate implementation

```
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, underscore 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 [2:0];
```

* Module

- Verilog describes a digital system as a set of modules.
- each module has an interface and contents here
- Modules communicate externally with
  input
  output
  inout (bidirectional port)
- Modules consists of a set of statements defining relationships between a module and its ports 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);
endmodule
```

# of primitive, out, in1, in2, ...

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 inputs 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

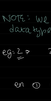
eg: exam 1

$q = in1 \& in2$

$p = \overline{in2}$

$R = in1 \& \overline{in2}$

$P1 = (1 + 3) + (2 \cdot \overline{3})$



			out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

2:1 multiplexer

in1 : 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	①	o0	o1	o2	o3
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

```
module decoder(
    input in1,
    input in2,
    input en,
    output out_0, out_1, out_2, out_3
    );
    assign out_0 = (en & ~in1 & ~in2);
    assign out_1 = (en & ~in1 & in2);
    assign out_2 = (en & in1 & ~in2);
    assign out_3 = (en & in1 & in2);
endmodule
```

eg 3 → eg = p0 + p1

p0 = ~i0 & ~i1

p1 = i0 & i1

STILL a parallel processing

→ XNOR gate

used as 1 if both inputs are same or not

eg 3 → NOTE: LHS must be of size 1 bit

2:1 multiplexer using always @

```
module test_1
    input [2:0] in1,
    input [2:0] in2,
    input sel,
    output reg [2:0] out_1
    );

    always @(*) begin
        if (sel == 1'b0)
            out_1 = in1;
        else
            out_1 = in2;
    end
```

eg 4 → 4:1 Mux

```
module mux(
    input [3:0] 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

eg 5 → D latch [active high clock] [posedge & negedge]

```
always @(*) begin
    if (clk)
        Q_out = D;
    else if (!clk)
        Q_out = Q_in;
end
```

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

HW: HDL Bits

Q1-12 — Lab 1 & 2

clk: Sept 2

submit: superior PSG

Lab no 1: Half 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: There 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 happen when input signal is active high

- Active low reset/clear/preset
  ↳ operation happen 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 DFF(clk,rst,d,q);
    input clk,rst,d ;
    output reg q;

    always @ (posedge clk or negedge rst)
    begin
        if (!rst)
            q <= 0 ;
        else
            q <= d;
    end
endmodule
```

* NOTE: <= means blocking or non-blocking we cannot assign inside always@ block because assign uses data flow but 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 inside code of D flip flop code

* MODULE PORTS : provide interface for the module
  * Interface to communicate with its environment
  input output inout

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

- Port direction can be input/output/inout

- An Input Pone
  driven by external entity

- An output Pone
  driven by internal entity

- An Inout Pone
  driven by both internal and external entities

* Module Interconnections

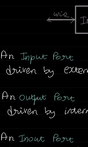
→ Named Association

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

→ Order Association

```
FA9    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'd 549
'h8FF	≡	32'h 8FF
0745	≡	32'o 745
4'b11	≡	4'b11
849	× →	8'h9 ✓
1	≡	000 , ... 0001
12'hz	≡	12 bit hexam number

* Negative Number

— [number] Signed

default : positive number unsigned

in hardware : 2' compliment

eg: -4'b11

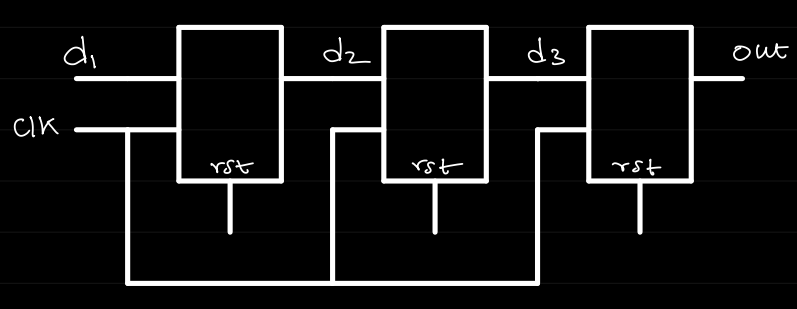
* LAB: ELD

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

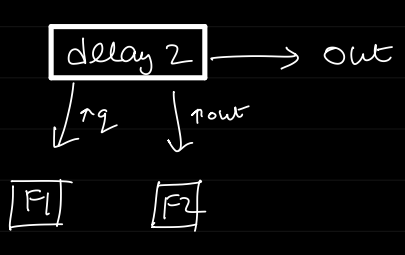
note: if we want 3 cycle delay, we pass the input through 3 d flip flops



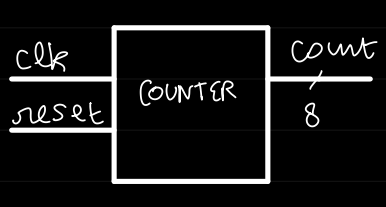

```
module delay_2(Din, clk, out);
    input Din, clk;
    output ~~reg~~* out;

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

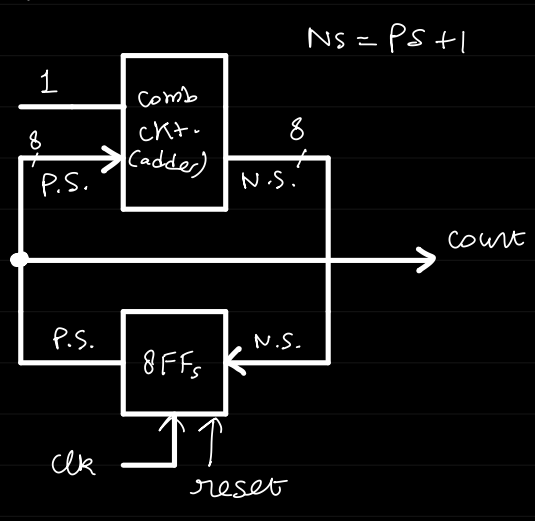
*Note: Q and out are reg inside each D_FF block but will be wire in this module because both are inputs from perspective of our module. Look up previous lecture for the same.



- 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:

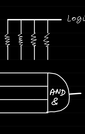
The testbench verilog file will be higher as compared to src file in context of heirarchy.

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

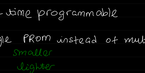
- Programmable Logic Device (PLD)
  Devices where
  Internal Architecture is predefined by manufacturer
  but are created in a way so that they can be
  configured in the field to perform variety of functions

- 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
  architect technology
  fuses control the
  connections
  fuses cannot be changed once
  blown



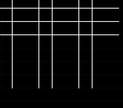
* PROM : programmable read-only memory (1970)



- 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 functions such as LUT & Store memories

  Very high amount of fuses required for complex CKTs
  programmable only once
  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
- being 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 brown people were more
  comfortable with SOP form

* Programmable Logic Device

→ SPLD : Simple
→ CPLD : complex

- Complex PLDs (CPLD) 1984

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

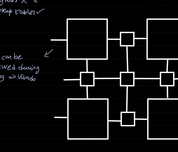
→ MegaPAL : interconnection of 4 PAL
high power consumption
Programmable AND logic
OR logic

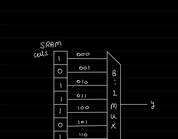
→ 1985 : Altera introduced CPLD using CMOS (high
density, low power) and EPROM / EEPROM (programmability)
EPROM
EEPROM
= multiple SPLDs

→ Added multiplexers so each SPLD so that only
the necessary part is processed
= to combat high power consumption

Signals from a logic block can travel through
adjacent blocks only

* FPGA : 1989
  gates ✓
  Lookup tables ✓





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 reading each one's state
  = hence name dynamic

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

* SRAM : Static RAM

- uses latches to store the data

  very fast for prototyping
  more space
  reprogrammable
  no refreshing needed
  eventually WORK EPROM in CPLD
  high speed, low capacity memory chips (L1 cache)
  difficult to reverse engineer

  uses 6 transistors
  large size of configuration cell
  expensive
  data can be changed due to radiations
  volatile
  will lose all data on power down
  WORK EPROM / Antifuse
  must be programmed on power up

  Can be used in Antifuse technology

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

we provide the address of the memory → 3 bits as here

you provide the data and the 3 byte address

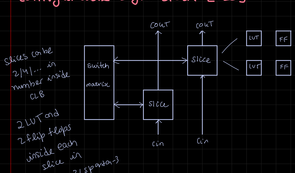
9 bits used to represent memory

* 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 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]



Slices come 2/4/... in number inside CLB

2 LUT and 2 flip flops inside each slice in Vertex-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 (4 LUT in ↳ 8 flip flop LUT)	14 (8 FF/slice)	2

total : 64 bit of data stored in LUTs
(Not the flip flop or carry chains storing)
≡ 512 bit of data in one CLB X sel between

• CLB : SLICES

① SLICEM : Full slice } read and write only
• LUT can be used for logic and memory/SRL (Shift register) } arithmetic chain

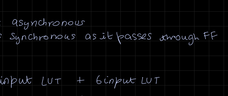
read address + write address

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

_N is asynchronous
_Q is synchronous as it passes through FF

6 input LUT + 6 input LUT
either 128 bit input
7 input LUT

FPGA → CLB → 2x slices → 4x LUT + Mux + carry chains

Slice L	Slice M
LUT used logical/arithmetic	LUT used for logic & 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 ;      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 ;
```

↳ 4 bit vector

• 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

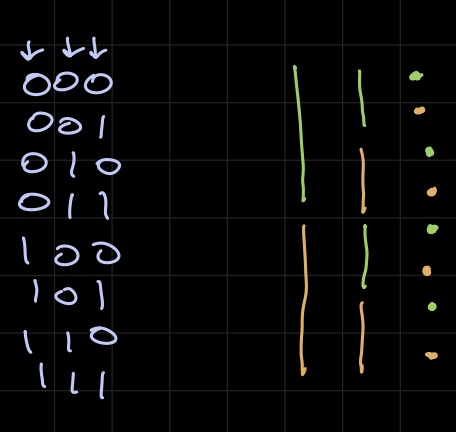
⇒ Lets say our program is an 8 bit upcounter
the program will run on the hardware
but how will you observe the output?

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

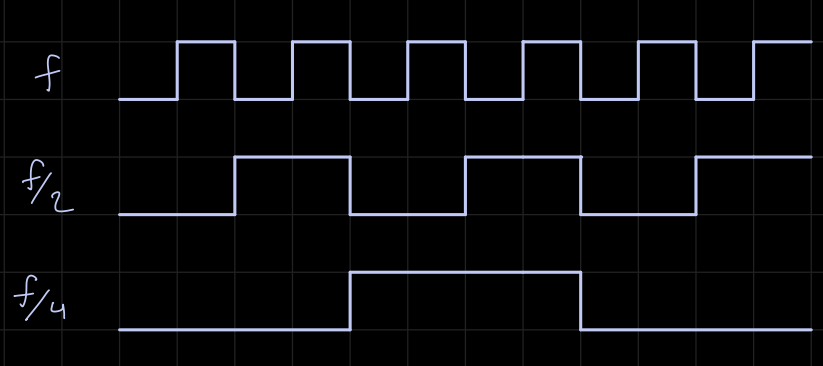
② Also, we need an additional circuit to
convert 100MHz → 1Hz
from oscillator

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



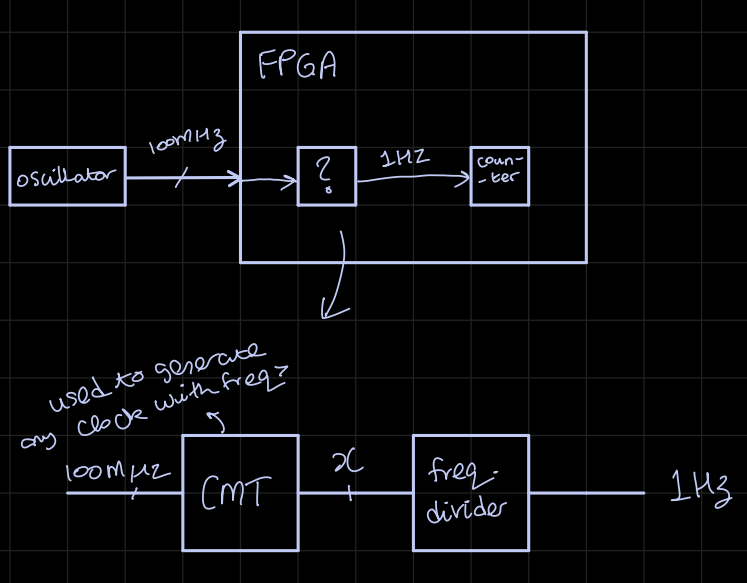
we can say this
is a frequency
divider circuit



frequency dividing with order of 2

time period for which clock is high
≡ duty cycle

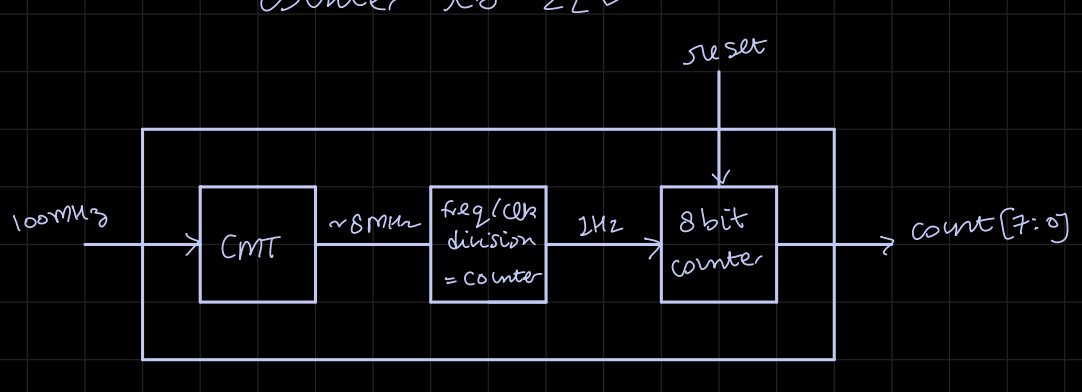
here, duty cycle is 50%



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

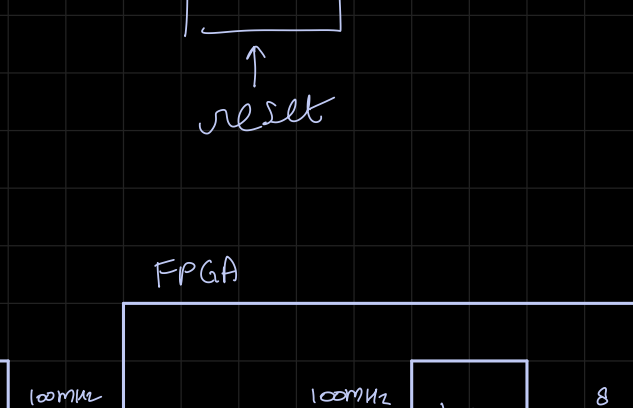
to instantiate clocking_wizard:

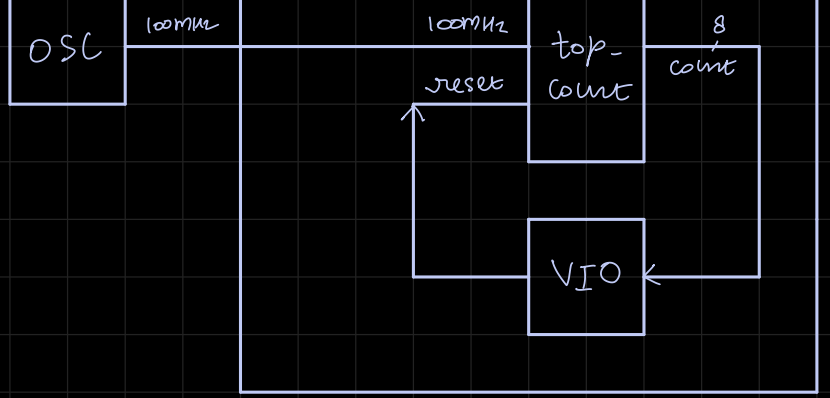
SOURCES
↓
IP SOURCES
↓
clk_wiz_0
├ ...
├ ...
└ Instantiation Template
└ clk_wiz_0.veo
↙
copy verilog code
from here to top_count.v

⇒ 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

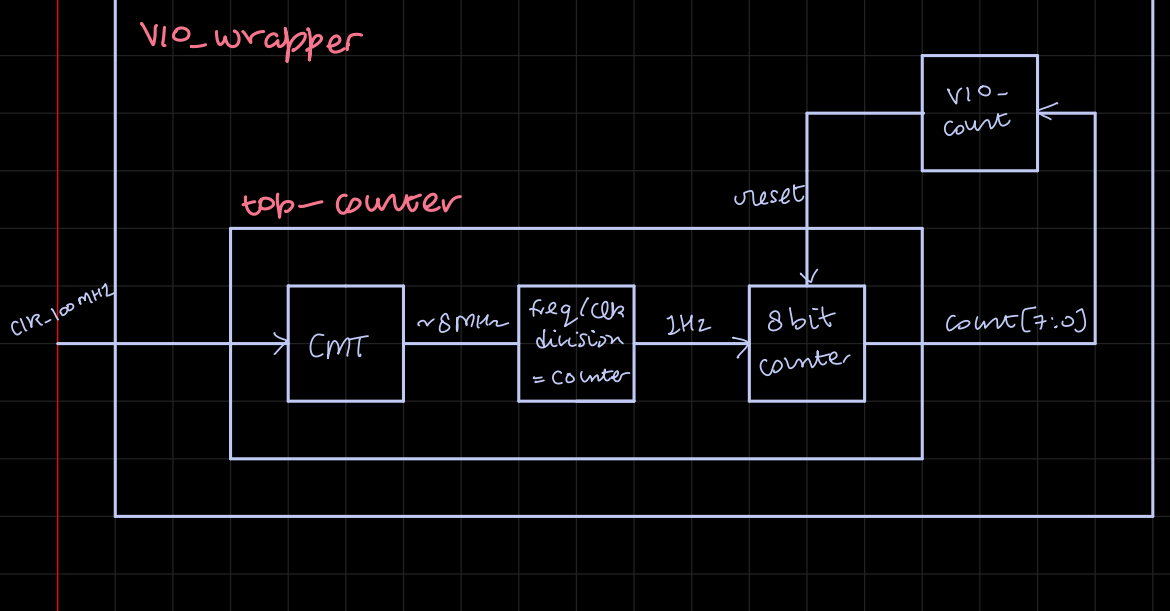




Virtual Input Output ≡ VIO

≡ 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

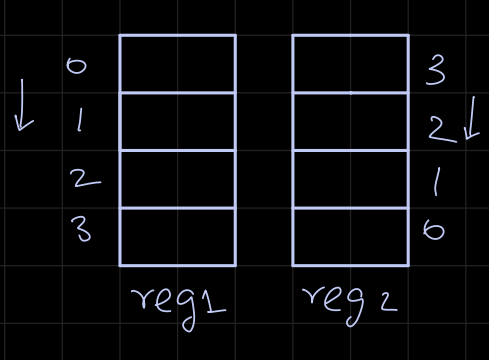
```
reg [7:0] my-reg [0:31];
```

↳ memory with 32 positions of 8bit size each

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

↳ 2dimensional memory

```
wire [1:0] reg1 [0:3];   // MSB LSB
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

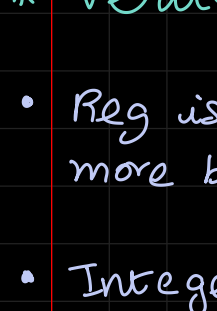
* OPERATORS

↳ unary
↳ Binary
↳ Ternary

} based on the number of operands

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

* BUS OPERATORS



[ ]	Bit/Port 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	$\times 2^x$
>>	shift right logical	$\div 2^x$

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

eg: 6 (≡ 4'b0110) $\xrightarrow{<<}$ 4'b1100 (≡ 8+4 = 12)
$\xrightarrow{>>}$ 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 (retain MSB when shifting towards LSB) works for signed 2's compliment

10001011 >>>3 = 11110001

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), − (subtraction), * (multiply), / (divide), % (modulus), ** (power)

- For multiplication, FPGAs have DSP48 dedicated for 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 number result in a max of (N+1) bits number
- For multiplication of two N bit number, 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 else: 0 (FALSE) (TRUE)

⇒ Reduction operators: output is also one bit

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

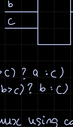
notation: <operator><operand>
eg: $\sim\&A[i] = \bigoplus_{i=0}^{n} \sim\&A[i]$

⇒ Conditional operators: condition ? true_val : false_val

2:1 mux → sel ? a : b

* PRACTICE:

```
module mux (
  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: always (used more in synthesizable code) and initial (used mostly in testbenches) in behavioural modelling
  - both run in parallel (with 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 for 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 (*) or parallely (*)
    - blocking assignment
    - 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 (was supposed to run parallel execution, and we cannot)
  (2) Incomplete sensitivity list
  (3) Incomplete branch & incomplete output assignments
      ↳ if/else error has present

* LAB : 4

10/09/24

⇒ ERROR : multi driver error
some 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)

(behavioural modelling)

always @

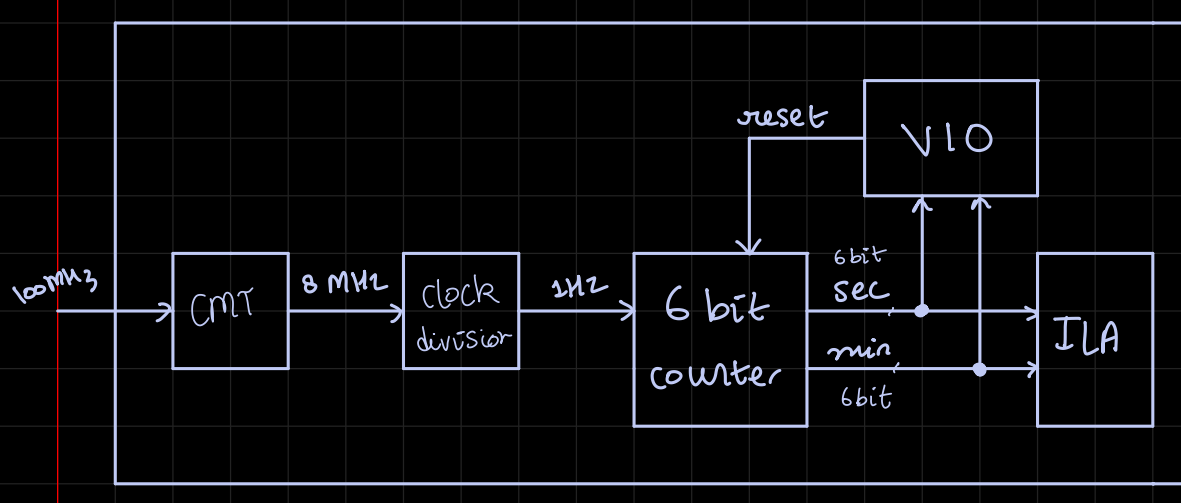
0 → 59 0 → 59

6 bit up counter

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

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

SOLUTIONS (e Guidelines)

① → have separate always block for each identifier

→ do not update multiple identifiers in one ALWAYS block

② designing MUX gate:

```
always @ (in1, in2)
begin
    out = (s1 & in1) | (s2 & in2)
end
```

Note: we are just explaining our logic through code to the tool. Some tools might give warning but some might not. Hence readability on behavioral logic

→ use always @(*) for combinational circuits.

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

③

```
always @ *
if (a<b)
    g1 = 1'b1
else if (a==b)
    g1 = 1'b1
```

→ Intention: designing combinational circuit

* Problem 1: no outputs at one always block
* Problem 2: Priority condition, only one of the two inputs will be getting 1 value and hence the other inputs will have a priority over the latter one which would cause wrong 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:

```
reg [1:0] s;

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

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

not considering case where s = 2'b01

Solution: either define all cases or use default keyword

```
default : y = 1'b1;
```

CASE	IF-ELSE
All cases are checked simultaneously	Priority based

* **CASE**
  - ↳ **full case**: all possible outcomes are covered
  - ↳ **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'b01 : out = a;
    2'b11 : out = b;
    default : out = c;
```

full case ✓
parallel ✗ because of ambiguity when sel = 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 behavioral 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

eg3

```
always @(posedge clk)
begin
    reg1 = in1;
    reg2 = @(posedge clk) in2;
    reg3 = reg1;
end
```

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

Here reg3 will have the previous value of reg1 but not in1

also it does not matter if in1 & in2 changed when clk has risen edge it will take the value at the initial posedge as calculate reg2

note: this code isn't synthesizable

eg4

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

```
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

condition: (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
```

eg5

```
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
b = a
```

(3) ② → ①

```
b = a
a = b
```

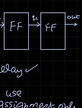
eg5 always @(posedge clk)

```
begin
    g1 = in;
    g2 = g1;
    out = g2;
end
```

2 clk cycle delay ✗

always @(posedge clk)

```
begin
    g1 ← in;
    g2 ← g1;
    out ← g2;
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

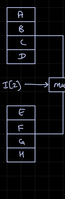
eg: I, IMUX, IQ

(MUX → multiplexer; Q → passed through flip flop (synchronous) not asynchronous in this case)

if we combine 2 6bit-LUTs: we get a 7bit LUT

↳ 128 locations

61 locations of 7bit size each

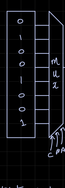


combining two 2bit LUTs to get a 3bit LUTs

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

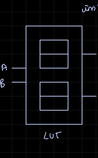
* 3input LUT



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

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

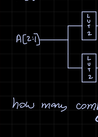
so, only 3-comb ckt can be implemented in one 3input LUT



this is 2input LUT, x2 how many comb ckt of 2 inputs can be implemented ?
1
because 2input = 4 location and so either of the output can be used at one

how many comb ckt of 1 input can be implemented ?
2
$2^{2^1} = 2$
because $n_{LUT} = 2$

• LUT 3 architecture



how many comb ckts for 3bit input ?
ONE

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

limitation

2 x 2bit inputs: $f_1(A,B) = 31$
$f_2(C,D) = 32$

but we only have 3 input 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:

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

3 inputs ✓
2 output ✓
8 locations ?

each LUT can store 4 location

$f_1(A,B)$ OR
$f_2(A,C)$ AND
×

but you need 8 locations i.e. 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 2 inputs 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 location 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
$f_1(A,B,C,D,E)$	6	1	—
$f_2(A,B,C,D,F)$	← 5	2	common input
	3 & 2 (other)	2	—

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

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

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

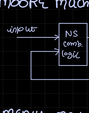
6 inputs: 2 for sel
3 upper LUT
2 lower LUT

$f_1(A,B,C)$ }
$f_2(D,E,F)$ } ×

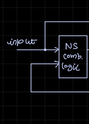
because we have only 5 inputs (just 1 is see)

* Finite state machine

① MOORE Machine



② MEALY Machine



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

FSM

mission code: write a FSM code for something..

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

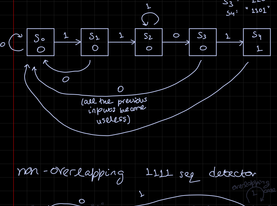
1011 sequence detector

types: overlapping / non-overlapping

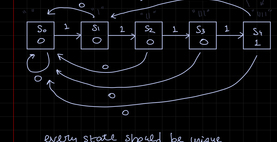
→ FSM: finite state machine
every state has a meaning

MOORE

S_0 : " "
S_1 : " 1 "
S_2 : " 11 "
S_3 : " 110 "
S_4 : " 1101 "



non-overlapping 1111 seq detector



every state should be unique

Verilog code for FSM (moore)

5 steps

(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
         ...
         ...
         ...
    S3 : if (din == 1)
             next_state = S4
         ...
         ...
         ...
```

⑤ output comb ckt

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

note: we will use 2 push buttons as inputs

PB3, IIA not needed
Logic
Only counter that will change from last clock

digital adder

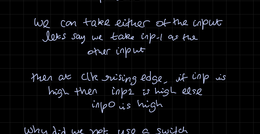
output of FSM: detect (1 bit)

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 inp1 as the other input

then at clk rising edge, if inp is high then inp2 is high else inp0 is high

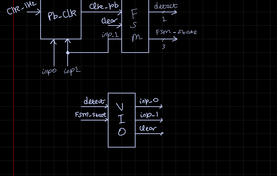
Why did we not use a switch, because when we have pushed down, it assumes you are continuously sending 1 and vice versa

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 — 19/09/24

→ SLICE ARCHITECTURE

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

note: memory
↳ LUT
↳ BRAM

* LUT

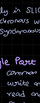
↳ combinational ckts
↳ memory
↳ shift register

① LUT as memory

- only in SLICEM
- Synchronous write operation, WEN must be high (write enable)
- asynchronous / synchronous read operation (read enable)

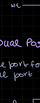
* Single Port

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



* Dual Port

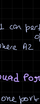
one port for async write + sync read
one port for async read



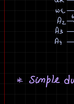
A2 can perform both read and write operations
where A2 can only do read operation

* 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: 64x1 quad port
↓ 64 locations
↑ bit : size of each location
size of din/dout = 1 bit
↓ 1 read/write port
↓ 3 read port

Note: In LUT architecture,
the topmost block has its read and write address lines shorted whereas rest the core with below 3 blocks.

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

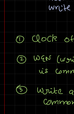
① Clock of all LUTs is common
② WEN (write enable) of all LUTs is common
③ Write address of all LUTs is common

So when you perform write operation on any LUT, write operation will happen on all LUTs
(= observation, might not be correct flow)
④ Read Address of all LUTs is different
⑤ For first LUT, WA and RA are same

* LUT as MEMORY : 64x1 single port

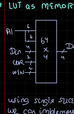
How many LUTs needed to implement this?
→ one 64x1 LUT ✓

* LUT as MEMORY : 64x4 single port



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

* 64x1: Dual Port (= 2LUTs)



writing same data Din in both LUTs (duplicate cell)
so both can read
so we can read at it separately at 2 ports and read/write at 1 port

128 bits at a time
we need 2 LUTs
⇒ 256 bits = 4 LUTs ✓

* 64x2 : dual port

↳ replicate the data more, more times

* 64x2 : quad port ✓

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

* 64x3 : simple dual port

just using 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 address

So we use 2 top and 2 bottom LUTs for 256 64 locations and bottom 64 locations respectively

but left is common all LUTs
Therefore we have two "LUT select" bit
MS & LS from cycle through LUTs

256 x 1 quad port ✗
128 x 1 quad port ✗
128 x 1 dual port ✓ ⇒ required 128x2 = 4 LUTs ✓

32 x 16 single port ✗ ⇒ repeat x 4
and 256 x 512

* Lecture : 13 — 18/09/21

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

```
module SRL (
    input in,
    input clen,
    input clk,
    output qb;
    )

wire qa, qb, qc, qd;

always @ (posedge clk or posedge clen)
    if(clen)
        qa <= 0;
    else
        qa <= in;

always @ (posedge clk or posedge clen)
    if(clen)
        qb <= 0;
    else
        b <= qa;

always @ (posedge clk or posedge clen)
    if(clen)
        qc <= 0;
    else
        c <= qb;

always @ (posedge clk or posedge clen)
    if(clen)
        qd <= 0;
    else
        d <= qc;
endmodule
```

D-FF(in, qa);
D-FF(qa, qb);
D-FF(qb, qc);
D-FF(qc, qd);

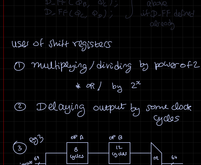
} Same as above
} with FF defined
below

Use of shift registers

① Multiplying / dividing by power of 2

* or / by 2^x

② Delaying output by some clock cycles



Bunch
Flip flops used

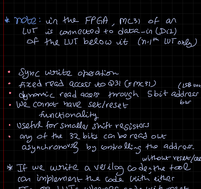
2 slice has 8 FFs
So, 8x16
slices required
to delay in 16 cycles
by 16 cycles

If a FF, & 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 the same time, we need to delay the cycle delay

to reduce number of FFs needed we use SRL31 on the FFs

* LUT as Shift Register

SRL31 is delayed version of input by 32 cycles



Bunch
How much cycles
to 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 (in the LUT array)

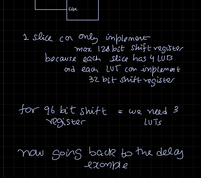
- Sync write operation.
- Area read access via O31 (MC31) (LUB only)
- Dynamic read access through 5bit address bus
- We power up SRL16/SRL32 functionality
- Useful for smaller shift registers
- Any of the 32 bits can be read out asynchronously by specifying the address without reset/clock
* If we write a Verilog code for the FF, can implement the code both either FFs or LUT. Synthesizer code with reset will be implemented using FF only.

When shifting, it shifts towards 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

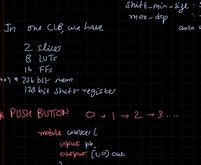
We use 2 LUTs



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

for 96 bit shift we need 3 register LUTs

now going back to the delay example



Bunch
Flip flops used

2 slice has 8 FFs
So, 8x16=128
slices required
to delay in 16 cycles
by 16 cycles

every slice has 4 LUT

6 bit input → 16 delay

max 32 bit delay by one LUT
but we need 16 bit delay so we use address bus
with max delay

So, 2 slice can give 128 bit

but we need 64x16

So, 16 slices needed

we just need to remove extra unnecessary clock buffering and we need to add pipelining to prefer LUT

136 → 16 slices needed
(FF) (LUT)

68 → 16 : CLBs needed

In one CLB we have one SLICEM

project settings → synthesis
↓
Shift-min-size : 3
max-dip : auto

In one CLB, we have

2 slices
8 LUTs
16 FFs
(4x4) * 2 = 32 bit mem
128 bit shift register

* PUSH BUTTON

0 → 1 → 2 → 3 ...

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

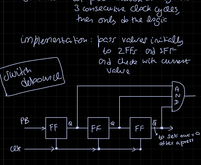
but we get ① → ② → ③

due to push button debounce

when we push once, it takes some time to reach

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

Implementation : past values initially to 2'FF and 3'FF and check with current value



* Lab :6 ⇒ AXI Interface 24/09/24

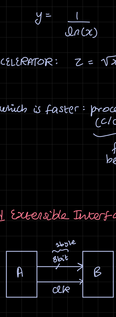
- basics of AXI Interface
- design of floating point arithmetic

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

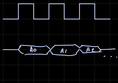
- HW: ACCELERATOR: $z = \sqrt{x} + \frac{1}{\log y} + 1.5$

- compare which is faster: processor (C/C++) vs FPGA (verilog)
  for communication between the two ⇒ AXI interface

* Advanced Extensible Interface (AXI)



data: A[10]
8 byte each
10 bytes each

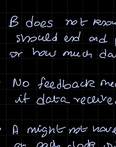


like I2C interface data sent on clock edge.

Problem:

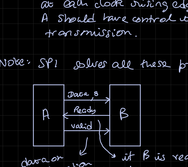
- B does not know when transmission should end and parse the data or how much data A should send
- No feedback mechanism from B→A if data received correctly or not
- A might not have all data ready on each clock rising edge. A should have control to pause transmission.

> 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 but B not ready

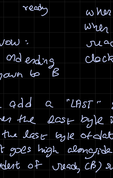
VALID before READY handshake

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

PROBLEM Now:

- Starting and ending point unknown to B

So, we add a "LAST" signal which goes high when the last byte 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)

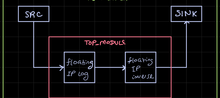
- Valid should not wait until Ready becomes one from a dead lock
- one valid is set to high, it CANNOT be reset until handshake happens
- DEST can set Ready independent of VALID it is allowed to go back to LOW after being set to high even before VALID is set to high

> Note: Clock is needed in order to know when transmission is occuring (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
one 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

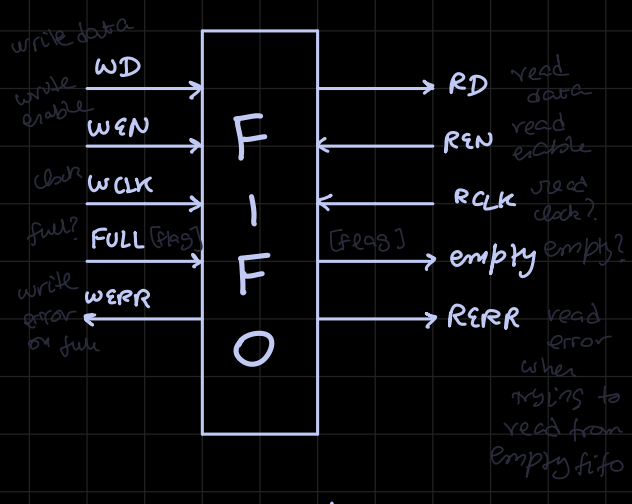
ultra RAM — 288Kb

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

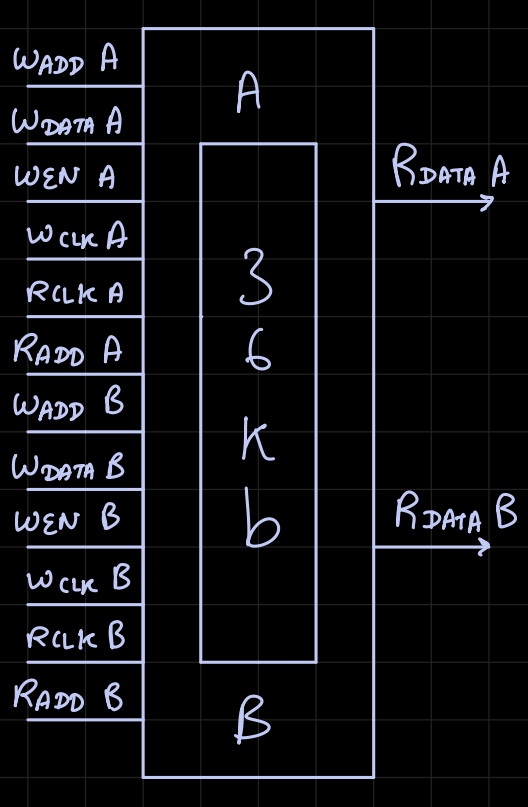
needs internal counter to get from where to read / write data

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

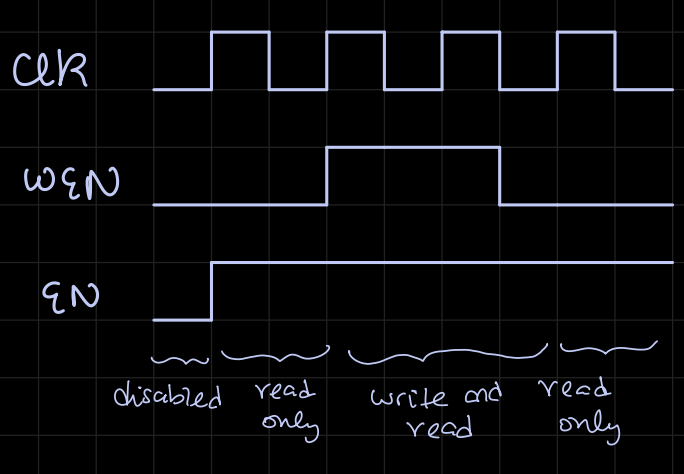


both ports are symmetrical and totally independent, sharing only the shared data.

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

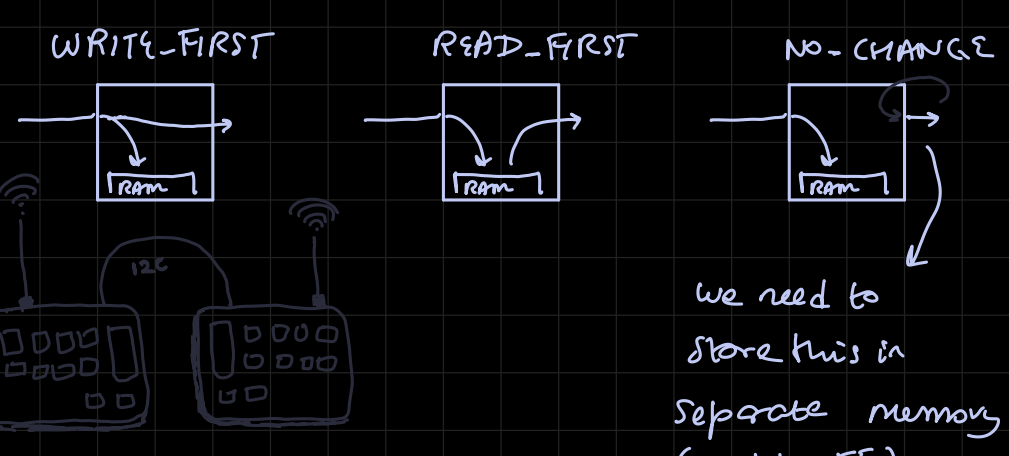


• Configurations (WRITE MODES)

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 ✓



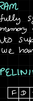
we need to store this in separate memory (could be FF)

* lecture : 15 24/09/21

→ BRAM

- fully synchronous operations
- memory reads/writes are controlled by the clk
- RD Support read-first and no-change modes, we have a latch and register at the end.

• PIPELINING



Increase in Max clock? ↓↓
↓↓
increases efficiency / throughput of existing instructions ✓

- output latches will be loaded only when write mode = read-first and write-first
- first data gets loaded into output latch then write operation in READ-FIRST

- first write operation to memory and then reads updated data. No output latch in WRITE-FIRST

- When WE is inactive and EN is active, read operation happens and the data outputed 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 block

→ we need 10bits for address but only 8 bits for data bus now, for 2K → 9 bits add bus = 11 bits data bus = 8 bits now, for 4K → 2 bits add bus = 12 bits data bus = 2 bits now, for 8K → 3 bits add bus = 13 bits data bus = 1 bit

USING PARITY BITS we can check for any errors in data

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

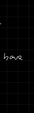
So, for 1K locations with add size 8 bit, we can add an additional parity bit

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

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

2K → 32 + 4
2K → 16 + 2
4K → 8 + 1
8K → 4 + 0
16K → 2 + 0
32K → 1 + 0

maximum address is 2^{15} address 2^{15} 15 bits



allows 2 Adjacent BRAM to cascade. Dual BRAM is 16 locks and not 15 bits

why do we have 4bits for WEN?
eg: WEN = 1000 ie enabled for one byte ie 8 + 8 + 8 + 8
we can write one byte data at a time using this

eg: F X X X
garbage value

for WEN = 0110
→ X A B X

looks fine ✓

This is called **Byte-Wide write enable**
→ allows writing byte bit (one byte) portion of incoming data.

if last read data = ABCD
new data in = XFXX
with mode = READ-first output = ABCD
but in WRITE-FIRST, output = ABFD
Garbage value at value and previous value

→ write enable is only for the data on write, WEN = 1 5 assumed that data in will have parity bit included

Dual Port

- Two separate write/read ports
- each BRAM can be clocked also two individual 18Kb BRAM

configurations:

Device	RAM size (bits)
32K	1
16K	2
8K	4
4K	8/9
2K	16/18
1K	32/36

② SIMPLE DUAL PORT BRAM

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

you can use half the BRAM if you want to use simple dual port mode

Device	RAM size (bits)
32K	1
16K	2
8K	4
4K	8/9
2K	16/18
1K	32/36
512	64/72

③ SINGLE PORT BLOCK RAM

- 3US Xenon port B from dual port BRAM. to achieve single port BRAM with port = A
- So, we have 18Kb single port BRAM

Device	RAM size (bits)
16K	1
8K	2
4K	4
2K	8
1K	16

* cascading only supported in dual port memory

• CONFLICT

- Series FPGA has a true dual port BRAM.
- No problem when both ports are reading (but it reading from same location)
- But both ports working different data, its a conflict, memory location ≡ data corruption
  Non-deterministic data.
- When one port is working and other is reading, write is successful but the reading read value may vary depending on clock cycles and write mode.

clock cycle	write port A	write port B	write mode A	write mode B	WEN A	WEN B	Data A	Data B	reading from value
common	RF	WF	RF	WF	0	0	old	old	no change
common	RF	WF	RF	WF	1	0	old	old	new (A)
common	WF	WF	RF	WF	1	0	new	old	new (A)
common	NC	WF	RF	WF	1	0	no change	X	new (A)
common	RF	WF	RF	WF	1	1	old	old	new (B)
common	WF	WF	RF	WF	1	1	new	new	new (B)
common	NC	WF	RF	WF	1	1	no change	X	X
common	RF	WF	RF	WF	1	1	X	X	X

↳ Data corruption occurs

both ports are supposed to write on same location, so what is stored and put in the output latch is undefined and it will simulate as data corruption. Hence we put X.

clock cycle	write port A	write port B	write mode A	write mode B	WEN A	WEN B	Data A	Data B	reading from value
independent	RF	WF	RF	WF	0	0	old	old	no change
independent	RF	WF	RF	WF	1	0	old	X	new (A)
independent	WF	WF	RF	WF	1	0	new	X	new (A)
independent	NC	WF	RF	WF	1	0	no change	X	new (A)
independent	RF	WF	RF	WF	0	1	X	old	new (B)
independent	WF	WF	RF	WF	0	1	X	new	new (B)
independent	NC	WF	RF	WF	0	1	X	no change	new (B)
independent	RF	WF	RF	WF	1	1	X	X	X

↳ Data corruption occurs

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

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

CPU → parallel processing
CPU + GPU
CPU + HW {ASIC, FPGA} → accelerators (specific use cases)
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

$$X(K) = \sum_{n=0}^{N-1} x[n] e^{-jK\omega n} = \sum_{n=0}^{N-1} x[n] e^{-jK\frac{2\pi}{N}n}$$

$X(K)$ ↓ 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)
$0 \to 3$

F.T. ⇒ matrix vector multiplication
$n : 0 \to N-1$
$K : 0 \to N-1$
no. of multiplications required : N^2
16 here

FFT: fast fourier transform ↓

$N \propto$ time to perform operation

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

$x[n] \longrightarrow \boxed{FFT} \longrightarrow X(K)$
configuration → FFT

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 1);
fft(x);
```

* Lecture : 16

ARM and FPGA communication ?
↳ AXI Protocol

advanced extensible interface ← AXI is native

Zynq = ARM

some IP provide both
some modern chips have only AXI

ARM $\xleftrightarrow{AXI}$ FPGA

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

AXI : very 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
- construction storing

AHB used 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

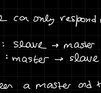
memory map

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

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

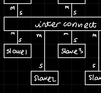
- **AXI :**

- 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



→ one error will know the start address and the end address and has to do only r/w operation

- **AXI memory mapped**



total channels 5

write transaction : 3 channels
read transaction : 2 channels

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

BURST TRANSFER

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

because in sending read addr and reading everything, lot of time is lost in just sending the read addr

Read Transaction

read address
AR = read control
R = read data

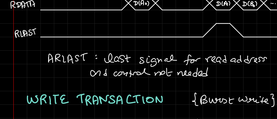
} 2 channels of Read transaction

	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 : one 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

WLAST signal is present only in read data & write data

* Lecture 17

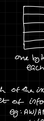
17/10/24

Today lecture on AXI
next week → AXI

- Byte addressable memory

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

eg:



- each of the ARM bus 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 then a amount of data from that address.

BURSTS can vary from 1 - 256 data transfers per burst

- each BURST transfer size can be 1 - 128 bytes (must be a form of 2^n etc)
- max data per burst → 256 x 128 = 32kb (i.e. for one address)

what if i want more : two options
no. of bursts : 256 → 129
OR each burst size : 128 → 64

how about 3 byte of data?
no. one each size = 1

4 bytes? → 1x4 or 4x1

3 bytes? → burst : 3
transfer size : 1

258 bytes? → 129 x 2

259 bytes? → need NULL transfer

Byte Lane strobe signal

for every 8 bit of data, indicates which bytes of data are valid.
BLSS size = burst transfer size

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

now for 259 bytes data transfer →

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

for 7 bytes data transfer and we fixed 4 data transfers/burst

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

expect most data : null & not valid

AXI : Read

AXLK
ARID[3:0] identifier for when we have multiple slaves master is interconnect
ARADDR
ARSIZE[2:0] size of each transfer (1 → 128)
ARVALID
ARREADY

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

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

INCREMENT MODE

as we send data → it increases at the data. NEXT to the current one

we send the ARADDR

now lets say we have 4 location of 4 bytes size each and we send singles eg. ARSIZE=1

then the entire data ABCD is overwritten and EFGH is written since we have disabled incrementing

111

WRAP MODE (not incremental mode)

eg. 0x1, 0x2, 0x3, 0x0

FIFO MODE / FIXED MODE

writing data to same address, say, A2

≡ 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 transfered 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 ↓

WR valid (WVALID)	1	1	1	1
strobe signal →	1	F	F	1

new signals : ID
Size
STROBE
BURST TYPE

AXI memory 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 small or 32bit data

for LITE → memory mapped in timing diagrams
↳ add specific address 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 takes power
  ↓
  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 how we use AXI : like this

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

next week Lab: FFT in C on ARM processor

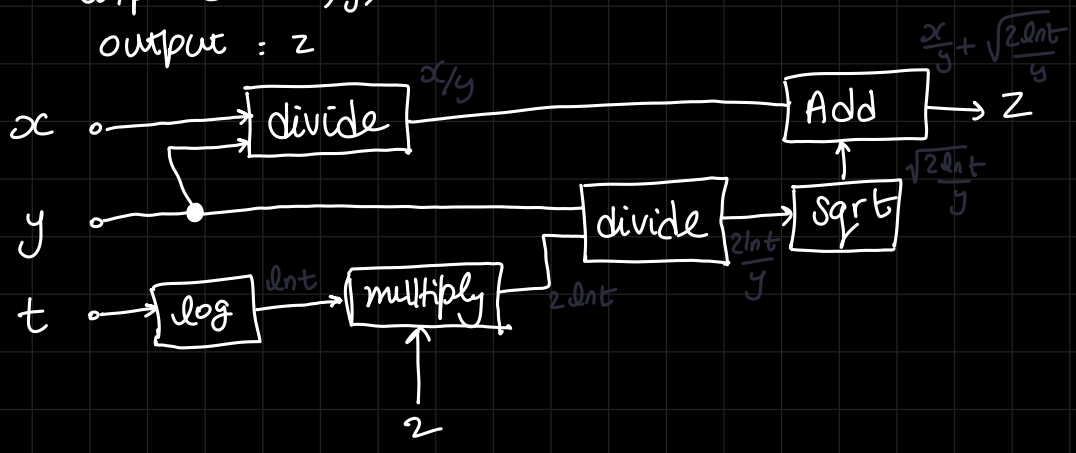
* CPU configs the DMA

* Lab 7 HW_2

objective: $z = \frac{x}{y} + \sqrt{\frac{2 \cdot \ln t}{y}}$

inputs : x, y, t

output : z



- Lecture : 18

AXI INTERCONNECT

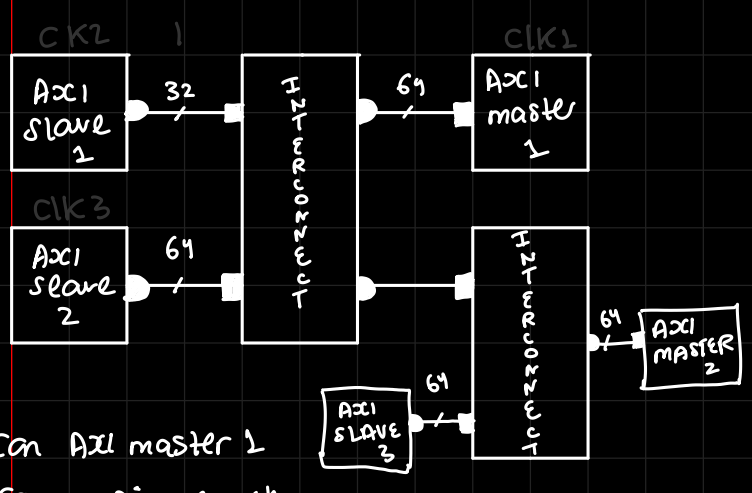
↳ capable of width conversion
↳ multiple number of masters & slaves
↳ capable of conversion between different AXI standards (AXI 3 ⇌ AXI 4)
↳ Clock domain transformations → since Processor has higher clock frequency than FPGA

→ higher clock freq preferred for better performance but needs more power

→ what if we connect one block which operates at 2Hz clock and other operating at 1Hz clock

↓

the 2Hz block is producing 2 samples per second whereas the other is producing one/s so, 2Hz block is loosing data. So, we need a buffer to capture the lost data. Hence we use AXI interface between the two blocks and the interconnect needs to have common clock which can be calculated so that there is no data loss btw the two blocks.



Can AXI master 1 communicate with AXI slave 3?
No, because only master can initiate communication but interconnect 1 → 2 is connected slave → Master

Strobe signal is not that useful in case of read transaction because the data would be of sufficient length or amount when returned from slave

---

- System

↳ A way of working, organizing or doing one / more tasks according to a fixed plan, program, set of rules

Embedded System : designed to run on its own without human intervention and may be required to respond to events in real time

↳ combination of HW & SW which together form a component of a large machine

embedded design, design, advanced programming, multicast, multiple, circuit, theory, E-devices, signals & systems

demands : higher integration, higher performance, low power consumption, low cost, flexibility, scalability

- Hardware : ASIC vs ASSP

PROCESSOR ⇌ FPGA (external link)
≡ system on board SoB

	ASIC	ASSP	2 chip sol^n
Performance	+	+	− (because of delay in communication)
Power	+	+	−
unit cost	+	+	• (Price : FPGA >> ASIC)
TCO	•	+	+
Risk	−	+	+
TTM (time to market)	−	+	+
Flexibility	−	−	+
Scalability	−	•	+

TCO ↓ total cost of ownership
↳ takes into account all the cost associated with development with such sol^n
↓
TCO is high for ASIC as depends on how many units to be produced

ASIC TCO ↓ very high because you are designing your own chip.

2 chip sol^n TCO ↓ fabrication : someone else is doing it for you

for a startup, ASSP is a better choice due to limited fabrication resources and lack of max power but you risk leaking your design since you are outsourcing fabrication

How to improve SoB's price and performance?
fuse the processor & FPGA
≡ System on a chip SoC

- Advantages of SoC

- Higher Performance
- Power Efficiency — less voltage required : < 2V
- Lighter footprint — device size and weight reduced
- Higher reliability
- Lower cost — per unit cost is reduced
- Less flexibility — SoC cannot be upgraded easily after manufacture
- Application Specific — cannot be easily adapted to other applications
- Complexity — requires advance skills

// Types : FPGA based SoCs OR ASIC based SoCs

- Lab 9
  Handwritten notes

- Lecture 20 12/11/24

pipelining ≠ parallel processing
↓
utilizing existing hardware for getting higher performance

better throughput

if pipelining not enough? → parallel processing

F → D → X → M → WB

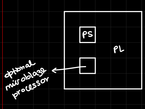
nowadays we have 9 different stages

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

DMA : direct memory access

Stream ⟹ memory mapped

memory controller
coprocessor



FPE: floating point engine
↳ a coprocessor specifically used for floating point operations

Eg of coprocessors: FPE / NEON / GPU / RTC

CNN: convolution neural network

- Convolution

$$Y_{(k)} = \sum_{i=0}^{i=N} w_i X_{(k-i)}$$ } MAC : multiplication & accumulate operation

very high overhead

$Y_k = w_0 X_k + w_1 X_{k-1} + \cdots$

$Y_0 = w_0 X_0 + w_1 X_{-1} + w_2 X_{-2} + \cdots = w_0 X_0$
we don't know previous values

$Y_1 = w_0 X_1 + w_1 X_0 + w_2 X_{-1} + \cdots = w_0 X_1 + w_1 X_0$

$Y_2 = w_0 X_2 + w_1 X_1 + w_2 X_0$

$Y_3 = w_0 X_3 + w_1 X_2 + w_2 X_1 + w_3 X_0$
because $w_3 = 0$ (given)

if $w_0, w_1, w_2 = 1$,
this is addition operation (accumulation)

parallel computing process by nature
↓
works on processor but best carried out in FPGA

KSPS: kilo samples per second
↳ clk rate × 1000 (to account for multiple word)
clk cycle needed for 1 operation

There is another coprocessor which is an intermediate of PS, PL

SIMD
single instruction, multiple data
= NEON Processor
only one op^n at a time

we have this in our ARM cortex A9.

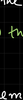
memory mapped: each external unit has to have a starting address & an end address

- Address Allocations

making an IP memory mapped

↓

memory mapped IP



load A
load B
add C A B
Set status "done"
processor reads "done"
then reads value of C

let each block = 32bit
then the mem requirement is = 128bit

now this IP is AXI compatible

for a mem requirement of 32 bytes, we have a 5bit addr bus
from 00000
to
11111

for 1kb → 10bit addr bus
1024 locations

for 1MB → 20bit addr bus
1GB → 30bit addr bus

let mem req = 1kb,

start addr : 0x4000_00FF

if it was all 0, then end addr would be 11111 11111
⇒ 0x3FF

now, end add = 0x4000_03FF

let start addr = 0x4000_0800
end addr = 0x4000_0BFF

let mem req = 1MB,

then we need to add FFFFF

Start : 0x C000_0000
mem : 1GB

add : 3FFF_FFFF

ans : FFFF_FFFF

Rule 2: Starting address should be multiple of mem requirement of the IP

eg: for mem = 1kb → addr bus = 10bit
= +0x03FF+1 = +0x400

SA1 0x0000_0000
SA2 0x0000_0400
SA3 0x0000_0800
SA4 0x0000_0C00

- Lecture: 21 14/11/24

256 byte valid ✓

mem req.

4kb	2kb
0x4000_0000	0x4000_0000
0x4000_1000	0x4000_0800
0x4000_2000	0x4000_1000

Address: 0x4000 - 0100
valid for → + 0x0000-00FF
256byte ← 8 bit ← 1111 1111

→ mem: 16MB ⇒ 24bit addr bus

+ 6F5

0x4000_0000
0x4100_0000
0x4200_0000

→ 4MB → 22 bit addr bus

+ 3, 5F → + 4000000

0x4000_0000
0x4040_0000
0x4080_0000
0x40C0_0000

→ 2MB → 0x4020_0000

0010 ... 00"0"

21 byte address bus

21x "0"

valid for 2mb ✓

eg: for 2Kb mem → 11 bit addr bus
↓
there must be atleast
11 zeroes in the LSB
for the address to be valid

given starting address and multiple IPs with fixed mem reqs
arrange them such that least wastage is their in the memory block

trick fit the largest IP first

eg: SA = 0x4000_1000

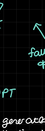
IP1 : 1Kb
IP2 : 4B
IP3 : 512B

let IP3 → 1 → 2

not ideal

correct answer : largest → smallest
so that all end addr are compatible for the next IP's starting addr

1 → 3 → 2



but what if starting address is not compatible with any IP given?

① fit the biggest to smallest IP

② fit the IP which will have closest starting address to the given SA

?

fall back option

- INTERRUPT

A signal generated to indicate to the processor that its attention is required

Specific responses to events

can be generated by the hardware processing units and peripherals and also within the processor itself

eg: internally generated / externally generated / hardware interrupt

can cause a change in the program flow

GIC: general interrupt controller

- Types

① maskable Interrupts (IRQs)

interrupts which can be ignored under certain conditions

② Non maskable Interrupts (NMI)

high priority interrupts that cannot be ignored

③ Inter-Processor Interrupts (IPI)

in a multi-processor environment, one processor might need to inform another processor(s) of an event.

memory read and write operations are slow, but we need to keep the processor as much free as possible, so we use a coprocessor unlike the DMA for this case

DMA: direct memory access

reduces load on processor
used to perform memory transfer

processor issues a memory transfer request to the DMA controller to carry out the memory operation

In this case, DMA acts as both, bus master and bus slave

↓ communicates with the mem controller

↓ sets up mem transfer by responding to reqs from processor

DMAC: DMA controller

enables data transfer from one resource to another without the involvement of the processor

it does no processing of data itself
it just transfers bytes in accordance with its configuration (done by the processor)

requires initialization by the software

useful for block storage devices
eg disk drives

allows IO devices to access memory directly without involving CPU

enables more efficient interrupt handling and higher throughput

DMA

DMA can carry out its memory operations in either burst mode or single cycle mode

In burst mode, DMAC keeps control of the memory bus until all the data buffer has been transferred and since the processor fetches instructions from the memory, the processor will freeze until transaction is completed.

In single cycle mode, DMA gives up the bus access after every transfer

Hence, single cycle mode is commonly used

- Zync Architecture