VATSAL JINDAL -58
230959166

Date : 10/4/2025

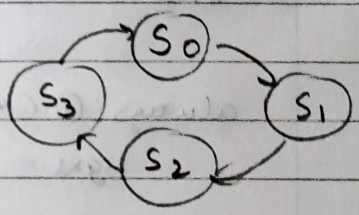
FPGA FISAC-2

Q1

Considering 4 roads : A, B, C, D
Priority order : A > B > C > D

We'll create a FSM with each state representing one road having a green light, others being red.

→ S0 : Road A is green
S_1 : Road B is green
S2 : Road C is green
S_3 : Road D is green.



⇒ Verilog code:

```
module Traffic_Light(
input clk, rst,
output reg[3:0] light);

Parameter S0 = 2'b00;
Parameter S1 = 2'b01;
Parameter S2 = 2'b10;
Parameter S3 = 2'b11;

reg [1:0] current_state, next_state;
always @ (posedge clk or posedge rst) begin
    if (reset)
        current_state <= S0;
    else
        current_state <= next_state;
    end
```

Date :

```
always @(*) begin
    case (current_state)
        S0 : next_state = S1;
        S1 : next_state = S2;
        S2 : next_state = S3;
        S3 : next_state = S0;
    default : next_state = S0;
    endcase
end

always @(*) begin
    light = 4'b0000;
    case (current_state)
        S0 : light = 4'b1000;   // A green
        S1 : light = 4'b0100;   // B green
        S2 : light = 4'b0010;   // C green
        S3 : light = 4'b0001;   // D green
    endcase
end
endmodule
```

Q2 Given specifications:

32 bit processor, opcode width : 5bits (32 possible instructions)

-> ALU operations = ADD, SUB, MUL, Divide

-> Memory operations = LOAD, STORE

-> Program memory = 512 kB Data memory = 512 kB

- Instruction set Architecture :

Date : _____

Instruction	Opcode	operation
ADD	00000	R[dst] = R[src1] + R[src2]
SUB	00001	R[dst] = R[src1] - R[src2]
MUL	00010	R[dst] = R[src1] * R[src2]
DIV	00011	R[dst] = R[src1] / R[src2]
LOAD	00100	R[dst] = mem[R[src1] + imm]
STORE	00101	R[st] = mem[R[src1] + imm]

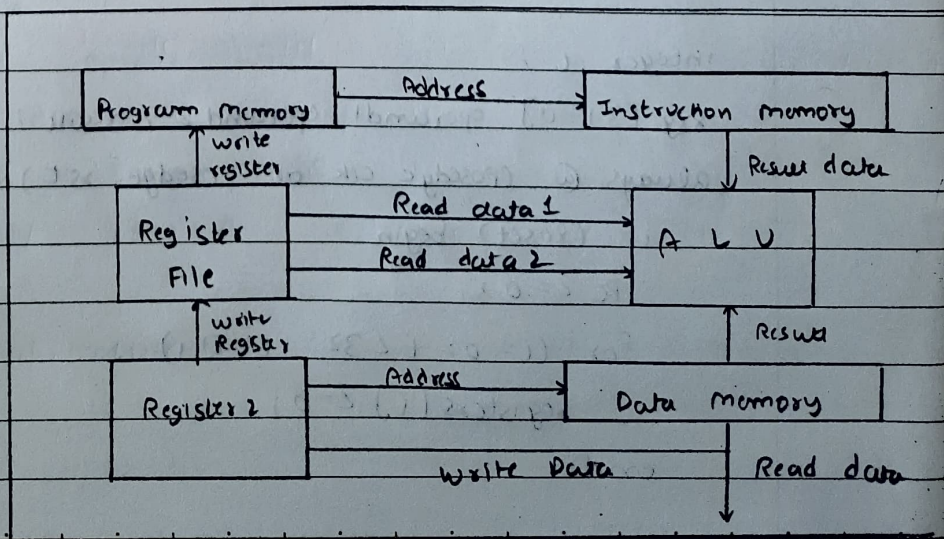
→ src 1 : source Register 1 imm = immediate value
src 2 = source register 2
dst = destination register

→ Code :

```
module RISC_Processor(
  input clk, reset);
  reg [31:0] instr_mem [0:131071] ; // 512kB = 131071 bits
  reg [31:0] data_mem [0:131071] ;
  reg [31:0] registers [0:31] ;
  reg [31:0] instr ;
  reg [4:0] opcode, dst, src1, src2;
  reg [11:0] imm ;
  reg [31:0] PC;

  integer i ;
  reg [31:0] operand1, operand2, result ;
  always @ (posedge clk or posedge rst) begin
    if (reset) begin
      PC <= 0 ;
      for (i= 0; i<32; i=i+1)
        registers[i] <= 0 ;
    end
```

Date :

```
else begin
    instr = instr_mem [pc>>2];
    opcode = instr [31:27];
    dst = instr [26:22];
    src1 = instr [21:17];
    src2 = instr [16:12];
    imm = instr [11:0];

    operand1 = registers [src1];
    operand2 = registers [src2];
case (opcode)
    5'b00000: result = operand1 + operand2;
    5'b00001: result = operand1 - operand2;
    5'b00010: result = operand1 * operand2;
    5'b00011: result = operand1 / operand2;
    5'b00100: result = data_mem [(operand1 + imm) >>2];
    5'b00101: data_mem [(operand1 + imm)>>2] = registers[dst];
default    : result = 0;
  endcase

if (opcode != 5'b00101)
    registers [dst] <= result;
    pc <= pc+1;
  end
end
endmodule
```



32-bit RISC Processor

VAIBAL JINDAL - 58
230959166

Date :

Q3

The top-down design flow in FPGA is a systematic approach that begins with high-level system specifications and progressively refines the design through hierarchical decomposition.

-> Stages of top-down design flow

(i) Specification = Define the system behaviour
(ii) Architectural Design = Identify the main modules
(iii) RTL coding = Writing a code in a HDL.
(iv) Simulation = Functional verification
(v) Synthesis = Convert into netlist.
(vi) Implementation = Place and route on FPGA.
(vii) Bitstream generation = Generate final FPGA configuration
(viii) Programming = Load the bitstream onto FPGA.
(ix) Validation = Verify on real hardware.

Q4

There are two main categories of FPGA in terms of their fabrics:

(i) SRAM based FPGAs (Xilinx, Altera) [Re-programmable]:
Uses LUTs to implement logic blocks & SRAMs to implement programmable switches.

(ii) Antifuse-based FPGAs (Actel, Lattice, Xilinx, QuickLogic, Cypress) [Permanent]:
Uses muxs to implement logic blocks & uses antifuses to implement programmable switches.

=> XILINX Board :

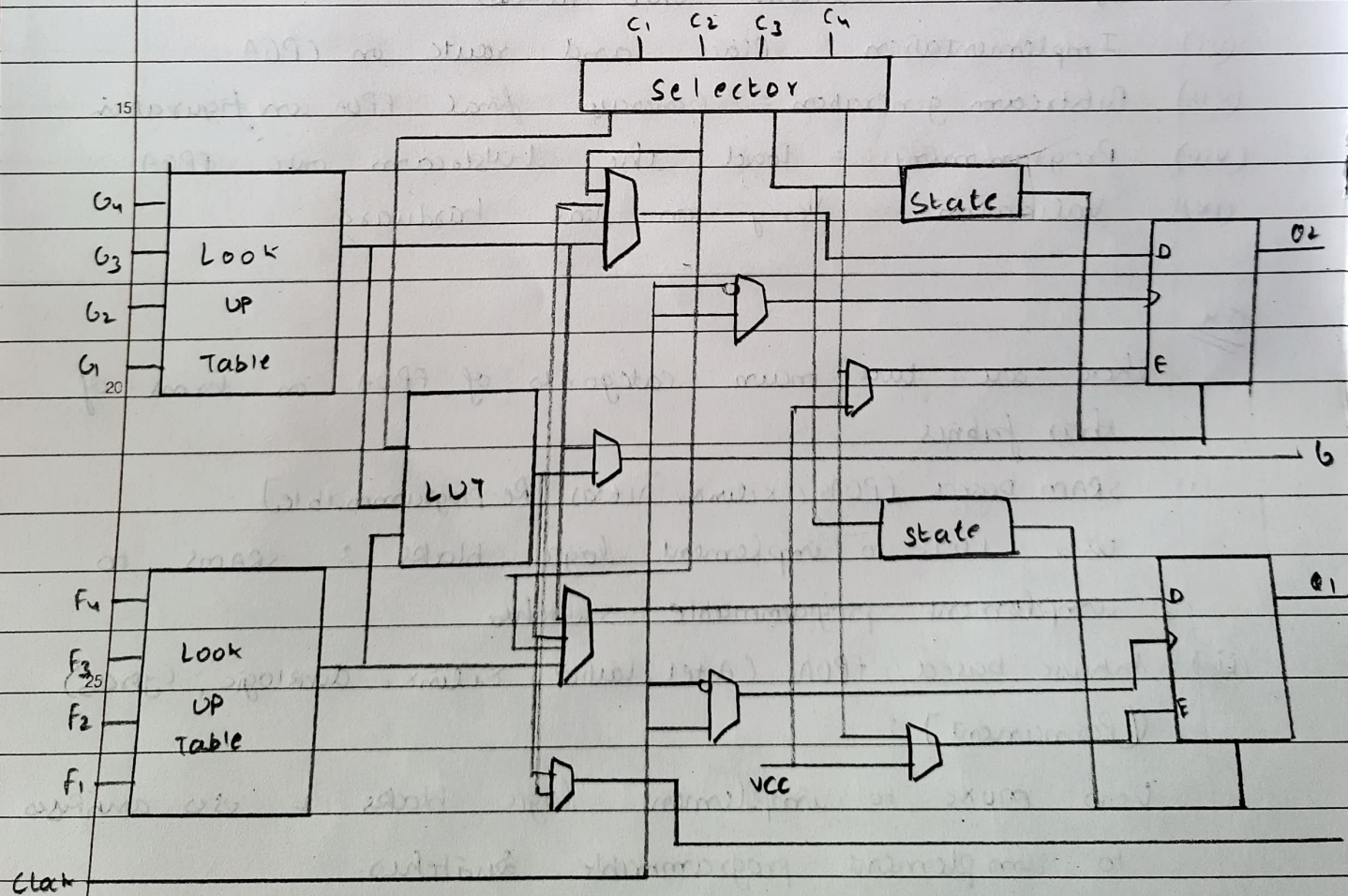
The basic structure of Xilinx FPGAs is array-based, meaning that each chip comprises a 2D array of

Date :

logic blocks that can be interconnected via horizontal and vertical routing channels.

The XC4000 features a logic block called configurable logic block (CLB) that is based on LUT. A LUT is a small one-bit wide memory array where the address lines for memory are inputs of logic block & one bit output from memory is the LUT output.

XC4000 CLB contains 3 seperate LUTs, these are 4-input LUTs that are fed by CLB inputs & the third LUT can be used in combination with the other 2.



XILINX XC4000 CLB

(i) Fine vs Coarse Grain Architecture:

Feature	Fine-Grain	Coarse-Grain
Logic Block Size	Bit-level (LUTs)	Word-level (ALU)
Flexibility	High	Lower
Speed	Slower (more routing)	Faster (less routing)
Efficiency	Good for control logic	Good for DSP
Ex.	Xilinx	GGRA, ASIC overlays

(ii) Logical mapping in FPGA

Logical mapping is the process of converting a high-level RTL design into basic FPGA resources like LUTs, Flip-flops and MUX.

⇒ Steps in logical mapping:

(i) HDL Design

(ii) Synthesis

(iii) mapping onto FPGA primitives

(iv) Results in a netlist placed on FPGA.

(iii) Partitioning for multiple FPGA systems:

When a design is too large to fit on a single FPGA, it's partitioned across multiple FPGAs. Following goals are achieved by good partitioning.

-> minimize interconnect delay

-> Balance resource usage across FPGAs

-> maximize parallelism.