EN3021 Group Project - Progress as of OCT10

210415N Navarathne D.M.G.B.
210594J Senevirathne I.U.B.
210609M Silva M.K.Y.U.N.

October 10, 2024

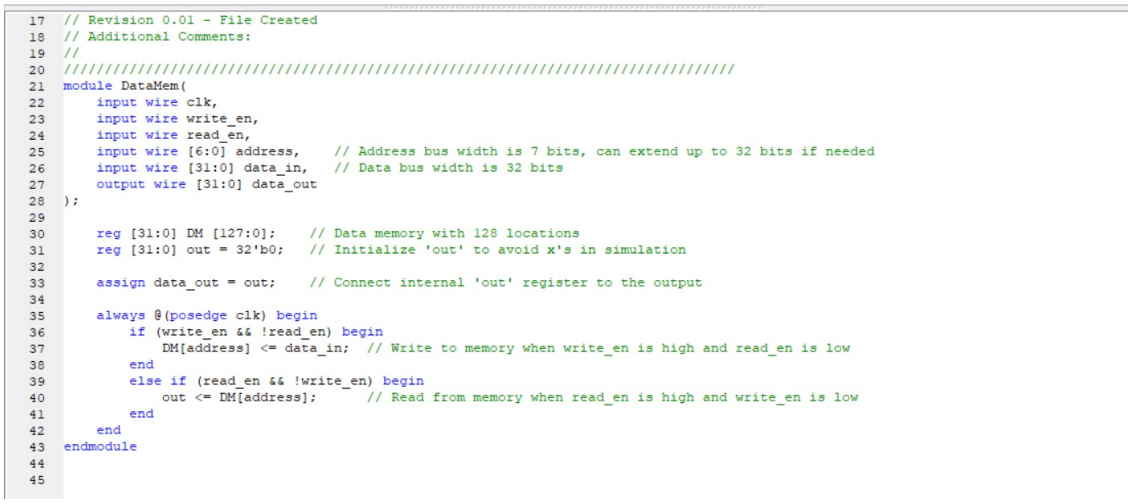
Current status: 30% Complete

Progress this week

We started the following parts of the project for the week OCT03 - OCT10:

- Started the hardware design in Verilog for the main modules of the processor. The following modules have been already completed.
  - ALU
  - Instruction memory, Data memory
  - Registry file
- Started designing the module integration of the processor

Data Memory



```
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module DataMem(
    input wire clk,
    input wire write_en,
    input wire read_en,
    input wire [6:0] address,    // Address bus width is 7 bits, can extend up to 32 bits if needed
    input wire [31:0] data_in,   // Data bus width is 32 bits
    output wire [31:0] data_out
);

    reg [31:0] DM [127:0];    // Data memory with 128 locations
    reg [31:0] out = 32'b0;   // Initialize 'out' to avoid x's in simulation

    assign data_out = out;    // Connect internal 'out' register to the output

    always @(posedge clk) begin
        if (write_en && !read_en) begin
            DM[address] <= data_in;  // Write to memory when write_en is high and read_en is low
        end
        else if (read_en && !write_en) begin
            out <= DM[address];      // Read from memory when read_en is high and write_en is low
        end
    end
endmodule


```

Figure 1: HDL design for Data memory

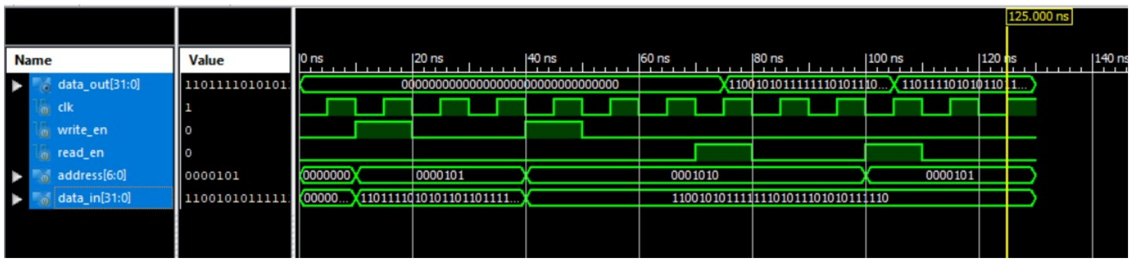


Figure 2: Testbench results for data memory

Here we have implemented the data memory such that it reads or writes the memory at the rising edge of the clock. Whether the address is being read or written is decided by the signals `read_en` and `write_en`.

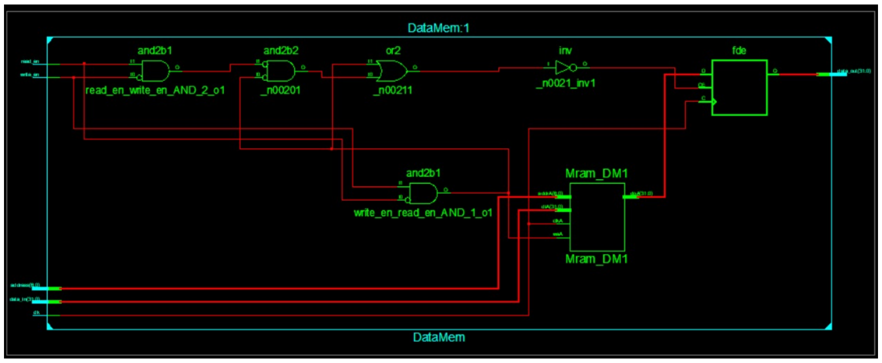
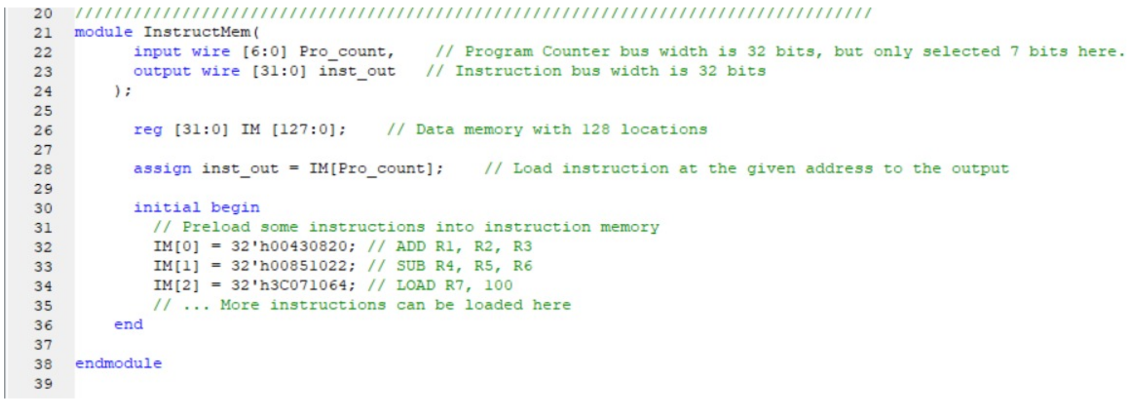


Figure 3: RTL view of the data memory

Instruction Memory



```
////////////////////////////////////////////////////////////////////////////////////
module InstructMem(
      input wire [6:0] Pro_count,     // Program Counter bus width is 32 bits, but only selected 7 bits here.
      output wire [31:0] inst_out   // Instruction bus width is 32 bits
    );

      reg [31:0] IM [127:0];    // Data memory with 128 locations

      assign inst_out = IM[Pro_count];    // Load instruction at the given address to the output

      initial begin
        // Preload some instructions into instruction memory
        IM[0] = 32'h00430820; // ADD R1, R2, R3
        IM[1] = 32'h00851022; // SUB R4, R5, R6
        IM[2] = 32'h3C071064; // LOAD R7, 100
        // ... More instructions can be loaded here
    end

endmodule
```

Figure 4: HDL design of the instruction memory

Here we have implemented the instruction memory in such that it can be read anytime without waiting for a clock input. For the testing purposes in this case we hard-coded the instructions into the HDL design.

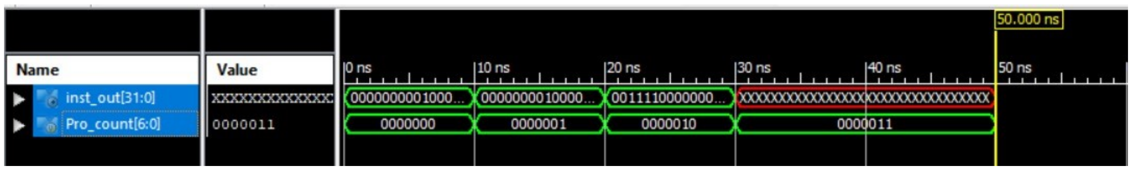


Figure 5: HDL design of the instruction memory

Registry File

Here we implemented the registry file such that it reads the pointed registers at all times, and writes into the register only at the rising edge of the clock signal. For the single cycle processor we only implemented the 32 ISA registers.

```
module Register_File(
 // inputs
    input wire[4:0] Read_reg01, // RS1 instruction value for reading register 01
    input wire[4:0] Read_reg02, // RS2 instruction value for reading register 02
    input wire[4:0] Write_reg,   // RD instruction value for writing register
    input wire[31:0] Write_data, // 32 bit data to be written to register

//outputs
    output reg [31:0] Read_data01,
    output reg [31:0] Read_data02,

// control signals
    input wire write_signal ,
    input wire clk
    //input wire rst
   );

    // Declare 32 registers, each 8 bits wide
    reg [31:0] registers [31:0];
    integer i;
     // Read logic
    always @(*) begin
        Read_data01 = registers[Read_reg01];
        Read_data02 = registers[Read_reg02];
    end
     // Write logic (triggered on clock edge)
    always @(posedge clk ) begin
        if (write_signal) begin
            // Write to the specified register
            registers[Write_reg] <= Write_data;
        end
    end

endmodule
```

Figure 6: HDL design of the registry file

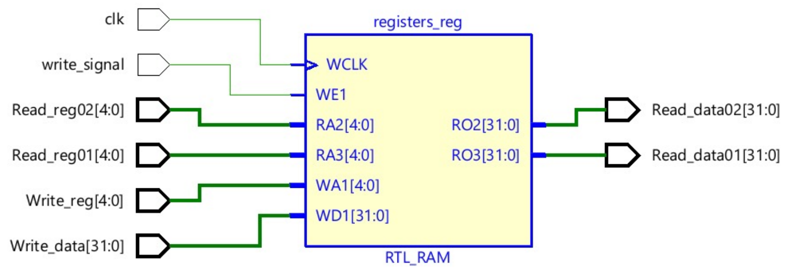
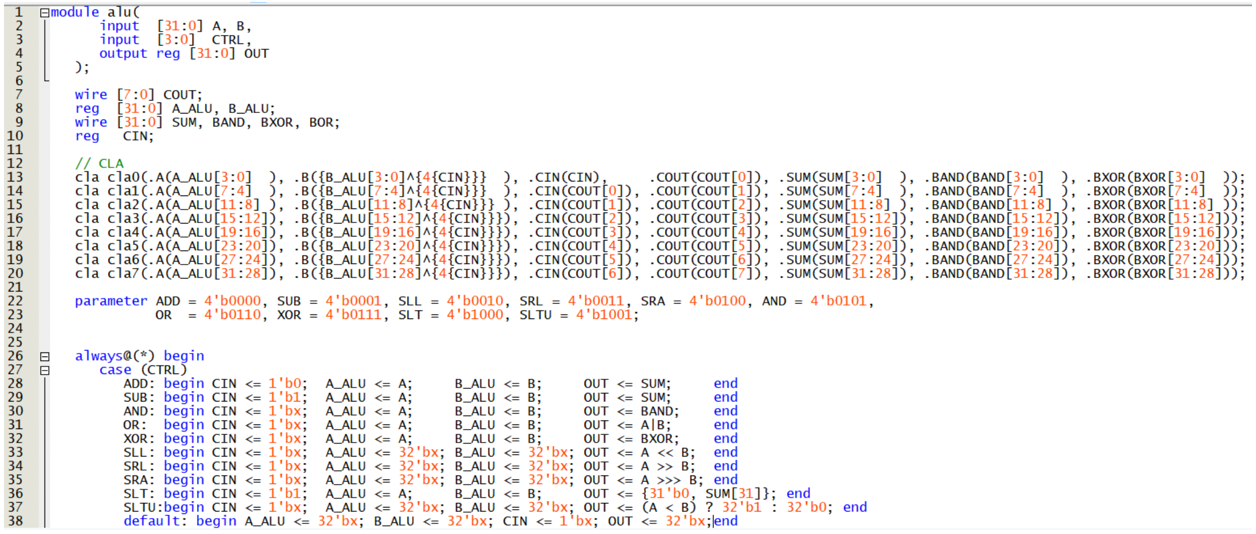


Figure 7: RTL view of the registry file

ALU

Here we designed so that the ALU has following operations:

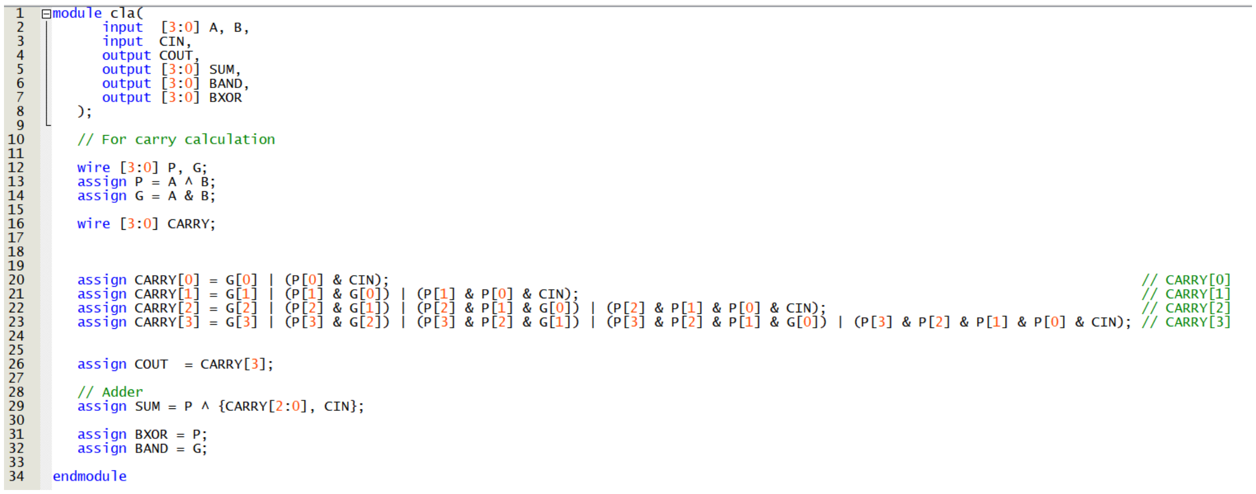
- Add
- Subtratction
- Left logical shift
- Right logical shift
- Right arithmetic shift
- Bitwise AND
- Bitwise XOR
- Bitwise OR
- Compare two signed integers
- Compare two unsigned integers



```
module alu(
        input   [31:0] A, B,
        input   [3:0]  CTRL,
        output reg [31:0] OUT
    );

    wire [7:0] COUT;
    reg  [31:0] A_ALU, B_ALU;
    wire [31:0] SUM, BAND, BXOR, BOR;
    reg   CIN;

    // CLA
    cla cla0(.A(A_ALU[3:0]  ), .B({B_ALU[3:0]^{4{CIN}}}  ), .CIN(CIN),     .COUT(COUT[0]), .SUM(SUM[3:0]  ), .BAND(BAND[3:0]  ), .BXOR(BXOR[3:0]  ));
    cla cla1(.A(A_ALU[7:4]  ), .B({B_ALU[7:4]^{4{CIN}}}  ), .CIN(COUT[0]), .COUT(COUT[1]), .SUM(SUM[7:4]  ), .BAND(BAND[7:4]  ), .BXOR(BXOR[7:4]  ));
    cla cla2(.A(A_ALU[11:8] ), .B({B_ALU[11:8]^{4{CIN}}} ), .CIN(COUT[1]), .COUT(COUT[2]), .SUM(SUM[11:8] ), .BAND(BAND[11:8] ), .BXOR(BXOR[11:8] ));
    cla cla3(.A(A_ALU[15:12]), .B({B_ALU[15:12]^{4{CIN}}}), .CIN(COUT[2]), .COUT(COUT[3]), .SUM(SUM[15:12]), .BAND(BAND[15:12]), .BXOR(BXOR[15:12]));
    cla cla4(.A(A_ALU[19:16]), .B({B_ALU[19:16]^{4{CIN}}}), .CIN(COUT[3]), .COUT(COUT[4]), .SUM(SUM[19:16]), .BAND(BAND[19:16]), .BXOR(BXOR[19:16]));
    cla cla5(.A(A_ALU[23:20]), .B({B_ALU[23:20]^{4{CIN}}}), .CIN(COUT[4]), .COUT(COUT[5]), .SUM(SUM[23:20]), .BAND(BAND[23:20]), .BXOR(BXOR[23:20]));
    cla cla6(.A(A_ALU[27:24]), .B({B_ALU[27:24]^{4{CIN}}}), .CIN(COUT[5]), .COUT(COUT[6]), .SUM(SUM[27:24]), .BAND(BAND[27:24]), .BXOR(BXOR[27:24]));
    cla cla7(.A(A_ALU[31:28]), .B({B_ALU[31:28]^{4{CIN}}}), .CIN(COUT[6]), .COUT(COUT[7]), .SUM(SUM[31:28]), .BAND(BAND[31:28]), .BXOR(BXOR[31:28]));

    parameter ADD = 4'b0000, SUB = 4'b0001, SLL = 4'b0010, SRL = 4'b0011, SRA = 4'b0100, AND = 4'b0101,
              OR  = 4'b0110, XOR = 4'b0111, SLT = 4'b1000, SLTU = 4'b1001;


    always@(*) begin
       case (CTRL)
          ADD: begin CIN <= 1'b0;  A_ALU <= A;      B_ALU <= B;      OUT <= SUM;      end
          SUB: begin CIN <= 1'b1;  A_ALU <= A;      B_ALU <= B;      OUT <= SUM;      end
          AND: begin CIN <= 1'bx;  A_ALU <= A;      B_ALU <= B;      OUT <= BAND;     end
          OR:  begin CIN <= 1'bx;  A_ALU <= A;      B_ALU <= B;      OUT <= A|B;      end
          XOR: begin CIN <= 1'bx;  A_ALU <= A;      B_ALU <= B;      OUT <= BXOR;     end
          SLL: begin CIN <= 1'bx;  A_ALU <= 32'bx; B_ALU <= 32'bx; OUT <= A << B;   end
          SRL: begin CIN <= 1'bx;  A_ALU <= 32'bx; B_ALU <= 32'bx; OUT <= A >> B;   end
          SRA: begin CIN <= 1'bx;  A_ALU <= 32'bx; B_ALU <= 32'bx; OUT <= A >>> B; end
          SLT: begin CIN <= 1'b1;  A_ALU <= A;      B_ALU <= B;      OUT <= {31'b0, SUM[31]}; end
          SLTU:begin CIN <= 1'bx;  A_ALU <= 32'bx; B_ALU <= 32'bx; OUT <= (A < B) ? 32'b1 : 32'b0; end
          default: begin A_ALU <= 32'bx; B_ALU <= 32'bx; CIN <= 1'bx; OUT <= 32'bx;end
```

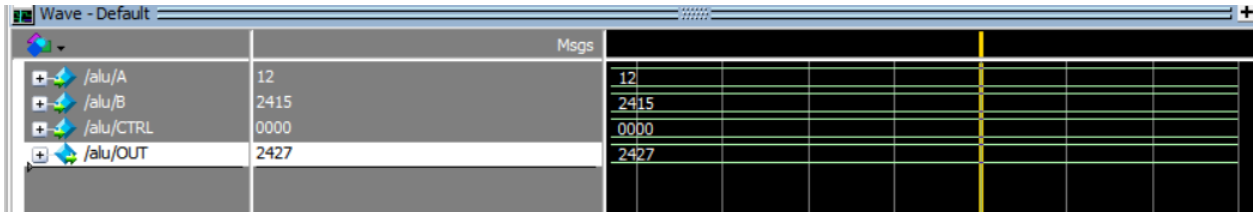
Figure 8: HDL design of the ALU

In this design we have used 675 logic elements, and no registers. For ADD, SUB and SLT, we have used a carry look ahead adder. For the AND and XOR operations, we have used the same XOR and AND gates used to calculate the carry generate and carry propagate.



```verilog
module cla(
        input   [3:0] A, B,
        input   CIN,
        output  COUT,
        output  [3:0] SUM,
        output  [3:0] BAND,
        output  [3:0] BXOR
    );

    // For carry calculation

    wire [3:0] P, G;
    assign P = A ^ B;
    assign G = A & B;

    wire [3:0] CARRY;



    assign CARRY[0] = G[0] | (P[0] & CIN);                                                                                       // CARRY[0]
    assign CARRY[1] = G[1] | (P[1] & G[0]) | (P[1] & P[0] & CIN);                                                                // CARRY[1]
    assign CARRY[2] = G[2] | (P[2] & G[1]) | (P[2] & P[1] & G[0]) | (P[2] & P[1] & P[0] & CIN);                                  // CARRY[2]
    assign CARRY[3] = G[3] | (P[3] & G[2]) | (P[3] & P[2] & G[1]) | (P[3] & P[2] & P[1] & G[0]) | (P[3] & P[2] & P[1] & P[0] & CIN); // CARRY[3]


    assign COUT  = CARRY[3];

    // Adder
    assign SUM = P ^ {CARRY[2:0], CIN};

    assign BXOR = P;
    assign BAND = G;

endmodule
```

Figure 9: HDL design for the CLA

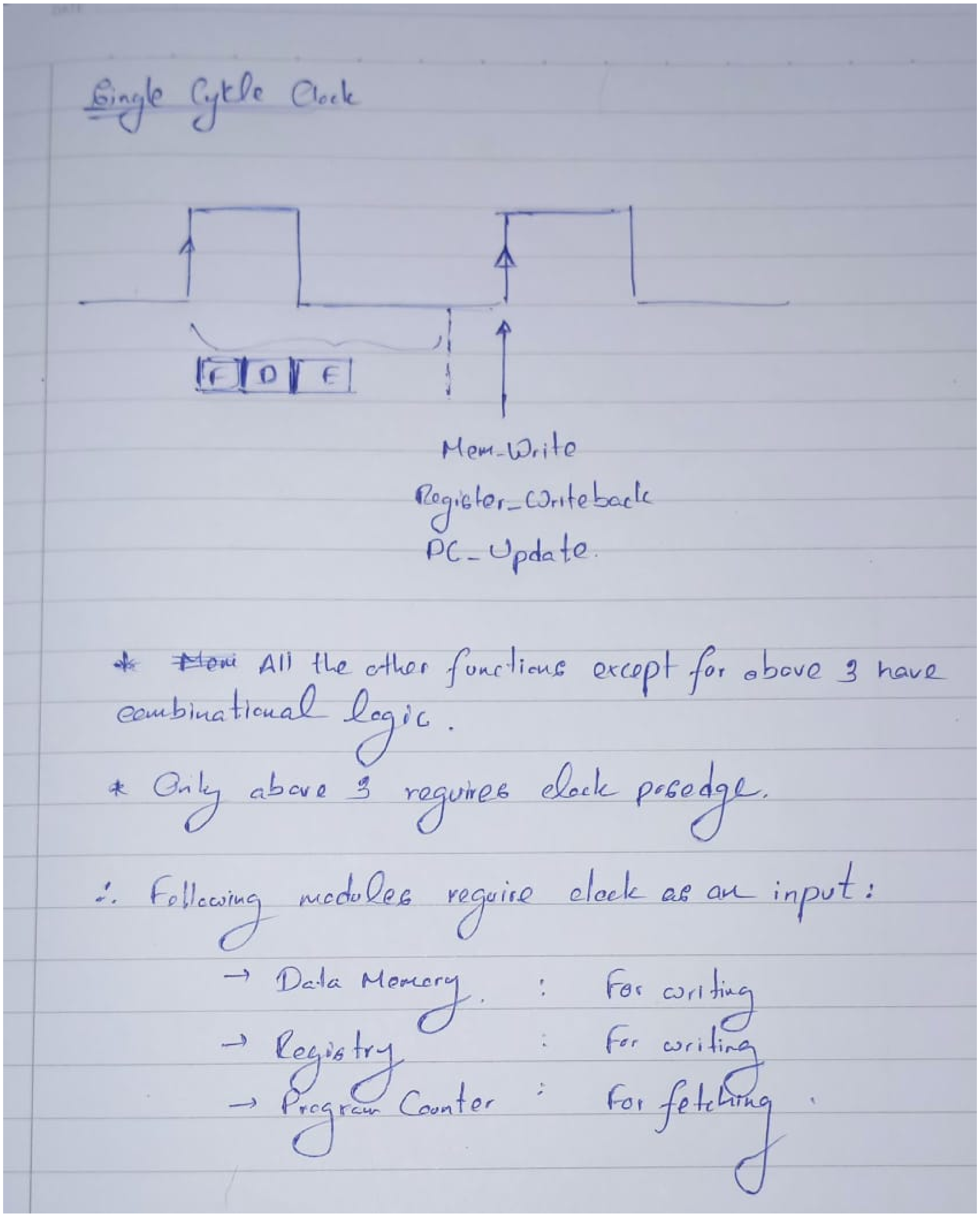


```
Wave - Default
                Msgs
/alu/A          12        12
/alu/B          2415      2415
/alu/CTRL       0000      0000
/alu/OUT        2427      2427
```

Figure 10: ModelSim Simulation

Members' Contribution

- ALU - Silva M.K.Y.U.N
- Instruction memory, Data memory - Nawarathna D.M.G.B
- Registry file - Senavirathne I.U.B.



Single Cycle Clock

Mem_Write
Register_Writeback
PC_Update.

* ~~Mem~~ All the other functions except for above 3 have combinational logic.

* Only above 3 requires clock posedge.

∴ Following modules require clock as an input:

→ Data Memory. : For writing
→ Registry : For writing
→ Program Counter : For fetching .

Figure 11: Control signals sequencing