NATIONAL UNIVERSITY OF SCIENCES & TECHNOLOGY

Digital System Design (EE-421)

Assignment # 1
Building a 4-bit Microprocessor using Verilog HDL

Submission Details

Submitted to:	Dr. Rehan Ahmed
Submitted by:	Muhammad Umer
Class:	BEE-12C
Semester:	6th
Dated:	7/3/2023
CMS ID:	345834

NUST
School of Electrical Engineering
and Computer Science (SEECS)

1 Table of Contents

2 Table of Figures

3 Introduction

In the field of digital electronics, microprocessors are the heart of every computing system, providing efficient and reliable processing capabilities. In this assignment, we present the implementation of a 4-bit number crunching machine designed using Verilog HDL. The purpose of this machine is to perform arithmetic and logical operations on 4-bit data and to execute instructions to control the processing flow.

The microprocessor is designed using a modular approach, where each part of the machine is defined as a separate submodule with varying levels of abstractions. These submodules include an instruction decoder, an arithmetic logic unit, registers, data selectors, and register selectors. The instruction decoder module, which is a ROM-PC interface, interprets the machine language instructions and sends signals to other modules to perform the appropriate operation. The control unit module coordinates the flow of data between the different submodules.

Following block diagram shows a high-level abstraction of the implemented design:

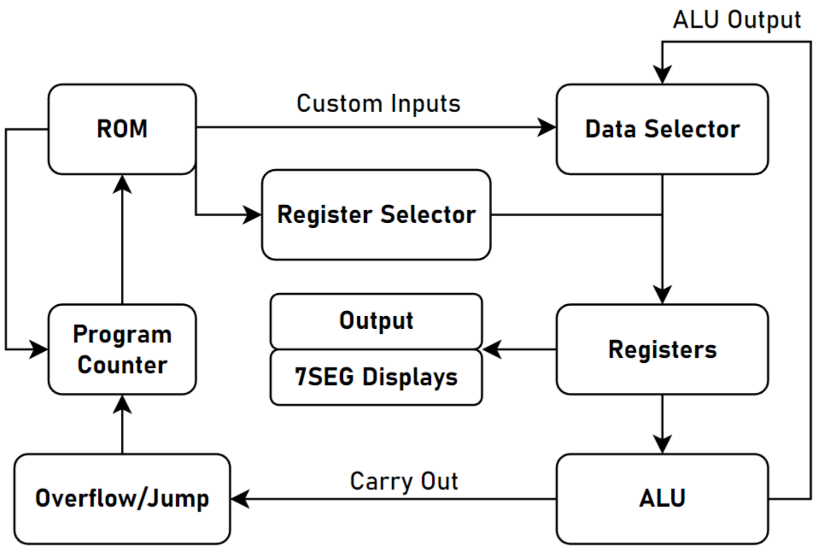


Figure 1: Design

4 Modules

In this section, we describe the Figure 1 modules with a brief description along with its Verilog code.

4.1 ROM & Program Counter

The module that directs and drives the rest of the microprocessor is the ROM-PC interface. The user burns a specific code in HEX format to the ROM and the program counter fetches the stored instructions through its output line (address). On each clock cycle, the counter increments by one, thereby also incrementing the address and the ROM outputs the next instruction in the appropriate memory location. The counter also supports custom loading capabilities allowing the interface to 'jump' to certain instructions.

```
module instruction_memory (
    input  [3:0] addr,
    output [7:0] out
);

   reg [7:0] rom[0:15];

   initial begin
      $readmemb("D:/IntelFPGA/projects/assignment_1/fibseries.txt", rom);
   end

   assign out = rom[addr];

endmodule
```

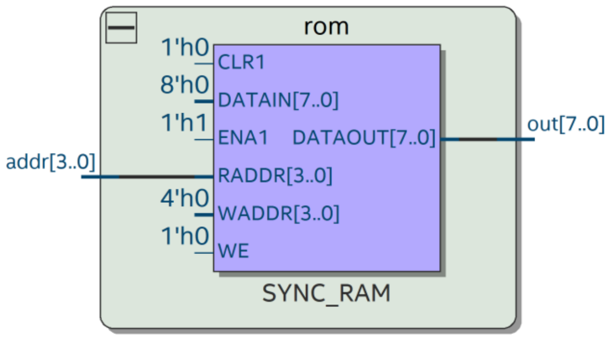


Figure 2: Instruction Memory

```
module counter (
    input [3:0] in,
    input load,
    clk,
    reset,
    output reg [3:0] count
);

   always @(posedge clk or negedge reset) begin

      if (reset == 0) begin
         count <= 4'd0;
      end else if (load == 1) begin
         count <= in;
      end else begin
         count <= count + 4'd1;
      end
   end

endmodule

module Load_Logic (
    input  c_out,
    jump,
    carry_jump,
    output load
);
   assign load = jump | (c_out & carry_jump);

endmodule
```

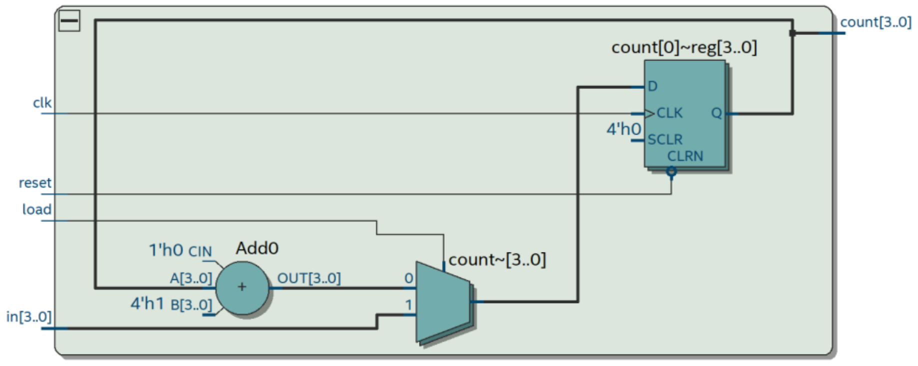


Figure 3: Counter

4.2 Data Selector

Data selector is used to either update the storage units of our microprocessor with the outputs of the logic unit or it can also load custom values to the register. The 4-bit input lines to the multiplexer are the custom inputs and the ALU outputs, amongst which the selector chooses what to store depending on the instruction stored in the ROM.

```
module reg_mux (
    input [3:0] a,
    input [3:0] b,
    input s,
    output reg [3:0] m
);

   always @(*) begin
      if (s != 1) begin
         m = a;
      end else begin
         m = b;
      end
   end

endmodule
```

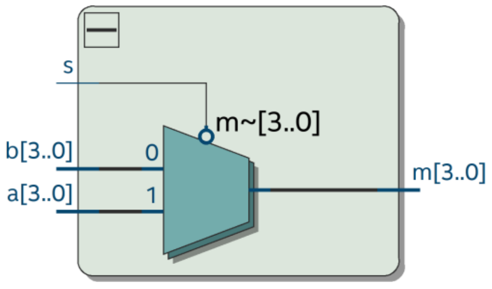


Figure 4: Data Selector

4.3 Registers & Register Selector

Registers, each 4-bit wide, are the storage units of the proposed microprocessor. Bits oncoming the output of the data selector are stored into the register enabled by the register selector. The register selector is a simple 2-to-4-line demultiplexer and is used to enable any of the three implemented registers, or not enable any at all.

```
module reg_demux (
    input [1:0] d,
    output reg [3:0] out
);

   always @(*) begin
      case (d)
         2'b00:   out = 4'b0001;
         2'b01:   out = 4'b0010;
         2'b10:   out = 4'b0100;
         default: out = 4'b1000;
      endcase
   end

endmodule
```

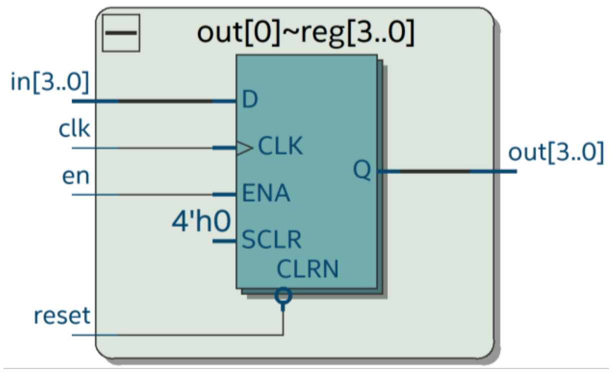


Figure 5: 4-Bit Wide Register

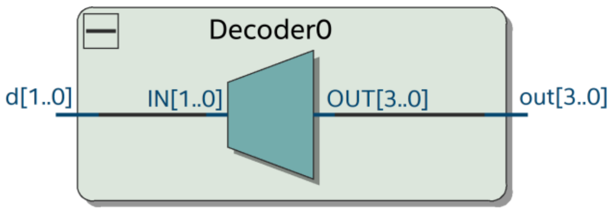


Figure 6: Register Selector

4.4 ALU

ALU is the central logical unit of the microprocessor; it is used to perform all the arithmetic calculations that are required to implement algorithms, etc. It performs either addition or subtraction based on the input *s*, which adds the 2's complement of input *b* when it is HIGH.

```
module alu (
    input [3:0] a,
    input [3:0] b,
    input s,
    output reg [3:0] sum,
    output reg c_out
);
   always @(*) begin
      if (s == 1) begin
         {c_out, sum} = a - b;
      end else begin
         {c_out, sum} = a + b;
      end
   end

endmodule
```

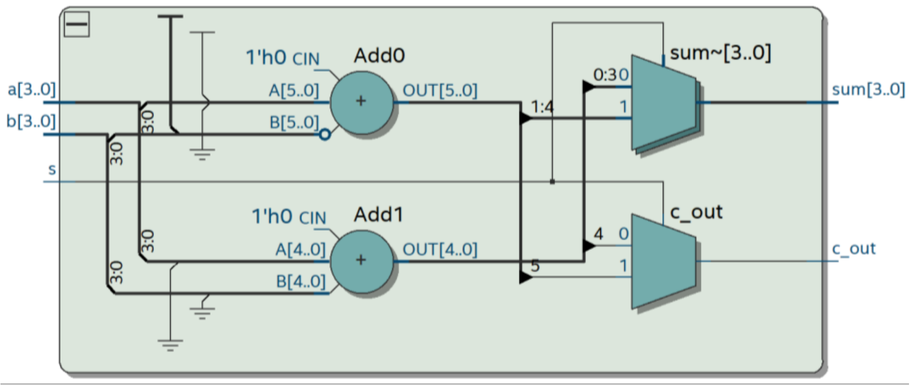


Figure 7: Arithmetic Logic Unit

4.5 7-SEG Decoder

To display the 4-bit output, which can go up to 1111 or 15, we need two separate 7-SEG displays. Using simple Boolean logic and a comparator, we implement the 4-bit 7-SEG decoder that checks if the input number > 9 and maps the higher bits back onto the lower bits, but with the HEX1 display ON. Essentially, 10, 11, 12, … are mapped onto 0, 1, 2, … with the leftmost 7-SEG display showing the digit one.

```
module decoder_7seg (  // 4-bit to 7-segment decoder
    input  [3:0] v,
    output [6:0] hex
);
   assign hex[0] = (v[2] & ~v[1] & ~v[0]) | (~v[3] & ~v[2] & ~v[1] & v[0]);
   assign hex[1] = (v[2] & ~v[1] & v[0]) | (v[2] & v[1] & ~v[0]);
   assign hex[2] = (~v[2] & v[1] & ~v[0]);
   assign hex[3] = (v[2] & ~v[1] & ~v[0]) | (v[2] & v[1] & v[0]) | (~v[3] & ~v[2] & ~v[1] &
                    v[0]);
   assign hex[4] = v[0] | (v[2] & ~v[1]);
   assign hex[5] = (~v[2] & v[1]) | (v[1] & v[0]) | (~v[3] & ~v[2] & v[0]);
   assign hex[6] = (~v[3] & ~v[2] & ~v[1]) | (v[2] & v[1] & v[0]);

endmodule

module comparator (  // > 9 comparator
    input [3:0] v,
    output z
);
   assign z = v[3] & (v[2] | v[1]);
endmodule

module mapper (
    input  [3:0] v,
    output [3:0] a
);

   assign a[3] = 0;
   assign a[2] = v[3] & v[2] & v[1];
   assign a[1] = v[3] & v[2] & ~v[1];
   assign a[0] = (v[3] & v[0]) & (v[2] | v[1]);

endmodule
```

```
module hex_output (
    input  [3:0] v,
    output [6:0] HEX0,
    output [6:0] HEX1
);

   wire [3:0] a, m;
   wire [3:0] z;
   assign z[3:1] = 3'b000;

   mapper bit_map (.v(v), .a(a));
   comparator comp (.v(v), .z(z[0]));
   reg_mux ltg_mux (.a(v), .b(a), .s(z[0]), .m(m));
   decoder_7seg digit0 (.v  (m), .hex(HEX0));
   decoder_7seg digit1 (.v  (z), .hex(HEX1));

endmodule
```

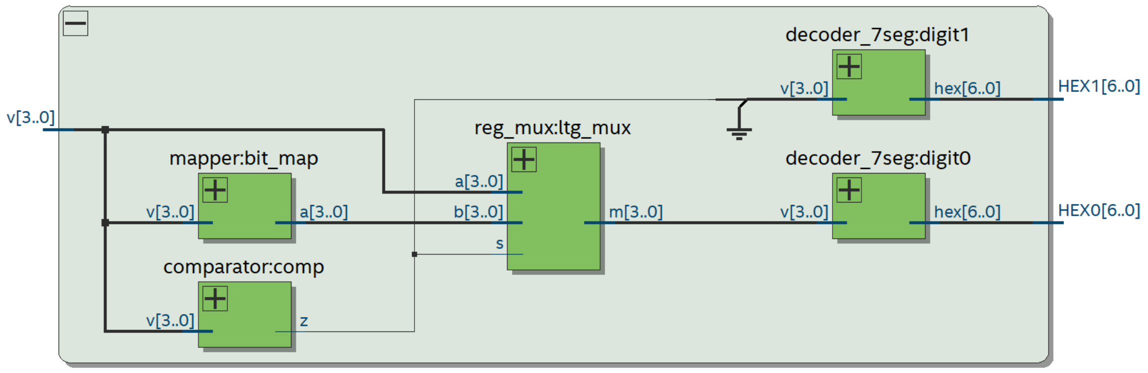


Figure 8: 4-Bit 7SEG Decoder

4.6 Custom Clock

As the DE-10 SoC board has an internal clock of 50 MHz, using it for providing the clock to the entire circuit would change the output too fast for a human observer. Hence, we implement a delay of 0.2 seconds using a clock divider.

```
module clock_divider (
    input clk_in,  // 50 MHz input clock
    output reg clk_out
);

   reg [23:0] count = 0;  // Initialize count to zero

   always @(posedge clk_in) begin
      count <= count + 1;

      if (count == 24'd4_999_999) begin
         count   <= 0;
         clk_out <= ~clk_out;  // Invert the output clock
      end
   end
endmodule
```

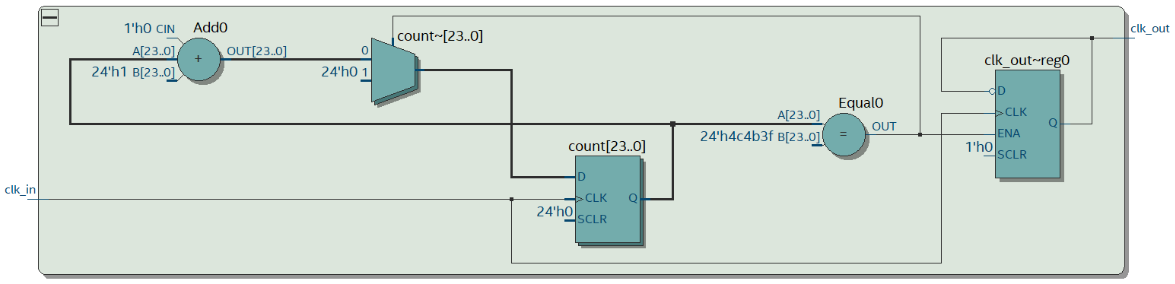


Figure 9: Clock Divider

5 Fibonacci Series (Algorithm & Hardware Demonstration)

To test the functionality of the number crunching machine, we used the Fibonacci series as a benchmark. The Fibonacci series is a sequence of numbers where each number is the sum of the two preceding numbers. Following figures elaborate on the instruction flow of the said series:

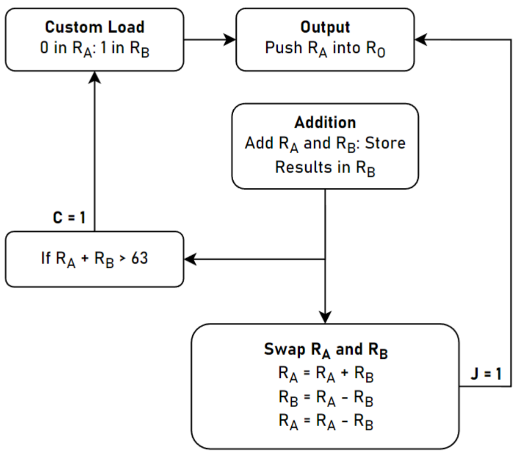
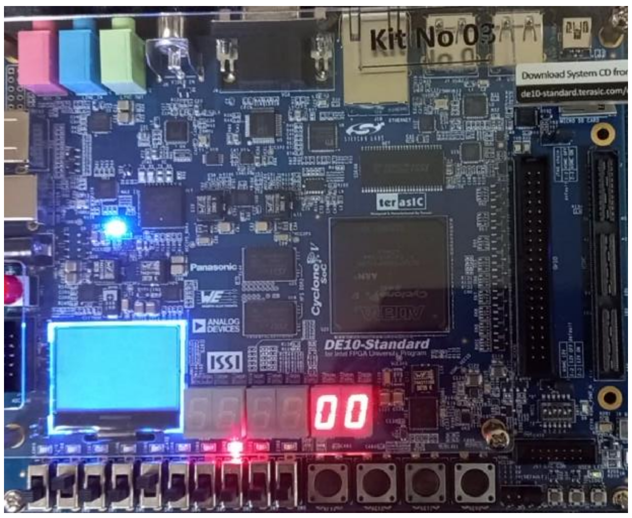


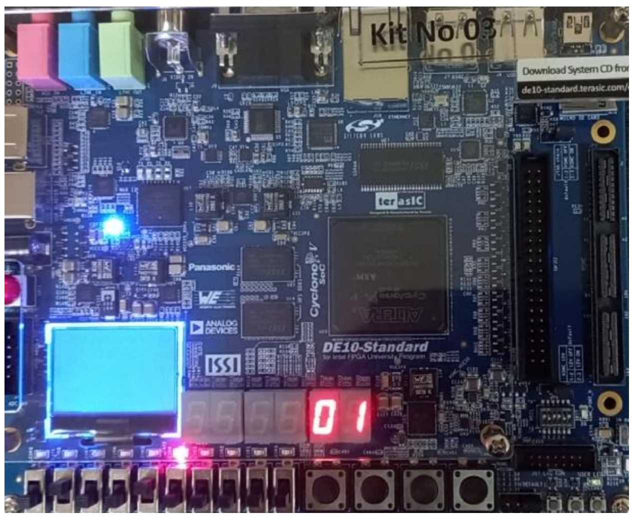
Figure 10: Fibonacci Series: Flowchart

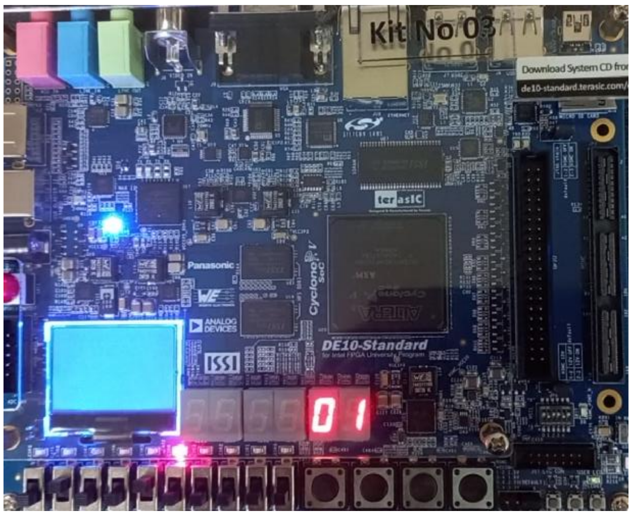
00001000	load zero to A
00011001	load 1 to B
00100000	push A to output
00010000	Add A to B
01110000	If A+B produce carry jump to start
00000000	Swap A and B
00010100	Swap A and B
00000100	Swap A and B
10110010	jump to 3rd instruction

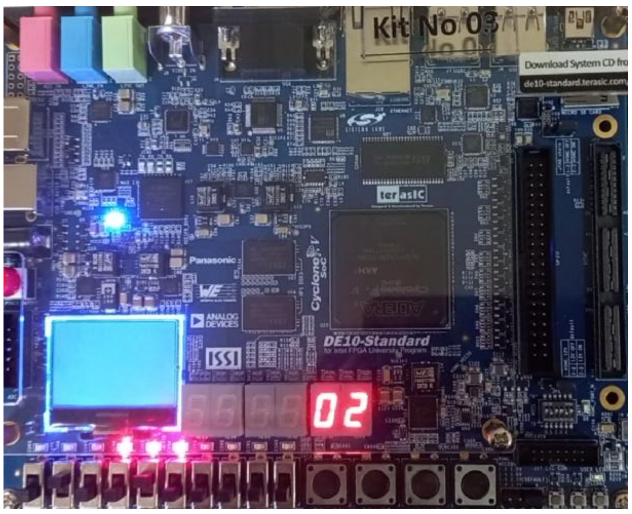
Figure 11: Fibonacci Series: Instructions

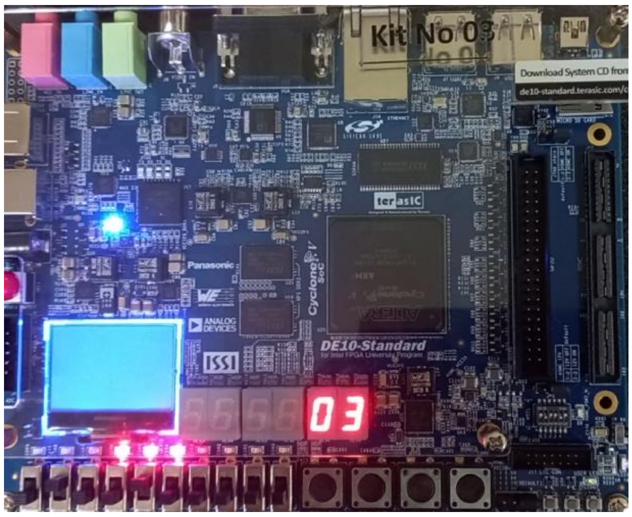
Following pictures display the Fibonacci series tested on the DE-10 FPGA board:

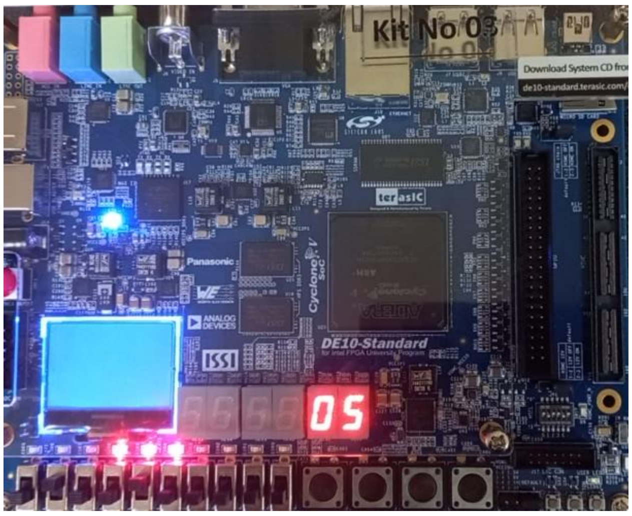


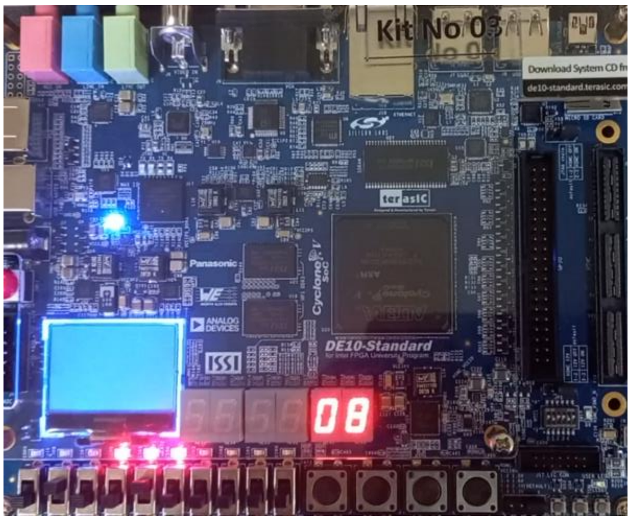


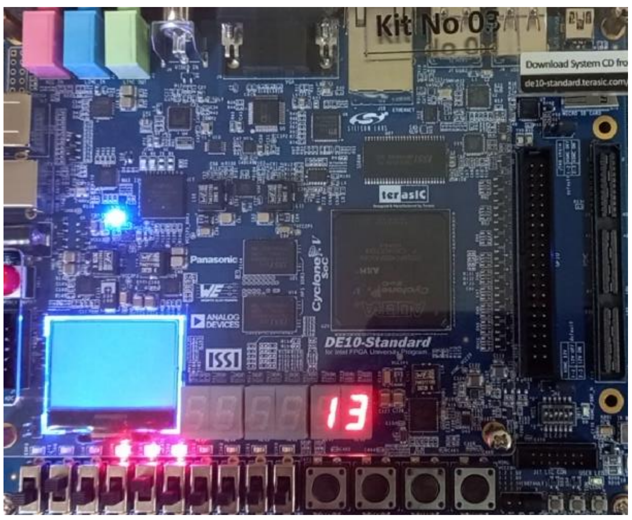

Kit No 03
01


Kit No 03
02


Kit No 03
03


Kit No 03
05


Kit No 03
08


Kit No 03
13

6 Conclusion

In this report, we presented the design and implementation of a 4-bit number crunching machine using Verilog HDL. The microprocessor was designed using a modular approach, with each submodule defined at varying levels of abstraction. We tested the functionality of the microprocessor by implementing the Fibonacci series, which verified that the machine performed accurate and consistent computations. The design of the number crunching machine demonstrates the importance of modular design and testing in creating efficient and reliable digital systems. The use of Verilog HDL allowed for the creation of a highly customizable and flexible microprocessor, which can be tailored to meet specific requirements and constraints.

7 Appendix

The code that interconnects all the modules explained in Section 4 is given below, followed by the resulting RTL Viewer snapshot.

```
module standard (
    //   input clk,
    input CLOCK_50,
    //   reset,
    input [3:0] KEY,
    output [9:0] LEDR,
    output [6:0] HEX0,
    output [6:0] HEX1
);

   // Creating clock divider
   wire clk;
   clock_divider unitCLK (
       .clk_in (CLOCK_50),
       .clk_out(clk)
   );

   // Creating reset key
   wire reset;
   assign reset = KEY[0];

   // Creating intermediary wires
   wire load, s_reg, s, c_out, alu_c, carry_jump, jump;
   wire [3:0] customIn, m, rA, rB, rO, en, aluOut, addr;
   wire [1:0] d;
   wire [7:0] instruction;

   // Assigning useful keywords to instruction set
   assign jump = instruction[7];
   assign carry_jump = instruction[6];
   assign d[1:0] = instruction[5:4];
   assign s_reg = instruction[3];
   assign s = instruction[2] | customIn[2];
   assign customIn[2:0] = instruction[2:0], customIn[3] = 1'b0;

   // Instruction Memory
   instruction_memory unitROM (
       .addr(addr),
       .out (instruction)
   );

   // Program Counter
   counter unitPC (
       .in(customIn),
       .load(load),
       .clk(clk),
       .reset(reset),
       .count(addr)
   );

   // Mux - Register Interface
   reg_mux unitM (
       .a(aluOut),
       .b(customIn),
       .s(s_reg),
       .m(m)
   );
   register_4b regA (
       .in(m),
       .en(en[0]),
       .clk(clk),
       .reset(reset),
       .out(rA)
   );
   register_4b regB (
       .in(m),
       .en(en[1]),
       .clk(clk),
       .reset(reset),
       .out(rB)
   );
   register_4b regO (
       .in(rA),
       .en(en[2]),
       .clk(clk),
       .reset(reset),
       .out(rO)
   );

   // ALU
   alu unitALU (
       .a(rA),
       .b(rB),
       .sum(aluOut),
       .s(s),
       .c_out(alu_c)
   );

   // Decoder
   reg_demux unitD (
       .d  (d),
       .out(en)
   );

   // Instruction Helpers
   load_logic unitL (
       .c_out(c_out),
       .jump(jump),
       .carry_jump(carry_jump),
       .load(load)
   );
   dFlipFlop unitFF (
       .in(alu_c),
       .clk(clk),
       .reset(reset),
       .out(c_out)
   );

   // 7 SEG Output
   hex_output unitDISP (
       .v(r0),
       .HEX0(HEX0),
       .HEX1(HEX1)
   );

   assign LEDR[7:0] = instruction;

endmodule

// Testbench
module standard_tb ();

   reg CLK;
   reg [3:0] KEY;
   wire [9:0] LEDR;
   wire [6:0] HEX0;
   wire [6:0] HEX1;

   standard dut (
       .CLOCK_50(CLK),
       .KEY(KEY),
       .LEDR(LEDR),
       .HEX0(HEX0),
       .HEX1(HEX1)
   );

   initial begin
      CLK = 0;
      forever #5 CLK = ~CLK;
   end

   initial begin
      KEY = 0;
      #10;
      KEY = 1;
   end

endmodule
```

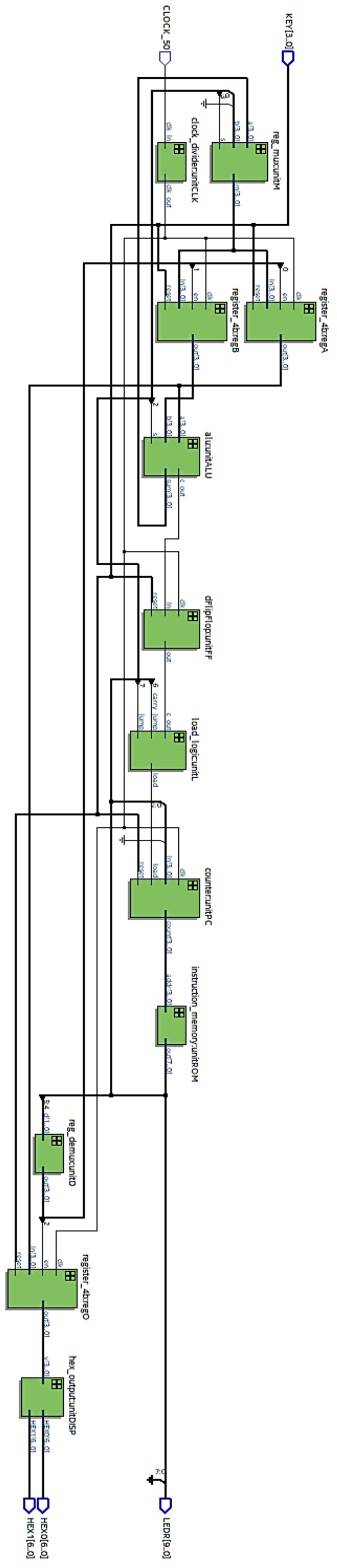


Figure 12: Complete Design