

**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 |

**Table of Contents**

[2 Table of Figures 1](#_Toc128933703)

[3 Introduction 2](#_Toc128933704)

[4 Modules 2](#_Toc128933705)

[4.1 ROM & Program Counter 2](#_Toc128933706)

[4.2 Data Selector 4](#_Toc128933707)

[4.3 Registers & Register Selector 4](#_Toc128933708)

[4.4 ALU 5](#_Toc128933709)

[4.5 7-SEG Decoder 6](#_Toc128933710)

[4.6 Custom Clock 7](#_Toc128933711)

[5 Fibonacci Series (Algorithm & Pictorial Representation) 8](#_Toc128933712)

[6 Conclusion 10](#_Toc128933713)

[7 Appendix 10](#_Toc128933714)

# Table of Figures

[Figure 1: Design 2](#_Toc128933715)

[Figure 2: Instruction Memory 3](#_Toc128933716)

[Figure 3: Counter 4](#_Toc128933717)

[Figure 4: Data Selector 4](#_Toc128933718)

[Figure 5: 4-Bit Wide Register 5](#_Toc128933719)

[Figure 6: Register Selector 5](#_Toc128933720)

[Figure 7: Arithmetic Logic Unit 6](#_Toc128933721)

[Figure 8: 4-Bit 7SEG Decoder 7](#_Toc128933722)

[Figure 9: Clock Divider 8](#_Toc128933723)

[Figure 10: Fibonacci Series: Flowchart 8](#_Toc128933724)

[Figure 11: Fibonacci Series: Instructions 8](#_Toc128933725)

[Figure 12: Complete Design 13](#_Toc128933726)

# 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:

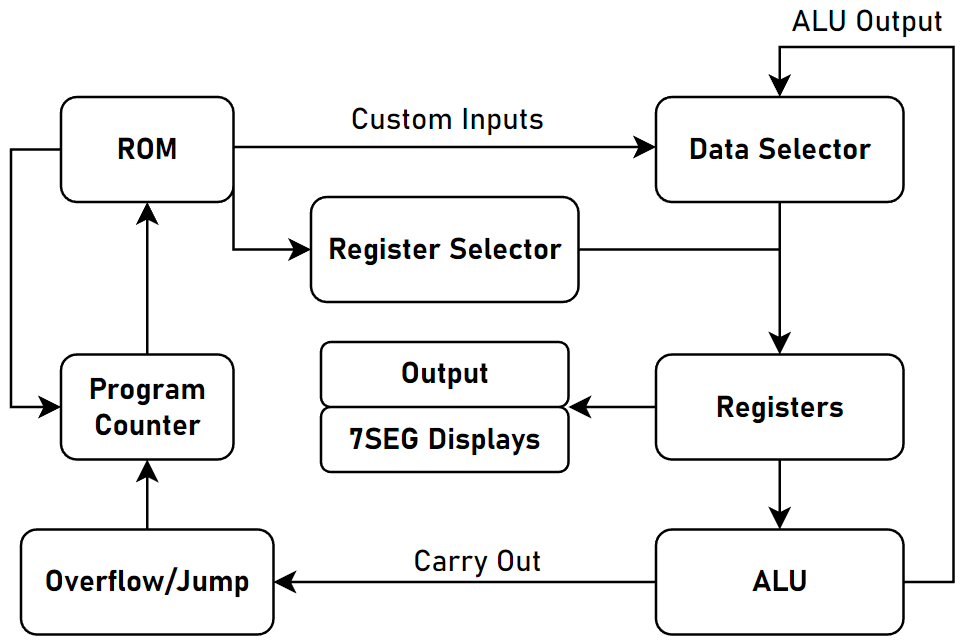


Figure : Design

# Modules

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

## 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

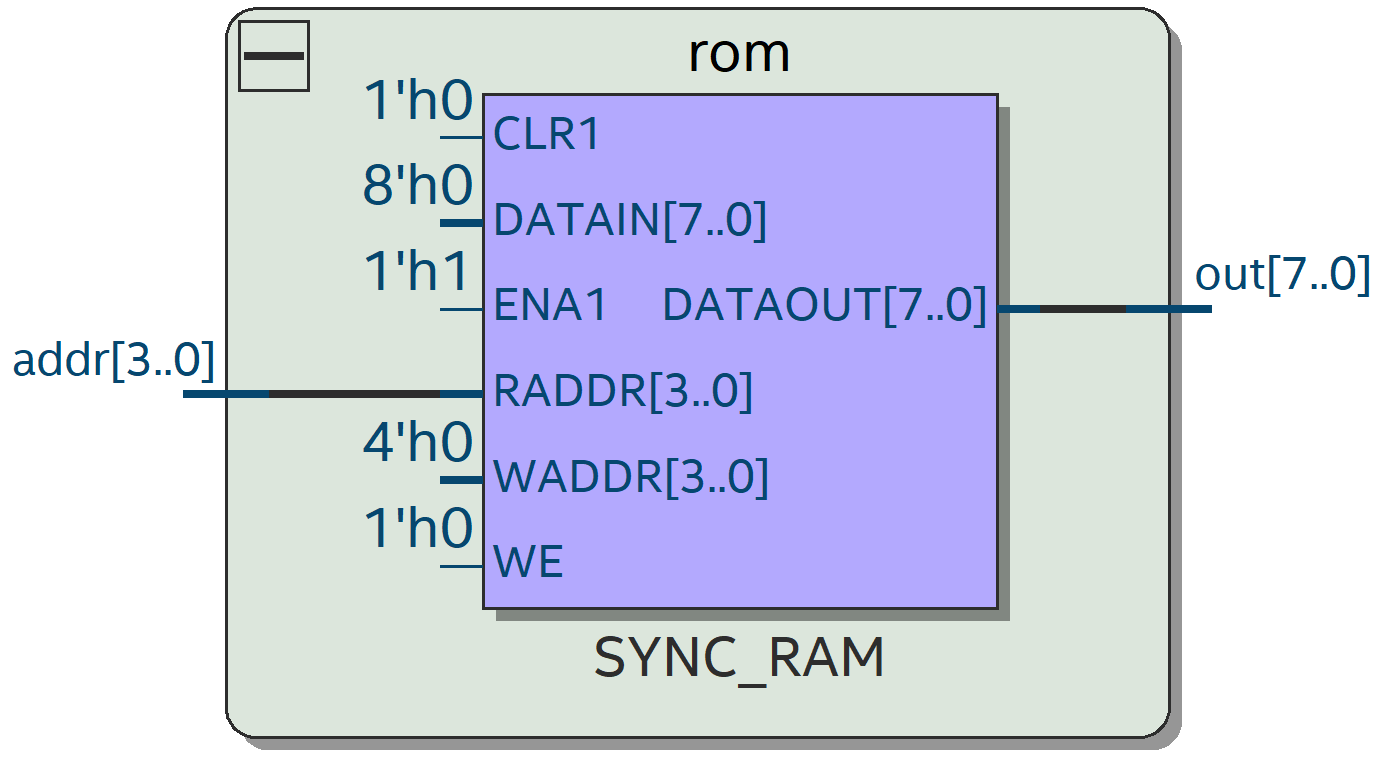


Figure : Instruction Memory

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

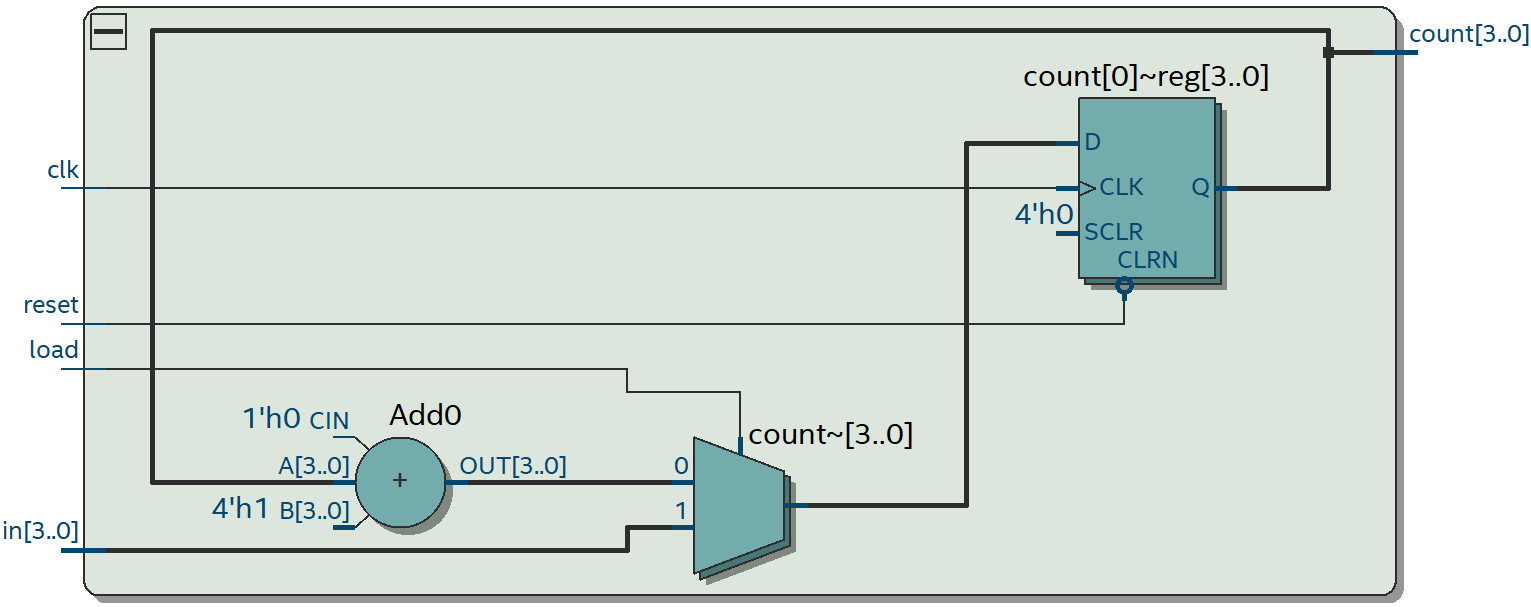


Figure : Counter

## 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

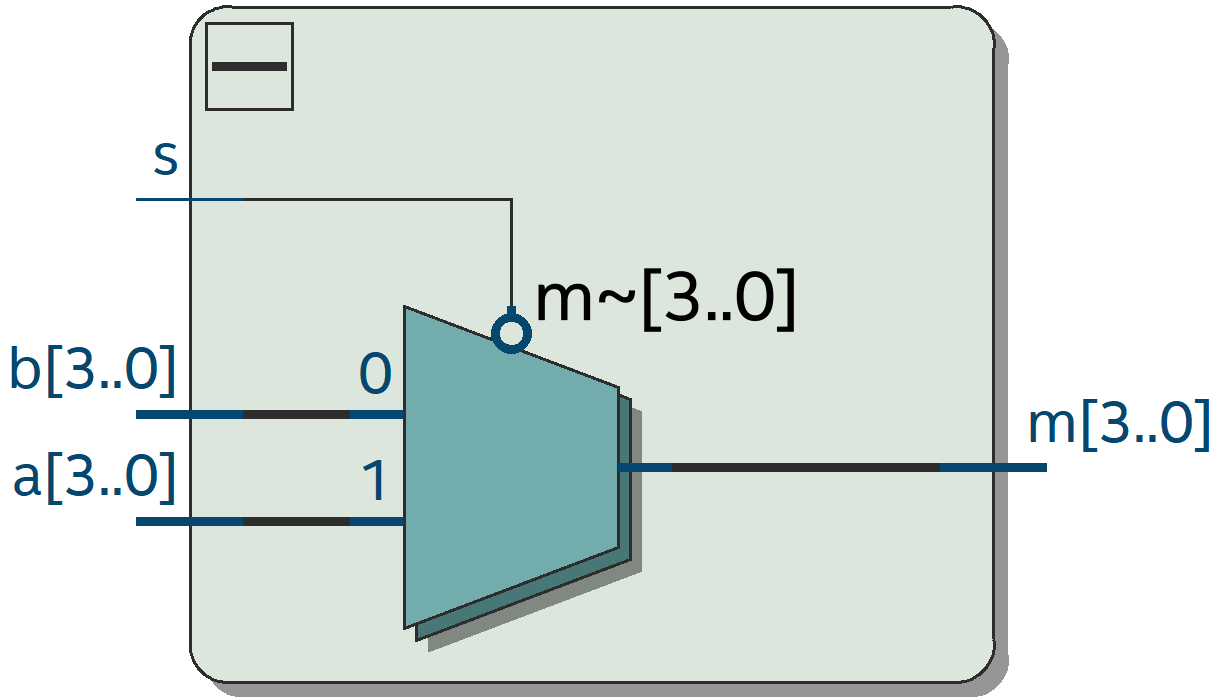


Figure : Data Selector

## 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 : 4-Bit Wide Register | Figure : Register Selector |

## 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 , which adds the 2’s complement of input 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

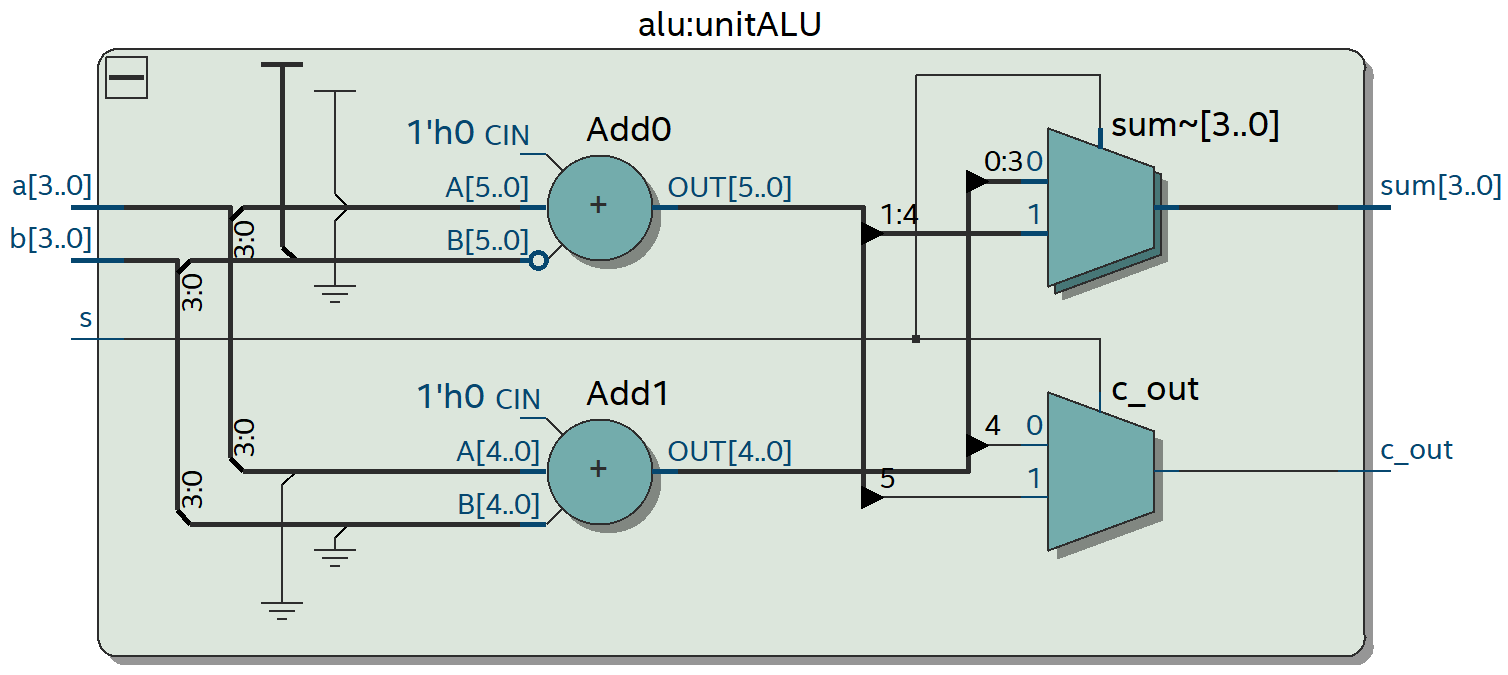


Figure : Arithmetic Logic Unit

## 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

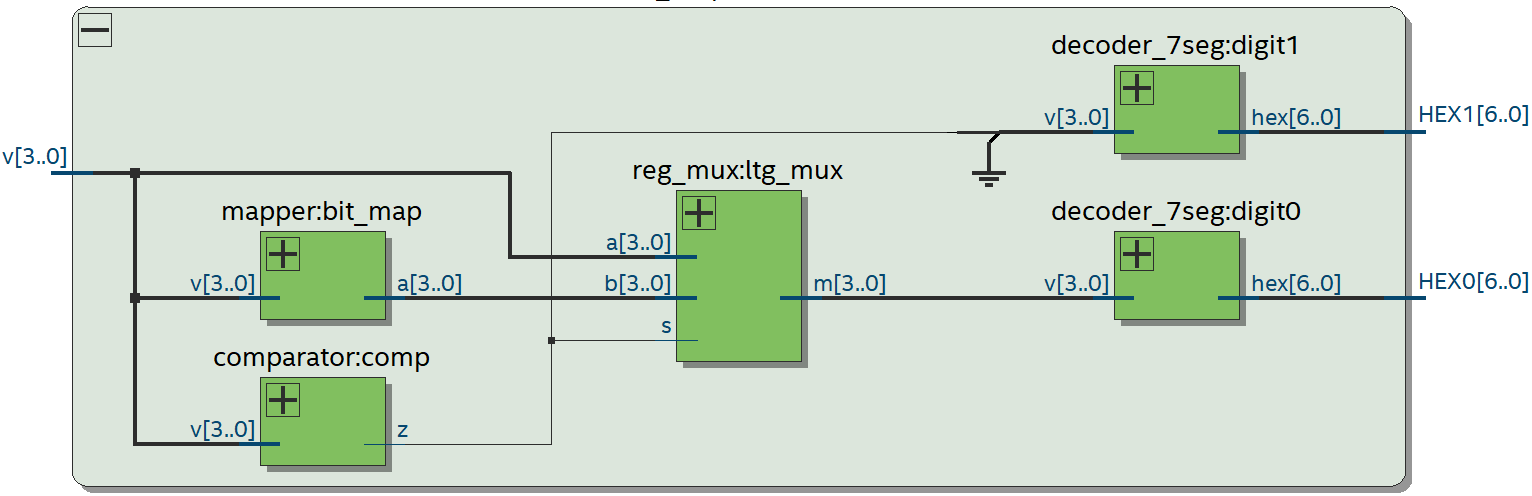


Figure : 4-Bit 7SEG Decoder

## 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

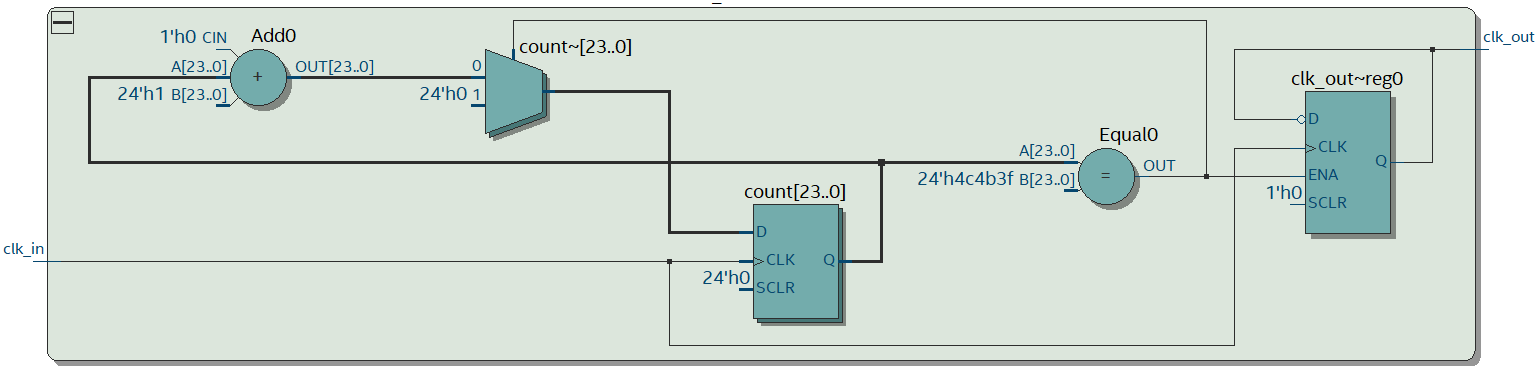


Figure : Clock Divider

# 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 : Fibonacci Series: Flowchart | Figure : Fibonacci Series: Instructions |

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

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# 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.

# 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(rO),

       .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

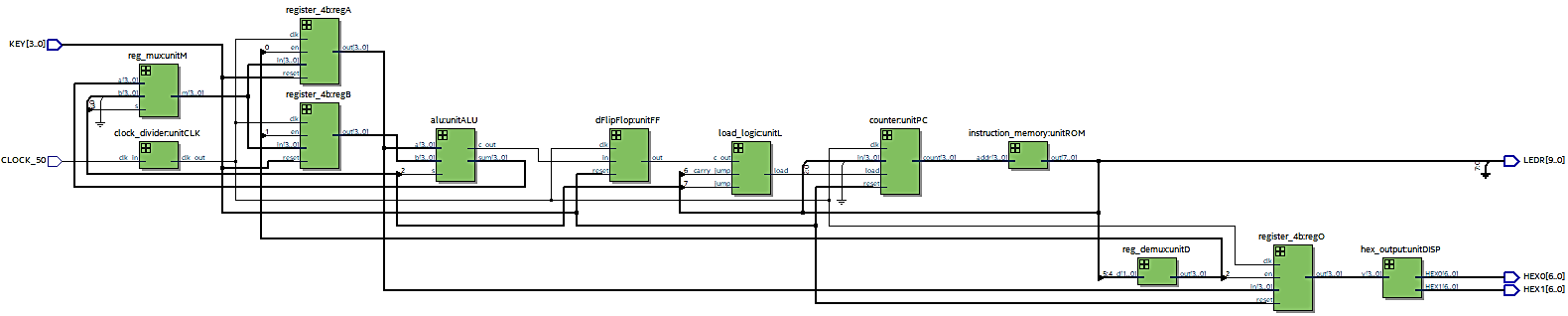


Figure : Complete Design