**Department of Electrical Engineering and   
Computer Science**

**Faculty Member:** Dr. Rehan Ahmed  **Dated:** 3/05/2023

**Semester:**  6th **Section:** BEE 12C

**EE-421:** **Digital System Design**

**Lab 10: Adders, Subtractors, and Multipliers**

Group Members

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | **PLO4-CLO3** | | **PLO5 -CLO4** | **PLO8 -CLO5** | **PLO9 -CLO6** |
| **Name** | **Reg. No** | **Viva / Quiz / Lab Performance** | **Analysis of data in Lab Report** | **Modern Tool Usage** | **Ethics and Safety** | **Individual and Teamwork** |
|  |  | **5 Marks** | **5 Marks** | **5 Marks** | **5 Marks** | **5 Marks** |
| Danial Ahmad | 331388 |  |  |  |  |  |
| Muhammad Umer | 345834 |  |  |  |  |  |
| Tariq Umar | 334943 |  |  |  |  |  |

**Table of Contents**

[2 Adders, Subtractors, and Multipliers 3](#_Toc134212822)

[2.1 Objectives 3](#_Toc134212823)

[2.2 Introduction 3](#_Toc134212824)

[2.3 Software 3](#_Toc134212825)

[3 Lab Procedure 4](#_Toc134212826)

[3.1 Part I 4](#_Toc134212827)

[3.2 Part II 6](#_Toc134212828)

[3.3 Part III 7](#_Toc134212829)

[4 Conclusion 11](#_Toc134212830)

# Adders, Subtractors, and Multipliers

## Objectives

The purpose of this exercise is to examine arithmetic circuits that add, subtract, and multiply numbers. Each circuit will be described in Verilog and implemented on an Intel FPGA DE10-Lite, DE0-CV, DE1-SoC, or DE2- 115 board.

## Introduction

The purpose of this exercise is to examine arithmetic circuits that add, subtract, and multiply numbers. Each circuit will be described in Verilog and implemented on an Intel FPGA DE10-Standard. Arithmetic circuits are the building blocks of many digital systems. They are used to perform basic operations on numbers, such as addition, subtraction, multiplication, and division. Arithmetic circuits can be implemented in a variety of ways, including using logic gates, flip-flops, and registers. In this exercise, we will implement three different arithmetic circuits on an Intel FPGA DE10-Standard. The first circuit will be an arithmetic adder, which is a simple and straightforward way to add two numbers. The second circuit will be the same as circuit one, albeit with the ability to subtract the numbers as well. The third circuit will be a multiplier, which is used to multiply two numbers together.

## Software

Quartus Prime is a comprehensive design software developed by Intel Corporation for designing digital circuits using Field-Programmable Gate Arrays (FPGAs). It is a leading software platform in the field of digital design, offering a range of advanced tools and features that enable users to easily create, debug, and verify complex digital circuits. With Quartus Prime, users can benefit from a streamlined design flow that facilitates the creation of digital circuits from concept to implementation. It provides an intuitive graphical user interface that allows users to easily design, test, and debug their circuits. Additionally, Quartus Prime supports a variety of popular programming languages, making it a versatile platform for digital designers of all levels.

# Lab Procedure

## Part I

1. Create a new Quartus project. Write Verilog code that describes the circuit in Figure 2.
2. Connect input A to switches SW7−0, use KEY0 as an active-low asynchronous reset, and use KEY1 as a manual clock input. The sum from the adder should be displayed on the red lights LEDR7−0, the registered carry signal should be displayed on LEDR8, and the registered overflow signal should be displayed on LEDR9. Show the registered values of A and S as hexadecimal numbers on the 7-segment displays HEX3−2 and HEX1 − 0.
3. Make the necessary pin assignments needed to implement the circuit on your DE-series board and compile the circuit.
4. Use timing simulation to verify the correct operation of the circuit. Once the simulation works properly, download the circuit onto your DE-series board and test it by using different values of A. Be sure to check that the overflow output works correctly.

module *task\_1* (

*input*  [3:0] KEY,

*input*  [9:0] SW,

*output* [9:0] LEDR,

*output* [6:0] HEX0,

*output* [6:0] HEX1,

*output* [6:0] HEX2,

*output* [6:0] HEX3

);

    wire [7:0] A = SW[7:0];

    wire clk = KEY[1];

    wire c\_out;

    wire [7:0] S;

    reg reg\_overflow;

    reg reg\_carry;

    reg [7:0] reg\_A, reg\_S;

    assign {c\_out, S} = reg\_A + reg\_S;

    always @(posedge clk) begin

        if (S != 0) begin

            reg\_S <= S;

        end else begin

            reg\_S <= reg\_S;

        end

        if ((reg\_A[7] == 0 && reg\_S[7] == 0 && S[7] == 1) |

            (reg\_A[7] == 1 && reg\_S[7] == 1 && S[7] == 0)) begin

            reg\_overflow <= 1'b1;

        end else begin

            reg\_overflow <= 1'b0;

        end

        reg\_A <= A;

        reg\_carry <= c\_out;

        reg\_S <= S;

    end

*decoder* digit0 (.in (reg\_S[3:0]), .HEX(HEX0));

*decoder* digit1 (.in (reg\_S[7:4]), .HEX(HEX1));

*decoder* digit2 (.in (A[3:0]), .HEX(HEX2));

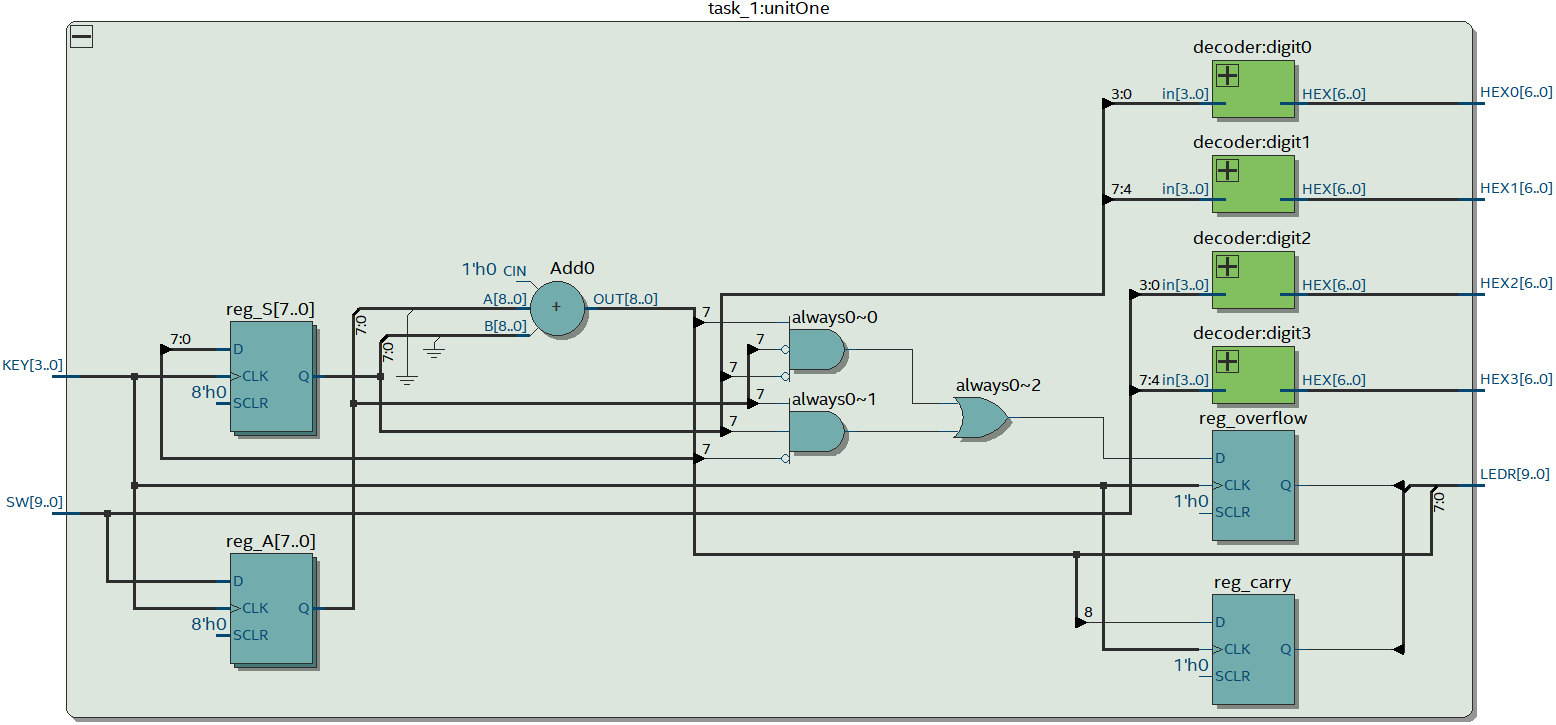
*decoder* digit3 (.in (A[7:4]), .HEX(HEX3));

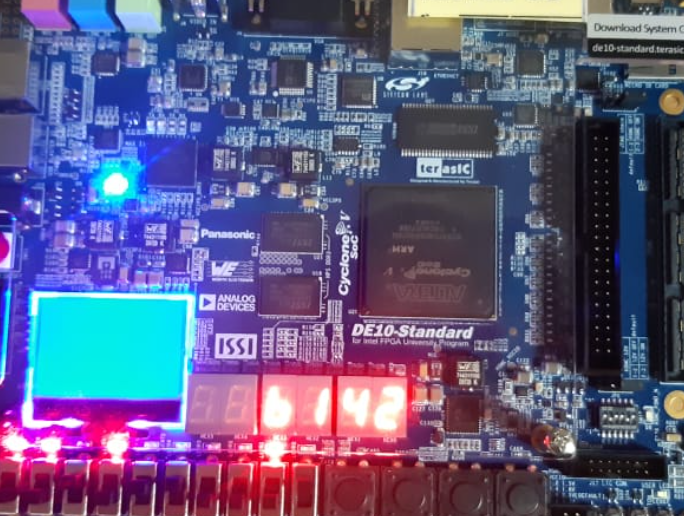
    assign LEDR[9]   = reg\_overflow;

    assign LEDR[8]   = reg\_carry;

    assign LEDR[7:0] = S;

endmodule





## Part II

Extend the circuit from Part I to be able to both add and subtract numbers. To do so, introduce an add\_sub input to your circuit. When add\_sub is 1, your circuit should subtract A from S, and when add\_sub is 0 your circuit should add A to S as in Part I.

module *task\_2* (

*input*  [3:0] KEY,

*input*  [9:0] SW,

*output* [9:0] LEDR,

*output* [6:0] HEX0,

*output* [6:0] HEX1,

*output* [6:0] HEX2,

*output* [6:0] HEX3

);

    initial reg\_S = 0;

    wire add\_sub = SW[8];

    wire [7:0] A = SW[7:0];

    wire clk = KEY[1];

    wire c\_out;

    wire [7:0] S;

    reg reg\_overflow;

    reg reg\_carry;

    reg [7:0] reg\_A, reg\_S;

    assign {c\_out, S} = (add\_sub)? reg\_S - reg\_A : reg\_S + reg\_A;

    always @(posedge clk) begin

        if (S != 0) begin

            reg\_S <= S;

        end else begin

            reg\_S <= reg\_S;

        end

        if ((reg\_A[7] == 0 && reg\_S[7] == 0 && S[7] == 1) |

            (reg\_A[7] == 1 && reg\_S[7] == 1 && S[7] == 0)) begin

            reg\_overflow <= 1'b1;

        end else begin

            reg\_overflow <= 1'b0;

        end

        reg\_A <= A;

        reg\_carry <= c\_out;

        reg\_S <= S;

    end

*decoder* digit0 (

        .in (reg\_S[3:0]),

        .HEX(HEX0)

    );

*decoder* digit1 (

        .in (reg\_S[7:4]),

        .HEX(HEX1)

    );

*decoder* digit2 (

        .in (A[3:0]),

        .HEX(HEX2)

    );

*decoder* digit3 (

        .in (A[7:4]),

        .HEX(HEX3)

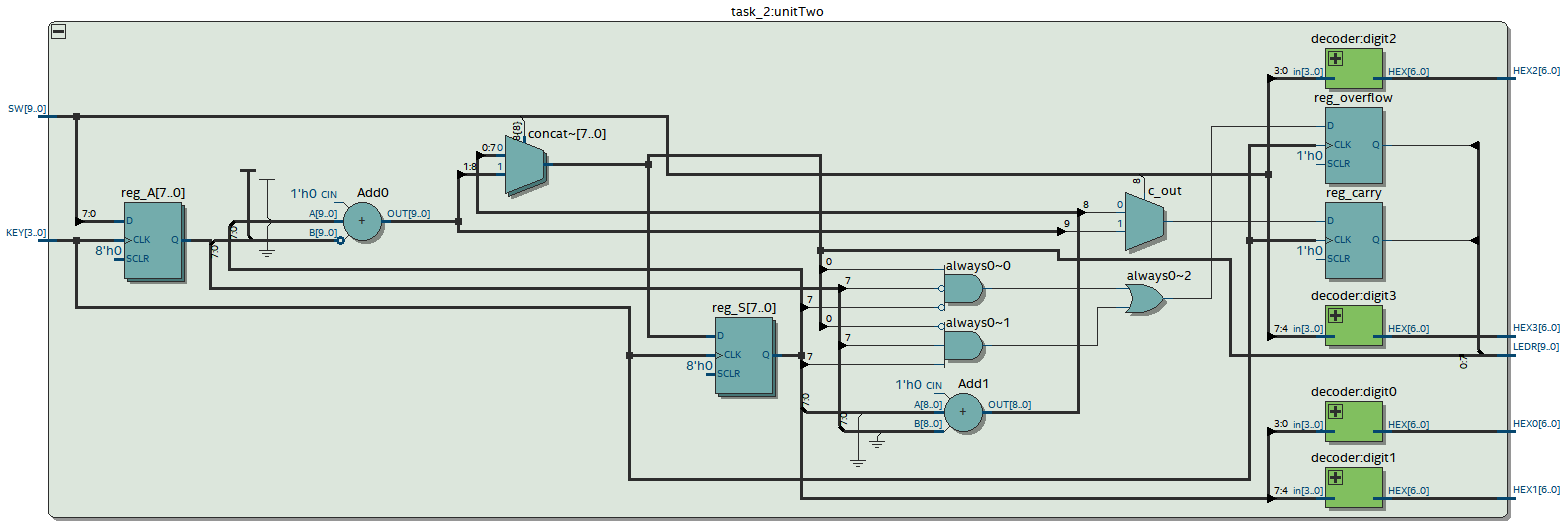
    );

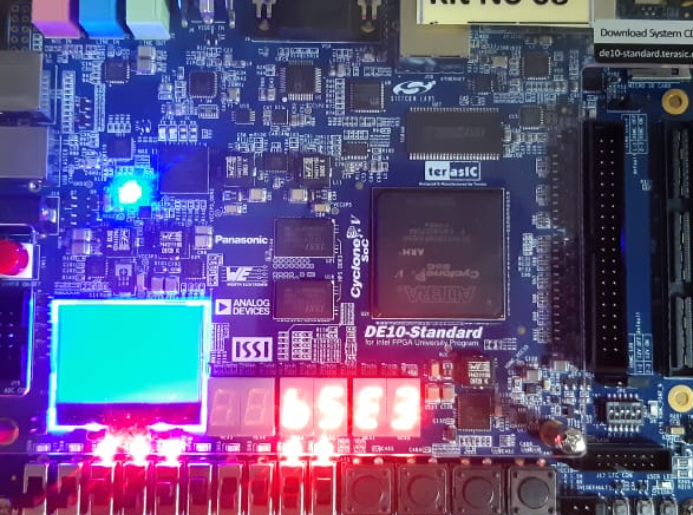
    assign LEDR[9]   = reg\_overflow;

    assign LEDR[8]   = reg\_carry;

    assign LEDR[7:0] = S;

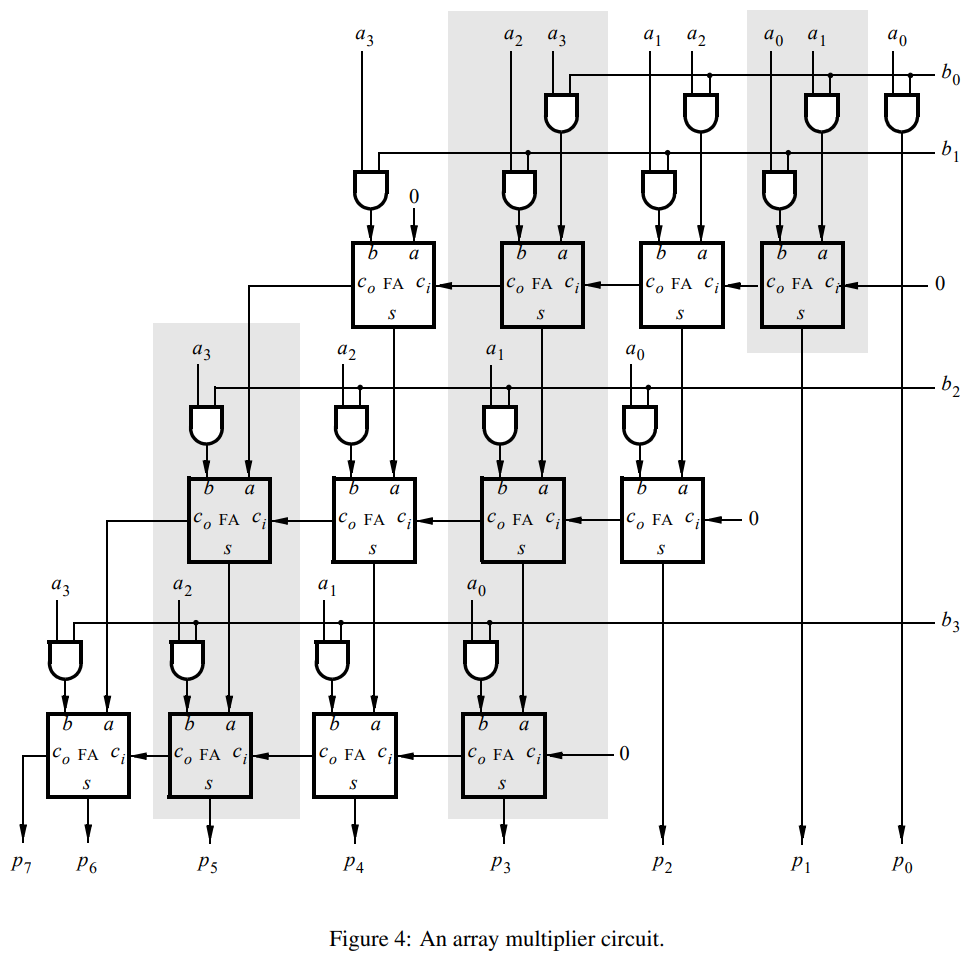
endmodule





## Part III

1. Create a new Quartus project.
2. Generate the required Verilog file. Use switches SW7−4 to represent the number A and switches SW3−0 to represent B. The hexadecimal values of A and B are to be displayed on the 7-segment displays HEX2 and HEX0, respectively. The result P = A × B is to be displayed on HEX5 − 4.



module *task\_3* (

*input*  [3:0] KEY,

*input*  [9:0] SW,

*output* [9:0] LEDR,

*output* [6:0] HEX0,

*output* [6:0] HEX1,

*output* [6:0] HEX2,

*output* [6:0] HEX3,

*output* [6:0] HEX4,

*output* [6:0] HEX5

);

    wire reset = KEY[2];

    wire clk = KEY[1];

    wire s = ~KEY[0];

    wire [3:0] DataA = SW[7:4];

    wire [3:0] DataB = SW[3:0];

    wire [7:0] P;

    reg [7:0] DataP;

    wire [7:0] A, sum;

    wire z;

    reg  done;

    reg [1:0] p\_state, n\_state;

    wire [7:0] B;

    reg enA, enB, enP, selP;

    integer k;

    parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;

    always @(\*) begin

        case (p\_state)

            S1:

            if (s == 0) n\_state = S1;

            else n\_state = S2;

            S2:

            if (z == 0) n\_state = S2;

            else n\_state = S3;

            S3:

            if (s == 1) n\_state = S3;

            else n\_state = S1;

            default: n\_state = 2'bxx;

        endcase

    end

    always @(posedge clk, negedge reset) begin

        if (reset == 0) p\_state <= n\_state;

        else p\_state <= n\_state;

    end

    always @(\*) begin

        enA  = 0;

        enB  = 0;

        enP  = 0;

        done = 0;

        selP = 0;

        case (p\_state)

            S1: enP = 1;

            S2: begin

                enA  = 1;

                enB  = 1;

                selP = 1;

                if (B[0]) enP = 1;

                else enP = 0;

            end

            S3: done = 1;

        endcase

    end

*shiftrne* ShiftB (

        DataB,

        LB,

        enB,

        1'b0,

        clk,

        B

    );

    defparam ShiftB.n = 4;

*shiftlne* ShiftA (

        {{4{1'b0}}, DataA},

        LA,

        enA,

        1'b0,

        clk,

        A

    );

    defparam ShiftA.n = 8;

    assign z   = (B == 0);

    assign sum = A + P;

    always @(selP, sum) for (k = 0; k < 2 \* 4; k = k + 1)

DataP[k] = selP ? sum[k] : 1'b0;

*regne* RegP (

        DataP,

        clk,

        Resetn,

        enP,

        P

    );

    defparam RegP.n = 8;

*decoder* digit0 (

        .in (DataA),

        .HEX(HEX0)

    );

*decoder* digit2 (

        .in (DataB),

        .HEX(HEX2)

    );

*decoder* digit4 (

        .in (DataP[7:4]),

        .HEX(HEX4)

    );

*decoder* digit5 (

        .in (DataP[7:4]),

        .HEX(HEX5)

    );

endmodule

module *shiftrne* (

*input* [n-1:0] in,

*output* [n-1:0] out,

*input* en,

*input* rst,

*input* clk,

*output* reg [n-1:0] LB

);

    parameter n = 4;

    integer i;

    always @(posedge clk, negedge rst) begin

        if (rst == 0) LB <= 0;

        else if (en == 1) begin

            LB[0] <= in[0];

            for (i = 1; i < n; i = i + 1) LB[i] <= in[i-1];

        end

    end

    assign out = LB;

endmodule

module *shiftlne* (

*input* [n-1:0] in,

*output* [n-1:0] out,

*input* en,

*input* rst,

*input* clk,

*output* reg [n-1:0] LA

);

    parameter n = 8;

    integer i;

    always @(posedge clk, negedge rst) begin

        if (rst == 0) LA <= 0;

        else if (en == 1) begin

            LA[n-1] <= in[n-1];

            for (i = 0; i < n - 1; i = i + 1) LA[i] <= in[i+1];

        end

    end

    assign out = LA;

endmodule

module *regne* (

*input* [n-1:0] in,

*input* clk,

*input* rst,

*input* en,

*output* reg [n-1:0] out

);

    parameter n = 8;

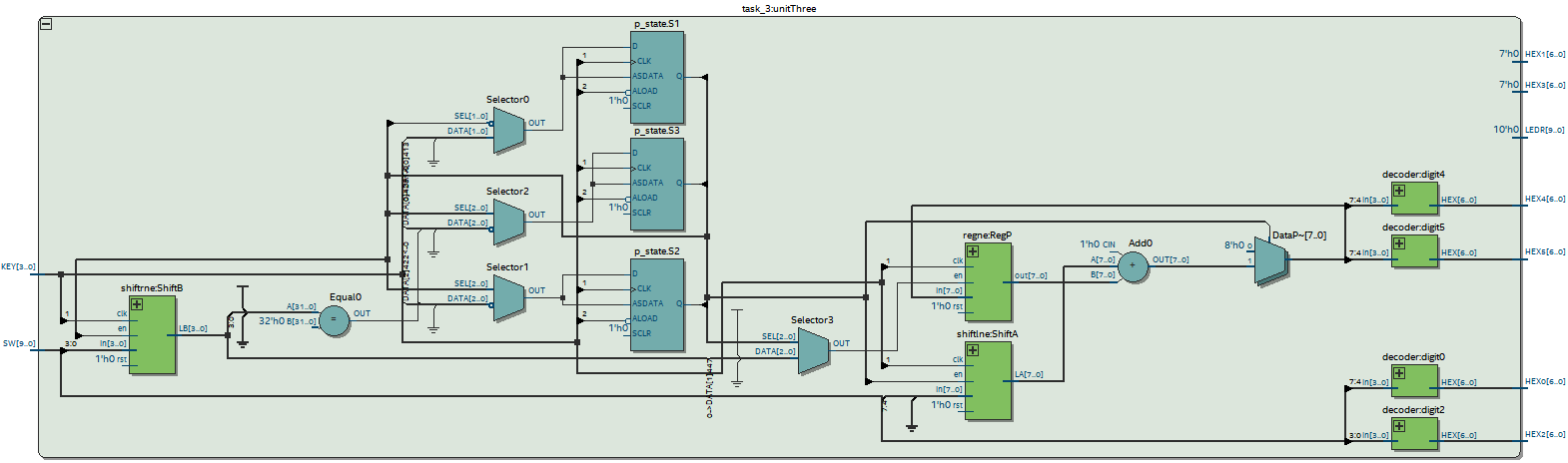
    always @(posedge clk, negedge rst) begin

        if (rst == 0) out <= 0;

        else if (en == 1) out <= in;

    end

endmodule



# Conclusion

In this exercise, we have examined arithmetic circuits that add, subtract, and multiply numbers. We have described each circuit in Verilog and implemented it on an Intel FPGA DE10-Standard. We have also tested the circuits by using them to perform a variety of arithmetic operations. The results of this exercise have shown that arithmetic circuits can be implemented efficiently on an FPGA. We have also learned that there are a variety of different ways to implement arithmetic circuits, each with its own advantages and disadvantages. This exercise has provided us with a good understanding of the basic principles of arithmetic circuits.