**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 11: 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](#_Toc134214872)

[2.1 Objectives 3](#_Toc134214873)

[2.2 Introduction 3](#_Toc134214874)

[2.3 Software 3](#_Toc134214875)

[3 Lab Procedure 4](#_Toc134214876)

[3.1 Part I 4](#_Toc134214877)

[3.2 Part II 8](#_Toc134214878)

[3.3 Part III 10](#_Toc134214879)

[4 Conclusion 12](#_Toc134214880)

# 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.

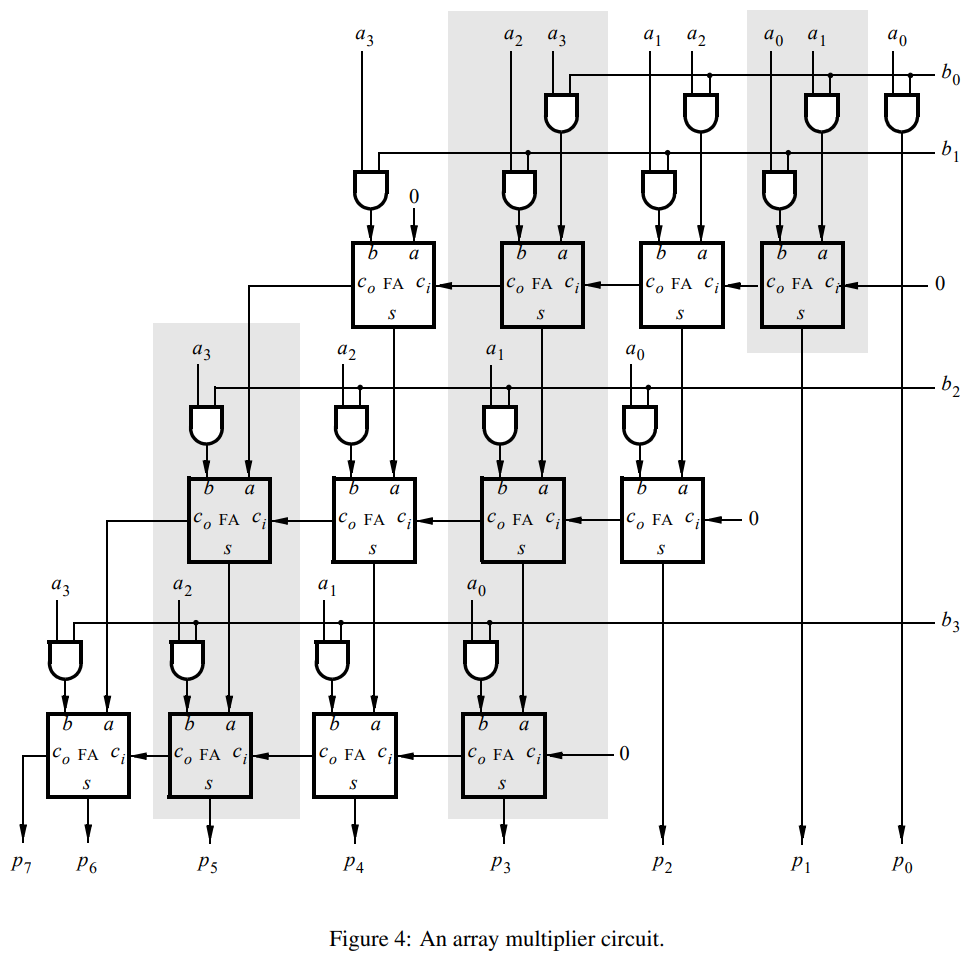
## Software

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.
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;

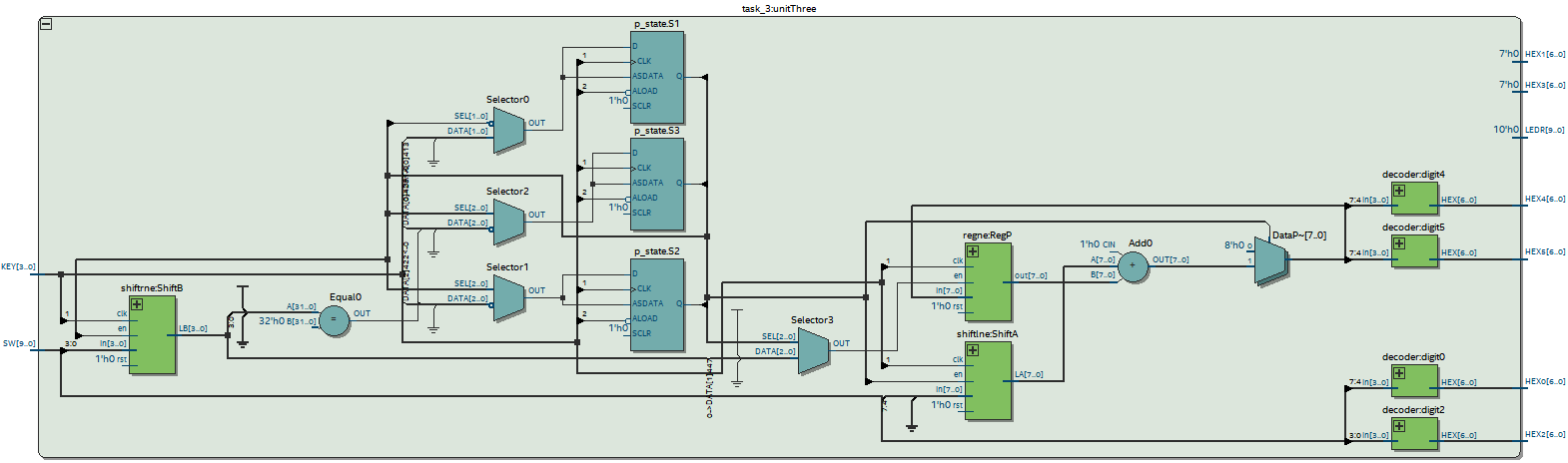
    always @(posedge clk, negedge rst) begin

        if (rst == 0) out <= 0;

        else if (en == 1) out <= in;

    end

endmodule



## Part II

1. Create a new Quartus project and write the required Verilog file.
2. Use switches SW7−0 to provide the data inputs to the circuit. Use SW9 as the enable signal EA for register A, and use SW8 as the enable for register B. When SW9 = 1 display the contents of register A on the red lights LEDR and display the contents of register B on these lights when SW8 = 1. Use KEY0 as a synchronous reset input and use KEY1 as a manual clock signal. Show the product P = A×B as a hexadecimal number on the 7-segment displays HEX3-0.
3. Make the necessary pin assignments needed to implement the circuit on your DE-series board and compile the circuit.
4. Test the functionality of your design by inputting various data values and observing the generated products.

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

);

    wire clk = KEY[1];

    wire rst = KEY[0];

    wire enA = SW[9];

    wire enB = SW[8];

    reg [7:0] A;

    reg [7:0] B;

    reg [15:0] P;

    reg [15:0] P\_reg;

    wire [7:0] shifted\_A[0:7];

    reg [15:0] p\_prod[0:7];

    integer i;

    // Register inputs

    always @(posedge clk, negedge rst) begin

        if (~rst) begin

            A <= 8'd0;

        end else if (enA) begin

            A <= SW[7:0];

        end

    end

    always @(posedge clk, negedge rst) begin

        if (~rst) begin

            B <= 8'd0;

        end else if (enB) begin

            B <= SW[7:0];

        end

    end

    // Shift A left by row index

    assign shifted\_A[0] = A << 0;

    assign shifted\_A[1] = A << 1;

    assign shifted\_A[2] = A << 2;

    assign shifted\_A[3] = A << 3;

    assign shifted\_A[4] = A << 4;

    assign shifted\_A[5] = A << 5;

    assign shifted\_A[6] = A << 6;

    assign shifted\_A[7] = A << 7;

    // Calculate partial product for each row

    always @(posedge clk, negedge rst) begin

        if (~rst) begin

            for (i = 0; i < 8; i = i + 1) begin

                p\_prod[i] <= 16'd0;

            end

        end else begin

            for (i = 0; i < 8; i = i + 1) begin

                p\_prod[i] <= shifted\_A[i] \* B;

            end

        end

    end

    // Add partial products to get final product

    always @(posedge clk, negedge rst) begin

        if (~rst) begin

            P <= 16'd0;

        end else begin

            P <= p\_prod[0] + p\_prod[1] + p\_prod[2] + p\_prod[3] +

            p\_prod[4] + p\_prod[5] + p\_prod[6] + p\_prod[7];

        end

    end

    // Register output

    always @(posedge clk, negedge rst) begin

        if (~rst) begin

            P\_reg <= 16'd0;

        end else begin

            P\_reg <= P;

        end

    end

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

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

*decoder* digit2 (.in (P[11:8]), .HEX(HEX2));

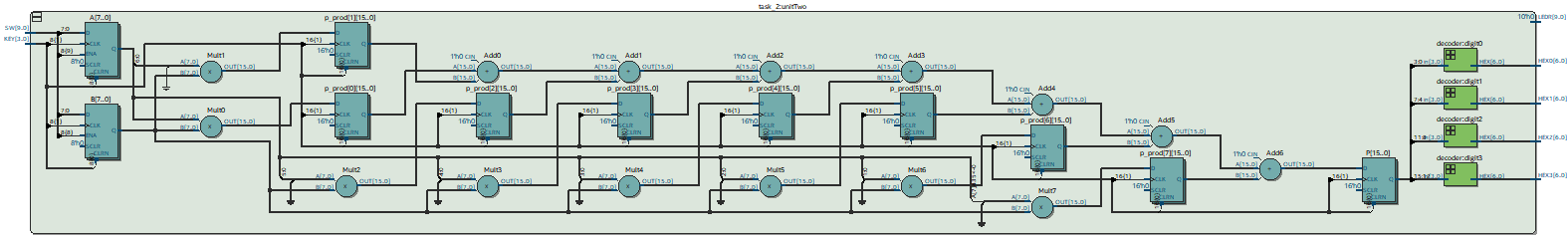
*decoder* digit3 (

        .in (P[15:12]),

        .HEX(HEX3)

    );

endmodule



## Part III

In this part you are to implement an 8 x 8 multiplier circuit by using the adder-tree approach. Inputs A and B, as well as the output P should be registered as in Part II.

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

);

    wire clk = KEY[1];

    wire rst = KEY[0];

    wire enA = SW[9];

    wire enB = SW[8];

    reg [7:0] A;

    reg [7:0] B;

    reg [15:0] P;

    reg [15:0] P\_reg;

    // calculate partial products using for loops

    integer i, j;

    reg [7:0] p[8][8];

    always @(\*) begin

        for (i = 0; i < 8; i = i + 1) begin

            for (j = 0; j < 8; j = j + 1) begin

                p[i][j] = A[i] & B[j] ? 1 << (i + j) : 0;

            end

        end

    end

    // sum partial products to get result

    always @(posedge clk, posedge rst) begin

        if (rst) begin

            A <= 0;

            B <= 0;

            P <= 0;

            P\_reg <= 0;

        end else begin

            // update inputs based on enable signals

            if (enA) A <= SW[7:0];

            if (enB) B <= SW[7:0];

            // sum partial products

            for (i = 0; i < 15; i = i + 1) begin

                reg [7:0] s;

                s = 0;

                for (j = 0; j < 8; j = j + 1) begin

                    if (i - j >= 0 && i - j < 8) begin

                        s = s + p[j][i-j];

                    end

                end

                P[i] = s[0];

                if (i < 15) begin

                    P[i+1] = s[1:0];

                end

            end

            P\_reg <= P;  // store the result for display

        end

    end

*decoder* digit0 (

        .in (P[3:0]),

        .HEX(HEX0)

    );

*decoder* digit1 (

        .in (P[7:4]),

        .HEX(HEX1)

    );

*decoder* digit2 (

        .in (P[11:8]),

        .HEX(HEX2)

    );

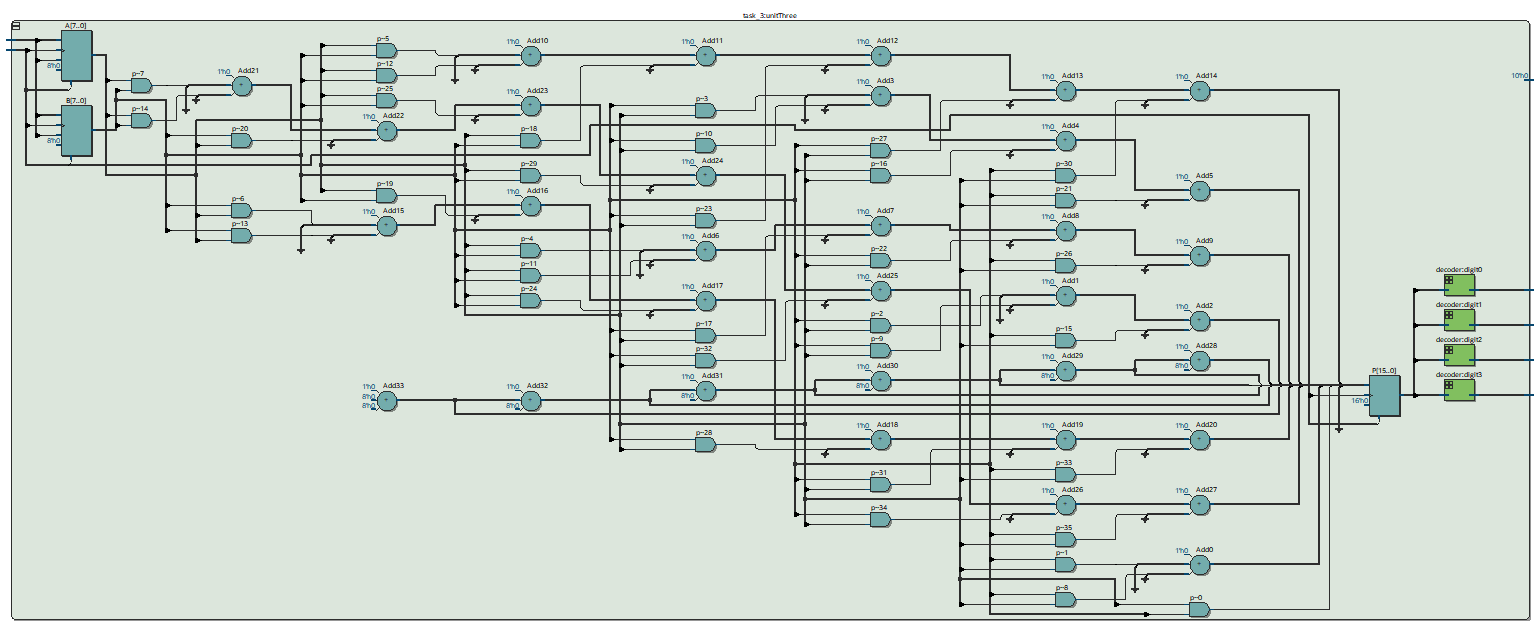
*decoder* digit3 (

        .in (P[15:12]),

        .HEX(HEX3)

    );

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.