# Laboratory Exercise 6

### Adders, Subtractors, and Multipliers

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.

**Part I**

Consider again the four-bit ripple-carry adder circuit used in lab exercise 2; its diagram is reproduced in Figure [1](#_bookmark32).

*b*3 *a*3 *c*3

FA

FA

FA

FA

*b*2 *a*2 *c*2

*b*1 *a*1 *c*1

*b*0 *a*0 *c*in

*carry*

*s*3 *s*2 *s*1 *s*0

Figure 1: A four-bit ripple carry adder.

This circuit can be implemented using a ’+’ sign in Verilog. For example, the following code fragment adds *n*-bit numbers *A* and *B* to produce outputs *sum* and *carry*:

**wire** [n-1:0] sum;

**wire** carry;

. . .

**assign** {carry, sum} = A + B;

Use this construct to implement a circuit shown in Figure [2](#_bookmark33). This circuit, which is often called an *accumulator*, is used to add the value of an input *A* to itself repeatedly. The circuit includes a carry out from the adder, as well as an *overflow* output signal. If the input *A* is considered as a 2’s-complement number, then *overflow* should be set to 1 in the case where the output *sum* produced does not represent a correct 2’s-complement result.

Perform the following steps:

1. Create a new Quartus project. Write Verilog code that describes the circuit in Figure [2](#_bookmark33).
2. Connect input *A* to switches *SW*7−0, use *KEY*0 as an active-low asynchronous reset, and use *KEY*1 as a manual clock input. The sum from the adder should be displayed on the red lights *LEDR*7−0, the registered carry signal should be displayed on *LEDR*8, and the registered *overflow* signal should be displayed on *LEDR*9. 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.

*A*

8

*R*

Q



8

*Clock*

Q *D*

Logic circuit

*+*

0

8

Q *D*

*R*

Q

*R*

Q

*overflow carry S*

*R*

Q

Figure 2: An eight-bit accumulator circuit.

## 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.

## Part III

Figure [3](#_bookmark34)*a* gives an example of paper-and-pencil multiplication *P* = *A* × *B*, where *A* = 11 and *B* = 12.

1 1

x 1 2

1 0 1 1

x 1 1 0 0

0 0 0 0

*a*3 *b*2 *a*3 *b*3 *a*2 *b*3

*p*7 *p*6 *p*5

*a*3 *b*1 *a*2 *b*2 *a*1 *b*3

*p*4

*a*3 *b*0 *a*2 *b*1 *a*1 *b*2 *a*0 *b*3

*p*3

*a*2 *b*0 *a*1 *b*0 *a*0 *b*0 *a*1 *b*1 *a*0 *b*1

*a*0 *b*2

*p*2

*p*1

*p*0

*a*3

x *b*3

*a*2 *a*1 *a*0

*b*2 *b*1 *b*0

2 2 0 0 0 0

1 1 1 0 1 1

1 3 2

1 0 1 1

1 0 0 0 0 1 0 0

* 1. Decimal b) Binary c) Implementation

Figure 3: Multiplication of binary numbers.

We compute *P* = *A B* as an addition of summands. The first summand is equal to *A* times the ones digit of *B*. The second summand is *A* times the tens digit of *B*, shifted one position to the left. We add the two summands to form the product *P* = 132.

×

Part *b* of the figure shows the same example using four-bit binary numbers. To compute *P* = *A B*, we first form summands by multiplying *A* by each digit of *B*. Since each digit of *B* is either 1 or 0, the summands are either

×

shifted versions of *A* or 0000. Figure [3](#_bookmark34)*c* shows how each summand can be formed by using the Boolean AND operation of *A* with the appropriate bit of *B*.

A four-bit circuit that implements *P* = *A B* is illustrated in Figure [4](#_bookmark35). Because of its regular structure, this type of multiplier circuit is called an *array multiplier*. The shaded areas correspond to the shaded columns in Figure [3](#_bookmark34)*c*. In each row of the multiplier AND gates are used to produce the summands, and full adder modules are used to generate the required sums.

×

*b*0



*a*3

*a*2 *a*3

*a*1 *a*2

*a*0 *a*1

*a*0

0

*b a*

*b a*

*b a*

*b a*

*co* FA *ci co* FA *ci co* FA *ci co* FA *ci s s s s*

*a*3

*a*2

*a*1

*a*0

*b a*

*b a*

*b a*

*b a*

*co* FA *ci co* FA *ci co* FA *ci s s s*

*co* FA *ci s*

0

*a*3

*a*2

*a*1

*a*0

*b a*

*b a*

*b a*

*b a*

*co* FA *ci co* FA *ci co* FA *ci s s s*

*co* FA *ci s*

0

*b*1

0

*b*2

*b*3

*p*7 *p*6 *p*5 *p*4 *p*3 *p*2 *p*1 *p*0

Figure 4: An array multiplier circuit.

Perform the following steps to implement the array multiplier circuit:

1. Create a new Quartus project.
2. Generate the required Verilog file. Use switches *SW*7−4 to represent the number *A* and switches *SW*3−0 to represent *B*. The hexadecimal values of *A* and *B* are to be displayed on the 7-segment displays *HEX*2 and *HEX*0, respectively. The result *P* = *A* × *B* is to be displayed on *HEX*5 − 4.
3. Make the necessary pin assignments needed to implement the circuit on your DE-series board, and compile the circuit.
4. Use simulation to verify your design.
5. Download your circuit onto your DE-series board and test its functionality.

## Part IV

In Part III, an array multiplier was implemented using full adder modules. At a higher level, a row of full adders functions as an *n*-bit adder and the array multiplier circuit can be represented as shown in Figure [5](#_bookmark36).

*a*3 *a*2

*a*1 *a*0

*b*0



*a*3

*a*2

*a*1

*a*0

0

*b*3 *co*

*a*3

*s*3

*b*2

*a*3

*a*2

*a*1

*a*2

n-bit Adder

*s*2

*a*0

*b*1 *a*1

*s*1

*b*0 *a*0 *ci*

*s*0

0

*b*3 *a*3 *co*

*s*3

*a*2

*b*2

*a*1

*a*2

n-bit Adder

*s*2

*a*0

*b*1

*a*1

*s*1

*b*0 *a*0 *ci*

*s*0

0

*a*3

*b*3 *a*3 *co*

*s*3

*b*2 *a*2

n-bit Adder

*s*2

*b*1

*a*1

*s*1

*b*0 *a*0 *ci*

*s*0

0

*b*1

*b*2

*b*3

*p*7 *p*6

*p*5 *p*4 *p*3

*p*2 *p*1 *p*0

Figure 5: An array multiplier implemented using *n*-bit adders.

Each *n*-bit adder adds a shifted version of *A* for a given row and the *partial product* of the row above. Abstracting the multiplier circuit as a sequence of additions allows us to build larger multipliers. The multiplier should consist of n-bit adders arranged in a structure shown in Figure [5](#_bookmark36). Use this approach to implement an 8 x 8 multiplier circuit with registered inputs and outputs, as shown in Figure [6](#_bookmark37).

*Data inputs*

*Clock*



8

*EA*

*EB*

*A*

*B*

16

*D*

Q

*Multiplier*

*E*

*D*

Q

*D*

Q

*E*

8

*P*

*R*

Q

Figure 6: A registered multiplier circuit.

Perform the following steps:

1. Create a new Quartus project and write the required Verilog file.
2. Use switches *SW*7−0 to provide the data inputs to the circuit. Use *SW*9 as the enable signal *EA* for register *A*, and use *SW*8 as the enable for register *B*. When *SW*9 = 1 display the contents of register A on the red lights LEDR, and display the contents of register B on these lights when *SW*8 = 1. Use *KEY*0 as a synchronous reset input, and use *KEY*1 as a manual clock signal. Show the product *P* = *A B* as a hexadecimal number on the 7-segment displays *HEX3-0*.

×

1. Make the necessary pin assignments needed to implement the circuit on your DE-series board, and compile the circuit.
2. Test the functionality of your design by inputting various data values and observing the generated products.

## Part V

Part IV showed how to implement multiplication *A B* as a sequence of additions, by accumulating the shifted versions of *A* one row at a time. Another way to implement this circuit is to perform addition using an adder tree. An adder tree is a method of adding several numbers together in a parallel fashion. This idea is illustrated in Figure [7](#_bookmark38). In the figure, numbers *A*, *B*, *C*, *D*, *E*, *F* , *G*, and *H* are added together in parallel. The addition *A* + *B* happens simultaneously with *C* + *D*, *E* + *F* and *G* + *H*. The result of these operations are then added in parallel again, until the final sum *P* is computed.

×

PART-1

//KARTHIK J-1MS21EE028

module Accumulator (

input [7:0] SW, // Input A (Switches SW7-0)

input KEY0, // Asynchronous Reset (Active Low)

input KEY1, // Manual Clock

output reg [7:0] LEDR, // Accumulated Sum (LEDR7-0)

output reg LEDR8, // Carry Output

output reg LEDR9, // Overflow Output

output [6:0] HEX0, HEX1, HEX2, HEX3 // 7-segment displays

);

reg [7:0] S; // Accumulated Sum

reg carry; // Carry Signal

wire [7:0] sum; // Adder Sum Output

wire carry\_out; // Adder Carry Output

wire overflow; // Overflow Signal

// Adder: Add input A to the current sum S

assign {carry\_out, sum} = S + SW;

// Overflow Detection for 2's Complement Arithmetic

assign overflow = (S[7] & SW[7] & ~sum[7]) | (~S[7] & ~SW[7] & sum[7]);

// Registering Values on Clock Edge or Reset

always @(posedge KEY1 or negedge KEY0) begin

if (!KEY0) begin

S <= 0;

carry <= 0;

LEDR8 <= 0; // Clear carry

LEDR9 <= 0; // Clear overflow

end else begin

S <= sum; // Update accumulated sum

carry <= carry\_out; // Update carry

LEDR8 <= carry\_out; // Update carry LED

LEDR9 <= overflow; // Update overflow LED

end

end

// Assign outputs

assign LEDR[7:0] = S; // Display accumulated sum on LEDs

// Display values on HEX displays

Hex\_Display hex3(SW[7:4], HEX3); // Display high nibble of A

Hex\_Display hex2(SW[3:0], HEX2); // Display low nibble of A

Hex\_Display hex1(S[7:4], HEX1); // Display high nibble of S

Hex\_Display hex0(S[3:0], HEX0); // Display low nibble of S

endmodule

// 7-segment display module

module Hex\_Display(input [3:0] nibble, output reg [6:0] HEX);

always @(\*) begin

case (nibble)

4'h0: HEX = 7'b1000000;

4'h1: HEX = 7'b1111001;

4'h2: HEX = 7'b0100100;

4'h3: HEX = 7'b0110000;

4'h4: HEX = 7'b0011001;

4'h5: HEX = 7'b0010010;

4'h6: HEX = 7'b0000010;

4'h7: HEX = 7'b1111000;

4'h8: HEX = 7'b0000000;

4'h9: HEX = 7'b0010000;

4'hA: HEX = 7'b0001000;

4'hB: HEX = 7'b0000011;

4'hC: HEX = 7'b1000110;

4'hD: HEX = 7'b0100001;

4'hE: HEX = 7'b0000110;

4'hF: HEX = 7'b0001110;

default: HEX = 7'b1111111; // Blank display

endcase

end

endmodule

PART-2

//KARTHIK J-1MS21EE028

module Accumulator\_Add\_Sub (

input [7:0] SW, // Input A (Switches SW7-0)

input KEY0, // Asynchronous Reset (Active Low)

input KEY1, // Manual Clock

input add\_sub, // Add/Subtract Control (0 = Add, 1 = Subtract)

output reg [7:0] LEDR, // Accumulated Sum (LEDR7-0)

output reg LEDR8, // Carry Output

output reg LEDR9, // Overflow Output

output [6:0] HEX0, HEX1, HEX2, HEX3 // 7-segment displays

);

reg [7:0] S; // Accumulated Sum

reg carry; // Carry Signal

wire [7:0] operand; // Operand for adder (A or -A)

wire [7:0] sum; // Adder Sum Output

wire carry\_out; // Adder Carry Output

wire overflow; // Overflow Signal

// Determine operand based on add\_sub signal

assign operand = add\_sub ? (~SW + 1) : SW;

// Adder: Add operand to the current sum S

assign {carry\_out, sum} = S + operand;

// Overflow Detection for 2's Complement Arithmetic

assign overflow = (S[7] & operand[7] & ~sum[7]) | (~S[7] & ~operand[7] & sum[7]);

// Registering Values on Clock Edge or Reset

always @(posedge KEY1 or negedge KEY0) begin

if (!KEY0) begin

S <= 0;

carry <= 0;

LEDR8 <= 0; // Clear carry

LEDR9 <= 0; // Clear overflow

end else begin

S <= sum; // Update accumulated sum

carry <= carry\_out; // Update carry

LEDR8 <= carry\_out; // Update carry LED

LEDR9 <= overflow; // Update overflow LED

end

end

// Assign outputs

assign LEDR[7:0] = S; // Display accumulated sum on LEDs

// Display values on HEX displays

Hex\_Display hex3(SW[7:4], HEX3); // Display high nibble of A

Hex\_Display hex2(SW[3:0], HEX2); // Display low nibble of A

Hex\_Display hex1(S[7:4], HEX1); // Display high nibble of S

Hex\_Display hex0(S[3:0], HEX0); // Display low nibble of S

endmodule

// 7-segment display module

module Hex\_Display(input [3:0] nibble, output reg [6:0] HEX);

always @(\*) begin

case (nibble)

4'h0: HEX = 7'b1000000;

4'h1: HEX = 7'b1111001;

4'h2: HEX = 7'b0100100;

4'h3: HEX = 7'b0110000;

4'h4: HEX = 7'b0011001;

4'h5: HEX = 7'b0010010;

4'h6: HEX = 7'b0000010;

4'h7: HEX = 7'b1111000;

4'h8: HEX = 7'b0000000;

4'h9: HEX = 7'b0010000;

4'hA: HEX = 7'b0001000;

4'hB: HEX = 7'b0000011;

4'hC: HEX = 7'b1000110;

4'hD: HEX = 7'b0100001;

4'hE: HEX = 7'b0000110;

4'hF: HEX = 7'b0001110;

default: HEX = 7'b1111111; // Blank display

endcase

end

endmodule

PART-3

//KARTHIK J-1MS21EE028

module part3 (SW, HEX5, HEX4, HEX2, HEX0);

input [7:0] SW;

output [0:6] HEX5, HEX4, HEX2, HEX0;

wire [3:0] A, B;

wire [7:0] P;

wire [3:1] C\_b1; // carries for row that ANDs with B1

wire [5:2] PP1; // partial products from row that ANDs with B1

wire [3:1] C\_b2; // carries for row that ANDs with B2

wire [6:3] PP2; // partial products from row that ANDs with B2

wire [3:1] C\_b3; // carries for row that ANDs with B3

assign A = SW[7:4];

assign B = SW[3:0];

assign P[0] = A[0] & B[0];

// module fa (a, b, ci, s, co);

fa b1\_a0 (A[1] & B[0], A[0] & B[1], 1'b0, P[1], C\_b1[1]);

fa b1\_a1 (A[2] & B[0], A[1] & B[1], C\_b1[1], PP1[2], C\_b1[2]);

fa b1\_a2 (A[3] & B[0], A[2] & B[1], C\_b1[2], PP1[3], C\_b1[3]);

fa b1\_a3 (1'b0, A[3] & B[1], C\_b1[3], PP1[4], PP1[5]);

// module fa (a, b, ci, s, co);

fa b2\_a0 (PP1[2], A[0] & B[2], 1'b0, P[2], C\_b2[1]);

fa b2\_a1 (PP1[3], A[1] & B[2], C\_b2[1], PP2[3], C\_b2[2]);

fa b2\_a2 (PP1[4], A[2] & B[2], C\_b2[2], PP2[4], C\_b2[3]);

fa b2\_a3 (PP1[5], A[3] & B[2], C\_b2[3], PP2[5], PP2[6]);

// module fa (a, b, ci, s, co);

fa b3\_a0 (PP2[3], A[0] & B[3], 1'b0, P[3], C\_b3[1]);

fa b3\_a1 (PP2[4], A[1] & B[3], C\_b3[1], P[4], C\_b3[2]);

fa b3\_a2 (PP2[5], A[2] & B[3], C\_b3[2], P[5], C\_b3[3]);

fa b3\_a3 (PP2[6], A[3] & B[3], C\_b3[3], P[6], P[7]);

// drive the displays through a 7-seg decoders

hex7seg digit\_5 (P[7:4], HEX5);

hex7seg digit\_4 (P[3:0], HEX4);

hex7seg digit\_3 (A, HEX2);

hex7seg digit\_2 (B, HEX0);

endmodule

module fa (a, b, ci, s, co);

input a, b, ci;

output s, co;

wire a\_xor\_b;

assign a\_xor\_b = a ^ b;

assign s = a\_xor\_b ^ ci;

assign co = (~a\_xor\_b & b) | (a\_xor\_b & ci);

endmodule

module hex7seg (hex, display);

input [3:0] hex;

output [0:6] display;

reg [0:6] display;

always @ (hex)

case (hex)

4'h0: display = 7'b0000001;

4'h1: display = 7'b1001111;

4'h2: display = 7'b0010010;

4'h3: display = 7'b0000110;

4'h4: display = 7'b1001100;

4'h5: display = 7'b0100100;

4'h6: display = 7'b0100000;

4'h7: display = 7'b0001111;

4'h8: display = 7'b0000000;

4'h9: display = 7'b0000100;

4'hA: display = 7'b0001000;

4'hb: display = 7'b1100000;

4'hC: display = 7'b0110001;

4'hd: display = 7'b1000010;

4'hE: display = 7'b0110000;

4'hF: display = 7'b0111000;

endcase

endmodule

PART-4

//KARTHIK J-1MS21EE028

module part4 (SW, KEY, LEDR, HEX3, HEX2, HEX1, HEX0);

input [9:0] SW;

input [1:0] KEY;

output [9:0] LEDR;

output [0:6] HEX3, HEX2, HEX1, HEX0;

wire Resetn, Clock;

reg [7:0] A, B;

reg [15:0] P\_reg;

wire [15:0] PP0, PP1, PP2, PP3, PP4, PP5, PP6;

wire [15:0] P;

assign Resetn = KEY[0];

assign Clock = KEY[1];

always @(posedge Clock)

begin

if (~Resetn)

begin

A <= 8'b0; B <= 8'b0; P\_reg <= 16'b0;

end

else

begin

if (SW[9]) A <= SW[7:0];

if (SW[8]) B <= SW[7:0];

P\_reg <= P;

end

end

assign LEDR[7:0] = SW[9] ? A : (SW[8] ? B : 8'b0);

assign LEDR[9:8] = SW[9:8];

assign PP0 = { 8'd0, A & {8{B[0]}} };

assign PP1 = PP0 + { 7'd0, A & {8{B[1]}}, 1'd0 };

assign PP2 = PP1 + { 6'd0, A & {8{B[2]}}, 2'd0 };

assign PP3 = PP2 + { 5'd0, A & {8{B[3]}}, 3'd0 };

assign PP4 = PP3 + { 4'd0, A & {8{B[4]}}, 4'd0 };

assign PP5 = PP4 + { 3'd0, A & {8{B[5]}}, 5'd0 };

assign PP6 = PP5 + { 2'd0, A & {8{B[6]}}, 6'd0 };

assign P = PP6 + { 1'd0, A & {8{B[7]}}, 7'd0 };

// drive the display through a 7-seg decoder

hex7seg digit\_3 (P\_reg[15:12], HEX3);

hex7seg digit\_2 (P\_reg[11:8], HEX2);

hex7seg digit\_1 (P\_reg[7:4], HEX1);

hex7seg digit\_0 (P\_reg[3:0], HEX0);

endmodule

module hex7seg (hex, display);

input [3:0] hex;

output [0:6] display;

reg [0:6] display;

always @ (hex)

case (hex)

4'h0: display = 7'b0000001;

4'h1: display = 7'b1001111;

4'h2: display = 7'b0010010;

4'h3: display = 7'b0000110;

4'h4: display = 7'b1001100;

4'h5: display = 7'b0100100;

4'h6: display = 7'b0100000;

4'h7: display = 7'b0001111;

4'h8: display = 7'b0000000;

4'h9: display = 7'b0001100;

4'hA: display = 7'b0001000;

4'hb: display = 7'b1100000;

4'hC: display = 7'b0110001;

4'hd: display = 7'b1000010;

4'hE: display = 7'b0110000;

4'hF: display = 7'b0111000;

endcase

endmodule

PART-5

//KARTHIK J-1MS21EE028

module part5 (SW, KEY, LEDR, HEX3, HEX2, HEX1, HEX0);

input [9:0] SW;

input [1:0] KEY;

output [9:0] LEDR;

output [0:6] HEX3, HEX2, HEX1, HEX0;

wire Resetn, Clock;

reg [7:0] A, B;

reg [15:0] P\_reg;

wire [15:0] PP0, PP1, PP2, PP3, PP4, PP5, PP6;

wire [15:0] P;

assign Resetn = KEY[0];

assign Clock = KEY[1];

always @(posedge Clock)

begin

if (~Resetn)

begin

A <= 8'b0; B <= 8'b0; P\_reg <= 16'b0;

end

else

begin

if (SW[9]) A <= SW[7:0];

if (SW[8]) B <= SW[7:0];

P\_reg <= P;

end

end

assign LEDR[7:0] = SW[9] ? A : (SW[8] ? B : 8'b0);

assign LEDR[9:8] = SW[9:8];

assign PP0 = { 8'd0, A & {8{B[0]}} } + { 7'd0, A & {8{B[1]}}, 1'd0 };

assign PP1 = { 6'd0, A & {8{B[2]}}, 2'd0 } + { 5'd0, A & {8{B[3]}}, 3'd0 };

assign PP2 = { 4'd0, A & {8{B[4]}}, 4'd0 } + { 3'd0, A & {8{B[5]}}, 5'd0 };

assign PP3 = { 2'd0, A & {8{B[6]}}, 6'd0 } + { 1'd0, A & {8{B[7]}}, 7'd0 };

assign PP4 = PP0 + PP1;

assign PP5 = PP2 + PP3;

assign P = PP4 + PP5;

// drive the displays through 7-seg decoders

hex7seg digit\_3 (P\_reg[15:12], HEX3);

hex7seg digit\_2 (P\_reg[11:8], HEX2);

hex7seg digit\_1 (P\_reg[7:4], HEX1);

hex7seg digit\_0 (P\_reg[3:0], HEX0);

endmodule

module hex7seg (hex, display);

input [3:0] hex;

output [0:6] display;

reg [0:6] display;

always @ (hex)

case (hex)

4'h0: display = 7'b0000001;

4'h1: display = 7'b1001111;

4'h2: display = 7'b0010010;

4'h3: display = 7'b0000110;

4'h4: display = 7'b1001100;

4'h5: display = 7'b0100100;

4'h6: display = 7'b0100000;

4'h7: display = 7'b0001111;

4'h8: display = 7'b0000000;

4'h9: display = 7'b0001100;

4'hA: display = 7'b0001000;

4'hb: display = 7'b1100000;

4'hC: display = 7'b0110001;

4'hd: display = 7'b1000010;

4'hE: display = 7'b0110000;

4'hF: display = 7'b0111000;

endcase

endmodule