**Verilog Project**

**Design Implemented** – Parameterized n-bit ALU

**Approach**

The implemented ALU takes two n-bit inputs din1 and din2 and performs arithmetic or logical operations on it depending on the value of bits mode and op.

**mode** decides whether a logical(0) or arithmetic(1) operation is performed on the given data.

**op** decides what operation would be performed

**In Arithmetic mode**

|  |  |
| --- | --- |
| **op[2:0]** | **Operation** |
| 0 | Addition |
| 1 | Subtraction |
| 2 | Multiplication |
| 3 | Division |
| 4 | Comparator |

**In Logical mode**

|  |  |
| --- | --- |
| **op[2:0]** | **Operation** |
| 0 | AND |
| 1 | OR |
| 2 | XOR |
| 3 | NOT |
| 4 | Logical Shift |
| 5 | Arithmetic Shift |

The output obtained is flopped in a 2n-bit register **dout** after the computation has been completed.

The lower n bits of **dout** are then converted from binary to BCD with three digits

Three seven segment displays on the DE2 board are then used to display the result on displays referred to as HEX[2], HEX[1], and HEX[0].

**Verification of ALU architecture in high-level language c++**

The verification was done using a C program ***alu\_test.c*** that takes in operands and the required operator from the user and outputs the result on the terminal.

#include <iostream>

using namespace std;

int main() {

int result,mode,op,din1,din2; //inputs

cout << "Enter mode: "; // choose between logical and arithmetic mode

cin >> mode;

cout << "Enter op: "; // decide the type of operation

cin >> op;

cout << "Enter operand 1: "; // Enter operand 1

cin >> din1;

cout << "Enter operand 2: "; // Enter operand 2

cin >> din2;

result = mode ? (op==0 ? din1 + din2 //addition

: op==1 ? din1 - din2 // subtraction

: op==2 ? din1 \* din2 // multiplication

: op==3 ? din1 / din2 // division

: op==4 ? din1 > din2 // comparison

: 0)

: (op==0 ? din1 & din2 // bitwise AND

: op==1 ? din1 | din2

: op==2 ? din1 ^ din2

: op==3 ? !din1

: op==4 ? din2 ? din1 << 1 : din1 >> 1

: op==5 ? din2 ? din1 << 1 : din1 >> 1

: 0 );

cout << "Result = " << result;

return 0;

}

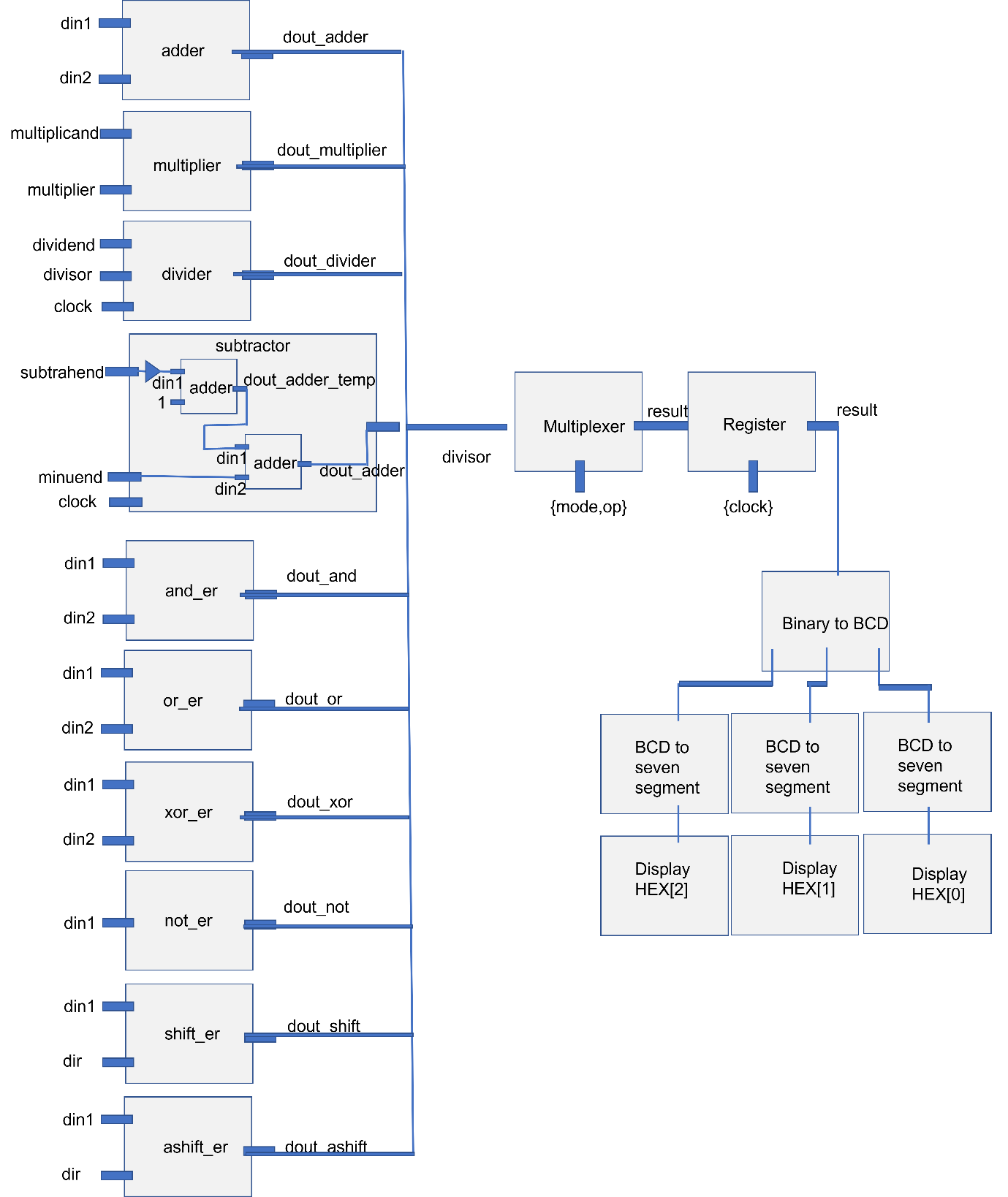
**Architecture**

The architecture can be broken down into the control unit, the display unit and the arithmetic/logical unit.

The job of the control unit is to ensure that the correct operation is performed according to the user specification.

The display unit converts the result obtained from the ALU and packages it into a format through which it can be displayed using seven-segment LEDs.

The arithmetic/logical unit as the name implies performs all the mathematical operations required

****

**Module Sketches**

**Graphical user interface

Description automatically generated**

**Arithmetic Modules**

**Timeline

Description automatically generated with medium confidence**

**Logical Modules**

**Novelty in Design**

* The fact that this is the first publicly available n-bit ALU makes my design novel. The complicated part was converting fixed bit width arithmetic blocks for any random bit width N. The trickiest parts of the design were the –
* Carry Look Ahead Adder
* Subtractor
* Multiplier
* Divider
* Binary to seven segment display interfacing
* In all the circuits mentioned, the **generate** Verilog command was used to dynamically define the structure of the hardware blocks.
* Another noteworthy point is the fact that the arithmetic blocks were optimized to get the result in minimum time. Implementation of the adder which is reused in subtractor and multiplier is done using carry lookahead logic using dynamic hardware instantiation saving on timing, power and area.
* Area and power optimization is done by sharing hardware between the adder and the subtractor. This has been implemented by reusing the adder unit to first generate a 2’s compliment of the minuend and then add it to the subtrahend
* Power optimization is implemented in the input and output stages as well as the modules are activated only when flags start and done are set to specific values.

**Verification Strategy**

Each individual module was tested individually using individual testbenches and finally combined to get the final design.

To verify the complete design, the din1 and din2 values were kept same and the value of the mode and op bits were changed. The observed result was compared to the theoretical value to verify operation.

**Appendix**

**Verilog Code**

module top #(parameter DWIDTH = 8)(

inout clk,

input mode,

input [2:0] op,

input [DWIDTH-1:0] din1,

input [DWIDTH-1:0] din2,

output reg [DWIDTH-1:0] dout

);

reg done;

reg [(DWIDTH<<1):0]counter;

reg [(DWIDTH<<1)-1:0] result; //to store the result of logical or arithmetic operation

wire [(DWIDTH<<1)-1:0] dout\_adder;

wire [(DWIDTH<<1)-1:0] dout\_subtractor;

wire [(DWIDTH<<1)-1:0] dout\_multiplier;

wire [(DWIDTH<<1)-1:0] dout\_divider;

wire [(DWIDTH<<1)-1:0] dout\_comparator;

adder add1(din1,din2,dout\_adder);

subtractor sub1(clk,din1,din2,dout\_subtractor);

multiplier mult1(din1,din2,dout\_multiplier);

divider div1(.clk(clk),.x(din1),.y(din2),.dout(dout\_divider));

comparator comp1(din1,din2,dout\_comparator);

wire [DWIDTH-1:0] dout\_and;

wire [DWIDTH-1:0] dout\_or;

wire [DWIDTH-1:0] dout\_xor;

wire [DWIDTH-1:0] dout\_not;

wire [DWIDTH-1:0] dout\_shift;

wire [DWIDTH-1:0] dout\_ashift;

and\_er and1(din1,din2,dout\_and);

or\_er or1(din1,din2,dout\_or);

xor\_er xor1(din1,din2,dout\_xor);

not\_er not1(din1,dout\_not);

shift\_er shift1(din1,dir,dout\_shift);

ashift\_er ashift1(din1,dir,dout\_ashift);

wire [11:0] ssd\_temp;

BCD\_Convert bin\_to\_bcd(dout[DWIDTH-1:0], ssd\_temp);

ssd\_out d2(ssd\_temp[11:8] , HEX[2]);

ssd\_out d1(ssd\_temp[7:4] , HEX[1]);

ssd\_out d0(ssd\_temp[3:0] , HEX[0]);

always @(posedge clk)

begin

if(done) begin

done <= 0;

counter <= 1;

end

else if(~done & mode & op==3) begin

counter <= counter << 1;

done <= (counter == 1<<(DWIDTH<<1) );

end

else

done <= 1;

end

always @(posedge clk)

begin

result <= mode ? (op==0 ? dout\_adder

: op==1 ? dout\_subtractor

: op==2 ? dout\_multiplier

: op==3 ? dout\_divider

: op==4 ? dout\_comparator

: 0)

: (op==0 ? dout\_and

: op==1 ? dout\_or

: op==2 ? dout\_xor

: op==3 ? dout\_not

: op==4 ? dout\_shift

: op==5 ? dout\_ashift

: 0 );

end

always @(posedge done)

begin

dout <= result;

end

endmodule

// Verilog code for carry look-ahead adder

module adder #(parameter DWIDTH = 8)(in1, in2, carry\_in, sum, carry\_out);

input [DWIDTH - 1:0] in1;

input [DWIDTH - 1:0] in2;

input carry\_in;

output [DWIDTH - 1:0] sum;

output carry\_out;

//assign {carry\_out, sum} = in1 + in2 + carry\_in;

wire [DWIDTH - 1:0] gen;

wire [DWIDTH - 1:0] pro;

wire [DWIDTH:0] carry\_tmp;

genvar j, i;

generate

//assume carry\_tmp in is zero

assign carry\_tmp[0] = carry\_in;

//carry generator

for(j = 0; j < DWIDTH; j = j + 1) begin: carry\_generator

assign gen[j] = in1[j] & in2[j];

assign pro[j] = in1[j] | in2[j];

assign carry\_tmp[j+1] = gen[j] | pro[j] & carry\_tmp[j];

end

//carry out

assign carry\_out = carry\_tmp[DWIDTH];

//calculate sum

//assign sum[0] = in1[0] ^ in2 ^ carry\_in;

for(i = 0; i < DWIDTH; i = i+1) begin: sum\_without\_carry

assign sum[i] = in1[i] ^ in2[i] ^ carry\_tmp[i];

end

endgenerate

endmodule

module subtractor #(parameter DWIDTH = 8) (clk, din1, din2, dout\_subtractor);

input clk;

input [DWIDTH-1:0] din1;

input [DWIDTH-1:0] din2;

output [DWIDTH-1:0] dout\_subtractor;

reg [DWIDTH-1:0] dout\_adder\_temp\_ff;

adder add2((~din2),1,dout\_adder\_temp);

adder add3(din1,dout\_adder\_temp\_ff,dout\_subtractor);

always @(posedge clk)

begin

dout\_adder\_temp\_ff <= dout\_adder\_temp;

end

endmodule

`timescale 1ns/1ps

`define DELAY 10

// Verilog project: Verilog code for multiplier using carry-look-ahead adders

module multiplier #(parameter DWIDTH = 8) (multicand, multiplier, product1, product2);

input [DWIDTH-1:0] multicand;

input [DWIDTH-1:0] multiplier;

output [DWIDTH-1:0] product1;

output [DWIDTH-1:0] product2;

wire [(DWIDTH + DWIDTH - 1):0] product;

assign product1 = product[DWIDTH-1:0];

assign product2 = product[(DWIDTH<<1)-1:DWIDTH];

wire [DWIDTH - 1:0] multicand\_tmp [DWIDTH-1:0];

wire [DWIDTH - 1:0] product\_tmp [DWIDTH-1:0];

wire [DWIDTH -1:0] carry\_tmp;

genvar i, j;

generate

//initialize values

for(j = 0; j < DWIDTH; j = j + 1) begin: for\_loop\_j

assign multicand\_tmp[j] = multicand & {DWIDTH{multiplier[j]}};

end

assign product\_tmp[0] = multicand\_tmp[0];

assign carry\_tmp[0] = 1'b0;

assign product[0] = product\_tmp[0][0];

for(i = 1; i < DWIDTH; i = i + 1) begin: for\_loop\_i

adder #(.DATA\_WID(DWIDTH)) add1 (

// Outputs

.sum(product\_tmp[i]),

.carry\_out(carry\_tmp[i]),

// Inputs

.carry\_in(1'b0),

.in1(multicand\_tmp[i]),

.in2({carry\_tmp[i-1],product\_tmp[i-1][7-:7]}));

assign product[i] = product\_tmp[i][0];

end //end for loop

assign product[(DWIDTH+DWIDTH-1):DWIDTH] = {carry\_tmp[DWIDTH-1],product\_tmp[DWIDTH-1][7-:7]};

endgenerate

endmodule

`timescale 1ns/1ps

//`define DELAY #10

// Verilog code for carry look-ahead adder

module adder (in1, in2, carry\_in, sum, carry\_out);

parameter DATA\_WID = 8;

input [DATA\_WID - 1:0] in1;

input [DATA\_WID - 1:0] in2;

input carry\_in;

output [DATA\_WID - 1:0] sum;

output carry\_out;

//assign {carry\_out, sum} = in1 + in2 + carry\_in;

wire [DATA\_WID - 1:0] gen;

wire [DATA\_WID - 1:0] pro;

wire [DATA\_WID:0] carry\_tmp;

genvar j, i;

generate

//assume carry\_tmp in is zero

assign carry\_tmp[0] = carry\_in;

//carry generator

for(j = 0; j < DATA\_WID; j = j + 1) begin: carry\_generator

assign gen[j] = in1[j] & in2[j];

assign pro[j] = in1[j] | in2[j];

assign carry\_tmp[j+1] = gen[j] | pro[j] & carry\_tmp[j];

end

//carry out

assign carry\_out = carry\_tmp[DATA\_WID];

//calculate sum

//assign sum[0] = in1[0] ^ in2 ^ carry\_in;

for(i = 0; i < DATA\_WID; i = i+1) begin: sum\_without\_carry

assign sum[i] = in1[i] ^ in2[i] ^ carry\_tmp[i];

end

endgenerate

endmodule

// Verilog project: Verilog code for multiplier using carry look ahead adder

`define DELAY 10

module multiplier\_tb();

parameter DWIDTH = 8;

parameter DWIDTH = 8;

// /\*AUTOREGINPUT\*/

// Beginning of automatic reg inputs (for undeclared instantiated-module inputs)

reg [DWIDTH-1:0] multicand; // To mul1 of cla\_multiplier.v

reg [DWIDTH-1:0]multiplier; // To mul1 of cla\_multiplier.v

// End of automatics

/\*AUTOWIRE\*/

// Beginning of automatic wires (for undeclared instantiated-module outputs)

wire [(DWIDTH+DWIDTH-1):0]product;// From mul1 of cla\_multiplier.v

// End of automatics

multiplier mul1(/\*AUTOINST\*/

// // Outputs

.product1 (product[(DWIDTH-1):0]),

.product2 (product[(DWIDTH<<1)-1:DWIDTH]),

// // Inputs

.multicand (multicand[DWIDTH-1:0]),

.multiplier (multiplier[DWIDTH-1:0]));

integer i;

initial begin

// #(`DELAY) //correct

multicand = 8'hFF;

multiplier = 8'h7F;

#(`DELAY) //correct

multicand = 8'h80;

multiplier = 8'hF0;

#(`DELAY) //faila

multicand = 8'h80;

multiplier = 8'hF0;

#(`DELAY) //correct

multicand = 8'hF0;

multiplier = 8'hF7;

#(`DELAY) //correct

multicand = 8'hFF;

multiplier = 8'hFF;

end

initial begin

$dumpfile("dump.vcd"); $dumpvars;

end

endmodule

module divider #(parameter DWIDTH=8) (

input wire logic clk,

input wire logic start, // start signal

input wire logic [DWIDTH-1:0] x, // dividend

input wire logic [DWIDTH-1:0] y, // divisor

// output logic [DWIDTH-1:0] q, // quotient

// output logic [DWIDTH-1:0] r // remainder

output logic [(DWIDTH<<1)-1:0] dout

);

reg busy; // calculation in progress

reg valid; // quotient and remainder are valid

reg dbz; // divide by zero flag

reg [DWIDTH-1:0] q;

reg [DWIDTH-1:0] r;

assign dout = {q,r};

logic [DWIDTH-1:0] y1; // copy of divisor

logic [DWIDTH-1:0] q1, q1\_next; // intermediate quotient

logic [DWIDTH:0] ac, ac\_next; // accumulator (1 bit wider)

logic [clogb2(DWIDTH)-1:0] i; // iteration counter

always @(\*) begin

if (ac >= {1'b0,y1}) begin

ac\_next = ac - y1;

{ac\_next, q1\_next} = {ac\_next[DWIDTH-1:0], q1, 1'b1};

end else begin

{ac\_next, q1\_next} = {ac, q1} << 1;

end

end

always @(posedge clk) begin

if (start) begin

valid <= 0;

i <= 0;

if (y == 0) begin // catch divide by zero

busy <= 0;

dbz <= 1;

end else begin // initialize values

busy <= 1;

dbz <= 0;

y1 <= y;

{ac, q1} <= {{DWIDTH{1'b0}}, x, 1'b0};

end

end else if (busy) begin

if (i == DWIDTH-1) begin // we're done

busy <= 0;

valid <= 1;

q <= q1\_next;

r <= ac\_next[DWIDTH:1]; // undo final shift

end else begin // next iteration

i <= i + 1;

ac <= ac\_next;

q1 <= q1\_next;

end

end

end

function integer clogb2;

input [31:0] value;

integer i;

begin

clogb2 = 0;

for(i = 0; 2\*\*i < value; i = i + 1)

clogb2 = i + 1;

end

endfunction

endmodule

//Testbench for divider

module divider\_tb();

parameter CLK\_PERIOD = 10; // 10 ns == 100 MHz

parameter DWIDTH = 8;

logic clk;

logic start; // start signal

// logic busy; // calculation in progress

// logic valid; // quotient and remainder are valid

// logic dbz; // divide by zero flag

logic [DWIDTH-1:0] x; // dividend

logic [DWIDTH-1:0] y; // divisor

logic [(DWIDTH<<1)-1:0] dout; // quotient

// logic [DWIDTH-1:0] r; // remainder

divider #(.DWIDTH(DWIDTH)) divider\_inst (.\*);

always #(CLK\_PERIOD / 2) clk = ~clk;

initial begin

$monitor("\t%d:\t%d /%d =%d (r =) (V=) (DBZ=)",

$time, x, y, dout/\*, valid, dbz\*/);

end

initial begin

clk = 1;

#100 x = 4'b0000; // 0

y = 4'b0010; // 2

start = 1;

#10 start = 0;

#50 x = 4'b0010; // 2

y = 4'b0000; // 0

start = 1;

#10 start = 0;

#50 x = 4'b0111; // 7

y = 4'b0010; // 2

start = 1;

#10 start = 0;

#50 x = 4'b1111; // 15

y = 4'b0101; // 5

start = 1;

#10 start = 0;

#50 x = 4'b0001; // 1

y = 4'b0001; // 1

start = 1;

#10 start = 0;

#50 x = 4'b1000; // 8

y = 4'b1001; // 9

start = 1;

#10 start = 0;

// ...

#50 $finish;

end

initial begin

$dumpfile("div.vcd");

$dumpvars;

end

endmodule

module comparator #(parameter DWIDTH=8) (a,b,dout\_comparator);

input [DWIDTH-1:0] a,b;

output reg [DWIDTH-1:0] dout\_comparator;

always @(a or b)

begin

if(a==b)

dout\_comparator = 0;

else if (a>b)

dout\_comparator = 2;

else

dout\_comparator = 1;

end

endmodule

module and\_er #(parameter DWIDTH = 8) (din1,din2,dout\_and\_er);

input [DWIDTH-1:0] din1;

input [DWIDTH-1:0] din2;

output [DWIDTH-1:0] dout\_and\_er;

assign dout\_and\_er = din1 & din2;

endmodule

module or\_er #(parameter DWIDTH = 8) (din1,din2,dout\_or\_er);

input [DWIDTH-1:0] din1;

input [DWIDTH-1:0] din2;

output [DWIDTH-1:0] dout\_or\_er;

assign dout\_or\_er = din1|din2;

endmodule

module xor\_er #(parameter DWIDTH = 8)(din1,din2,dout\_xor\_er);

input [DWIDTH-1:0] din1;

input [DWIDTH-1:0] din2;

output [DWIDTH-1:0] dout\_xor\_er;

assign dout\_xor\_er = din1 | din2;

endmodule

module not\_er #(parameter DWIDTH = 8)(din1,dout\_not\_er);

input [DWIDTH-1:0] din1;

output [DWIDTH-1:0] dout\_not\_er;

assign dout\_not\_er = ~din1;

endmodule

module shift #(parameter DWIDTH = 8)(

input [DWIDTH-1:0] dir,

input [DWIDTH-1:0] din,

output [DWIDTH-1:0]dout

);

assign dout = dir ? din<<1

: din>>1;

endmodule

module ashift #(parameter DWIDTH = 8)(

input [DWIDTH-1:0] dir,

input [DWIDTH-1:0] din,

output [DWIDTH-1:0]dout

);

assign dout = dir ? {din[DWIDTH-1],din[DWIDTH-2:0]<<1}

: {din[DWIDTH-1],din[DWIDTH-2:0]>>1};

endmodule

`timescale 1ns / 1ps

module BCD\_convert #(parameter DWIDTH = 8)(

input [DWIDTH-1:0] bin\_in,

output reg [11:0] BCD\_out

);

integer i;

always @(bin\_in)

begin

BCD\_out = 0;

for (i = 0; i < DWIDTH+1; i = i + 1) //use a loop to shift the binary input the same number of times as it has number of bits. In this case, always 10.

begin

//Add 3 to any set representing a decimal output if the binary total is 5 or more to ensure proper carrying

if (BCD\_out[3:0] > 4)

BCD\_out[3:0] = BCD\_out[3:0] + 3;

if (BCD\_out[7:4] > 4)

BCD\_out[7:4] = BCD\_out[7:4] + 3;

if (BCD\_out[11:8] > 4)

BCD\_out[11:8] = BCD\_out[11:8] + 3;

BCD\_out = {BCD\_out[DWIDTH+1:0], bin\_in[DWIDTH+1 - i]};

end

end

endmodule

`timescale 1ns / 1ps

module display\_out(

input [3:0] LED\_BCD,

output reg [6:0] BCD\_display

);

always @(\*) //convert the binary decimal representation to a format for the seven segment display

begin

case (LED\_BCD)

4'b0000: BCD\_display = 7'b0000001; // out = 0

4'b0001: BCD\_display = 7'b1001111; // out = 1

4'b0010: BCD\_display = 7'b0010010; // out = 2

4'b0011: BCD\_display = 7'b0000110; // out = 3

4'b0100: BCD\_display = 7'b1001100; // out = 4

4'b0101: BCD\_display = 7'b0100100; // out = 5

4'b0110: BCD\_display = 7'b0100000; // out = 6

4'b0111: BCD\_display = 7'b0001111; // out = 7

4'b1000: BCD\_display = 7'b0000000; // out = 8

4'b1001: BCD\_display = 7'b0000100; // out = 9

4'b1010: BCD\_display = 7'b1111110; // out = -

4'b1011: BCD\_display = 7'b1111111; // out = no segments lit up

endcase

end

endmodule