CE/EE/EECT 6325

VLSI Design

Project #1

Arbitrary Digital Design:

**Arithmetic Logic Unit**

**With Code Converter**

Due Date: January 26, 2017

**Submitted by: Akshay S Landge (asl170130)**

**& Navya Sri Sreeram(nxs161131)**

**Submission Date: January 26, 2017**

**General Description:**

Arithmetic Logic Unit (ALU) with Code Converter is a combinational circuit that performs arithmetic and logical operations on unsigned binary numbers as well as converts these binary numbers into gray code, excess-3 code and binary coded decimals. All the operations can be performed on 16 bit binary numbers except for excess-3 code converter which can be performed only on 4 bit binary numbers. Since there are 16 operations and any one is to be selected we have a 16:1 multiplexer which enables to choose any one of the operations performed by the ALU with code converter which is stored in the flip-flops. In the entire design the flip-flop module is a sequential module which uses clock as its synchronizing input.

The design which we have proposed as a part of our project is scalable one that means the number of operations as well as the input bit size can be increased for higher levels of data.

*Limitations*: This ALU with code converter is a combinational circuit which works with only unsigned integer binary numbers and excess-3 code convertor will convert only inputs up to (12)10 i.e. (1100)2.

These above limitations will be removed in the later stages of semester as we progress with the project.

Report includes:

1. Block Diagram
2. Verilog Code
3. Test Bench
4. Simulation Waveform
5. **Block Diagram**

n1

n2

n3

q

clk

ALU with Code Converter

ALU

Code Converter

Add

Gray Code

Excess-3 Code

Binary Coded Decimal

Subtract

Multiplication

Division

Bitwise AND

Bitwise OR

Bitwise XOR

Increment

Bitwise NOT

Modular Division

Shift Left

Shift Right

Bitwise XNOR

Input Block

16:1

Mux

32bit

Register

1. **Verilog Code**

module code(gray, ex3,sum, cout, sub, csub, mul, div, rem, incre, lshift, rshift, b1, b2, b3, b4, b5, n1, n2, n3, clk, mux\_out, q, reset, select, o, t, h, th, tth, bcd);

input [15:0] n1,n2;

input clk;

input [3:0]select ;

input reset;

input [3:0] n3; //input for excess-3 code conversion can be from 0 to 12

output reg[3:0] ex3; //excess-3 is valid only for 4 bits

output reg[3:0] o, t, h, th, tth;

/\* o is BCD output in ones position

t is BCD output in tens position

h is BCD output in hundreds position

th is BCD output in thousands position

tth is BCD output in ten thousands position \*/

output reg[15:0] sum, sub, div, rem, incre, lshift, rshift, b1, b2, b3, b4, b5;

output reg[15:0] gray; // gray code output

output reg[31:0] mul, q, mux\_out, bcd;

output reg cout, csub;

reg [16:0] add,diff;

integer i,k;

reg [15:0] temp;

always @ (\*) begin

//arithmetic operations

// addition module

add= n1+n2;

sum= add [15:0];

cout= add [16];

// subtract module

diff= n1-n2; //considering n1 to be higher than n2

sub= diff [15:0];

csub= diff [16];

// multiply module

mul= n1\*n2;

//division module

div= n1/n2;

//modulor divsion

rem= n1%n2;

//increment module

incre= n1+1;

//logical shift operation

//left shift

lshift= n1<<1;

//right shift

rshift= n1>>1;

//bitwise operations

//bitwise and

b1= n1&n2;

//bitwise or

b2= n1|n2;

//bitwise xor

b3= n1^n2;

//bitwise xnor

b4= ~(n1^n2);

//bitwise not

b5= ~n1;

//code converters

//gray code converter module

gray[15] = n1[15];

for (i=14; i>=0; i=i-1) begin

gray[i] = n1[i+1]^n1[i];

end

//excess-3 code converter module

ex3 = n3+4'b0011;

end

//bcd code converter module

initial begin

o=4'b0000;

t=4'b0000;

h=4'b0000;

th=4'b0000;

tth=4'b0000;

k=0;

temp[15:0]=16'b0000000000000000;

end

always @ (\*) begin

temp[15:0]=n1[15:0];

end

always @ (temp) begin

for (k=0; k<16; k=k+1) begin

if (tth>4) begin

tth[3:0] = tth[3:0]+3; //add 3 if the number in ten thousands position is greater than 4

end

if (th>4) begin

th[3:0] = th[3:0]+3; //add 3 if the number in thousands position is greater than 4

end

if (h>4) begin

h[3:0] = h[3:0]+3; //add 3 if the number in hundreds position is greater than 4

end

if (t>4) begin

t[3:0] = t[3:0]+3; //add 3 if the number in tens position is greater than 4

end

if (o>4) begin

o[3:0] = o[3:0]+3; //add 3 if the number in ones position is greater than 4

end

tth = tth<<1;

tth[0] = th[3]; //shifting MSB of thousands position to LSB of ten thousands position

th = th<<1;

th[0] = h[3]; //shifting MSB of hundreds position to LSB of thousands position

h = h<<1;

h[0] = t[3]; //shifting MSB of tens position to LSB of hundreds position

t = t<<1;

t[0] = o[3]; //shifting MSB ones position to LSB of tens position

o = o<<1;

o[0]= temp[15]; //shifting MSB of the number n1 to LSB of ones position

temp = temp<<1;

end

bcd = {tth[3:0],th[3:0],h[3:0],t[3:0],o[3:0]};

end

// 16:1 multiplexer module

always @(\*)

case (select)

4'b 0000: begin mux\_out= sum; end

4'b 0001: begin mux\_out= sub; end

4'b 0010: begin mux\_out= mul; end

4'b 0011: begin mux\_out= div; end

4'b 0100: begin mux\_out= rem; end

4'b 0101: begin mux\_out= incre; end

4'b 0110: begin mux\_out= lshift; end

4'b 0111: begin mux\_out= rshift; end

4'b 1000: begin mux\_out= b1; end

4'b 1001: begin mux\_out= b2; end

4'b 1010: begin mux\_out= b3; end

4'b 1011: begin mux\_out= b4; end

4'b 1100: begin mux\_out= b5; end

4'b 1101: begin mux\_out= gray; end

4'b 1110: begin mux\_out= bcd; end

4'b 1111: begin mux\_out= ex3; end

endcase

//flip flop module

always @(posedge clk) begin

if (reset) begin

q= 0;

end

else begin

q= mux\_out;

end

end

endmodule

1. **Test Bench for BCD output**

module codetb;

// Inputs

reg [15:0] n1;

reg [15:0] n2;

reg [3:0] n3;

reg clk;

reg reset;

reg [3:0] select;

// Outputs

wire [15:0] gray;

wire [3:0] ex3;

wire [15:0] sum;

wire cout;

wire [15:0] sub;

wire csub;

wire [31:0] mul;

wire [15:0] div;

wire [15:0] rem;

wire [15:0] incre;

wire [15:0] lshift;

wire [15:0] rshift;

wire [15:0] b1;

wire [15:0] b2;

wire [15:0] b3;

wire [15:0] b4;

wire [15:0] b5;

wire [31:0] mux\_out;

wire [31:0] q;

wire [3:0] o;

wire [3:0] t;

wire [3:0] h;

wire [3:0] th;

wire [3:0] tth;

wire [31:0] bcd;

// Instantiate the Unit Under Test (UUT)

code uut (

.gray(gray),

.ex3(ex3),

.sum(sum),

.cout(cout),

.sub(sub),

.csub(csub),

.mul(mul),

.div(div),

.rem(rem),

.incre(incre),

.lshift(lshift),

.rshift(rshift),

.b1(b1),

.b2(b2),

.b3(b3),

.b4(b4),

.b5(b5),

.n1(n1),

.n2(n2),

.n3(n3),

.clk(clk),

.mux\_out(mux\_out),

.q(q),

.reset(reset),

.select(select),

.o(o),

.t(t),

.h(h),

.th(th),

.tth(tth),

.bcd(bcd)

);

initial begin

// Initialize Inputs

n1 = 16'b 0000110000000000;

n2 = 16'b 0000000000000001;

n3 = 4'b 0010;

clk = 0;

reset = 0;

select = 4'b 1110;

end

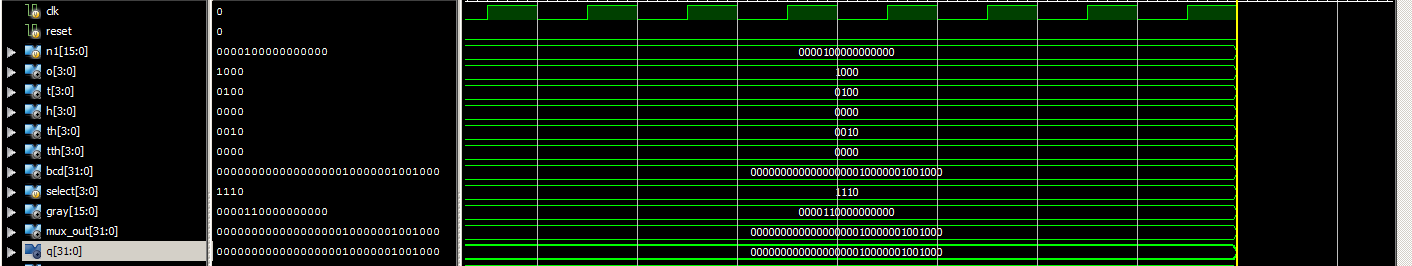
always begin

#10 clk=~clk;

end

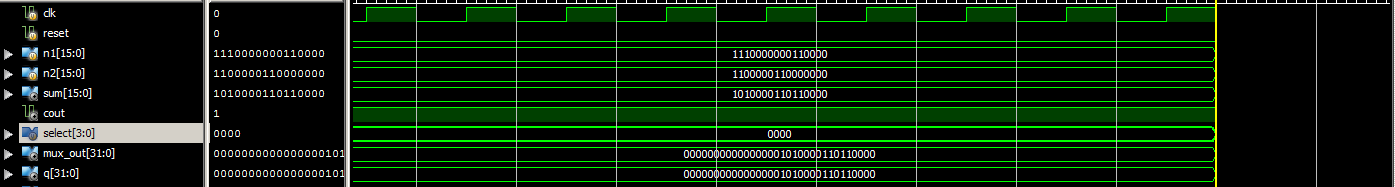
endmodule

1. **Simulation Waveforms**
2. Waveforms for conversion of Binary to BCD:



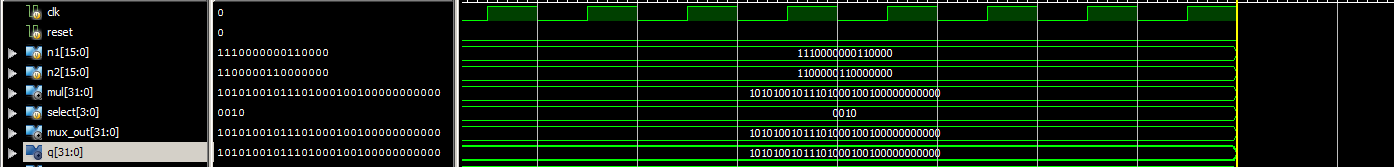
These waveforms give us the required BCD output which is the same in both the multiplexer output mux\_out and the flip-flop output q. Also, the individual BCD outputs of the input n1 is given by the waveforms of o, t, h, th and tth which represent the ones, tens, hundreds, thousands and ten thousands positions respectively.

1. Waveforms for Addition of 2 numbers:



The mux output waveform corresponds to the addition of two numbers n1 and n2. In this output the sum gives the final output which is then stored in the flip-flop by using a multiplexer. The cout bit represents the output carry bit for the addition operation.

1. Waveforms for Multiplication of 2 numbers:



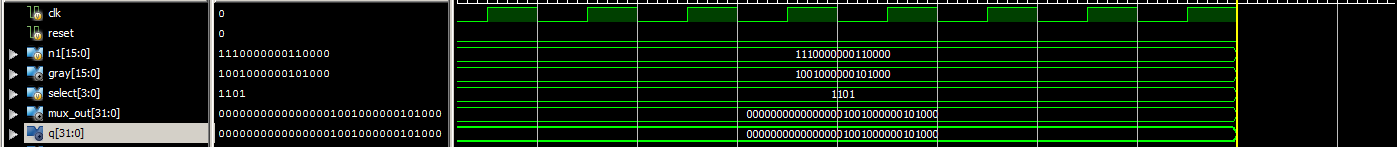
The diagram above shows the output of the multiplication operatoion which is n1\*n2. The multiplication of two 16-bit numbers gives a 32-bit number. Hence, the final output mul is a 32-bit number and can be selected using opcode 0010.

1. Waveform for Logical Bitwise AND Operation:



The waveforms b1 corresponds to the bitwise AND operation which can be chosen through the opcode 1000.

1. Waveform for Binary to Gray Code Conversion:



The waveform gray corresponds to the code conversion block where binary code is converted to a gray code.