For FPGA synthesis, the inputs and output of the 4-bit adder/subtractor must be connected as follows:

Mode input: Switch 0

1st value input: Switches 4:1

2nd value input: Switches 8:5

Sum/Difference output: LEDs 13:10

Carry output: LED14

Mystery V output signal: LED15

My implemented modules

module Board\_interface(

input sw0, //Is for the MODE (M)

input sw1,

input sw2,

input sw3,

input sw4,

input sw5,

input sw6,

input sw7,

input sw8,

output led0,

output led1,

output led2,

output led3,

output led4,

output led5,

output led6,

output led7,

output led8,

output led10,

output led11,

output led12,

output led13,

output led14,

output led15

);

//(sum, cout, x, y, m, v)

Lab4\_Part1 dut(sum, cout, {sw4,sw3,sw2,sw1}, {sw8,sw7,sw6,sw5}, {sw0}, v);

wire [3:0]sum;

wire cout, v;

**Part 0 – Review of Two’s Complement Numbers**

Two’s complement (“2C”) is the most common method of representing signed integers in digital logic. For an 8-bit number, unsigned range is 0 to 255 inclusive, while the 8-bit 2C range is from -128 to 127.

There are a few methods, but one of the easiest is to do two steps, 1st take the binary number and flip all the bits, EX: 00110011 => 11001100 then add 1 to the 1C value. EX: 11001100 + 1 = 11001101 = -128 + 77 = -55.

Problems – Convert these numbers into 2C format with 8-bit values.

1) -42

=> 00101010 => 11010101 +1 = **11010110**

=> -128 + 64 + 16 + 4 + 2 = **-42**

2) -122 => 01111010 => 10000101 + 1 = **10000110**

=> -128 + 4 + 2 = **-122**

3) +42

=> 00101010

4) -77

=> 01001101 => 10110010 + 1 = **10110011**

=> -128 + 32 + 16 + 2 + 1 = **-77**

**Part1 – 4-bit ripple carry adder/subtractor**

This part requires the use of the Basys3 to implement a 4-bit ripple carry adder/subtractor, that is, a module than can either add or subtract two 4-bit binary numbers.

Chart, box and whisker chart

Description automatically generated

The XOR gates ensures that either B or A may pass but not both, the “M” input has a double function, 1st flip the XOR bit and 2nd add 1-bit to C0, this functionality is exactly what I did before with the exercises above, 1st flip and 2nd add 1.

EENG2131 – Digital Systems LAB#4

Antonio Veguilla Hernandez Student ID: P000166790

**Part2 – DFF with async clear**

For this section a new module was required for a positive edge triggered flip flop with an asynchronous clear input. The clear input is active low. The testbench for this module was required to emulate the following waveform:

A picture containing music, keyboard

Description automatically generated

My implement modules

module Lab4\_Part3\_4(

input clk,

input d,

input clrN,

output q,

output qn

);

reg q;

always @(posedge clk or negedge clrN)

begin

if(~clrN)

begin

q <= 0;

end

else

begin

q <= d;

end

end

assign qn = ~q;

endmodule

module testbench\_Lab4\_Part3\_4;

reg clk, d, clrN;

wire q, qn;

Lab4\_Part3\_4 dut(clk, d, clrN, q, qn);

initial begin

d = 0;

clk = 1;

clrN = 1;

#8

d = 1;

clk = 1;

clrN = 1;

**Part 1 – 4-bit ripple carry adder/subtractor**

assign led0 = sw0; //Mode Input

assign led1 = sw1; //y LSB 1st bit

assign led2 = sw2; //y 2nd bit

assign led3 = sw3; //y 3rd bit

assign led4 = sw4; //y MSB 4th bit

assign led5 = sw5; //x LSB 1st bit

assign led6 = sw6; //x 2nd bit

assign led7 = sw7; //x 3rd bit

assign led8 = sw8; //x MSB 4th bit

assign led10 = sum[0];

assign led11 = sum[1];

assign led12 = sum[2];

assign led13 = sum[3];

assign led14 = cout;

assign led15 = v;

endmodule

module FullAdder(sum, cout, x, y, cin);

input x;

input y;

input cin;

output cout;

output sum;

wire e, f, g;

xor #10 (e, x, y);

xor #10 (sum, e, cin);

and #10 (f, e, cin);

and #10 (g, x, y);

or #10 (cout, f, g);

endmodule

module Lab4\_Part1 (sum, cout, x, y, m, v);

output [3:0] sum;

output cout, v;

input [3:0] x, y;

input m;

wire [3:1] c;

xor(d[0], x[0], m);

xor(d[1], x[1], m);

xor(d[2], x[2], m);

xor(d[3], x[3], m);

xor(v, c[3], cout);

wire [3:0]d;

FullAdder FA0 (sum[0], c[1], d[0], y[0], m); //sum, cout, x, y, cin

FullAdder FA1 (sum[1], c[2], d[1], y[1], c[1]);

FullAdder FA2 (sum[2], c[3], d[2], y[2], c[2]);

FullAdder FA3 (sum[3], cout, d[3], y[3], c[3]);

endmodule

**Part 2 – DFF with async clear**

#2

d = 1;

clk = 0;

clrN = 1;

#10

d = 1;

clk = 1;

clrN = 1;

#2

d = 0;

clk = 1;

clrN = 1;

#8

d = 0;

clk = 0;

clrN = 1;

#2

d = 1;

clk = 0;

clrN = 1;

#8

d = 1;

clk = 1;

clrN = 1;

#5

d = 1;

clk = 1;

clrN = 0;

#5

d = 1;

clk = 0;

clrN = 0;

#10

d = 1;

clk = 1;

clrN = 0;

#5

d = 0;

clk = 1;

clrN = 0;

#5

d = 0;

clk = 0;

clrN = 0;

#10

d = 0;

clk = 1;

clrN = 0;

**Part2 – DFF with async clear**

#5

d = 1;

clk = 1;

clrN = 0;

#5

d = 1;

clk = 0;

clrN = 0;

#10

d = 1;

clk = 1;

clrN = 0;

end

endmodule

**Part 3 – DFF with sync clear**

A new module was required to simulate the behavior of a D flip flop with a synchronous clear.

My implemented modules:

module DFFwithSyncClear(

input clk,

input d,

input syncClr,

output q,

output qn

);

reg q;

always @(posedge clk)

begin

if(~syncClr)

begin

q <= 0;

end

else

begin

q <= d;

end

end

assign qn = ~q;

endmodule

module testbench\_DFFwithSyncClear;

reg clk, d, syncClr;

wire q, qn;

DFFwithSyncClear dut(clk, d, syncClr, q, qn);

initial begin

d = 0;

clk = 1;

syncClr = 1;

#8

d = 1;

clk = 1;

syncClr = 1;

#2

d = 1;

clk = 0;

syncClr = 1;

#10

d = 1;

clk = 1;

syncClr = 1;

#2

d = 0;

clk = 1;

syncClr = 1;

**Part3 – DFF with sync clear**

#8

d = 0;

clk = 0;

syncClr = 1;

#2

d = 1;

clk = 0;

syncClr = 1;

#8

d = 1;

clk = 1;

syncClr = 0;

#5

d = 1;

clk = 1;

syncClr = 0;

#5

d = 1;

clk = 0;

syncClr = 0;

#10

d = 1;

clk = 1;

syncClr = 0;

#5

d = 0;

clk = 1;

syncClr = 0;

#5

d = 0;

clk = 0;

syncClr = 0;

#10

d = 0;

clk = 1;

syncClr = 0;

#5

d = 1;

clk = 1;

syncClr = 0;

#5

d = 1;

clk = 0;

syncClr = 0;

#10

d = 1;

clk = 1;

syncClr = 0;

end

endmodule

**Part 4 – Latch with async clear**

A new module was required to simulate the behavior of a negative latch (latch that is transparent when clock is low) with an asynchronous clear input.

My implemented modules:

module LatchWithAsyncClear(

input clk,

input d,

input AsyncClr,

output q,

output qn

);

reg q;

always @(clk or d or AsyncClr)

begin

if(~clk && AsyncClr)

begin

q <= d;

end

else if(~AsyncClr)

begin

q <= 0;

end

end

assign qn = ~q;

endmodule

module testbench\_LatchWithAsyncClear;

reg clk, d, AsyncClr;

wire q, qn;

LatchWithAsyncClear dut1(clk, d, AsyncClr, q, qn);

initial begin

d = 0;

clk = 1;

AsyncClr = 1;

#8

d = 1;

clk = 1;

AsyncClr = 1;

#2

d = 1;

clk = 0;

AsyncClr = 1;

#10

d = 1;

clk = 1;

AsyncClr = 1;

**Part4 – Latch with async clear**

#2

d = 0;

clk = 1;

AsyncClr = 1;

#8

d = 0;

clk = 0;

AsyncClr = 1;

#2

d = 1;

clk = 0;

AsyncClr = 1;

#8

d = 1;

clk = 1;

AsyncClr = 1;

#5

d = 1;

clk = 1;

AsyncClr = 0;

#5

d = 1;

clk = 0;

AsyncClr = 0;

#10

d = 1;

clk = 1;

AsyncClr = 0;

#5

d = 0;

clk = 1;

AsyncClr = 0;

#5

d = 0;

clk = 0;

AsyncClr = 0;

#10

d = 0;

clk = 1;

AsyncClr = 0;

#5

d = 1;

clk = 1;

AsyncClr = 0;

#5

d = 1;

clk = 0;

AsyncClr = 0;

**Part 4 – Latch with async clear**

#10

d = 1;

clk = 1;

AsyncClr = 0;

end

endmodule

**Conclusion:**

During this lab, I was able to learn and understand how the flip flops behave with the different settings. From my perspective, this was great to see it working, that helps me a lot to have a better and more deeper understanding of this type of systems. Also, the parts 0 & 1 were great too, that help me refresh the Two’s complement (2C) theory and how a 4-bit ripple adder/subtractor functions using the same principles of flipping the bits and then add 1 bit to make the subtract possible.

An appendix section was included to include the screenshot for parts 2, 3 & 4.

Appendix

Screenshot for Part2 – DFF with async clear simulation:

Graphical user interface, application

Description automatically generated

Screenshot for Part3 – DFF with sync clear simulation:

Graphical user interface, application

Description automatically generated

Screenshot for Part4 – DFF Latch with Async clear simulation:

Graphical user interface, application

Description automatically generated