

Simple 16-Bit Processor ALU

By Levi Randall and

James Samawi

EGCP 446 Fall 2020

**Abstract**

**A simplified structure of an Arithmetic logic unit (ALU) was proposed at the beginning of the Fall semester. Certain specifications were instructed like increasing the number of bits to be a 16-bit ALU design, as well as the addition of a non-Boothe’s multiplier circuit. We were able to fully program all aspects of the project circuit using Verilog within the Xilinx Vivado Integrated Developer Environment (IDE) as hardware modules at the gate level. Within this report, we will be discussing the specifications and challenges when it came to creating the Verilog design.**

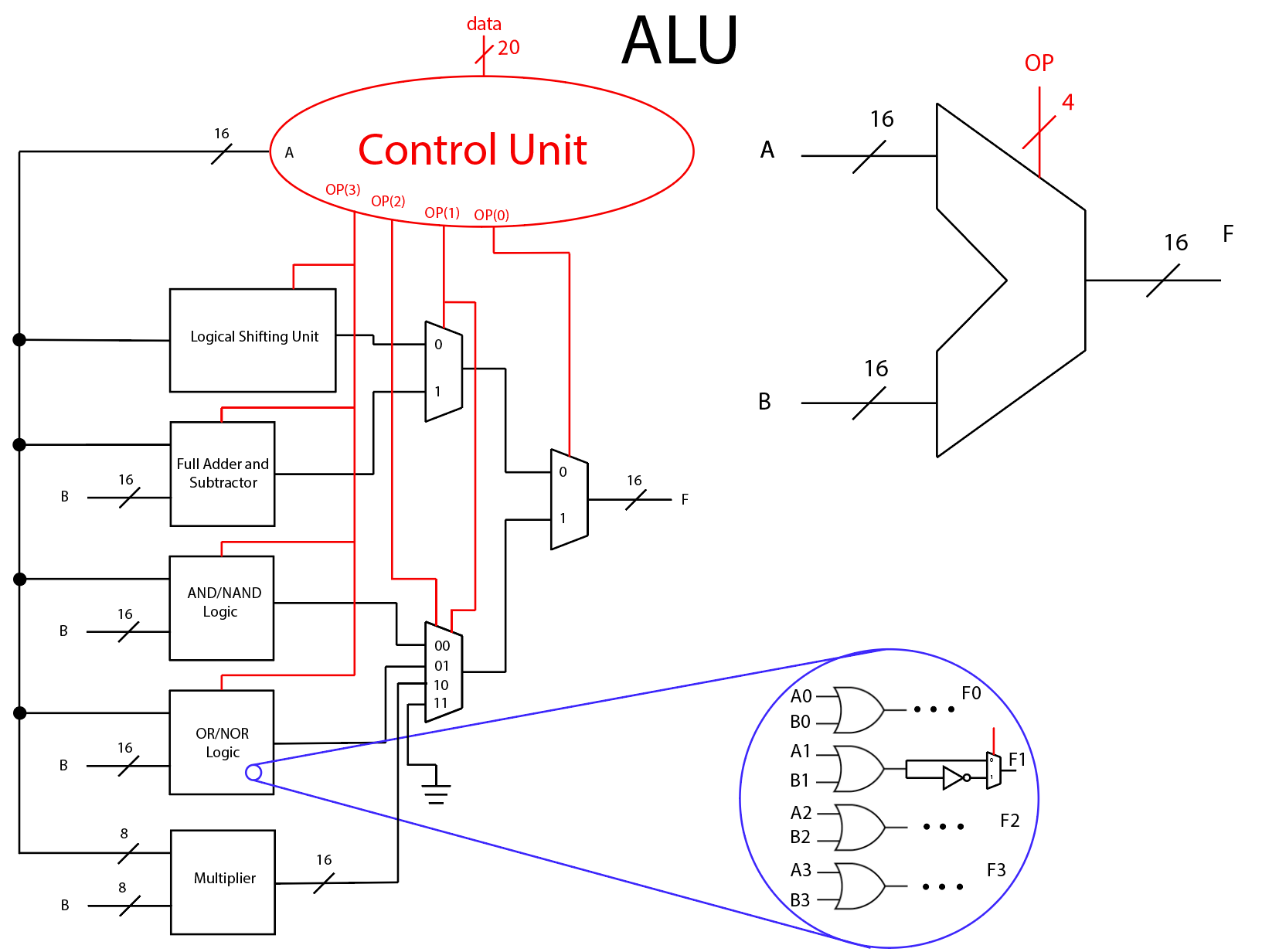
**Introduction**

Arithmetic logic unit (ALU) circuits are needed everywhere and are needed in most computers to do the bulk of the computer’s processes. The arithmetic and logical operations that the ALU can perform are essential for a computer to perform almost any task. In essence, an ALU is a structure of hardware primarily created using logic gates that perform these operations when given specific input signals or voltages. Along with these voltages, many gates come with a timing delay, meaning timing control is another very important aspect creating any large circuit. Furthermore, circuits such as multipliers can be more taxing on time in a circuit and must be taken account of when observing the output of a given system. All these circuits are linked together by multiplexers(mux(es))that control the observed output and the overall structure is orchestrated by an operation code or opcode given by an outside source of hardware typically summed up and designated as a Control Unit. Most ALU circuits can perform all the operations of Left/Right Shifting, Adding/Subtracting, AND/NAND, OR/NOR and for those with more advanced hardware Multiply.

**Specifications**

The inputs to the ALU are the 20-bit *data* and the 16-bit *B.* The *data* input comprises of a 4-bit opcode appended to the 16-bit operand 1.The *b* input acts as operand 2. The user must input both operands as binary numbers, and the binary opcode which determines the exact operation should also be included in the four leftmost bits in the 20-bit input. Shown in Fig. 1 below, our ALU specifications are the following:

* Logical Left Shift
* Logical Right Shift
* Full Adder
* Full Subtractor
* Binary multiplier
* Bitwise AND
* Bitwise NAND
* Bitwise OR
* Bitwise NOR



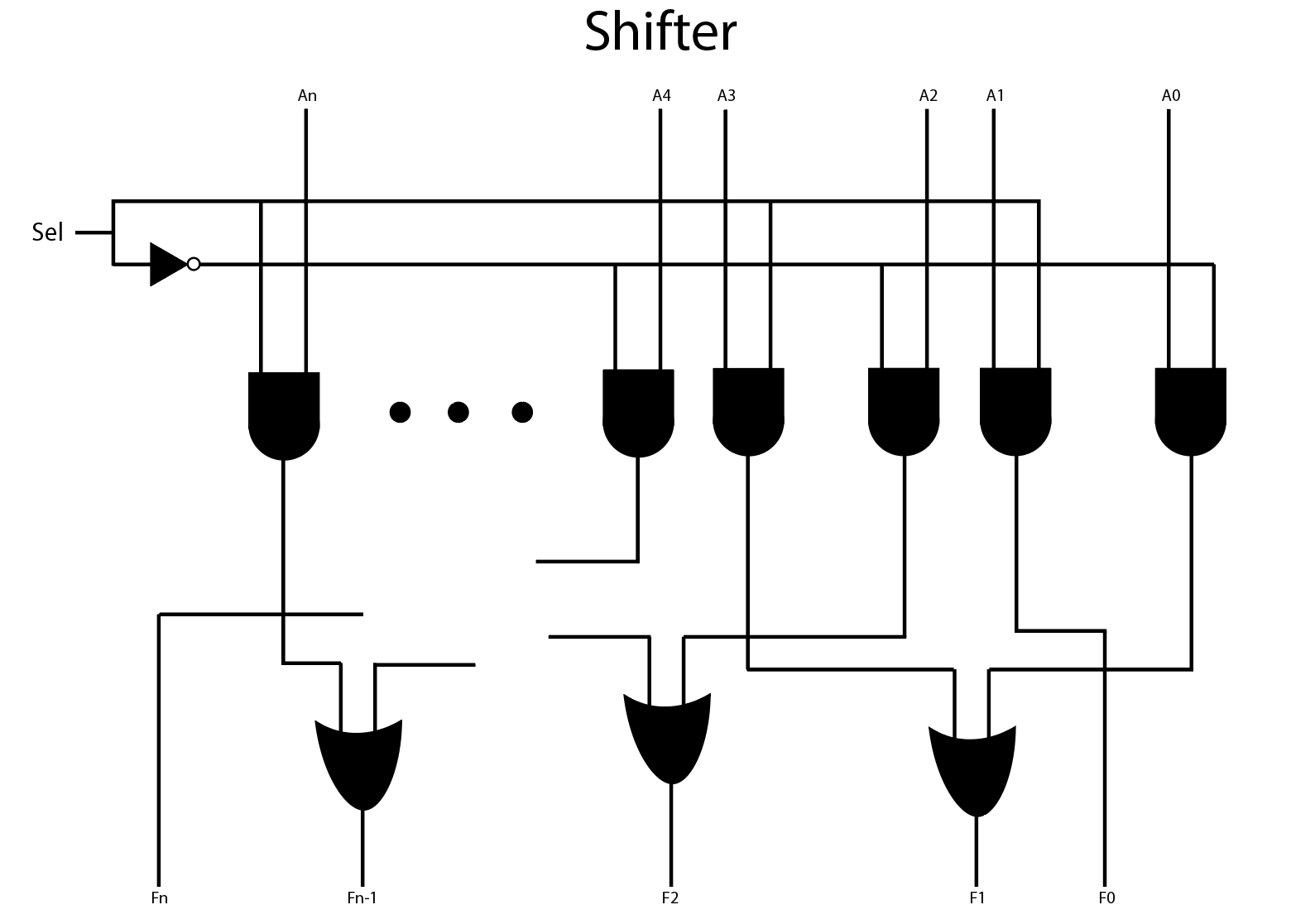
**Fig. 1) Overall Schematic of ALU Design**

**Procedure and Tools**

This project was completed using Xilinx Vivado as our IDE of choice. Verilog design files were used in unison with test benches for verification of hardware behavior. The ALU is implemented entirely in gate level modeling, which is amongst the lowest-level designs Verilog can offer (next to switch level modeling).

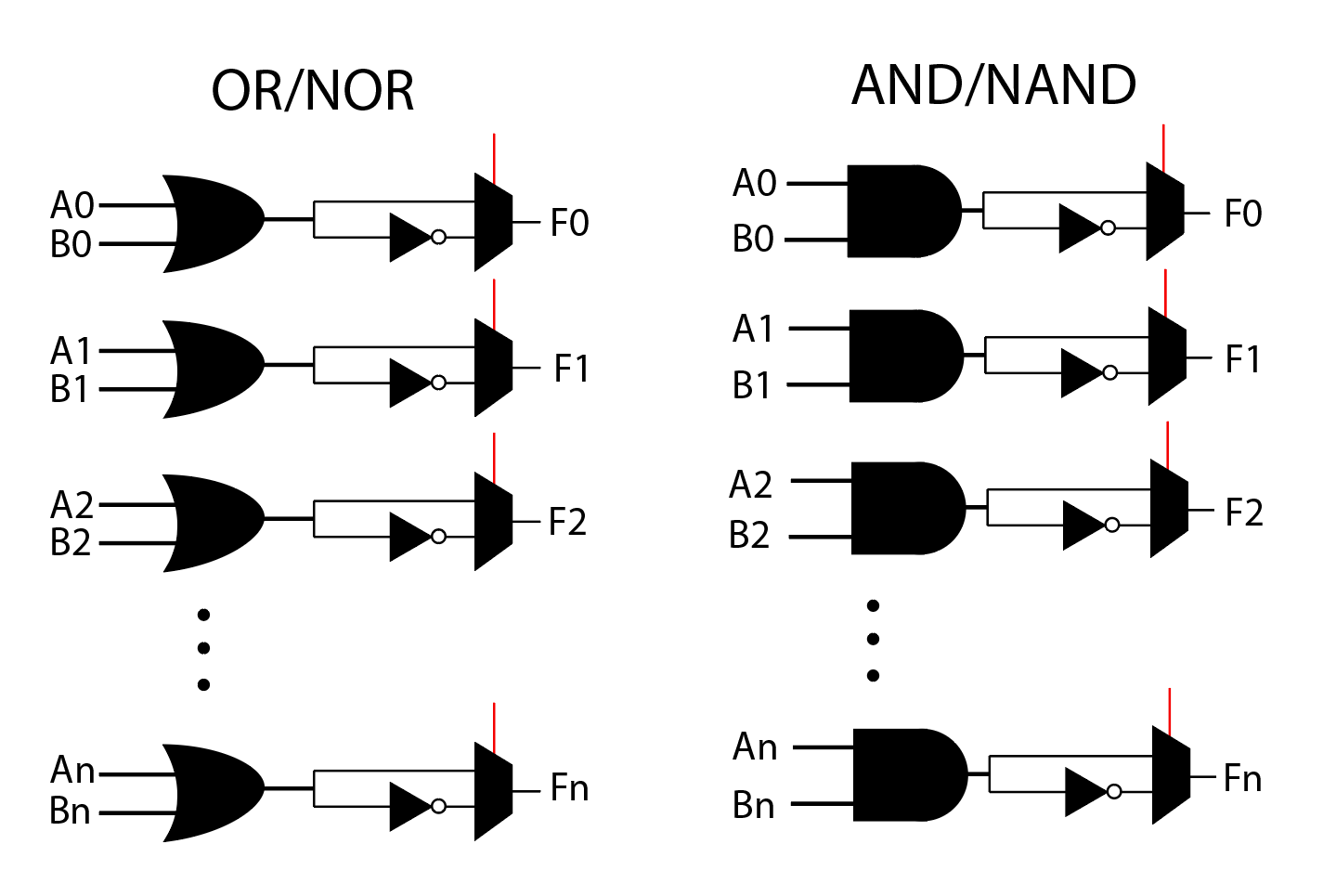
**Results and Discussion**

The Logical Shifter operator takes input *data* and will perform a logical left shift or a logical right shift to all the binary values of the operand, while the opcode determines the direction of shifting. We found two ways of implementing a bitwise logical shifter circuit in the gate level. The configuration in Fig. 2 is the schematic that we implemented, and this particular circuit saves two OR gates in the most significant bit and least significant bit outputs (bits 15 and 0 for our case). The cost of this is the confusing ordering of the AND gates input and how they wire to the proceeding OR gates, which was troublesome implementing in the Verilog design code. *Sel* in Fig. 2 is to be replaced with OP(3) as the fourth bit position in *data’s* opcode will determine the direction of single bit shifting by either a zero or a one in the leftmost bit position.



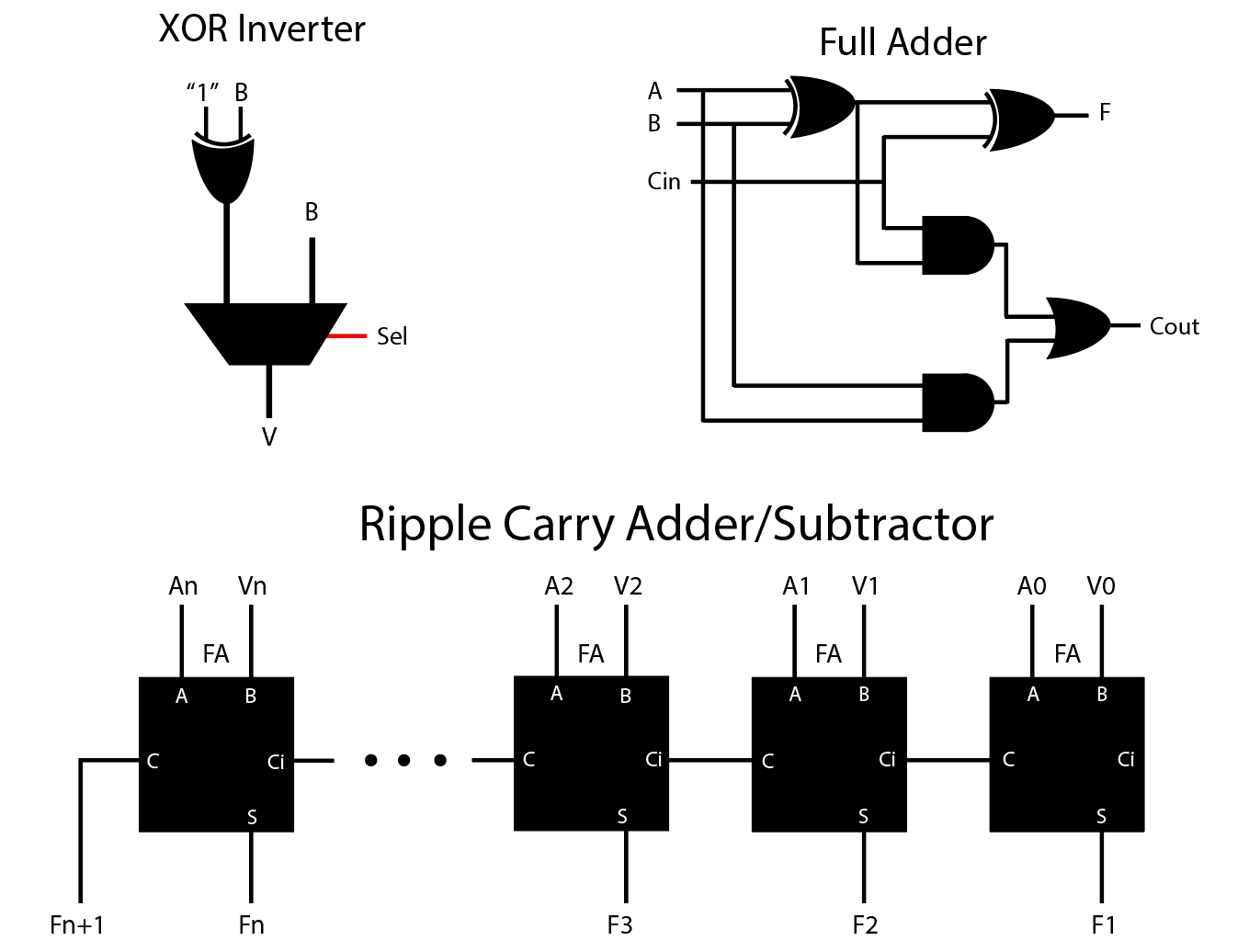
**Fig. 2) Logical Left/Right Shifter Unit**

Moving on to the elementary gates (AND, NAND, OR, NOR), in every way, these circuits were the easiest to create. Behaviorally, they only required a simple test bench to model their behavior. And lastly, the verification was simple as the output was an elementary gate operation. Structurally, they were simple with only a row of gates and a row of inverters being needed followed by a row of the already built 2-1 multiplexer circuits (Fig. 3). The addition of muxes proceeding each gate is a clever method of switching between a non-inverted gate (AND) and an inverted gate (NAND) without creating another inverted gate circuit from the ground-up. Operand one and two will undergo the bitwise gate operations, and the leftmost bit of the opcode will determine if the gate will be inverted or not by accepting the second input of the mux that contains a single inverter which inverts the output of the whole circuit.



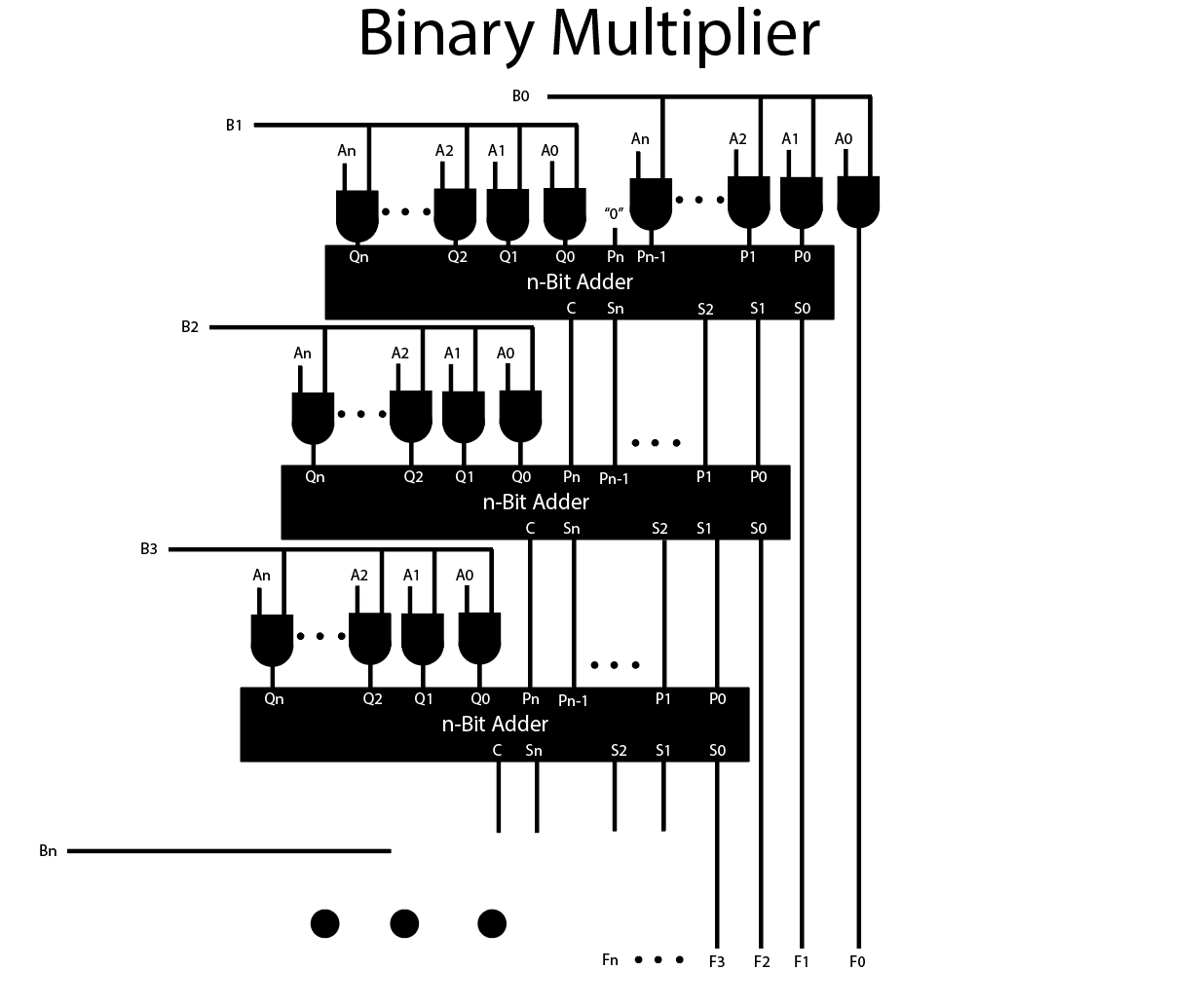
**Fig. 3) Elementary Gates Schematic and Mux Design**

A full adder module was implemented in Verilog in the gate level and was instantiated to create a 16-bit ripple carry adder which now accepts a 16-bit input and output. To implement the ripple carry adder, we decided to add the XOR module in Fig. 4 which inverts the input *B* which means the positive operand 1 is now adding a negative operand 2. This of course, is decided by the fourth position of opcode (in the place of *Sel*) and by its value of either a zero or a one.



**Fig. 4) Schematic of RCA and RCS with other Modules**

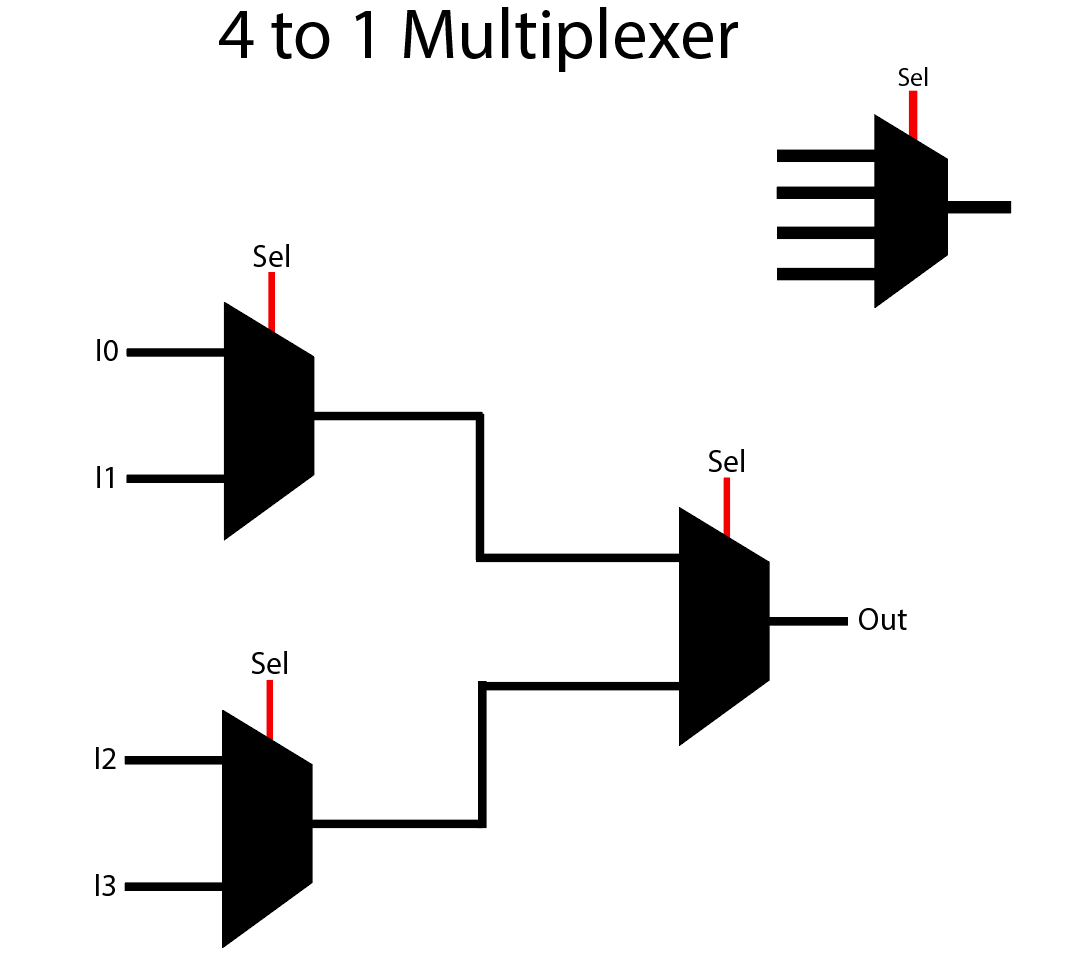
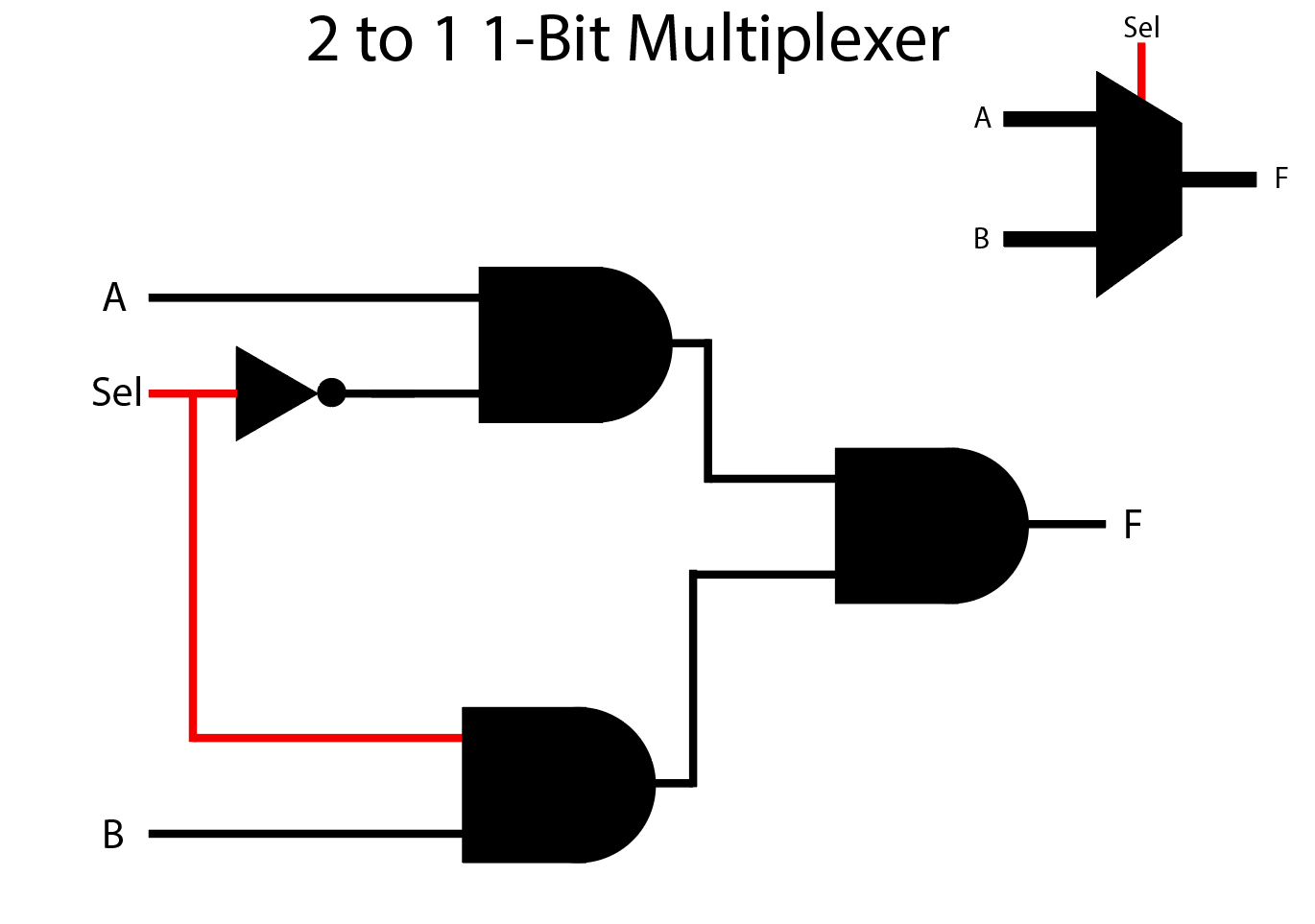
The binary multiplier module takes 8-bit input operands *A* and *B* and yields a 16-bit output without the influence of an opcode. The schematic in Fig. 5 shows the peculiar cascading design we went for. Bits of input *B* and the 8-bit *A* input undergo an AND operation and funneled through the 8-bit RCA and so on. This circuit was the most time consuming to program and consisted of the most gates (over sixty gates).



**Fig. 5) Binary Multiplier Schematic**

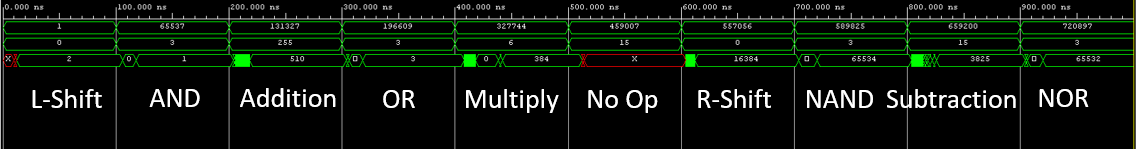
The 2-1 and 4-1 muxes (Fig. 6) work in conjunction with the Control Unit to pass information along and to ultimately decide which operator the user wishes to use with the given operands. There are three outermost muxes, consisting of two 2-1 muxes and a single 4-1 mux (Fig. 1). The 2-1 mux decides between logical shifting and RCA/RCS, while the 4-1 mux decides between the remaining three operators (with the fourth input of the mux grounded as it is not in use). The final 2-1 mux will decide between the previous muxes under the guidance of the user who specifies the LSB of opcode to be a zero or a one.

The 2-1 mux can also be found in two previous operator circuits. The elementary operators use a 2-1 mux to determine whether the output is inverted or not. The RCA/RCS circuit uses a 2-1 mux to determine whether the inverted operand 2 will be used to subtract or the non-inverted operand 2 will be used to add.



**Fig. 6) 2-1 Multiplexer (left) and 4-1 Multiplexer (right)**

Below is a snippet of the testbench waveform of the top-level Verilog module for the ALU (Fig. 7). The first row is the *data* input containing the opcode, the second row is input *B*, and the third and final row is the output. The data shown below is in decimal, although some values are better viewed in binary (such as *data* and logical shift operations). The No Op condition is when opcode = “1110” which means the fourth input of the 4-bit mux is selected, and since that is grounded, the output will be of an unknown value “X”. The outputs for every operation have been tested and yield correct results. Starting from the left of Fig. 7, L-shift gives 2 (10b) when operands are 1 and 0, AND gives 1 when operands are 1 and 3 (11b), Addition gives 510 when operands are 255 (11111111b) and 255, Multiply gives 384 when operands are 64 (1000000b) and 6, and so on.



**Fig. 7) ALU Top-Level Testbench w/ Control Unit**

**Future Work**

As for most Verilog/VHDL projects, it would not feel like a completed project without synthesizing and implementing the HDL code onto an FPGA. We foresee a project like this being implemented in an FPGA using buttons/switches for user input and 7-segment displays to show the output, almost like a calculator. A menu of some sort can be used to first gather user input (preferably in binary, since *data* contains the opcode deciding which operation to do). Once the user has finished entering the binary values of *data* and *B*, the user presses a button signifying they have finished, and then the 7-segment displays will display the answer in decimal.

Although, for a 16-bit output, there must be a total of five 7-segment displays, and most good FPGA boards come with four. Alternatives would be using external LEDs to display the output in binary, or using an LCD display to implement a proper menu, or to simply use five external 7-segment displays rather than internal ones. If the FPGA supports VGA or HDMI, there is possibility to display the output or menu on a monitor.

**Conclusion**

This simple 16-bit ALU and its Verilog gate level modeling came with its challenges, but in the end yielded correct results and the purpose was there and the goal was achieved. Modeling an ALU comes with the benefit of its versatility of computing arithmetic and logic operations within other Verilog projects, and is a necessary hardware module to possess.

**Appendix**

///ALU

`timescale 1ns / 1ps

module ALU(F,data,B);

output [15:0] F;

input [19:0] data; // A and OP Derive from Data

input [15:0] B;

wire [3:0] OP;

wire [15:0] A;

control unit(A,OP,data);

hardware toplevel(F,A,B,OP);

endmodule

//Control Unit

`timescale 1ns / 1ps

module control(A,OP,data);

output [3:0]OP;

output [15:0]A;

input [19:0]data;

assign OP[3:0] = data[19:16];

assign A = data[15:0];

endmodule

///Hardware

`timescale 1ns / 1ps

module hardware(F,A,B,OP);

output [15:0] F;

input [15:0] A;

input [15:0] B;

input [3:0] OP;

wire [15:0] shift\_out;

wire [15:0] rca\_out;

wire [15:0] and\_out;

wire [15:0] or\_out;

wire [15:0] multi\_out;

wire [15:0] m421\_out;

wire [15:0] m221\_out;

logic\_shift shifter(shift\_out,A,OP[3]);

RCA rca(rca\_out,A,B,OP[3]);

AND annd(and\_out,A,B,OP[3]);

OR oor(or\_out,A,B,OP[3]);

Multiplier multi(multi\_out,A[7:0],B[7:0]);

Mux4to1 m421(m421\_out,and\_out,or\_out,multi\_out,"0",OP[2:1]);

Mux2to1 m221a(m221\_out,shift\_out,rca\_out,OP[1]);

Mux2to1 m221b(F,m221\_out,m421\_out,OP[0]);

endmodule

/// Shifter

`timescale 1ns / 1ps

// this module shifts p with respect to select bit, it either shifts once left (s=0) or once right (s=1)

module logic\_shift(

output [15:0] r,

input [15:0] p,

input s,

wire [27:0] x

);

and #2(x[27],s,p[15]);

and #2(r[15],~s,p[14]);

and #2(x[25],s,p[14]);

and #2(x[26],~s,p[13]);

and #2(x[23],s,p[13]);

and #2(x[24],~s,p[12]);

and #2(x[21],s,p[12]);

and #2(x[22],~s,p[11]);

and #2(x[19],s,p[11]);

and #2(x[20],~s,p[10]);

and #2(x[17],s,p[10]);

and #2(x[18],~s,p[9]);

and #2(x[15],s,p[9]);

and #2(x[16],~s,p[8]);

and #2(x[13],s,p[8]);

and #2(x[14],~s,p[7]);

and #2(x[11],s,p[7]);

and #2(x[12],~s,p[6]);

and #2(x[9],s,p[6]);

and #2(x[10],~s,p[5]);

and #2(x[7],s,p[5]);

and #2(x[8],~s,p[4]);

and #2(x[5],s,p[4]);

and #2(x[6],~s,p[3]);

and #2(x[3],s,p[3]);

and #2(x[4],~s,p[2]);

and #2(x[1],s,p[2]);

and #2(x[2],~s,p[1]);

and #2(r[0],s,p[1]);

and #2(x[0],~s,p[0]);

or #2(r[14],x[27],x[26]);

or #2(r[13],x[25],x[24]);

or #2(r[12],x[23],x[22]);

or #2(r[11],x[21],x[20]);

or #2(r[10],x[19],x[18]);

or #2(r[9],x[17],x[16]);

or #2(r[8],x[15],x[14]);

or #2(r[7],x[13],x[12]);

or #2(r[6],x[11],x[10]);

or #2(r[5],x[9],x[8]);

or #2(r[4],x[7],x[6]);

or #2(r[3],x[5],x[4]);

or #2(r[2],x[3],x[2]);

or #2(r[1],x[1],x[0]);

endmodule

/// RCA

`timescale 1ns / 1ps

module RCA(

output[15:0] r,

input [15:0] p,

input [15:0] q,

input sel);

wire [15:0] c;

wire [15:0] x;

wire [15:0] y;

xor #2(x[0],q[0],"1");

xor #2(x[1],q[1],"1");

xor #2(x[2],q[2],"1");

xor #2(x[3],q[3],"1");

xor #2(x[4],q[4],"1");

xor #2(x[5],q[5],"1");

xor #2(x[6],q[6],"1");

xor #2(x[7],q[7],"1");

xor #2(x[8],q[8],"1");

xor #2(x[9],q[9],"1");

xor #2(x[10],q[10],"1");

xor #2(x[11],q[11],"1");

xor #2(x[12],q[12],"1");

xor #2(x[13],q[13],"1");

xor #2(x[14],q[14],"1");

xor #2(x[15],q[15],"1");

Mux2to1 ua(y,q,x,sel);

fulladder u0( r[0], c[0], p[0], y[0], sel);

fulladder u1( r[1], c[1], p[1], y[1], c[0]);

fulladder u2( r[2], c[2], p[2], y[2], c[1]);

fulladder u3( r[3], c[3], p[3], y[3], c[2]);

fulladder u4( r[4], c[4], p[4], y[4], c[3]);

fulladder u5( r[5], c[5], p[5], y[5], c[4]);

fulladder u6( r[6], c[6], p[6], y[6], c[5]);

fulladder u7( r[7], c[7], p[7], y[7], c[6]);

fulladder u8( r[8], c[8], p[8], y[8], c[7]);

fulladder u9( r[9], c[9], p[9], y[9], c[8]);

fulladder u10(r[10],c[10], p[10],y[10],c[9]);

fulladder u11(r[11],c[11],p[11],y[12],c[10]);

fulladder u12(r[12],c[12],p[12],y[12],c[11]);

fulladder u13(r[13],c[13],p[13],y[13],c[12]);

fulladder u14(r[14],c[14],p[14],y[14],c[13]);

fulladder u15(r[15],c[15],p[15],y[15],c[14]);

endmodule

///Full Adder

`timescale 1ns / 1ps

module fulladder(r,cout,p,q,cin);

input p;

input q;

input cin;

output cout;

output r;

wire x,y,z;

xor #2(x,p,q);

xor #2(r,x,cin);

and #2(y,x,cin);

and #2(z,p,q);

or #2(cout, y,z);

endmodule

/// Multiplier

`timescale 1ns / 1ps

module Multiplier(r,p,q);

input [7:0] p;

input [7:0] q;

output [15:0] r;

wire [7:0] x1;

wire [7:0] x2;

wire [8:0] x3;

wire [7:0] x4;

wire [8:0] x5;

wire [7:0] x6;

wire [8:0] x7;

wire [7:0] x8;

wire [8:0] x9;

wire [7:0] x10;

wire [8:0] x11;

wire [7:0] x12;

wire [8:0] x13;

wire [7:0] x14;

wire [8:0] x15;

and #2 (r[0],p[0],q[0]); // Output Goes Straight to the End

and #2(x1[0],p[1],q[0]);

and #2(x1[1],p[2],q[0]);

and #2(x1[2],p[3],q[0]);

and #2(x1[3],p[4],q[0]);

and #2(x1[4],p[5],q[0]);

and #2(x1[5],p[6],q[0]);

and #2(x1[6],p[7],q[0]);

assign x1[7] = 0;

and #2(x2[0],p[0],q[1]);

and #2(x2[1],p[1],q[1]);

and #2(x2[2],p[2],q[1]);

and #2(x2[3],p[3],q[1]);

and #2(x2[4],p[4],q[1]);

and #2(x2[5],p[5],q[1]);

and #2(x2[6],p[6],q[1]);

and #2(x2[7],p[7],q[1]);

Bit8Adder u1(x3[8],x3[7:0], x1,x2);

assign r[1] = x3[0];

and #2(x4[0],p[0],q[2]);

and #2(x4[1],p[1],q[2]);

and #2(x4[2],p[2],q[2]);

and #2(x4[3],p[3],q[2]);

and #2(x4[4],p[4],q[2]);

and #2(x4[5],p[5],q[2]);

and #2(x4[6],p[6],q[2]);

and #2(x4[7],p[7],q[2]);

Bit8Adder u2(x5[8],x5[7:0],x3[8:1], x4);

assign r[2] = x5[0];

and #2(x6[0],p[0],q[3]);

and #2(x6[1],p[1],q[3]);

and #2(x6[2],p[2],q[3]);

and #2(x6[3],p[3],q[3]);

and #2(x6[4],p[4],q[3]);

and #2(x6[5],p[5],q[3]);

and #2(x6[6],p[6],q[3]);

and #2(x6[7],p[7],q[3]);

Bit8Adder u3(x7[8],x7[7:0],x5[8:1], x6);

assign r[3] = x7[0];

and #2(x8[0],p[0],q[4]);

and #2(x8[1],p[1],q[4]);

and #2(x8[2],p[2],q[4]);

and #2(x8[3],p[3],q[4]);

and #2(x8[4],p[4],q[4]);

and #2(x8[5],p[5],q[4]);

and #2(x8[6],p[6],q[4]);

and #2(x8[7],p[7],q[4]);

Bit8Adder u4(x9[8],x9[7:0],x7[8:1], x8);

assign r[4] = x9[0];

and #2(x10[0],p[0],q[5]);

and #2(x10[1],p[1],q[5]);

and #2(x10[2],p[2],q[5]);

and #2(x10[3],p[3],q[5]);

and #2(x10[4],p[4],q[5]);

and #2(x10[5],p[5],q[5]);

and #2(x10[6],p[6],q[5]);

and #2(x10[7],p[7],q[5]);

Bit8Adder u5(x11[8],x11[7:0],x9[8:1], x10);

assign r[5] = x11[0];

and #2(x12[0],p[0],q[6]);

and #2(x12[1],p[1],q[6]);

and #2(x12[2],p[2],q[6]);

and #2(x12[3],p[3],q[6]);

and #2(x12[4],p[4],q[6]);

and #2(x12[5],p[5],q[6]);

and #2(x12[6],p[6],q[6]);

and #2(x12[7],p[7],q[6]);

Bit8Adder u6(x13[8],x13[7:0],x11[8:1], x12);

assign r[6] = x13[0];

and #2(x14[0],p[0],q[7]);

and #2(x14[1],p[1],q[7]);

and #2(x14[2],p[2],q[7]);

and #2(x14[3],p[3],q[7]);

and #2(x14[4],p[4],q[7]);

and #2(x14[5],p[5],q[7]);

and #2(x14[6],p[6],q[7]);

and #2(x14[7],p[7],q[7]);

Bit8Adder u7(x15[8],x15[7:0],x13[8:1], x14);

assign r[7] = x15[0];

assign r[15:8] = x15[8:1];

endmodule

/// Mux4to1

`timescale 1ns / 1ps

module Mux4to1(

output [15:0] out,

input [15:0] a,

input [15:0] b,

input [15:0] c,

input [15:0] d,

input [1:0] sel);

wire [15:0]x;

wire [15:0]y;

Mux2to1 ua(x,a,b,sel[0]);

Mux2to1 ub(y,c,d,sel[0]);

Mux2to1 uc(out,x,y,sel[1]);

endmodule

/// Mux2to1

`timescale 1ns / 1ps

module Mux2to1(out, a, b, sel);

output[15:0] out;

input [15:0] a;

input [15:0] b;

input sel;

wire [15:0] selextend;

wire [15:0] x;

wire [15:0] y;

wire [15:0] z;

assign selextend = {16{sel}};

nand #2(x[0],b[0],selextend[0]);

nand #2(x[1],b[1],selextend[1]);

nand #2(x[2],b[2],selextend[2]);

nand #2(x[3],b[3],selextend[3]);

nand #2(x[4],b[4],selextend[4]);

nand #2(x[5],b[5],selextend[5]);

nand #2(x[6],b[6],selextend[6]);

nand #2(x[7],b[7],selextend[7]);

nand #2(x[8],b[8],selextend[8]);

nand #2(x[9],b[9],selextend[9]);

nand #2(x[10],b[10],selextend[10]);

nand #2(x[11],b[11],selextend[11]);

nand #2(x[12],b[12],selextend[12]);

nand #2(x[13],b[13],selextend[13]);

nand #2(x[14],b[14],selextend[14]);

nand #2(x[15],b[15],selextend[15]);

not #2(z[0],selextend[0]);

not #2(z[1],selextend[1]);

not #2(z[2],selextend[2]);

not #2(z[3],selextend[3]);

not #2(z[4],selextend[4]);

not #2(z[5],selextend[5]);

not #2(z[6],selextend[6]);

not #2(z[7],selextend[7]);

not #2(z[8],selextend[8]);

not #2(z[9],selextend[9]);

not #2(z[10],selextend[10]);

not #2(z[11],selextend[11]);

not #2(z[12],selextend[12]);

not #2(z[13],selextend[13]);

not #2(z[14],selextend[14]);

not #2(z[15],selextend[15]);

nand #2(y[0],a[0],z[0]);

nand #2(y[1],a[1],z[1]);

nand #2(y[2],a[2],z[2]);

nand #2(y[3],a[3],z[3]);

nand #2(y[4],a[4],z[4]);

nand #2(y[5],a[5],z[5]);

nand #2(y[6],a[6],z[6]);

nand #2(y[7],a[7],z[7]);

nand #2(y[8],a[8],z[8]);

nand #2(y[9],a[9],z[9]);

nand #2(y[10],a[10],z[10]);

nand #2(y[11],a[11],z[11]);

nand #2(y[12],a[12],z[12]);

nand #2(y[13],a[13],z[13]);

nand #2(y[14],a[14],z[14]);

nand #2(y[15],a[15],z[15]);

nand #2(out[0], x[0],y[0]);

nand #2(out[1], x[1],y[1]);

nand #2(out[2], x[2],y[2]);

nand #2(out[3], x[3],y[3]);

nand #2(out[4], x[4],y[4]);

nand #2(out[5], x[5],y[5]);

nand #2(out[6], x[6],y[6]);

nand #2(out[7], x[7],y[7]);

nand #2(out[8], x[8],y[8]);

nand #2(out[9], x[9],y[9]);

nand #2(out[10], x[10],y[10]);

nand #2(out[11], x[11],y[11]);

nand #2(out[12], x[12],y[12]);

nand #2(out[13], x[13],y[13]);

nand #2(out[14], x[14],y[14]);

nand #2(out[15], x[15],y[15]);

endmodule

/// OR

`timescale 1ns / 1ps

module OR(r, p, q, sel);

output[15:0] r;

input [15:0] p;

input [15:0] q;

input sel;

wire [15:0]x;

wire [15:0]y;

or #2(x[0], p[0], q[0]);

or #2(x[1], p[1], q[1]);

or #2(x[2], p[2], q[2]);

or #2(x[3], p[3], q[3]);

or #2(x[4], p[4], q[4]);

or #2(x[5], p[5], q[5]);

or #2(x[6], p[6], q[6]);

or #2(x[7], p[7], q[7]);

or #2(x[8], p[8], q[8]);

or #2(x[9], p[9], q[9]);

or #2(x[10], p[10], q[10]);

or #2(x[11], p[11], q[11]);

or #2(x[12], p[12], q[12]);

or #2(x[13], p[13], q[13]);

or #2(x[14], p[14], q[14]);

or #2(x[15], p[15], q[15]);

not #2(y[0], x[0]);

not #2(y[1], x[1]);

not #2(y[2], x[2]);

not #2(y[3], x[3]);

not #2(y[4], x[4]);

not #2(y[5], x[5]);

not #2(y[6], x[6]);

not #2(y[7], x[7]);

not #2(y[8], x[8]);

not #2(y[9], x[9]);

not #2(y[10], x[10]);

not #2(y[11], x[11]);

not #2(y[12], x[12]);

not #2(y[13], x[13]);

not #2(y[14], x[14]);

not #2(y[15], x[15]);

Mux2to1 inversion(r,x,y,sel);

endmodule

/// AND

`timescale 1ns / 1ps

module AND(r, p, q, sel);

output[15:0] r;

input [15:0] p;

input [15:0] q;

input sel;

wire [15:0]x;

wire [15:0]y;

and #2(x[0], p[0], q[0]);

and #2(x[1], p[1], q[1]);

and #2(x[2], p[2], q[2]);

and #2(x[3], p[3], q[3]);

and #2(x[4], p[4], q[4]);

and #2(x[5], p[5], q[5]);

and #2(x[6], p[6], q[6]);

and #2(x[7], p[7], q[7]);

and #2(x[8], p[8], q[8]);

and #2(x[9], p[9], q[9]);

and #2(x[10], p[10], q[10]);

and #2(x[11], p[11], q[11]);

and #2(x[12], p[12], q[12]);

and #2(x[13], p[13], q[13]);

and #2(x[14], p[14], q[14]);

and #2(x[15], p[15], q[15]);

not #2(y[0], x[0]);

not #2(y[1], x[1]);

not #2(y[2], x[2]);

not #2(y[3], x[3]);

not #2(y[4], x[4]);

not #2(y[5], x[5]);

not #2(y[6], x[6]);

not #2(y[7], x[7]);

not #2(y[8], x[8]);

not #2(y[9], x[9]);

not #2(y[10], x[10]);

not #2(y[11], x[11]);

not #2(y[12], x[12]);

not #2(y[13], x[13]);

not #2(y[14], x[14]);

not #2(y[15], x[15]);

Mux2to1 inversion(r,x,y,sel);

endmodule

/// TB ALU

`timescale 1ns / 1ps

module tb\_ALU;

// Inputs

reg [19:0] data;

reg [15:0] B;

// Outputs

wire [15:0] F;

ALU dut( .F(F),.data(data), .B(B));

initial begin

data =20'b00000000000000000001; //LShift

B =16'b0000000000000000;

#100

data =20'b00010000000000000001; // AND

B =16'b0000000000000011;

#100

data =20'b00100000000011111111; // ADD

B =16'b0000000011111111;

#100

data =20'b00110000000000000001; // OR

B =16'b0000000000000011;

#100

data =20'b01010000000001000000; // Multi

B =16'b0000000000000110;

#100

data =20'b01110000000011111111; // 'X'

B =16'b0000000000001111;

#100

data =20'b10001000000000000000; // RShift

B =16'b0000000000000000;

#100

data =20'b10010000000000000001; // NAND

B =16'b0000000000000011;

#100

data =20'b10100000111100000000; // Sub

B =16'b0000000000001111;

#100

data =20'b10110000000000000001; // NOR

B =16'b0000000000000011;

#100

#1000 $finish;

end

endmodule

// For OP code set first 4 bits of data

// LShift = 0000 AND = 0001

// Add = 0010 OR = 0011

// LShift = 0100 Multi = 0101

// Add = 0110 'X' = 0111

// RShift = 1000 NAND = 1001

// Sub = 1010 NOR = 1011

// RShift = 1100 Multi = 1101

// Sub = 1110 '0' = 1111

// Last 16 Bits are A input

// Template

// #20 data =20'b0000 0000 0000 0000 0000;

// #20 B =16'b0000 0000 0000 0000;