ECPE 173 – Spring 2013
Lab Project 3

For the next set of lab projects, we will use a Java simulation environment developed at MIT for a course called 6.004. This simulator will allow us to design hardware without having to consider details at the transistor level or limit us to FPGAs. Additionally, the use of Java means that it should be possible to run the simulator on all computing platforms as long as you have Java Runtime Environment installed.

The simulation environment is JSIM, which models hardware from the transistor level up. It provides both gate and timing level simulations. While in some ways not as complex, feature-rich, nor flashy, it is similar in functionality to Quartus. It, however, has a cleaner and more user-friendly simulation environment than Quartus/ModelSim.

Our Sakai site has documentation on and the software for JSIM. Begin the project by reading through the user manual and reviewing the quick start instructions (ignore all references to the Athena computation system at MIT). Create a folder for your work in your U:/ drive. Create a sub-folder and download all of the library files from the library folder on Sakai. Download *jsim.jar* to the top-level work folder. Double-click on the file to open the simulation environment and explore its features.

We will be building an ALU in this lab project, which will form the central component of our processor that we will construct in future labs. Our ALU will accept two 32-bit, twos-complement inputs (A and B) and return a 32-bit output (Y). In building this design, we will use hierarchical design practices and develop different logic for each function we want our design to support, combining them together at the end. As this will be part of a processor eventually, we will design control logic optimized for the ALU and, in a future lab, connect that to the overall control system of our processor. Thus, we will design ALU function codes for the control independent of our instruction format. Our ALU will compute the operations based on ALUFN as outlined in Table 1.

ALUFN[4:0]	Operation	Output value Y[31:0]
1*abcd*	Bitwise Boolean	Y[i] = F_{abcd}(A[i], B[i])
00000	32-bit ADD	Y = A+B
00001	32-bit SUBTRACT	Y = A-B
00010	32-bit MULTIPLY (optional)	Y = A*B
00101	CMPEQ	Y = (A == B)
00111	CMPLT	Y = (A < B)
01101	CMPLE	Y = (A <= B)
01000	Shift left (SHL)	Y = A << B
01001	Shift right (SHR)	Y = A >> B
01011	Shift right, sign extended (SRA)	Y = A >> B (sign extended)

Table 1: Operations and Control Codes

Table 2 defines the Boolean operations further.

Operation	*ALUFN[3:0]*
AND	1000
OR	1110
XOR	0110
"A"	1010

Table 2: Boolean Operations

1. We will start by creating the overall structure of our ALU with dummy components. Create a new folder for this lab. Download the checkoff files from Sakai into that folder. While in that folder, create a new file called *lab3.jsim* (you can use either the text editor in JSIM or of your choosing). Paste in the following code:

```
.include "../6004/nominal.jsim"
.include "../6004/stdcell.jsim"
.include "lab3_test_bool.jsim"

.subckt BOOL alufn[3:0] A[31:0] B[31:0] OUT[31:0]
* * *xdummy OUT[31:0] constant0
Xmux4 A[31:0] B[31:0] alufn[0]#32 alufn[1]#32 alufn[2]#32 alufn[3]#32 OUT[31:0]
mux4
.ends

.subckt ARITH alufn[1:0] A[31:0] B[31:0] OUT[31:0] Z V N
xdummy OUT[31:0] Z V N constant0
.ends

.subckt SHIFT alufn[1:0] A[31:0] B[31:0] OUT[31:0]
xdummy OUT[31:0] constant0
.ends

.subckt CMP alufn3 alufn1 Z V N OUT[31:0]
xdummy OUT[31:0] constant0
.ends

.subckt alu alufn[4:0] a[31:0] b[31:0] out[31:0] z v n

* * * Generate outputs from each of BOOL, SHIFT, ARITH, CMP subcircuits:
xbool alufn[3:0] a[31:0] b[31:0] boolout[31:0] BOOL
xshift alufn[1:0] a[31:0] b[31:0] shiftout[31:0] SHIFT
xarith alufn[1:0] a[31:0] b[31:0] arithout[31:0] z v n ARITH
xcmp alufn[3] alufn[1] z v n cmpout[31:0] CMP

* * * Combine them, using three multiplexors:
xmux1 alufn[4]#32 nonbool[31:0] boolout[31:0] out[31:0] mux2
xmux2 alufn[2]#32 arithshift[31:0] cmpout[31:0] nonbool[31:0] mux2
xmux3 alufn[3]#32 arithout[31:0] shiftout[31:0] arithshift[31:0] mux2
.ends
```

The first two includes define the cell library for the underlying transistors. You downloaded these files earlier. Modify the paths to match the file locations. The

third include is for the checkoffs of your design, which you just downloaded into your working directory; make sure this path points to this first file. Run a gate-level simulation of the file and examine the results. Make sure you understand this code and the JSIM environment.

2. Design the Boolean logic first by replacing the dummy logic in the BOOL subcircuit. Consider Table 2 – how does ALUFN relate to the logic operation you want to compute? Since it is bitwise, how would you design the circuit for A[0] and B[0]? Note that JSIM does not support HDLs, but rather digital gate designs only. Once you figure that out, think about how to replicate it for all 32 bits. Now replace the dummy logic (you may need to look at the quick reference for the logic gates you have available). Run a gate-level simulation and examine the results. Click the green check mark to verify that your logic works.

3. Design the adder/subtractor logic. In designing this logic, consider the trade-offs between delay and logic, and choose an appropriate adder structure. In addition to the result, design logic for:

Z = true when the output, S, is zero
V = true when there is an overflow
N = true when S is negative

Replace ARITH with your design. Modify the checkoff file of line 3 to:

.include "lab3_test_adder.jsim"

Run a gate-level simulation and examine the results. Click the green check mark to verify the design.

4. Design the comparator logic. Use the Z, V, and N signals from the ARITH logic as needed. Replace COMP with your design. Modify the checkoff file of line 3 to:

.include "lab3_test_cmp.jsim"

Run a gate-level simulation and examine the results. Notice the delays incurred by having to wait for the ARITH module to compute first and consider if you can make any design modifications to speed up the system. Click the green check mark to verify the design.

5. Design the shift logic. For the shift, B[4:0] indicates the amount to shift A. In designing the shifter, consider using muxes to create a cascading shifter based on B. Replace SHIFT with your design. Modify the checkoff file of line 3 to:

.include "lab3_test_shift.jsim"

Run a gate-level simulation and examine the results. Click the green check mark to verify the design.

6. Verify your full design up to this point. Modify the checkoff file of line 3 to:

.include "lab3checkoff_1.jsim"

Run a gate-level simulation and examine the results. Click the green check mark to verify the design.

7. Turn in your JSIM code and a screenshot of the successful simulation of the complete design.

EXTRA CREDIT:

Add in a multiplier. Modify your ARITH logic to multiply A and B if ALUFN[1]=1. It should only output the lowest 32-bits of the resulting product. Verify for yourself that you can achieve this through two (1) partial products using AND gates, and (2) a ripple-carry add structure to sum the partial products (see Figure 1). Also verify that, by only examining the lower 32 bits, this structure will work with both signed and unsigned numbers.

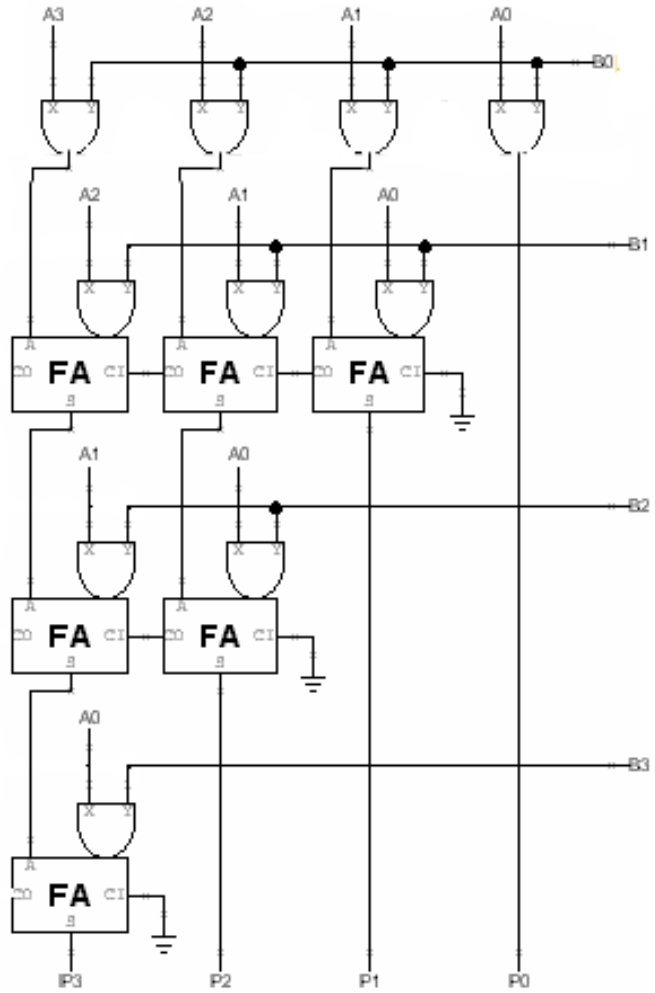

Figure 1: Multiplier Circuit

Modify the checkoff file of line 3 to:

.include "lab3_test_mult.jsim"

Run a gate-level simulation and examine the results. Click the green check mark to verify the design.

Once the multiplier functions, you can verify the complete design by modifying the checkoff file of line 3 to:

```
.include “lab3checkoff_2.jsim"
```

Run a gate-level simulation and examine the results. Click the green check mark to verify the design.

Turn in your JSIM code and a screenshot of the successful simulation of the complete design.