0) Prelab

Designing:

- Always design on paper first
- Start small, gradually going outwards

Compiling:

- Use a *.do file which is essentially a batch file. Add all the commands to compile each module to the *.do file, and then run it from the vsim command line. That greatly simplifies compilation. The same can be done when testing - you can easily start a simulation, add certain signals to the waveform, and run for a specified amount of time all from the .do file.

Testing:

- Create extensive test benches that test every case possible
- Make sure you are testing exactly what you think you are testing
- Add as much as you can to the waveform, change radix to hex.

1) 1-bit ALU

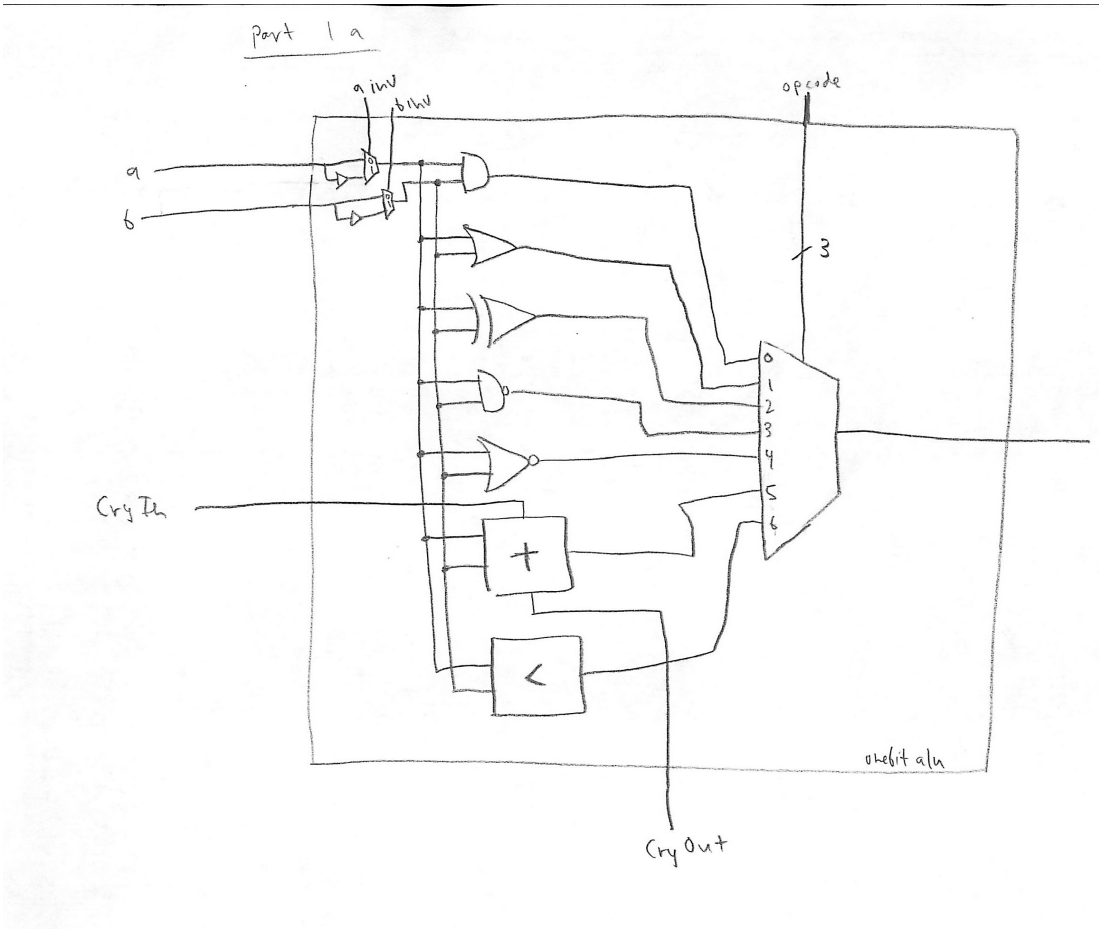
(a)

The schematic that shows the 1-bit ALU that can support the add/sub, slt, and, or, xor, nand, and nor operations is shown below. The inputs and outputs needed (shown in VHDL):

```
i_Aalu   :  in  std_logic;
i_Balu   :  in  std_logic;
i_CryIn  :  in  std_logic;
i_OpCode :  in  std_logic_vector( 2 downto 0 );
i_Ainv   :  in  std_logic;
i_Binv   :  in  std_logic;
o_Result :  out std_logic;
o_CryOut :  out std_logic
```

Opcodes:

000	and
001	or
010	xor
011	nand
100	nor
101	add
110	slt

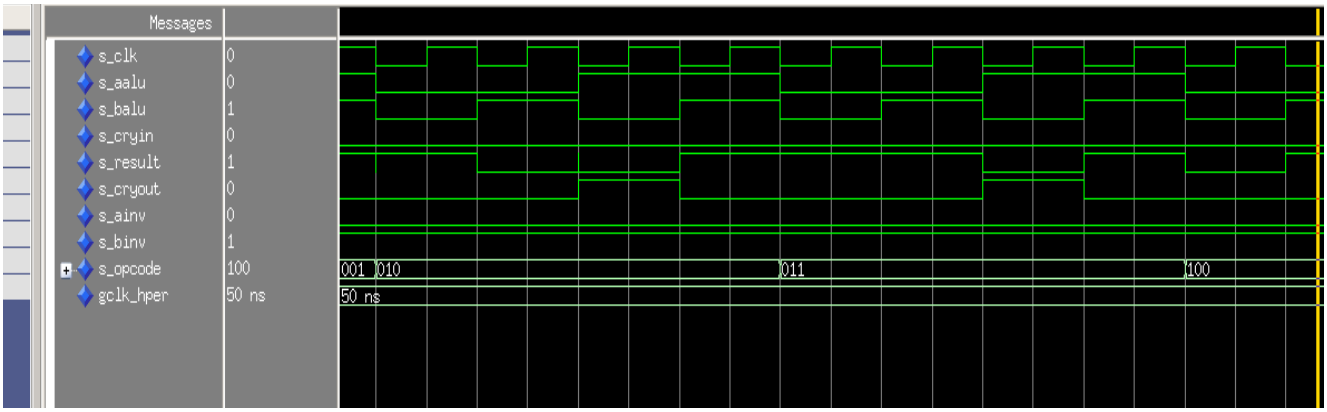


(b)

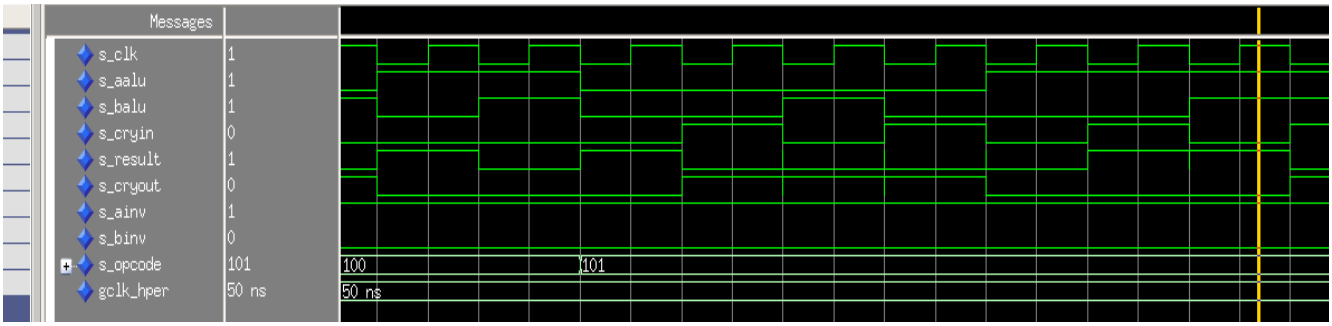
We used structural VHDL code to implement the 1-bit ALU for this part of the lab. The way we went about doing this was by having the circuit component take in both inputs and output the output, then they all get fed into a 7-to-1 MUX that selects which ALU operation is wanted, which is dictated by the i_OpCode standard logic vector. The OpCode table is given on the schematic above.

(c)

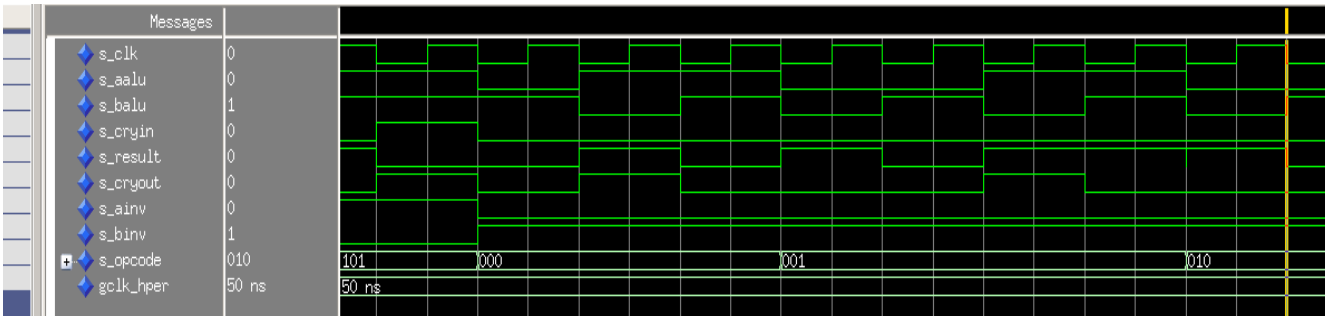
We created a VHDL testbench for this part of the lab to test each operation individually. The output waveforms are as follows:



Waveform 1: B inverted, with a glimpse of xor, nand, and nor.



Waveform 2: A inverted, glimpse of add.



Waveform 3: B inverted, glimpse of and and or.

2) 32-bit ALU

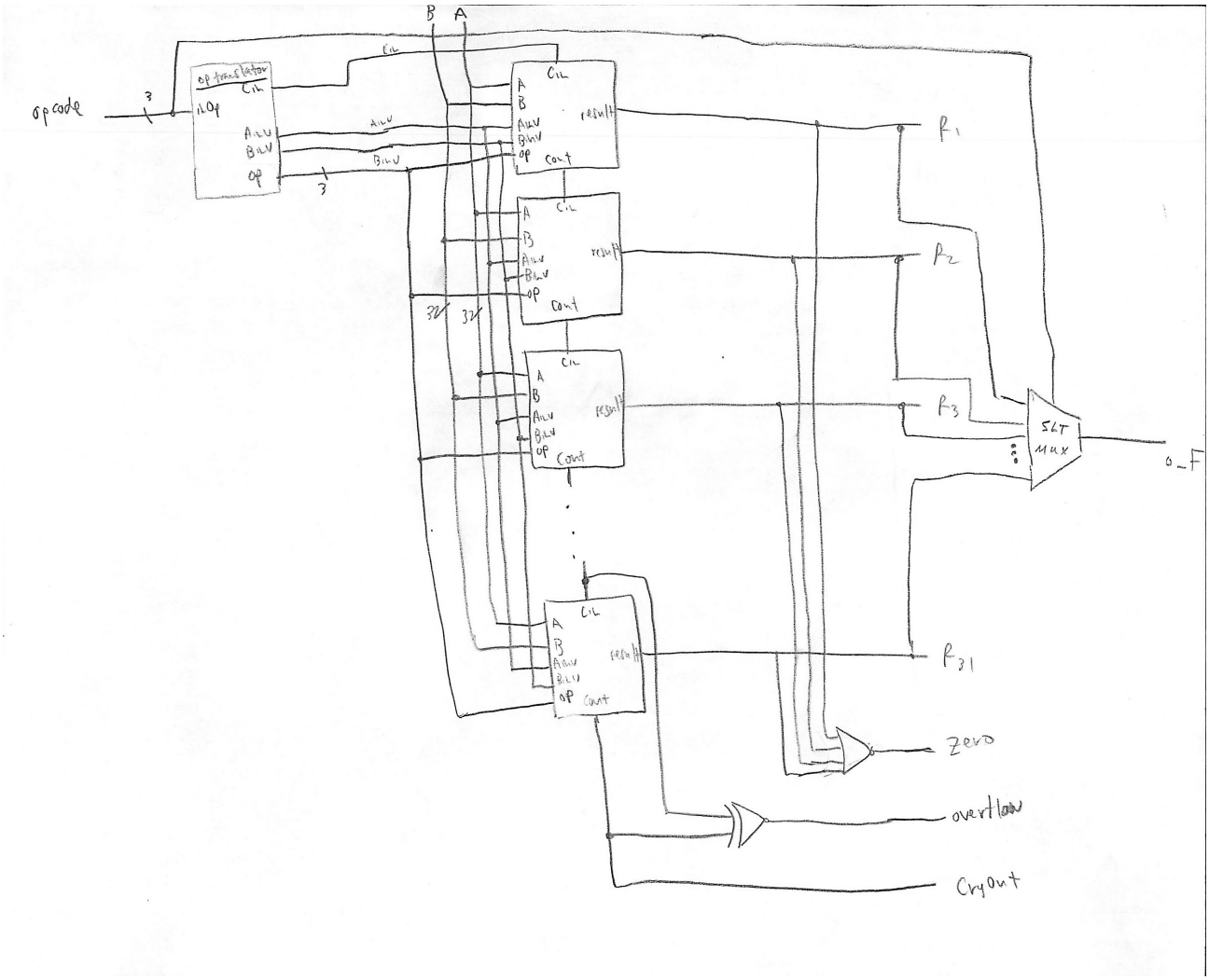
(a)

The schematic for the 32-bit ALU is shown below. It basically cascades 1-bit ALUs and then also has operations to find other outputs. Zero is calculated by doing the *nor* operation on each bit of the output from the 1-bit ALUs. Basically what this does is check to see if any of the output bits are 1, and if so, then zero will be 0, saying that the output is not all 0's. Overflow is calculated by doing the *xor* operation on the Carry-In bit and the Carry-Out bit of the last (32nd) 1-bit ALU. SLT (set less than) is calculated using a specialized sort of mux. Basically what this does is that it uses the opcode from the original input as a selector bit, and if the opcode calls for any operation other than SLT, this mux will

just output the result gathered from each of the 1-bit ALUs in a 32-bit vector. If the opcode says that SLT is the desired operation, then this mux will do A-B operation, and if the result's most significant bit is 1, then SLT will return a 1, and if the most significant bit is 0, then SLT will return a 0.

Opcodes:

000	and
001	or
010	xor
011	nand
100	nor
101	add
110	slt
111	sub

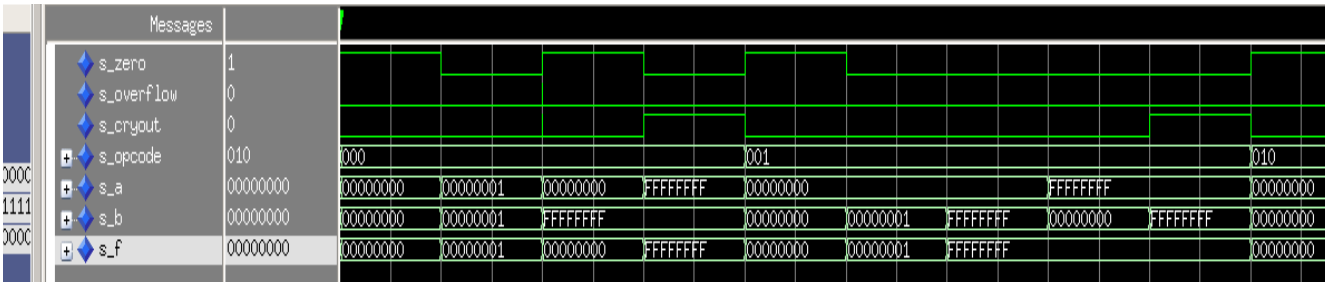


(b)

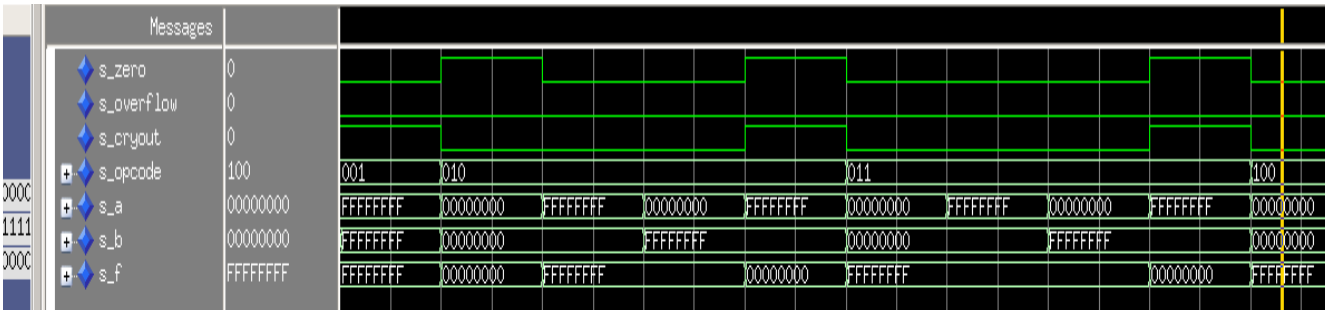
When implementing the 32-bit ALU, we did not encounter that many challenges, but we did however need to make an opcode translator that translates the opcode from the 32-bit ALU into the opcode for the 1-bit ALUs. This component takes in the opcode from the 32-bit level, and outputs the control bits for the 1-bit ALUs Ainv, Binv, Cin, and the OpCode for the 1-bit ALU level.

(c)

We created a VHDL testbench for the 32-bit ALU. The waveform screenshots are as follows:

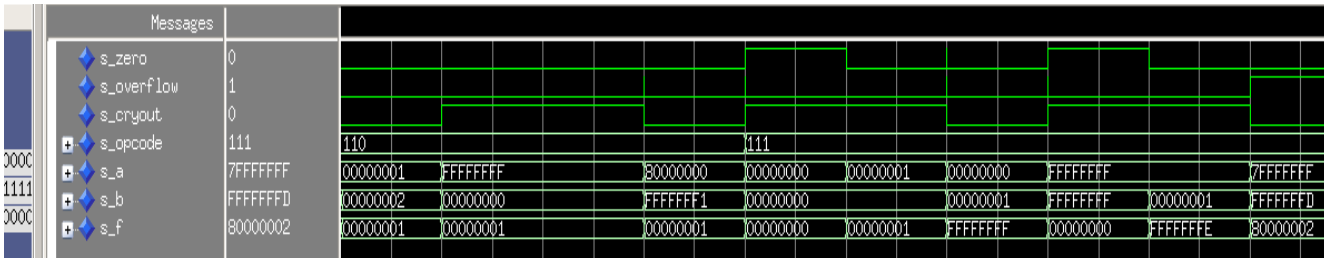


Waveform 4: And, Or



Waveform 5: Xor, Nand

Waveform 6: Nor, Add

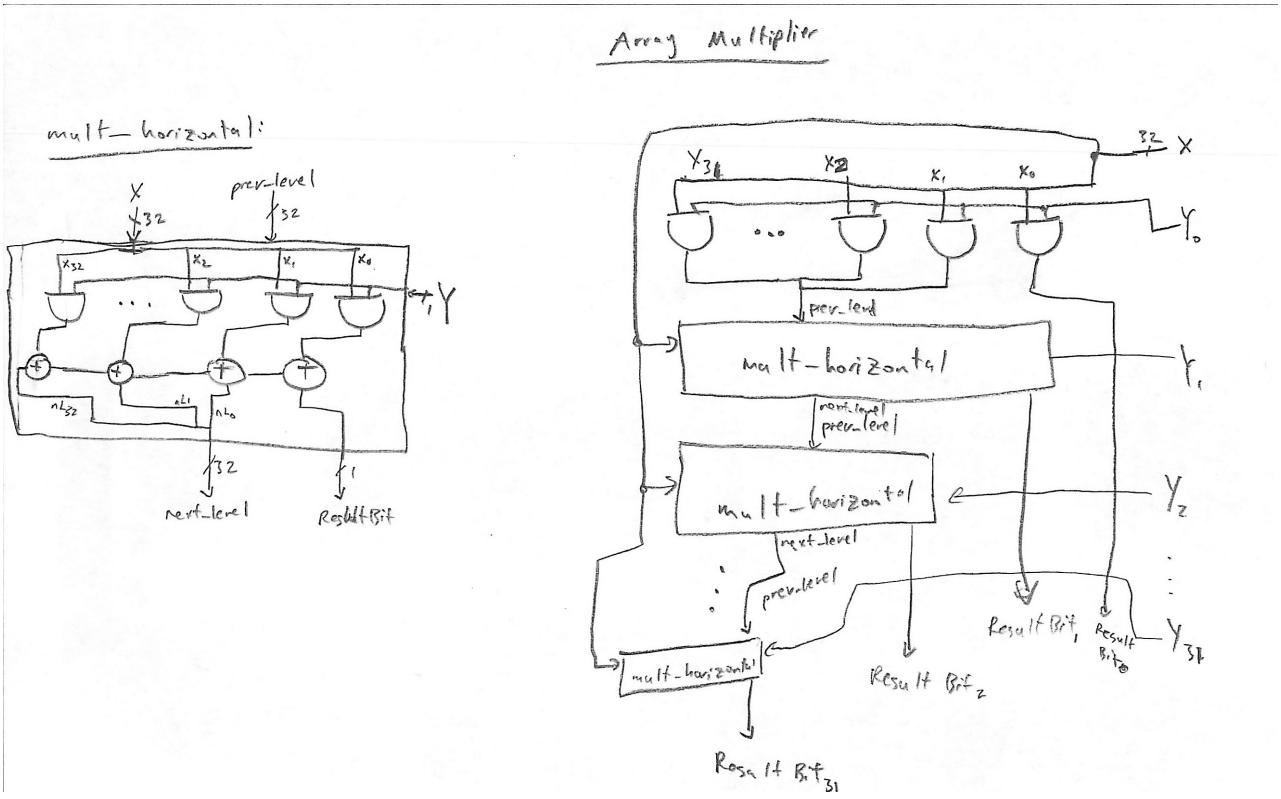


Waveform 7: SLT, Sub

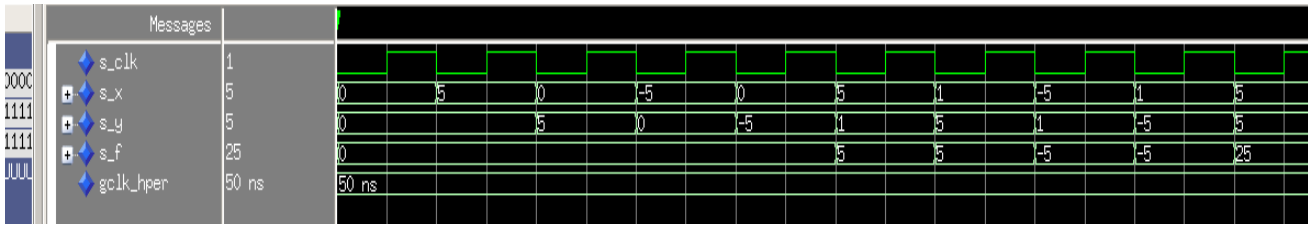
3) Array Multiplier

(a)

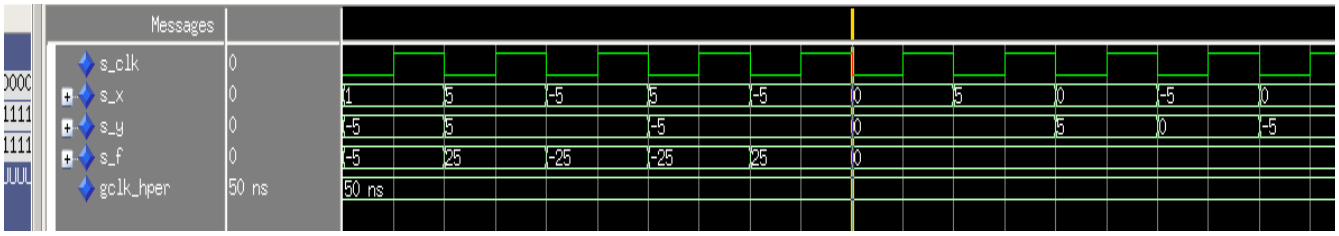
We used structural VHDL to implement the array multiplier for our lab. The way we did this is that we had a component for each horizontal row (consisting of and gates and full adders, 32-wide). We then cascaded these horizontal components 32 times. Each horizontal row component gets all of the results of the previous level's additions, plus the last carry-out as inputs. The outputs for this component are the information for the next level and the ResultBit. The ResultBit is the bit from that row's addition (farthest to the right). All of the ResultBits from each of the 32 horizontal rows are then concatenated together to create the final output vector. At the end of the array multiplier, a 64-bit signal is acquired, but a 32-bit response is requested, so we just chopped off the top 32-bits. A schematic is shown below of the mult_horizontal component as well as the entire Array Multiplier implemented using these components.



(b)
We created a VHDL testbench for the Array Multiplier. We verified that the design works for pos*pos, pos*neg, neg*pos, neg*neg, pos*0, neg*0, 0*neg, 0*pos. The following waveforms use the decimal radix for quick verification.



*Waveform 8: 0*0=0, 5*0=0, 0*5=0, -5*0=0, 0*-5=0, 5*1=5, 1*5=5, -5*1=.5, 1*-5=-5*

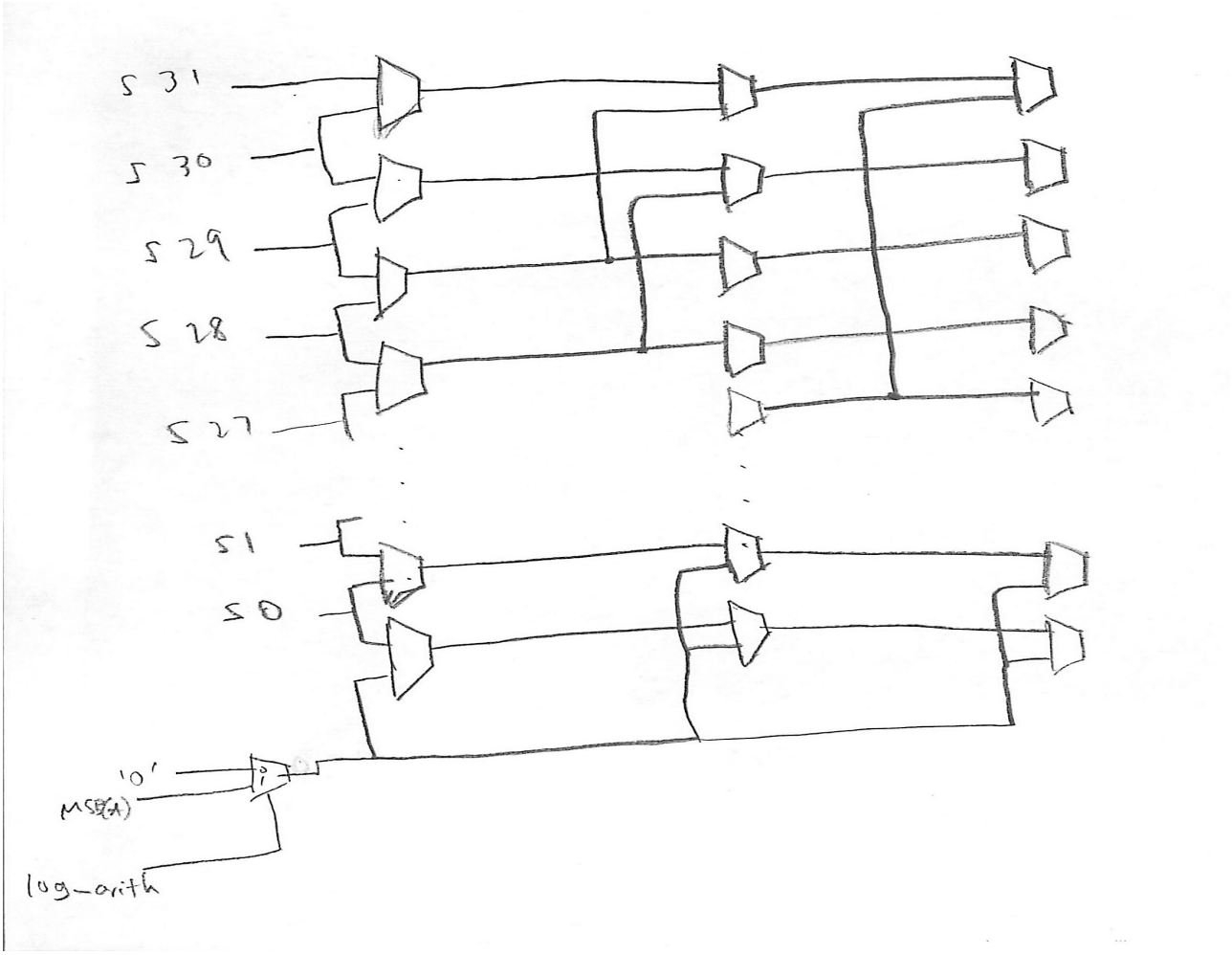


*Waveform 9: 5*5=25, -5*5=-25, 5*-5=-25, -5*-5=25*

4) Barrel Shifter

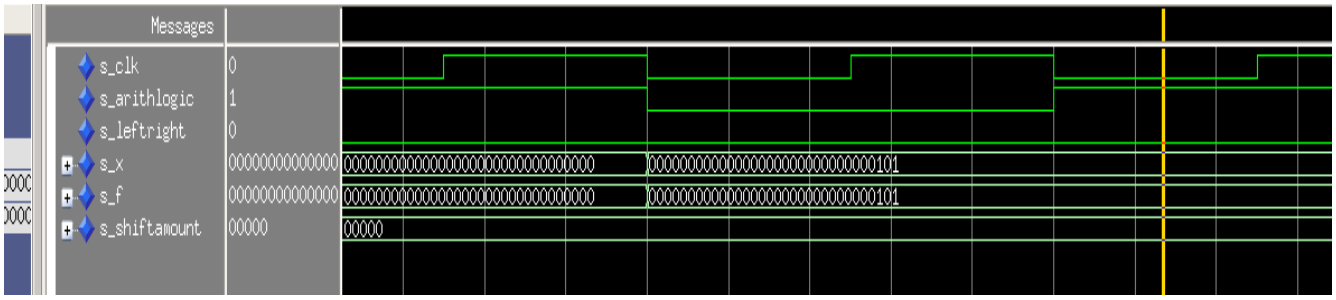
(a)
There are 2 types of shifting right: logical and arithmetic. Shift right logical (`srl`) shifts the bits to the right, and shifts in 0's. Shift right arithmetic (`sra`) shifts the bits to the right, and shifts in the sign bit, to ensure that the sign of the original number is preserved. There is no shift left arithmetic (`sla`) instruction, because when shifting left, the sign bit will always be preserved, so there is no need to differentiate between a logical or arithmetic operation.

(b)
We created structural VHDL code to implement the barrel shifter, with the ability to do both arithmetic and logical shifting. To select between arithmetic and logical shifting, a control bit is input to the top-level barrel shifter. This is then a control for a mux that selects either '0' (logical), or MSB(input) (arithmetic). Because the arithmetic shift keeps the sign bit and shifts that in instead of the '0', if the arithmetic shift is requested, the sign bit, or the MSB(input) will be what is shifted in. A very rough schematic that shows the general outline of this design is shown below. Most important on this schematic is the cascading muxes and the '0'/MSB(A) input selected by log_arith (control bit to decide between 0-logical or 1-arithmetic shifting).

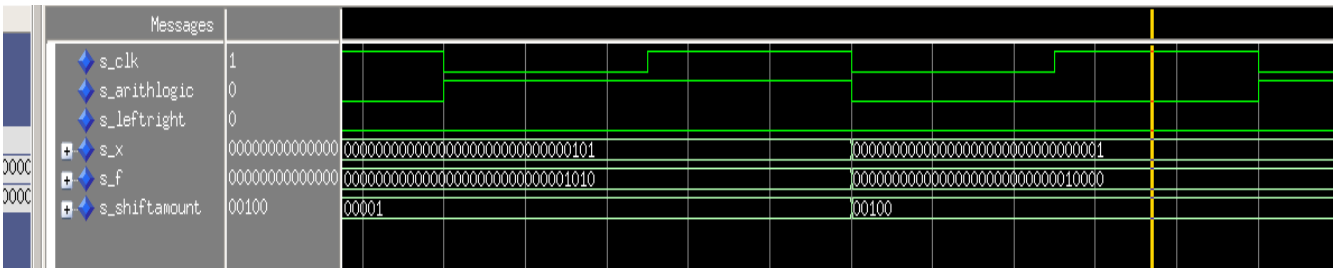


(c)
The nice part about our design for the barrel shifter is that adding a quick control signal to signify whether or not we want to do a left or right shift. When the left shift control signal is signified, then the control signal that decides between logical and arithmetic shifting will automatically be set to '0'. To implement the left shift given our design for the right shift, all we have to do is read the input in reverse. We had to swap the bit order for the output as well when the sign is being changed.

(d)
We created a VHDL testbench for this design. The waveform outputs are shown below.

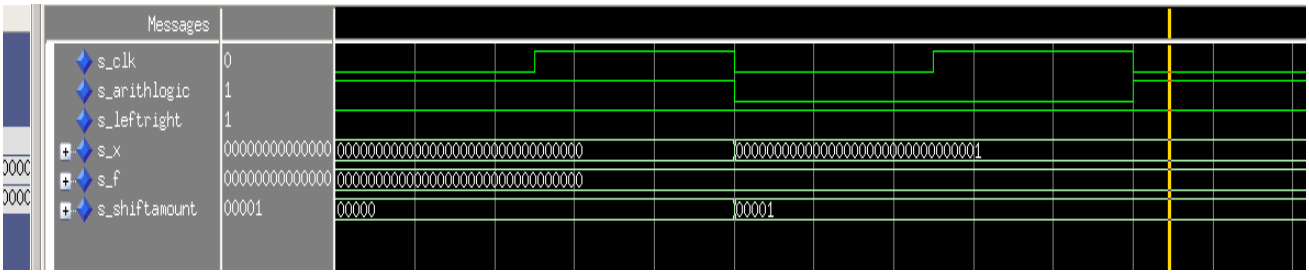


Waveform 10: Shift Left 0

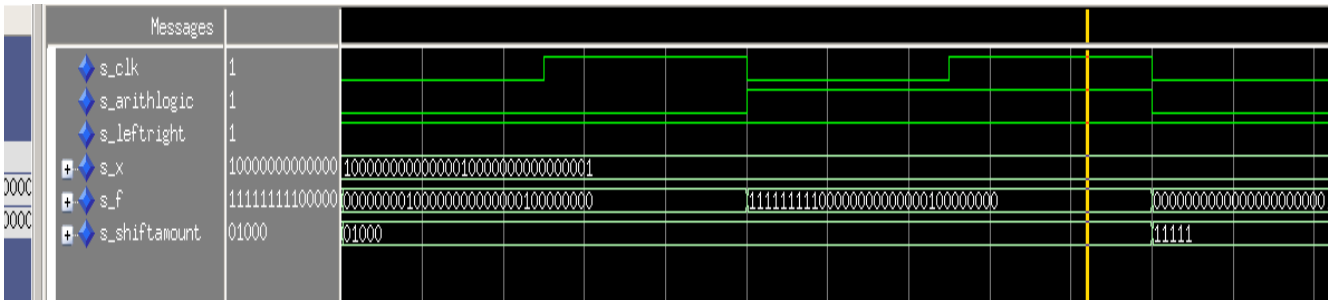


Waveform 11: Shift left various amounts

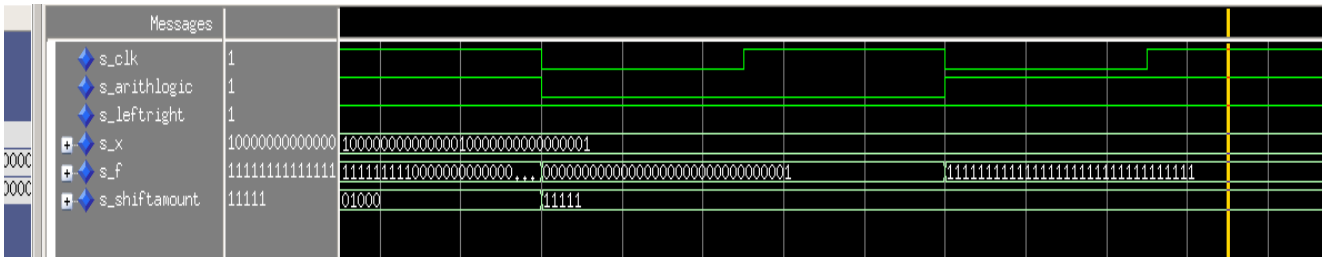
Note that there are other shift amounts tested, but are not necessary to show. See *tb_shifter* for detailed test information.



Waveform 12: Shift right 0 and shift right 1, 1. Output should be 0 for both



Waveform 13: Shift right logical and arithmetic



Waveform 14: More shift right with logical and arithmetic