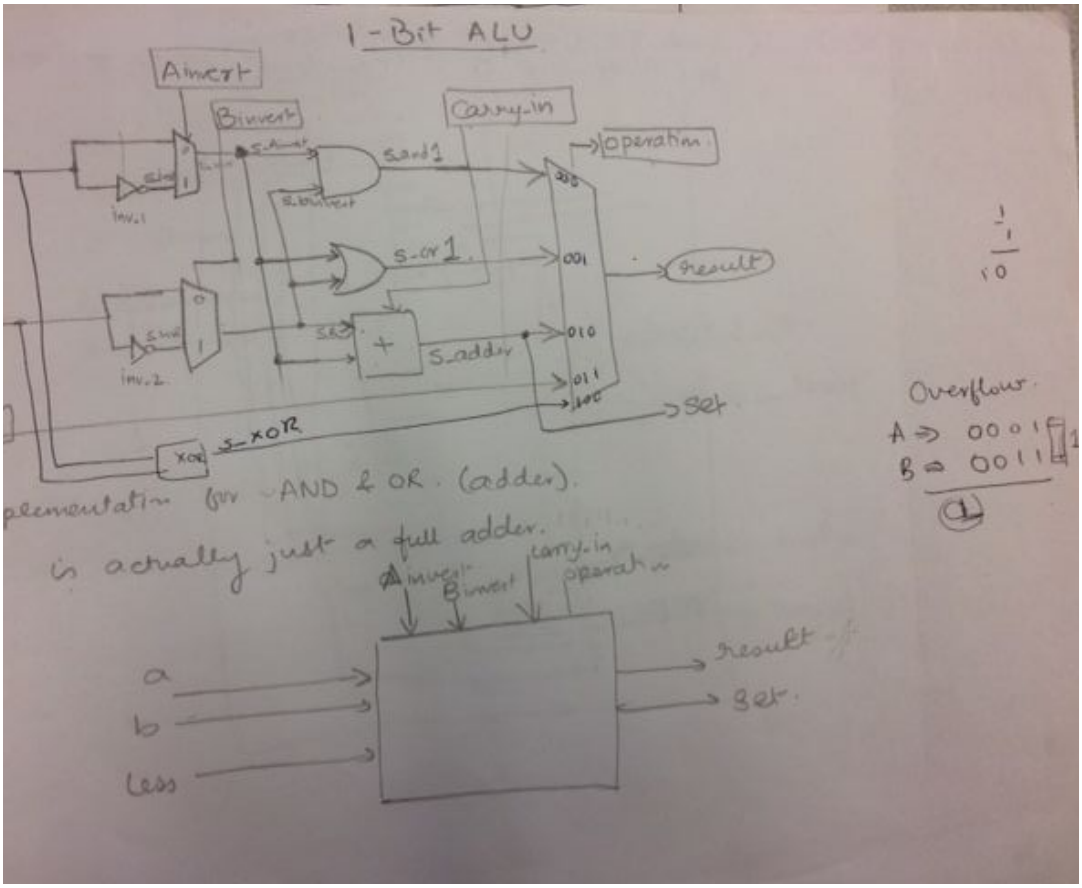
Project A

Chris Kelley
Souparni Agnihotri
Fahmida Joyti

Prelab:

1) Draw the schematic before implementing in VHDL
2) Make small components instead of one big component
3) Comment out parts of the code to improve readability
4) Meet twice a week.
5) Save all your work.

Part 1: *1 bit ALU*



We used three inputs - A, B and Less and four control signals- A_invert, B_invert, Carry_in and operation.
We had two outputs - Result and Set that we pass back in the 32-bit ALU.

Project A

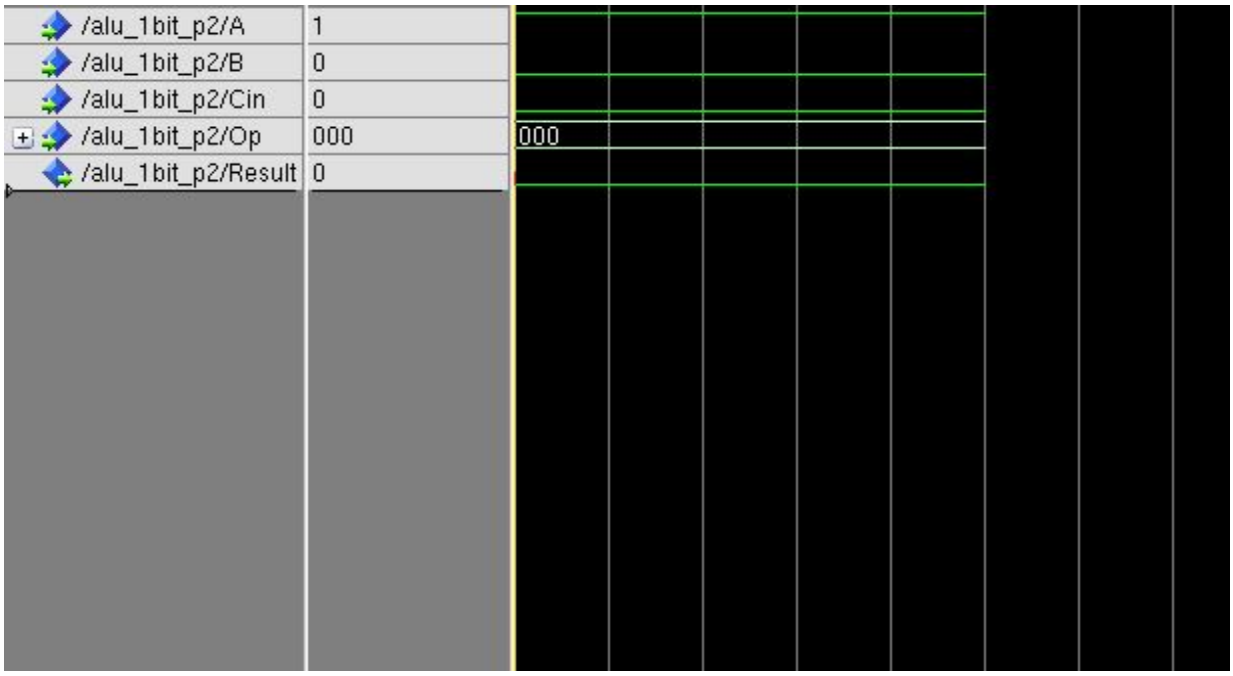
We implemented this structurally and made sure to pass in the appropriate signals to the various components so that they work correctly.

The code is structured exactly according to the way the diagram is drawn out.

SCREENSHOTS:

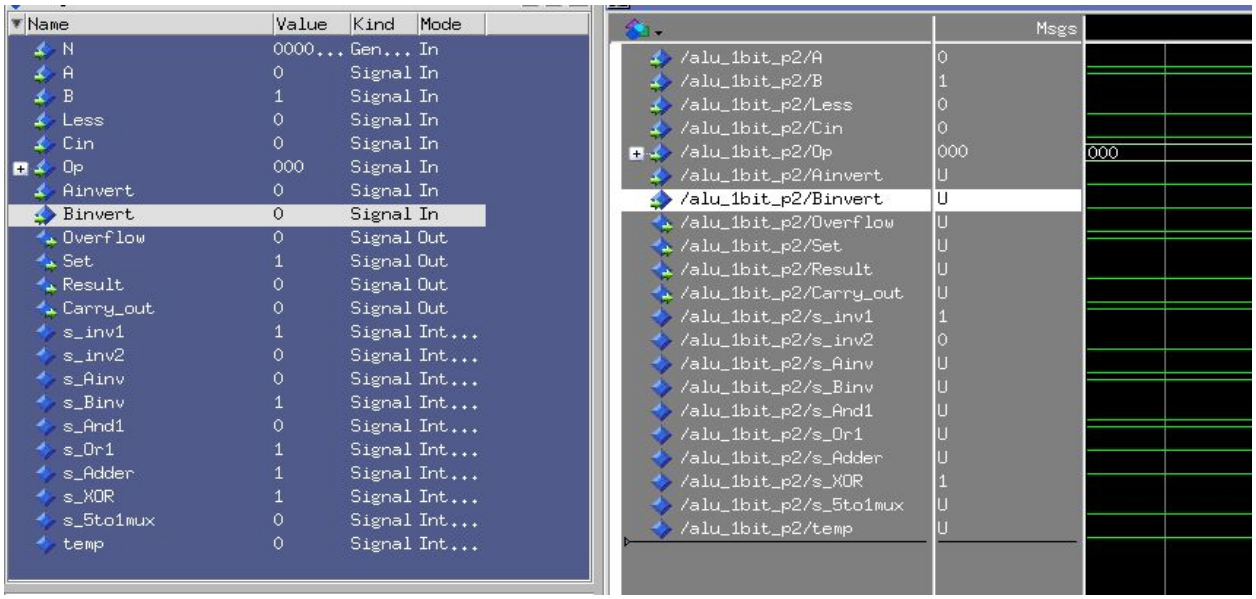
And operation

A=1 B=0 Cin=0 OP="000"



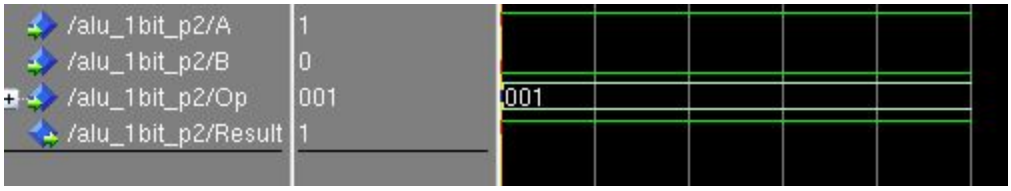
1 and 0 = 0. OP code is looking for and operation in our 5 to 1 mux.

A=0 B=1 Cin=0 OP="000" - switch of A, B values from above. Result is still 0.



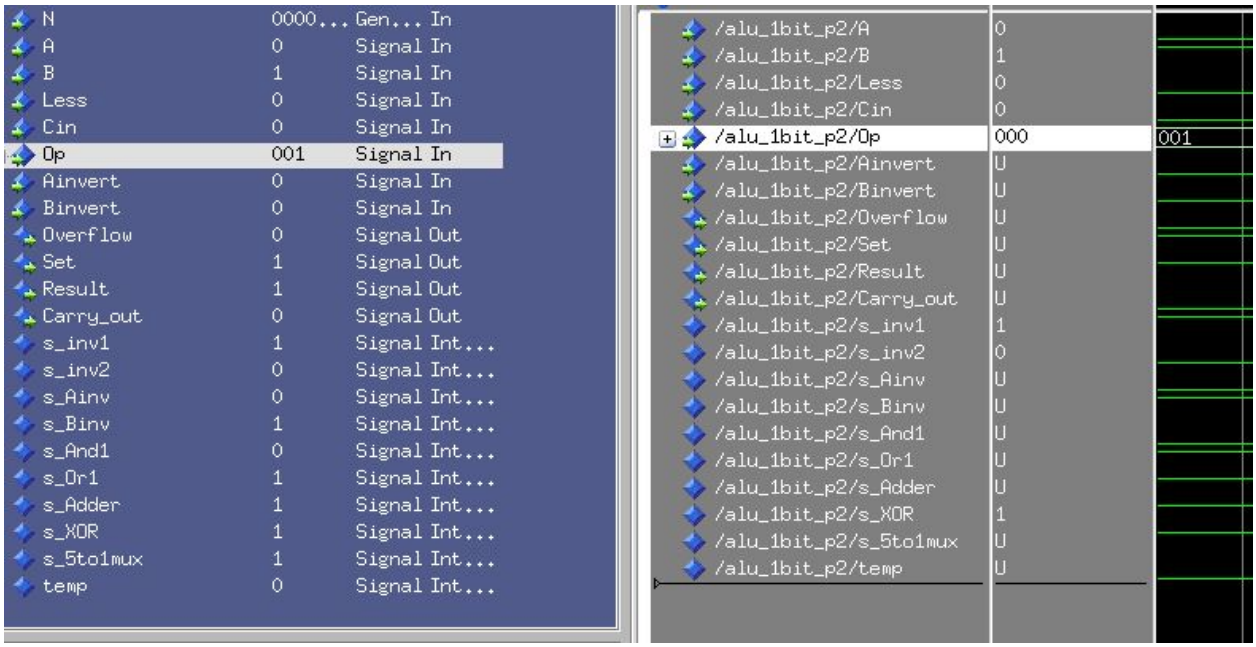
Project A

Or operation



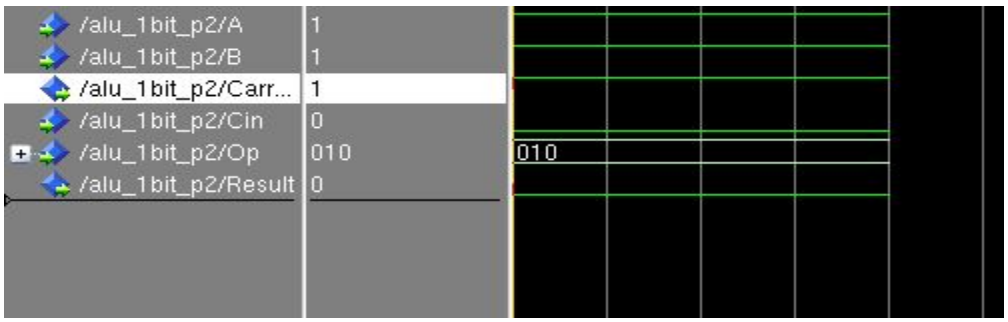
OP code changes to 001 selecting the second option in our 5 to 1 mux.

Reverse of other or operation



Add operation

1 +1

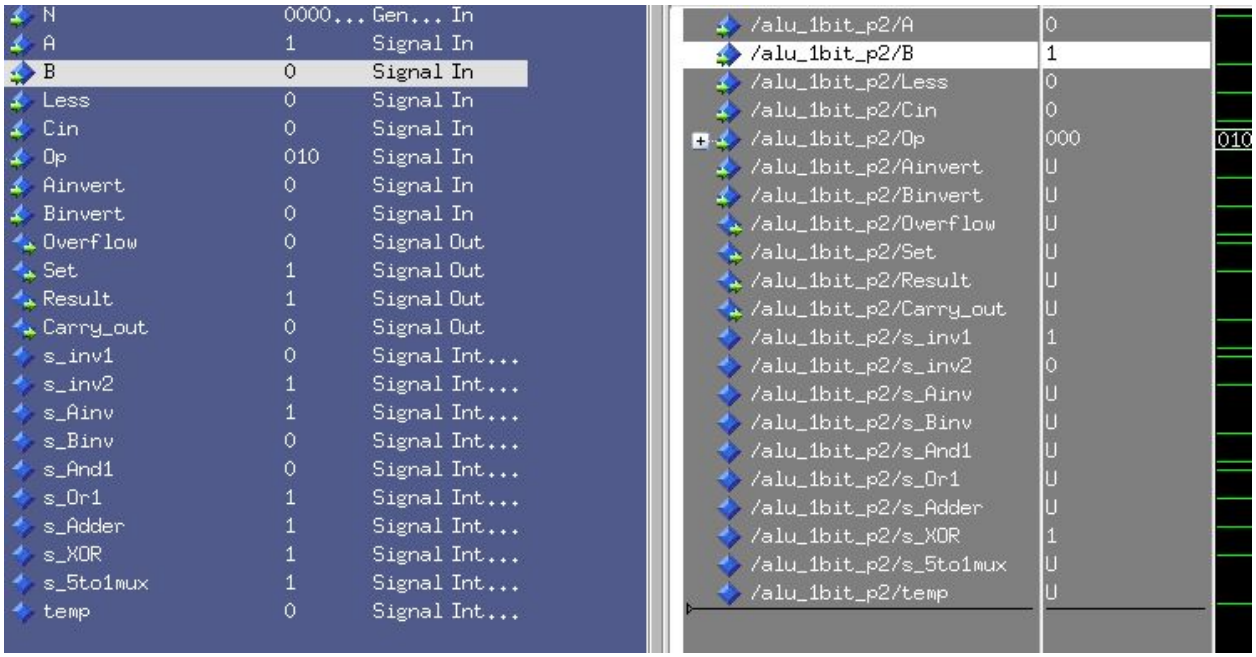


Project A

Zero because 1 bit addition doesn't allow to show 2. The carry out is one which would represent 2 in binary.

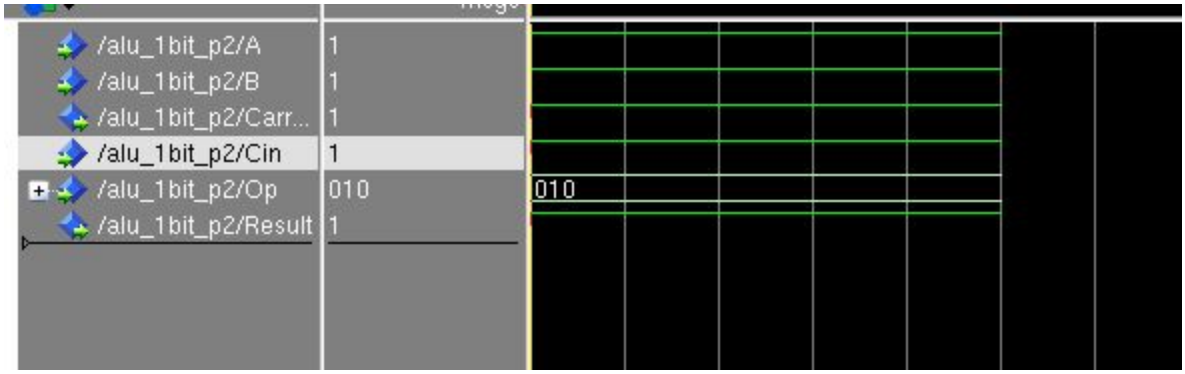
A<B (0+1)

B<A (1+0)



Subtraction operation

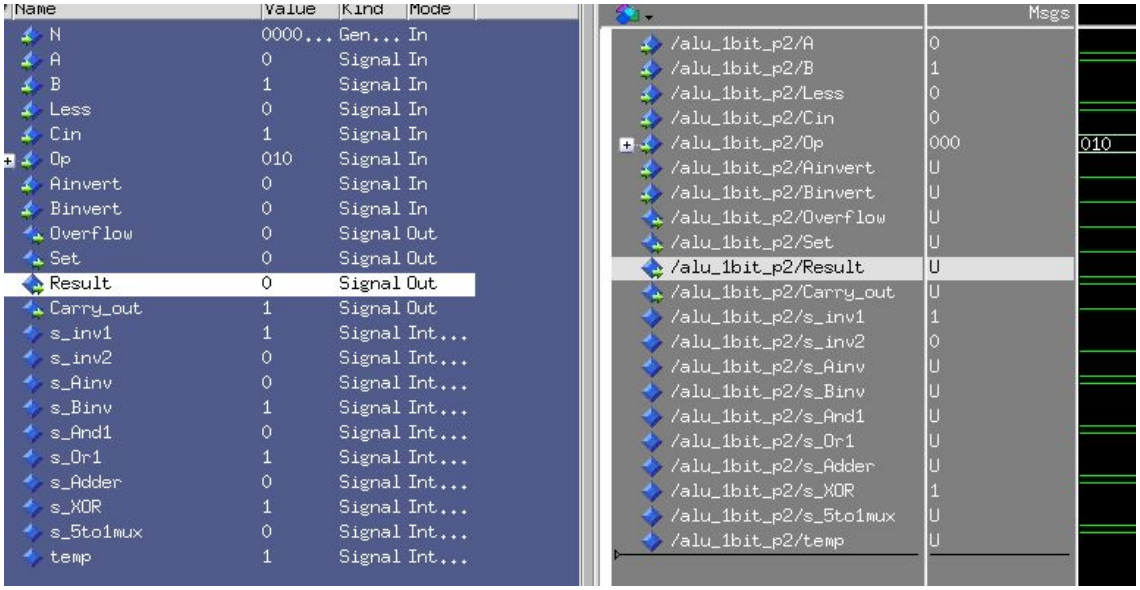
1-1



Project A

The cin bit makes the operation a subtraction problem even though the op code doesn't change. The full adder handles the math for us.

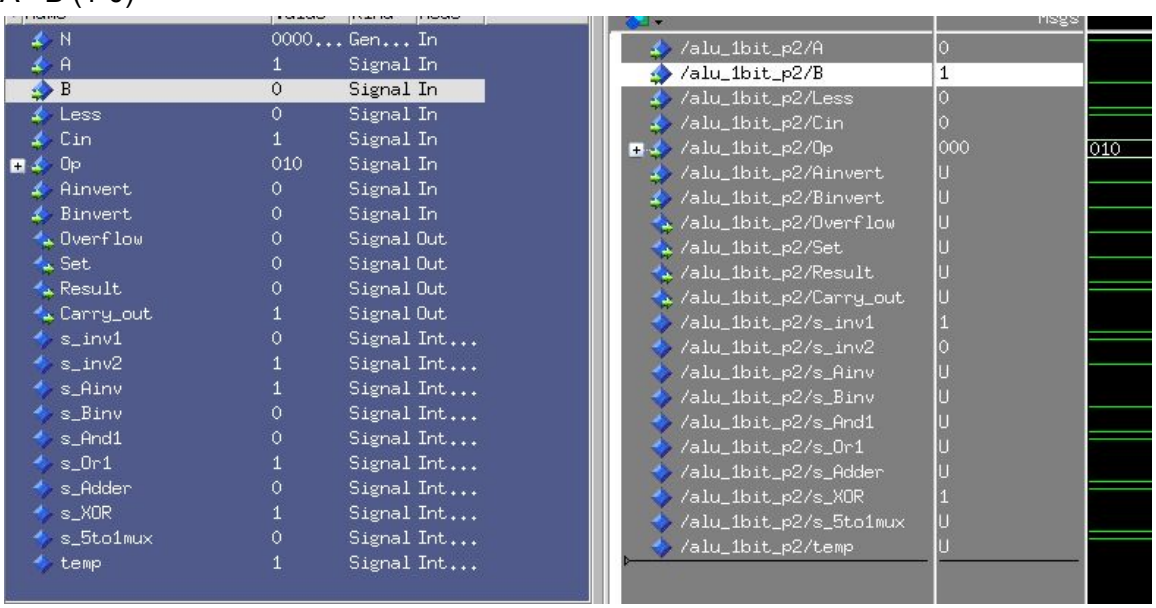
B<A (0-1)



Name	Value	Kind	Mode
N	0000...	Gen...	In
A	0	Signal	In
B	1	Signal	In
Less	0	Signal	In
Cin	1	Signal	In
Op	010	Signal	In
Ainvert	0	Signal	In
Binvert	0	Signal	In
Overflow	0	Signal	Out
Set	0	Signal	Out
Result	0	Signal	Out
Carry_out	1	Signal	Out
s_inv1	1	Signal	Int...
s_inv2	0	Signal	Int...
s_Ainv	0	Signal	Int...
s_Binv	1	Signal	Int...
s_And1	0	Signal	Int...
s_Or1	1	Signal	Int...
s_Adder	0	Signal	Int...
s_XOR	1	Signal	Int...
s_5to1mux	0	Signal	Int...
temp	1	Signal	Int...

Msgs		
/alu_1bit_p2/A	0	
/alu_1bit_p2/B	1	
/alu_1bit_p2/Less	0	
/alu_1bit_p2/Cin	0	
/alu_1bit_p2/Op	000	010
/alu_1bit_p2/Ainvert	U	
/alu_1bit_p2/Binvert	U	
/alu_1bit_p2/Overflow	U	
/alu_1bit_p2/Set	U	
/alu_1bit_p2/Result	U	
/alu_1bit_p2/Carry_out	U	
/alu_1bit_p2/s_inv1	1	
/alu_1bit_p2/s_inv2	0	
/alu_1bit_p2/s_Ainv	U	
/alu_1bit_p2/s_Binv	U	
/alu_1bit_p2/s_And1	U	
/alu_1bit_p2/s_Or1	U	
/alu_1bit_p2/s_Adder	U	
/alu_1bit_p2/s_XOR	1	
/alu_1bit_p2/s_5to1mux	U	
/alu_1bit_p2/temp	U	

A >B (1-0)



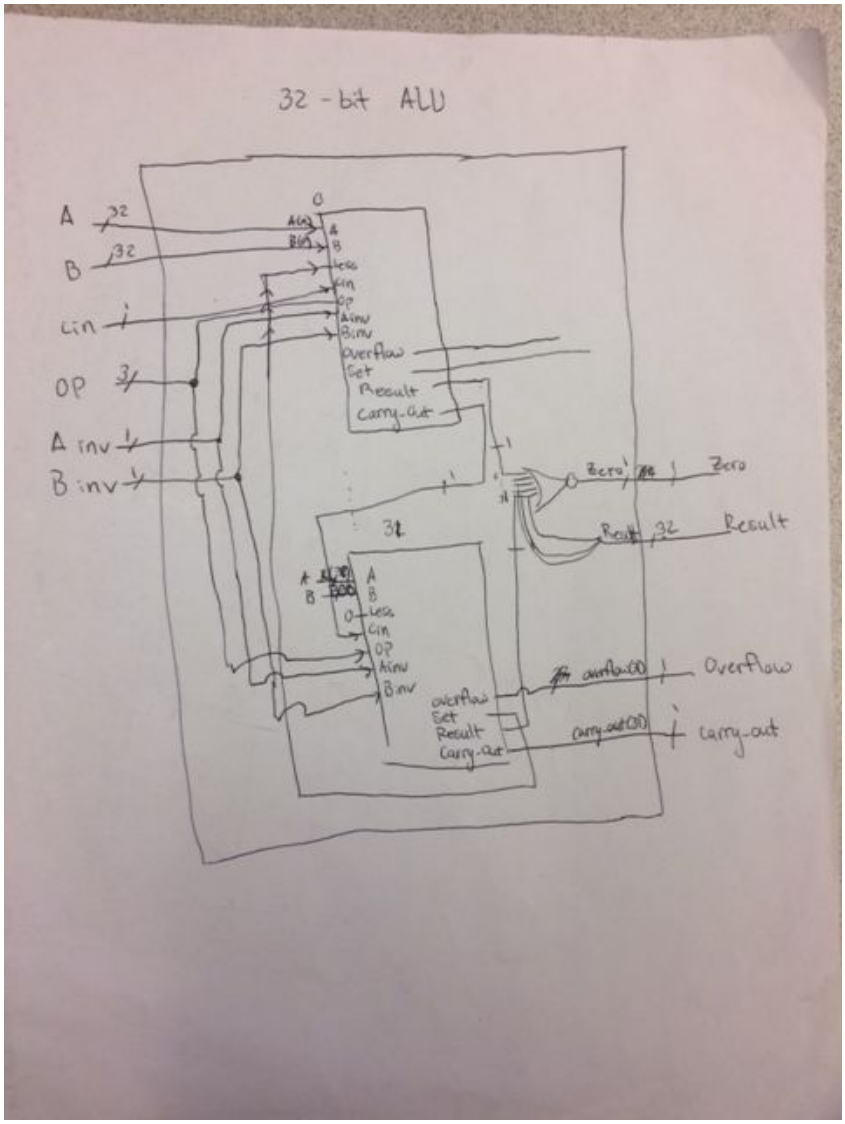
Name	Value	Kind	Mode
N	0000...	Gen...	In
A	1	Signal	In
B	0	Signal	In
Less	0	Signal	In
Cin	1	Signal	In
Op	010	Signal	In
Ainvert	0	Signal	In
Binvert	0	Signal	In
Overflow	0	Signal	Out
Set	0	Signal	Out
Result	0	Signal	Out
Carry_out	1	Signal	Out
s_inv1	0	Signal	Int...
s_inv2	1	Signal	Int...
s_Ainv	1	Signal	Int...
s_Binv	0	Signal	Int...
s_And1	0	Signal	Int...
s_Or1	1	Signal	Int...
s_Adder	0	Signal	Int...
s_XOR	1	Signal	Int...
s_5to1mux	0	Signal	Int...
temp	1	Signal	Int...

Msgs		
/alu_1bit_p2/A	0	
/alu_1bit_p2/B	1	
/alu_1bit_p2/Less	0	
/alu_1bit_p2/Cin	0	
/alu_1bit_p2/Op	000	010
/alu_1bit_p2/Ainvert	U	
/alu_1bit_p2/Binvert	U	
/alu_1bit_p2/Overflow	U	
/alu_1bit_p2/Set	U	
/alu_1bit_p2/Result	U	
/alu_1bit_p2/Carry_out	U	
/alu_1bit_p2/s_inv1	1	
/alu_1bit_p2/s_inv2	0	
/alu_1bit_p2/s_Ainv	U	
/alu_1bit_p2/s_Binv	U	
/alu_1bit_p2/s_And1	U	
/alu_1bit_p2/s_Or1	U	
/alu_1bit_p2/s_Adder	U	
/alu_1bit_p2/s_XOR	1	
/alu_1bit_p2/s_5to1mux	U	
/alu_1bit_p2/temp	U	

We would show the less operation but it cannot be fed by the set out, unless we had additional outside logic.

Project A

Part 2: ***n bit ALU***



We are setting the last bit of the output as overflow.

Zero basically acts as a beq function in MIPS. So we are passing in all the results obtained into an NOT gate and if the zero is 0 the all the results are equal.

For the slt, we are passing in the value obtained in SET into the LESS input. This will help us compare if the values are smaller or greater than each other.

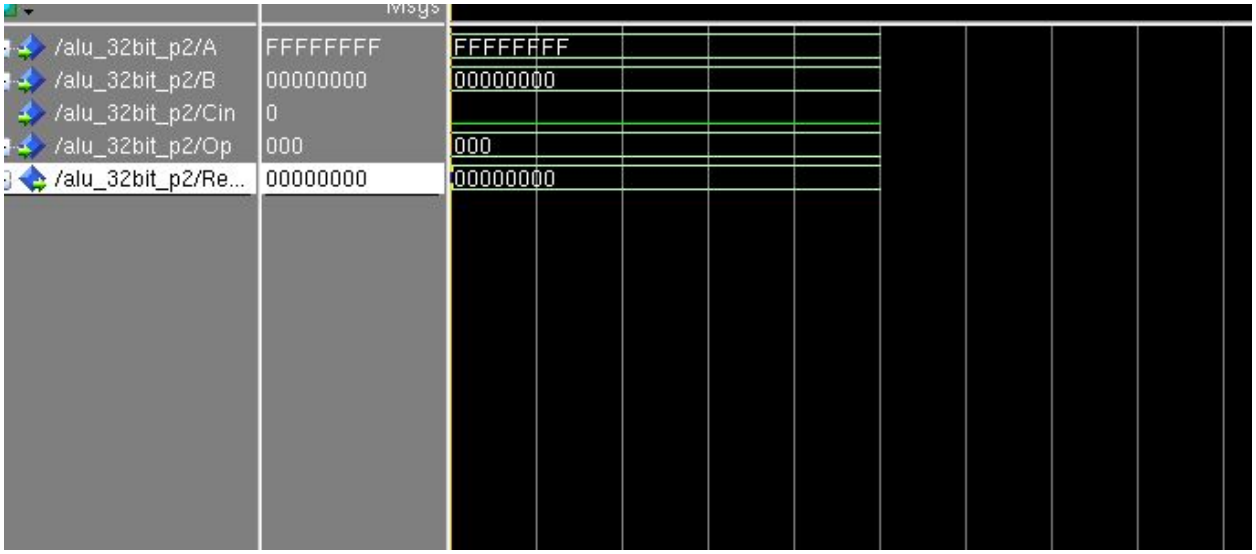
Here are the screenshots with the corresponding memory maps

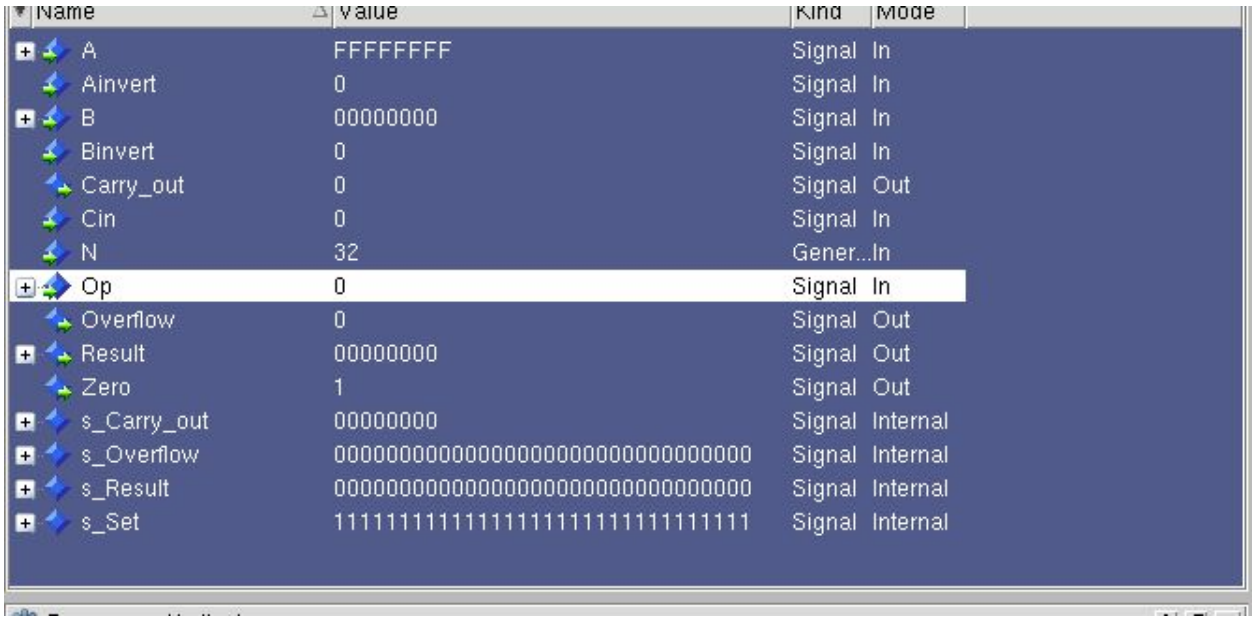
Project A

SCREENSHOTS:

And Operation

A>B

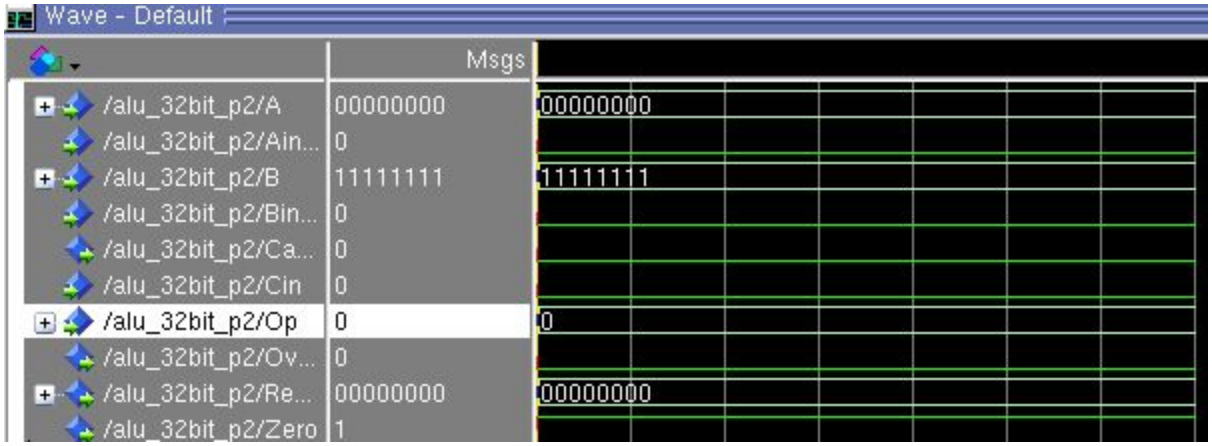




Name	Value	Kind	Mode
A	FFFFFFFF	Signal	In
Ainvert	0	Signal	In
B	00000000	Signal	In
Binvert	0	Signal	In
Carry_out	0	Signal	Out
Cin	0	Signal	In
N	32	Gener...	In
Op	0	Signal	In
Overflow	0	Signal	Out
Result	00000000	Signal	Out
Zero	1	Signal	Out
s_Carry_out	00000000	Signal	Internal
s_Overflow	00000000000000000000000000000000	Signal	Internal
s_Result	00000000000000000000000000000000	Signal	Internal
s_Set	11111111111111111111111111111111	Signal	Internal

Project A

B>A

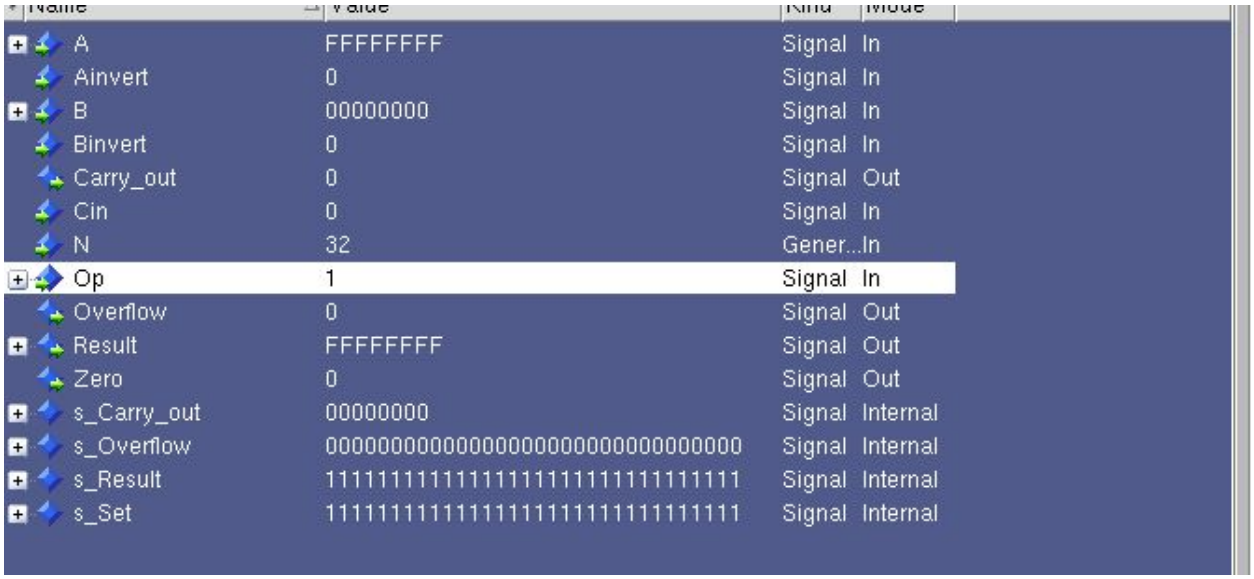


Name	Value	Kind	Mode
A	00000000	Signal	In
Ainvert	0	Signal	In
B	11111111	Signal	In
Binvert	0	Signal	In
Carry_out	0	Signal	Out
Cin	0	Signal	In
N	32	Gener...	In
Op	0	Signal	In
Overflow	0	Signal	Out
Result	00000000	Signal	Out
Zero	1	Signal	Out
s_Carry_out	00000000	Signal	Internal
s_Overflow	00000000000000000000000000000000	Signal	Internal
s_Result	00000000000000000000000000000000	Signal	Internal
s_Set	00010001000100010001000100010001	Signal	Internal

Project A

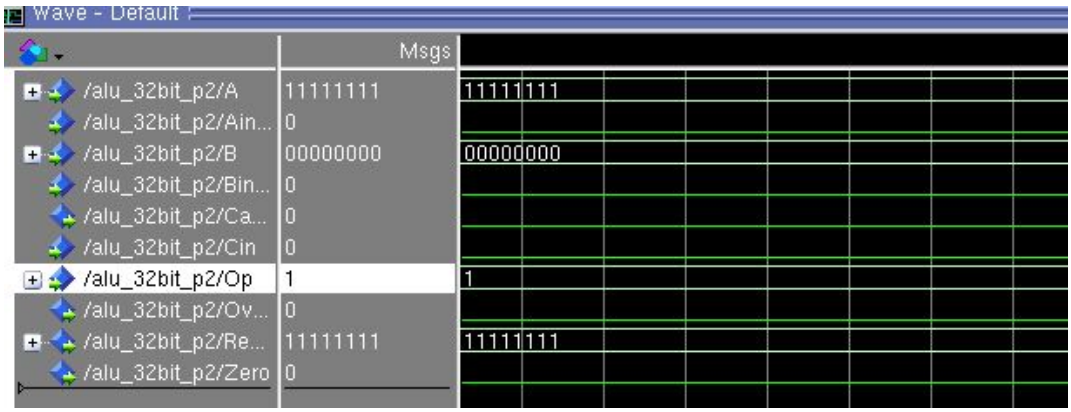
Or operation

A>B



Name	Value	Kind	Mode
A	FFFFFFFF	Signal	In
Ainvert	0	Signal	In
B	00000000	Signal	In
Binvert	0	Signal	In
Carry_out	0	Signal	Out
Cin	0	Signal	In
N	32	Gener...	In
Op	1	Signal	In
Overflow	0	Signal	Out
Result	FFFFFFFF	Signal	Out
Zero	0	Signal	Out
s_Carry_out	00000000	Signal	Internal
s_Overflow	00000000000000000000000000000000	Signal	Internal
s_Result	11111111111111111111111111111111	Signal	Internal
s_Set	11111111111111111111111111111111	Signal	Internal

B<A



Project A

Name	Value	Kind	Mode
A	11111111	Signal	In
Ainvert	0	Signal	In
B	00000000	Signal	In
Binvert	0	Signal	In
Carry_out	0	Signal	Out
Cin	0	Signal	In
N	32	Gener...	In
Op	1	Signal	In
Overflow	0	Signal	Out
Result	11111111	Signal	Out
Zero	0	Signal	Out
s_Carry_out	00000000	Signal	Internal
s_Overflow	00000000000000000000000000000000	Signal	Internal
s_Result	00010001000100010001000100010001	Signal	Internal
s_Set	00010001000100010001000100010001	Signal	Internal

Alu 32 bit addition

Project A

A	0FFF00FF	Signal	In
Ainvert	0	Signal	In
B	F000FF00	Signal	In
Binvert	0	Signal	In
Carry_out	0	Signal	Out
Cin	0	Signal	In
N	32	Gener...	In
Op	2	Signal	In
Overflow	0	Signal	Out
Result	FFFFFFFF	Signal	Out
Zero	0	Signal	Out
s_Carry_out	00000000	Signal	Internal
s_Overflow	00000000000000000000000000000000	Signal	Internal
s_Result	11111111111111111111111111111111	Signal	Internal
s_Set	11111111111111111111111111111111	Signal	Internal

Processes (Active)

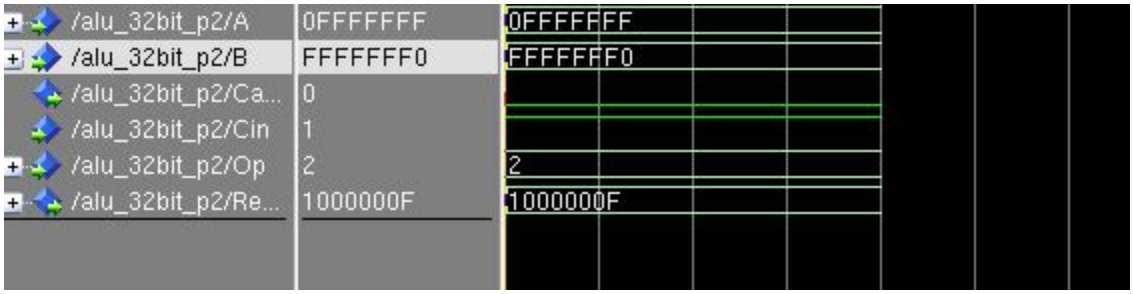
Alu 32 bit subtraction

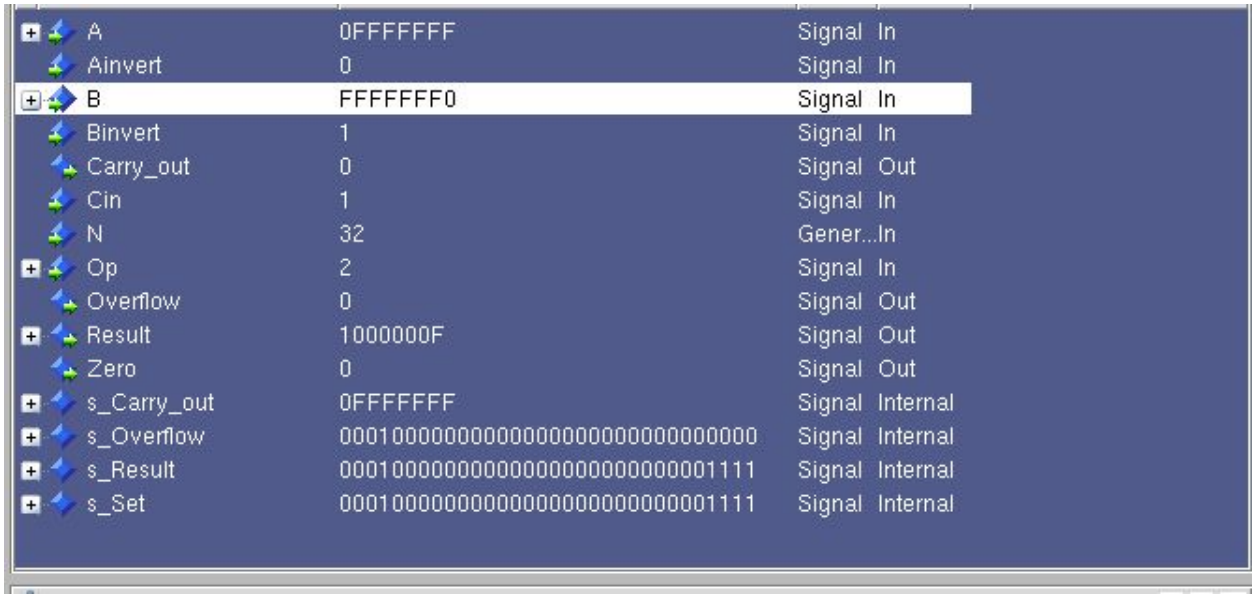
/alu_32bit_p2/A	FFFFFF00	FFFFFF00
/alu_32bit_p2/B	0FFFFF00	0FFFFF00
/alu_32bit_p2/Cin	1	
/alu_32bit_p2/Op	010	010
/alu_32bit_p2/Re...	F0000000	F0000000

A	FFFFFF00	Signal	In
Ainvert	0	Signal	In
B	0FFFFF00	Signal	In
Binvert	1	Signal	In
Carry_out	1	Signal	Out
Cin	1	Signal	In
N	32	Gener...	In
Op	2	Signal	In
Overflow	0	Signal	Out
Result	F0000000	Signal	Out
Zero	0	Signal	Out
s_Carry_out	FFFFFFFF	Signal	Internal
s_Overflow	00000000000000000000000000000000	Signal	Internal
s_Result	11110000000000000000000000000000	Signal	Internal
s_Set	11110000000000000000000000000000	Signal	Internal

Subtraction with a less than b

Project A





‘

Xor operation

A>B

Project A

A	FFF000FF	Signal	In
Ainvert	0	Signal	In
B	000FFF00	Signal	In
Binvert	0	Signal	In
Carry_out	0	Signal	Out
Cin	0	Signal	In
N	32	Gener...	In
Op	4	Signal	In
Overflow	0	Signal	Out
Result	FFFFFFFF	Signal	Out
Zero	0	Signal	Out
s_Carry_out	00000000	Signal	Internal
s_Overflow	00000000000000000000000000000000	Signal	Internal
s_Result	11111111111111111111111111111111	Signal	Internal
s_Set	11111111111111111111111111111111	Signal	Internal

Processes (Active)

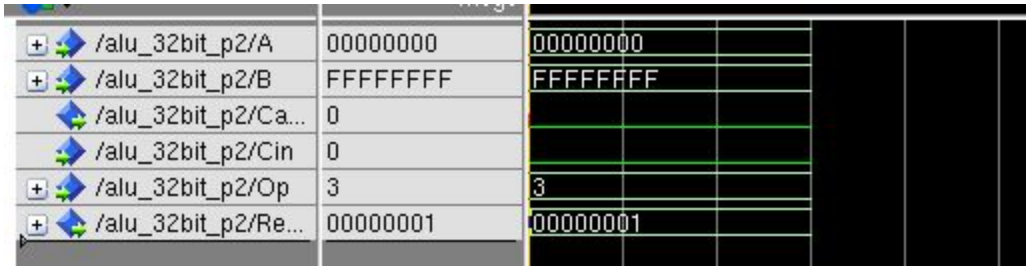
Xor A = B

	Msgs	
/alu_32bit_p2/A	FFFFFFFF	FFFFFFFF
/alu_32bit_p2/B	FFFFFFFF	FFFFFFFF
/alu_32bit_p2/Cin	0	
/alu_32bit_p2/Op	4	4
/alu_32bit_p2/Re...	00000000	00000000

Project A

A	FFFFFFFF	Signal	In
Ainvert	0	Signal	In
B	FFFFFFFF	Signal	In
Binvert	0	Signal	In
Carry_out	1	Signal	Out
Cin	0	Signal	In
N	32	Gener...	In
Op	4	Signal	In
Overflow	0	Signal	Out
Result	00000000	Signal	Out
Zero	1	Signal	Out
s_Carry_out	FFFFFFFF	Signal	Internal
s_Overflow	00000000000000000000000000000001	Signal	Internal
s_Result	00000000000000000000000000000000	Signal	Internal
s_Set	11111111111111111111111111111110	Signal	Internal

Set less than A < B



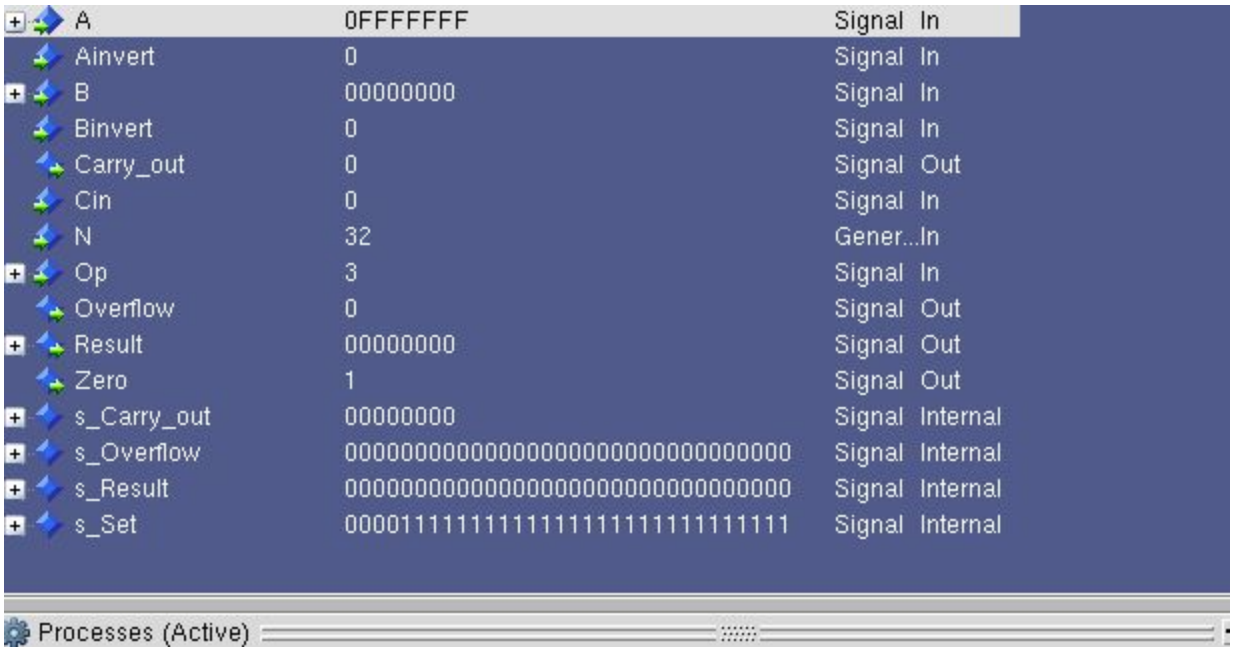
A	00000000	Signal	In
Ainvert	0	Signal	In
B	FFFFFFFF	Signal	In
Binvert	0	Signal	In
Carry_out	0	Signal	Out
Cin	0	Signal	In
N	32	Gener...	In
Op	3	Signal	In
Overflow	0	Signal	Out
Result	00000001	Signal	Out
Zero	0	Signal	Out
s_Carry_out	00000000	Signal	Internal
s_Overflow	00000000000000000000000000000000	Signal	Internal
s_Result	00000000000000000000000000000001	Signal	Internal
s_Set	11111111111111111111111111111111	Signal	Internal

Processes (Active)

Name	Type (filtered)	State	Order	Parent Path

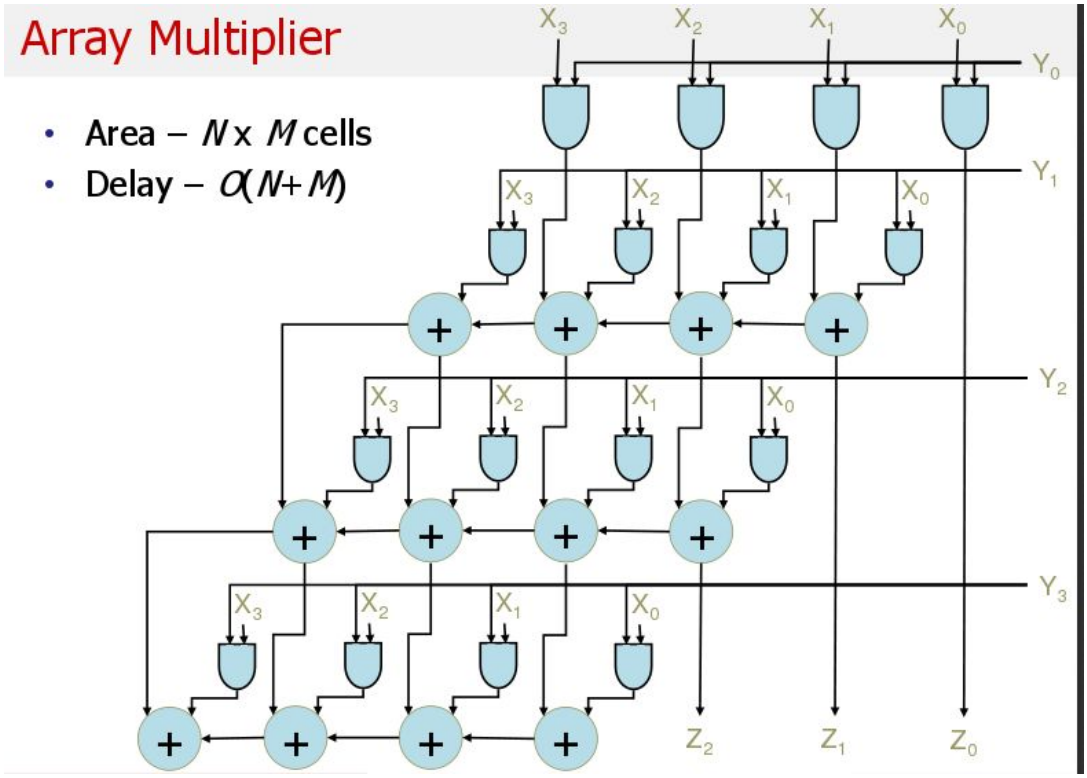
Project A

Set less than A > B



Part 3: Array Multiplier

Project A



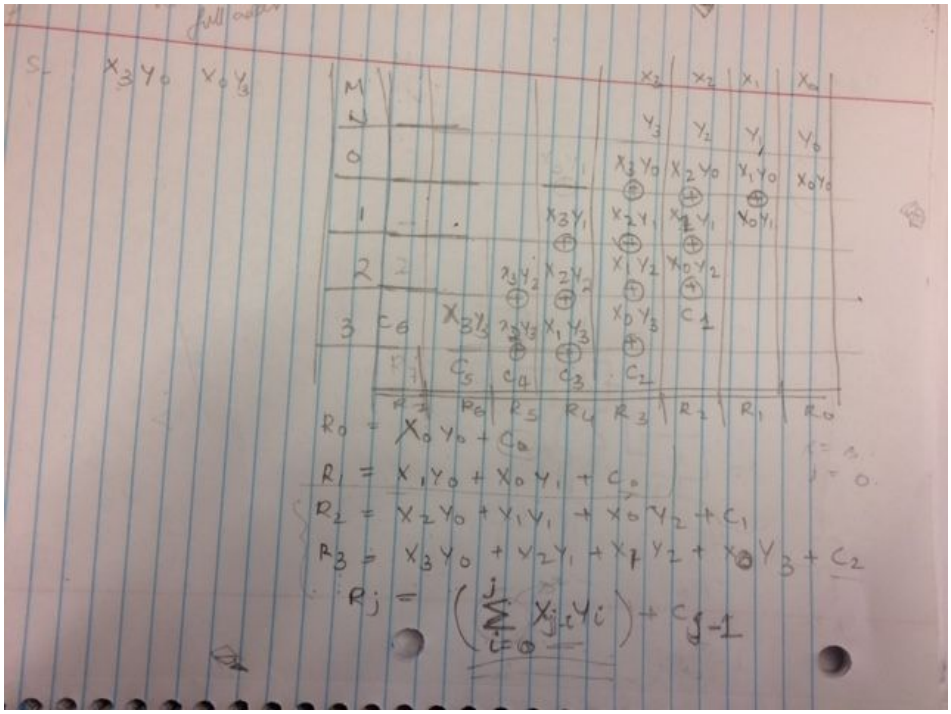
We used the multiplier example given in the professor's slides as reference.

We tried implementing the multiplier several different ways. Here are a few things we tried:

1. We set the first result obtained ($x_0 Y_0$) separately, passing it into the and gate directly. The result obtained was R_0. After that, we made a for loop to iterate through each row (thus, incrementing x) while keeping Y constant. Then we updated Y, as soon as it was done with all 32 bits of X. The problem we ran into was that we were unable to implement the carry efficiently through this method. Also, the "i" in the for loop was going out of bounds so we tried to implement a while loop to keep that in check. But we started to have a lot of trouble with variable declaration.

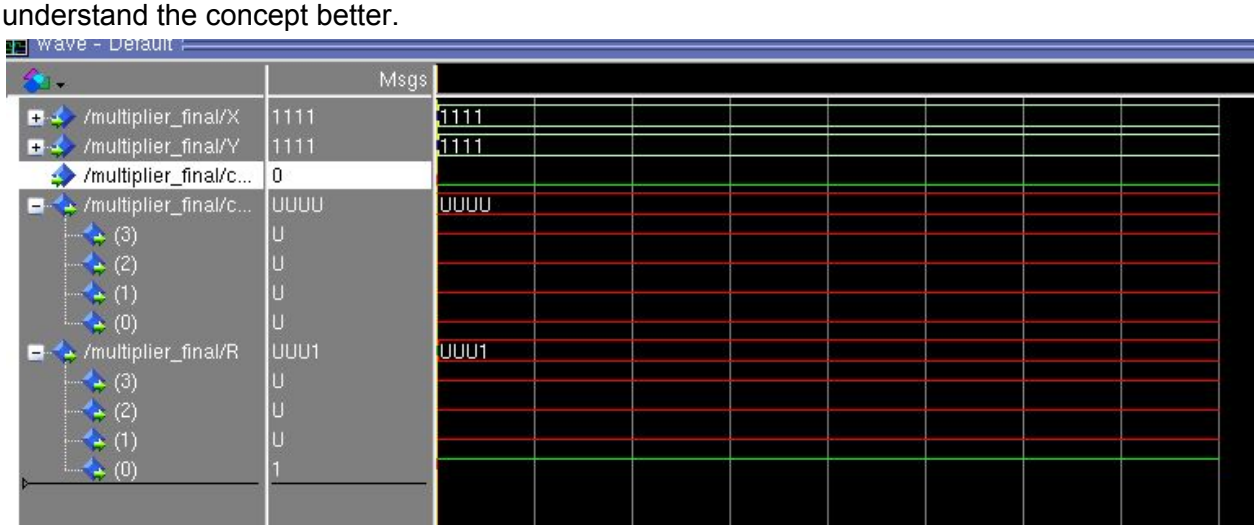
2. The next attempt, we used a completely different approach. We decided to use a nested for-loop in order to iterate through each level of the multiplier. The problem with this was that we were unable to pass in the outputs obtained from the and gates into the inputs of the full adder. Because we needed the full adder to be outside the generate-loop but the input values depended on the variables (i and j) of the generate loops. The main logic behind this implementation is described in the diagram below:

Project A

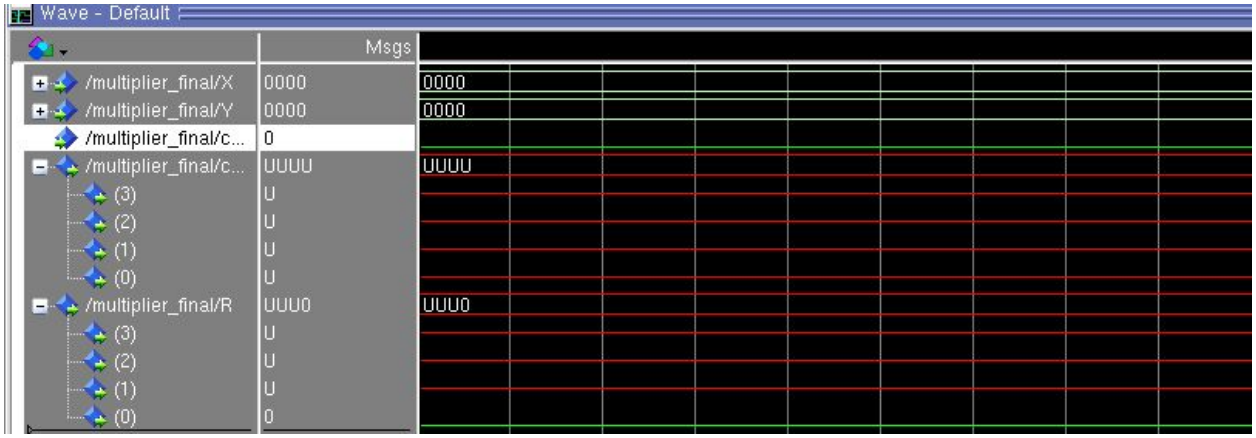


3. Another attempt was made when we decided to make the entire and gate - full adder block into a single module and then iterate through them in the final design. But again we were having troubles with passing the inputs into the full adder.

All in all, we believe our main problem lay in the fact that we were trying to implement the code in a very object-oriented manner which led us to overcomplicating the issue at hand. I believe we had the logic mapped out correctly and we just needed to code with a clear mind. I think we would be able get it working if we had just a little more time. Here is a screenshot of the output obtained. For the sake of simplicity, we have reduced this to a 4-bit multiplier in order to help us understand the concept better.



Project A

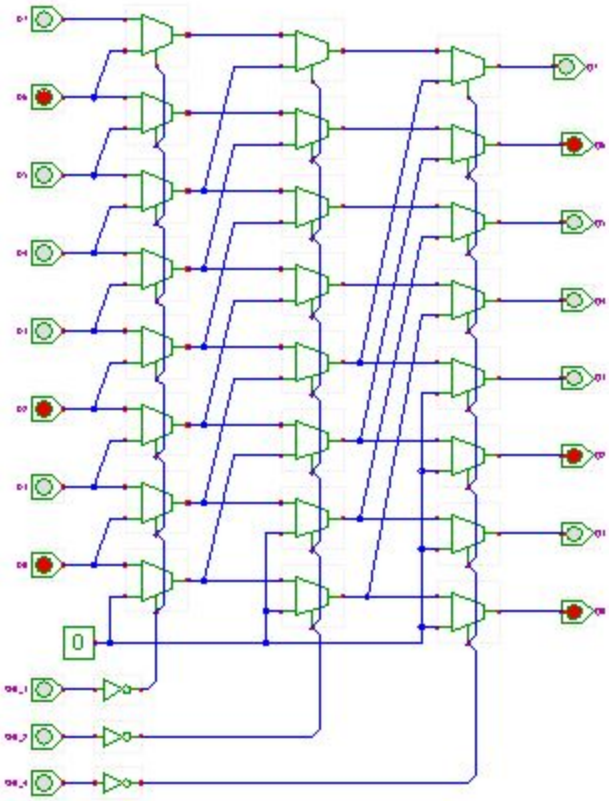


Part 4: *Barrel Shifter*

1. The difference between a srl and sra instruction is the introduction of the arithmetic value. Sra means shift right arithmetic and is concerned with signed values. Shift right logical pads the shift with zero values regardless of the shifting values sign. An arithmetic shift will pad with the MSB value of the number being shifted(0 or 1). There is no sla instruction in MIPS because you will always have to pad with zeros when shifting left, which equates to a sll. There is no way to pad with ones because we don't know what comes after the zeroth bit. Since sll already pads with zeros there is no point creating another instruction to do the same operation.

Project A

Barrel-shifter (8 bit)



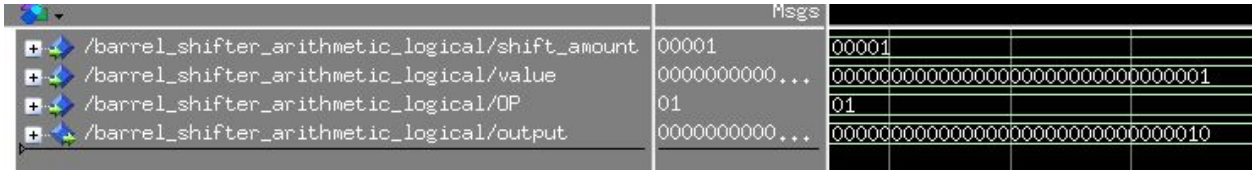
2. Above is the given diagram of a left shift barrel shifter. We used this to make a 32 bit right shifter. To accommodate for arithmetic shifts we made a zero pad shifter and a ones pad shifter. We just assumed to make both shifted values padded ones and zeroes. Both are fed into a mux and the select bit is value(31).(MSB). That mux feeds into a second paired with srl with a select signal of OP(1) which says if it's an arithmetic operation or not. That will support both arithmetic and logical shifts.

3. To support left shifts we used the right shifter and inputted a reversed shift value. Shift_amount is untouched. To get the correct output after the shift we reversed that output. We used a final mux driven by sll final value and our right mux logic, and select if its a right or left shift with OP(0).

Left Shift: sll / sla

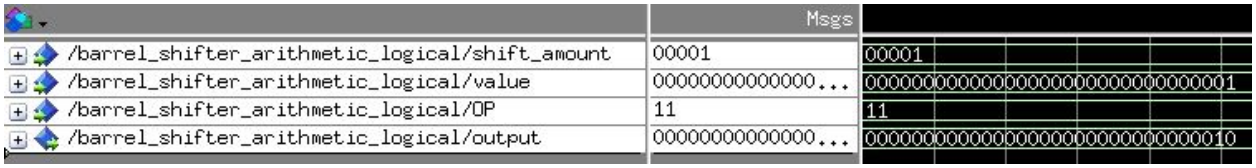
OP = "01" , "11" shift_amount:"00001"' value:x"00000001"

Project A



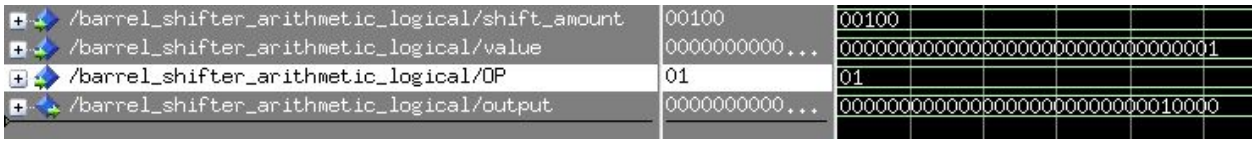
Shifting left one bit logical. We added “11” to show our else case.

OP = “01” , “11” shift_amount:”00001”’ value:x”00000001”



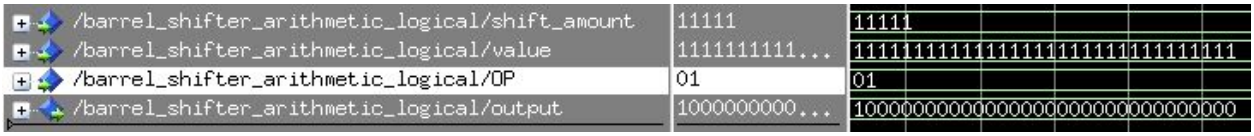
Here we prove the else case making the OP code 11. It is the same as sll.

OP = “01” shift_amount:”00100”’ value:x”00000001”



Sll with different shift amount.

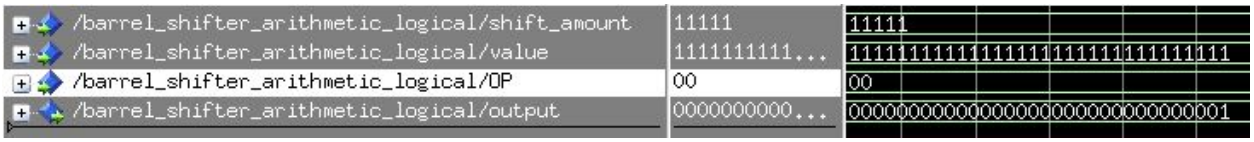
OP = “01” shift_amount:”11111”’ value:x”FFFFFFFF”



Showing max shift amount and max value passed in.

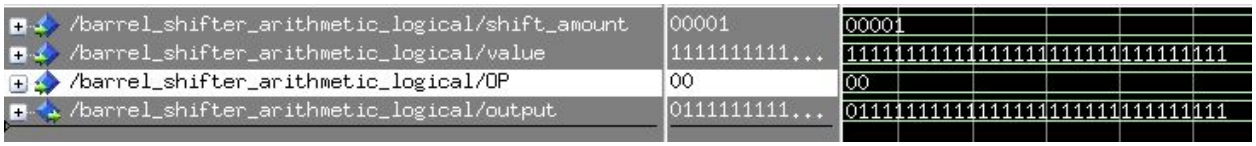
Right Shift: srl

OP = “00” shift_amount:”11111”’ value:x”FFFFFFFF”



Shifting the max amount with highest value. Shows the zero padding even though MSB is 1.

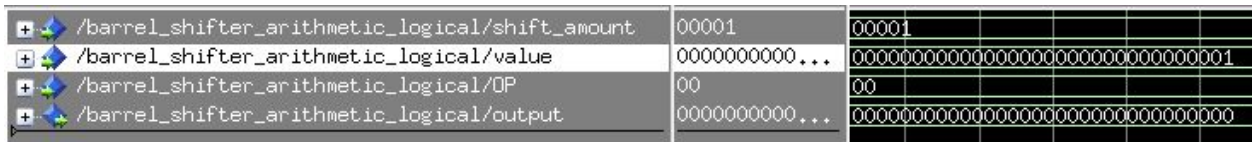
OP = “00” shift_amount:”00001”’ value:x”FFFFFFFF”



shifting right by one bit.

OP = “00” shift_amount:”00001”’ value:x”00000001”

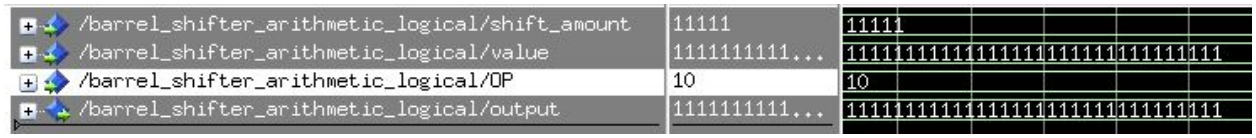
Project A



Shows off srl other edge case of sifting a value and it not rotating to the MSB position.

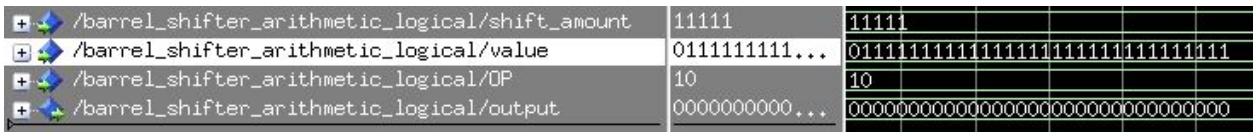
Arithmetic Right Shift: sra

OP = “10” shift_amount:”11111”’ value:x”FFFFFFFF”



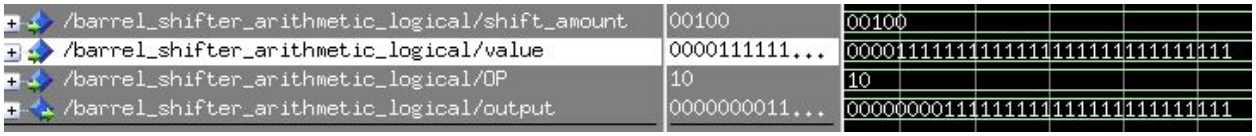
MSB = 1. Shifting highest value max amount. Shows the 1 padding resulting in no difference in the shifts.

OP = “10” shift_amount:”11111”’ value:x”7FFFFFFF”



Direct comparison showing of MSB sensitivity. MSB =0 so pads with zeros.

OP = “10” shift_amount:”00100”’ value:x”0FFFFFFF”



Shows of the zero padding when MSB = 0. Shows different shift amount.