Project 2

Part 1a (8-bit 4-to-1 multiplexer)

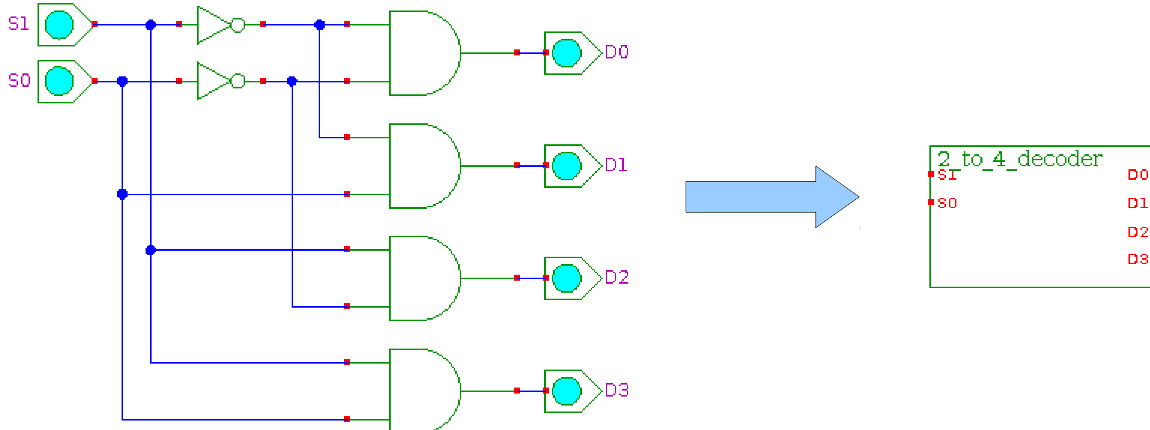
I – Design:

Part 1a-a: (2-to-4 decoder)

Gates: 2 inverters, 4 two-input AND gates

Inputs: Bits S_1 and S_0

Output: Bits D_0, D_1, D_2, and D_3, where only the bit determined by S_1S_0 is set to 1

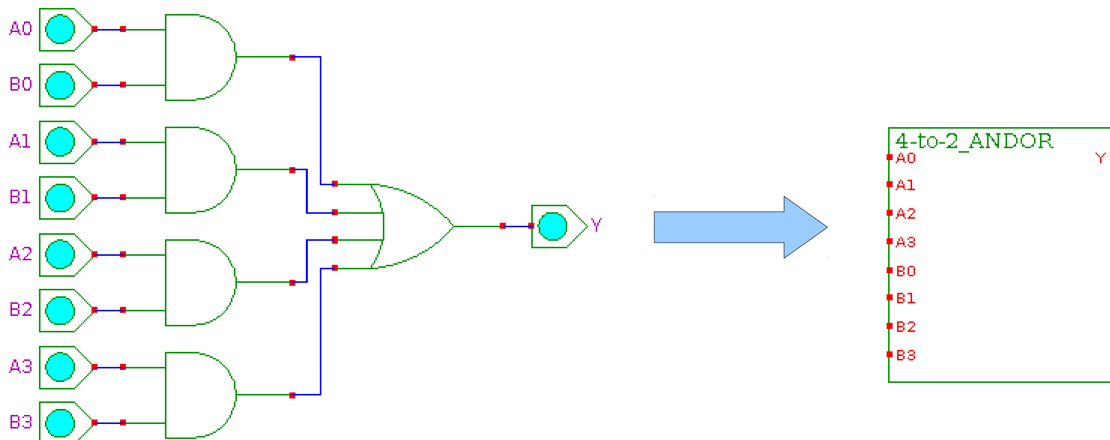


Part 1a-b: (4-to-2 ANDOR)

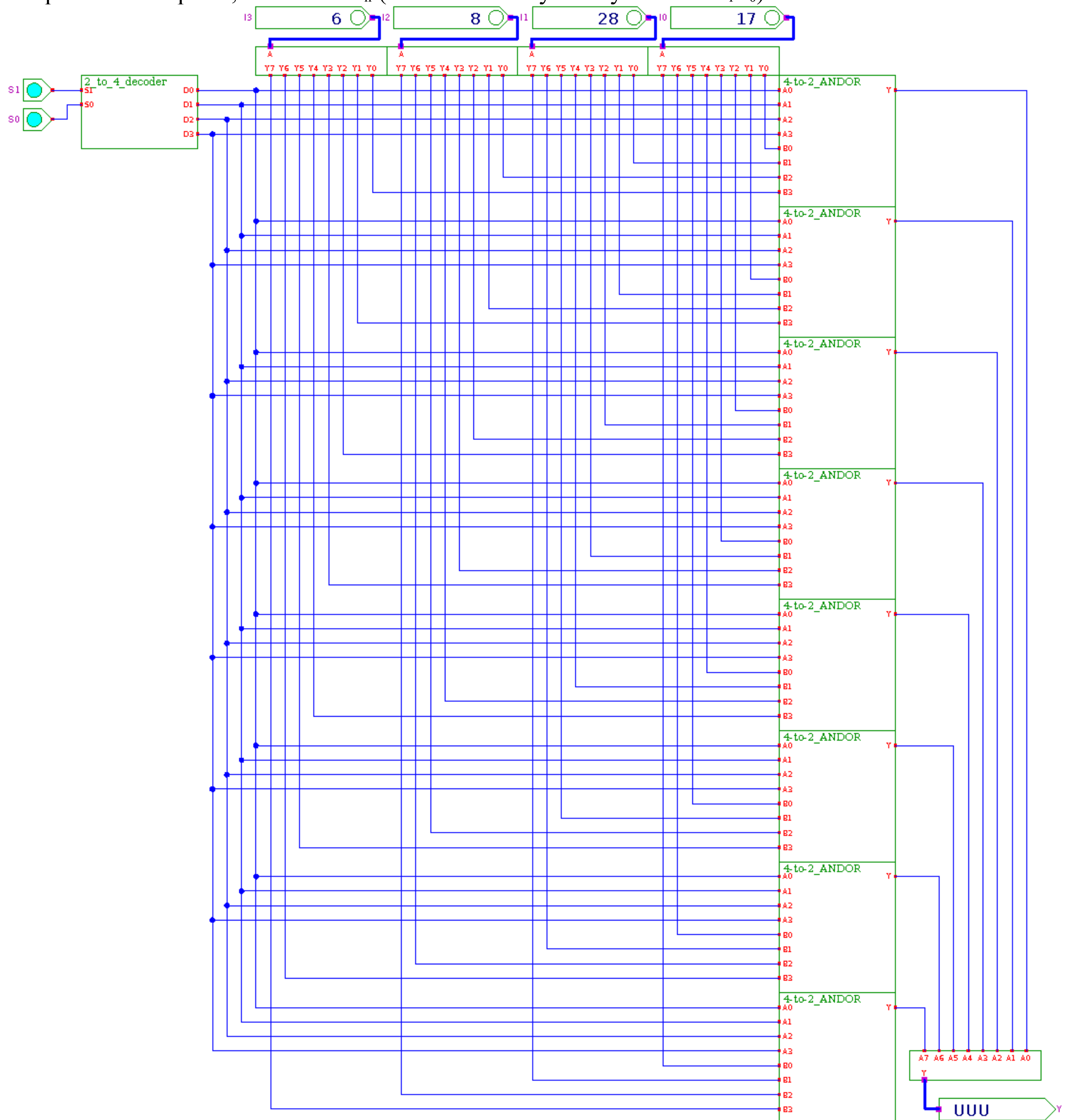
Gates: 4 two-input AND gates, 1 4-input OR gate

Inputs: Bits A_0, A_1, A_2, A_3, B_0, B_1, B_2, and B_3

Output: Bit Y, where $Y = A_0B_0 + A_1B_1 + A_2B_2 + A_3B_3$



Part 1a: (8-bit 4-to-1 multiplexer)
Gates: 1 two-to-four decoder component (part 1a-a), 8 four-to-one ANDOR components (part 1a-b)
Inputs: 8-bit inputs I_0, I_1, I_2, and I_3, and 1-bit inputs S_1 and S_0
Output: 8-bit output Y, where Y = I_n (n determined by binary number S_1S_0)

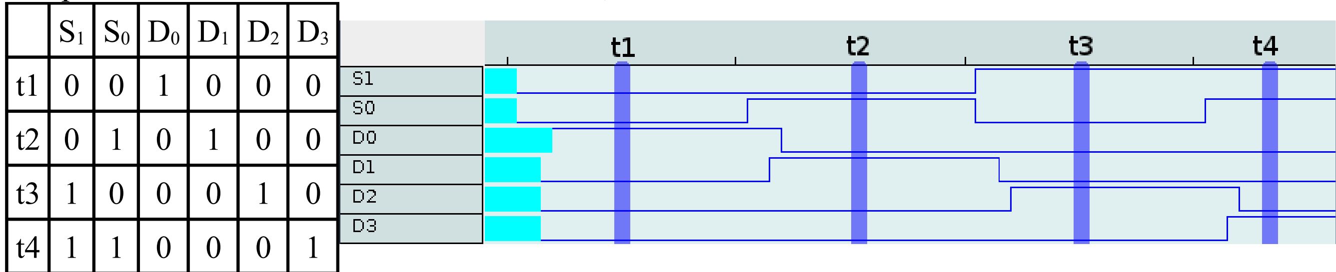


II - Reasoning:
A usual multiplexer mainly consists of an encoder; the multiplexer's various input bits are AND-ed with the encoder's output and then logically OR-ed back together again. Following that logic, an eight-bit four-to-one multiplexer ought to be just the same as a normal four-to-one multiplexer, with the exception that there out to be eight times as ANDOR gates on the input (since each input consists of

eight bits, not one). This circuit was designed with that philosophy in mind; as such it has a high gate input cost and takes up plenty of space. The gates are fairly well distributed, though, which gives the design a gate delay of 50 ns. That's not too slow considering the amount of inputs the circuit has.

III - Verification:

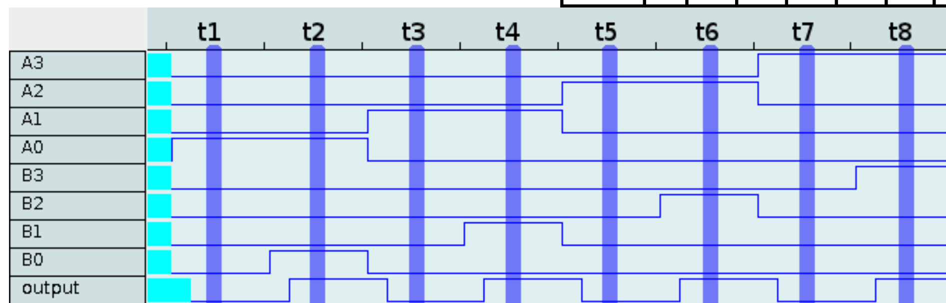
The 2-to-4 decoder has four possible input combinations, shown in the truth table below. The component's waveform matches the truth table, which means it works as it should.



	S_1	S_0	D_0	D_1	D_2	D_3
t1	0	0	1	0	0	0
t2	0	1	0	1	0	0
t3	1	0	0	0	1	0
t4	1	1	0	0	0	1

The 4-to-2 ANDOR component can be expressed logically as $Y = A_0B_0 + A_1B_1 + A_2B_2 + A_3B_3$. However, the 4-to-2 ANDOR is to be used in conjunction with the 2-to-4 decoder, which means only one of the four A inputs can be set to 1 at any given time. Because of this, the 4-to-2 ANDOR's expression can be simplified to $Y = B_n$, where n corresponds to which of the 2-to-4 decoder's outputs D_n is set to 1. Therefore, the only input combinations that matter for the 4-to-2 ANDOR are the 8 possible combinations shown in the table to the right. The waveform below matches the truth table, so this component works.

		A_3	A_2	A_1	A_0	B_3	B_2	B_1	B_0	Y
n = 0	t1	0	0	0	1	X	X	X	0	0
	t2	0	0	0	1	X	X	X	1	1
n = 1	t3	0	0	1	0	X	X	0	X	0
	t4	0	0	1	0	X	X	1	X	1
n = 2	t5	0	1	0	0	X	0	X	X	0
	t6	0	1	0	0	X	1	X	X	1
n = 3	t7	1	0	0	0	0	X	X	X	0
	t8	1	0	0	0	1	X	X	X	1



If the 2-to-4 decoder and 4-to-2 ANDOR both work, and each 4-to-2 ANDOR in the multiplexer is hooked up to the outputs of a 2-to-4 decoder as well as to the kth bits of inputs I_0, I_1, I_2, and I_3, then the entire multiplexer can be expressed logically as $Y = I_n$, where n corresponds to which of the 2-to-4 decoder's outputs D_n is set to 1. Since the output D_n in the 2-to-4 decoder is determined by S_0 and S_1, where n corresponds to the binary number S_1S_0, the multiplexer can also be expressed as $Y = I_n$, where n corresponds to the binary number S_1S_0. Coincidentally, that's the exact specification for an 8-bit 4-to-1 multiplexer. It was proven earlier that the 2-to-4 decoder and 4-to-2 ANDOR components do indeed both work, which means that this design for an 8-bit 4-to-1 multiplexer also works. To err on the side of caution, here are is a truth table with a handful of input combination and their corresponding output, as well as a matching waveform. In case there was any doubt, yes, this component works.

	S_1	S_0	I_0	I_1	I_2	I_3	S
t1	0	0	14	255	128	47	14
t2	0	1	14	255	128	47	255
t3	1	0	14	255	128	47	128
t4	1	1	14	255	128	47	47

Part 1b (8-bit adder)

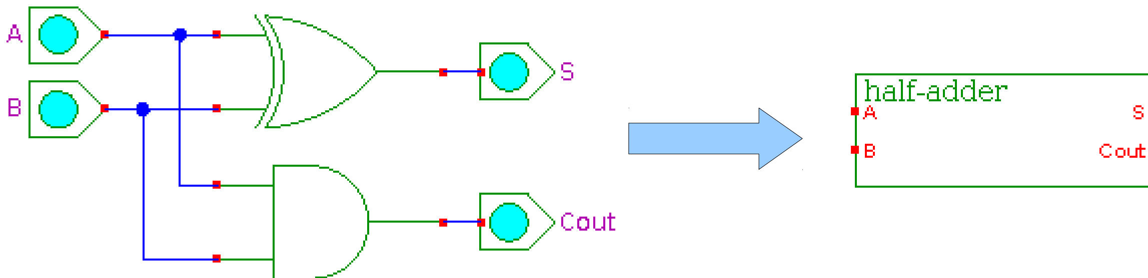
I – Design:

Part 1b-a: (half adder)

Gates: 1 two-input XOR gate, 1 two-input AND gate

Inputs: Bits A and B

Outputs: Bits S and C_{out}, where the binary number $C_{out}S$ is equal to A plus B

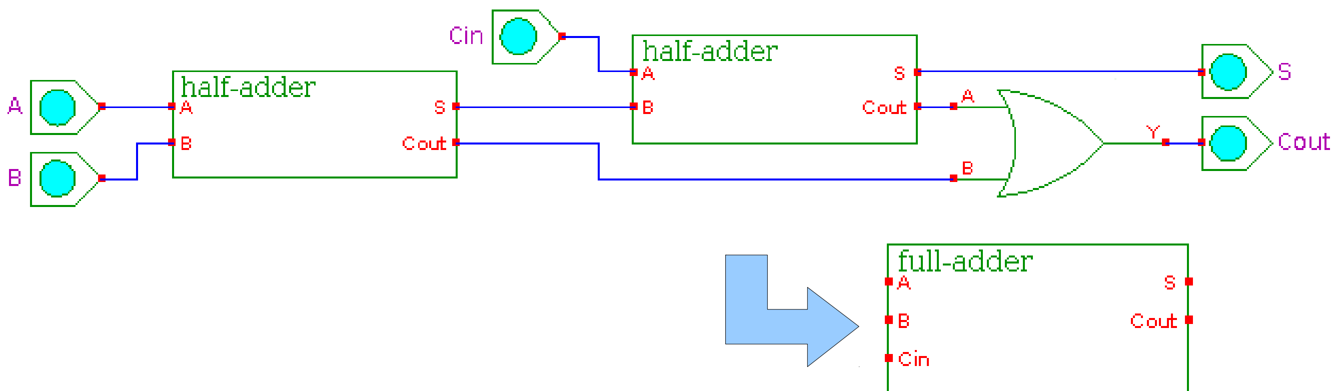


Part 1b-b: (full adder)

Gates: 1 two-input OR gate, 2 half adder components (part 1b-a)

Inputs: Bits A, B, and C_{in}

Outputs: Bits S and C_{out}, where the binary number $C_{out}S$ is equal to A plus B plus C_{in}

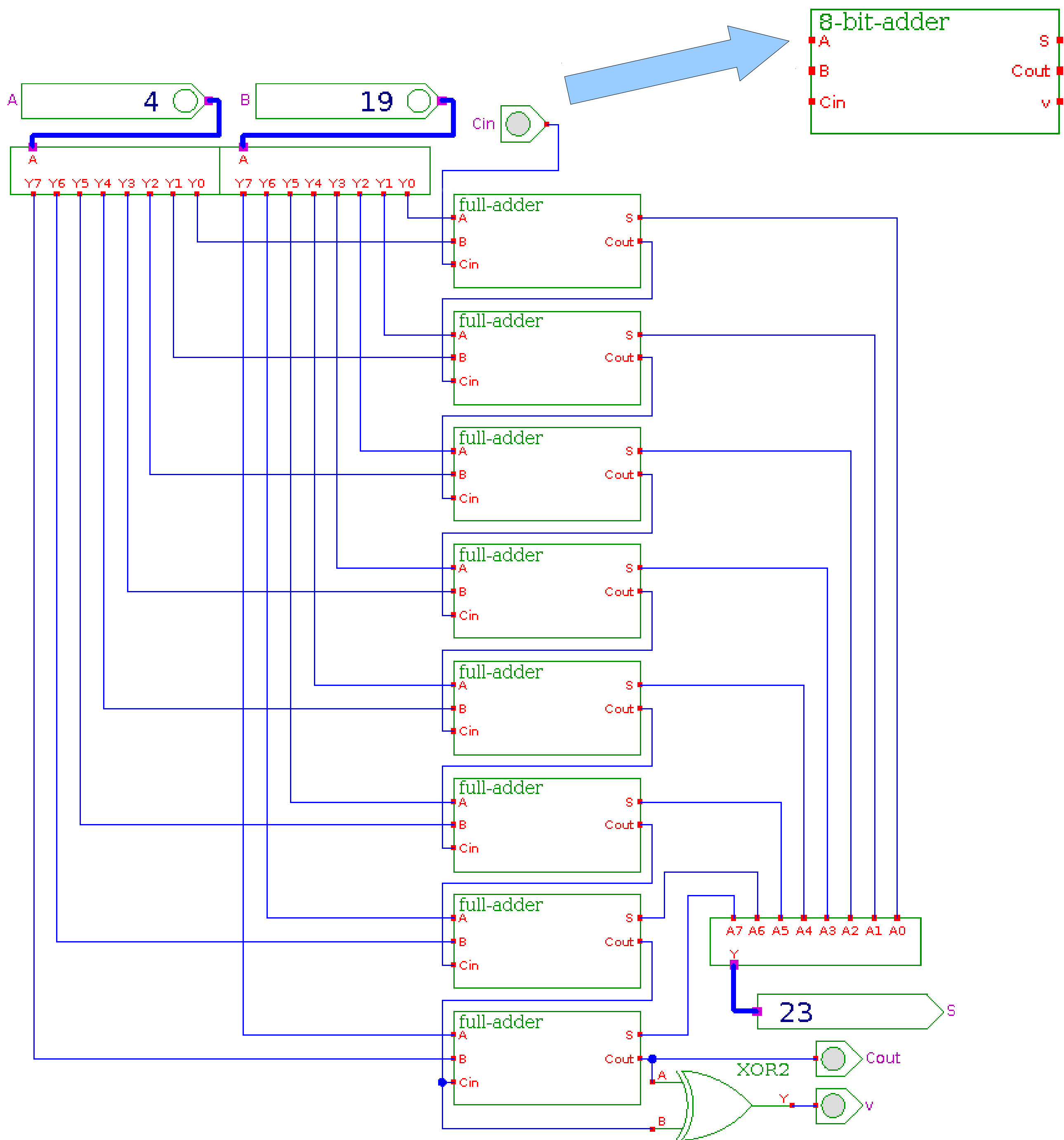


Part 1b: (8-bit adder)

Gates: 2 expander components, 1 mergebits component, 8 full adder components (part 1b-b)

Inputs: 2 eight-bit inputs A and B, and 1 single-bit input C_{in}

Outputs: 1 eight-bit output S, and 2 single-bit outputs C_{out} and v, where the binary number $C_{out}S_7S_6S_5S_4S_3S_2S_1S_0$ is equal to the sum of A, B, and C_{in} (S_n is the nth bit of S) and v indicates if there was signed (2's complement) overflow



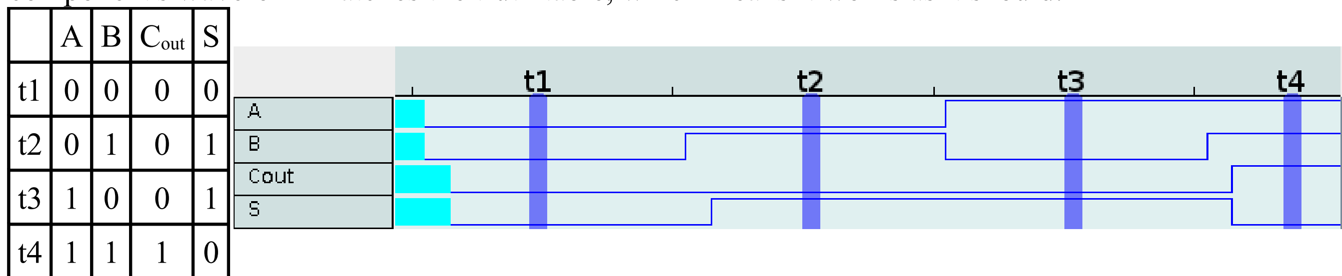
II - Reasoning:

This particular design gets the job done, but it isn't too efficient. Since carrying is implemented by daisy-chaining the full adders' carry-in/carry-out bits, each full adder has to wait for the previous one to

output before it can do anything. Because of this timing irregularity, the circuit's output doesn't necessarily change all at once to reflect changes in input. Instead, it might fluctuate until all the full adders finish carrying. It's terribly slow and gives the circuit a gate delay ranging from 40 to 160 ns. The design's main advantage, however, is its simplicity; it doesn't use many components and is fairly easy to implement. Since the sum of two eight-bit numbers might exceed the possible values that eight bits can hold, an extra output bit (C_{out}) is provided. If the sum produced is greater than 255 (the highest value an unsigned eight-bit number can hold), the sum value will overflow and the extra output bit will be set to 1 (which indicates that the true sum is actually 256 more than what the output suggests). The circuit can also process signed numbers in 2's complement encoding, since adding numbers in 2's complement encoding is almost identical to adding unsigned numbers. If the input is signed, overflow is indicated by v. A carry-in input (C_{in}) is also provided, which allows for more complex calculations (such as incrementing the sum of A and B by one if a certain external condition is met). If such extra input isn't needed, C_{in} can just be set to zero and the circuit defaults to calculating A plus B.

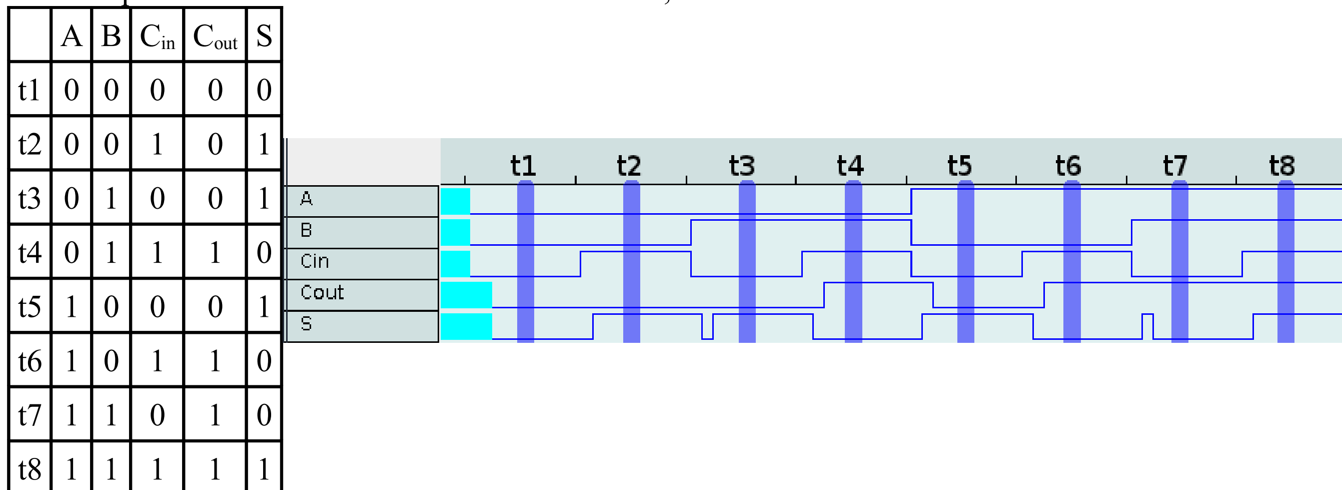
III - Verification:

The half adder component has four possible input combinations, shown in the truth table below. The component's waveform matches the truth table, which means it works as it should.



	A	B	C_{out}	S
t1	0	0	0	0
t2	0	1	0	1
t3	1	0	0	1
t4	1	1	1	0

The the full adder component has eight possible input combinations, shown in the truth table below. The component's waveform matches the truth table, which means it works as it should.

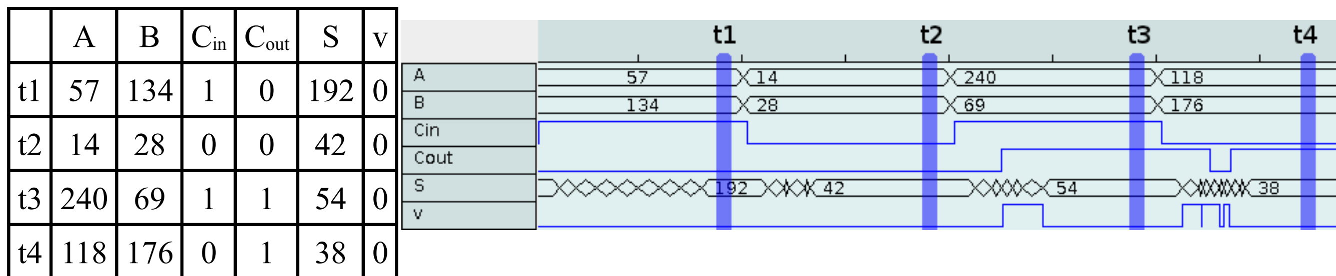


	A	B	C_{in}	C_{out}	S
t1	0	0	0	0	0
t2	0	0	1	0	1
t3	0	1	0	0	1
t4	0	1	1	1	0
t5	1	0	0	0	1
t6	1	0	1	1	0
t7	1	1	0	1	0
t8	1	1	1	1	1

The topmost full adder's C_{in} input bit is connected to the 8-bit adder's own C_{in} input bit, and its other two inputs are bits 0 of A and B. If the full adder works correctly, its output should be A_0 plus B_0 plus C_{in}. That full adder's C_{out} output bit is then fed into the next full adder's C_{in}, and its other two inputs are set to bits 1 of A and B. This means the second full adder computes A_1 plus B_1, plus the carry-out bit obtained from adding A_0, B_0, and C_{in}. The rest of the full adders follow this pattern as well, basing their own C_{in} off of the computations of the previous full adders. Because of all this daisy-chaining, the entire circuit's output S is determined by the expression $S_7 = A_7 + B_7 +$ the carry-out from computing

($S_6 = A_6 + B_6$ + the carry-out from computing ($S_5 = A_5 + B_5$ + the carry-out from computing ($S_4 = A_4 + B_4$ + the carry-out from computing ($S_3 = A_3 + B_3$ + the carry-out from computing ($S_2 = A_2 + B_2$ + the carry-out from computing ($S_1 = A_1 + B_1$ + the carry-out from computing ($S_0 = A_0 + B_0 + C_{in}$))))))), and the entire circuit's C_{out} is the carry-out outputted from the last full adder (which is obtained from computing S_7). This expression can be further simplified to $Y = A + B + C_{in}$, where Y is the 9-bit binary number $C_{out}S_7S_6S_5S_4S_3S_2S_1S_0$. Output v is determined by XOR-ing the last two carry-bits in the operation. Since that is exactly how overflow is determined when adding 2's complement encoded numbers, that means v determines if there was signed 2's complement overflow when adding A and B. This proves that the circuit computes the sum of A, B, and C_{in} and accounts for signed 2's complement overflow. Since that's what the circuit is supposed to do, that means the circuit works as intended. Just to make certain, though, here are a few test cases showing the circuit's desired outputs versus its actual waveforms.

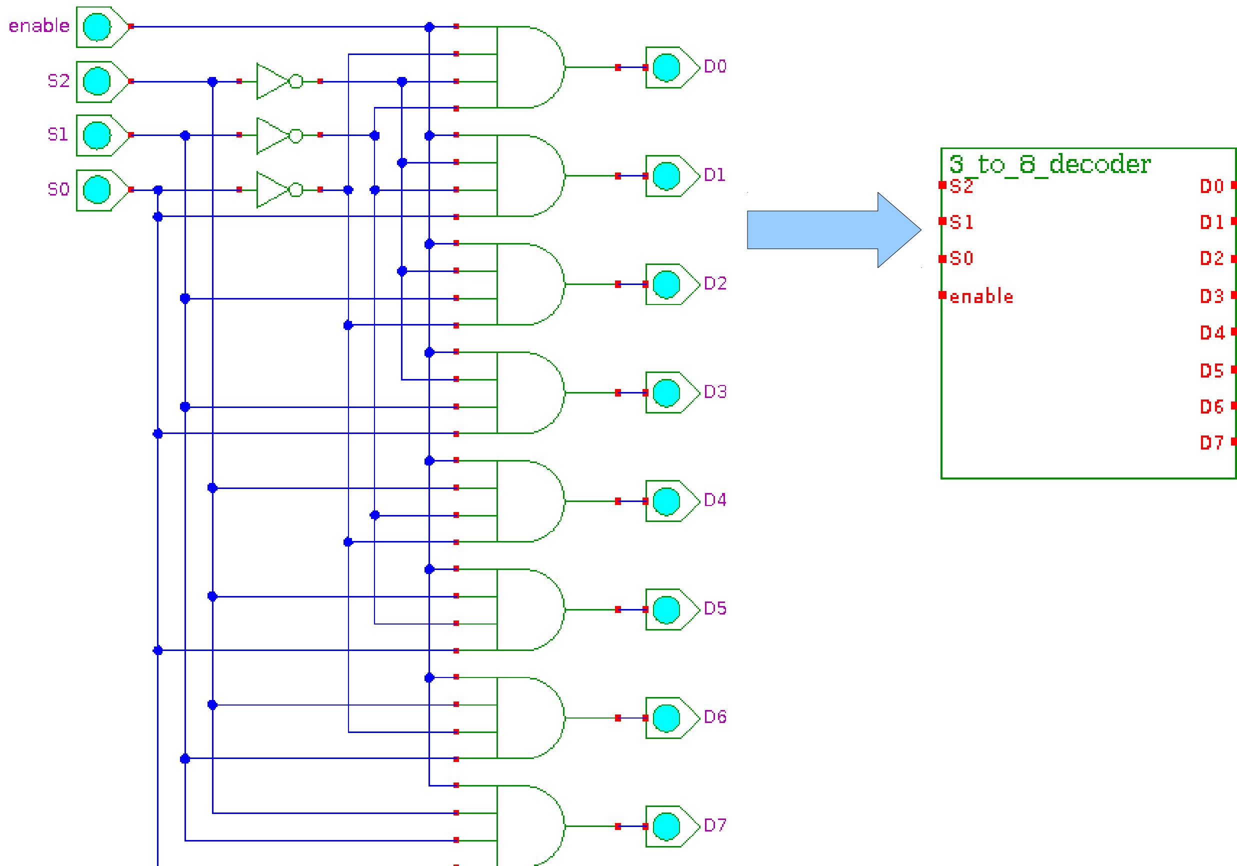
	A	B	C_{in}	C_{out}	S	v
t1	57	134	1	0	192	0
t2	14	28	0	0	42	0
t3	240	69	1	1	54	0
t4	118	176	0	1	38	0



The waveform matches the truth table, so the circuit works properly (albeit very slowly).

Part 2a (8-bit ALU)

I – Design:
Part 2a-a: (3-to-8 decoder)
Gates: 8 four-input AND gates, 3 inverters
Inputs: Bits S_2, S_1, S_0, and enable
Output: Bits D_0, D_1, D_2, D_3, D_4, D_5, D_6, and D_7, where only the bit determined by $S_2S_1S_0$ is set to 1
If enable is 0, all the outputs default to 0

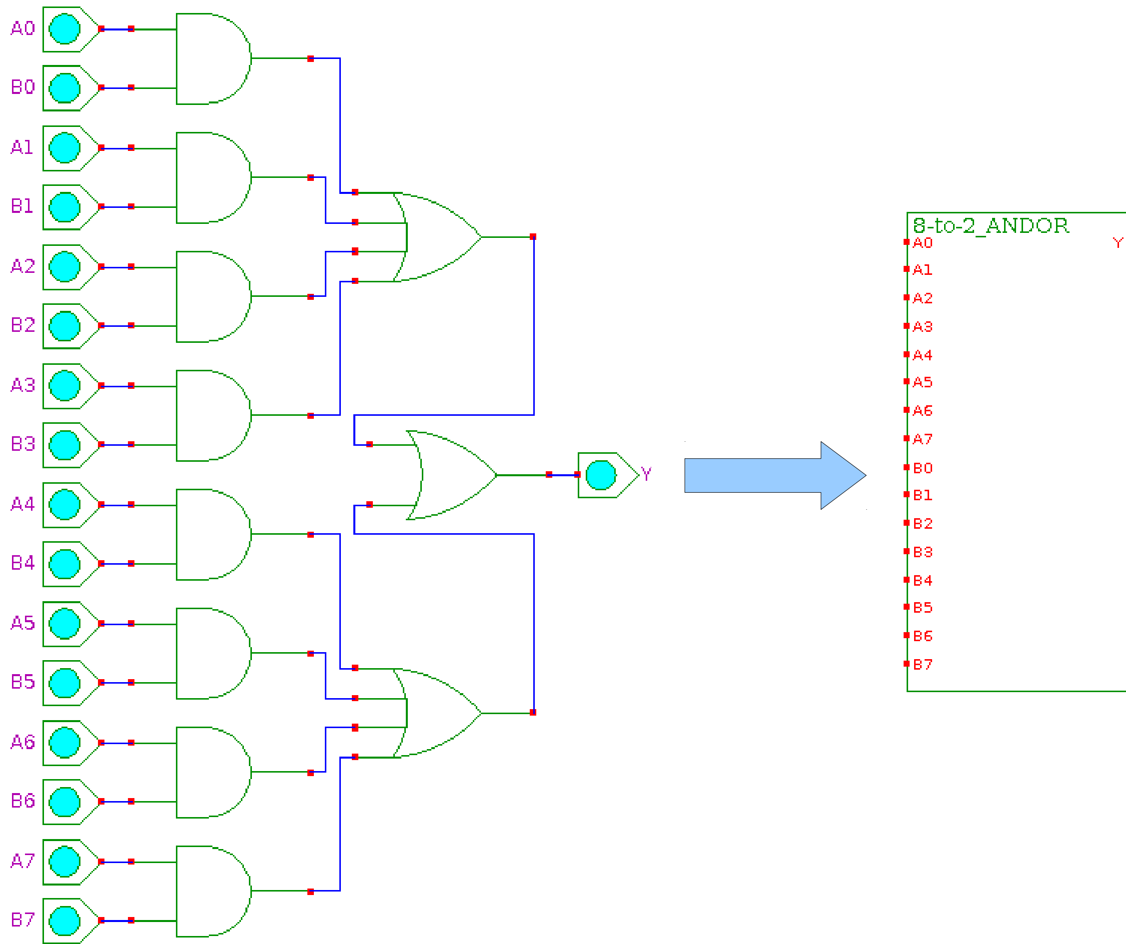


Part 2a-b: (8-to-2 ANDOR)

Gates: 8 two-input AND gates, 2 four-input OR gates, 1 two-input OR gate

Inputs: Bits A_0, A_1, A_2, A_3, A_4, A_5, A_6, A_7, B_0, B_1, B_2, B_3, B_4, B_5, B_6, and B_7

Output: Bit Y, where $Y = A_0B_0 + A_1B_1 + A_2B_2 + A_3B_3 + A_4B_4 + A_5B_5 + A_6B_6 + A_7B_7$



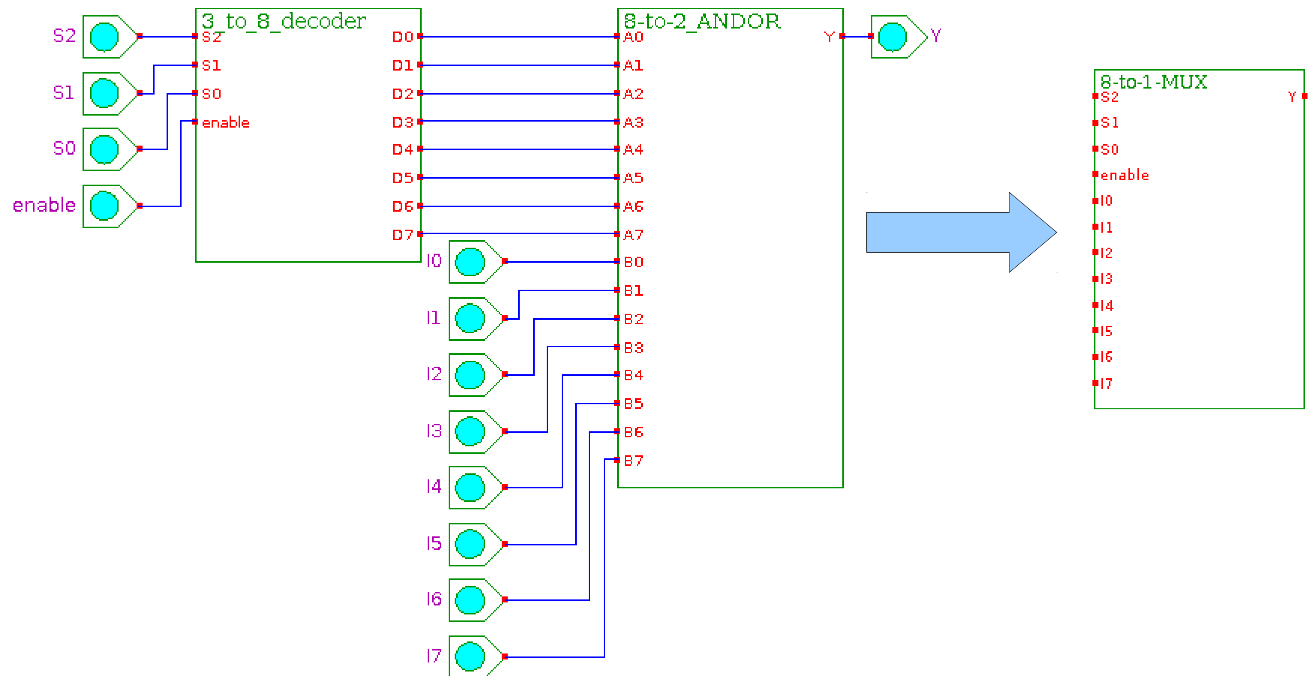
Part 2a-c: (8-to-1 multiplexer)

Gates: 1 3-to-8 decoder component (part 2a-a), 1 8-to-2 ANDOR component (part 2a-b)

Inputs: Bits S_2, S_1, S_0, I_0, I_1, I_2, I_3, I_4, I_B, I_6, and I_7

Output: Bit Y, where Y = I_n (n determined by the binary number $S_2S_1S_0$)

If enable is set to 0, Y defaults to 0



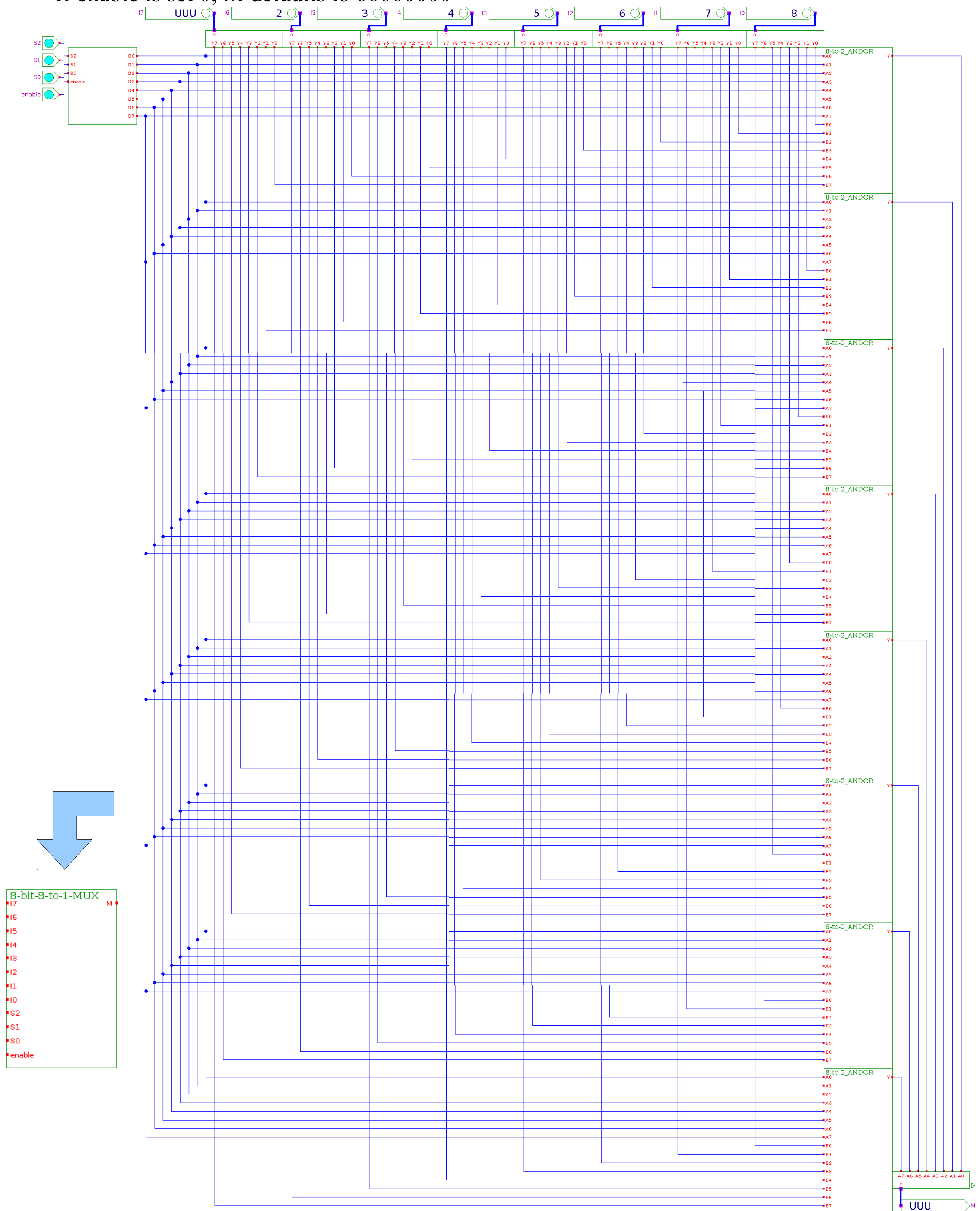
Part 2a-d: (8-bit 8-to-1 multiplexer)

Gates: 1 3-to-8 decoder component (part 2a-a), 8 8-to-2 ANDOR components (part 2a-b)

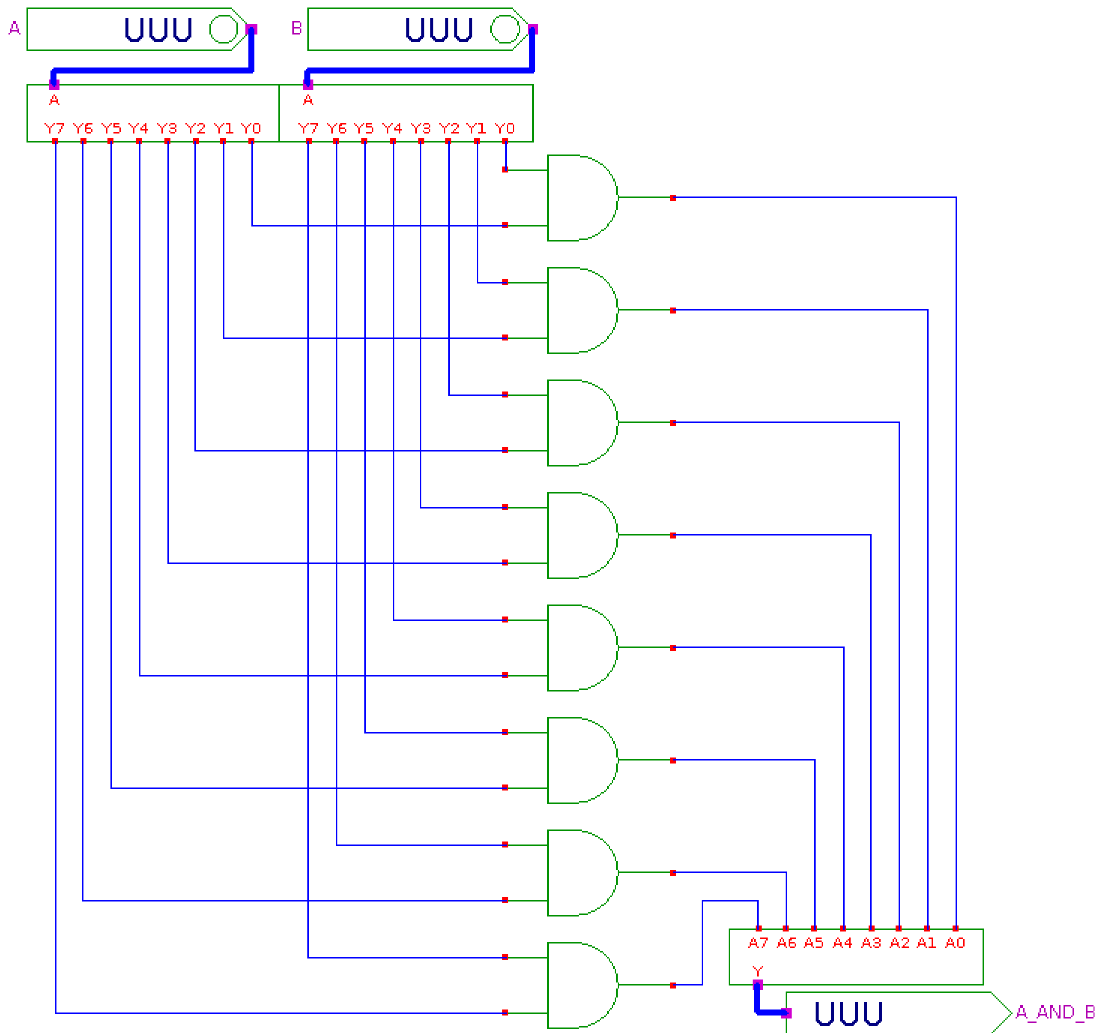
Inputs: 8-bit inputs I_0, I_1, I_2, I_3, I_4, I_5, I_6, and I_7, and 1-bit inputs S_2, S_1, S_0, and enable

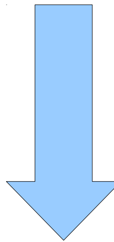
Output: 8-bit output M, where $M = I_n$ (n determined by binary number $S_2S_1S_0$)

If enable is set 0, M defaults to 00000000

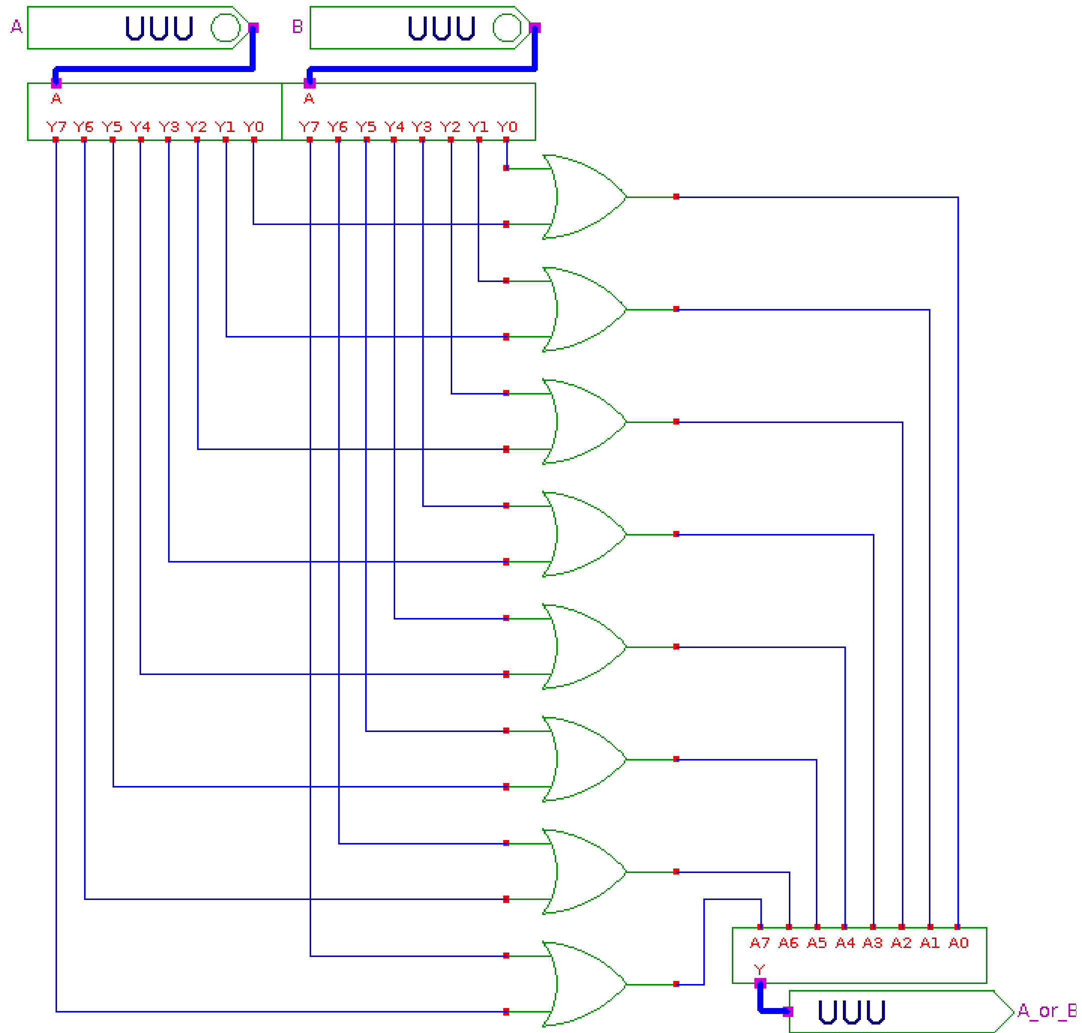


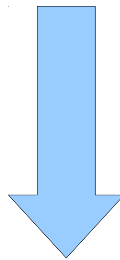
Part 2a-e: (8-bit AND)
Gates: 8 two-input AND gates
Inputs: 8-bit inputs A and B
Output: 8-bit output A_AND_B, where A_AND_B is the logical-AND of A and B

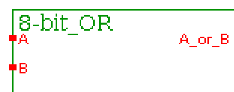




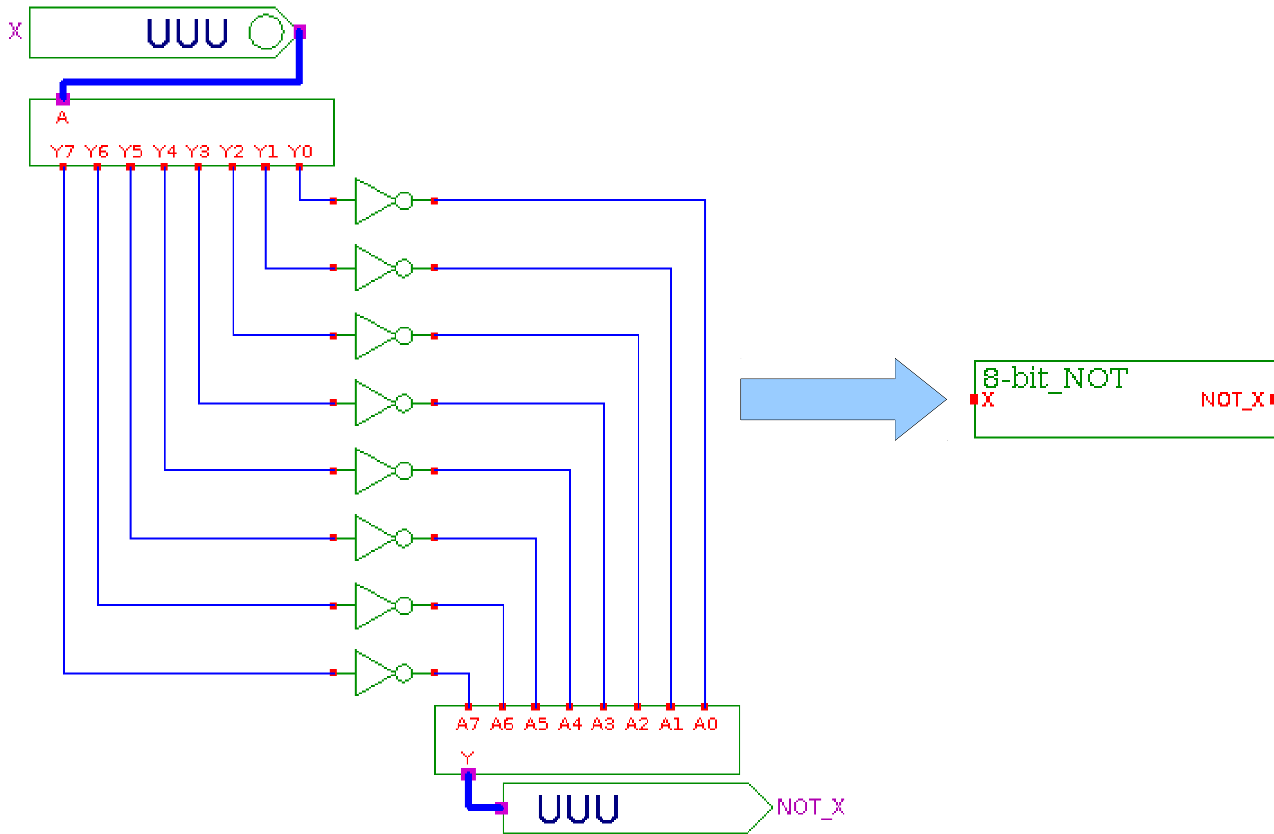
Part 2a-f: (8-bit OR)
Gates: 8 two- input OR gates
Inputs: 8-bit inputs A and B
Output: 8-bit output A_or_B, where A_or_B is the logical-OR of A and B



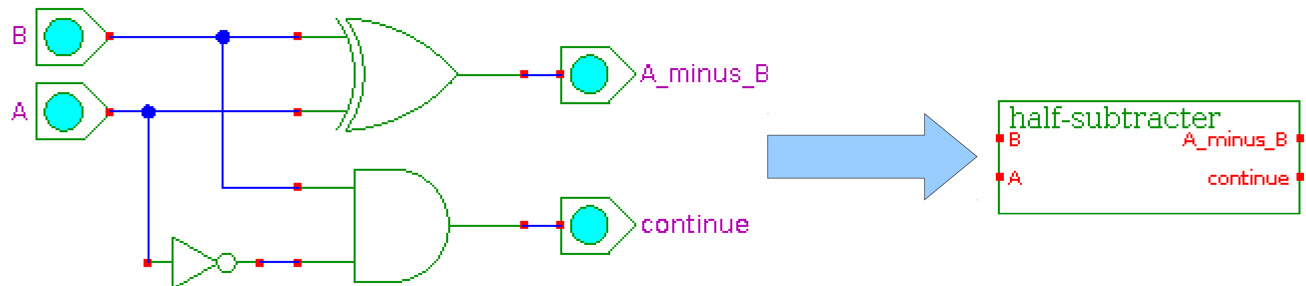




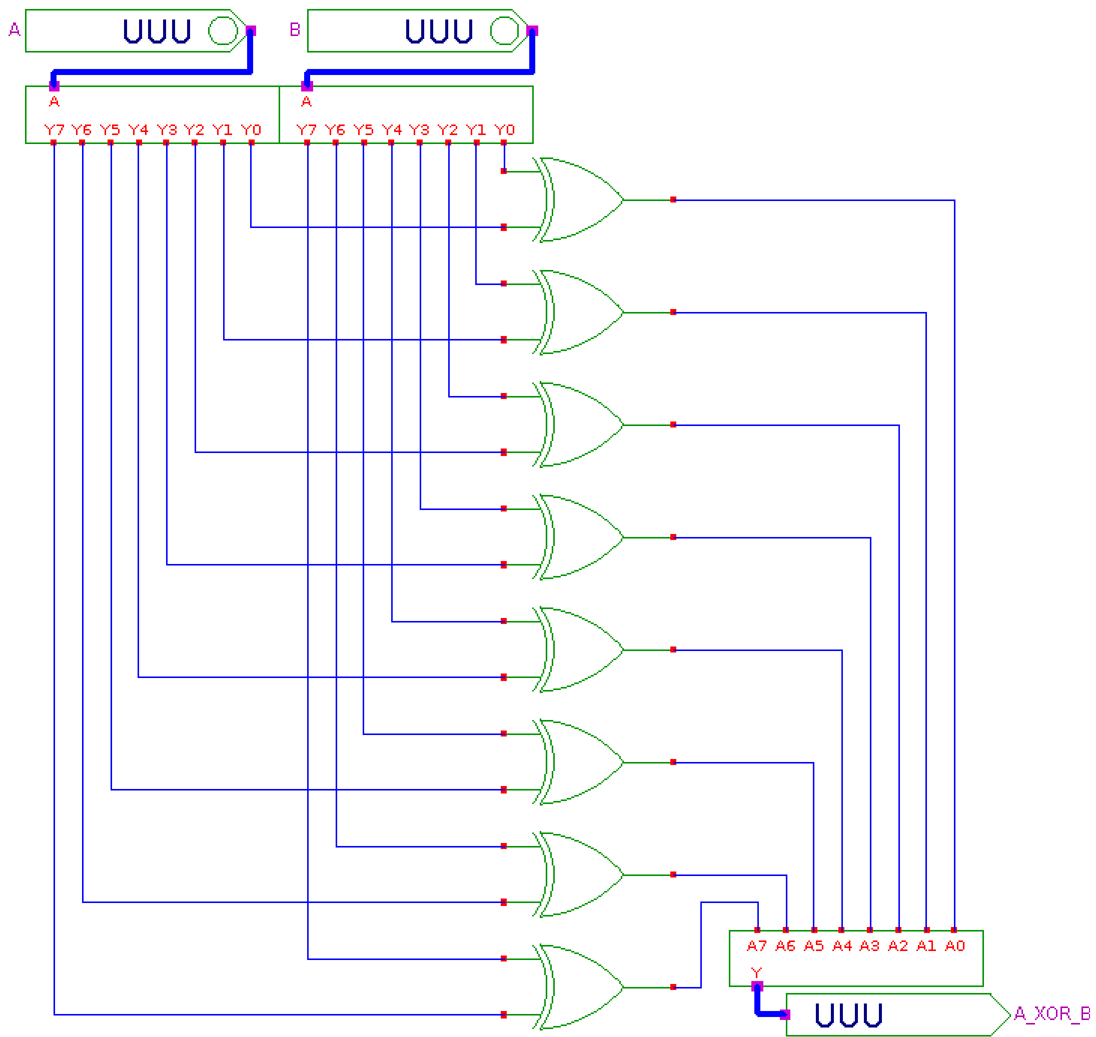
Part 2a-g: (8-bit NOT)
Gates: 8 inverters
Inputs: 8-bit input X
Output: 8-bit output NOT_X, where NOT_X is the logical-NOT of X

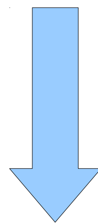


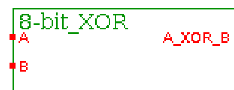
Part 2a-h: (half-subtracter)
Gates: 1 2-input XOR gate, 1 two-input AND gate
Inputs: 1-bit inputs A and B
Output: 1-bit outputs A_minus_B and continue, where A_minus_B = A – B, and continue is set to 1 when A – B produces underflow



Part 2a-i: (8-bit XOR)
Gates: 8 two-input XOR gates
Inputs: 8-bit inputs A and B
Output: 8-bit output A_XOR_B, where A_XOR_B is the logical-XOR of A and B





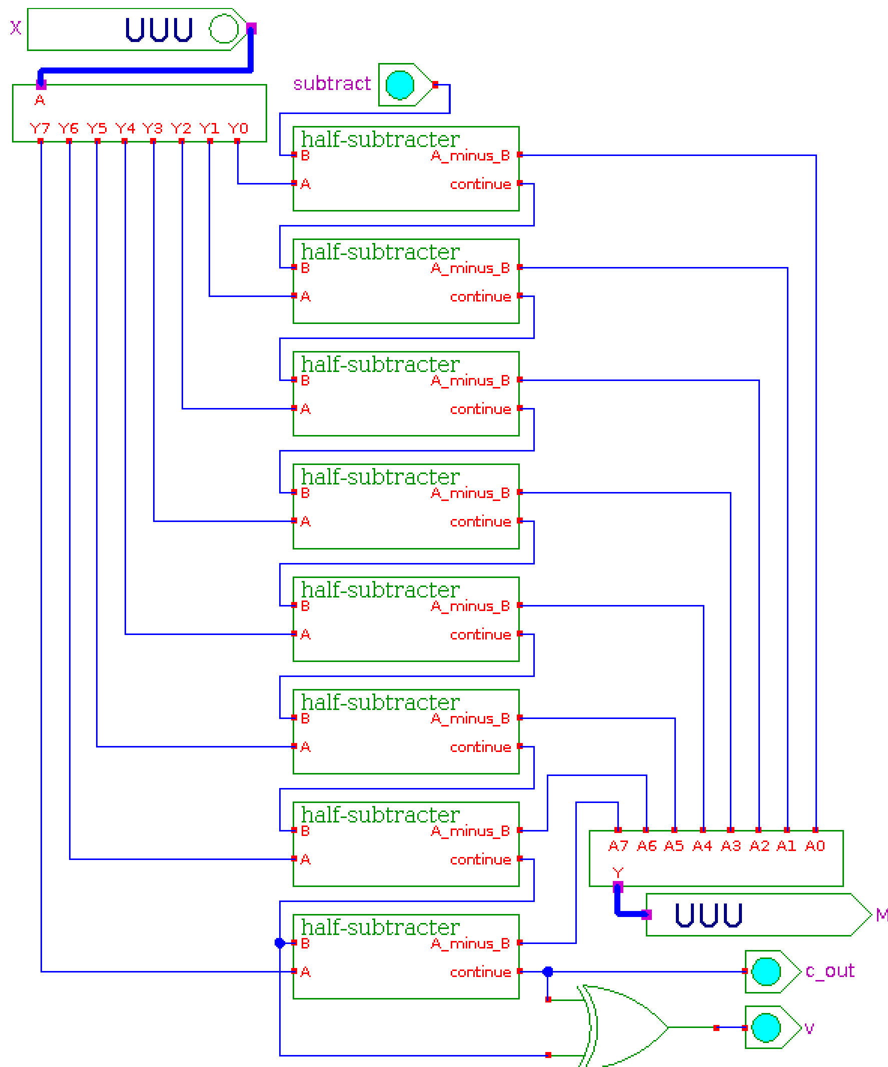


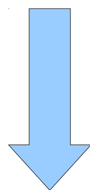
Part 2a-j: (decrementer)

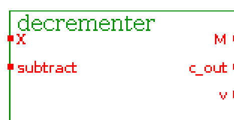
Gates: 8 half-subtracter components (part 2a-h)

Inputs: 1 8-bit input X, and 1-bit input subtract

Output: 1 eight-bit output X, and 2 single-bit outputs C_{out} and v, where the binary number $C_{out}X_7X_6X_5X_4X_3X_2X_1X_0$ is equal to X - subtract (X_n is the nth bit of X) and v indicates if there was signed (2's complement) overflow





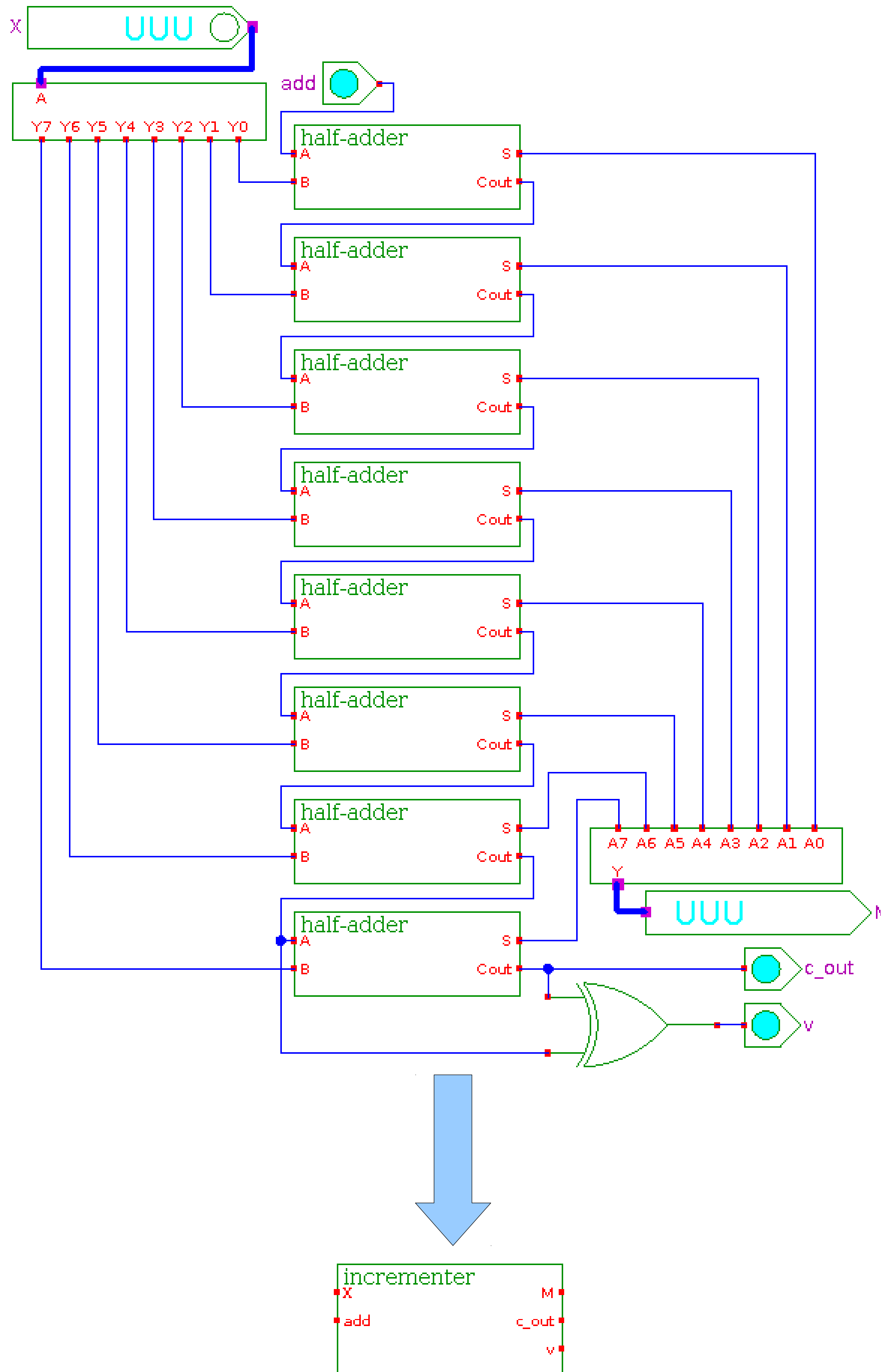


Part 2a-k: (incrementer)

Gates: 8 half-adder components (part 1b-a)

Inputs: 1 8-bit input X, and 1-bit input add

Output: 1 eight-bit output X, and 2 single-bit outputs C_{out} and v, where the binary number $C_{out}X_7X_6X_5X_4X_3X_2X_1X_0$ is equal to X plus subtract (X_n is the nth bit of X) and v indicates if there was signed (2's complement) overflow



Part 2a: (8-bit ALU)

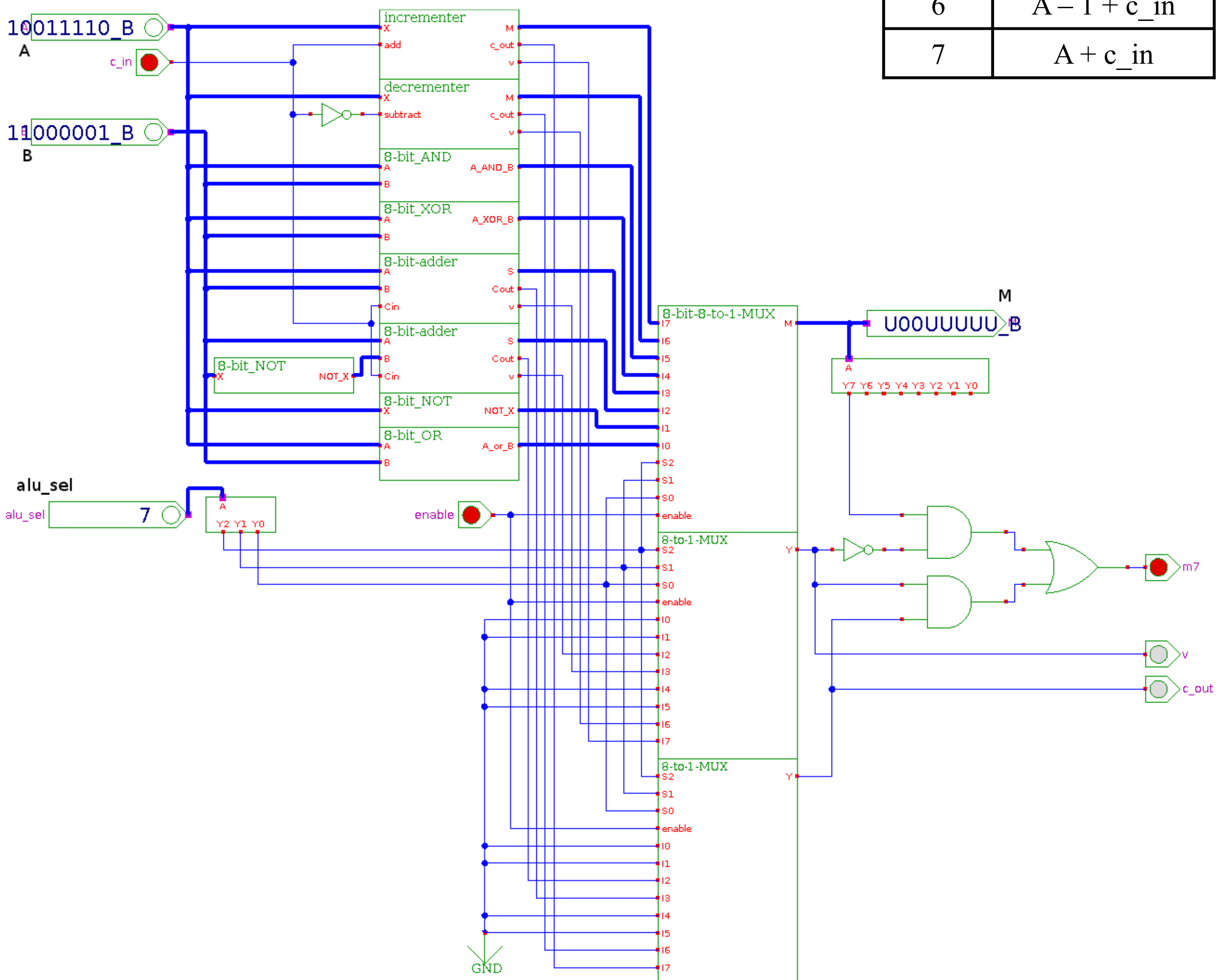
Gates: 1 8-bit adder component (part 1b), 2 8-to-1 multiplexer components (part 2a-c),
1 8-bit 8-to-1 multiplexer component (part 2a-d), 1 8-bit AND component (part 2a-e),
1 8-bit OR component (part 2a-f), 2 8-bit NOT components (part 2a-g),
1 8-bit XOR component (part 2a-i), 1 decrementer component (part 2a-j),
1 incrementer component (part 2a-k), 2 inverters, 2 two-input AND gates, 1 two-input OR gate

Inputs: 8-bit inputs A and B, 1-bit inputs c_in and enable, and 3-bit input alu_sel

Output: 8-bit output M, and 1-bit outputs m7, v, and out; outputs are determined as shown below

Name	Description
alu_sel	Chooses operation ALU performs
enable	Enables ALU
M	Result of operation performed
m7	Sign bit if signed numbers
v	Overflow assuming signed numbers
c_out	Carry out

alu_sel	action
0	A OR B
1	not(A)
2	A + not(B) + c_in
3	A + B + c_in
4	A XOR B
5	A AND B
6	A – 1 + c_in
7	A + c_in



II – Reasoning:
This design relies heavily on multiplexers, which seems adequate considering how the circuit's behavior depends entirely upon the value for alu_sel (if alu_sel is 0 compute A OR B, if alu_sel is 1 compute not(A), etc). Each of the ALU's operations is handled individually in a separate component, and the result of each operation is fed into the multiplexer. This means that the circuit computes all eight operations at once, but only the result of the desired operation gets passed to the output.

The 8-to-1 multiplexer components that produce outputs v and c_out are connected to GND on inputs I_0, I_1, I_4, and I_5. This is because the operations performed by the ALU when alu_sel is set to 0, 1, 4, or 5 are purely logical (OR, NOT, XOR, and AND, respectively), which means they don't produce any sort of overflow or perform any carrying. Due to that, v and c_out can be safely set to 0 for those values of alu_sel.

Notice the use of the decrementer and incrementer components to compute A - 1 + c_in and A + c_in, respectively. These components work very quickly, but they could have just as easily been replaced with 8-bit adders, where one of the adder's inputs was fixed (set to 0 to compute A + c_in, and set to -1 – in 2's complement – to compute A - 1 + c_in). However, as described in the Reasoning section for part 1b (pg. 6), the 8-bit adder component is very slow. By using incrementer/decrementer components instead, A - 1 + c_in and A + c_in can be computed much faster.

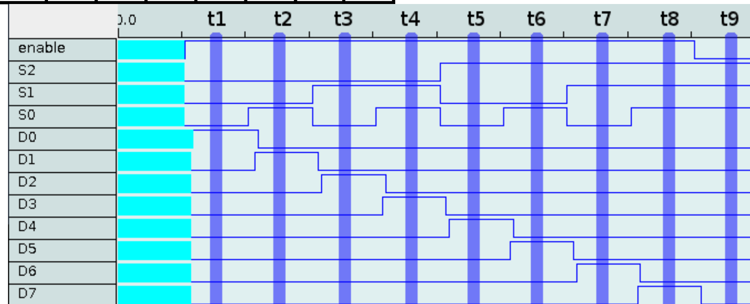
The incrementer/decrementer components are designed using the same daisy-chaining philosophy as the full adder component (see pg. 6), where the carry bits from one half-adder/subtracter ripple into the next. The only difference between these components and the full adder is that the incrementer and decrementer components have one less input. That is, instead of computing A + B + c_in, the incrementer component only computes A + c_in. Likewise, the decrementer component only computes A – c_in. However, the half-subtracter doesn't work with carry-out bits like the half-adder did; it works with borrow-bits, which indicate whether a one needs to be borrowed from the next bit in order to complete the desired operation. In short, the carry-out bit means "add one to the next bit" whereas the borrow-bit means "subtract one from the next bit."

In hindsight, these incrementer/decrementer components were unnecessary; the ALU will likely be used in a CPU in conjunction with a clock, which means that it will be treated as if each operation takes the same amount of time to complete. Since the ALU contains an 8-bit adder (used to compute A + not(B) + c_in and A + B + c_in), every computation will be assumed to be as slow as an 8-bit adder. This in turn negates any speed advantage obtained from using incrementer/decrementer components. However, if the ALU were used in a such a way that it's allowed to take three cycles to compute A + not(B) + c_in and A + B + c_in, and only one cycle for every other operation, then the use of the incrementer/decrementer components would make perfect sense. A multi-cycle CPU like that is far beyond the scope of this project, but it's good to know that the inclusion of such components can still make the ALU more efficient.

III – Verification:

	enable	S_2	S_1	S_0	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
t1	1	0	0	0	1	0	0	0	0	0	0	0
t2	1	0	0	1	0	1	0	0	0	0	0	0
t3	1	0	1	0	0	0	1	0	0	0	0	0
t4	1	0	1	1	0	0	0	1	0	0	0	0
t5	1	1	0	0	0	0	0	0	1	0	0	0
t6	1	1	0	1	0	0	0	0	0	1	0	0
t7	1	1	1	0	0	0	0	0	0	0	1	0
t8	1	1	1	1	0	0	0	0	0	0	0	1
t9	0	X	X	X	0	0	0	0	0	0	0	0

Since the 3-to-8 decoder's output defaults to 0 whenever enable is set to 0, inputs S_0, S_1, and S_2 should only matter when enable is set to 1. This means that the only possible input combinations that matter are the nine combinations shown in the truth table to the left. The waveforms for those combinations matches the table, so the circuit works as intended.



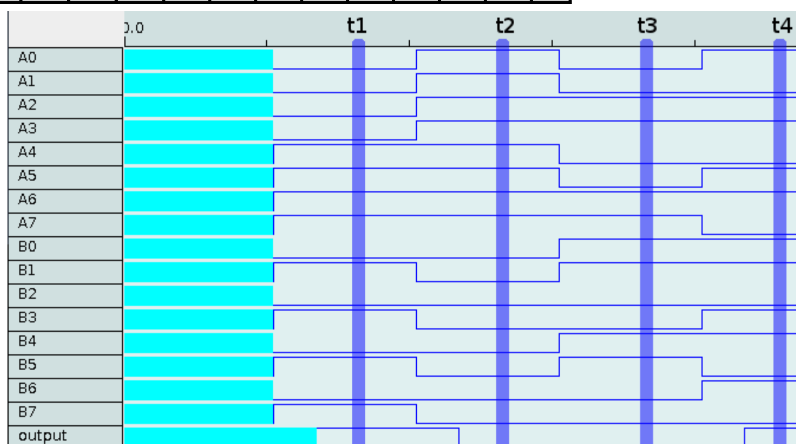
The 8-to-2 ANDOR component can be expressed as:

$$Y = (A_0B_0 + A_1B_1 + A_2B_2 + A_3B_3) + (A_4B_4 + A_5B_5 + A_6B_6 + A_7B_7)$$

which is the same as: $Y = A_0B_0 + A_1B_1 + A_2B_2 + A_3B_3 + A_4B_4 + A_5B_5 + A_6B_6 + A_7B_7$

This happens to be an exact description of the 8-to-2 ANDOR component's behavior, which means the component does exactly what it's supposed to. Just to make sure, here are a few test cases showing the component's desired outputs versus its actual waveform. The waveform matches the truth table, so the component works as it should.

	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	Y
t1	0	0	0	0	1	1	1	1	0	1	0	1	0	1	0	1	1
t2	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
t3	0	0	1	1	0	0	1	1	1	1	0	0	1	1	0	0	1
t4	1	0	1	1	0	1	1	0	1	1	0	1	1	0	1	0	1

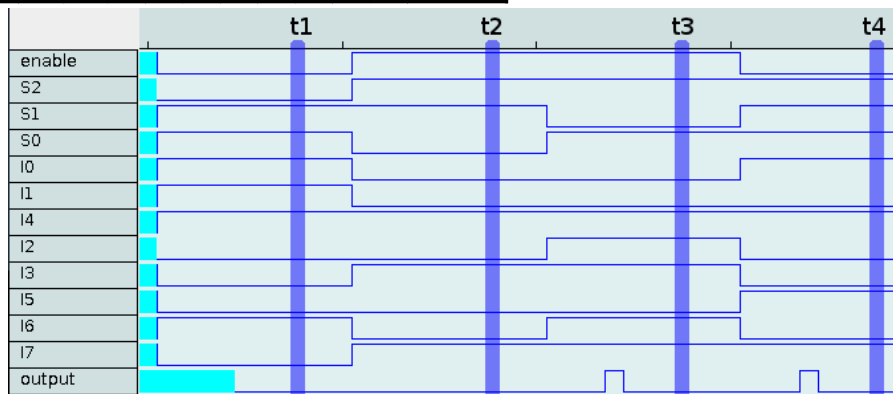


The outputs Dn of the 3-to-8 decoder component in the 8-to-1 multiplexer are fed into the inputs A_n of an 8-to-2 ANDOR, and that 8-to-2 ANDOR's inputs B_n are connected to inputs I_n. Because of this, the 8-to-1 multiplexer can be expressed as:

$$Y = D_0I_0 + D_1I_1 + D_2I_2 + D_3I_3 + D_4I_4 + D_5I_5 + D_6I_6 + D_7I_7$$

Due to the behavior of the 3-to-8 decoder, only one of the D outputs is on at a given time – except when enable is 0, in which case all of it's outputs are 0 – and that output is determined by inputs S_0, S_1, and S_2. Because of this, the expression simplifies to $Y = I_n$, where n is the binary number $S_2S_1S_0$, but if enable is set to 0 then Y defaults to 0. This is the desired behavior for an 8-to-1 multiplexer, which means that this component works properly. Just to make sure, though, here are a few test cases showing the component's desired outputs versus its actual waveform. The waveform matches the truth table, so the component works as it should.

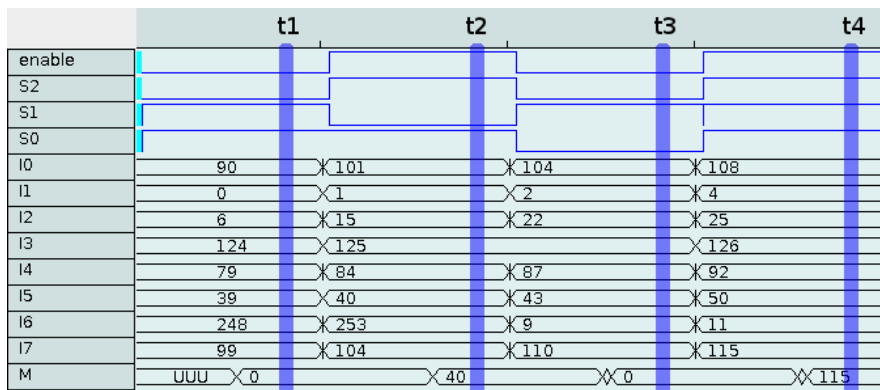
	enable	S_2	S_1	S_0	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	output
t1	0	0	1	1	1	1	1	0	0	0	1	0	0
t2	1	1	1	0	0	0	1	0	1	0	0	1	0
t3	1	1	0	1	0	0	1	1	1	0	1	1	0
t4	0	1	1	1	1	0	1	0	0	1	0	1	0



If the 3-to-8 decoder and 8-to-2 ANDOR components both work, and each 8-to-2 ANDOR components in the 8-bit 8-to-1 multiplexer is hooked up to the outputs of a 3-to-8 decoder as well as to the kth bits of inputs I_0, I_1, I_2, I_3, and I_4, then the 8-bit 8-to-1 multiplexer can be expressed logically as $M = I_n$, where n corresponds to which of the 3-to-8 decoder's outputs D_n is set to 1. Since the output D_n in the 3-to-8 decoder is determined by S_0, S_1, and S_2, where n corresponds to the binary number $S_2S_1S_0$, the 8-bit 8-to-1 multiplexer can also be expressed as $M = I_n$, where n corresponds to the binary number $S_2S_1S_0$. Since the outputs of the 3-to-8 decoder default to 0 if enable is set to 0, and since those outputs are used to determine which input to select, then output M of the 8-bit 8-to-1 multiplexer defaults to 00000000 if enable is set to 0; otherwise $M = I_n$, where n corresponds to the binary number $S_2S_1S_0$. Coincidentally, that's the exact specification for an 8-bit 8-to-1 multiplexer. It was proven earlier that the 3-to-8 decoder and 8-to-2 ANDOR components do indeed both work, which means that this design for an 8-bit 8-to-1 multiplexer also works. To err on the side of caution, here are is a truth table with a handful of input combination and their corresponding output, as well as a matching waveform. In case there was any doubt, yes, this component works.

	enable	S_2	S_1	S_0	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	M
t1	0	0	1	1	90	0	6	124	79	39	248	99	0
t2	1	1	0	1	101	1	15	125	84	40	253	104	40
t3	0	0	1	0	104	2	22	125	87	43	9	110	0
t4	1	1	1	1	108	4	25	126	92	50	11	115	115

(waveform on next page)

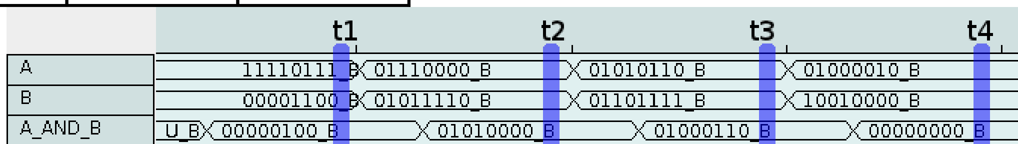


The 8-bit AND component can be expressed algebraically as:

$A_AND_B_0 = A_0B_0$, $A_AND_B_1 = A_1B_1$, $A_AND_B_2 = A_2B_2$, $A_AND_B_3 = A_3B_3$,
$A_AND_B_4 = A_4B_4$, $A_AND_B_5 = A_5B_5$, $A_AND_B_6 = A_6B_6$, $A_AND_B_7 = A_7B_7$

This can be simplified to $A_AND_B_n = A_nB_n$ $(0 \leq n \leq 7)$, where n is the nth bit of a given number. Simplified even further, this yields A_AND_B = AB, where A, B, and A_AND_B are eight-bit numbers. This is exactly what the circuit is supposed to do (produce the logical AND of two eight-bit numbers), which means it works properly. Just to err on the safe side, though, here are a few test cases showing the component's desired outputs versus its actual waveform. The waveform matches the truth table, so the component works properly.

	A	B	A_AND_B
t1	11110111	00001100	00000100
t2	01110000	01011110	01010000
t3	01010110	01101111	01000110
t4	01000010	10010000	00000000

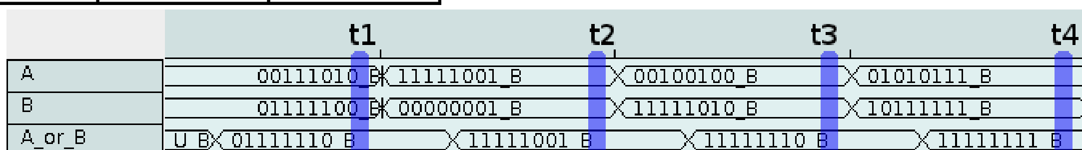


The 8-bit OR component can be expressed algebraically as:

$A_or_B_0 = A_0 + B_0$, $A_or_B_1 = A_1 + B_1$, $A_or_B_2 = A_2 + B_2$, $A_or_B_3 = A_3 + B_3$,
$A_or_B_4 = A_4 + B_4$, $A_or_B_5 = A_5 + B_5$, $A_or_B_6 = A_6 + B_6$, $A_or_B_7 = A_7 + B_7$

This can be simplified to $A_or_B_n = A_n + B_n$ $(0 \leq n \leq 7)$, where n is the nth bit of a given number. Simplified even further, this yields A_or_B = A + B, where A, B, and A_or_B are eight-bit numbers. This is exactly what the circuit is supposed to do (produce the logical OR of two eight-bit numbers), which means it works properly. Just to err on the safe side, though, here are a few test cases showing the component's desired outputs versus its actual waveform. The waveform matches the truth table, so the component works properly.

	A	B	A_or_B
t1	00111010	01111100	01111110
t2	11111001	00000001	11111001
t3	00100100	11111010	11111110
t4	01010111	10111111	11111111

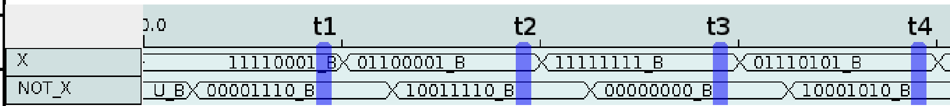


The 8-bit NOT component can be expressed algebraically as:

$$NOT_X_0 = \overline{X_0},\ \ NOT_X_1 = \overline{X_1},\ \ NOT_X_2 = \overline{X_2},\ \ NOT_X_3 = \overline{X_3},$$
$$NOT_X_4 = \overline{X_4},\ \ NOT_X_5 = \overline{X_5},\ \ NOT_X_6 = \overline{X_6},\ \ NOT_X_7 = \overline{X_7}$$

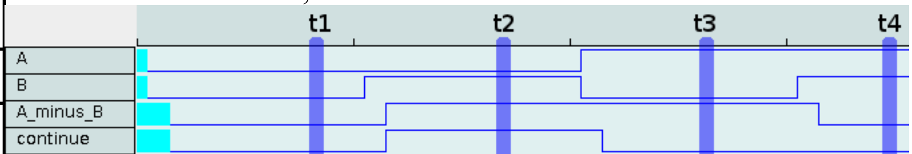
This can be simplified to $NOT_X_n = \overline{X_n}$ $(0 \leq n \leq 7)$, where n is the nth bit of a given number. Simplified even further, this yields $NOT_X = \overline{X}$, where X and NOT_X are eight-bit numbers. This is exactly what the circuit is supposed to do (produce the logical NOT of its input), which means it works properly. Just to err on the safe side, though, here are a few test cases showing the component's desired outputs versus its actual waveform. The waveform matches the truth table, so the component works properly.

	X	NOT_X
t1	11110001	00001110
t2	01100001	10011110
t3	11111111	00000000
t4	01110101	10001010



	A	B	A_minus_B	continue
t1	0	0	0	0
t2	0	1	1	1
t3	1	0	1	0
t4	1	1	0	0

The half-subtracter component has four possible input combinations, shown in the truth table to the left. The component's waveform matches the truth table, which means it works as it should.

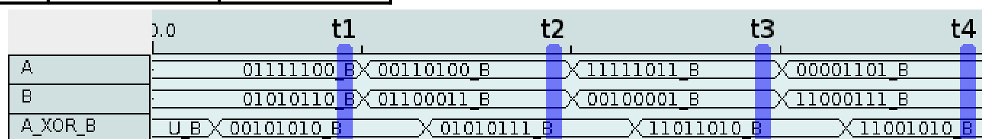


The 8-bit XOR component can be expressed algebraically as:

$$A_XOR_B_0 = A_0 \oplus B_0,\ \ A_XOR_B_1 = A_1 \oplus B_1,\ \ A_XOR_B_2 = A_2 \oplus B_2,\ \ A_XOR_B_3 = A_3 \oplus B_3,$$
$$A_XOR_B_4 = A_4 \oplus B_4,\ \ A_XOR_B_5 = A_5 \oplus B_5,\ \ A_XOR_B_6 = A_6 \oplus B_6,\ \ A_XOR_B_7 = A_7 \oplus B_7$$

This can be simplified to $A_XOR_B_n = A_n \oplus B_n$ $(0 \leq n \leq 7)$, where n is the nth bit of a given number. Simplified even further, this yields $A_XOR_B = A \oplus B$, where A, B, and A_AND_B are eight-bit numbers. This is exactly what the circuit is supposed to do (produce the logical XOR of two eight-bit numbers), which means it works properly. Just to err on the safe side, though, here are a few test cases showing the component's desired outputs versus its actual waveform. The waveform matches the truth table, so the component works properly.

	A	B	A_XOR_B
t1	01111100	00101010	00101010
t2	00110100	01100011	01010111
t3	11111011	00100001	11011010
t4	00001101	11000111	11001010

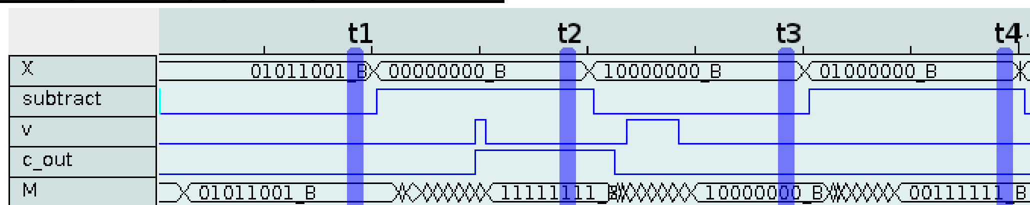


The decrementer's topmost half-subtracter has two inputs bits, one of which is connected to the decrementer's own input bit subtract, and the other of which is connected to bit 0 of input X. If the half-subtracter works correctly, its output should be X_0 minus subtract. That half-subtracter's continue output bit is then fed into the next half-subtracter's B input bit, and its other input, A, is set to bit 1 of X. This means the second half-subtracter computes X_1 minus the borrow-bit (continue) bit obtained from computing X_0 - subtract. The rest of the half-subtracters follow this pattern as well, basing their own B input off of the computations of the previous half-subtracters. Because of all this daisy-chaining, the entire decrementer's output M is determined by the expression $M_7 = X_7$ – the borrow-bit from

computing ($M_6 = X_6$ – the borrow-bit from computing ($M_5 = X_5$ – the borrow-bit from computing ($M_4 = X_4$ – the borrow-bit from computing ($M_3 = X_3$ – the borrow-bit from computing ($M_2 = X_2$ – the borrow-bit from computing ($M_1 = X_1$ – the borrow-bit from computing ($M_0 = X_0$ - subtract))))))), and the entire circuit's c_out is the borrow-bit outputted from the last half-subtracter (which is obtained from computing M_7). This expression can be further simplified to Y = X - subtract, where Y is the 9-bit binary number c_out$S_7S_6S_5S_4S_3S_2S_1S_0$ (c_out is the sign bit, since the borrow-bit means "subtract one from the next bit"). Output v is determined by XOR-ing the last two borrow-bits in the operation. Since that is exactly how overflow is determined when subtracting 2's complement encoded numbers, that means v determines if there was signed 2's complement overflow when subtracting subtract from X.

	X	subtract	v	c_out	M
t1	01011001	0	0	0	01011001
t2	00000000	1	0	1	11111111
t3	10000000	0	0	0	100000000
t4	01000000	1	0	0	00111111

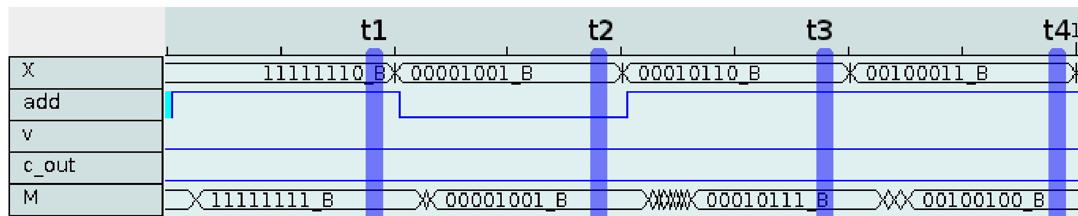
This proves that the decrementer computes X – subtract and accounts for signed 2's complement overflow. Since that's what the decrementer is supposed to do, that means the component works as intended. Just to make certain, though, here are a few test cases showing the component's desired outputs versus its actual waveforms.



The incrementer's topmost half-adder has two inputs bits, one of which is connected to the incrementer's own input bit add, and the other of which is connected to bit 0 of input X. If the half-adder works correctly, its output should be X_0 plus add. That half-adder's C_{out} output bit is then fed into the next half-adder's A input bit, and its other input, B, is set to bit 1 of X. This means the second half-adder computes X_1 plus the carry-out bit (C_{out}) obtained from computing X_0 + add. The rest of the half-adders follow this pattern as well, basing their own A input off of the computations of the previous half-adders. Because of all this daisy-chaining, the entire incrementer's output M is determined by the expression $M_7 = X_7$ + the carry-out from computing ($M_6 = X_6$ + the carry-out from computing ($M_5 = X_5$ + the carry-out from computing ($M_4 = X_4$ + the carry-out from computing ($M_3 = X_3$ + the carry-out from computing ($M_2 = X_2$ + the carry-out from computing ($M_1 = X_1$ + the carry-out from computing ($M_0 = X_0$ + add))))))), and the entire circuit's c_out is the carry-out outputted from the last half-adder (which is obtained from computing M_7). This expression can be further simplified to Y = X + add, where Y is the 9-bit binary number c_out$S_7S_6S_5S_4S_3S_2S_1S_0$. Output v is determined by XOR-ing the last two carry-bits in the operation. Since that is exactly how overflow is determined when subtracting 2's complement encoded numbers, that means v determines if there was signed 2's complement overflow when adding X and add. This proves that the incrementer computes X + add and accounts for signed 2's complement overflow. Since that's what the incrementer component is supposed to do, that means the component works as intended. Just to make certain, though, here are a few test cases showing the component's desired outputs versus its actual waveforms.

	X	add	v	c_out	M
t1	11111110	1	0	0	11111111
t2	00001001	0	0	0	00001001
t3	00010110	1	0	0	00010111
t4	001000011	1	0	0	00100100

(waveform on next page)



The ALU's output m7 is supposed to indicate output M's sign bit if M is a signed number. If M were signed, that would mean that its most significant bit (M_7) would be its sign bit. If the ALU produced overflow, however, the sign bit would be the ALU's carry-out bit (when an operation results in overflow, the sign bit overflows into the carry-out bit). As such, m7 can be expressed as m7 = $(\overline{\text{overflow}})M_7$ + (overflow)(carry-out bit) (M_7 refers to bit 7 of output M). Overflow and carry-out correspond to outputs v and c_out, so this translates to m7 = $\overline{v}M_7$ + v(c_out). This is exactly how m7 is determined in the ALU circuit, which means that – assuming outputs M, v, and c_out work correctly – output m7 works as it should.

The ALU contains an 8-bit 8-to-1 multiplexer whose output is connected to the ALU's M output, and whose select bits S_2, S_1, and S_0 are connected to bits 2, 1, and 0, respectively, of alu_sel. This means that whatever is connected to the 8-bit 8-to-1 multiplexer's input I_{alu_sel} will be outputted as M. additional input, enable, is connected to the 8-bit 8-to-1 multiplexer's enable input, so output M is equal to the input I_{alu_sel} of the 8-bit 8-to-1 multiplexer, except when enable is set to 0, in which case the 8-bit 8-to-1 multiplexer's output defaults to 00000000, which would set M to 00000000. The ALU also contains two 8-to-1 multiplexers – one which outputs to v, and another which outputs to c_out – and these have their select select and enable inputs connected to the ALU's alu_sel and enable inputs the same way they were connected for the 8-bit 8-to-1 multiplexer. This in turn means that c_out and v are each determined by their 8-to-1 multiplexer's corresponding input I_{alu_sel}, and that both c_out and v default to 0 if enable is set to 0. Since m7 = $\overline{v}M_7$ + v(c_out), and since M, c_out, and v default to 0 when enable is 0, then that means m7 defaults to 0 whenever enable is set to 0. Effectively, this means that all of the ALU's outputs default to 0 whenever enable is 0, which means that enable works properly

Inputs I_0, I_1, I_4, and I_5 on both of the single-bit 8-to-1 multiplexers are connected to GND, which means that v and c_out default to 0 when alu_sel is 0, 1, 4 or 5. This makes sense, since those values correspond to the ALU's logical operations, and logical operations aren't supposed to produce any overflow, carry-out, or sign values. Since m7 = $\overline{v}M_7$ + v(c_out), and v defaults to 0 in those cases, m7 also defaults to 0 in those cases. This means that only the value of output M matters when alu_sel is set to 0, 1, 4, or 5.

Input D_0 of the 8-bit 8-to-1 multiplexer is connected to the output of an 8-bit OR component that takes ALU's inputs A and B as its inputs. As such, D_0 corresponds to the logical OR of inputs A and B. When enable is set to 1 and alu_sel is set to 0, the value on D_0 gets passed on to output M, meaning that M would be equal to A OR B under those circumstances. Since this is the exact operation that is supposed to happen when alu_sel is set to 0, this proves that the ALU works for all cases where alu_sel is 0.

Input D_1 of the 8-bit 8-to-1 multiplexer is connected to the output of an 8-bit NOT component that takes ALU's inputs A as its input. As such, D_1 corresponds to the logical NOT of input A. When enable is set to 1 and alu_sel is set to 1, the value on D_1 gets passed on to output M, meaning that M would be equal to not(A) under those circumstances. Since this is the exact operation that is supposed to happen when alu_sel is set to 1, this proves that the ALU works for all cases where alu_sel is 1.

The input D_2 of each of the three multiplexers is connected to the outputs of an 8-bit adder component that takes as its inputs A, c_in, and the output of an 8-bit NOT component that takes B as its input. The 8-bit NOT component's output corresponds to not(B). Since this is fed into the 8-bit adder, that means that the 8-bit adder calculates A + not(B) + c_in. The 8-bit adder outputs the results of this operation in the form of 8-bit outputs S (sum), v (overflow), and C_{out} (carry-out). S is fed into D_2 of the 8-bit 8-to-1 multiplexer, v is fed into D_2 of output v's 8-to-1 multiplexer, and C_{out} is fed into D_2 of c_out's 8-to-1 multiplexer. When enable is set to 1 and alu_sel is set to 2, the value on D_2 for each multiplexer get passed on to outputs M, v, and c_out meaning that under those circumstances M would be equal to the sum A + not(B) + c_in, v would be equal to the overflow produced when computing A + not(B) + c_in, and c_out would be equal to the carry-out for A + not(B) + c_in. Since this is the exact operation that is supposed to happen when alu_sel is set to 2, this proves that outputs M, v, and c_out work properly when alu_sel is set to 2. This in turn proves that m7 works as it should whenever alu_sel is set to 2, which means the entire ALU works for all cases where alu_sel is 2.

The input D_3 of each of the three multiplexers is connected to the outputs of an 8-bit adder component that takes as its inputs A, B, and c_in. That means that the 8-bit adder calculates A + B + c_in. The 8-bit adder outputs the results of this operation in the form of 8-bit outputs S (sum), v (overflow), and C_{out} (carry-out). S is fed into D_3 of the 8-bit 8-to-1 multiplexer, v is fed into D_3 of output v's 8-to-1 multiplexer, and C_{out} is fed into D_3 of c_out's 8-to-1 multiplexer. When enable is set to 1 and alu_sel is set to 3, the value on D_3 for each multiplexer get passed on to outputs M, v, and c_out meaning that under those circumstances M would be equal to the sum A + B + c_in, v would be equal to the overflow produced when computing A + B + c_in, and c_out would be equal to the carry-out for A + B + c_in. Since this is the exact operation that is supposed to happen when alu_sel is set to 3, this proves that outputs M, v, and c_out work properly when alu_sel is set to 3. This in turn proves that m7 works as it should whenever alu_sel is set to 3, which means the entire ALU works for all cases where alu_sel is 3.

Input D_4 of the 8-bit 8-to-1 multiplexer is connected to the output of an 8-bit XOR component that takes ALU's inputs A and B as its inputs. As such, D_4 corresponds to the logical XOR of inputs A and B. When enable is set to 1 and alu_sel is set to 4, the value on D_4 gets passed on to output M, meaning that M would be equal to A XOR B under those circumstances. Since this is the exact operation that is supposed to happen when alu_sel is set to 4, this proves that the ALU works for all cases where alu_sel is 4.

Input D_5 of the 8-bit 8-to-1 multiplexer is connected to the output of an 8-bit AND component that takes ALU's inputs A and B as its inputs. As such, D_5 corresponds to the logical AND of inputs A and B. When enable is set to 1 and alu_sel is set to 5, the value on D_5 gets passed on to output M, meaning that M would be equal to A AND B under those circumstances. Since this is the exact operation that is supposed to happen when alu_sel is set to 0, this proves that the ALU works for all cases where alu_sel is 5.

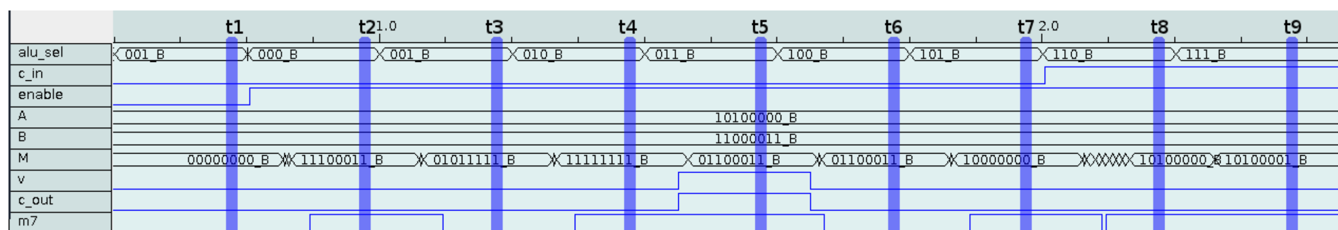
The input D_6 of each of the three multiplexers is connected to the outputs of a decrementer component that takes as its inputs A, and the output of an inverter whose input is c_in. The inverter outputs not(c_in), so that means the decrementer calculates A - not(c_in). The decrementer outputs the results of this operation in the form of 8-bit outputs M (sum), v (overflow), and c_out (carry-out). M is fed into D_6 of the 8-bit 8-to-1 multiplexer, v is fed into D_6 of output v's 8-to-1 multiplexer, and c_out is fed into D_6 of c_out's 8-to-1 multiplexer. When enable is set to 1 and alu_sel is set to 6, the value on D_6 for each multiplexer get passed on to outputs M, v, and c_out meaning that under those circumstances M would be equal to the sum A – not(c_in), v would be equal to the overflow produced when computing A –

not(c_in), and c_out would be equal to the carry-out for A – not(c_in). The desired operation for when alu_sel is set to 6 is A – 1 + c_in. If c_in was set to 1 the operation would equate to A – 1 + 0 = A – 1. If c_in was set to 0, the operation would equate to A – 1 + 1 = A. When c is set to 0, A – not(c_in) equates to A – not(0) = A – 1. When c is set to 1, A – not(c_in) equates to A – not(1) = A. This shows that A – 1 + c_in and A – c_in are equivalent, which means that outputs M, v, and c_out work properly when alu_sel is set to 6. This in turn proves that m7 works as it should whenever alu_sel is set to 6, which means the entire ALU works for all cases where alu_sel is 6.

The input D_7 of each of the three multiplexers is connected to the outputs of an incrementer component that takes A and c_in as its inputs. That means that the incrementer calculates A + c_in. The incrementer outputs the results of this operation in the form of 8-bit outputs M (sum), v (overflow), and c_out (carry-out). M is fed into D_7 of the 8-bit 8-to-1 multiplexer, v is fed into D_7 of output v's 8-to-1 multiplexer, and c_out is fed into D_7 of c_out's 8-to-1 multiplexer. When enable is set to 1 and alu_sel is set to 7, the value on D_7 for each multiplexer get passed on to outputs M, v, and c_out meaning that under those circumstances M would be equal to the sum A + c_in, v would be equal to the overflow produced when computing A + c_in, and c_out would be equal to the carry-out for A + c_in. Since this is the exact operation that is supposed to happen when alu_sel is set to 3, this proves that outputs M, v, and c_out work properly when alu_sel is set to 3. This in turn proves that m7 works as it should whenever alu_sel is set to 7, which means the entire ALU works for all cases where alu_sel is 7.

The ALU performs the correct operations on A and B for all possible cases of alu_sel and enable, therefore the entire ALU circuit works correctly. Just to make certain, though, here are a few test cases showing the circuit's desired outputs versus its actual waveforms.

	alu_sel	c_in	enable	A	B	M	v	c_out	m7
t1	1	0	0	1010000	11000011	00000000	0	0	0
t2	0	0	1	1010000	11000011	11100011	0	0	1
t3	1	0	1	1010000	11000011	01011111	0	0	0
t4	2	0	1	1010000	11000011	11111111	0	0	1
t5	3	0	1	1010000	11000011	01100011	1	1	1
t6	4	0	1	1010000	11000011	01100011	0	0	0
t7	5	0	1	1010000	11000011	10000000	0	0	1
t8	6	1	1	1010000	11000011	10100000	0	0	1
t9	7	1	1	1010000	11000011	10100001	0	0	1



The waveform matches the truth table, so the circuit works properly.