Exercise 5.1

(a) The input signals to this system are: # dimes and # of nickels in the reservoir, N_D and N_N, respectively, and coin deposited, C. The outputs are: no change, E, # dimes and # of nickels to return, R_D and R_N, respectively.

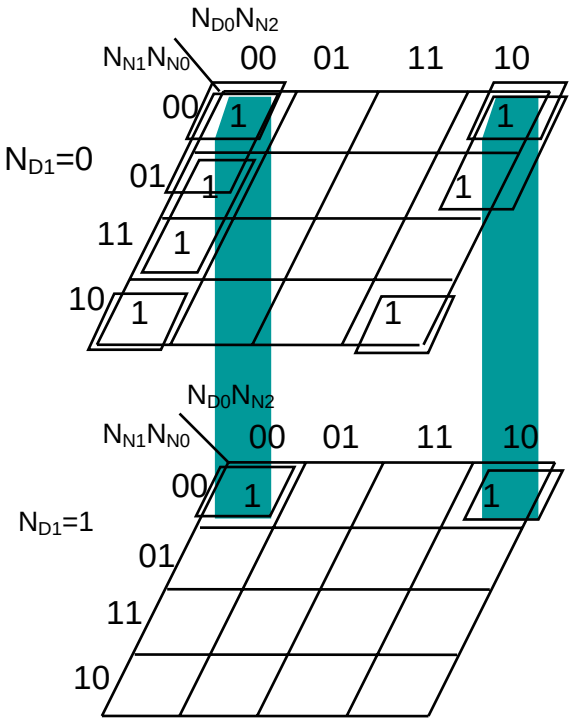
(b) N_D and N_N are the number of dimes and nickels left in the machine. C tells us that a quarter has been deposited so we need to make change. E signals to the machine that it should light up a "No Change Available" sign. R_D and R_N tell the machine how many dimes and nickels to return.

(c) Minimized gate-level implementation:

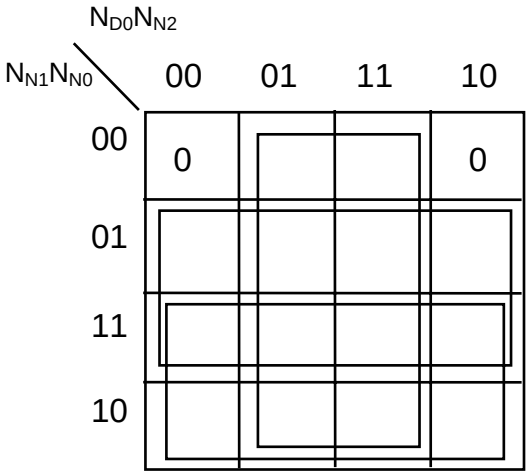
N_{D1}	N_{D0}	N_{N2}	N_{N1}	N_{N0}	E	R_{D1}	R_{D0}	R_{N2}	R_{N1}	R_{N0}
0	0	0	0	0	1	0	0	0	0	0
		0	0	1	1	0	0	0	0	0
		0	1	0	1	0	0	0	0	0
		0	1	1	1	0	0	0	0	0
0	0	1	0	0	1	0	0	0	0	0
		1	0	1	0	0	0	1	0	1
		1	1	0	0	0	0	1	0	1
		1	1	1	0	0	0	1	0	1
0	1	0	0	0	1	0	0	0	0	0
		0	0	1	1	0	0	0	0	0
		0	1	0	1	0	0	0	0	0
		0	1	1	0	0	1	0	1	1
0	1	1	0	0	0	0	1	0	1	1
		1	0	1	0	0	1	0	1	1
		1	1	0	0	0	1	0	1	1
		1	1	1	0	0	1	0	1	1
1	0	0	0	0	1	0	0	0	0	1
		0	0	1	0	1	0	0	0	1
		0	1	0	0	1	0	0	0	1
		0	1	1	0	1	0	0	0	1
1	0	1	0	0	0	1	0	0	0	1
		1	0	1	0	1	0	0	0	1
		1	1	0	0	1	0	0	0	1
		1	1	1	0	1	0	0	0	1
1	1	0	0	0	1	1	0	0	0	1
		0	0	1	0	1	0	0	0	1
		0	1	0	0	1	0	0	0	1
		0	1	1	0	1	0	0	0	1
1	1	1	0	0	0	1	0	0	0	1
		1	0	1	0	1	0	0	0	1
		1	1	0	0	1	0	0	0	1
		1	1	1	0	1	0	0	0	1

The input C is not included in the truth table because we never return change unless C is 1. However, we do still need to compute E.

If we look at the truth table carefully, we can see that not all of the outputs will require 5-variable K-maps. R_{D1} and R_{D0} can be implemented with 4-variable K-maps, which means that we will have to AND N_{D1} to the output those functions. Similarly, R_{N2} only requires a 3-variable K-map. Also, notice that R_{D0} and R_{N1} are the same.



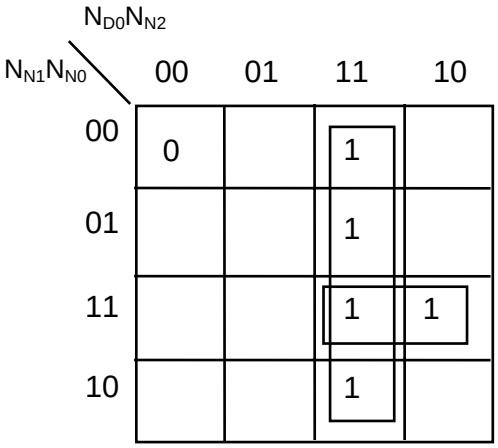
K-map for E



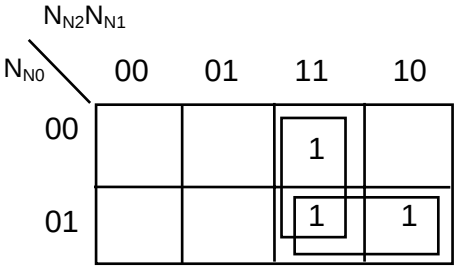
K-map for R_{D1}

$E = N_{D1}'N_{N2}'N_{N0}' + N_{N2}'N_{N1}'N_{N0}' + N_{D1}'N_{N2}'N_{N1}' + N_{D1}'N_{D0}'N_{N2}' + N_{D1}'N_{D0}'N_{N1}'N_{N0}'$

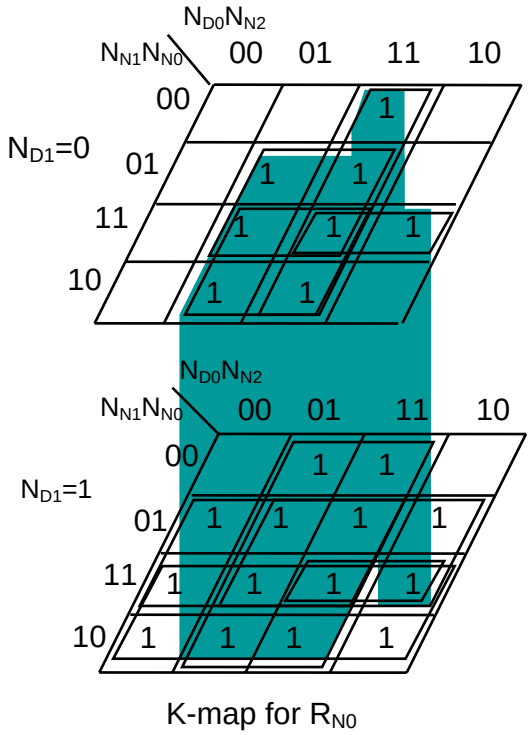
$R_{D1} = N_{D1}(N_{N2} + N_{N1} + N_{N0})$



K-map for R_{D0} & R_{N1}



K-map for R_{N2}



K-map for R_{N0}

$R_{D0} = R_{N1} = (N_{D0}N_{N2} + N_{D0}N_{N1}N_{N0})N_{D1}'$

$R_{N2} = (N_{D0}N_{N2} + N_{D0}N_{N1})N_{D1}'N_{D0}'$

$R_{N0} = N_{D0}N_{N2} + N_{N2}N_{N0} + N_{N2}N_{N1} + N_{D0}N_{N1}N_{N0} + (N_{N2} + N_{N1} + N_{N0})N_{D1}$

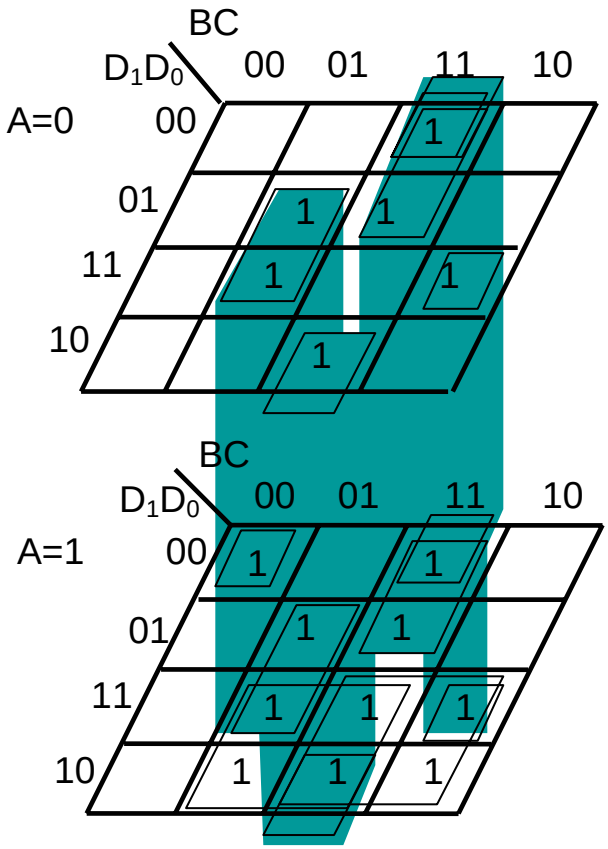
Exercise 5.2

This logic unit is a simple ALU which uses the control bits, A, B, and C, to determine the operation to be performed on D_1 and D_0.

(a) Truth table for Z:

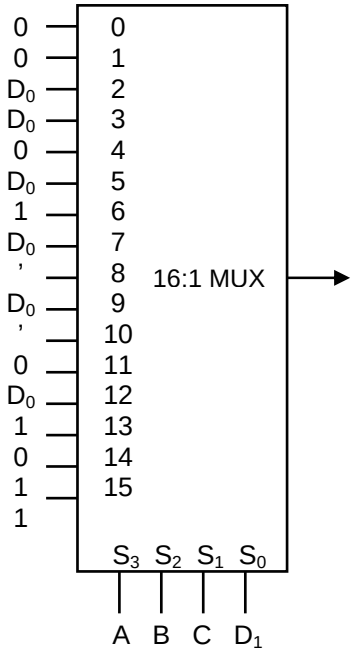
A	B	C	D_1	D_0	Z
0	0	0	0	0	0
			0	1	0
			1	0	0
			1	1	0
0	0	1	0	0	0
			0	1	1
			1	0	0
			1	1	1
0	1	0	0	0	0
			0	1	0
			1	0	0
			1	1	1
0	1	1	0	0	1
			0	1	1
			1	0	1
			1	1	0
1	0	0	0	0	1
			0	1	0
			1	0	0
			1	1	0
1	0	1	0	0	0
			0	1	1
			1	0	1
			1	1	1
1	1	0	0	0	0
			0	1	0
			1	0	1
			1	1	1
1	1	1	0	0	1
			0	1	1
			1	0	1
			1	1	1

K-map for Z:

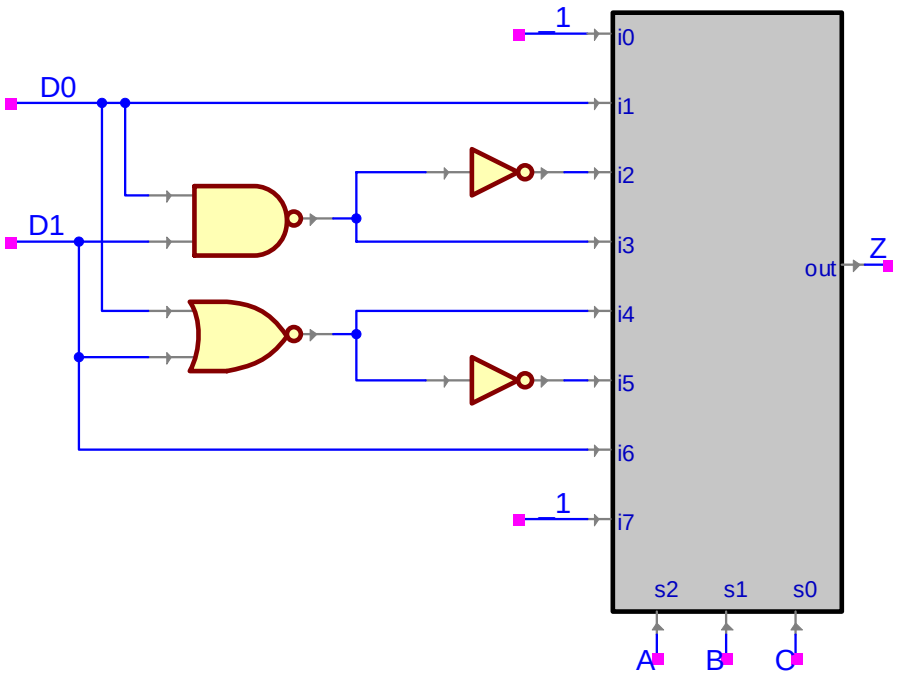


$Z(A,B,C,D_1,D_0) = BCD_1' + B'CD_0 + BCD_0' + BC'D_1D_0 + ABD_1 + ACD_1 + AB'C'D_1'D_0'$

(b) 16:1 Mux for Z:



(c) 8:1 MUX for Z:

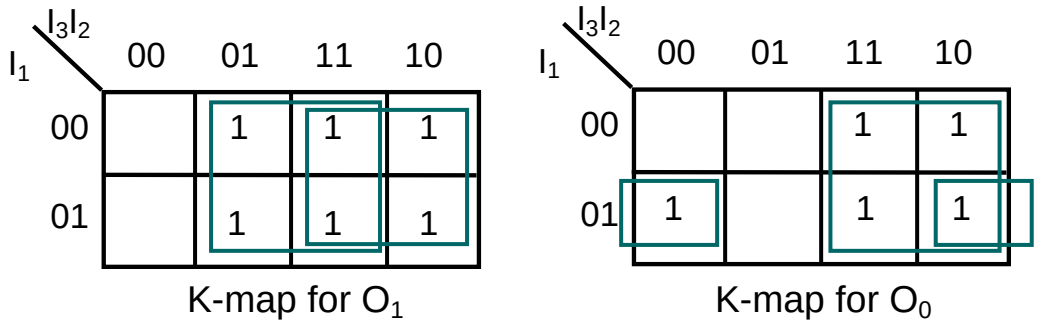


Exercise 5.3

(a) Truth table for O_1 and O_0:

I_3	I_2	I_1	O_1	O_0
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

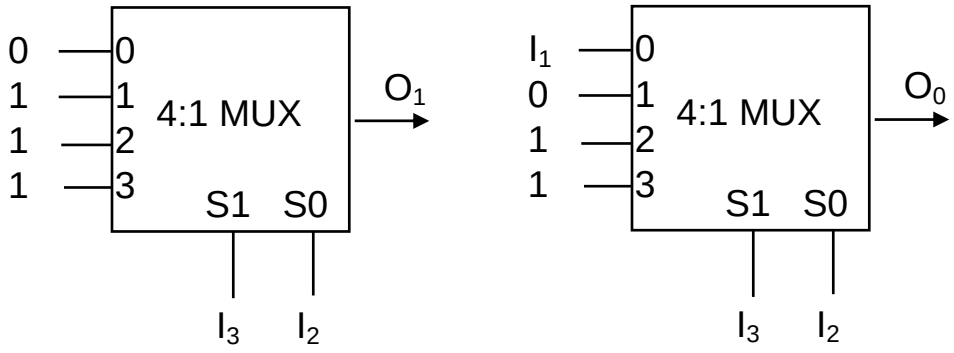
(b) Gate-level implementation:



$O_1 = I_3 + I_2$

$O_0 = I_3 + I_2'I_1$

(c) 4:1 multiplexor implementation:



Exercise 5.4

(a) The day offset block takes the month as input and returns a 9-bit value which corresponds to the number of days in the year up to that month.

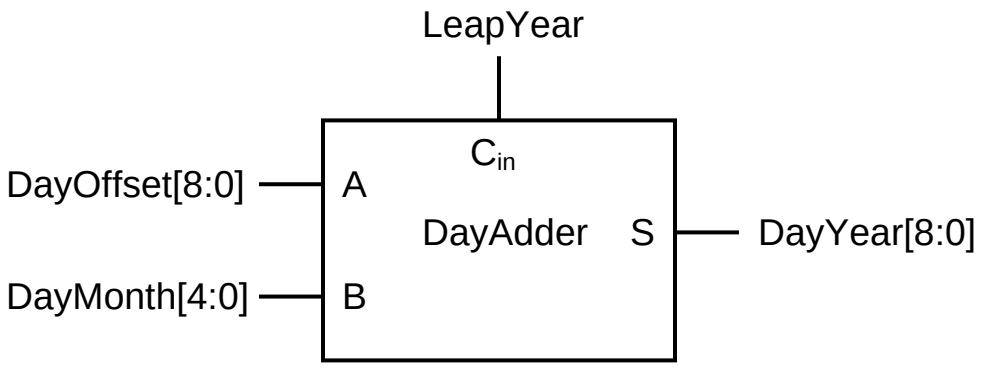
Block Diagram for DayOffset

Verilog module:

```
module DayOffset(input wire [3:0] M, output reg [8:0] D);

   always@(*)
         begin
               case(M)
                     4'b0001: D = 0;
                     4'b0010: D = 31;
                     4'b0011: D = 59;
                     4'b0100: D = 90;
                     4'b0101: D = 120;
                     4'b0110: D = 151;
                     4'b0111: D = 181;
                     4'b1000: D = 212;
                     4'b1001: D = 243;
                     4'b1010: D = 273;
                     4'b1011: D = 304;
                     4'b1100: D = 334;
                     default: D = 0;
               endcase
         end
endmodule
```

(b) The adder takes the output from the DayOffset module, the day of the month, and a carry-in, which is tied to a leap year signal. Tying the leap year to the carry in allows us to add an extra day if it a leap year.



Block diagram for DayAdder

(c) This implementation of a day-of-the-year system would require 9 4-input LUTs to implement.

Exercise 5.5

(a) Truth table for the half-subtractor D = A - B:

A	B	B_I	D	B_L
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

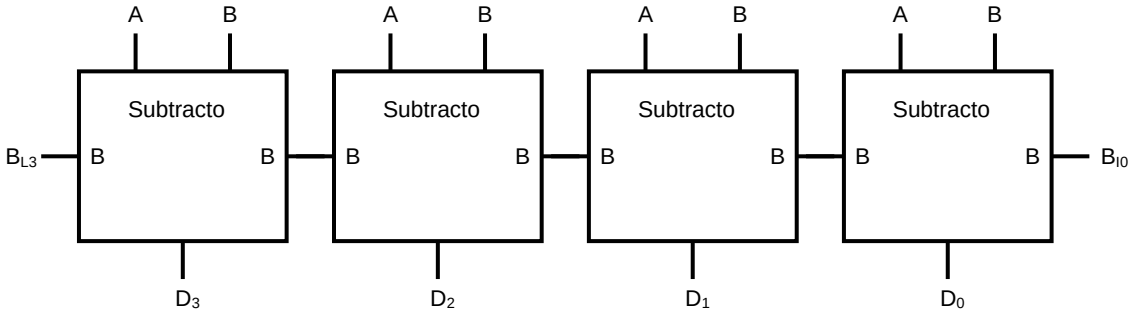
$D = A'B'B_I + A'BB_I' + AB'B_I' + ABB_I$

Or simply:

D = A xor B xor B_I

$B_L = A'B + A'B_I + BB_I$

(b) Block diagram for a 4-bit subtractor:



(c) This multi-bit subtractor works with 2's compliment numbers just as the full-adder does.

(d) Underflow can be detected by checking if the MSB tries to borrow from the left: B_{L3} is 1.

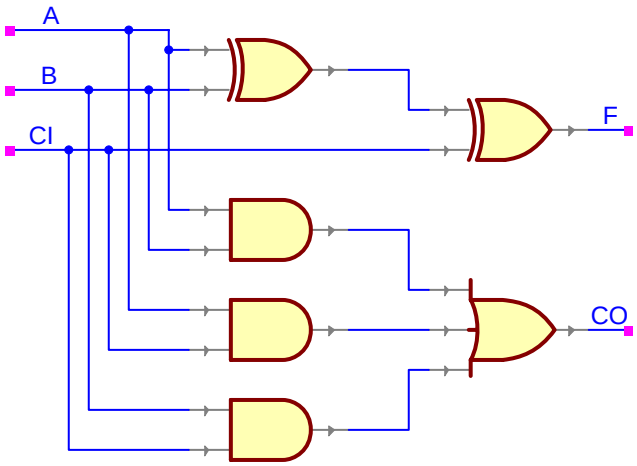
Exercise 5.6

The basic module we will use is the half-adder. As long as the operands are 2's compliment numbers, we can easily use the adder to subtract two numbers. We do this by inverting the bits of one operand and setting the carry-in of the first half-adder to one.

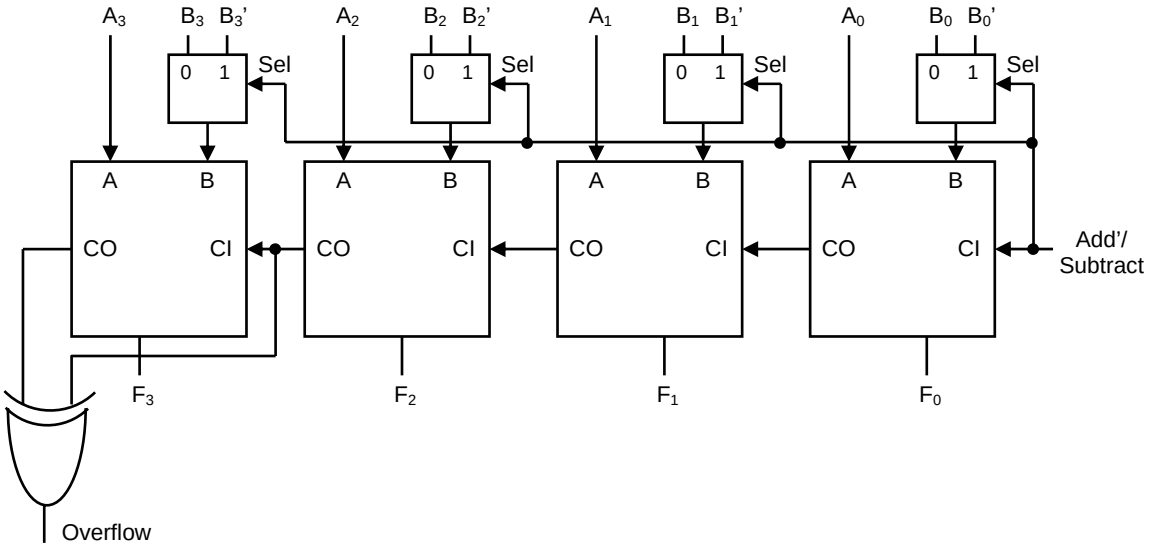
The carry-in, CI, and borrow-in, BI, can be shared, as well as the carry-out, CO, and borrow from left, BL, signals. Here are the functions for each block:

F = (A xor B xor CI)

CO = (AB + ACI + BCI)



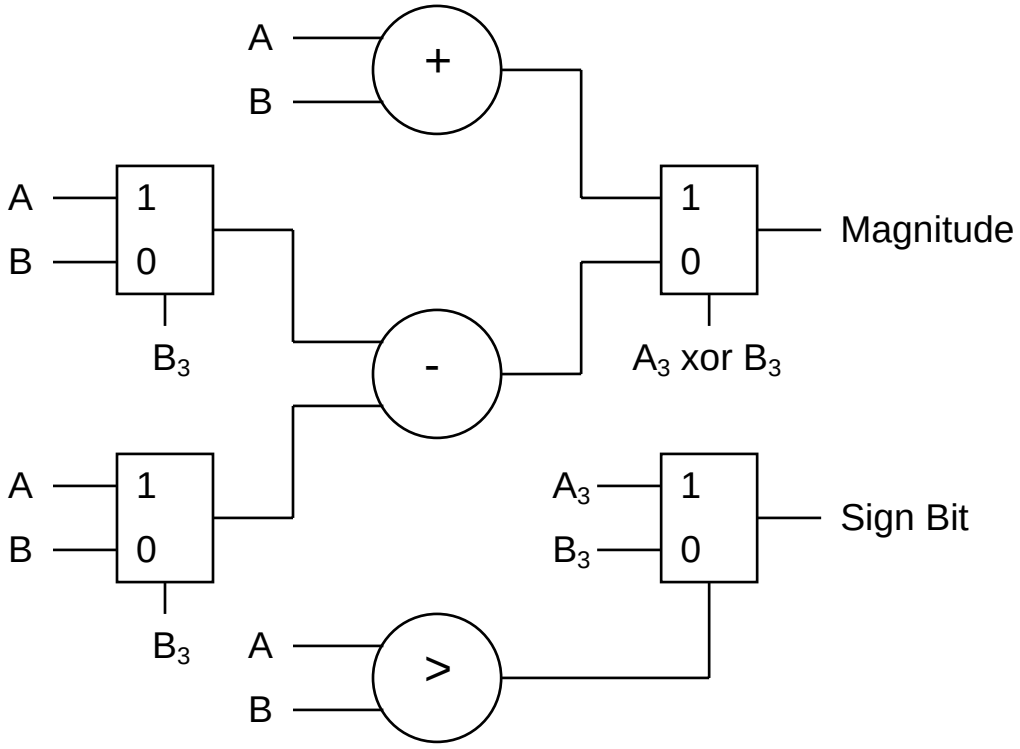
Circuit diagram for each block



Combinational adder/subtractor block diagram

Exercise 5.7

A and B are the 3-bit magnitudes of the operands. The sign bits are compared and if equal, the result is the sum of the magnitudes with the sign of the operands. If B is negative, it is subtracted from A. If B is positive, A must be negative (since we are assuming that they are unequal), so A is subtracted from B. The sign bit is chosen from the result of the comparison, but the selection is relevant only if the signs differ. We use once mux for both cases to save logic. Overflow is passed through the magnitude mux from the adder and subtractor blocks.



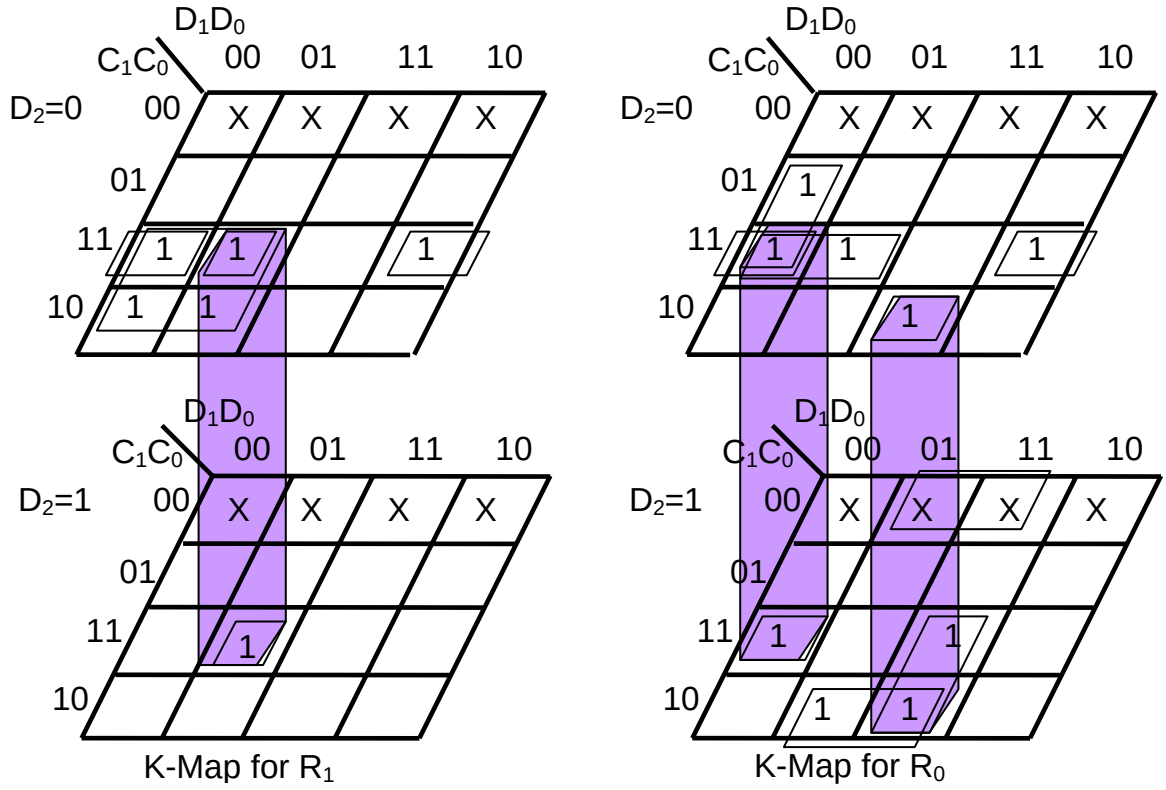
Exercise 5.8

(a) Here is the truth table for R_0 and R_1:

Note that since division by zero will never be requested, there are eight don't-cares for both R_1 and R_0.

D_2	D_1	D_0	C_1	C_0	R_1	R_0
0	0	0	0	0	X	X
0	0	0	0	1	0	1
0	0	0	1	0	1	0
0	0	0	1	1	1	1
0	0	1	0	0	X	X
0	0	1	0	1	0	0
0	0	1	1	0	1	0
0	0	1	1	1	1	1
0	1	0	0	0	X	X
0	1	0	0	1	0	0
0	1	0	1	0	0	0
0	1	0	1	1	1	1
0	1	1	0	0	X	X
0	1	1	0	1	0	0
0	1	1	1	0	0	1
0	1	1	1	1	0	0
1	0	0	0	0	X	X
1	0	0	0	1	0	0
1	0	0	1	0	0	0
1	0	0	1	1	0	1
1	0	1	0	0	X	X
1	0	1	0	1	0	0
1	0	1	1	0	0	1
1	0	1	1	1	1	0
1	1	0	0	0	X	X
1	1	0	0	1	0	0
1	1	0	1	0	0	0
1	1	0	1	1	0	0
1	1	1	0	0	X	X
1	1	1	0	1	0	0
1	1	1	1	0	0	1
1	1	1	1	1	0	1

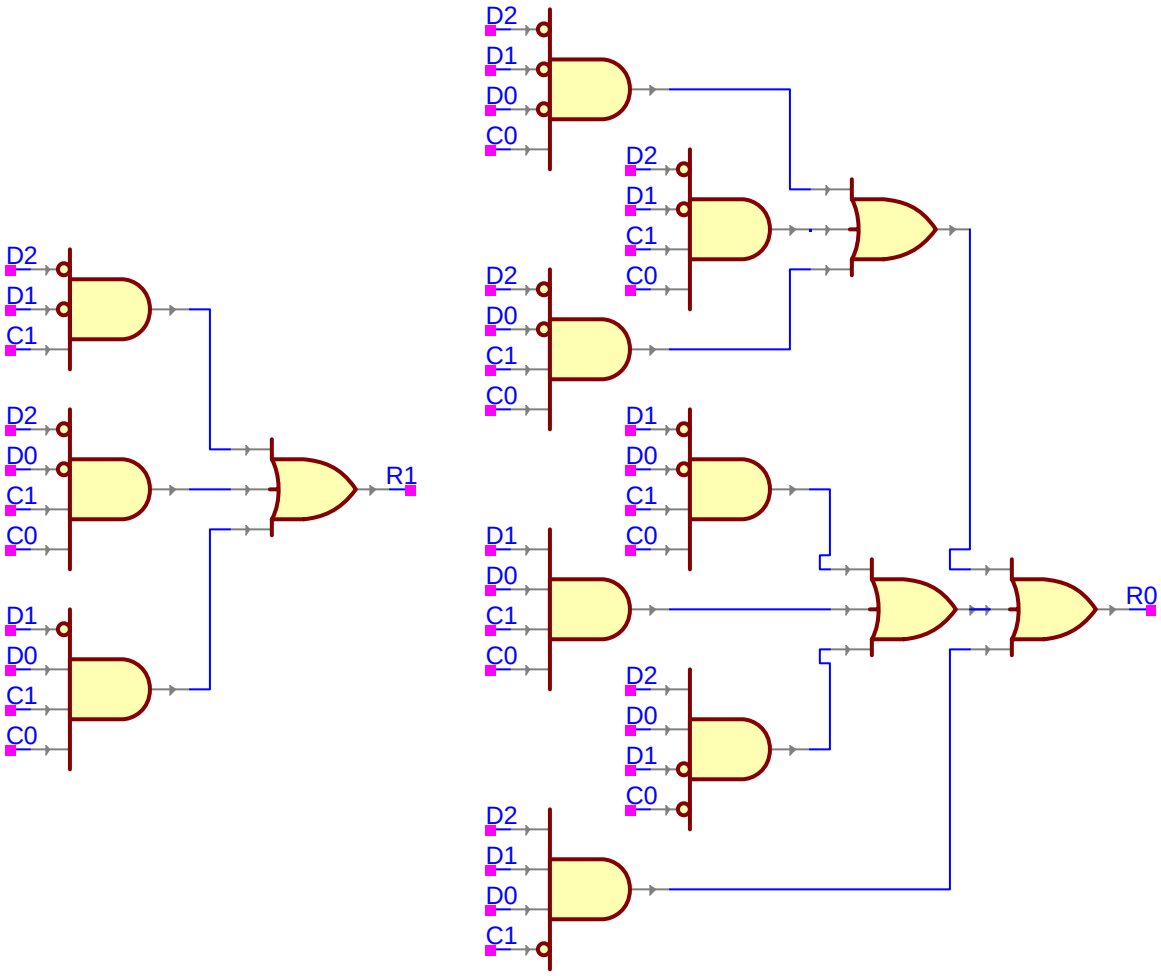
(b) K-Maps for R_1 and R_0:



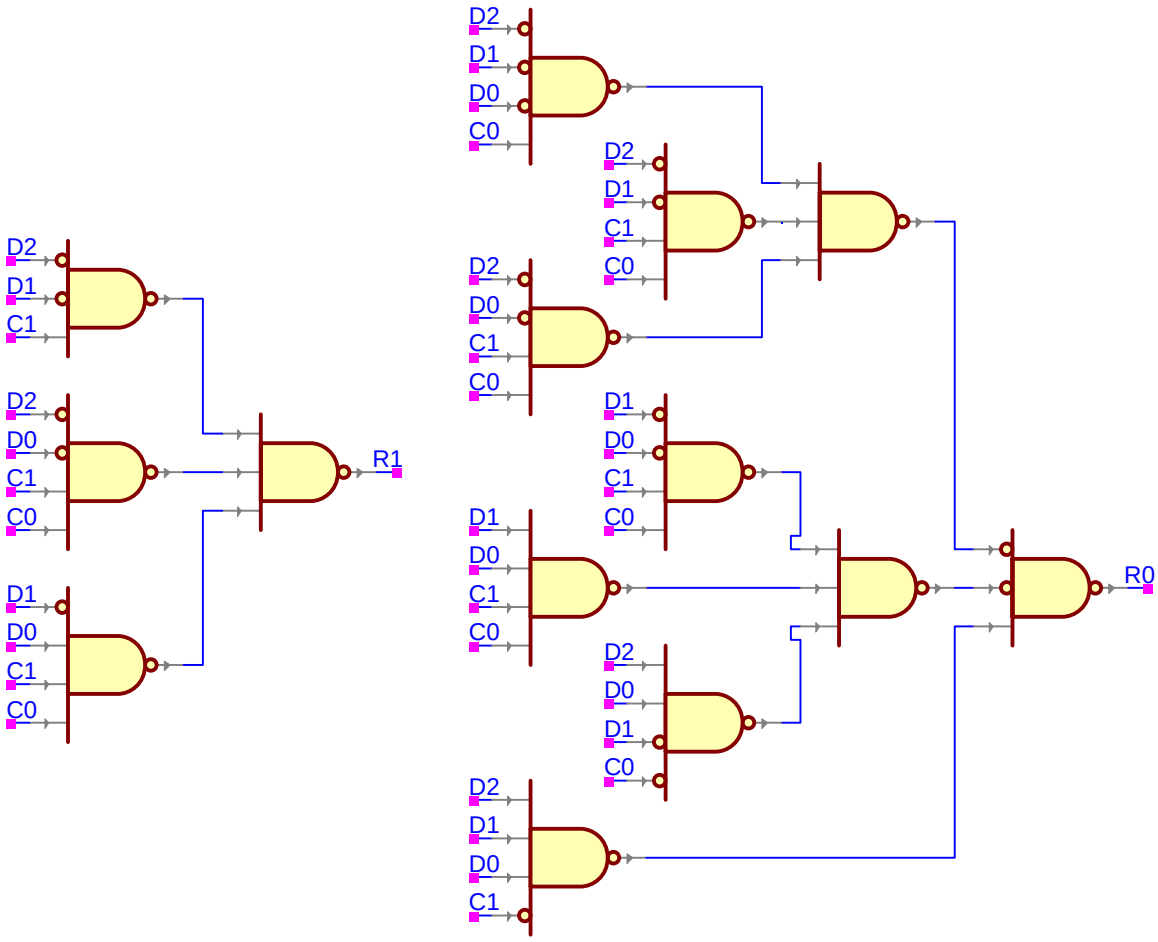
$R_1 = D_2'D_1'C_1 + D_2'D_0'C_1C_0 + D_1'D_0C_1C_0$

$R_0 = D_2'D_1'D_0'C_0 + D_2'D_1'C_1C_0 + D_2'D_0'C_1C_0 + D_1'D_0'C_1C_0 + D_1D_0C_1C_0' + D_2D_1D_0C_1 + D_2D_0C_0'$

(c) Circuit Schematics:



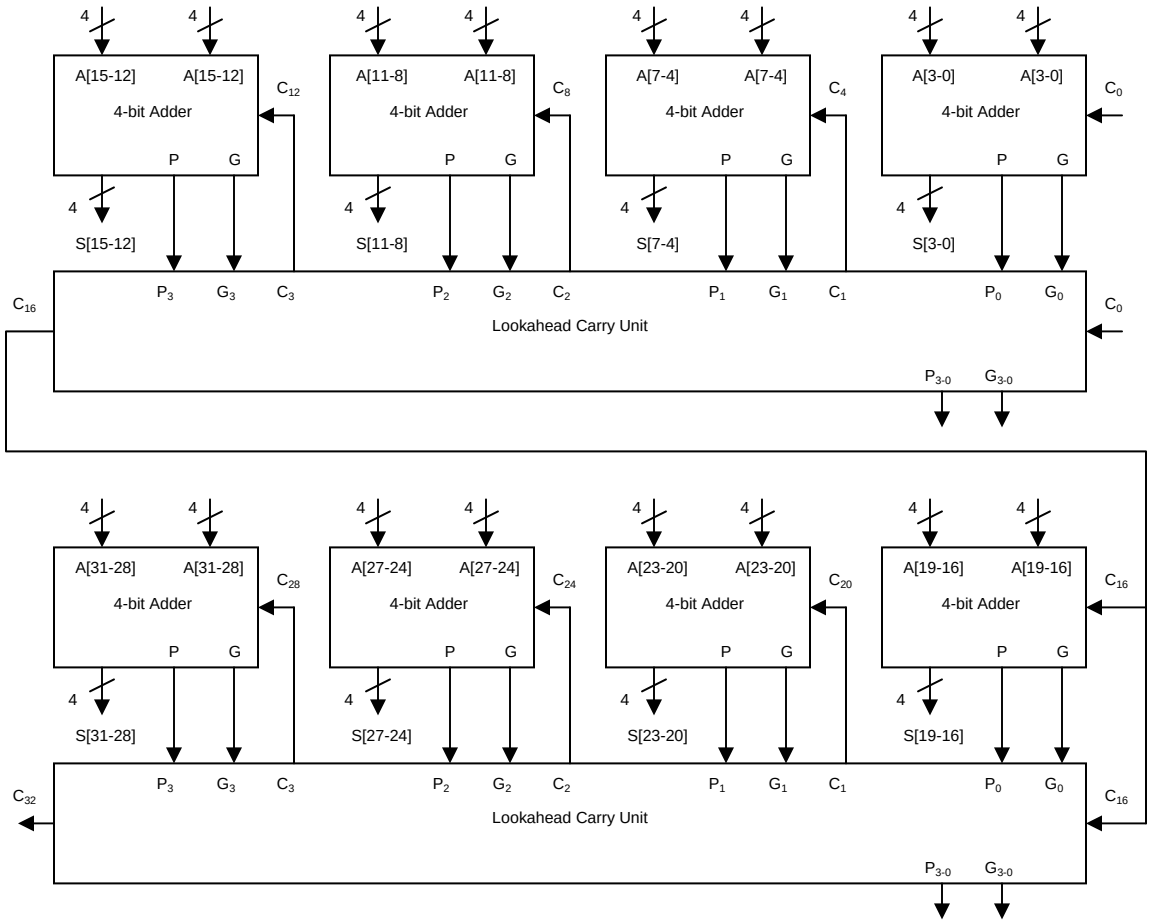
Initial Circuits



NAND-gate implementation

Exercise 5.9

(a) Block diagram for adder with carry lookahead:



32-bit Carry Lookahead Adder

The same idea is used for the 64-bit adder. We would use 4 of the 16-bit carry lookahead adders connected in series.

(b) The 32-bit implementation will have a critical delay of 13 time units. The 64-bit implementation will have a critical delay of 23 time units.

Exercise 5.10

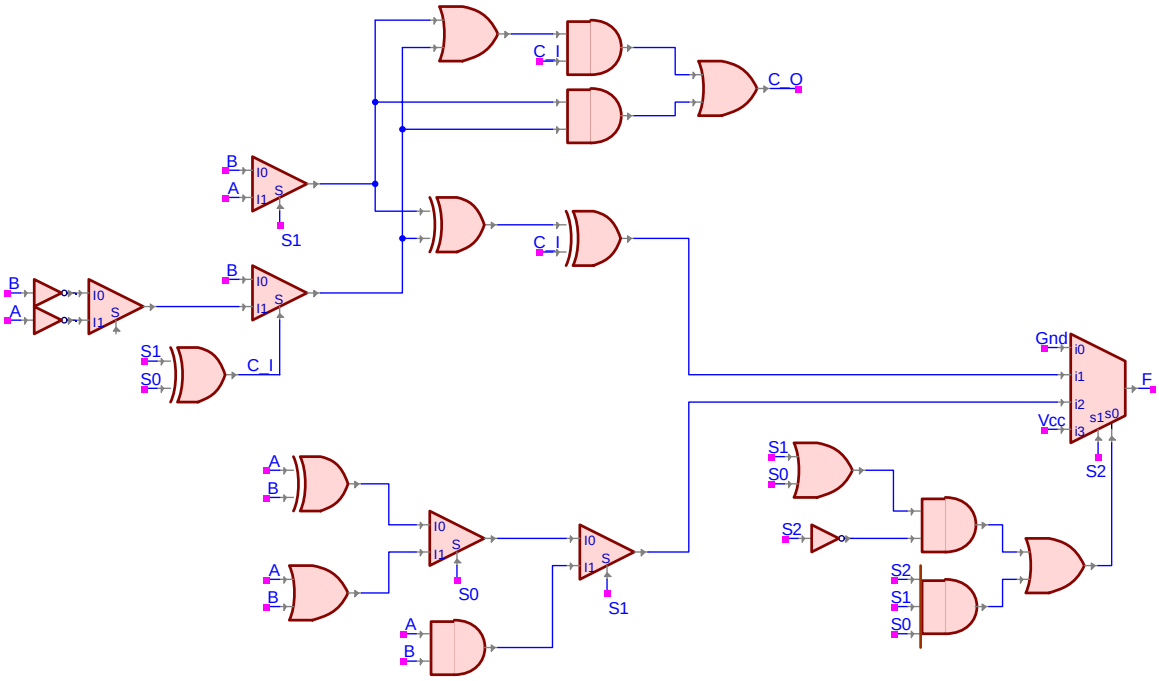
The critical path in carry-select adders is the highest order sum bit plus the delay through the multiplexor. Assuming that the 8-bit carry lookahead adder is implemented using two 4-bit adders and carry logic, the critical delay for the 8-bit adder will be 7. The multiplexor will add 2 to the overall delay, which makes the critical path for a 16-bit carry-select adder 9.

This is much better than a 16-bit ripple-carry adder which would have a critical delay of 18. The 16-bit carry-lookahead adder is slightly faster, with a critical delay of 8. However, as the adder gets bigger, the carry-select adder will be much faster than any other adder. A 32-bit carry-select adder, implemented in the same way as the 16-bit adder, has a critical delay of just 1 more.

Exercise 5.11

The main functionality of the ALU can be broken into four parts: 0, arithmetic, logic, and 1. Each of these is pieces shares functionality. We can use the control signals and small multiplexors to get the correct functionality. The four components come together with a 4:1 multiplexor, which is controlled with some logic that selects the correct input for a given S2-S0 value.

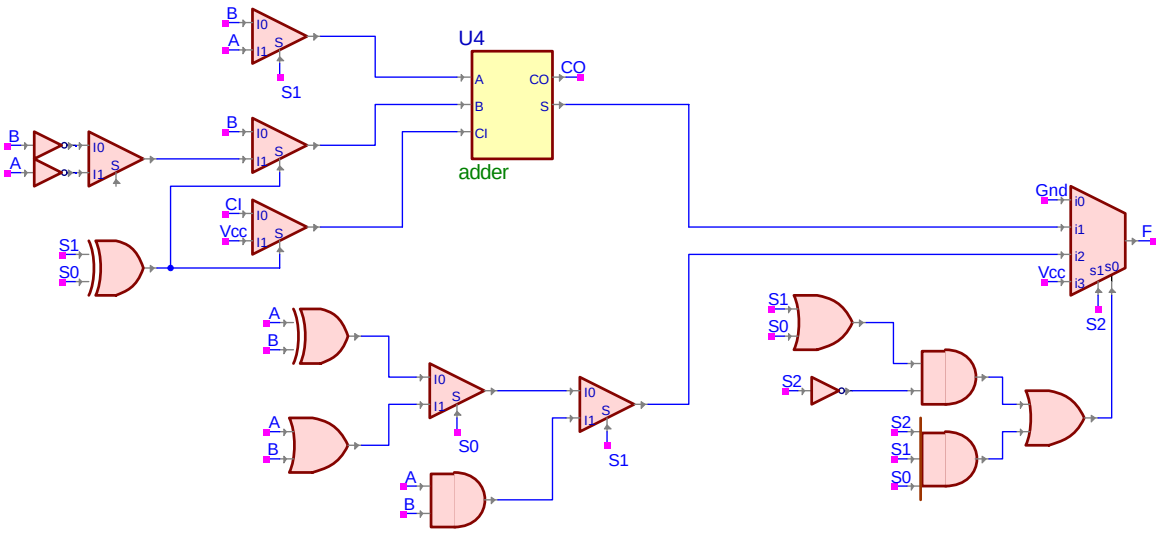
The separate components are grouped together in the following circuit schematic:



Exercise 5.12

The main functionality of the ALU can be broken into four parts: 0, arithmetic, logic, and 1. Each of these is pieces shares functionality. We can use the control signals and small multiplexors to get the correct functionality. The four components come together with a 4:1 multiplexor, which is controlled with some logic that selects the correct input for a given S2-S0 value.

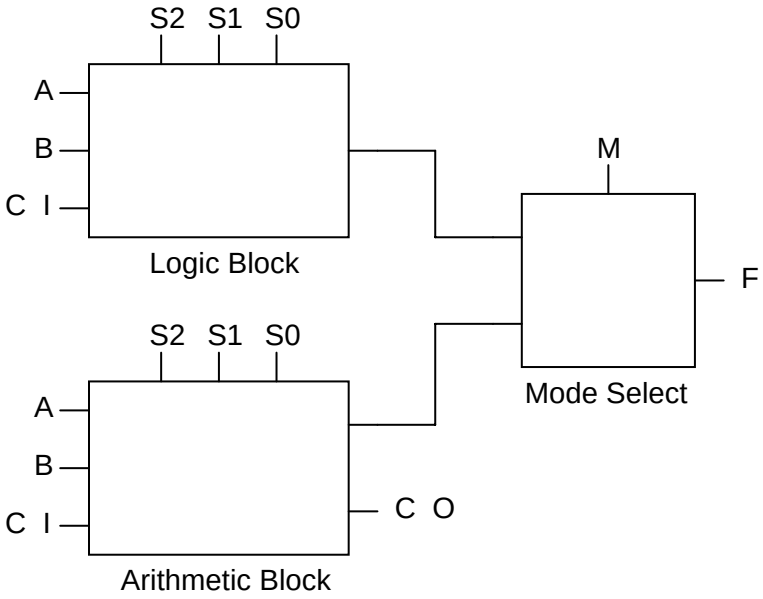
The separate components are grouped together in the following circuit schematic:



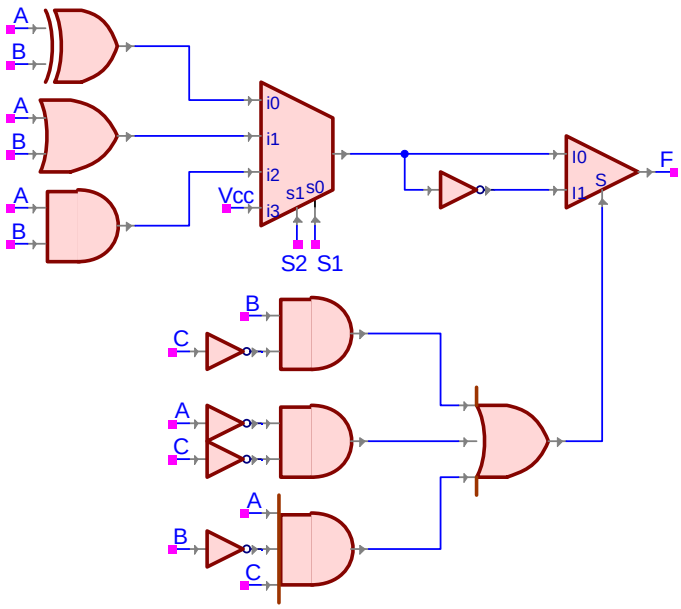
Exercise 5.13

Since this is a large design, we will implement it in a hierarchical-fashion using block diagrams at the top level and describe each of the components separately.

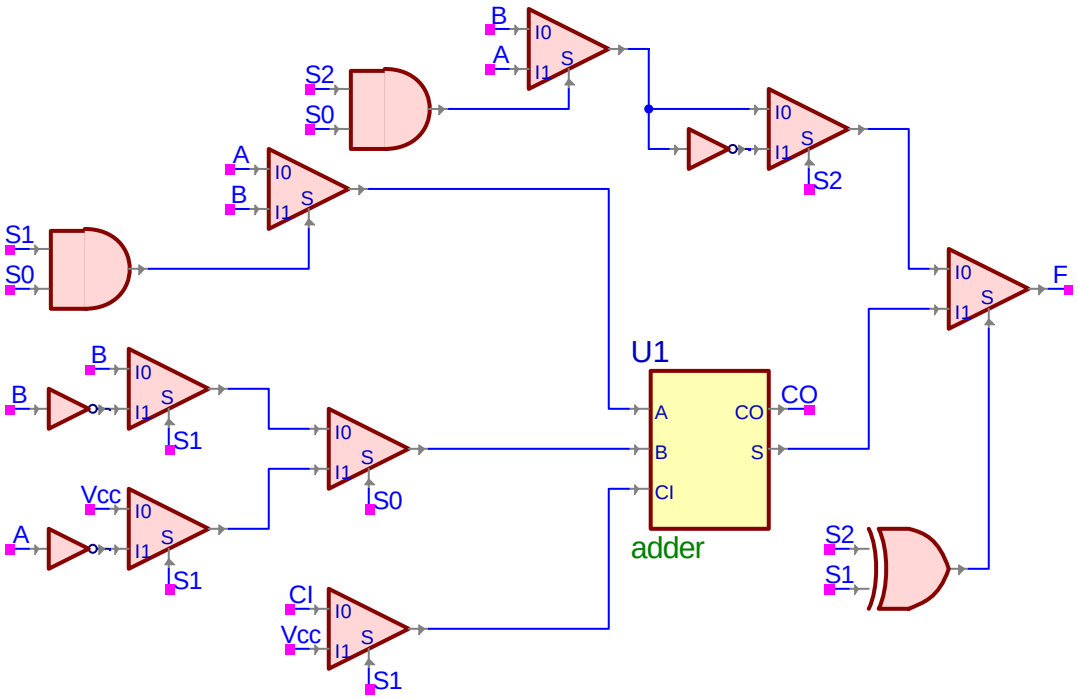
The top-level block diagram looks as follows:



The logic block is implemented with a 4:1 and 2:1 multiplexor along with some combinational logic. It works by first doing the logical operation and selecting the correct one with the 4:1 mux, then it negates the result if it needs to. One thing to note is this block doesn't do anything with CO, which is okay because we don't care what its value is for logic operations. The resulting circuit looks as follows:



This logic block uses a bunch of 2:1 multiplexors to select the correct inputs for the half-adder. It then uses more 2:1 multiplexors to output the correct value. The resulting circuit schematic looks as follows:



Lastly, the mode select block is just a 2:1 multiplexor that selects the arithmetic block if M is 1 and the logic block if M is 0.

Exercise 5.14

Since we implemented the ALU in Exercise 5.12 with an adder module, changing the design to use a carry-lookahead adder is simple. All of the wires for A and B would change into 4-bit busses. The carry-in input to the adder would remain unchanged. The half-adder would be replaced by a 4-bit carry-lookahead adder, which would take the two 4-bit A and B operands and the CI. It would output a 4-bit result and a carry-out. Everything else in the design would remain unchanged except for the fact that A and B are now 4-bit values.

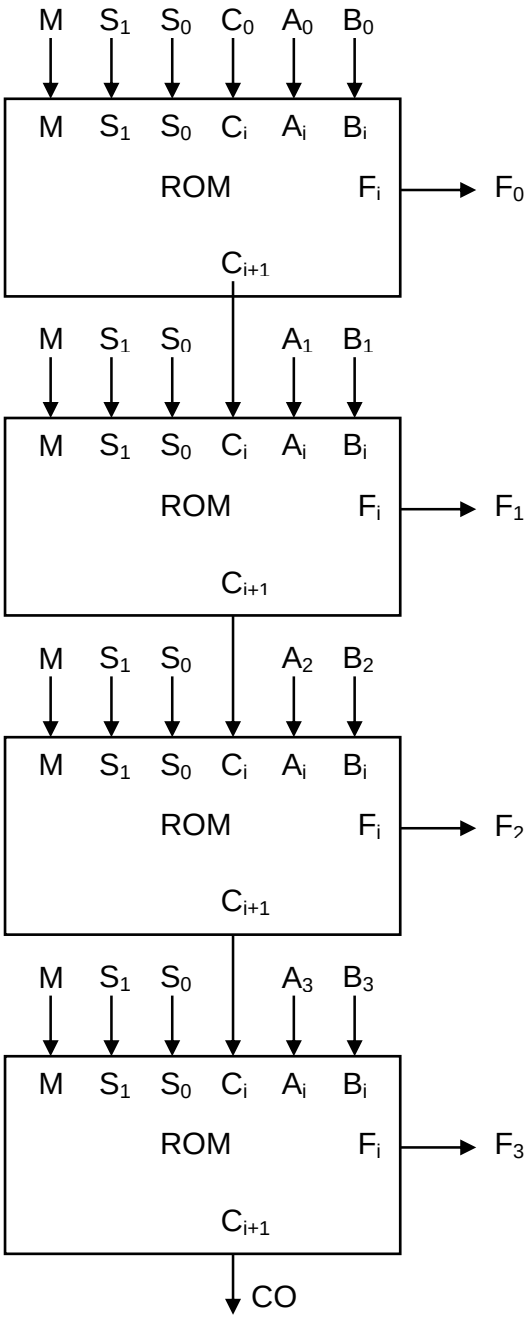
Exercise 5.15

Since we implemented the ALU in Exercise 5.13 with an adder module, changing the design to use a carry-lookahead adder is simple. All of the wires for A and B would change into 4-bit busses. The carry-in input to the adder would remain unchanged. The half-adder would be replaced by a 4-bit carry-lookahead adder, which would take the two 4-bit A and B operands and the CI. It would output a 4-bit result and a carry-out. Everything else in the design would remain unchanged except for the fact that A and B are now 4-bit values.

Exercise 5.16

One convenient property of a ROM is if you have a truth table, it is simple to implement a ROM. Each of the inputs to the truth table become the addresses inputs to the ROM and the outputs from the truth table are the outputs from the ROM. So, M, S_1, S_0, C_i, A_i, and B_i, become are the address inputs from most to least significant. The two outputs from the ROM are: F_i, and C_{i+1}.

We can implement a 4-bit ALU by cascading four of these ROMs as follows:



Exercise 5.17

The function for F_i is:

$F_i = S_iB_i$ xor (MC_i xor (S_0 xor A_i))

It is much easier to expand complicated expression if you express them as in terms of their operations with single variables and then substitute the complex expressions back into the equation at the end. So F_i becomes:

$F_i = W$ xor X, where $W = S_iB_i$, and $X = MC_i$ xor (S_0 xor A_i)

We can further simplify X as follows:

$X = Y$ xor Z, where $Y = MC_i$ and $Z = S_0$ xor A_i

Now we can begin our expansion:

$F_i = WX' + W'X$

$X = YZ' + Y'Z$

$Z = S_0A_i' + S_0'A_i$

Now we need to find expressions for W', X', Y', and Z' before we begin to substitute in:

$W' = S_i' + B_i'$

$X' = YZ + Y'Z'$

$Y' = M' + C_i'$

$Z' = S_0A_i + S_0'A_i'$

Now we can substitute everything back in and collect the terms. The resulting expression is:

$F_i = MS_1S_0C_iA_i'B_i + MS_1S_0'C_iA_iB_i + M'S_1S_0A_iB_i + M'S_1S_0'A_i'B_i + S_1S_0C_i'A_iB_i + S_1S_0'C_i'A_i'B_i + MS_1'S_0C_iA_i + MS_1'S_0'C_iA_i' + M'S_1'S_0A_i' + M'S_1'S_0'A_i + S_1'S_0C_i'A_i' + S_1'S_0'C_i'A_i + MS_0C_iA_iB_i' + MS_0'C_iA_i'B_i' + M'S_0A_i'B_i' + M'S_0'A_iB_i' + S_0C_i'A_i'B_i' + S_0'C_i'A_iB_i'$

This is correct because for every row in the truth table that F_i is true, we have a term in the expression that corresponds to that entry.

Applying the same method for C_{i+1}, we get the following expression:

$C_{i+1} = MS_1S_0C_iA_iB_i + MS_1S_0'C_iA_i'B_i + M'S_1S_0A_iB_i + M'S_1S_0'A_i'B_i + S_1S_0C_i'A_iB_i + S_1S_0'C_i'A_i'B_i + MS_0C_iA_i' + MS_0'C_iA_i$

Again, all of the terms in the expression match up to the rows in the truth table which are true for C_{i+1}.

Exercise 5.18

BCD addition is similar to binary addition except that if the result is greater than nine, we need to add six to it to keep it in BCD form.

(a) 0001 + 0100 = 0101

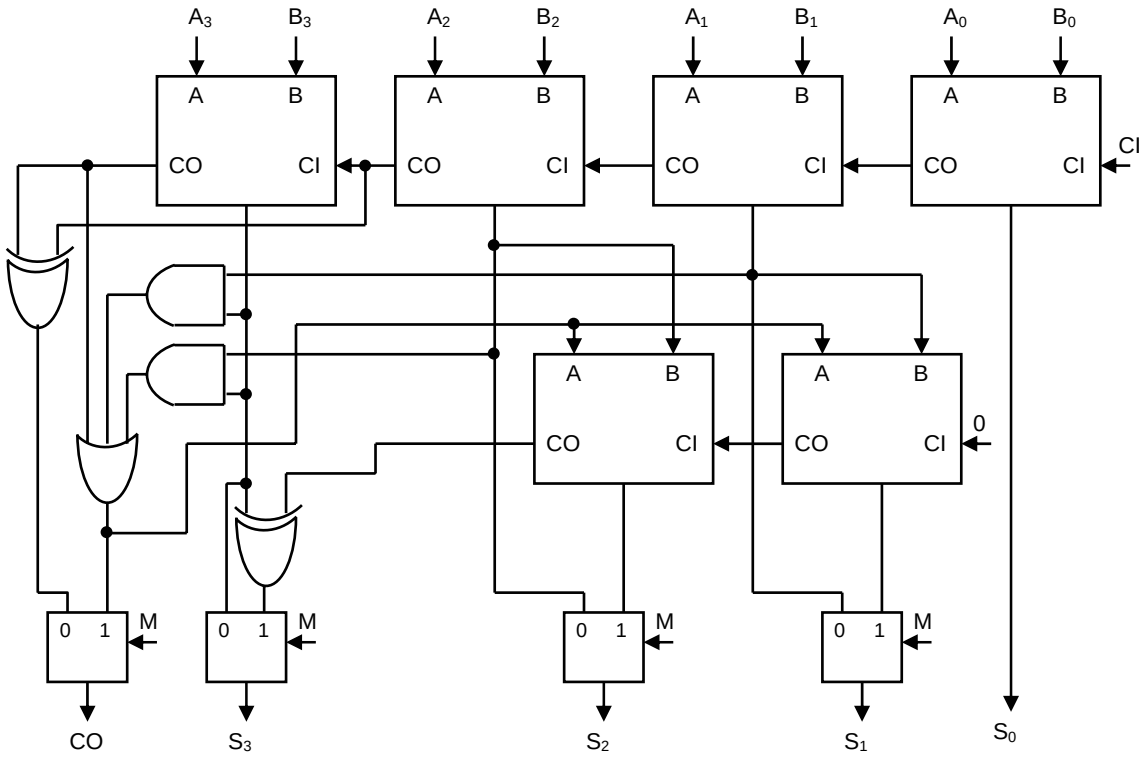
(b) 1000 + 1001 = 1 0001 + 0110 = 1 0111

(c) 0111 + 0011 = 1010 + 0110 = 1 0000

(d) 1001 1001 + 0001 0001 = 1010 1010 + 0110 0110 = 1 0001 0000

Exercise 5.19

By adding 2:1 mulitplexors on the outputs that are different for the BCD adder, CO, S_3, S_2, and S_1, we are able to create an adder that works as a standard binary adder or a BCD adder. The input M is the control signal for the added multiplexors. The resulting circuit looks as follows:



Exercise 5.20

A = 1101 and B = 1011. The result bits are in bold.

S_0: A_0B_0 =>	**1**	
S_1: A_1B_0 =>	0	
A_0B_1 =>	+ 1	
	1	
S_2: A_0B_2 =>	0	
A_1B_1 =>	+ 0	
	0	
A_2B_0 =>	+ 1	
	1	
S_3: A_0B_3 =>	1	
A_1B_2 =>	+ 0	
	1	
A_2B_1 =>	+ 1	
	1 0	Carry-out for second S4 add carry-in
A_3B_0 =>	+ 1	
	1	
S_4: A_1B_3 =>	0	
A_2B_2 =>	+ 0	
	0 1	Carry-in from S3
A_3B_1 =>	+ 1	
	1 0	Carry-out for second S5 add operand
	+ 0	Operand from third S3 add carry-out
	0	
S_5: A_2B_3 =>	1	
A_3B_2 =>	+ 0	
	1	
	+ 1	Operand from second S5 add carry-out
	1 **0**	
S_6: A_3B_3 =>	1 1	Carry-in from S5
	+ 1	Operand from second S5 add carry-out
	1 **0**	
S_7:	**1**	Bit from S6 carry-out

Result: 10001111, which is correct.

Exercise 5.21

A = 1101 and B = 1011. The ones that are carry-outs/ins are underlined and the result bits are in bold.

P_0: $(A_0B_0) + 0 = \mathbf{1}$

P_1: $(A_1B_0) + 0 = 0 + (A_0B_1) = \mathbf{1}$

P_2: $(A_2B_0) + 0 = 1 + (A_1B_1) = 1 + (A_0B_2) = \mathbf{1}$

P_3: $(A_3B_0) + 0 = 1 + (A_2B_1) = \underline{1}\ 0 + (A_1B_2) = 0 + (A_0B_3) = \mathbf{1}$

P_4: $(A_3B_1) + 0 + \underline{1} = \underline{1}\ 0 + (A_2B_2) = 0 + (A_1B_3) = \mathbf{0}$

P_5: $(A_3B_2) + \underline{1} = 1 + (A_2B_3) = \underline{1}\ \mathbf{0}$

P_6: $(A_3B_3) + 0 + \underline{1} = \underline{1}\ \mathbf{0}$

P_7: **1**

Result: 10001111, which is correct.

Exercise 5.22

The worst-case propagation delay through the multiplier in Figure 5.25 is 15 gates. This is the carry-out path through the first adders in S_1 and S_2, then the Sum paths for the first two adders and the carry-out path for the last adder in S_3, then the carry-out path through the last adder in S_4, S_5, and S_6.

Exercise 5.23

Designing a 2's compliment multiplier is not as difficult as it may seem at first. A nice property about multiplying signed numbers is the sign of the output is relatively easy to figure out. If the signs are the same, the result is positive, if they are different, the result is negative. This can be accomplished by XORing the highest-order bit of both operands together.

The operands need to be converted into a positive number before they are passed into the magnitude multiplier. This can be accomplished by flipping the bits and adding 1 if the high order bit is a one.

Putting the result back into 2's compliment is achieved in the same fashion. If the result of the XOR on the high order bits of the operands is a 1, the result's bits need to be flipped and have 1 added to them to turn it back into a negative 2's compliment number.

One thing to note is the range for a 4x4 2's compliment multiplier is -64 to 49. So the multiplier would still work if the result was 7 bits, rather than eight.

Exercise 5.25

The 4x4-combinational multiplier, in Figure 5.26, can be slightly rearranged to provide better propogation delay. The way to do this is to have the carry-outs pass their value to the adder that is diagonal from them rather than in front of them. This will net a propagation delay which is 4 gates better. The reason for this is that the carry-out has to propagate through one more gate then the sum does, so the adders in each row will all have valid inputs 1 gate delay sooner.