EECS 151/251A Homework 10

Due Sunday, April 22nd, 2018

Problem 1: LFSR

A particular linear feedback shift register (LFSR) is built using the primitive polynomial $x^{20}+x^3+1$. How many unique states does it have?

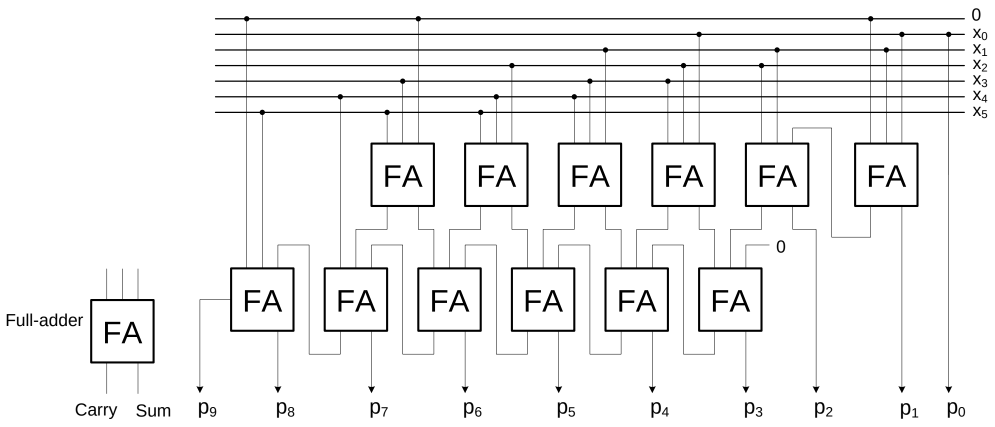
Problem 2: Constant Multiplier

Consider the design of a circuit for multiplying a constant, C, with a signed two-complement variable, X, such that $Y = C \times X$.

In this problem, let $C = 13_{10}$, and assume X is a 6-bit variable. Using only *full-adder blocks* (1-bit adders) draw a multiplier circuit. Your design objective is to minimize the total number of full-adder blocks, as well as the delay from input to output. Give priority to cost over delay.

Problem 3: Another Constant Multiplier

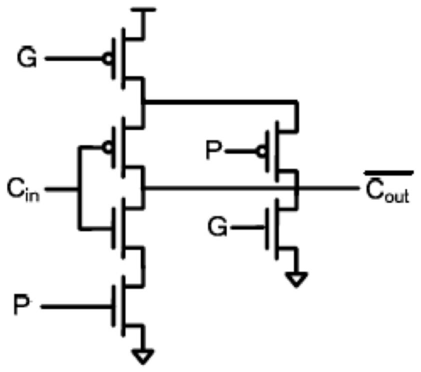
The circuit shown below is used to multiply the 6-bit number X by a 6-bit constant value, C. It is made up of instances of a full-adder cell. The full-adder takes as input 3 1-bit signals and outputs a 1-bit sum and a 1-bit carry.



What is the value of C?

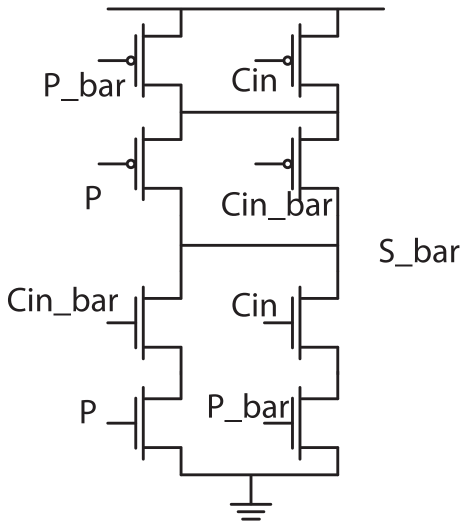
Problem 4: Adders

(a) Shown below is a static CMOS implementation of a gate that computes the carry-out at a particular bit position. If the P signal fed into the gate is calculated using $P = A + B$ (instead of $P = A \oplus B$, which is usually the case) would the output of this gate still be correct? Why or why not? If not, suggest a modification that gives the right output.



The only difference between the two cases is when $A = B = 1$. In that case the old version $(P = A \oplus B)$ is 0, but the new version $(P = A + B)$ is 1. However, the generate signal is 1, so the output of this gate is still low, and is therefore correct.

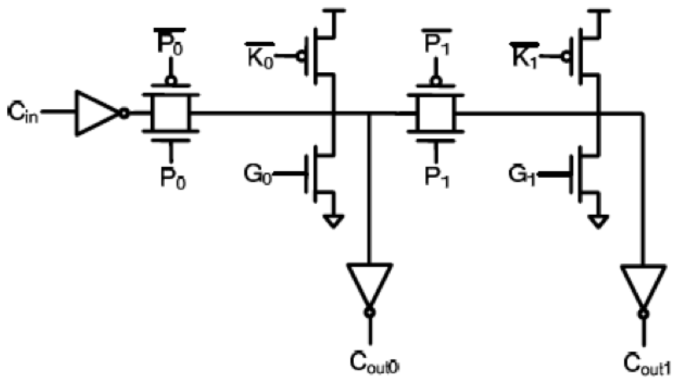
(b) Show the static CMOS implementation of a gate that computes the sum at a particular bit position $(S = P \oplus C_{in})$. Again, if the P signal fed into the gate is calculated using $P = A + B$, is the output of the gate correct? Why or why not? If not, suggest a modification that gives the right output.



In the case that $A = B = 1$, the output needs to be $S = C_{in}$, but that is not the case since $1 \oplus C_{in} = \overline{C_{in}}$. This means that the output is not correct.

One example of how to get the correct result is by using the generate signal. The logical expression becomes $S = (P \cdot \overline{G}) \oplus C_{in}$

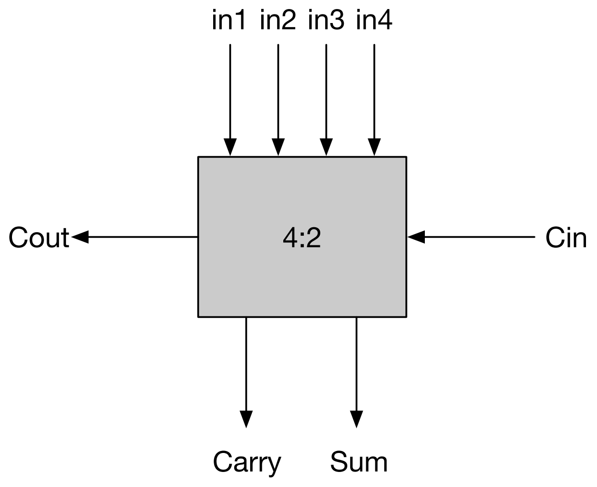
(c) The gate shown below is called a Manchester carry chain, and it computes the carry-out for two bit positions. Does this gate give the correct output if $P = A + B$? Why or why not?



Looking at individual bits, each gate works fine. But in the two-bit case, this gate will not function properly, because it allows cases where there is a fight between two devices that are trying to pull C_{out0} to ground and to VDD. For example, this happens when the first stage kills and the second generates a carry, and this case is avoided when the propagate signal is computed correctly (using an XOR).

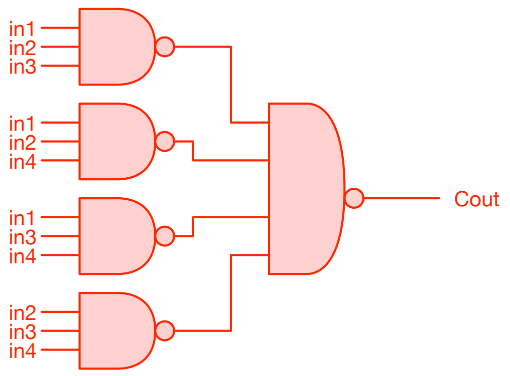
Problem 5: Multipliers

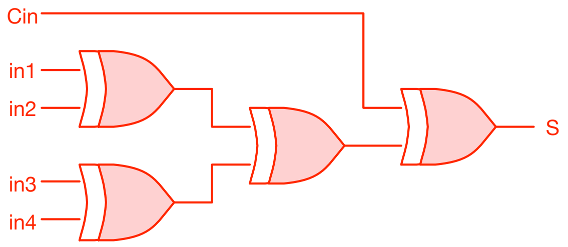
(a) Implement the 4:2 compressor described in the truth table below using NOT, AND, OR, and XOR gates. C_{out} and $Carry$ are both weight 2, while Sum and all inputs are weight 1. In the truth table, N refers to the total number of in signals at logic 1.

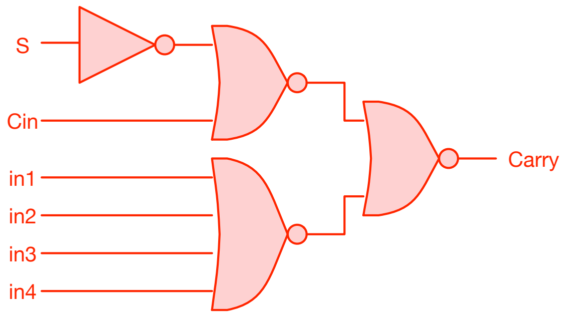


N	C_{in}	C_{out}	$Carry$	Sum
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	1	0	1
4	0	1	1	0
0	1	0	0	1
1	1	0	1	0
2	1	0	1	1
3	1	1	1	0
4	1	1	1	1

Here is a solution for the truth table given. There are solutions that use full adders for 4:2 compressors, but those have a slightly different truth table. There may be other possible solutions for this truth table.





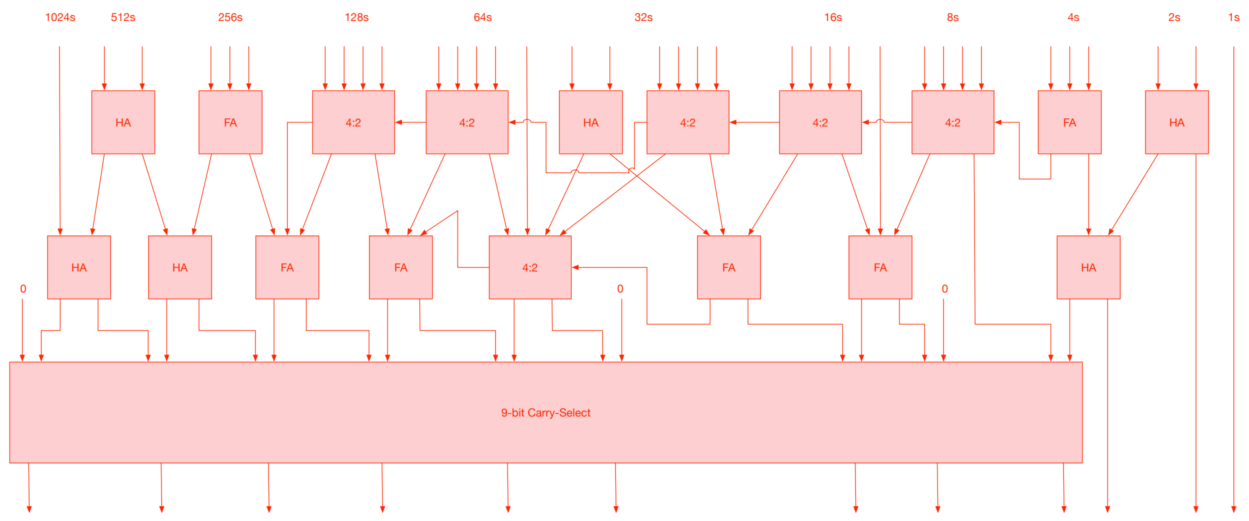


(b) Use this 4:2 compressor to implement a 6x6 unsigned Wallace-tree multiplier.

Start by listing the reduction layers:

```
0: 0 1 2 3 4 5 6 5 4 3 2 1
1: 0 2 2 3 2 4 3 3 1 2 1 1
2: 1 2 2 2 2 1 2 1 2 1 1 1
```

Now we are done, since we are left with two binary numbers to add. We can implement that with a 9-bit carry-select adder.



(c) Now use a 3:2 compressor (a full adder) to implement a 6x6 unsigned Wallace-tree multiplier. How many more reduction layers do you need?

```
0:  0  1  2  3  4  5  6  5  4  3  2  1
1:  0  2  2  2  4  4  4  3  3  2  1  1
2:  1  2  2  2  3  3  3  2  2  1  1  1
3:  2  2  2  2  2  2  2  2  1  1  1  1
```

You need 1 more layer.

(d) Convert your multiplier from (b) to a signed multiplier. How many more compressors did you use?

Now we need to sign extend each partial product:

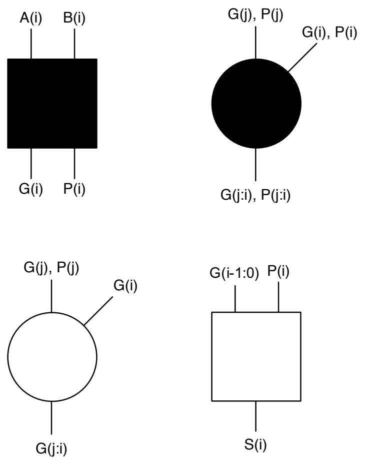
```
0:  6  6  6  6  6  6  6  5  4  3  2  1
1:  4  4  4  4  4  4  3  3  1  2  1  1
2:  2  2  2  2  2  1  2  1  2  1  1  1
```

In (b) we used 10 compressors on the first layer, and 8 on the second for a total of 18. In this problem, we now use 18 on the first layer, and 9 on the second, for a total of 27. We therefore use 9 more compressors.

Problem 6: Tree Adders

The goal of this problem is to design a 10-bit Kogge-Stone adder optimized for delay:

(a) Design the following logic blocks at a gate level. You may draw the gates or write the Boolean functions. Review your notes or the textbook to determine what goes inside each block. There are also numerous resources available online about these logarithmic adders. Use the given inputs and outputs as hints.



Black square: $G_i = A_iB_i$, $P_i = A_i \oplus B_i$
Black circle: $G_{j:i} = G_j + P_jG_i$, $P_{j:i} = P_jP_i$
White circle: $G_{j:i} = G_j + P_jG_i$
White square: $S_i = P_i \oplus G_{i-1:0}$

(b) Using the logic blocks you designed in part (a), design a 10-bit logarithmic adder with a carry input and a carry output. Use a radix-2 Kogge-Stone implementation. What is the critical path of your design? Give a block-level estimate, assuming that more complex blocks have more delay.

Lecture 20 Slide 22 shows the basic idea of a Kogge-Stone tree. The first stage is the black boxes: here we generate the bit propagate (P_i) and generate (G_i) signals that will be used by the tree. For the actual tree, the Kogge-Stone implementation first groups the (P_i, G_i) in groups of 2, therefore generating $(P_{1:0}, G_{1:0})$, $(P_{2:1}, G_{2:1})$ etc. Then those signals are grouped again in groups of 2 to form $(P_{3:0}, G_{3:0})$, $(P_{4:1}, G_{4:1})$ etc.

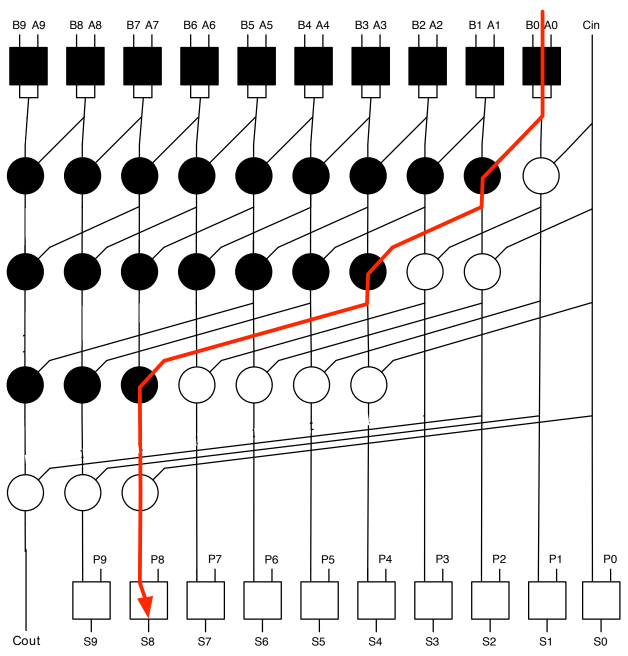
The key here is that you need to incorporate the C_{in} signal into the tree. Remember that in order to get a sum bit you need $S_i = P_i \oplus C_i = P_i \oplus G_{i-1:0}$. Therefore for S_0 you need P_0 and C_{in}, for S_1 you need P_1 and $G_0 + P_0C_{in}$ etc. So we add the white circles to generate the carries needed for the final sum, including the C_{in}.

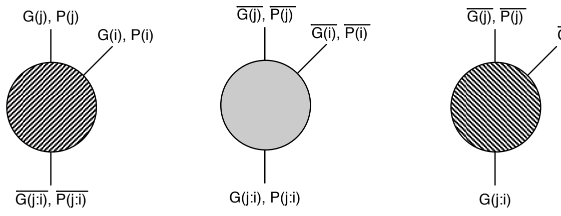
The critical path is shown on the tree (one example - there are multiple critical paths). In this case, a block-level estimate of the critical path is: $t_d = t_{blacksquare} + 3 * t_{blackcircle} + t_{whitecircle} + t_{whitesquare}$.

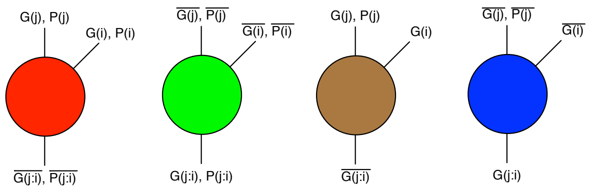
(c) **EECS 251A Only.** To save logic depth, we can create inverting logic stages instead of non-inverting logic stages. This removes the inverter at the output of each prefix block (the block that creates G(j:i) and P(j:i)). Design the following logic blocks and use them to modify your logarithmic adder from before. How many inverters were removed from your critical path?

We've made the circles colored to provide better visual difference. Note that you also need the inverted form of the last block above. Here they are for reference:

Red circle: $\overline{G_{j:i}} = \overline{G_j + P_jG_i}$, $\overline{P_{j:i}} = \overline{P_jP_i}$
Green circle: $G_{j:i} = \overline{G_j(P_j + G_i)}$, $P_{j:i} = \overline{P_j + P_i}$
Brown circle: $\overline{G_{j:i}} = \overline{G_j + P_jG_i}$
Blue circle: $G_{j:i} = \overline{G_j(P_j + G_i)}$

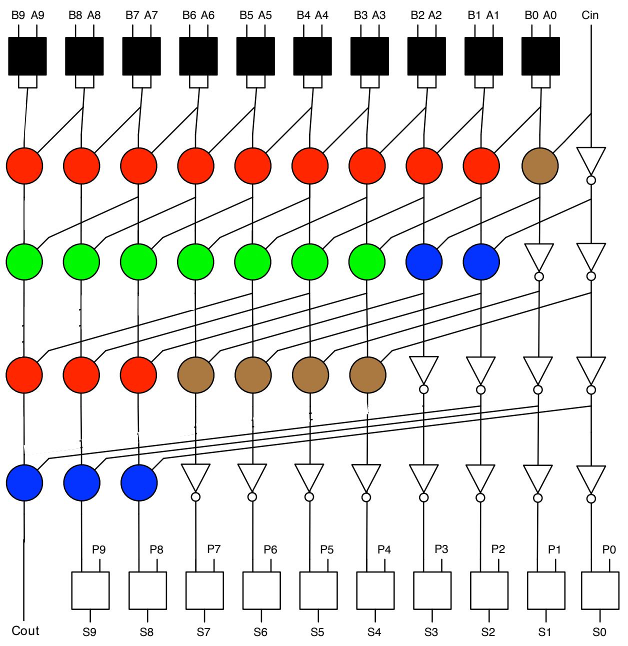



G(j), P(j)
G(i), P(i)
$\overline{G(j:i)}$, $\overline{P(j:i)}$
$\overline{G(j)}$, $\overline{P(j)}$
$\overline{G(i)}$, $\overline{P(i)}$
G(j:i), P(j:i)
$\overline{G(j)}$, $\overline{P(j)}$
$\overline{G(i)}$
G(j:i)


G(j), P(j)
G(i), P(i)
$\overline{G(j:i)}$, $\overline{P(j:i)}$
$\overline{G(j)}$, $\overline{P(j)}$
$\overline{G(i)}$, $\overline{P(i)}$
G(j:i), P(j:i)
G(j), P(j)
G(i)
$\overline{G(j:i)}$
$\overline{G(j)}$, $\overline{P(j)}$
$\overline{G(i)}$
G(j:i)

So each of the above circuits has one fewer inverter on each path. Now we can replace the blocks in the above diagrams with inverted forms, mapping inverted outputs to inverted inputs. Add inverters if the final output is inverted or if a polarity inversion is required.

The critical path has four fewer inverters, assuming it remains unchanged from before. Note all the inverters we had to add to ensure the polarity of each signal was correct. Since these are not on the critical path, likely they do not harm our delay.


B9 A9
B8 A8
B7 A7
B6 A6
B5 A5
B4 A4
B3 A3
B2 A2
B1 A1
B0 A0
Cin
P9
P8
P7
P6
P5
P4
P3
P2
P1
P0
Cout
S9
S8
S7
S6
S5
S4
S3
S2
S1
S0