Problem 1

a) In the first part, we need one XOR delay to obtain the first propagate, then two gate delays to reach the $c_{out}$. Then for the next 6 stages we need 2 gate delays from $c_{in}$ to $c_{out}$ in the final stage we have $c_{in}$ to sum XOR delay. The total delay is
\[ t_{total} = 4t_p + 6*2t_p + 2t_p = 18t_p \]

b) In part b we assume that the CLA logic functions implemented are
\[
\begin{align*}
    c_o3 &= g_3 + p_3 g_2 + p_2 g_1 + p_1 g_0 + p_0 c_{in} \\
    c_o2 &= g_2 + p_2 g_1 + p_1 g_0 + p_0 c_{in} \\
    c_o1 &= g_1 + p_1 g_0 + p_0 c_{in} \\
    c_o0 &= g_0 + p_0 c_{in}
\end{align*}
\]

Ripple the carry between two blocks

We spend an XOR delay to obtain g’s and p’s. Then from inputs to $c_{o3}$ we go through one 5 input AND and a 5 input OR, then the $c_{o3}$ is passed to the next stage and again it generates the $c_{o2}$ and finally goes through an XOR gate $c_{o2}$ sees 4 input AND an 4 input OR.
\[ 2tp + 0.25*5^2 tp + 0.25*5^2 tp + 0.25*4^2 tp + 0.25*4^2 tp + 2tp = (2 + 6.25 + 6.25 + 4 + 4 + 2)tp = 24.5tp \]

Select the sum out between two alternatives

As an alternative (faster) solution the second block performs a carry select operation. In the case both sums are generated in the second block. And we only need to choose using
a MUX. A MUX implements the function \( F = as + bs' \). So it has a delay with a two input AND, and a two input OR.

\[
t_{\text{total}} = 2tp + 0.25*5^2 tp + 0.25*5^2 tp + tp + tp = (2 + 6.25 + 6.25 + 1 + 1)tp = 16.5tp
\]

c) In the CLA adder it takes an XOR delay to generate the individual \( p_i, g_i \), 4-input AND and 4-input OR to generate the block \( P \) and \( G \)'s (\( 4tp + 4tp \)). From the outputs of the top level (in the diagram) it takes an additional 4-AND, 4-NAND delays (\( 4tp + 4tp \)) to generate mid level block \( P, G \). Then, 2 input AND, 2 input OR make us go through bottom level (\( tp + tp \)). After an additional 2 input AND, 2 input OR we go through middle level (\( tp + tp \)) and reach back at top level in the diagram. In this top level 2 input AND, 2 input OR (\( tp + tp \)) is needed to generate the final carry and, a final XOR (\( 2tp \)) is needed to obtain the sum.

\[
(2 + 4 + 4 + 4 + 2 + 2 + 2 + 2)tp = 26tp
\]

In the RCA case we again have \( 4tp + 30*2tp + 2tp = 66tp \).

As we can clearly see the as the number of bits increase the carry look ahead adder has a distinct advantage. But for adders with less than 10-bits it’s usually wiser to do the implementation simply in ripple carry.

The inputs of the top level are the individual \( p_i, g_i \). As mentioned in part b) the equations implemented are

\[
p_{i+3:i} = p_{i+3} p_{i+2} p_{i+1} p_i
\]

\[
g_{i+3:i} = g_{i+3} + p_{i+3} g_{i+2} + p_{i+3} g_{i+1} + p_{i+3} g_i
\]

we can see that the worst case delay is 4 input AND + 4 input OR

\( g_{i+3:i} \) means a carry is generated within the “block encompassing bit positions \( i+3 \) to \( i \)”

\( p_{i+3:i} \) means the carry in of the block is passed to the carry out of the block.

The mid level blocks implement

\[
p_{i+15:i} = p_{i+15:i+12} p_{i+11:i+8} p_{i+7:i+4} p_{i+3:i}
\]

\[
g_{i+15:i} = g_{i+15:i+12} + p_{i+15:i+12} g_{i+11:i+8} + p_{i+15:i+12} g_{i+11:i+8} + p_{i+15:i+12} p_{i+11:i+8} g_{i+7:i+4} + p_{i+15:i+12} p_{i+11:i+8} p_{i+7:i+4} g_{i+3:i}
\]
Once we have the $p_{i:k}$ and $g_{i:k}$’s and $c_{o(k-1)}$ (i.e. the carry out at stage k-1), we can obtain the carry out of stage i using the relation

$$c_{oi} = g_{i:k} + p_{i:k} c_{o(k-1)}$$

(Has a delay of 2input AND and 2-input OR)

Meaning that to get a carry out at i’th bit position, the block encompassing i - k should either generate a carry or pass the carry coming as $c_{o(k-1)}$

**Problem 2**

**a)** For the first part of this problem the simplest way of implementation is check to see if $B-A < 0$ if so the sign of the difference would be 1 to signify that $B < A$ the block diagram for this would be

The MSB of the output is at the same time the output of the comparator

**b)** Assuming the adder is a Ripple carry adder. The worst case delay of the comparator would be, $4*t_p + (N-2)*2t_p + 2*t_p = (N+1)*2t_p$
Problem 3

For this third problem we will implement the technique called Baugh-Wooley multipliers.

Consider the multiplication of two signed binary numbers where the paranthesis signify negative weighted bit.

\[
\begin{array}{c|cccc}
(a_3) & a_2 & a_1 & a_0 \\
(b_3) & b_2 & b_1 & b_0 \\
\hline
(a_3b_0) & a_2b_0 & a_1b_0 & a_0b_0 \\
(a_3b_1) & a_2b_1 & a_1b_1 & a_0b_1 \\
(a_3b_2) & a_2b_2 & a_1b_2 & a_0b_2 \\
a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 \\
\end{array}
\]

\[
\begin{array}{c|cccc}
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 \\
\hline
(a_3) & a_2 & a_1 & a_0 \\
(b_3) & b_2 & b_1 & b_0 \\
a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 \\
a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 \\
a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 \\
\end{array}
\]

As we see above the partial products can be classified as positive and negative terms. We are trying to add the positive partial products (above dashed line) and subtract the negative partial products (those below the dashed line).

The negatively weighted partial products can be interpreted as two numbers that must be subtracted. Instead of subtracting, they can be negated and the whole array can be added. We need to perform this negation smartly.

For \( a_3 = 1 \) we want to negate the number

\[
0 \ 0 \ (a_3b_2) \ (a_3b_1) \ (a_3b_0) + 1
\]

This requires inverting all the bits and adding a 1. The logic function that produces the above results for \( a_3 = 1 \) and zero for \( a_3 = 0 \) is \( a_3 \overline{b_1} \). For the leading two positions we can interpret is either as:

\[
a_3 \ 0 \ ...
\]

or equivalently as:

\[
1 \ \overline{a_4} \ ...
\]

+1
If the method is used also for negating the partial products starting with \( b_3 \) and the results are entered back into the initial partial sum array. We would get.

\[
\begin{array}{cccc}
(a_3) & a_2 & a_1 & a_0 \\
(b_3) & b_2 & b_1 & b_0 \\
\hline
a_3b_3 & 0 & a_2b_2 & a_1b_2 & a_0b_2 \\
\hline
b_3 & a_2b_3 & a_1b_3 & a_0b_3 \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
\]

We can implement these changes in the following way.
Using the block below as the processing element of the main array we can obtain the result \( p[7:0] \).