

UNIVERSITY OF CALIFORNIA
College of Engineering
Department of Electrical Engineering and Computer Sciences
Last modified on April 18, 2002 by Tufan Karalar (tufan@eecs.berkeley.edu)

Jan M. Rabaey, Andrei Vlademirescu
 EECS 141

Homework #9 Solutions

Due 04/19/02, 5pm, in 558 Cory

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

$$t_{total} = 4 * t_p + 6 * 2t_p + 2t_p = 18t_p$$

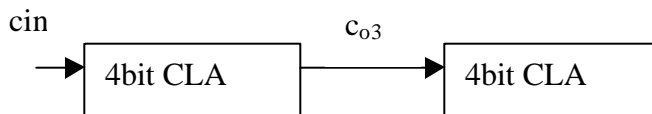
b) In part b we assume that the CLA logic functions implemented are

$$c_{o3} = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_{in}$$

$$c_{o2} = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_{in}$$

$$c_{o1} = g_1 + p_1 g_0 + p_1 p_0 c_{in}$$

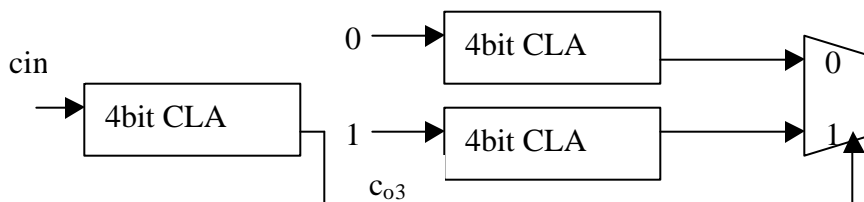
$$c_{o0} = g_0 + p_0 c_{in}$$



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 and 4 input OR.

$$2t_p + 0.25 * 5^2 t_p + 0.25 * 5^2 t_p + 0.25 * 4^2 t_p + 0.25 * 4^2 t_p + 2t_p = (2 + 6.25 + 6.25 + 4 + 4 + 2)t_p = 24.5t_p$$



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.

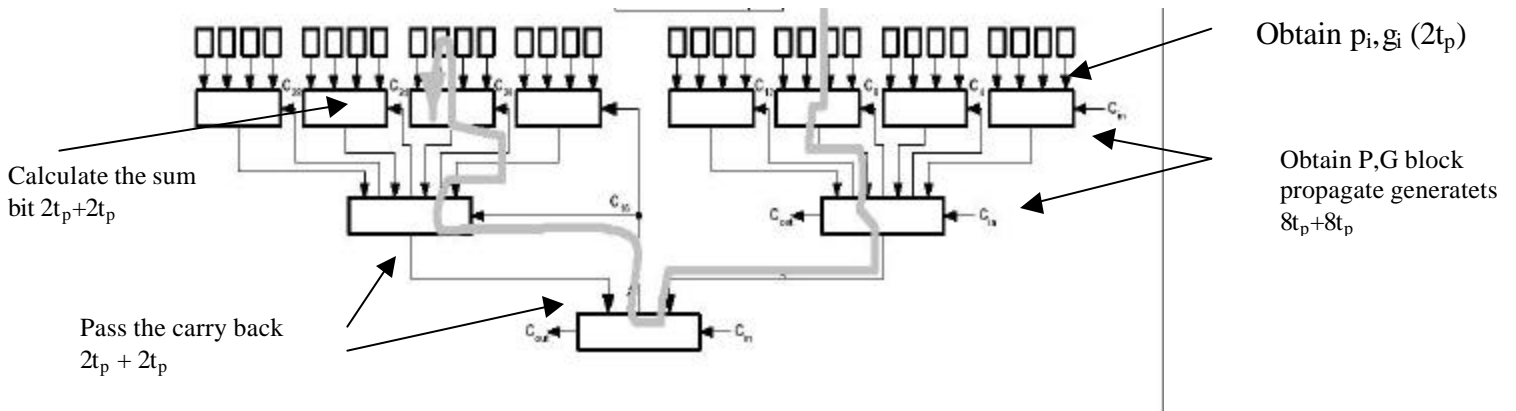
$$t_{total} = 2t_p + 0.25 \cdot 5^2 t_p + 0.25 \cdot 5^2 t_p + t_p + t_p = (2 + 6.25 + 6.25 + 1 + 1)t_p = 16.5t_p$$

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

$$(2 + 4 + 4 + 4 + 4 + 2 + 2 + 2 + 2)t_p = 26t_p$$

In the RCA case we again have $4t_p + 30 \cdot 2t_p + 2t_p = 66t_p$.

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 its 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

$$p_{i+3:i} = p_{i+3} p_{i+2} p_{i+1} p_i$$

$$g_{i+3:i} = g_{i+3} + p_{i+3} g_{i+2} + p_{i+3} p_{i+2} g_{i+1} + p_{i+3} p_{i+2} p_{i+1} g_i$$

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

$$p_{i+15:i} = p_{i+15:i+12} p_{i+11:i+8} p_{i+7:i+4} p_{i+3:i}$$

$$g_{i+15:i} = g_{i+15:i+12} + p_{i+15:i+12} g_{i+11:i+8} + p_{i+15:i+12} p_{i+11:i+8} g_{i+7:i+4} + p_{i+15:i+12} p_{i+11:i+8} p_{i+7:i+4} g_{i+3:i}$$

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

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

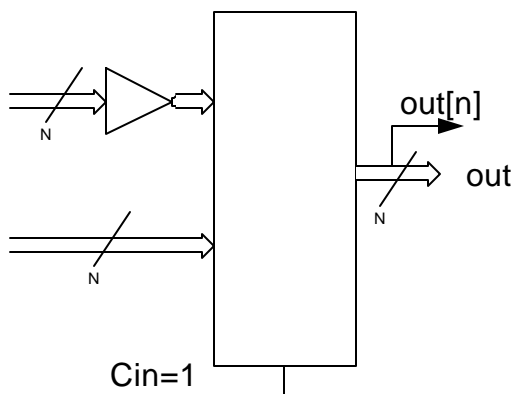
(Has a delay of 2input AND and 2-input OR)

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, $4*t_p + (N-2)*2*t_p + 2*t_p = (N+1)*2*t_p$

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.

				(a ₃)	a ₂	a ₁	a ₀	
				(b ₃)	b ₂	b ₁	b ₀	
				(a ₃ b ₀)	a ₂ b ₀	a ₁ b ₀	a ₀ b ₀	
			(a ₃ b ₁)	a ₂ b ₁	a ₁ b ₁	a ₀ b ₁		
		(a ₃ b ₂)	a ₂ b ₂	a ₁ b ₂	a ₀ b ₂			
	a ₃ b ₃	(a ₂ b ₃)	(a ₁ b ₃)	(a ₀ b ₃)				
p7	p6	p5	p4	p3	p2	p1	p0	
				(a ₃)	a ₂	a ₁	a ₀	
				(b ₃)	b ₂	b ₁	b ₀	
					a ₂ b ₀	a ₁ b ₀	a ₀ b ₀	
				a ₂ b ₁	a ₁ b ₁	a ₀ b ₁		
	a ₃ b ₃	0	a ₂ b ₂	a ₁ b ₂	a ₀ b ₂			
				(a ₃ b ₁)	(a ₃ b ₀)			
		(a ₃ b ₂)	(a ₁ b ₃)	(a ₀ b ₃)				
p7	p6	p5	p4	p3	p2	p1	p0	

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

$$\begin{array}{ccccccc}
 0 & 0 & (a_3b_2) & (a_3b_1) & (a_3b_0) & & \\
 & & & & & & +1
 \end{array}$$

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 $a_3\overline{b_i}$. For the leading two positions we can interpret is either as:

$$a_3 \overline{a_3} \dots$$

or equivalently as:

$$\begin{array}{c}
 1 \quad \overline{a_4} \dots \\
 +1
 \end{array}$$

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.

				(a_3)	a_2	a_1	a_0
				(b_3)	b_2	b_1	b_0
					a_2b_0	a_1b_0	a_0b_0
				a_2b_1	a_1b_1	a_0b_1	
	$\overline{a_3b_3}$	0	$\overline{a_2b_2}$	$\overline{a_1b_2}$	$\overline{a_0b_2}$		
	$\overline{a_3}$	$\overline{a_3b_2}$	$\overline{a_3b_1}$	$\overline{a_3b_0}$			
	$\overline{b_3}$	$\overline{a_2b_3}$	$\overline{a_1b_3}$	$\overline{a_0b_3}$			
1				a_3			
				b_3			
P_7	P_6	p_5	p_4	p_3	p_2	p_1	p_0

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 $p[7:0]$

