Carry Lookahead Adders
Ivor Page[1]

4.1 Simplest Carry-Lookahead System

Recall the following equations from the section on Adder Circuits:

$$\begin{aligned} g_i &= x_i y_i \\ p_i &= x_i \oplus y_i \\ t_i &= \overline{a_i} = x_i + y_i \end{aligned}$$

The *transfer* signal t_i can be used in preference to p_i since it is easier and quicker to generate. Although we will use the signal t_i in practice, we shall use p_i in the following since it makes the derivation easier to understand.

The carry-lookahead system is obtained by unwinding the recurrence relation for c_{i+1}:

$$\begin{aligned} c_i &= g_{i-1} + c_{i-1}p_{i-1} \\ &= g_{i-1} + (g_{i-2} + c_{i-2}p_{i-2})p_{i-1} \\ &= g_{i-1} + g_{i-2}p_{i-1} + c_{i-2}p_{i-2}p_{i-1} \\ \ldots & \\ &= g_{i-1} + g_{i-2}p_{i-1} + g_{i-3}p_{i-2}p_{i-1} + g_{i-4}p_{i-3}p_{i-2}p_{i-1} + c_{i-4}p_{i-4}p_{i-3}p_{i-2}p_{i-1} \end{aligned}$$

We could continue unrolling until the entire adder length is covered, i.e. until we reach c_{in}, but the fan-in of the gates grows linearly with the number of stages covered by them. Fan-in much greater than 4 is impractical for single CMOS gates because of poor noise immunity, poor rise and fall times, and therefore poor delay.

For most implementations, we stop at carry-lookahead blocks of size 4. Recall, in the following, t_i can be used in place of p_i.

$$\begin{aligned} c_4 &= g_3 + g_2p_3 + g_1p_2p_3 + g_0p_1p_2p_3 + c_0p_0p_1p_2p_3 \\ c_3 &= g_2 + g_1p_2 + g_0p_1p_2 + c_0p_0p_1p_2 \\ c_2 &= g_1 + g_0p_1 + c_0p_0p_1 \\ c_1 &= g_0 + c_0p_0 \end{aligned}$$

Figure 1 shows a complete 4-bit CLA adder. On the right of the drawing are the logic diagrams of the two blocks that generate the p, t and sum signals. Each lower block combines the c_i signal with t_i and g_i to give the sum output.

$$z_i = x_i \oplus y_i \oplus c_i = p_i \oplus c_i = \overline{g_i}t_i \oplus c_i$$

[1]University of Texas at Dallas

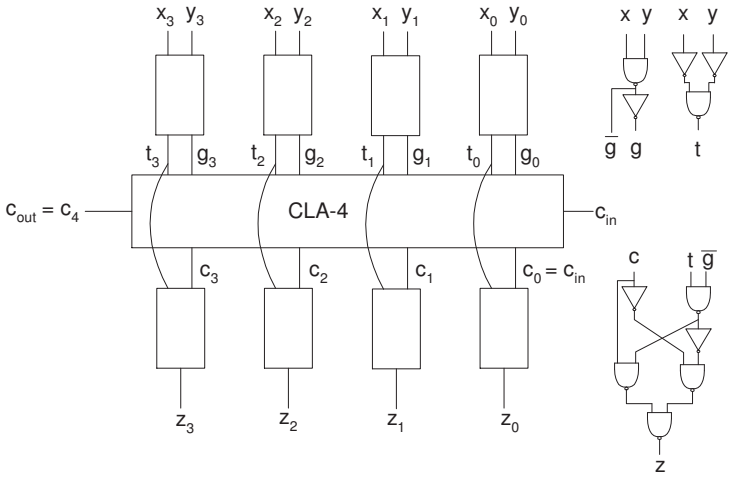

Figure 1: Four bit Carry-Lookahead Adder

If implemented strictly as implied, the delay through the carry-lookahead block will be just $2D$, but note that the AND gate in two of the equations has a fan-in of 4, and there are OR gates with fan-ins of 2,3,4, and 5. Fortunately we don't need the carry-out signal c_4. If CLA blocks larger than size 4 are needed, then the 2 levels of the CLA become 4 levels or more in order to preserve the fain-in of 4. We shall need two additional signals, the block propagate signal P and the block generate signal G. Think of these as the result of combining 4 neighboring bits of the adder into one HEX digit. Each HEX digit can generate or propagate a carry.

$$\begin{aligned} G &= g_3 + g_2p_3 + g_1p_2p_3 + g_0p_1p_2p_3 \\ P &= p_3p_2p_1p_0 \end{aligned}$$

These block propagate and generate signals can be combined in groups of four by using a second level CLA-4 block. The process of adding levels continues in this fashion. For a k bit adder there will be $\lceil log_4\ k \rceil$ levels.

Figure 3 shows a 32-bit adder using 3 levels of CLA blocks. Most of the detail has been excluded to make the carry paths more clear.

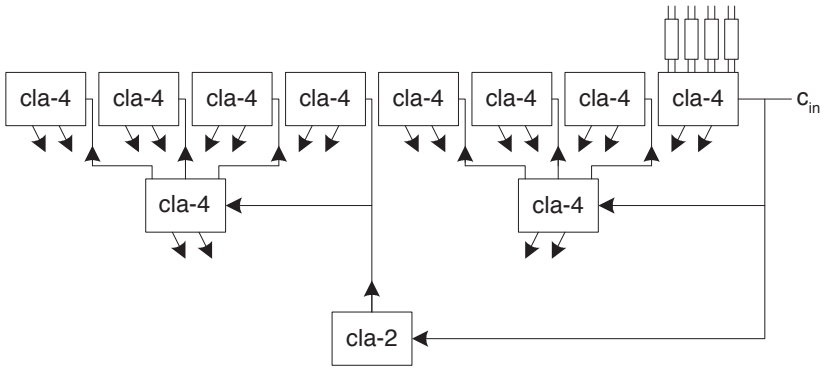

Figure 2: 32 Bit CLA adder using CLA-4 blocks

The inputs to each CLA block in the first level are four pairs of p and g signals and a carry-in. The CLA blocks produce group propagate and generate signals, P and G that are passed down the tree, plus three carry signals that are passed back up the tree. The lowest level P and G signals are not used. If a carry-out from the entire adder is needed, it can be produced by Anding the lowest level CLA's P signal with the carry in.

A 64-bit adder is obtained by making the lowest level CLA unit a CLA-4 block and replicating the upper levels.

Below is a 20 bit adder using CLA-3 blocks. All the connections are shown.

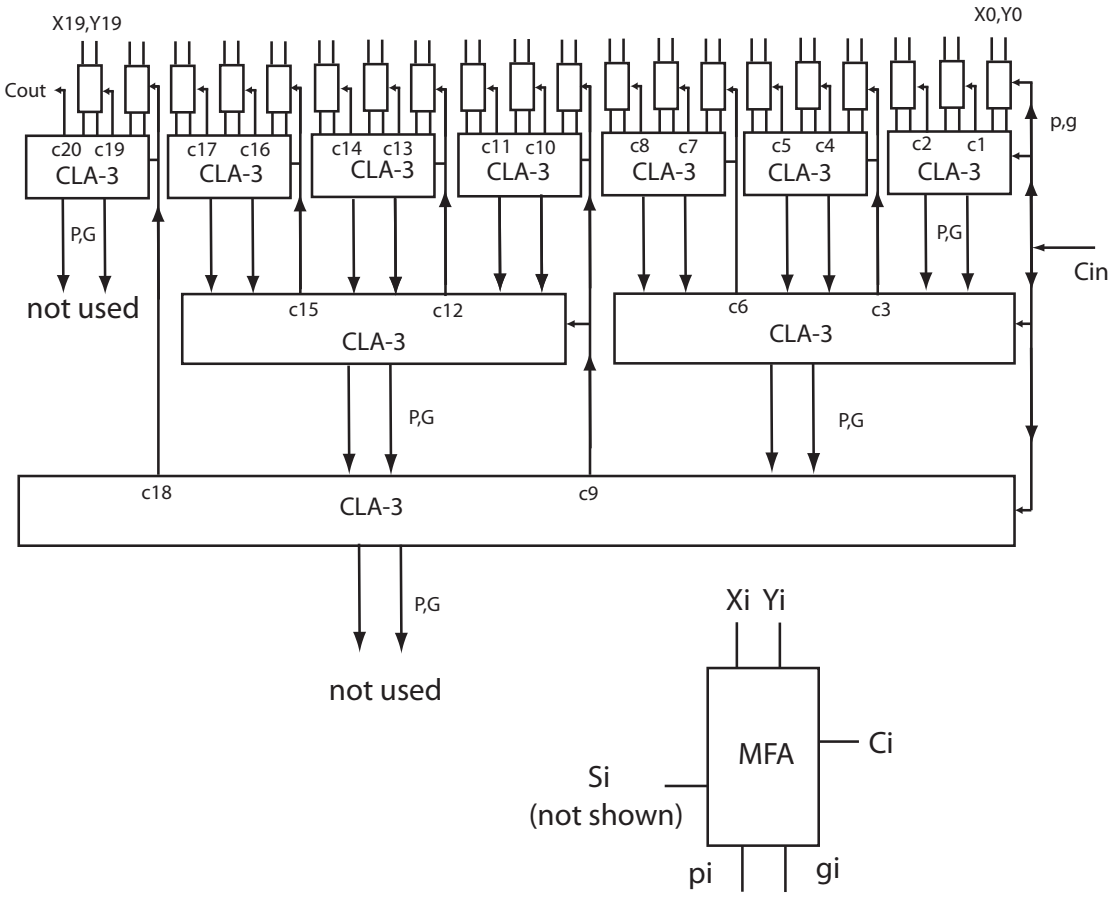

Figure 3: 20 Bit CLA adder using CLA-3 blocks

The delay through the adder using CLA-m units comprises:

- time to produce the p_i, g_i signals, approx $2D$
- propagation through $h = log_m\ k$ levels of CLAs on the way down the tree, approximately $2hD$
- propagation through $h - 1$ levels of CLAs on the way up the tree, approximately $2(h-1)D$
- final production of sum signals, approximately $3D$

The total delay is then approximately $5D + (4h-2)D = 5D + (4log_m\ k - 2)D$. For a 64-bit adder with CLA-4 blocks the time is $15D$, and for a 256-bit adder with CLA-4 blocks, the time is $19D$. We have achieved log-time addition. Contrast the delay of the CLA adder with the worst case delay for the ripple-carry adder, $(2k+1)D$.

Consider the circuit cost of ripple-carry and CLA adders. In the original NAND version of the ripple-carry adder there are $2k$ half adders plus k inter-stage OR gates. The total number of gate inputs is $2k \times 8 + k \times 2 = 18k$. In CMOS, each input represents 2 transistors, so the total number of transistors is $36k$. As we saw, there are full-adder circuits requiring only 28 transistors.

For a 64 bit adder, the number of transistors would be 2,304 for the half-adder NAND version and 1,792 for the 28T version.

In a 64-bit adder using CLA-4 units, there are 21 CLA-4 blocks each with 45 gate inputs, making a total of 945 gate inputs. The t_i and g_i signals require 7 gate inputs per stage, and the final sum circuit requires a further 10 gate inputs, making a total of 1088 gate inputs, for a grand total of 4066 transistors. This is a 77% increase over the number of transistors needed in the NAND version employing half-adders.

The reduction in delay is from $129D$ to $15D$, a reduction of more than 8 times.

In a 256-bit adder, the full NAND version based on half-adders requires 9,216 transistors, while the adder based on CLA-4 units requires 16,354 transistors. Again, only a 77% increase. The delay reduces from $513D$ to $19D$, a reduction of 27 times.

In general, the number of CLA-m units required in a k bit adder is:

$$\begin{aligned} &= \sum_{i=1}^{log_m\ k} \frac{k}{m^i} \\ &= \frac{k}{m} + \frac{k}{m^2} + \cdots + 1 \\ &= \frac{k}{(m-1)} \left(1 - \left(\frac{1}{m}\right)^{log_m\ k}\right) \\ &= \frac{(k-1)}{(m-1)} \end{aligned}$$

When using CLA-4 blocks, the total number of transistors needed is then $90(k-1)/3+34k = 64k - 30$. For the NAND based circuit using half-adders, the number of transisitors needed is $36k$. The increase about 1.78 times.

4.1.1 Summary

Type	Normalized Delay	Number of Transistors
Ripple-Carry with Half-Adders	$(2k+1)D$	$36k$
CLA-4 Based	$5D + (4log_4\ k - 2)D$	$64k - 30$

4.2 Ling Adder

The Ling design propagates $h_i = c_i + c_{i-1}$ instead of c_i. We derive the equations for the signals in simple steps.

$$c_{i-1}p_{i-1} = c_{i-1}p_{i-1} + g_{i-1}p_{i-1} + p_{i-1}c_{i-1}p_{i-1}$$

This seems a pretty strange equation. We have the term $c_{i-1}p_{i-1}$ appearing three times and the remaining term $g_{i-1}p_{i-1}$ is zero. If we rewrite the RHS as $A + B + AC$, where $B = 0$, we can see that the RHS simplifies to A, so the equation is correct.

$$\begin{aligned} c_{i-1}p_{i-1} &= c_{i-1}p_{i-1} + (g_{i-1} + p_{i-1}c_{i-1})p_{i-1} \\ &= (c_{i-1} + c_i)p_{i-1} \\ &= h_i p_{i-1} \end{aligned}$$

This result is needed below:

$$\begin{aligned} c_i &= g_{i-1} + c_{i-1}p_{i-1} \\ &= h_i g_{i-1} + h_i p_{i-1} \\ &= h_i(g_{i-1} + p_{i-1}) \\ &= h_i t_{i-1} \end{aligned}$$

In the second line above, $h_i g_{i-1}$ is substituted for g_{i-1}. This substitution is clearly correct for $g_{i-1} = 0$. When $g_{i-1} = 1$, $h_i = 1$ since $c_i = 1$. The second term in that line comes from the result above.

And finally,

$$\begin{aligned} h_i &= c_i + c_{i-1} \\ &= (g_{i-1} + c_{i-1}p_{i-1}) + c_{i-1} \\ &= g_{i-1} + c_{i-1} \\ &= g_{i-1} + h_{i-1}t_{i-2} \end{aligned}$$

The significance of this result is seen when this equation is unrolled:

$$\begin{aligned} h_i &= g_{i-1} + h_{i-1}t_{i-2} \\ &= g_{i-1} + t_{i-2}(g_{i-2} + h_{i-2}t_{i-3}) \\ &= g_{i-1} + g_{i-2} + h_{i-2}t_{i-2}t_{i-3} \\ &= g_{i-1} + g_{i-2} + g_{i-3}t_{i-3}t_{i-2} + h_{i-3}t_{i-4}t_{i-3}t_{i-2} \\ &= g_{i-1} + g_{i-2} + g_{i-3}t_{i-2} + g_{i-4}t_{i-3}t_{i-2} + h_{i-4}t_{i-4}t_{i-3}t_{i-2} \end{aligned}$$

The term h_i requires only 14 gate inputs, as compared with to 19 for the previous CLA-4 design's c_4 signal. Similar savings accrue for the other CLA signals.

4.3 Parallel Prefix Computation

A parallel prefix computation can be defined with any associative operator. Here is an example using integers and the + operator.

Given:	x_0	x_1	x_2	$\cdots$	x_{k-1}
Find:	$s_0 =$	$s_1 =$	$s_2 =$	$\cdots$	$s_{k-1} =$
	x_0	$x_0 + x_1$	$x_0 + x_1 + x_2$	$\cdots$	$x_0 + x_1 + \cdots + x_{k-1}$

There are many ways to use the divide and conquer approach to compute these sums in parallel. The first scheme is illustrated in Figure 4. Two prefix sum units each operate on 1/2 of the inputs. The first unit operates on inputs $x_0, x_1, \cdots, x_{k/2-1}$ and produces the answers $s_0, s_1, \cdots, s_{k/2-1}$. The second unit is identical to the first, and operates on inputs $x_{k/2}, x_{k/2+1}, \cdots, x_{k-1}$. It produces partial sums $p_{k/2}, p_{k/2+1}, \cdots, p_{k-1}$. The $s_{k/2-1}$ output from the first unit must be added to each of these partal sums to form the remaining sum outputs: $s_{k/2} = p_{k/2} + s_{k/2-1}, s_{k/2+1} = p_{k/2+1} + s_{k/2-1}, \cdots s_{k-1} = p_{k-1} + s_{k/2-1}$.

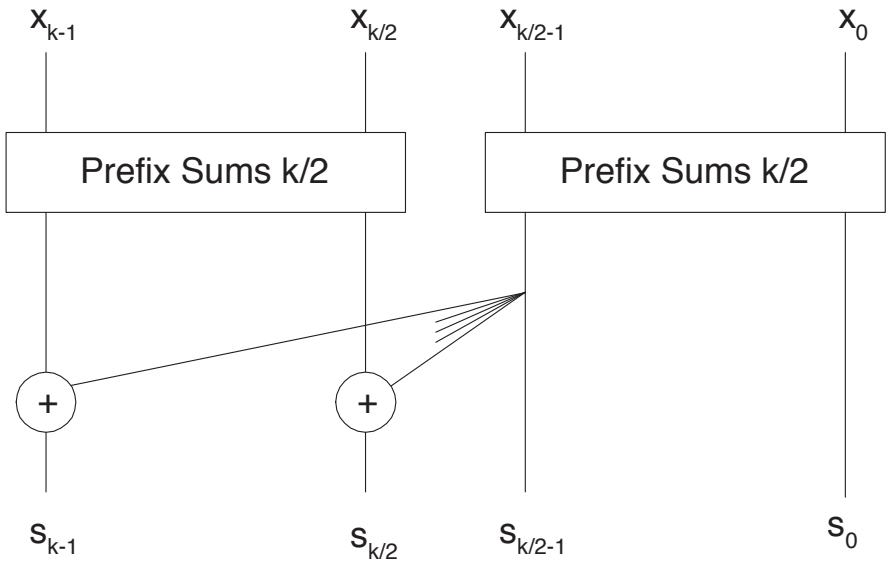

Figure 4: First Scheme for Prefix Sum Calculation

The delay through the entire network is characterized by the recurrence:

$$D(k) = D(k/2) + 1 = log_2\ k$$

This states that repeated application of this particular divide and conquer strategy leads to $log_2\ k$ levels of adders. The cost of the network is similarly characterized and gives the number of adders:

$$C(k) = 2C(k/2) + k/2 = (k/2)log_2\ k$$

The second divide and conquer method computes sums of pairs of inputs, $x_0 + x_1, x_2 + x_3, \cdots, x_{k-2} + x_{k-1}$. These sums can again be added in pairs, and so on, The results of these computations at various levels can be added to form all the prefix sums needed. For example,

$$\begin{aligned} s_6 &= x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \\ &= [(x_0 + x_1) + (x_2 + x_3)] + (x_4 + x_5) + x_6 \end{aligned}$$

Figure 5 shows three levels of prefix sum computation (in blue rectangles) with the direct results shown in black and the additional adders and results shown in mauve.

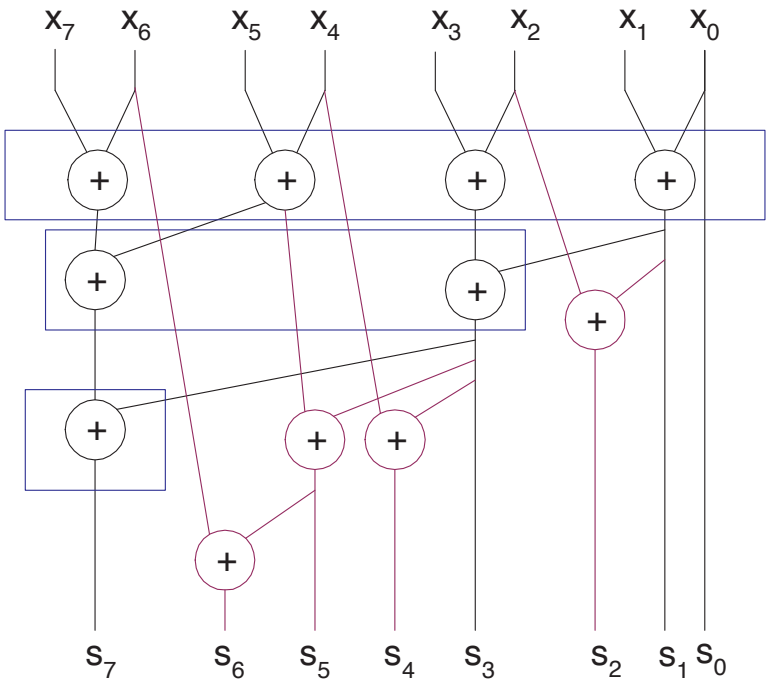

Figure 5: Second Scheme for Prefix Sum Calculation

The delay through this network is given by:

$$D(k) = D(k/2) + 2 = 2log_2\ k - 2$$

The cost of the second scheme is given by the recurrence:

$$C(k) = C(k/2) + k - 1 = 2k - 2 - log_2\ k$$

The first design is faster, $log_2\ k$ levels as opposed to $2log_2\ k - 2$ but it is also much more expensive, $(k/2)log_2\ k$ adders as opposed to $2k - 2 - log_2\ k$ adders. The first design also has huge fan-out for the output $s_{k/2-1}$. That output must be added to $k/2$ partial sums from the second prefix sum unit.

There are many variations on this theme.

Figure 6 shows the Brent Kung parallel prefix network for 16 inputs. Its delay is $2log_2\ k - 2$ levels and its cost is $2k - 2 - log_2\ k$ cells.

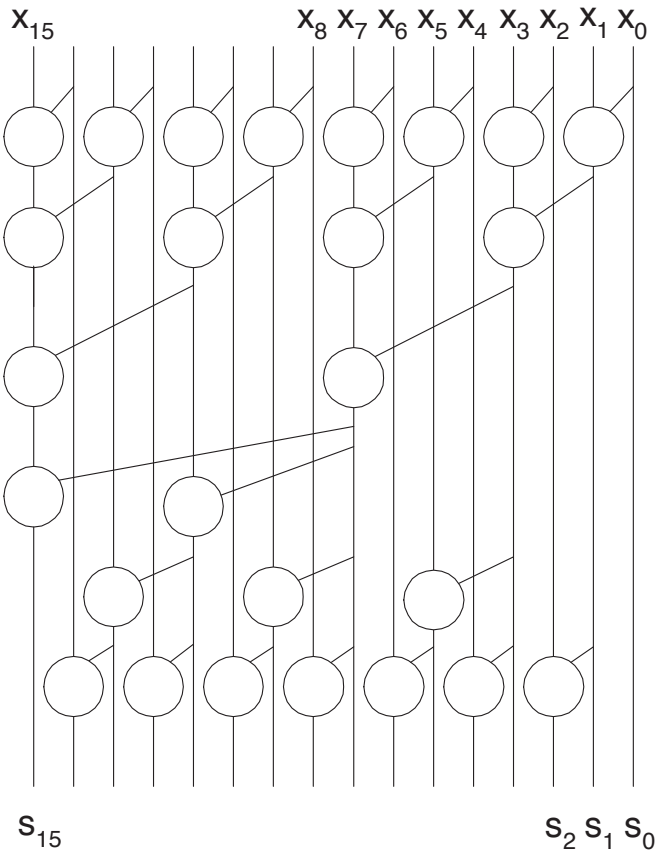

Figure 6: Brent-Kung Parallel Prefix Network for $k = 16$

Figure 7 shows the Kogge Stone parallel prefix network for 16 inputs. Its delay is $log_2\ k$ levels and its cost is $klog_2\ k - k + 1$ cells. It has optimal speed for a parallel prefix network.

These ideas and those from other parallel prefix networks can be combined to form compromises between speed and cost, as discussed in the text.

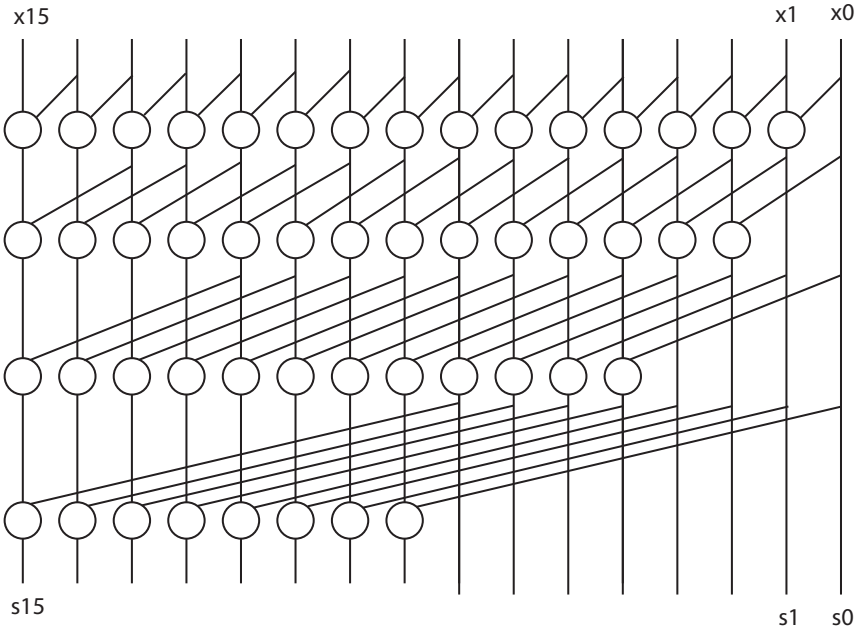

Figure 7: Kogge-Stone Parallel Prefix Network for $k = 16$

The following table summarizes the results. L is the delay through one network level and C is the cost of one adder cell.

4.3.1 Summary of Results for Prefix Sum Networks

	Delay in	Cost	$k = 32$		$k = 64$		$k = 128$	
Type	levels	in cells	Delay	Cost	Delay	Cost	Delay	Cost
Scheme 1	$log_2\ k$	$(k/2)log_2\ k$	5L	80C	6L	192C	7L	448C
Scheme 2	$2log_2\ k - 2$	$2k - 2 - log_2\ k$	8L	57C	10L	120C	12L	247C
Brent Kung	$2log_2\ k - 2$	$2k - 2 - log_2\ k$	8L	57C	10L	120C	12L	247C
Kogge Stone	$log_2\ k$	$klog_2\ k - k + 1$	5L	129C	6L	321C	7L	769C

Kogge Stone has the clear speed advantage. Its cost is considerably higher than that of Brent Kung. See the comparison at the end of this module with the CLA-4 based adder.

4.4 Carry Generation using a Parallel Prefix Computation

We have been considering networks that compute the prefix sums of a vector of integers. In the networks presented, all the cells were adders. Our task in carry generation is to compute carries as prefix computations. To do so, we define a carry function, $\hat{c}$, and replace the adder cells in the prefix sum networks with units that compute this function.

Consider two contiguous regions of the input digits, say digit positions $[i, j-1]$ and $[j, h]$ $i < j < h$. Denote P and G signals for the two regions by $g' = g_{[i,j-1]}$, $p' = p_{[i,j-1]}$, and $g'' = g_{[j,h]}$, $p'' = p_{[j,h]}$, and group them as follows: (g', p') and (g'', p''). Then the P and G signals for the combined region are

$$\begin{aligned} g &= g'' + g'p'' \\ p &= p'p'' \end{aligned}$$

These two equations define the $\hat{c}$ operator.

Define the *carry operator*, $\hat{c}$ to combine the two regions:

$$(g, p) = (g', p')\hat{c}(g'', p'')$$

Observe that $c_{i+1} = g_{[0,i]}$ if $c_{in} = 0$. A carry-in to the adder can be considered a carry generated by stage -1, so we set $p_{-1} = 0$, $g_{-1} = c_{in}$ and compute $g_{[-1,i]}$ for all i.

The problem is,

given:	(g_0, p_0)	(g_1, p_1)	$\cdots$	(g_{k-2}, p_{k-2})	(g_{k-1}, p_{k-1})
Find:	$(g_{[0,0]}, p_{[0,0]})$	$(g_{[0,1]}, p_{[0,1]})$	$\cdots$	$(g_{[0,k-2]}, p_{[0,k-2]})$	$(g_{[0,k-1]}, p_{[0,k-1]})$

The desired signal pairs can be found by evaluating all the prefixes of:

$$(g_0, p_0)\hat{c}(g_1, p_1)\hat{c}\cdots\hat{c}(g_{k-2}, p_{k-2})\hat{c}(g_{k-1}, p_{k-1})$$

The carry operator $\hat{c}$ is associative, meaning the order of evaluation does not affect the result:

$$[(g', p')\hat{c}(g'', p'')]\hat{c}(g''', p''') = (g', p')\hat{c}[(g'', p'')\hat{c}(g''', p''')]$$

The operator $\hat{c}$ is not commutative, meaning, in general,

$$(g', p')\hat{c}(g'', p'') \neq (g'', p'')\hat{c}(g', p')$$

Although we can change the order of evaluation of the carry signals, we cannot rearrange the left-to-right positioning of the inputs to the $\hat{c}$ functional units. All of the networks discussed for prefix sums preserve this order, and so all of them are suitable for carry generation.

4.4.1 Summary of Results for all Adders Discussed

Using Nand gates, the time to generate the p and g signals is $2D$ and the $\hat{c}$ function also has delay $2D$. The final sums can be generated in time $3D$. The $\hat{c}$ function requires 7 inputs. With these assumptions in mind, the following table summarizes the adder times and transistor counts for all adders discussed. The transistor counts are presented as the number of transistors for p, g and final sum generation, plus the those for the carry network.

	$k = 32$		$k = 64$		$k = 128$	
Type	Delay	Transistors	Delay	Transistors	Delay	Transistors
Ripple Carry	33D	1152	129D	2304	257	4608
CLA-4 Based	15D	1088+930	15D	2176+1890	19D	4352+3810
First Prefix Scheme	10D	1088+1120	12D	2176+2688	14D	4352+6272
Second Prefix Scheme	16D	1088+798	20D	2176+1680	24D	4352+3458
Brent Kung	16D	1088+798	20D	2176+1680	24D	4352+3458
Kogge Stone	10D	1088+1806	12D	2176+4494	14D	4352+10766

The Kogge-Stone 64 bit adder has 2.9 times the number of transistors needed in the ripple-carry design, but the speed advantage is a factor of 11.7. In comparison, the CLA-4 based 64 bit adder requires 61% fewer transistors that the Kogge-Stone version, but is 36% slower.

The values in the table are strictly for Nand implememtations. The delay in calculating p, g, and final sum signals has been assumed to be 5D. Note that this is a large fraction of the total delay for the fastest adders and it is very likely that clever CMOS circuit designs could substantially reduce this value. A much more careful analysis would also take into account the fan-in and fan-out of the gates throughout these networks, and would account for the lengths and types of connections in the VLSI layout. Such analysis is beyond the scope of this course.

In some modern applications low power consumption is also important. Very careful attention must then be paid to reducing the number of transistors and equalizing signal paths so as to prevent unnecessary transitions of gate outputs. We shall study these techniques in a separate module.