Carry Lookahead Adders

A Ripple Carry Adder (RCA) is a very area-efficient adder design. Unfortunately, it is also slow. The maximum delay of an RCA is from the carry-in input (or the vector inputs at position 0 if there is not a carry-in input) to the carry out, passing through each full adder along the way.

As with many design problems in digital logic, we can make tradeoffs between area and performance (delay). In the case of adders, we can create faster (but larger) designs than the RCA. The Carry Lookahead Adder (CLA) is one of these designs (there are others too, but we will only look at the CLA). We separate the carry chain (the logic that propagates the carry through the full adders of the RCA) from the sum logic (the parts of the full adders that produce the sum). Figure 1shows two different organizations of the same full adder.

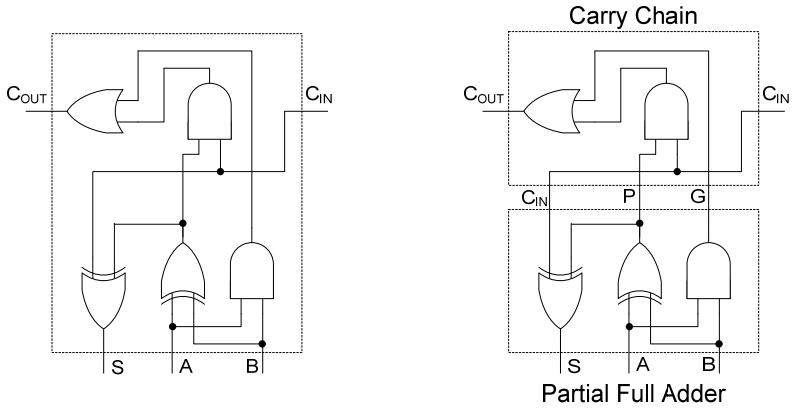

Figure 1: A Full adder (left) and a full adder with the carry logic separated from the sum logic (right). The logic in the bottom half of the right figure is called a "Partial Full Adder" (PFA). The connections between the PFA and the Carry Chain (apart from the carry-in) are labeled with new letters, "G" and "P" that we will explain later.

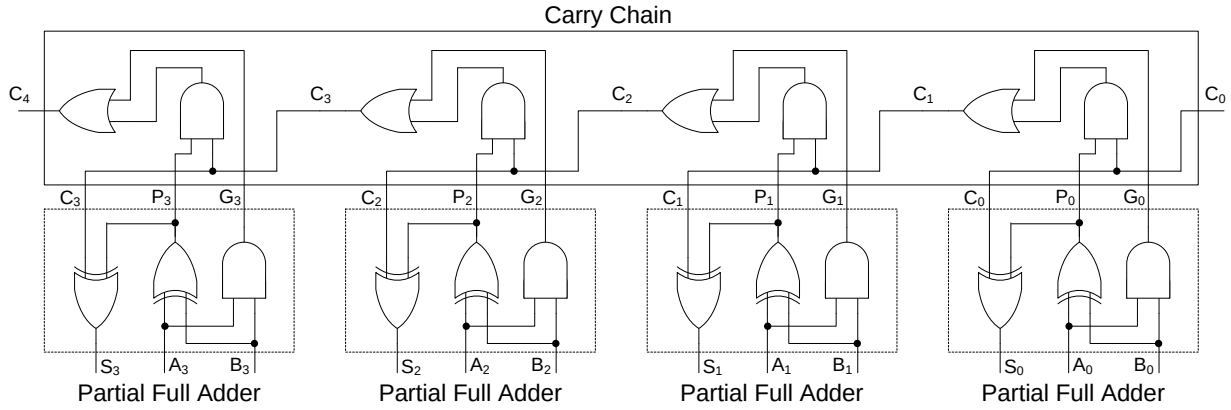

Figure 2: Reforming a Ripple Carry Adder (RCA) formed from Partial Full Adders (PFAs) plus a ripple carry chain

Now, we can redraw a RCA, replacing normal Full Adders with the Partial Full Adders plus the carry logic from Figure 1 right. The result is shown in Figure 2. For now, don't worry about why the PFA outputs are labeled G and P. The carry chain logic is "multi-level". Optimized multi-level logic generally results in a smaller but slower circuit than an optimized two level implementation. For a Carry Lookahead Adder (CLA), we convert the multi-level carry chain into a two-level carry chain, as shown in Figure 3 left. This figure illustrates only the changed carry chain logic. Partial Full Adders are still needed, just like in Figure 2—only the carry chain has been replaced. A complete 4-bit CLA is shown in Figure 3 right. The box labeled "Look Ahead" contains the logic of Figure 3 left. The figure uses the P and G signals output by the PFAs – we will explain these signals in more depth later.

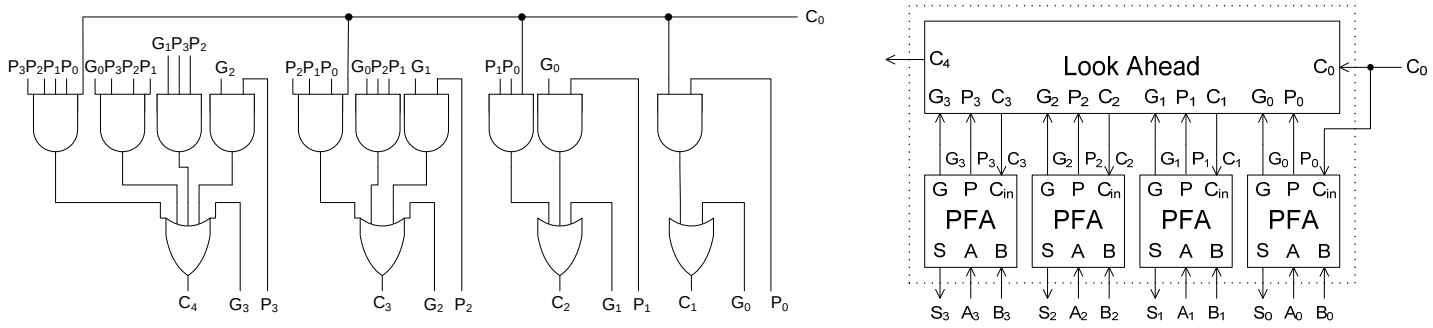

Figure 3: Left: A 4-bit Carry Lookahead carry chain block. Right: The 4-bit Carry Lookahead carry chain block reconnected to the partial full adders to form the complete 4-bit Carry Lookahead adder.

Although converting the multi-level RCA carry chain to two-level form explains the design of Figure 3, we can also explain further what is happening by considering the actual underlying mathematics of binary addition. This is also where we will see the reasons for why PFA outputs are labeled G and P. Basically, what a Carry Lookahead carry chain does is to enumerate all the cases where the carry-in to each position will equal 1. The circuit does this **before** actually adding values at each bit position (hence "look ahead").

So, think about the operation of a full adder in an RCA, which adds A_N, B_N, and a carry-in C_N to produce the sum S_N and the carry-out C_{N+1} for position N. For this addition, we have several possibilities:

- None of the inputs are 1 (all are zero) → 0 + 0 + 0 = 00 → $C_{N+1} = 0$, $S_N = 0$
- Exactly one of the inputs is a 1 (any of them) → 1 + 0 + 0 = 01 → $C_{N+1} = 0$, $S_N = 1$
- Exactly two of the inputs are 1s (any two) → 1 + 1 + 0 = 10 → $C_{N+1} = 1$, $S_N = 0$
- All three inputs are 1s → 1 + 1 + 1 = 11 → $C_{N+1} = 1$, $S_N = 1$

Now let's think about this in terms of knowing A_N and B_N (values of A and B at position N), but not (yet) knowing C_N. What can we determine based solely on the math? Remember that a value XORed with 0 is unchanged, and a value XORed with a 1 is inverted.

- Neither A_N nor B_N are 1 (both are zero) → $0 + 0 + C_N$ → $C_{N+1} = 0$ $S_N = C_N$
- One of A_N or B_N are 1 (the other is zero) → $1 + 0 + C_N$ → $C_{N+1} = C_N$ $S_N = \overline{C_N}$
- Both A_N and B_N are 1 → $1 + 1 + C_N$ → $C_{N+1} = 1$ $S_N = C_N$

From the above, we can see that position N will **Generate** a carry-out of 1 if both A_N and B_N are 1—regardless of the values of A and B at lower bit positions, or the value of C0 (the carry-in to the adder). We don't have to do the math at the lower positions to know this if A_N and B_N are both 1. Likewise, we can see that if A_N and B_N are different (one of them is 1 and the other is 0), that the carry-out from position N (C_{N+1})will be equal to the carry-in to position N (C_N). In other words, position N will **Propagate** the carry. A two-input XOR gate will output a 1 if the two inputs are different, and a 0 if they're the same. So we can calculate the "generate" value for position N as $G_N = A_N$ AND B_N, and we can calculate the "propagate" value[1] for position N as $P_N = A_N$ XOR B_N. These are calculations we need to perform in a full adder anyway, as we can see in Figure 1.

So now we can re-examine our earlier analysis, and present it in terms of Generates and Propagates. For this table we assume that we use $P_N = A_N$ XOR B_N instead of $P_N = A_N$ OR B_N (see footnote).

[1] Technically an OR gate can be used instead of an XOR to compute P_N, but we need to XOR A_N and B_N regardless as part of the S_N sum calculation. So using an OR gate doesn't actually simplify the circuit. We don't "lose" information by using an XOR instead of an OR—if both A_N and B_N are 1 (the only case that differs between OR and XOR), then G_N will be a 1, and handled by the equation.

- Neither A_N nor B_N are 1 (both are zero) → $G_N = 0$ $P_N = 0$
- One of A_N or B_N are 1 (the other is zero) → $G_N = 0$ $P_N = 1$
- Both A_N and B_N are 1 → $G_N = 1$ $P_N = 0$

As stated above, the carry-lookahead adder's carry calculation enumerates all of the possible ways in which the carry-in to a particular position could be a 1. This is similar to how we build Boolean functions—by enumerating the input combinations that cause the output to be a 1.

We can calculate the carry-out from position 0 as $C_1 = G_0 + C_0P_0$. This equation says that we will have a carry-out of 1 from position 0 if either we **G**enerate a carry from position zero, OR if we both have a **C**arry-in of 1 AND we **P**ropagate the carry through position 0. Notice how the previous sentence matches to the equation—this is very important.

Now consider the carry-out of position 1. This value will be 1 if position 1 **G**enerates a carry, OR position 0 **G**enerates a carry and position 1 **P**ropagates the carry, OR if there is a **C**arry-in of 1 AND it is **P**ropagated through position 0 AND it is **P**ropagated through position 1. The equation is thus $C_2 = G_1 + G_0P_1 + C_0P_0P_1$. Again, notice how we are just enumerating all of the ways that the carry-out for a position might be a 1, and that the equation matches the verbal sentence that describes this. Using similar techniques, we can also compute C_3 and C_4. Figure 3 left shows the logic for computing the carry values for a carry-lookahead block. The right part of the figure shows how the lookahead carry logic is recombined with the PFAs to create the complete 4-bit carry-lookahead adder. Remember that we still need to add the C_N value to the sum of A_N and B_N. But the PFA does not compute C_{N+1}, it just computes G_N = A_N AND B_N, P_N = A_N XOR B_N, and S_N = (A_N XOR B_N) XOR C_N = P_N XOR C_N. (We can reuse the P_N XOR gate in the sum calculation).

Remember, the key to understanding CLAs and being able to determine the carry equations is to remember what the equations *mean*, instead of trying to memorize the equations themselves. If you remember how CLAs work, and what the equations mean, you can easily write out the equation for any C value, given enough paper and pencils. For example:

$$C_7 = G_6 + G_5P_6 + G_4P_5P_6 + G_3P_4P_5P_6 + G_2P_3P_4P_5P_6 + G_1P_2P_3P_4P_5P_6 + G_0P_1P_2P_3P_4P_5P_6 + C_0P_0P_1P_2P_3P_4P_5P_6$$

No one would want to memorize that (not even your instructor), but fortunately we can easily figure it out by thinking about it in terms of enumerating all of the ways that C_7 might be a 1.

Keep in mind that for a CLA we do <u>not</u> want to "factor out" values to "simplify" the circuit. Although C_2 could be computed and written as $C_2 = G_1 + (G_0 + C_0P_0)P_1$, this version of the equation would indicate a multi-level solution (and is in fact identical to the ripple carry calculation of C_2 as shown in Figure 2). This is an example of how the form of logic equations has implications on the resulting hardware.

Notice the similarities between the carry calculations for different bit positions. The equations are simpler for carry calculations closer to the least-significant bit. Higher bit positions have more complex calculations, because there are more *different* ways that the carry-out could be a 1. However, within a given carry calculation, The terms in the carry calculation are smaller for "closer" bit positions (i.e., when the G index is high), and larger for "further" bit positions closer to the least significant bit (i.e., when the G index is low). Generated carries from further away have to be propagated through more positions to reach the current one. Finally, the number of terms and size of the largest term grow at the same rate. For C_7, the OR gate has 8 inputs, and the largest AND gate has 8 inputs. Each carry calculation has one fewer AND gate than the number of inputs to the OR gate—the Generate value for the position immediately to the right goes directly into the OR gate.

Hierarchical Carry Lookahead Adders

Unfortunately, creating a 64-bit adder using this method would be prohibitively large—notice how the number of terms increases as we calculate the carry-in for higher bit positions, and how the last term in the equation also grows. Calculating C_{64} directly would require a 65-input OR gate and AND gates ranging from 2-inputs all the way to 65 inputs (one of each size). Frequently, designers limit CLA blocks to 4 bits in size, and build larger adders by combining these blocks hierarchically.

To do this, we add two outputs to our lookahead block that computes our carry values: a "group Generate" and a "group Propagate". For a 4-bit CLA block we can label these as G_{0-3} and P_{0-3}, respectively. These two values are still a single bit each – they are not a vector! The values instead indicate if the entire lookahead block will **Generate** a carry-out of 1, or will **Propagate** the carry-in to the carry-out. Again, we can figure out the equations for the group Generate and the group Propagate by thinking about how binary addition works. The group of 4 bit positions will Generate a carry if position 3 **G**enerates a carry, OR if position 2 **G**enerates a carry AND position 3 **P**ropagates it, or if position 1 **G**enerates a carry and position 2 **P**ropagates it AND position 3 **P**ropagates it, OR if position 0 **G**enerates a carry AND position 1 **P**ropagates it AND position 2 **P**ropagates it AND position 3 **P**ropagates it. The equation for the group generate is therefore $G_{0-3} = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3$. The group of four bits will Propagate a carry if the carry can propagate all the way from position 0 through position 3. So $P_{0-3} = P_0P_1P_2P_3$.

Fortunately, we already have logic that uses Generates and Propagates to compute carry values—our original lookahead block. We can use the same block to compute the carry-ins at a higher level of hierarchy. The lookahead block doesn't care if the Generates and Propagates that it uses represent a single bit or many bits. The meaning of the equations is the same.

Figure 4 shows a 16-bit CLA constructed from four 4-bit CLAs plus an additional lookahead block. The lookahead blocks compute the carry bits based on the Generate and Propagate values they receive. The Partial Full Adders (PFA) produce the sum values based on the A, B, and C value for each bit position. The dotted lines represent the 4-bit CLAs, with the internal circuitry also shown. All five lookahead blocks are identical except the topmost does not compute the group Generate/Propagate signals—this is the only difference between them. Figure 5 shows the same structure, but with the 4-bit CLA internals hidden.

Note that the 16-bit CLA is constructed using the same ideas and techniques we used for the 4-bit CLA. In the 4-bit CLA, our "block" is a single PFA. In a 16-bit CLA, our "block" is a complete 4-bit CLA. In the 4-bit CLA, we have four PFAs plus one lookahead block. In a 16-bit CLA, we have four 4-bit CLAs plus one lookahead block. Figure 5 therefore looks similar to the contents of a dotted box in Figure 4. We can create larger CLAs by adding one or more levels of hierarchy. We create a 4-bit CLA from four PFAs (could consider these as "1-bit CLAs") and one 4-bit lookahead block. We can create a 16-bit CLA from four 4-bit CLAs plus one 4-bit lookahead block. We can create a 64-bit CLA from four 16-bit CLAs plus one 4-bit lookahead block. We can create a 256-bit CLA from four 64-bit CLAs plus one 4-bit lookahead block. The pattern is the same.

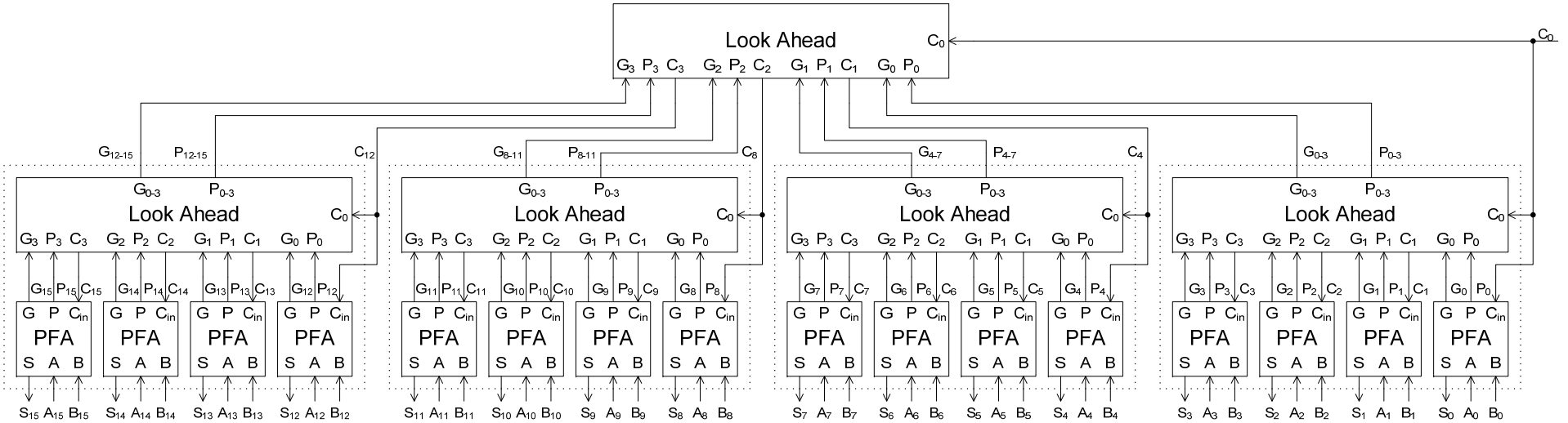

Figure 4: A 16-bit CLA built using hierarchy. 4-bit CLAs are shown inside dotted boxes.

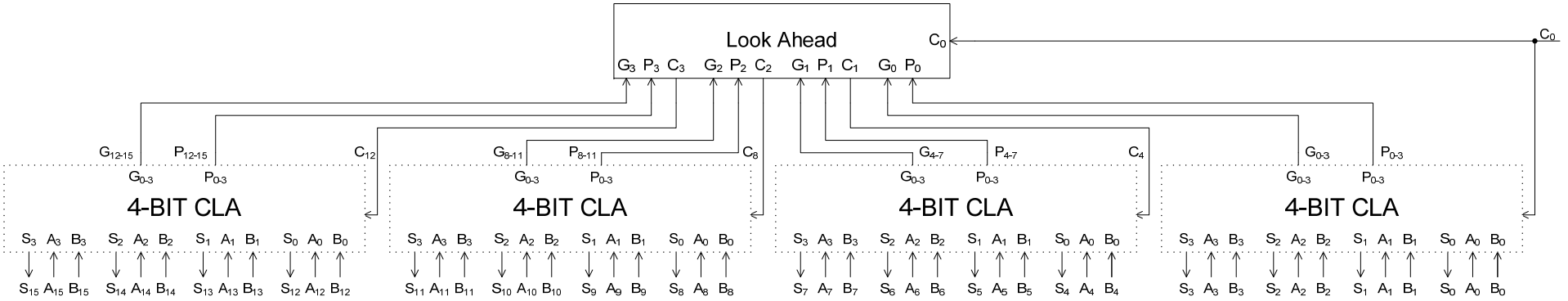

Figure 5: The same 16-bit CLA as Figure 4, but with the internals of the 4-bit CLAs hidden.