

## Charming Adders

### Implementing an Adder in an FPGA

#### LEARNING THE ROPES

Ingo Cyliax

Designing with FPGAs is similar to working with other digital components, and opens the door to interesting design permutations. This month, Ingo presents how structures are implemented in FPGA logic blocks.



Designing with FPGAs is a lot like designing with other digital components. You have some idea of what functions you would like to implement in your design, and the FPGA provides the basic building blocks to implement these. As you recall, FPGAs are made up of an array of logic blocks wired together with routing resources. Except for some functions, the routing facilities are only used to connect the blocks. This month, I'm going to show you some examples of how familiar structures are implemented in FPGA logic blocks.

One of the most common digital functional blocks is the adder, which is used in counters and arithmetic functions. The adder is a structure that is difficult to implement efficiently. The truth table for a full adder, an adder that has a carry input and carry output, is shown in Table 1. You'll need the  $C_{IN}$  and  $C_{OUT}$  signals to cascade adders to implement adders for larger word sizes.

A simple 1-bit full adder can be constructed with the following functions:

$$\begin{aligned} \text{Sum} &= A \text{ XOR } B \text{ XOR } C_{IN} \\ C_{OUT} &= (A \times B) + (A \times C_{IN}) + (B \times C_{IN}) \end{aligned}$$

To build a 16-bit adder, you simply chain 16 adders together (see Figure 1) by wiring the  $C_{OUT}$  to the  $C_{IN}$  of the next stage (see Figure 2). This particular design also has input and output registers to hold values.

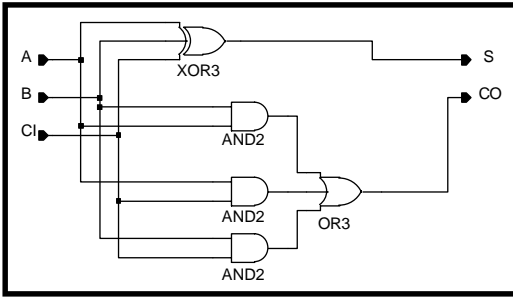
#### IMPLEMENTING

What happens when you implement this in an FPGA? Each bit is implemented as two functions—one that computes the sum and one that computes the carry output signal. Although it was convenient to draw my adders using logic gates, as shown in Figure 1, FPGAs typically implement logic blocks as small look-up tables, like the one shown in Table 1. I started my design with a truth table, but I actually entered the design using logic gates and the FPGA's mapper and then figured out the look-up table based on what functions the logic gates implemented.

When I implemented this circuit in an XC4010E FPGA, I ended up with 30 CLBs, and the timing estimator showed that this adder will run at 25.75 MHz for the particular speed grade I chose. At first it might seem straightforward to figure out how many CLBs are needed to implement,

Inputs			Outputs	
A	B	C <sup>IN</sup>	Sum	C <sup>OUT</sup>
0	0	0 <sup>IN</sup>	0	0 <sup>OUT</sup>
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

**Table 1**—Here is the truth table for a full adder. A full adder is an adder that takes the two operands (A and B) and a carry input, and produces both the sum and a carry output.



**Figure 1**—The logic to implement a native full adder. It takes the carry input, adds it with the two operands (A and B), and generates the output's sum and carry out.

but it's not.

There are two 4:1 look-up tables in each CLB (G and F). In addition, there is a small LUT (H), and the output of G and F LUTs that can perform functions on the outputs using one external signal input. Each CLB has two registers and two non-registered outputs. Muxes select the source for each of the registers (F, G, H, or an external input signal) and the source (F or G) for the non-registered output signal. Figure 3 shows the CLB I used.

The output registers (*Sum[15:0]*) get absorbed in the CLBs that implement the adder logic, meaning that the CLBs in the LUTs implement the sum of the two inputs plus the carry from the previous stage. Knowing this, my initial estimate would have put the CLB at:

- 2 × 16-bit input registers
- 16 CLBs
- 16 × 1-bit adders plus registers
- 16 CLBs
- Total
- 32 CLBs

Somewhat the software is able to optimize out two more CLBs. To find out what it did, look at the output net list from the place-and-route phase in the tool-chain. The net list is saved in an *.ncd* file, which is a binary database. There is a tool called EPIC that you can use to visualize what the physical representation of the chip is and edit it. However, I decided to look at the net list directly, using the tool *ncdread* to read the list and write it out in text form. Listing 1 shows a couple of sections from this list.

The sections that have pin numbers (P46, P5, etc.) for their site infor-

mation are I/O blocks (IOBs), while sites that use row/column labels (CLB\_R20C11, CLB\_R1C11, etc.) are CLBs. They are connected using the networks listed by each input pin name. By going through the net list, I deduced a couple of optimizations that the software was able to perform.

It turns out that because the  $C_{IN}$  to the 16-bit adder is zero and the  $C_{OUT}$  from the adder is not used, these can be optimized out. This saves a couple of LUTs and

makes up about one CLB. Because CLBs can output both non-register and register versions of the LUT outputs, some of the lower order CLBs can be used to implement the sum and  $C_{OUT}$  directly from a single LUT. The carry from the previous stage is then implemented in the H function generator by inverting the inputs for the registered input.

Essentially, the mapping software was able to pack and merge functions into LUTs, limited by the number of input and output signals that are actually used.

#### Listing 1

```

NC_COMP:0 - <A<0>> site = P46
  Config String: <OUTMUX:#OFF OMUX:#OFF OKMUX:#OFF TRI:#OFF
  IMUX:#OFF IKMUX:#OFF ISR:#OFF I2MUX:I I1MUX:#OFF PULL:#OFF
  SLEW:#OFF OSR:#OFF OCE:#OFF ICE:#OFF>
  7 pins -
  pin 5 - I2: <&_A_16>
NC_COMP:1 - <A<10>> site = P5
  Config String: <OUTMUX:#OFF OMUX:#OFF OKMUX:#OFF TRI:#OFF
  IMUX:#OFF IKMUX:#OFF ISR:#OFF I2MUX:#OFF I1MUX:I PULL:#OFF
  SLEW:#OFF OSR:#OFF OCE:#OFF ICE:#OFF>
  7 pins -
  pin 4 - I1: <&_A_6>
...
NC_COMP:50 - <S<9>> site = P67
  Config String: <OUTMUX:0 OMUX:0 OKMUX:#OFF TRI:#OFF IMUX:#OFF
  IKMUX:#OFF ISR:#OFF I2MUX:#OFF I1MUX:#OFF PULL:#OFF SLEW:SLOW
  OSR:#OFF OCE:#OFF ICE:#OFF>
  7 pins -
  pin 0 - 0: <&_A_39>
NC_COMP:51 - <U11/QA0> site = CLB_R20C11
  Config String: <CLKX:CLK ECX:EC CLKY:CLK DY:DIN XMUX:#OFF
  YMUX:#OFF G3MUX:#OFF G2MUX:#OFF F4MUX:#OFF CARRY:#OFF XQMUX:QX
  YQMUX:QY ECY:EC DX:H H1:C1 DIN:C2 SR:C3 EC:C4 FCARRY:#OFF
  H:#LUT:H=H1 H0:#OFF H2:#OFF RAMCLK:#OFF RAM:#OFF GCARRY:#OFF
  G:#OFF: F:#OFF: CINMUX:#OFF SETX:SR SETY:SR SRX:RESET SRY:RESET>
  19 pins -
  pin 0 - C1: <&_A_15>
  pin 1 - C2: <&_A_16>
  pin 2 - C3: <Net00006>
  pin 3 - C4: <Net00003>
  pin 13 - K: <Net00007>
  pin 16 - XQ: <U11/QA1>
  pin 18 - YQ: <U11/QA0>
NC_COMP:52 - <U11/QA10> site = CLB_R1C11
  Config String: <CLKX:CLK ECX:EC CLKY:CLK DY:H XMUX:#OFF YMUX:#OFF
  G3MUX:#OFF G2MUX:#OFF F4MUX:#OFF CARRY:#OFF XQMUX:QX YQMUX:QY
  ECY:EC DX:DIN H1:C1 DIN:C4 SR:C3 EC:C2 FCARRY:#OFF H:#LUT:H=H1
  H0:#OFF H2:#OFF RAMCLK:#OFF RAM:#OFF GCARRY:#OFF G:#OFF: F:#OFF:
  CINMUX:#OFF SETX:SR SETY:SR SRX:RESET SRY:RESET>
  19 pins -
  pin 0 - C1: <&_A_5>
  pin 1 - C2: <Net00003>
  pin 2 - C3: <Net00006>
  pin 3 - C4: <&_A_6>
  pin 13 - K: <Net00007>
  pin 16 - XQ: <U11/QA10>
  pin 18 - YQ: <U11/QA11>

```

## ADDER ARCHITECTURE

The first adder I designed is called a ripple adder. The carry ripples down each adder stage until it reaches the last stage. It's a simple adder but not efficient, especially if the word is long.

There are other adder architectures you can design with. One of these is the carry look-ahead adder. In a nutshell, this adder tries to compute the carries that are needed in later stages. However, this introduces more logic than a pure-ripple adder. A compromise is to build small-to-medium size look-ahead adder blocks and use ripple carry between the stages.

I'm going to use the following notation:  $C_i$  is the carry input for the  $i$ -th stage,  $A_i$  and  $B_i$  are the two inputs, and  $S_i$  is the sum. The first stage is easy:

$$S_0 = A_0 \text{ XOR } B_0 \text{ XOR } C_0 \\ C1 = (A_0 \times C_0) + (B_0 \times C_0) + (A_0 \times B_0)$$

The second stage is:

$$S1 = A1 \text{ XOR } B1 \text{ XOR } C1 \\ C2 = C1 \times (A1 + B1) + (A1 \times B1)$$

Collect the  $C1$  term and substitute:

$$S1 = A1 \text{ XOR } B1 \text{ XOR } ((A_0 \times C_0) + (B_0 \times C_0) + (A_0 \times B_0)) \\ C2 = ((A_0 \times C_0) + (B_0 \times C_0) + (A_0 \times B_0)) \times (A1 + B1) + (A1 \times B1)$$

Well, you get the idea. This gets messy fast. Figure 4 shows a 4-bit carry look-ahead adder implementation. It's easy to get lost. I made two or three mistakes before getting it right. To build a 16-bit adder, I chained four of the 4-bit adders together, as seen in Figure 5.

Implementing a 16-bit carry look-ahead in the same part, I get 36 CLBs, and the design runs at 26.2 MHz. As expected, it's bigger but only a little faster. It turns out that, for 4-bit carry look-ahead adders, much of the logic gets absorbed and packed in four input LUTs similar to the ripple carry adder implementation and, therefore, doesn't buy you much over a ripple adder. The effect would have been greater if I built an 8-bit carry look-ahead section instead of the 4-bit sections. However, an 8-bit carry

look-ahead adder is huge and difficult to enter by hand without making some mistakes.

## THE SERIAL ADDER

As I have pointed out in earlier articles, designing with FPGAs opens many interesting design permutations. Let's look at what it would take to implement a serial adder (see Figure 6). FPGAs have fast register architectures, and all the logic required fits into a single CLB. There is a register that stores the carry, so instead of chaining the carry from stage to stage, the carry stays in one place and data is shifted through each stage.

When I implemented this design, I got 25 CLBs that run at 98 MHz. The CLBs contain the shift registers for the input and output, but only one CLB actually contains adder logic. This is the one that is associated with the carry register. At 98 MHz, a 16-bit adder will effectively run at about 6 MHz, however, it's small. If the data is already in serial form (LSB first), the input shift registers can be eliminated, and the adder will implement in 2 CLBs and run at over 200 MHz.

It took me the better part of two days to implement and test these adders from scratch, and the results haven't been spectacular. Maybe I should have used the 16-bit adder in the library.

The design implements in 25 CLBs, running at 78.5 MHz! How's this possible? The adder circuit in the library uses fast carry chains that are a feature in the architecture. Each adjacent CLB has a special connection with logic to generate carry for the next stage and use the carry from the previous stage (see Figure 7). Clearly, this is the way to go. Each adder bit in the schematic has a special configuration block symbol that is used to configure the CLB, telling it where to get its inputs from and how to route the output. Also, special blocks are used to place the CLBs. They will be placed into columns, so the carry chain can be used.

## LESSONS LEARNED

There are a couple lessons that can be learned from this adder example.

First, it's difficult to predict how the mapper and place-and-route tool will implement your design. In many cases, it will merge and pack logic if it can fit them into a single LUT/CLB structure, sometimes giving you surprising results. The few times when it gets in the way, you should use floor planning to force the implementation tool to leave the placement of your design the way you intended it. You can tell it which logic you want to keep in a LUT and where on the chips you want the CLB to be located—either in absolute or relative coordinates.

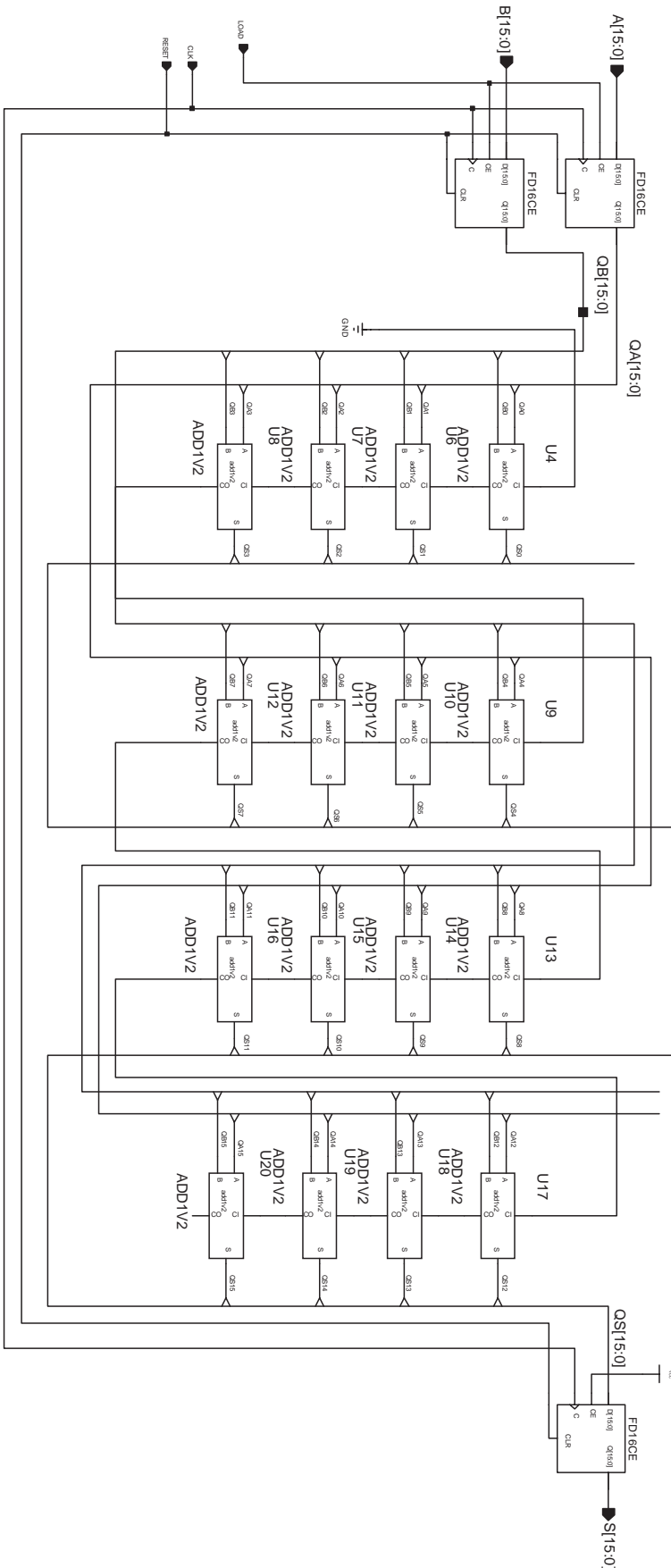
Secondly, many times, library components implement the design components well. In most cases, the library components are thoroughly debugged and optimized by engineers that understand the underlying FPGA architecture. Unless you have a custom requirement, you'll end up spending a lot of time, and usually your component won't be any better (in time and space resources).

Of course, your mileage may vary. It may turn out that you have a customized functionality for which no component exists. Or, you may figure out how to optimize a structure better, based on your domain knowledge of the problem you're trying to solve. For example, vendor libraries are not likely to have components that will help you optimize neural network designs or async (self-timed) circuits. In those cases, you're on your own. ☒

*Ingo Cyliax has written for Circuit Cellar on topics such as embedded systems, FPGA design, and robotics. He is a research engineer at Derivation Systems Inc., a San Diego-based formal synthesis company, where he works on formal-method design tools for high-assurance systems and develops embedded-system products. You may reach him at [cyliax@derivation.com](mailto:cyliax@derivation.com).*

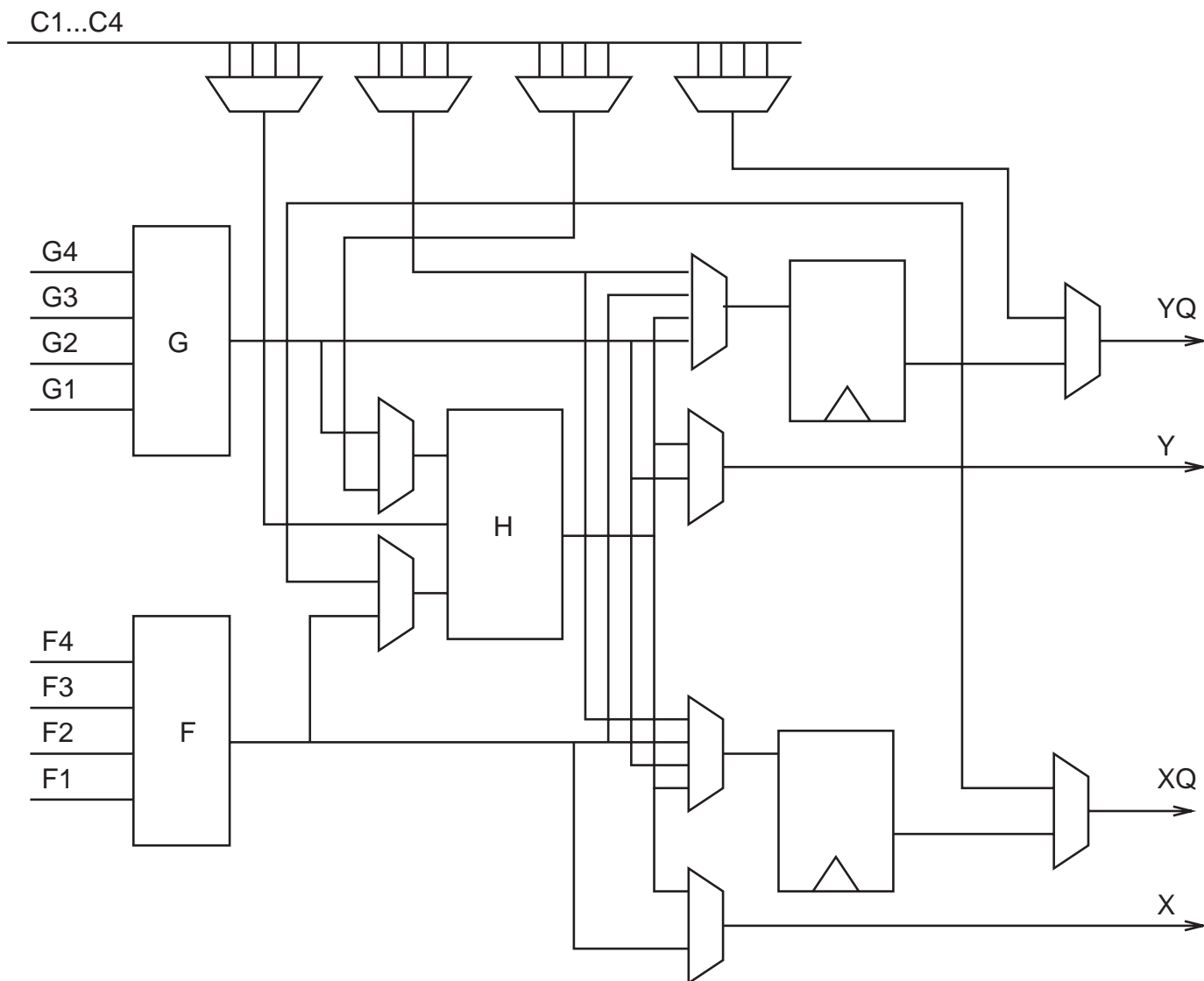
Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, [subscribe@circuitchellar.com](mailto:subscribe@circuitchellar.com) or [www.circuitchellar.com/subscribe.htm](http://www.circuitchellar.com/subscribe.htm).

Figure 2—To build a 16-bit adder, all you need to do is chain all 16 single-bit adders together. This is called a ripple carry adder because the carry ripples through all the stages.



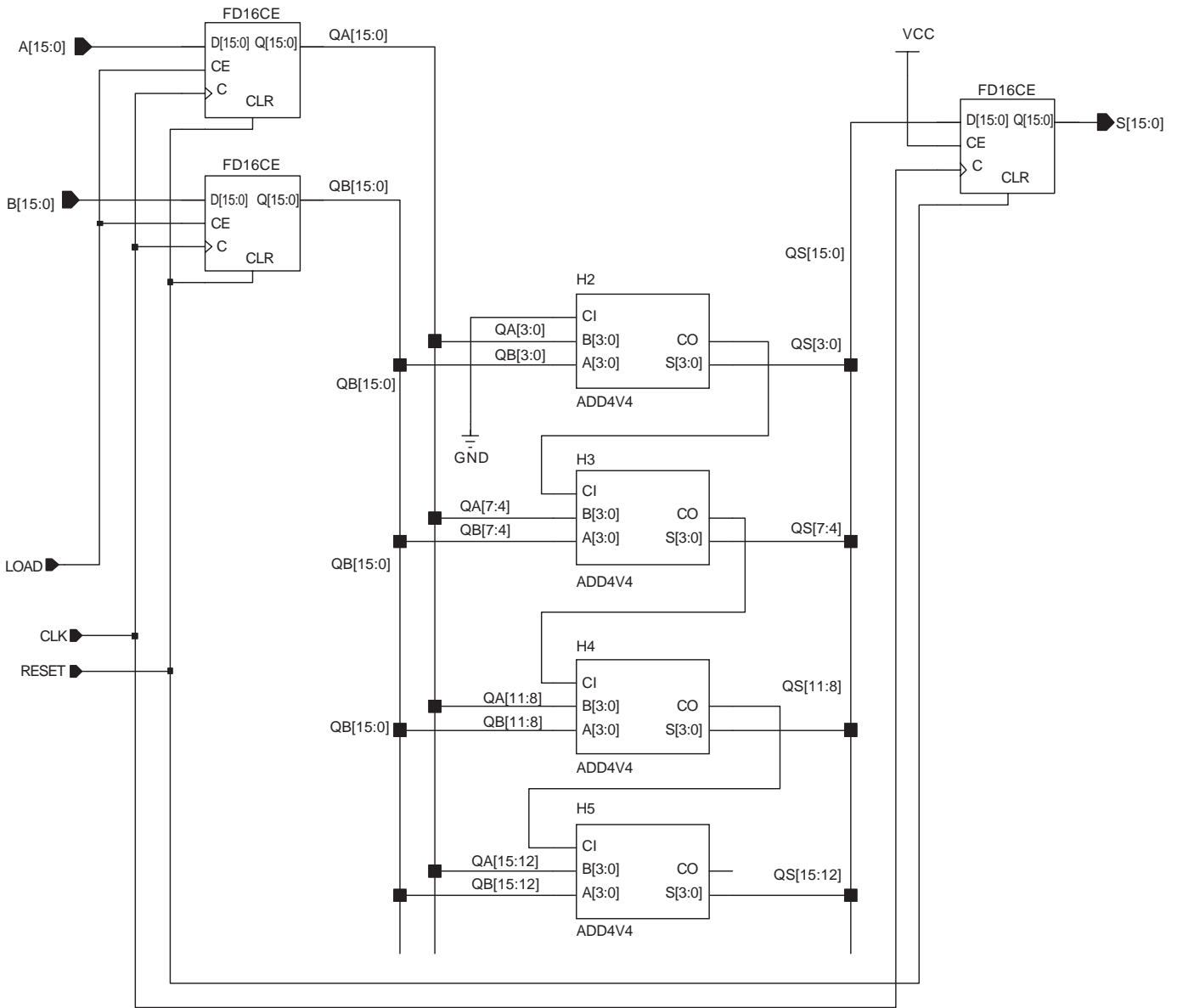
Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, subscribe@circuitchellar.com or www.circuitchellar.com/subscribe.htm.

Figure 3—The configurable logic block for a xc4000 FPGA looks something like this. There are three look-up table (LUT) function generators (F, G, and H), as well as two registers (XQ and YQ).



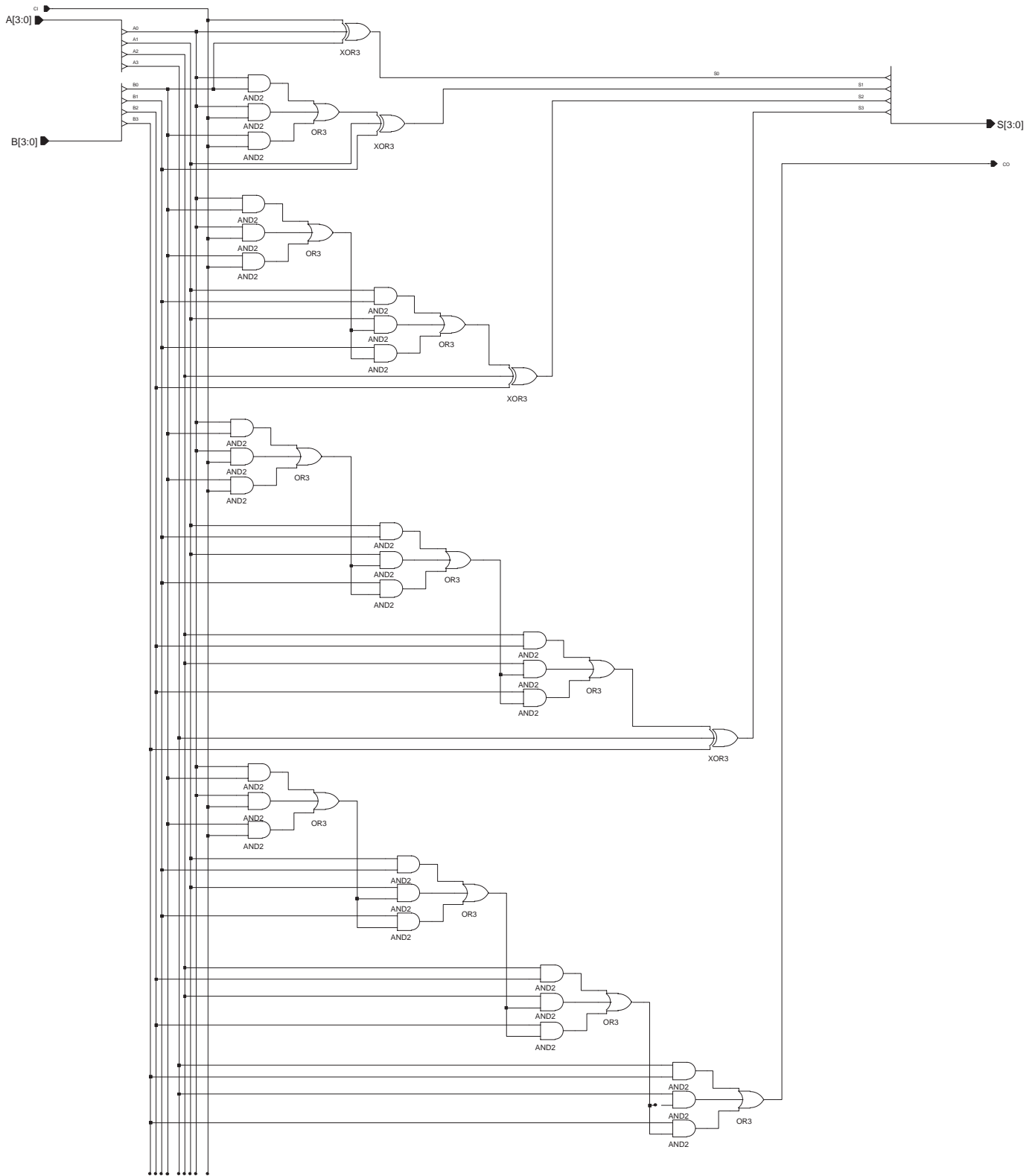
Circuit Cellar, the Magazine for Computer Applications.  
Reprinted by permission. For subscription information,  
call (860) 875-2199, [subscribe@circuitcellar.com](mailto:subscribe@circuitcellar.com) or  
[www.circuitcellar.com/subscribe.htm](http://www.circuitcellar.com/subscribe.htm).

**Figure 5**—A 16-bit adder built up from 4-bit carry look-ahead adder cells. It's a compromise to use small carry look-ahead adders and then wire them up as a ripple carry between the stages.



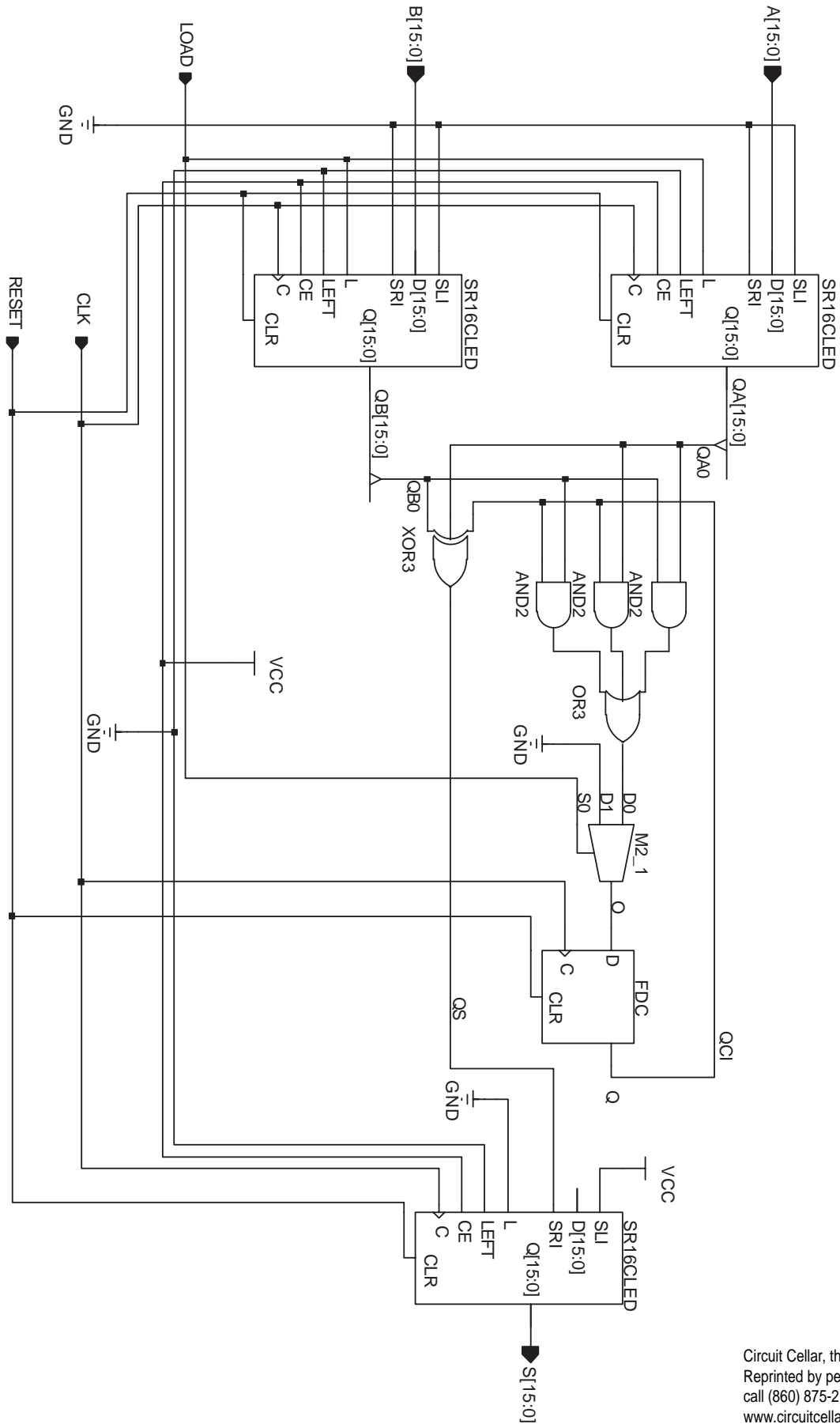
Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, [subscribe@circuitcellar.com](mailto:subscribe@circuitcellar.com) or [www.circuitcellar.com/subscribe.htm](http://www.circuitcellar.com/subscribe.htm).

**Figure 4**—A 4-bit carry look-ahead adder. Their strategy is to generate the carry at each stage in parallel directly dependent on the inputs.



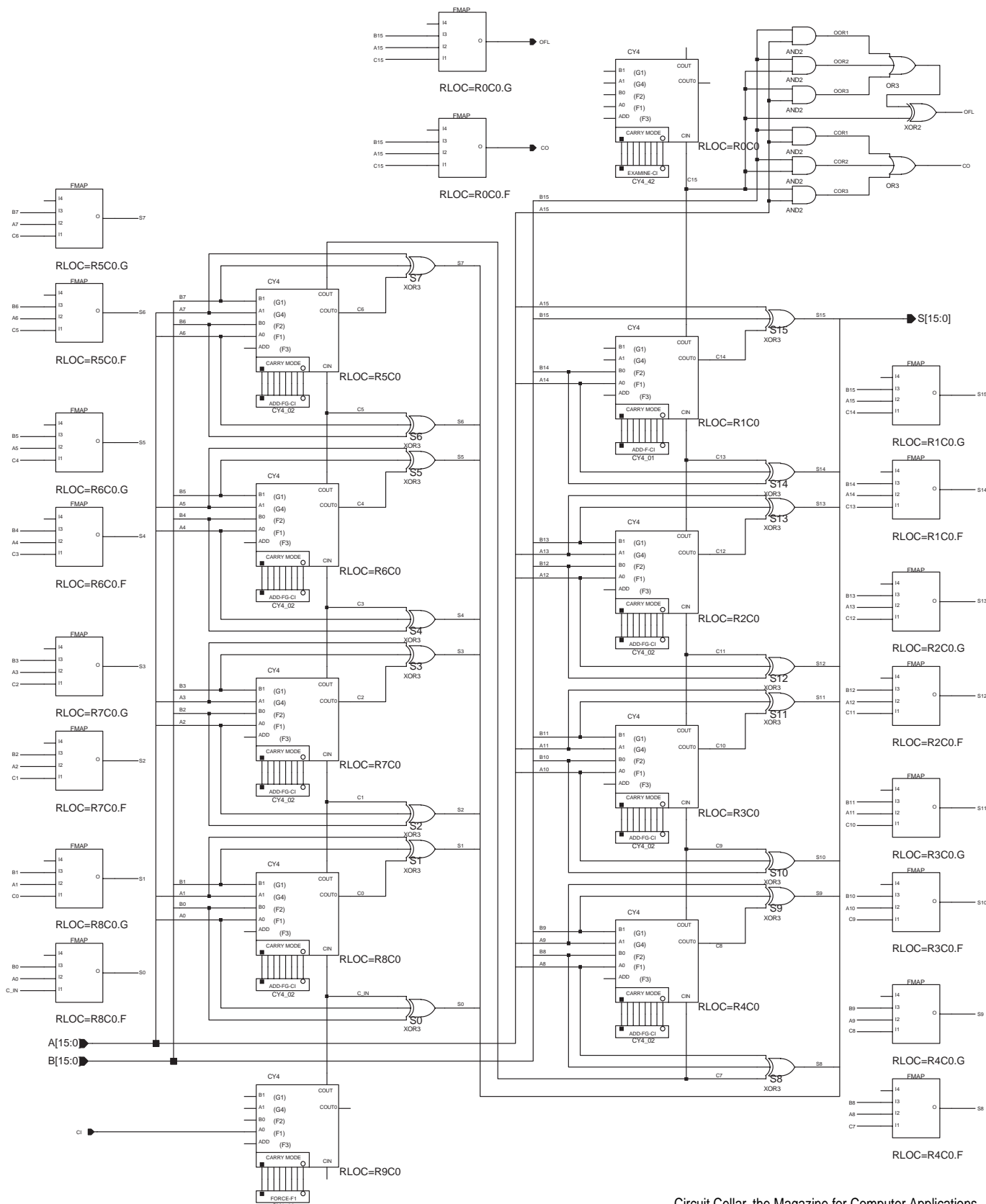
Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, [subscribe@circuitchellar.com](mailto:subscribe@circuitchellar.com) or [www.circuitchellar.com/subscribe.htm](http://www.circuitchellar.com/subscribe.htm).

Figure 6—A single bit serial adder can be constructed efficiently, especially if the operands are already in bit serial format. Single bit adders also run fast.



Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, subscribe@circuitchellar.com or www.circuitchellar.com/subscribe.htm.

**Figure 7**—The implementation of the 16-bit fast adders from the vendor library. These adders use a fast carry chain, which is a feature of the architecture and directives to floorplan the CLBs, so they line up in columns.



Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, subscribe@circuitscellar.com or www.circuitcellar.com/subscribe.htm.