7.6 Faster addition: Carry lookahead

(Original section[1])

The key to speeding up addition is determining the carry in to the high-order bits sooner. There are a variety of schemes to anticipate the carry so that the worst-case scenario is a function of the $\log_2$ of the number of bits in the adder. These anticipatory signals are faster because they go through fewer gates in sequence, but it takes many more gates to anticipate the proper carry.

A key to understanding fast-carry schemes is to remember that, unlike software, hardware executes in parallel whenever inputs change.

Fast carry using "infinite" hardware

As we mentioned earlier, any equation can be represented in two levels of logic. Since the only external inputs are the two operands and the CarryIn to the least significant bit of the adder, in theory we could calculate the CarryIn values to all the remaining bits of the adder in just two levels of logic.

For example, the CarryIn for bit 2 of the adder is exactly the CarryOut of bit 1, so the formula is

$$\mathbf{CarryIn2} = (\mathbf{b1} \cdot \mathbf{CarryIn1}) + (\mathbf{a1} \cdot \mathbf{CarryIn1}) + (\mathbf{a1} \cdot \mathbf{b1})$$

Similarly, CarryIn1 is defined as

$$\mathbf{CarryIn1} = (\mathbf{b0} \cdot \mathbf{CarryIn0}) + (\mathbf{a0} \cdot \mathbf{CarryIn0}) + (\mathbf{a0} \cdot \mathbf{b0})$$

Using the shorter and more traditional abbreviation of *ci* for CarryIn*i*, we can rewrite the formulas as

$$\mathbf{c2} = (\mathbf{b1} \cdot \mathbf{c1}) + (\mathbf{a1} \cdot \mathbf{c1}) + (\mathbf{a1} \cdot \mathbf{b1})$$

$$\mathbf{c1} = (\mathbf{b0} \cdot \mathbf{c0}) + (\mathbf{a0} \cdot \mathbf{c0}) + (\mathbf{a0} \cdot \mathbf{b0})$$

Substituting the definition of c1 for the first equation results in this formula:

$$\begin{aligned}\mathbf{c2} = {} & (\mathbf{a1} \cdot \mathbf{a0} \cdot \mathbf{b0}) + (\mathbf{a1} \cdot \mathbf{a0} \cdot \mathbf{c0}) \cdot (\mathbf{a1} \cdot \mathbf{b0} \cdot \mathbf{c0}) \\ & + (\mathbf{b1} \cdot \mathbf{a0} \cdot \mathbf{b0}) + (\mathbf{b1} \cdot \mathbf{a0} \cdot \mathbf{c0}) + (\mathbf{b1} \cdot \mathbf{b0} \cdot \mathbf{c0}) + (\mathbf{a1} \cdot \mathbf{b1})\end{aligned}$$

You can imagine how the equation expands as we get to higher bits in the adder; it grows rapidly with the number of bits. This complexity is reflected in the cost of the hardware for fast carry, making this simple scheme prohibitively expensive for wide adders.

Fast carry using the first level of abstraction: Propagate and generate

Most fast-carry schemes limit the complexity of the equations to simplify the hardware, while still making substantial speed improvements over ripple carry. One such scheme is a *carry-lookahead adder*. In COD Chapter 1 (Computer Abstractions and Technology), we said computer systems cope with complexity by using levels of abstraction. A carry-lookahead adder relies on levels of abstraction in its implementation.

Let's factor our original equation as a first step:

$$\begin{aligned}\mathbf{c}i + 1 &= (\mathbf{b}i \cdot \mathbf{c}i) + (\mathbf{a}i \cdot \mathbf{c}i) + (\mathbf{a}i \cdot \mathbf{b}i) \\ &= (\mathbf{a}i \cdot \mathbf{b}i) + (\mathbf{a}i + \mathbf{b}i) \cdot \mathbf{c}i\end{aligned}$$

If we were to rewrite the equation for c2 using this formula, we would see some repeated patterns:

$$\mathbf{c2} = (\mathbf{a1} \cdot \mathbf{b1}) + (\mathbf{a1} \cdot \mathbf{b1}) \cdot ((\mathbf{a0} \cdot \mathbf{b0}) + (\mathbf{a0} + \mathbf{b0}) \cdot \mathbf{c0})$$

Note the repeated appearance of $(\mathbf{a}i \cdot \mathbf{b}i)$ and $(\mathbf{a}i + \mathbf{b}i)$ in the formula above. These two important factors are traditionally called *generate* (g*i*) and *propagate* (p*i*):

$$\mathbf{g}i = \mathbf{a}i \cdot \mathbf{b}i$$

$$\mathbf{p}i = \mathbf{a}i + \mathbf{b}i$$

Using them to define c*i* + 1, we get

$$\mathbf{c}i + 1 = \mathbf{g}i + \mathbf{p}i \cdot \mathbf{c}i$$

To see where the signals get their names, suppose g*i* is 1. Then

$$\mathbf{c}i + 1 = \mathbf{g}i + \mathbf{p}i \cdot \mathbf{c}i = 1 + \mathbf{p}i \cdot \mathbf{c}i = 1$$

That is, the adder *generates* a CarryOut (c*i* + 1) independent of the value of CarryIn (c*i*). Now suppose that g*i* is 0 and p*i* is 1. Then

$$\mathbf{c}i + 1 = \mathbf{g}i + \mathbf{p}i \cdot \mathbf{c}i = 0 + 1 \cdot \mathbf{c}i = \mathbf{c}i$$

That is, the adder *propogates* CarryIn to a CarryOut. Putting the two together, CarryIn*i* + 1 is a 1 if either g*i* is 1 or both p*i* is 1 and CarryIn*i* is 1.

As an analogy, imagine a row of dominoes set on edge. The end domino can be tipped over by pushing one far away, provided there are no gaps between the two. Similarly, a carry out can be made true by a generate far away, provided all the propagates between them are true.

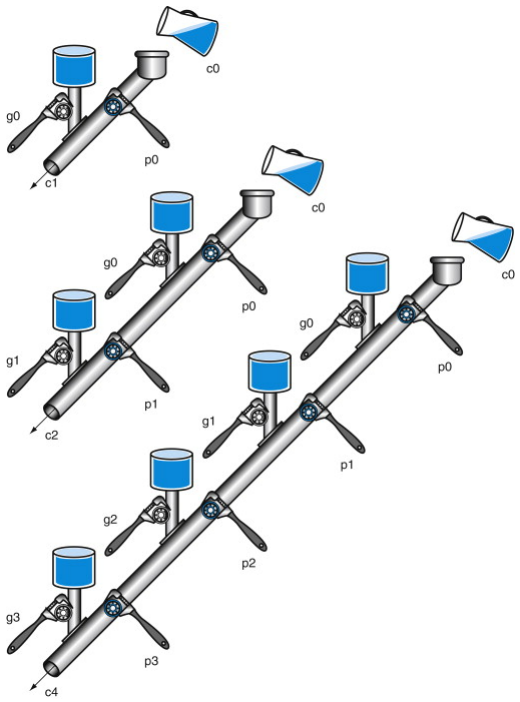
Relying on the definitions of propagate and generate as our first level of abstraction, we can express the CarryIn signals more economically. Let's show it for 4 bits:

$$\begin{aligned}\mathbf{c1} &= \mathbf{g0} + (\mathbf{p0} \cdot \mathbf{c0}) \\ \mathbf{c2} &= \mathbf{g1} + (\mathbf{p1} \cdot \mathbf{g0}) + (\mathbf{p1} \cdot \mathbf{p0} \cdot \mathbf{c0}) \\ \mathbf{c3} &= \mathbf{g2} + (\mathbf{p2} \cdot \mathbf{g1}) + (\mathbf{p2} \cdot \mathbf{p1} \cdot \mathbf{g0}) + (\mathbf{p2} \cdot \mathbf{p1} \cdot \mathbf{p0} \cdot \mathbf{c0}) \\ \mathbf{c4} &= \mathbf{g3} + (\mathbf{p3} \cdot \mathbf{g2}) + (\mathbf{p3} \cdot \mathbf{p2} \cdot \mathbf{g1}) + (\mathbf{p3} \cdot \mathbf{p2} \cdot \mathbf{p1} \cdot \mathbf{g0}) + (\mathbf{p3} \cdot \mathbf{p2} \cdot \mathbf{p1} \cdot \mathbf{p0} \cdot \mathbf{c0})\end{aligned}$$

These equations just represent common sense: CarryIn*i* is a 1 if some earlier adder generates a carry and all intermediary adders propagate a carry. The figure below uses plumbing to try to explain carry lookahead.

Figure 7.6.1: A plumbing analogy for carry lookahead for 1 bit, 2 bits, and 4 bits using water pipes and valves (COD Figure A.6.1).

The wrenches are turned to open and close valves. Water is shown in color. The output of the pipe (c*i* + 1) will be full if either the nearest generate value (g*i*) is turned on or if the *i* propagate value (p*i*) is on and there is water further upstream, either from an earlier generate or a propagate with water behind it. CarryIn (c0) can result in a carry out without the help of any generates, but with the help of *all* propagates.



Even this simplified form leads to large to large equations and, hence, considerable logic even for a 16-bit adder. Let's try moving to two levels of abstraction.

Fast carry using the second level of abstraction

First, we consider this 4-bit adder with its carry-lookahead logic as a single building block. If we connect them in ripple carry fashion to form a 16-bit adder, the add will be faster than the original with a little more hardware.

To go faster, we'll need carry lookahead at a higher level. To perform carry look ahead for 4-bit adders, we need to propagate and generate signals at this higher level. Here they are for the four 4-bit adder blocks:

$$P0 = p3 \cdot p2 \cdot p1 \cdot p0$$
$$P1 = p7 \cdot p6 \cdot p5 \cdot p4$$
$$P2 = p11 \cdot p10 \cdot p9 \cdot p8$$
$$P3 = p15 \cdot p14 \cdot p13 \cdot p12$$

This is, the "super" propagate signal for the 4-bit abstraction (P*i*) is true only if each of the bits in the group will propagate a carry.

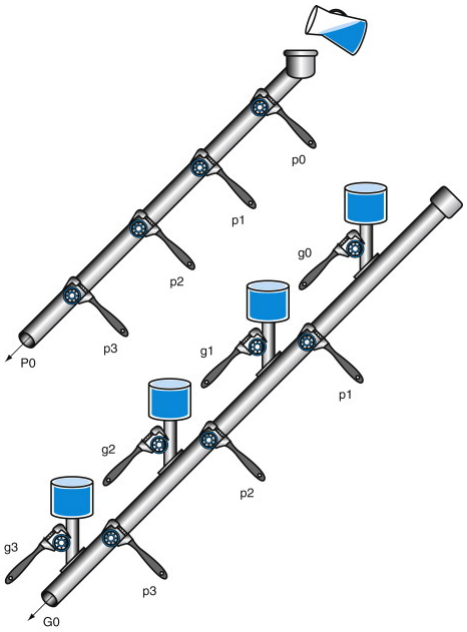
For the "super" generate signal (G*i*), we care only if there is a carry out of the most significant bit of the 4-bit group. This obviously occurs if generate is true for that most significant bit; it also occurs if an earlier generate is true *and* all the intermediate propagates, including that of the most significant bit, are also true:

$$G0 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0)$$
$$G1 = g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4)$$
$$G2 = g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8)$$
$$G3 = g15 + (p15 \cdot g14) + (p15 \cdot g14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12)$$

The figure below updates our plumbing analogy to show P0 and G0.

Figure 7.6.2: A plumbing analogy for the next-level carry-lookahead signals P0 and G0 (COD Figure A.6.2).

P0 is open only if all four propagates (p*i*) are open, while water flows in G0 only if at least one generate (g*i*) is open and all the propagates downstream from that generate are open.

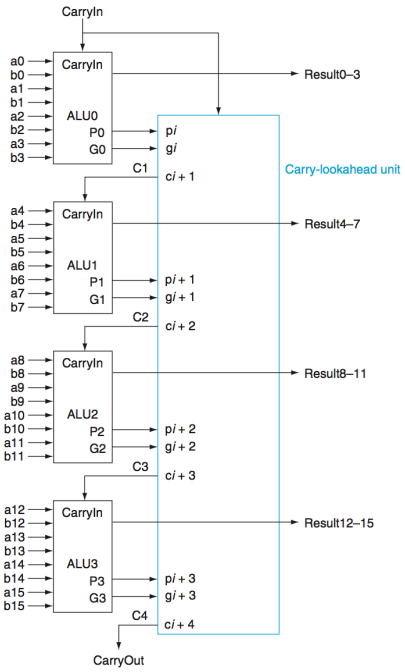


Then the equations at this higher level of abstraction for the carry in for each 4-bit group of the 16-bit adder (C1, C2, C3, C4 in the figure below) are very similar to the carry out equations for each bit of the 4-bit adder (c1, c2, c3, c4):

$$C1 = G0 + (P0 \cdot c0)$$
$$C2 = G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0)$$
$$C3 = G2 + (P2 \cdot G1) + (P2 \cdot P1 \cdot G0) + (P2 \cdot P1 \cdot P0 \cdot c0)$$
$$C4 = G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0)$$

Figure 7.6.3: Four 4-bit ALUs using carry lookahead to form a 16-bit adder (COD Figure A.6.3).

Note that the carries come from the carry-lookahead unit, not from the 4-bit ALUs.



The figure above shows 4-bit adders connected with such a carry-lookahead unit. The exercises explore the speed differences between these carry schemes, different notations for multibit propagate and generate signals, and the design of a 64-bit adder.

Example 7.6.1: Both levels of the propagate and generate.

Determine the g*i*, p*i*, P*i*, and G*i* values of these two 16-bit numbers:

$$\begin{aligned}\mathbf{a:}&\quad 0001\ 1010\ 0011\ 0011_{\mathbf{two}}\\ \mathbf{b:}&\quad 1110\ 0101\ 1110\ 1011_{\mathbf{two}}\end{aligned}$$

Also, what is CarryOut15 (C4)?

Answer

Aligning the bits makes it easy to see the values of generate g*i* (a*i* . b*i*) and propagate p*i* (a*i* + b*i*):

$$\begin{aligned}\mathbf{a:}&\quad 0001\ 1010\ 0011\ 0011\\ \mathbf{b:}&\quad 1110\ 0101\ 1110\ 1011\\ \mathbf{g}i\mathbf{:}&\quad 0000\ 0000\ 0010\ 0011\\ \mathbf{p}i\mathbf{:}&\quad 1111\ 1111\ 1111\ 1011\end{aligned}$$

where the bits are numbered 15 to 0 from left to right. Next, the "super" propagates (P3, P2, P1, P0) are simply the AND of the lower-level propagates:

$$\begin{aligned}\mathbf{P3} &= \mathbf{1.1.1.1} = \mathbf{1}\\ \mathbf{P2} &= \mathbf{1.1.1.1} = \mathbf{1}\\ \mathbf{P1} &= \mathbf{1.1.1.1} = \mathbf{1}\\ \mathbf{P0} &= \mathbf{1.0.1.1} = \mathbf{0}\end{aligned}$$

The "super" generates are more complex, so use the following equations:

$$\begin{aligned}\mathbf{G0} &= \mathbf{g3} + (\mathbf{p3.g2}) + (\mathbf{p3.p2.g1}) + (\mathbf{p3.p2.p1.g0})\\ &= 0 + (1.0) + (1.0.1) + (1.0.1.1)\\ &= 0 + 0 + 0 + 0\\ &= 0\\ \mathbf{G1} &= \mathbf{g7} + (\mathbf{p7.g6}) + (\mathbf{p7.p6.g5}) + (\mathbf{p7.p6.p5.g4})\\ &= 0 + (1.0) + (1.1.1) + (1.1.1.0)\\ &= 0 + 0 + 1 + 0\\ &= 1\\ \mathbf{G2} &= \mathbf{g11} + (\mathbf{p11.g10}) + (\mathbf{p11.p10.g9}) + (\mathbf{p11.p10.p9.g8})\\ &= 0 + (1.0) + (1.1.0) + (1.1.1.0)\\ &= 0 + 0 + 0 + 0\\ &= 0\\ \mathbf{G3} &= \mathbf{g15} + (\mathbf{p15.g14}) + (\mathbf{p15.p14.g13}) + (\mathbf{p15.p14.p13.g12})\\ &= 0 + (1.0) + (1.1.0) + (1.1.1.0)\\ &= 0 + 0 + 0 + 0\\ &= 0\end{aligned}$$

Finally, CarryOut15 is

$$\begin{aligned}\mathbf{C4} &= \mathbf{G3} + (\mathbf{P3.G2}) + (\mathbf{P3.P2.G1}) + (\mathbf{P3.P2.P1.G0}) + (\mathbf{P3.P2.P1.P0.c0})\\ &= 0 + (1.0) + (1.1.1) + (1.1.1.0) + (1.1.1.0.0)\\ &= 0 + 0 + 1 + 0 + 0\\ &= 1\end{aligned}$$

Hence, there *is* a carry out when adding these two 16-bit numbers.

The reason carry lookahead can make carries faster is that all logic begins evaluating the moment the clock cycle begins, and the result will not change once the output of each gate stops changing. By taking the shortcut of going through fewer gates to send the carry in signal, the output of the gates will stop changing sooner, and hence the time for the adder can be less.

To appreciate the importance of carry lookahead, we need to calculate the relative performance between it and ripple carry adders.

Example 7.6.2: Speed of ripple carry versus carry lookahead.

One simple way to model time for logic is to assume each AND or OR gate takes the same time for a signal to pass through it. Time is estimated by simply counting the number of gates along the path through a piece of logic. Compare the number of *gate delays* for paths of two 16-bit adders, one using ripple carry and one using two-level carry lookahead.

Answer

COD Figure A.5.5 (Adder hardware for the CarryOut signal) shows that the carry out signal takes two gate delays per bit. Then the number of gate delays between a carry in to the least significant bit and the carry out of the most significant is 32 X 2 = 64.

For carry lookahead, the carry out of the most significant bit is just C4, defined in the example. It takes two levels of logic to specify C4 in terms of P*i* and G*i* (the OR of several AND terms). P*i* is specified in one level of logic (AND) using p*i*, and G*i* is specified in two levels using p*i* and g*i*, so the worst case for this next level of abstraction is two levels of logic. p*i* and g*i* are each one level of logic, defined in terms of a*i* and b*i*. If we assume one gate delay for each level of logic in these equations, the worst case is 2 + 2 + 1 = 5 gate delays.

Hence, for the path from carry in to carry out, the 16-bit addition by a carry-lookahead adder is six times faster, using this very simple estimate of hardware speed.

Summary

Carry lookahead offers a faster path than waiting for the carries to ripple through all 32 1-bit adders. This faster path is paved by two signals, generate and propagate. The former creates a carry regardless of the carry input, and the latter passes a carry along. Carry lookahead also gives another example of how abstraction is important in computer design to cope with complexity.

PARTICIPATION ACTIVITY 7.6.1: Check yourself: Carry ripple and carry lookahead adder speeds.

1) Using the simple estimate of hardware speed above with gate delays, what is the relative performance of a ripple carry 8-bit add versus a 64-bit add using carry-lookahead logic?
   - A 64-bit carry-lookahead adder is three times faster: 8-bit adds are 16 gate delays and 64-bit adds are 7 gate delays.
   - They are about the same speed, since 64-bit adds need more levels of logic in the 16-bit adder.
   - 8-bit adds are faster than 64 bits, even with carry lookahead.

Elaboration

We have now accounted for all but one of the arithmetic and logical operations for the core LEGv8 instruction set: the ALU in COD Figure A.5.14 (The symbol commonly used to represent an ALU) omits support of shift instructions. It would be possible to widen the ALU multiplexor to include a left shift by 1 bit or a right shift by 1 bit. But hardware designers have created a circuit called a barrel shifter, which can shift from 1 to 63 bits in no more time than it takes to add two 64-bit numbers, so shifting is normally done outside the ALU.

Elaboration

The logic equation for the Sum output of the full adder can be expressed more simply by using a more powerful gate than AND and OR. An exclusive OR gate is true if the two operands disagree; that is,

$$\mathbf{x \neq y \Rightarrow 1 \text{ and } x == y \Rightarrow 0}$$

In some technologies, exclusive OR is more efficient than two levels of AND and OR gates. Using the symbol $\oplus$ to represent exclusive OR, here is the new equation:

$$\mathbf{Sum = a \oplus b \oplus CarryIn}$$

Also, we have drawn the ALU the traditional way, using gates. Computers are designed today in CMOS transistors, which are basically switches. CMOS ALU and barrel shifters take advantage of these switches and have many fewer multiplexors than shown in our designs, but the design principles are similar.

Elaboration

Using lowercase and uppercase to distinguish the hierarchy of generate and propagate symbols break down when you have more than two levels. An alternate notation that scales is $g_{i..j}$ and $p_{i..j}$ for the generate and propagate signals for bits i to j. Thus, $g_{1..1}$ is generated for bit 1, $g_{4..1}$ is for bits 4 to 1, and $g_{16..1}$ is for bits 16 to 1.

(*1) This section is in original form.

Provide feedback on this section