

Numbers, Addition, More Addition

Prof. Loh

CS3220 - Processor Design - Spring 2005

January 27, 2005

1 Review and Basics

1.1 Binary Numbers

Any positive integer n can be represented by a sequence of binary digits (i.e. bits) $b_k b_{k-1} \dots b_2 b_1 b_0$, where $b_i \in \{0, 1\}$:

$$n = \sum_{i=0}^k b_i 2^i$$

where $k = \lceil \log_2 n \rceil$.

Example: $17 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 10001_2$

The subscript notation will sometimes be used to denote the base of a number. It will not be included when the context makes the base obvious. A prefix of 0x denotes base 16 or hexadecimal notation.

1.2 Two's Complement

A k bit number can be used to represent any 2^k positive integers. If a representation for negative numbers is required, then an extra bit is needed for the sign of the number. In one's complement notation, the value $-n$ is represented by

$$\overline{1b_k b_{k-1} \dots b_2 b_1 b_0}$$

where $n = b_k b_{k-1} \dots b_2 b_1 b_0$ and $\overline{b_i}$ is the negation of b_i . This representation is undesirable because positive and negative numbers can not be directly manipulated without converting signs, and there are two representations of zero (a positive and a negative zero).

In two's complement notation, $2^{k-1} - 1$ positive numbers plus zero and 2^{k-1} negative numbers are mapped to the 2^k possible k bit numbers.

Example:, for $k = 3$:

Decimal Value	2's Complement
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Numbers in two's complement have some useful properties:

- No redundant zeros
- Most significant bit determines if number is negative
- Least significant bit determines if number is odd/even
- -1 is “next” to 0
- All numbers (positive or negative) can be added without performing sign conversions (sign extension may be needed though)

Quick way to convert n to $-n$: invert all of the bits of $n = b_k b_{k-1} \dots b_2 b_1 b_0$ and then add 1.

Example: (verify with previous table)

```

3 = 011
  → 100 (conversion to binary)
  → 101 (add one)

```

1.3 Circuit Analysis

It is important to be able to take a circuit, and analyze it to determine the gate delay, wire delays, and required area for layout. Although any circuit you may implement will have fixed parameters (such as numbers being 32 bits wide), asymptotic analysis can still be very important for this, because it can still give us an idea of how good a circuit it. And parameters may vary from one generation of a processor to the next.

As an example, we will design a left barrel shifter, and then determine the asymptotic gate delay, wire delay, and area requirements. This should also serve as a quick review of some basic principles from logic design.

The left barrel shifter performs the following computation:

$$z := x \ll_B y \equiv (x \ll y) | (x \gg (n - y))$$

where n is the total number of bits in our number (such as 16, 32, etc.). Note that this is not particularly meaningful if $y > n$ (shifting a 32-bit number by more than 32 bits is silly). The \ll , \gg , and $|$ are interpreted with the standard C semantics.

Figure 1 shows the circuit layout for an 8-bit left barrel shifter. The argument y specifies how many positions to shift by. The top row shifts by four positions if $y_2 = 1$, otherwise it doesn't shift at all. Similarly, the second row shifts by two positions if $y_1 = 1$. In general, for an n -bit shifter, the i -th row from the bottom (counting from zero!) shifts the bits by 2^i positions. This simply requires several rows of 2-to-1 muxes, each driven by the y bits.

Note that each y_i must drive the select input for n muxes. If there are too many gates connected together like this, the amount of time required to charge or discharge the gates' inputs becomes quite large. To prevent unnecessary

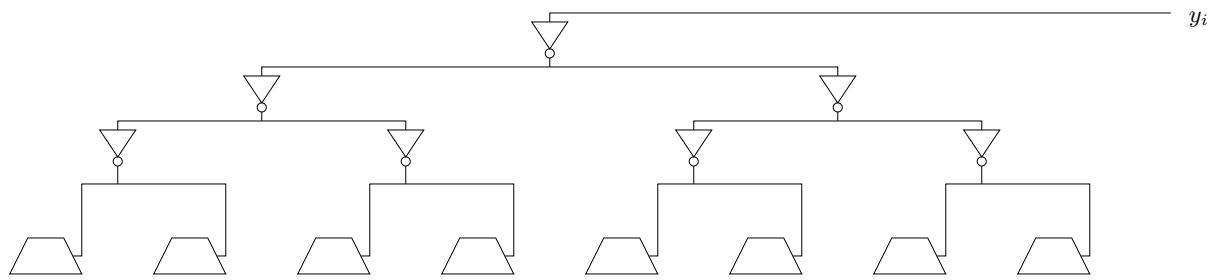
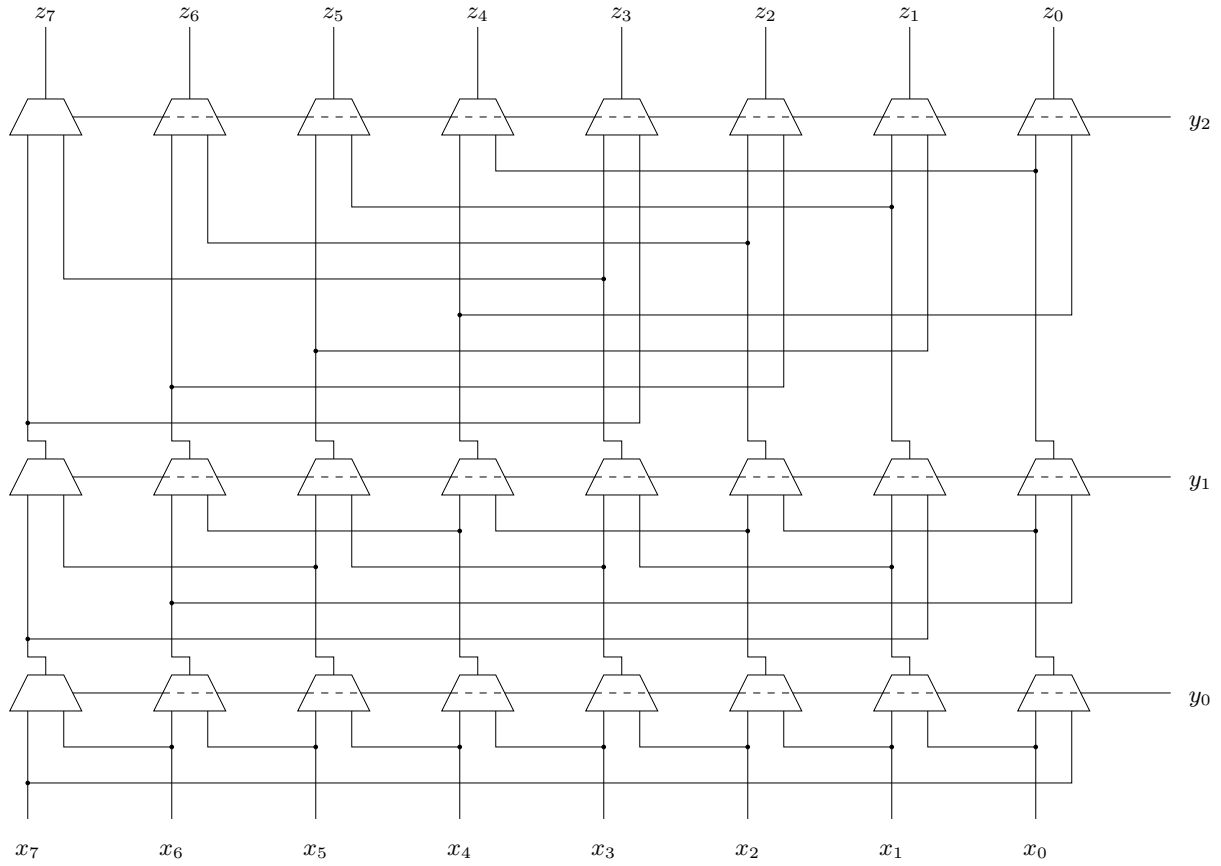


Figure 1: Top: an 8-bit left barrel shifter. Bottom: a fan-out tree for driving the y_i signals to the mux select inputs.

circuit delays, the *fan-out* of a gate is often limited to 4 gates, but this depends on many other factors as well (wire lengths, the size of the driving gate, the sizes of the gates being driven, etc.). The bottom of Figure 1 shows a fan-out tree with a fan-out limitation of 2. Another feature of the fan-out tree (apart from being faster than driving all n gates at the same time) is that the signal will reach all of the inputs at about the same time. For some applications, this may not matter. For others (such as clock distribution), it can be very critical.

Let us first analyze what the gate delay will be. The gate delay is the number of gates that a signal must pass through in the worst case. For arbitrary circuits, this can be a very difficult problem to solve, because there may be many paths through the circuit. Because the shifter has a very regular structure, it is easy to analyze. We already made the observation that we really only want to shift for values of y that are less than $\lg n^1$. Therefore, we limit y to be $\lg n$ -bits wide. From the earlier explanation of how the shifter works, we know that for every bit of y , we need one layer of muxes. Thus, $\lg n$ layers of muxes are needed, and the total gate delay is $O(\lg n)$.

Wire delay and circuit area are functions of a particular layout. If a different layout is used, the wire delay and circuit area can change. In this circuit, it is perhaps a little easier to first analyze the area. The area is simply the width times the height. The width of the circuit is $O(n)$ because for any row, there are n muxes. Another way to argue that the width is $O(n)$ is that each of the muxes takes two inputs, which gives us $2n$ wires in parallel, which will also require a width of $O(n)$. How about the height? The bottom row requires enough room for two wire tracks to run horizontally, plus the height of one mux. The next row requires four tracks of wiring, and the following requires eight. The i -th row requires $2 \cdot 2^i$ rows for wiring, plus the height of the mux. Thus,

$$height(n) = \sum_{i=0}^{\lg n - 1} (2 \cdot 2^i + O(1))$$

The $O(1)$ is for the height of the mux. It is some constant, but we are not particularly concerned with the exact value. With some more algebraic manipulation:

$$\begin{aligned} height(n) &= 2 \sum_{i=0}^{\lg n - 1} 2^i + \sum_{i=0}^{\lg n - 1} O(1) \\ &= 2 \cdot (2^{\lg n} - 1) + \lg n \cdot O(1) \\ &= 2(n - 1) + O(\lg n) \\ &= O(n) \end{aligned}$$

Therefore, the total area for an n -bit shifter is $O(n) \times O(n) = O(n^2)$.

From the circuit layout in Figure 1, the longest distance a signal will travel will be x_7 when $y = 7 = 111_2$. In the bottom level, x_7 must travel all the way to the right. This distance is $O(n)$, because there are $n - 1$ other wires (or n mux widths) that the signal must cross. In the second row, the signal can only travel a distance of two columns; four columns in the next row, and so on. You can see that in the worst case, the signal will be able to travel back to column 6. The total distance traveled will be approximately across the circuit and back, which is still $O(n)$. There is also some vertical distance that has to be traveled by all of the signals. From our area calculation, we know that the height of the chip is $O(n)$. We can also see that every x signal starts at the bottom and makes its way to the top, and no signals ever travel in the downward direction. Therefore, the vertical component of the wire delay is also $O(n)$. The total wire delay is the sum of the horizontal and vertical components, which still gives us $O(n)$.

We did ignore the y signals in the above computation. The height of the fan-out tree is $O(\lg n)$, and there is one per row. So the true height is really $O(n)$ (from before) + $\lg n \cdot O(\lg n)$ for one fan-out tree per row. This totals up to $O(n + (\lg n)^2)$ which is still $O(n)$, so our earlier results do not change.

Often in circuit analysis, we simply ignore any effects that are due to the wires. Whether or not we should worry about the wires depends on the situation. For a small 8-bit shifter like the one we just looked at, the overall circuit delay will probably be dominated by the gate delay. But if we have a 64-bit shifter, there will be about 128 tracks of wire running horizontally, and 64 muxes to a row. The wires now have to travel some fairly substantial distances, and

¹We will use \lg to mean \log_2 , \ln for the natural logarithm, and just \log when the base doesn't matter. $\log_y x = \frac{\log_b x}{\log_b y}$ for any base b , and therefore $O(\log_a n) = O(\log_b n)$, and so we may often just omit the base.

thus the wire delays may have a significant impact on the overall circuit delay. Anyone who has done a bit of VLSI layout should be able to appreciate the importance of worrying about your wires.

Asymptotic analysis doesn't tell us exactly how fast or big our circuits are going to be, but it is an important tool for designing circuits, and for choosing what circuits you want to use. Often, you can tradeoff delay for area (i.e. you can get a slower circuit that takes up less room).

2 Addition

2.1 The Full Adder: 1-bit addition

In elementary school base 10 addition, we align the two numbers we want to add (the operands) and then proceed one column at a time, carrying over digits as necessary:

$$\begin{array}{r}
 \text{carry:} \quad \quad 1 \quad \quad 1 \\
 \text{x:} \quad \quad \quad 1 \ 2 \ 3 \ 4 \ 5 \\
 \text{y:} \quad + \quad 3 \ 8 \ 1 \ 7 \ 2 \\
 \hline
 \text{z:} \quad \quad 5 \ 0 \ 5 \ 1 \ 7
 \end{array}$$

To add binary numbers by hand, we can do the same thing:

$$\begin{array}{r}
 \text{carry:} \quad \quad \quad 1 \ 1 \ 1 \\
 \text{x:} \quad \quad \quad 0 \ 0 \ 1 \ 0 \ 1 \\
 \text{y:} \quad + \quad 1 \ 0 \ 1 \ 1 \ 1 \\
 \hline
 \text{z:} \quad \quad 1 \ 1 \ 1 \ 0 \ 0
 \end{array}$$

Notice that each column i takes three inputs x_i, y_i and the carry from the previous column (c_i), and generates two outputs z_i and the carry to the next column (c_{i+1}). The initial carry in (c_0) is zero. The full adder is a circuit that computes z_i and c_{i+1} given x_i, y_i and c_i . This is the truth table for the full adder:

x_i	y_i	c_i	c_{i+1}	z_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The block diagram for the full adder is shown in Figure 2.

2.2 Ripple Carry Adder

The Ripple Carry Adder (RCA) is the simplest adder. It mimics the “elementary school” approach of adding numbers by forming a straight chain of full adders, connecting the carry-out of one stage to the carry-in of the next. The block diagram for the RCA is shown in Figure 3. The carry out of one stage is fed into the carry in of the next stage. The initial carry in is set to zero, and the last carry out is ignored (or it can be used to detect arithmetic overflow). The gate delay is $O(n)$ because the carry must sequentially “ripple” through all n full adders. For the straight layout, the area required is $O(n)$ and the wire delay is $O(n)$.

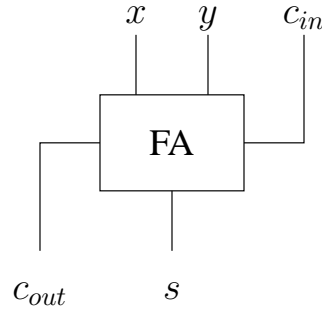


Figure 2: The full adder block diagram.

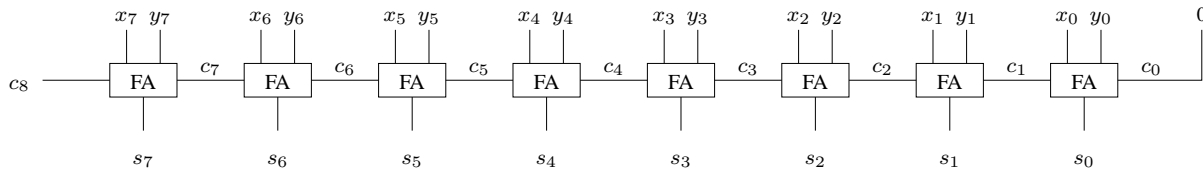


Figure 3: The ripple carry adder (RCA).

2.3 Carry Skip Adder

The last bit (most significant bit) of the RCA must wait for the carry information to make its way through all previous $n - 1$ stages before it (the last stage) may proceed. To reduce this delay, we introduce the concept of the propagate signal. Our goal is to identify situations where the carry information can be “short cutted”. Let’s take a look at one bit’s worth of addition (not necessarily the first bit!):

x	y	c_{out}	
0	0	0	← always 0, regardless of c_{in}
0	1	?	
1	0	?	← c_{out} always equals c_{in}
1	1	1	← always 1, regardless of c_{in}

In the first and last case, it doesn’t matter what the value of the incoming carry is. The middle two cases are more interesting because these are the cases where the circuit will have to generate a signal based on the carry in. Fortunately, the signal generated is simply the same as the carry in. For example, if $x_4 = 0$ and $y_4 = 1$, then $c_5 = c_4$. This effect can be “chained”: if $x_4 = 0, x_3 = 1$ and $y_4 = 1, y_3 = 0$, then $c_5 = c_4 = c_3$. In this case, if we detect that both stages 3 and 4 should simply pass on (or *propagate*) the earlier carry in, then the carry logic for bits 3 and 4 can be completely circumvented.

Figure 4 shows how the carry logic can be “skipped”. To compute if bit i may propagate its carry, it simply has to check for the case that only one of its inputs (x and y) is a one. This can be easily computed with an xor, and this signal is called the propagate signal (P_i). If both P_i and P_{i+1} are one, then the carry in for these two blocks is passed directly on to the next stage. The delay of two levels of carry logic has been converted into a single multiplexer delay. Therefore, the overall number of gate delays has been approximately halved. The key insight here is that although we can not compute all of the carry in bits at the same time (since they are dependent on each other), we can compute in parallel whether a certain bit will end up propagating the carry bit or not. This is possible because P_i depends only on x_i and y_i , both of which are immediately available.

The number of bits to put into a “skip block” is denoted by k . The circuit just discussed would be a carry skip adder with $k = 2$. Figure 5 shows a carry skip adder for $n = 16$ (the number of bits) and $k = 4$. Let us trace the worst case gate delay for the circuit. First, a carry signal may be generated by the very first block of k bits. This requires

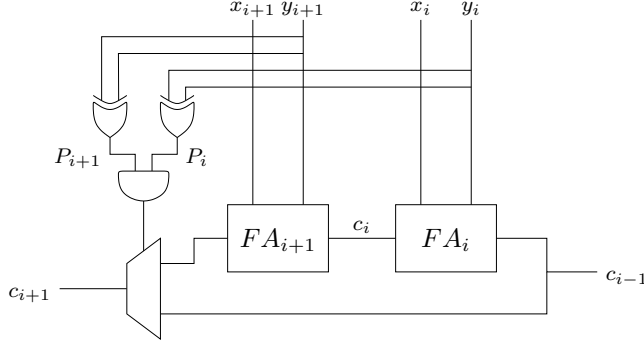


Figure 4: A carry-skip adder block ($k = 2$).

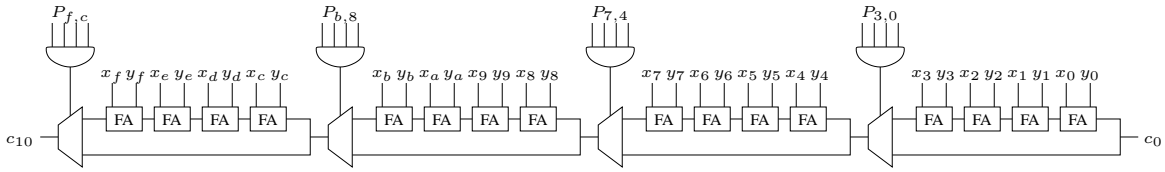


Figure 5: An $n = 16$ bit carry skip adder with $k = 4$ bit skip blocks. $P_{i,j}$ denotes all of the propagate signals from P_i to P_j . Hexadecimal notation is used for the indices that are greater than 9 due to space constraints.

going through k full adders. In the worst case, this carry signal now has to propagate all the way to the last block of k bits. This requires going through $\frac{n}{k} - 1$ MUXs. Finally, the carry may have to propagate through the last block of k bits, incurring another k full adder delays. The total delay is thus:

$$\tau(\text{carryskip}(n, k)) = 2 \cdot k \cdot \tau(\text{FA}) + \left(\frac{n}{k} - 1\right) \cdot \tau(\text{MUX})$$

where $\tau(X)$ is the propagation delay for a gate of type X . Treating all gate delays as the same (i.e. $O(1)$), the total gate delay is $O(k + \frac{n}{k})$. The question that should be asked now is how is the value of k chosen to minimize the overall delay?

To minimize the expression $O(k + \frac{n}{k})$, we find k such that the derivate of the expression is zero. In the following, the big- O notation will be omitted in some steps.

$$\begin{aligned} & \frac{d}{dk} \left(k + \frac{n}{k}\right) \quad \text{differentiate} \\ & = 1 - \frac{n}{k^2} = 0 \quad \text{set equal to zero} \\ & \Rightarrow k^2 = n \\ & \Rightarrow k = O(\sqrt{n}) \end{aligned}$$

So to minimize the total gate delay of the carry skip adder, k should be proportional to \sqrt{n} . Substituting this back into the original expression for the gate delay, we get $O(\sqrt{n} + \frac{n}{\sqrt{n}}) = O(\sqrt{n})$. This is a considerable improvement over the RCA. In the next section, we will again exploit the fact that we can compute propagation information for all bits in parallel to build an even faster adder.

2.4 Lookahead Carry Adder

In this section, we will only concern ourselves with how fast we can compute the carry bits. Once we have that, the remaining full adders only add an additional $O(1)$ gate delay.

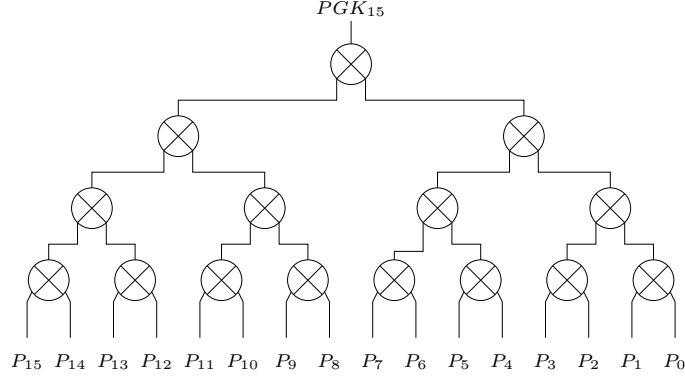


Figure 6: A tree of PGK operators to compute PGK_{15} .

The Lookahead Carry Adder (LCA) extends the propagate signal beyond a simple propagate/don't propagate to a three valued signal. The possible values of this new propagate signal are *propagate* (P), *generate* (G), and *kill* (K). The meaning of generate is that regardless of the incoming carry bit, we will “generate” a carry bit (carry out is 1). The meaning of kill is the opposite, in that the incoming carry bit is “killed” and forced to zero. To generate the P_i 's, we use the following table:

x	y	P
0	0	K
0	1	P
1	0	P
1	1	G

Each block of bits will generate a PGK signal. For two adjacent blocks, the circuit combines the two PGK signals to output a PGK signal for the combined block. This is computed as follows:

PGK_l	PGK_r	$PGK_{out} = PGK_l \otimes PGK_r$
K	X	K
P	K	K
P	P	P
P	G	G
G	X	G

X denotes “don't care” or any input. PGK_l is the PGK signal for the left block (more significant), and PGK_r is the PGK signal for the right block (less significant). The \otimes is the associative PGK combining operator. If PGK_l is K, then the combined PGK signal must also be K. This is due to the fact that when a kill signal is generated, it is independent of any earlier carries. Similarly for the case of $PGK_l = G$. If PGK_l is P, then the circuit should propagate the PGK signal from the right block. A tree of PGK operators is shown in Figure 6. If $PGK_{15} = K$, then it means that regardless of what the carry in for the whole block of 16 bits is, the carry out will be zero. On the other hand, if $PGK_{15} = P$, then the carry in can bypass the entire 16-bit block.

The PGK tree computes the signal PGK_i in $\lceil \lg i \rceil$ \otimes -delays. What we want is n signals, from PGK_0 all the way up to PGK_{n-1} . We could simply use n trees, each computing one of the PGK_i 's. This is not a good solution because it unnecessarily uses up a lot of space.

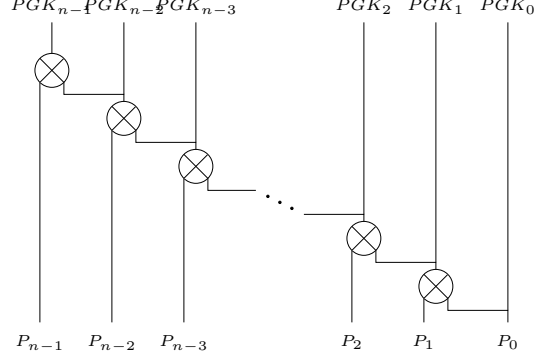


Figure 7: A linear chain of \otimes operators to compute all of the PGK_i signals.

Observe again that what we want to compute is:

$$\begin{aligned}
 PGK_0 &= P_0 \\
 PGK_1 &= P_1 \otimes P_0 \\
 PGK_2 &= P_2 \otimes P_1 \otimes P_0 \\
 &\vdots \\
 PGK_{n-1} &= P_{n-1} \otimes P_{n-2} \otimes \cdots \otimes P_1 \otimes P_0 \\
 &\text{or} \\
 PGK_j &= \bigotimes_{i=0}^j P_n
 \end{aligned}$$

There is a very regular structure. The PGK equations can instead be written recursively as follows:

$$PGK_i = \begin{cases} P_0 & \text{if } i = 0 \\ P_i \otimes PGK_{i-1} & \text{otherwise} \end{cases}$$

One simple way to compute all of the PGK_i 's is to directly implement this recursive formula in hardware. This is shown in Figure 7. The problem with this approach is that it takes $O(n)$ gate delays to get PGK_{n-1} , which does us no good.

Let us return to the PGK-tree in Figure 6. We will use the subtree that combines P_0 through P_3 as an example. Remember that the full adder in bit position i actually wants the signal from one position earlier, $i - 1$. So bit position 1 wants PGK_0 , which is simply P_0 . We can pass this signal directly to position 1, as illustrated in Figure 8. Position 2 wants PGK_1 , which is conveniently the output of the right subtree. Lastly, position 3 want PGK_2 , which can be computed by combining P_2 with the output of the right subtree.

Let us consider the next subtree covering bit positions 4 through 7. Position 4 wants PGK_3 , which (similar to PGK_1) is simply the value produced by the subtree to the right. Figure 9 illustrates this. The computation for bit position 5 is similar to position 3. Bit position 6 is more interesting. We do not want to simply take PGK_4 and combine it with P_5 . Such an approach will ultimately lead to a linear chain of \otimes 's. To compute PGK_5 , observe that we already have $PGK_{4,5}$ as well as $PGK_3 = PGK_{0,3}$ from the other subtrees (let $PGK_{j,k} = \bigotimes_{i=j}^k P_n$). Also note that PGK_3 is already being passed down the tree for the subtree covering bit positions 4 and 5. The signals cross paths at the parent of the 4,5 subtree and the 6,7 subtree. At this point, we can combine $PGK_{0,3}$ with $PGK_{4,5}$ to obtain PGK_5 . Once we have this signal, computing PGK_6 is the same as computing PGK_1 or PGK_3 .

The reason why this works is that we can compute different "sections" of the PGK signals separately and in parallel. This is due to the fact that the \otimes operator is associative. The general node is shown in Figure 10. The node

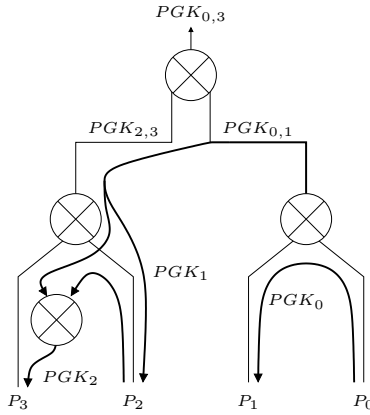


Figure 8: A subtree for computing PGK_0 through PGK_3 . The bold lines indicate signals that are directly used to compute the final PGK values that go back down the tree.

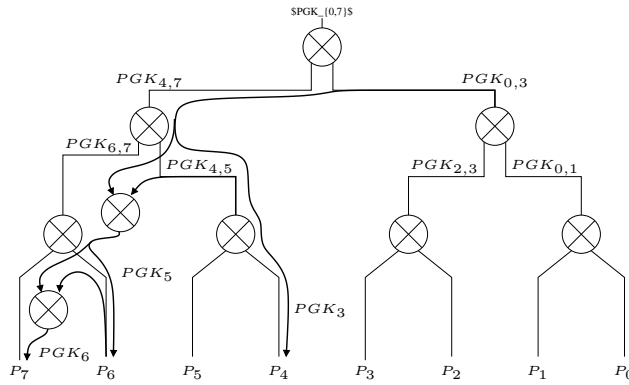


Figure 9: PGK circuitry to compute PGK_5 and PGK_6 . The gates for PGK_0 through PGK_4 were omitted to reduce clutter.

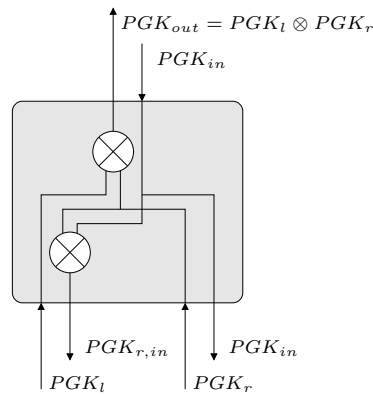


Figure 10: The general node for the PGK-tree.

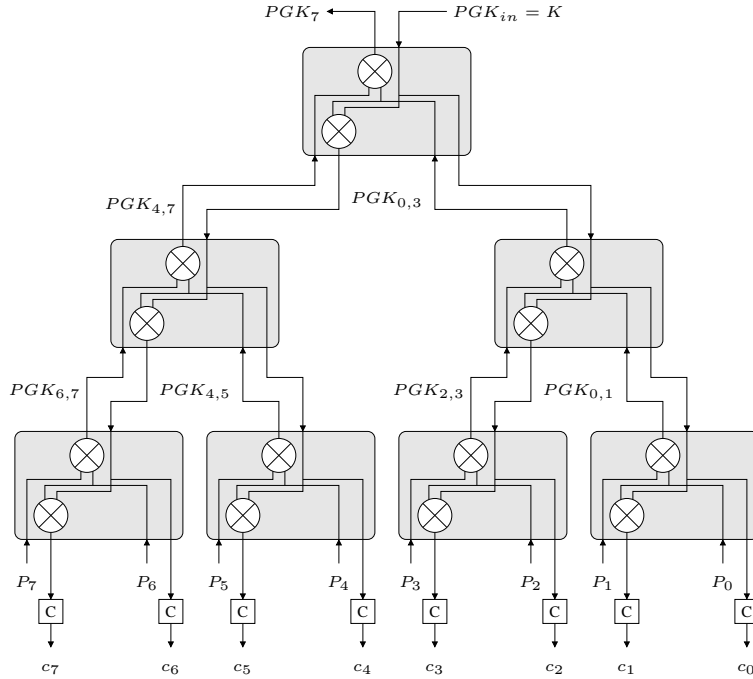


Figure 11: A parallel prefix circuit for $n = 8$.

contains one gate for computing the “upward” signal (the subterms of the prefix), and one gate for computing the “downward” signals (combining the subterms). Although Figure 9 is a little messy, you should be able to identify these two gates and the corresponding wires for the parent of the 4,5 subtree and the 6,7 subtree. The full tree for $n = 8$ is shown in Figure 11. The initial signal into the tree is K , since there is no carry in from before bit position zero. Since the PGK signals are two bit values, they must be converted back into a single bit carry at the leaves. The circuit that performs this is the box labeled C in Figure 11. Assuming an encoding of $K = 00, P = 01, G = 11$, then the conversion simply involves dropping the least significant bit of the PGK signal.

The computation is called a “prefix” computation, because each PGK_i is a prefix of the final PGK signal $P_{n-1} \otimes \dots \otimes P_1 \otimes P_0$. The circuit presented is called a parallel prefix tree because it computes all of the prefixes in parallel. The gate delay through the prefix tree is $(2 \lg n - 1) \cdot \tau(\otimes)$ (the signal starting from P_0 goes through $\lg n - 1$ gates on its way up the tree, and then at the root, it starts back down the tree, passing through $\lg n$ gates before it reaches the leftmost leaf. The only other computations are the initial computation for the P_i 's and then the final full adder. The total gate delay is thus $O(\lg n)$. This is a very big gain over our previous $O(n)$ and $O(\sqrt{n})$ implementations.

The parallel prefix circuit can actually be used to compute the prefixes of any associative operator. For instance, if we have $\vec{x} = \{x_{n-1}, \dots, x_1, x_0\}$, the prefix sums for \vec{x} is defined as

$$X_i = \sum_{n=0}^i x_n$$

By replacing the \otimes operator in Figure 11 with an addition operator, the same circuit can be used to compute the n different X_i 's.

As drawn in Figure 11, the parallel prefix tree requires $O(n \lg n)$ area and $O(n)$ wire delays. Figure 12 illustrates an alternative layout for the parallel prefix circuit called an H-tree layout. The H-tree layout can be used to compactly layout binary tree structures. There are \sqrt{n} leaves on each side, and each side has a length of \sqrt{n} leaves and $\sqrt{n} - 1$ parallel prefix nodes. Therefore the side length is $O(\sqrt{n})$. The total area required is therefore only $O(n)$ (can't do any better than that since there are n leaves in the tree). The wire delay is $O(\sqrt{n})$ (with a square layout, each side has

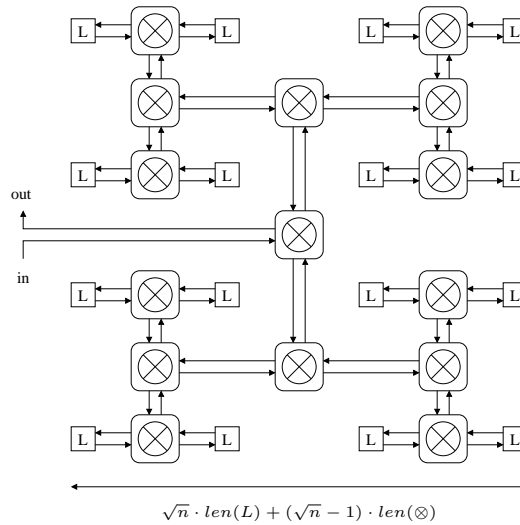
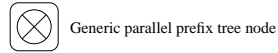


Figure 12: An H-tree layout of the parallel prefix tree circuit. The root of the tree is in the center.

length $O(\sqrt{n})$, and the longest path will have to cross the circuit horizontally ($O(\sqrt{n})$) and vertically ($O(\sqrt{n})$) for a total distance of $O(2\sqrt{n}) = O(\sqrt{n})$. Like the parallel prefix tree, the H-tree is also a recursive structure. You can build a $2n$ -node H-tree out of 2 separate n -node H-trees by simply putting them side by side, and connecting their individual roots with a new parallel prefix node (the new root). Unfortunately for the adder, this layout does not buy us anything because we still need room to get the $O(n)$ wires into and out of the circuit. Other binary tree circuits can take advantage of the H-tree layout though.