

ECE199JL: Introduction to Computer Engineering **Fall 2012**
Notes Set 2.3

Example: Bit-Sliced Addition

In this set of notes, we illustrate basic logic design using integer addition as an example. By recognizing and mimicking the structured approach used by humans to perform addition, we introduce an important abstraction for logic design. We follow this approach to design an adder known as a ripple-carry adder, then discuss some of the implications of the approach and highlight how the same approach can be used in software. In the next set of notes, we use the same technique to design a comparator for two integers.

One Bit at a Time

Many of the operations that we want to perform on groups of bits can be broken down into repeated operations on individual bits. When we add two binary numbers, for example, we first add the least significant bits, then move to the second least significant, and so on. As we go, we may need to carry from lower bits into higher bits. When we compare two (unsigned) binary numbers with the same number of bits, we usually start with the most significant bits and move downward in significance until we find a difference or reach the end of the two numbers. In the latter case, the two numbers are equal.

When we build combinational logic to implement this kind of calculation, our approach as humans can be leveraged as an abstraction technique. Rather than building and optimizing a different Boolean function for an 8-bit adder, a 9-bit adder, a 12-bit adder, and any other size that we might want, we can instead design a circuit that adds a single bit and passes any necessary information into another copy of itself. By using copies of this **bit-sliced** adder circuit, we can mimic our approach as humans and build adders of any size, just as we expect that a human could add two binary numbers of any size. The resulting designs are, of course, slightly less efficient than designs that are optimized for their specific purpose (such as adding two 17-bit numbers), but the simplicity of the approach makes the tradeoff an interesting one.

Abstracting the Human Process

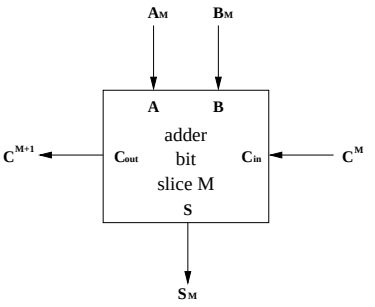
Think about how we as humans add two N-bit numbers, A and B. An illustration appears to the right, using $N = 8$. For now, let's assume that our numbers are stored in an unsigned representation. As you know, addition for 2's complement is identical except for the calculation of overflow. We start adding from the least significant bit and move to the left. Since adding two 1s can overflow a single bit, we carry a 1 when necessary into the next column. Thus, in general, we are actually adding three input bits. The carry from the previous column is usually not written explicitly by humans, but in a digital system we need to write a 0 instead of leaving the value blank.

carry C	0	0	1	1	0	0	1	1	(0)
A		0	0	1	1	1	0	1	1
B	+	0	0	1	1	0	0	0	1
sum S		0	1	1	0	1	1	0	0

information flows in this direction

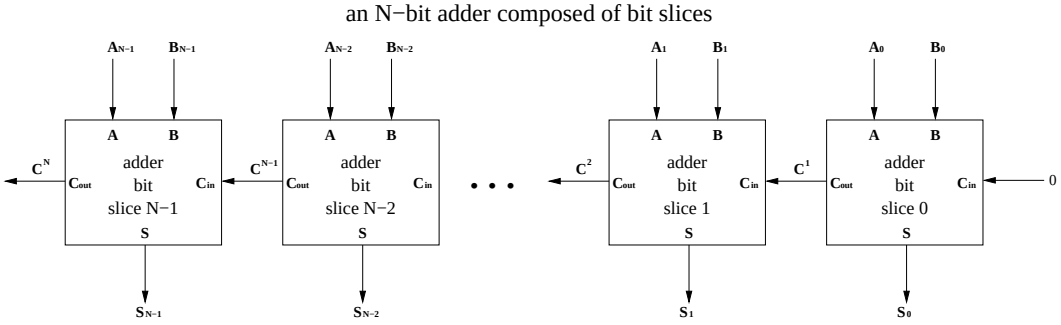
Focus now on the addition of a single column. Except for the first and last bits, which we might choose to handle slightly differently, the addition process is identical for any column. We add a carry in bit (possibly 0) with one bit from each of our numbers to produce a sum bit and a carry out bit for the next column. Column addition is the task that our bit slice logic must perform.

The diagram to the right shows an abstract model of our adder bit slice. The inputs from the next least significant bit come in from the right. We include arrowheads because figures are usually drawn with inputs coming from the top or left and outputs going to the bottom or right. Outside of the bit slice logic, we index the carry bits using the



bit number. The bit slice has C^M provided as an input and produces C^{M+1} as an output. Internally, we use C_{in} to denote the carry input, and C_{out} to denote the carry output. Similarly, the bits A_M and B_M from the numbers A and B are represented internally as A and B, and the bit S_M produced for the sum S is represented internally as S. The overloading of meaning should not confuse you, since the context (designing the logic block or thinking about the problem as a whole) should always be clear.

The abstract device for adding three inputs bits and producing two output bits is called a **full adder**. You may also encounter the term **half adder**, which adds only two input bits. To form an N-bit adder, we integrate N copies of the full adder—the bit slice that we design next—as shown below. The result is called a **ripple carry adder** because the carry information moves from the low bits to the high bits slowly, like a ripple on the surface of a pond.



Designing the Logic

Now we are ready to design our adder bit slice. Let's start by writing a truth table for C_{in} and S, as shown on the left below. To the right of the truth tables are K-maps for each output, and equations for each output are then shown to the right of the K-maps. We suggest that you work through identification of the prime implicants in the K-maps and check your work with the equations.

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	1	0	0	1
0	0	1	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

Cout

Cin \ AB	00	01	11	10
0	0	0	1	0
1	0	1	1	1

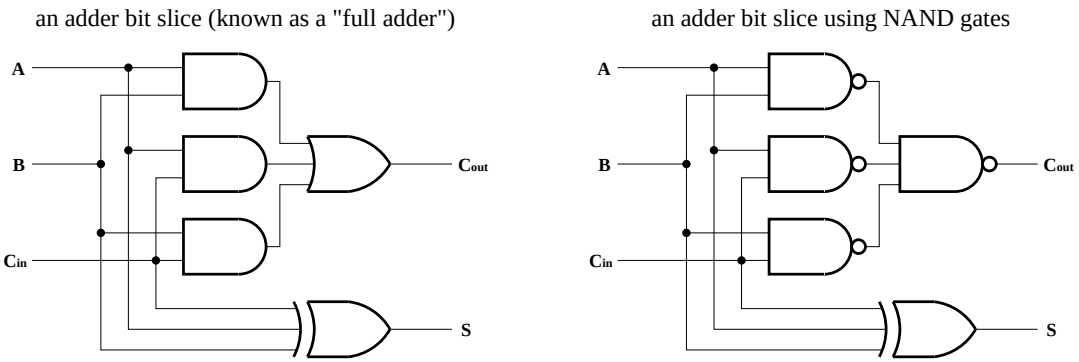
$$C_{out} = A\ B + A\ C_{in} + B\ C_{in}$$

S

Cin \ AB	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$$\begin{aligned} S &= A\ B\ C_{out} + A\ \overline{B}\ \overline{C_{out}} + \\ &\quad \overline{A}\ B\ \overline{C_{out}} + \overline{A}\ \overline{B}\ C_{out} \\ &= A \oplus B \oplus C_{out} \end{aligned}$$

The equation for C_{out} implements a **majority function** on three bits. In particular, a carry is produced whenever at least two out of the three input bits (a majority) are 1s. Why do we mention this name? Although we know that we can build any logic function from NAND gates, common functions such as those used to add numbers may benefit from optimization. Imagine that in some technology, creating a majority function directly may produce a better result than implementing such a function from logic gates. In such a case, we want the person designing the circuit to know that can make use of such an improvement. We rewrote the equation for S to make use of the XOR operation for a similar reason: the implementation of XOR gates from transistors may be slightly better than the implementation of XOR based on NAND gates. If a circuit designer provides an optimized variant of XOR, we want our design to make use of the optimized version.

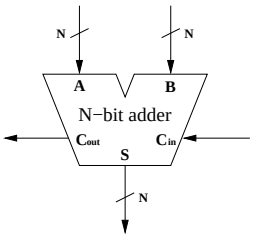


The gate diagrams above implement a single bit slice for an adder. The version on the left uses AND and OR gates (and an XOR for the sum), while the version on the right uses NAND gates, leaving the XOR as an XOR.

Let's discuss the design in terms of area and speed. As an estimate of area, we can count gates, remembering that we need two transistors per input on a gate. For each bit, we need three 2-input NAND gates, one 3-input NAND gate, and a 3-input XOR gate (a big gate; around 30 transistors). For speed, we make rough estimates in terms of the amount of time it takes for a CMOS gate to change its output once its input has changed. This amount of time is called a **gate delay**. We can thus estimate our design's speed by simply counting the maximum number of gates on any path from input to output. For this measurement, using a NAND/NOR representation of the design is important to getting the right answer. Here we have two gate delays from any of the inputs to the C_{out} output. The XOR gate may be a little slower, but none of its inputs come from other gates anyway. When we connect multiple copies of our bit slice logic together to form an adder, the A and B inputs to the outputs is not as important as the delay from C_{in} to the outputs. The latter delay adds to the total delay of our comparator on a per-bit-slice basis—this propagation delay gives rise to the name "ripple carry." Looking again at the diagram, notice that we have two gate delays from C_{in} to C_{out}. The total delay for an N-bit comparator based on this implementation is thus two gate delays per bit, for a total of $2N$ gate delays.

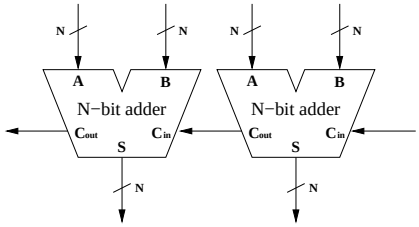
Adders and Word Size

Now that we know how to build an N-bit adder, we can add some detail to the diagram that we drew when we introduced 2's complement back in Notes Set 1.2, as shown to the right. The adder is important enough to computer systems to merit its own symbol in logic diagrams, which is shown to the right with the inputs and outputs from our design added as labels. The text in the middle marking the symbol as an adder is only included for clarity: *any time you see a symbol of the shape shown to the right, it is an adder* (or sometimes a device that can add and do other operations). The width of the operand input and output lines then tells you the size of the adder.



You may already know that most computers have a **word size** specified as part of the Instruction Set Architecture. The word size specifies the number of bits in each operand when the computer adds two numbers, and is often used widely within the microarchitecture as well (for example, to decide the number of wires to use when moving bits around). Most desktop and laptop machines now have a word size of 64 bits, but many phone processors (and desktops/laptops a few years ago) use a 32-bit word size. Embedded microcontrollers may use a 16-bit or even an 8-bit word size.

Having seen how we can build an N-bit adder from simple chunks of logic operating on each pair of bits, you should not have much difficulty in understanding the diagram to the right. If we start with a design for an N-bit adder—even if that design is not built from bit slices, but is instead optimized for that particular size—we can create a $2N$-bit adder by simply connecting two copies of the N-bit adder. We give the adder for the less significant bits (the one on the right in the figure) an initial carry of 0, and pass the carry produced by the adder for the less significant bits into the carry input of the adder for the more significant bits. We calculate overflow based on the results of the adder for more significant bits (the one on the left in the figure), using the method appropriate to the type of operands we are adding (either unsigned or 2's complement).



You should also realize that this connection need not be physical. In other words, if a computer has an N-bit adder, it can handle operands with $2N$ bits (or $3N$, or $10N$, or $42N$) by using the N-bit adder repeatedly, starting with the least significant bits and working upward until all of the bits have been added. The computer must of course arrange to have the operands routed to the adder a few bits at a time, and must ensure that the carry produced by each addition is then delivered to the carry input (of the same adder!) for the next addition. In the coming months, you will learn how to design hardware that allows you to manage bits in this way, so that by the end of our class, you will be able to design a simple computer on your own.