

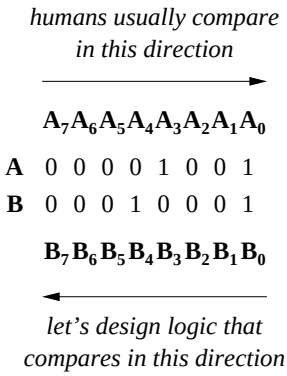
ECE199JL: Introduction to Computer Engineering **Fall 2012**
Notes Set 2.4

Example: Bit-Sliced Comparison

This set of notes develops comparators for unsigned and 2's complement numbers using the bit-sliced approach that we introduced in Notes Set 2.3. We then use algebraic manipulation and variation of the internal representation to illustrate design tradeoffs.

Comparing Two Numbers

Let's begin by thinking about how we as humans compare 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, so we can just think of them as binary numbers with leading 0s. We handle 2's complement values later in these notes.



As humans, we typically start comparing at the most significant bit. After all, if we find a difference in that bit, we are done, saving ourselves some time. In the example to the right, we know that $A < B$ as soon as we reach bit 4 and observe that $A_4 < B_4$. If we instead start from the least significant bit, we must always look at all of the bits.

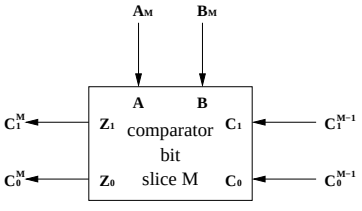
When building hardware to compare all of the bits at once, however, hardware for comparing each bit must exist, and the final result must be able to consider all of the bits. Our choice of direction should thus instead depend on how effectively we can build the corresponding functions. For a single bit slice, the two directions are almost identical. Let's develop a bit slice for comparing from least to most significant.

An Abstract Model

Comparison of two numbers, A and B, can produce three possible answers: $A < B$, $A = B$, or $A > B$ (one can also build an equality comparator that combines the $A < B$ and $A > B$ cases into a single answer).

As we move from bit to bit in our design, how much information needs to pass from one bit to the next? Here you may want to think about how you perform the task yourself. And perhaps to focus on the calculation for the most significant bit. You need to know the values of the two bits that you are comparing. If those two are not equal, you are done. But if the two bits are equal, what do you do? The answer is fairly simple: pass along the result from the less significant bits. Thus our bit slice logic for bit M needs to be able to accept three possible answers from the bit slice logic for bit $M - 1$ and must be able to pass one of three possible answers to the logic for bit $M + 1$. Since $\lceil \log_2(3) \rceil = 2$, we need two bits of input and two bits of output in addition to our input bits from numbers A and B.

The diagram to the right shows an abstract model of our comparator 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 these comparison bits using the bit number. The bit slice has C_1^{M-1} and C_0^{M-1} provided as inputs and produces C_1^M and C_0^M as outputs. Internally, we use C_1 and C_0 to denote these inputs, and Z_1 and Z_0 to denote the outputs. Similarly, the bits A_M and B_M from the numbers A and B are represented internally simply as A and B. 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.



A Representation and the First Bit

We need to select a representation for our three possible answers before we can design any logic. The representation chosen affects the implementation, as we discuss later in these notes. For now, we simply choose the representation to the right, which seems reasonable.

C_1	C_0	meaning
0	0	$A = B$
0	1	$A < B$
1	0	$A > B$
1	1	not used

Now we can design the logic for the first bit (bit 0). In keeping with the bit slice philosophy, in practice we simply use another copy of the full bit slice design for bit 0 and attach the C_1C_0 inputs to ground (to denote $A = B$). Here we tackle the simpler problem as a warm-up exercise.

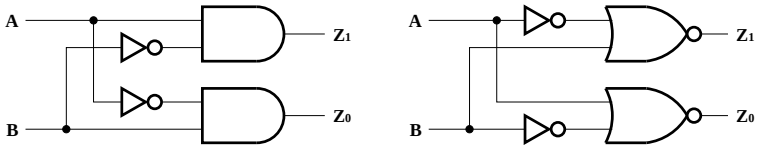
The truth table for bit 0 appears to the right (recall that we use Z_1 and Z_0 for the output names). Note that the bit 0 function has only two meaningful inputs—there is no bit to the right of bit 0. If the two inputs A and B are the same, we output equality. Otherwise, we do a 1-bit comparison and use our representation mapping to select the outputs. These functions are fairly straightforward to derive by inspection. They are:

A	B	Z_1	Z_0
0	0	0	0
0	1	0	1
1	0	1	0
1	1	0	0

$$Z_1 = A\,\overline{B}$$
$$Z_0 = \overline{A}\,B$$

These forms should also be intuitive, given the representation that we chose: $A > B$ if and only if $A = 1$ and $B = 0$; $A < B$ if and only if $A = 0$ and $B = 1$.

Implementation diagrams for our one-bit functions appear to the right. The diagram to the immediate right shows the implementation as we might initially draw it, and the diagram on the far right shows the implementation converted to NAND/NOR gates for a more accurate estimate of complexity when implemented in CMOS. The exercise of designing the logic for bit 0 is also useful in the sense that the logic structure illustrated forms the core of the full design in that it identifies the two cases that matter: $A < B$ and $A > B$.



Now we are ready to design the full function. Let's start by writing a full truth table, as shown on the left below.

A	B	C_1	C_0	Z_1	Z_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	x	x
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	x	x
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	x	x
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	1	0
1	1	1	1	x	x

A	B	C_1	C_0	Z_1	Z_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	1	0
x	x	1	1	x	x

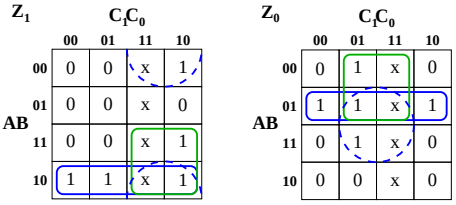
A	B	C_1	C_0	Z_1	Z_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	1	0
other				x	x

In the truth table, we marked the outputs as "don't care" (x's) whenever $C_1C_0 = 11$. You might recall that we ran into problems with our ice cream dispenser control in Notes Set 2.2. However, in that case we could not safely assume that a user did not push multiple buttons. Here, our bit slice logic only accepts inputs

from other copies of itself (or a fixed value for bit 0), and—assuming that we design the logic correctly—our bit slice never generates the 11 combination. In other words, that input combination is impossible (rather than undesirable or unlikely), so the result produced on the outputs is irrelevant.

It is tempting to shorten the full truth table by replacing groups of rows. For example, if $AB = 01$, we know that $A < B$, so the less significant bits (for which the result is represented by the C_1C_0 inputs) don't matter. We could write one row with input pattern $ABC_1C_0 = 01$xx and output pattern $Z_1Z_0 = 01$. We might also collapse our "don't care" output patterns: whenever the input matches $ABC_1C_0 =$xx11, we don't care about the output, so $Z_1Z_0 =$xx. But these two rows overlap in the input space! In other words, some input patterns, such as $ABC_1C_0 = 0111$, match both of our suggested new rows. Which output should take precedence? The answer is that a reader should not have to guess. *Do not use overlapping rows to shorten a truth table.* In fact, the first of the suggested new rows is not valid: we don't need to produce output 01 if we see $C_1C_0 = 11$. Two valid short forms of this truth table appear to the right of the full table. If you have an "other" entry, as shown in the rightmost table, this entry should always appear as the last row. Normal rows, including rows representing multiple input patterns, are not required to be in any particular order. Use whatever order makes the table easiest to read for its purpose (usually by treating the input pattern as a binary number and ordering rows in increasing numeric order).

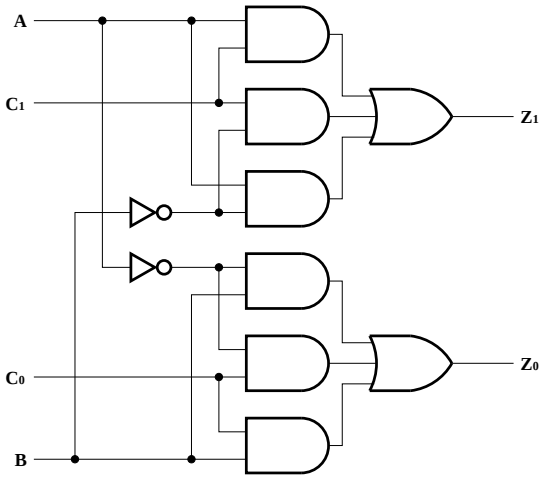
In order to translate our design into algebra, we transcribe the truth table into a K-map for each output variable, as shown to the right. You may want to perform this exercise yourself and check that you obtain the same solution. Implicants for each output are marked in the K-maps, giving the following equations:



$$Z_1 = A\,\overline{B} + A\,C_1 + \overline{B}\,C_1$$
$$Z_0 = \overline{A}\,B + \overline{A}\,C_0 + B\,C_0$$

An implementation based on our equations appears to the right. The figure makes it easy to see the symmetry between the inputs, which arises from the representation that we've chosen. Since the design only uses two-level logic (not counting the inverters on the A and B inputs, since inverters can be viewed as 1-input NAND or NOR gates), converting to NAND/NOR simply requires replacing all of the AND and OR gates with NAND gates.

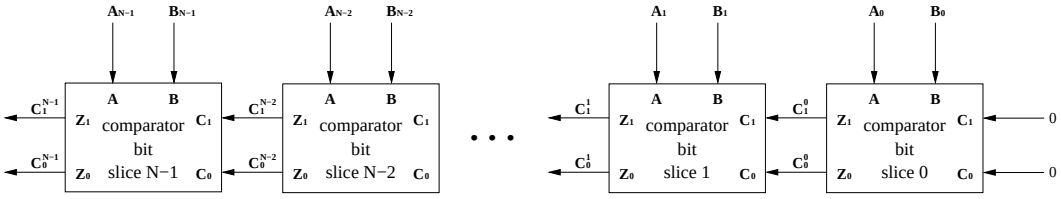
a comparator bit slice (first attempt)



Let's discuss the design's efficiency roughly 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. Our initial design uses two inverters, six 2-input gates, and two 3-input gates.

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, but, as we have discussed, the diagram above is equivalent on a gate-for-gate basis. Here we have three gate delays from the A and B inputs to the outputs (through the inverters). But when we connect multiple copies of our bit slice logic together to form a comparator, as shown on the next page, the delay from the A and B inputs to the outputs is not as important as the delay from the C_1 and C_0 inputs to the outputs. The latter delay adds to the total delay of our comparator on a per-bit-slice basis. Looking again at the diagram, notice that we have only two gate delays from the C_1 and C_0 inputs to the outputs. The total delay for an N-bit comparator based on this implementation is thus three gate delays for bit 0 and two more gate delays per additional bit, for a total of $2N + 1$ gate delays.

an N−bit unsigned comparator composed of bit slices



Optimizing Our Design

We have a fairly good design at this point—good enough for a homework or exam problem in this class, certainly—but let's consider how we might further optimize it. Today, optimization of logic at this level is done mostly by computer-aided design (CAD) tools, but we want you to be aware of the sources of optimization potential and the tradeoffs involved. And, if the topic interests you, someone has to continue to improve CAD software!

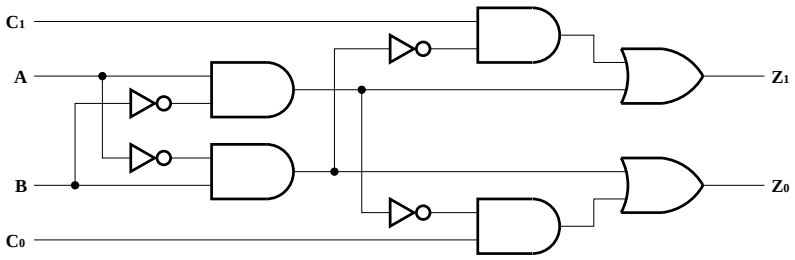
The first step is to manipulate our algebra to expose common terms that occur due to the design's symmetry. Starting with our original equation for Z_1, we have

$$\begin{aligned} Z_1 &= A\,\overline{B} + A\,C_1 + \overline{B}\,C_1 \\ &= A\,\overline{B} + \left(A + \overline{B}\right)\,C_1 \\ &= A\,\overline{B} + \overline{\overline{A}\,B}\,C_1 \\ \text{Similarly,}\quad Z_0 &= \overline{A}\,B + \overline{A\,\overline{B}}\,C_0 \end{aligned}$$

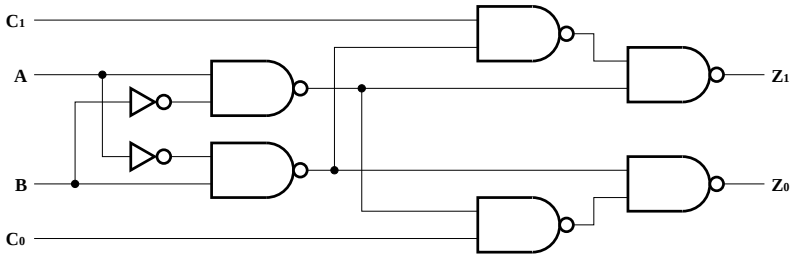
Notice that the second term in each equation now includes the complement of first term from the other equation. For example, the Z_1 equation includes the complement of the $\overline{A}B$ product that we need to compute Z_0. We may be able to improve our design by combining these computations.

An implementation based on our new algebraic formulation appears to the right. In this form, we seem to have kept the same number of gates, although we have replaced the 3-input gates with inverters. However, the middle inverters disappear when we convert to NAND/NOR form, as shown below to the right. Our new design requires only two inverters and six 2-input gates, a substantial reduction relative to the original implementation.

a comparator bit slice (optimized)



a comparator bit slice (optimized, NAND/NOR)



Is there a disadvantage? Yes, but only a slight one. Notice that the path from the A and B inputs to the outputs is now four gates (maximum) instead of three. Yet the path from C_1 and C_0 to the outputs is still only two gates. Thus, overall, we have merely increased our N-bit comparator's delay from $2N+1$ gate delays to $2N + 2$ gate delays.

Extending to 2's Complement

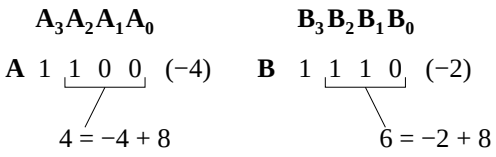
What about comparing 2's complement numbers? Can we make use of the unsigned comparator that we just designed?

Let's start by thinking about the sign of the numbers A and B. Recall that 2's complement records a number's sign in the most significant bit. For example, in the 8-bit numbers shown in the first diagram in this set of notes, the sign bits are A_7 and B_7. Let's denote these sign bits in the general case by A_s and B_s. Negative numbers have a sign bit equal to 1, and non-negative numbers have a sign bit equal to 0. The table below outlines an initial evaluation of the four possible combinations of sign bits.

A_s	B_s	interpretation	solution
0	0	$A \geq 0$ AND $B \geq 0$	use unsigned comparator on remaining bits
0	1	$A \geq 0$ AND $B < 0$	$A > B$
1	0	$A < 0$ AND $B \geq 0$	$A < B$
1	1	$A < 0$ AND $B < 0$	unknown

What should we do when both numbers are negative? Need we design a completely separate logic circuit? Can we somehow convert a negative value to a positive one?

The answer is in fact much simpler. Recall that 2's complement is defined based on modular arithmetic. Given an N-bit negative number A, the representation for the bits $A[N-2:0]$ is the same as the binary (unsigned) representation of $A + 2^{N-1}$. An example appears to the right.



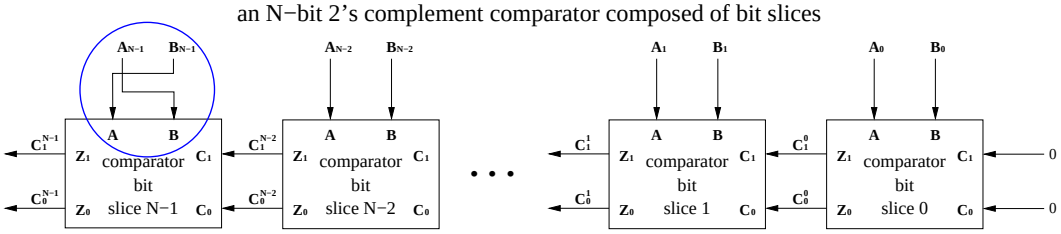
Let's define $A_r = A + 2^{N-1}$ as the value of the remaining bits for A and B_r similarly for B. What happens if we just go ahead and compare A_r and B_r using an $(N-1)$-bit unsigned comparator? If we find that $A_r < B_r$ we know that $A_r - 2^{N-1} < B_r - 2^{N-1}$ as well, but that means $A < B$! We can do the same with either of the other possible results. In other words, simply comparing A_r with B_r gives the correct answer for two negative numbers as well.

All we need to design is a logic block for the sign bits. At this point, we might write out a K-map, but instead let's rewrite our high-level table with the new information, as shown to the right.

A_s	B_s	solution
0	0	pass result from less significant bits
0	1	$A > B$
1	0	$A < B$
1	1	pass result from less significant bits

Looking at the table, notice the similarity to the high-level design for a single bit of an unsigned value. The only difference is that the two $A \neq B$ cases are reversed. If we swap A_s and B_s, the function is identical. We can simply use another bit slice but swap these two inputs. Implementation of an N-bit 2's complement comparator based on our bit slice comparator is shown below. The blue circle highlights the only change from the N-bit unsigned comparator, which is to swap the two inputs on the sign bit.

an N−bit 2's complement comparator composed of bit slices



Further Optimization

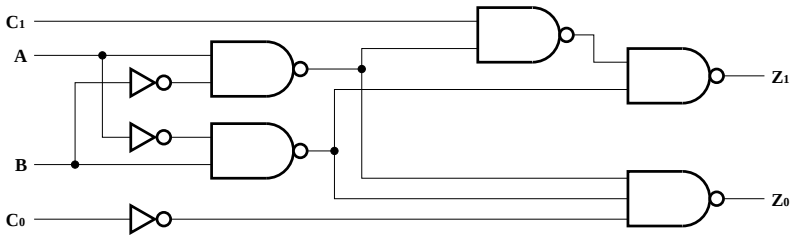
Let's return to the topic of optimization. To what extent did the representation of the three outcomes affect our ability to develop a good bit slice design? Although selecting a good representation can be quite important, for this particular problem most representations lead to similar implementations.

C_1	C_0	original	alternate
0	0	$A = B$	$A = B$
0	1	$A < B$	$A > B$
1	0	$A > B$	not used
1	1	not used	$A < B$

Some representations, however, have interesting properties. Consider the alternate representation on the right, for example (a copy of the original representation is included for comparison). Notice that in the alternate representation, $C_0 = 1$ whenever $A \neq B$. Once we have found the numbers to be different in some bit, the end result can never be equality, so perhaps with the right representation—the new one, for example—we might be able to cut delay in half?

An implementation based on the alternate representation appears in the diagram to the right. As you can see, in terms of gate count, this design replaces one 2-input gate with an inverter and a second 2-input gate with a 3-input gate. The path lengths are the same, requiring $2N+2$ gate delays for an N-bit comparator. Overall, it is about the same as our original design.

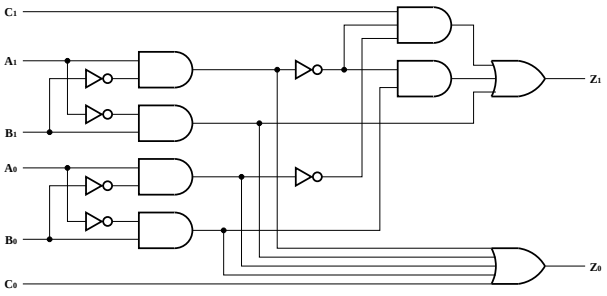
a comparator bit slice (alternate representation)



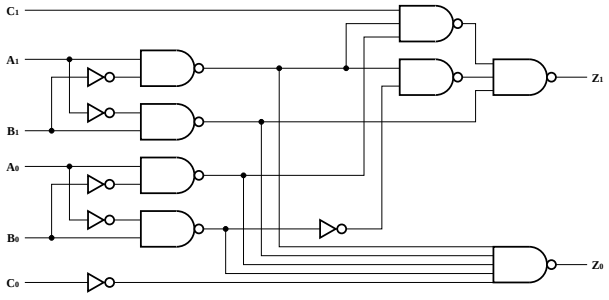
Why didn't it work? Should we consider still other representations? In fact, none of the possible representations that we might choose for a bit slice can cut the delay down to one gate delay per bit. The problem is fundamental, and is related to the nature of CMOS. For a single bit slice, we define the incoming and outgoing representations to be the same. We also need to have at least one gate in the path to combine the C_1 and C_0 inputs with information from the bit slice's A and B inputs. But all CMOS gates invert the sense of their inputs. Our choices are limited to NAND and NOR. Thus we need at least two gates in the path to maintain the same representation.

One simple answer is to use different representations for odd and even bits. Instead, we optimize a logic circuit for comparing two bits. We base our design on the alternate representation. The implementation is shown below. The left shows an implementation based on the algebra, and the right shows a NAND/NOR implementation. Estimating by gate count and number of inputs, the two-bit design doesn't save much over two single bit slices in terms of area. In terms of delay, however, we have only two gate delays from C_1 and C_0 to either output. The longest path from the A and B inputs to the outputs is five gate delays. Thus, for an N-bit comparator built with this design, the total delay is only $N + 3$ gate delays. But N has to be even.

a comparator 2−bit slice (alternate representation)



a comparator 2−bit slice (alternate representation, NAND/NOR)



As you can imagine, continuing to scale up the size of our logic block gives us better performance at the expense of a more complex design. Using the alternate representation may help you to see how one can generalize the approach to larger groups of bits—for example, you may have noticed the two bitwise comparator blocks on the left of the implementations above.