ECE199JL: Introduction to Computer Engineering Fall 2012
Notes Set 2.1

Optimizing Logic Expressions

The second part of the course covers digital design more deeply than does the textbook. The lecture notes will explain the additional material, and we will provide further examples in lectures and in discussion sections. Please let us know if you need further material for study.

In the last notes, we introduced Boolean logic operations and showed that with AND, OR, and NOT, we can express any Boolean function on any number of variables. Before you begin these notes, please read the first two sections in Chapter 3 of the textbook, which discuss the operation of **complementary metal-oxide semiconductor** (**CMOS**) transistors, illustrate how gates implementing the AND, OR, and NOT operations can be built using transistors, and introduce DeMorgan's laws.

This set of notes exposes you to a mix of techniques, terminology, tools, and philosophy. Some of the material is not critical to our class (and will not be tested), but is useful for your broader education, and may help you in later classes. The value of this material has changed substantially in the last couple of decades, and particularly in the last few years, as algorithms for tools that help with hardware design have undergone rapid advances. We talk about these issues as we introduce the ideas.

The notes begin with a discussion of the "best" way to express a Boolean function and some techniques used historically to evaluate such decisions. We next introduce the terminology necessary to understand manipulation of expressions, and use these terms to explain the Karnaugh map, or K-map, a tool that we will use for many purposes this semester. We illustrate the use of K-maps with a couple of examples, then touch on a few important questions and useful ways of thinking about Boolean logic. We conclude with a discussion of the general problem of multi-metric optimization, introducing some ideas and approaches of general use to engineers.

Defining Optimality

In the notes on logic operations, you learned how to express an arbitrary function on bits as an OR of minterms (ANDs with one input per variable on which the function operates). Although this approach demonstrates logical completeness, the results often seem inefficient, as you can see by comparing the following expressions for the carry out C from the addition of two 2-bit unsigned numbers, $A = A_1A_0$ and $B = B_1B_0$.

$$\begin{aligned}
C &= A_1B_1 + (A_1 + B_1)A_0B_0 && (1)\\
&= A_1B_1 + A_1A_0B_0 + A_0B_1B_0 && (2)\\
&= \overline{A_1}\ A_0\ B_1\ B_0 + A_1\ \overline{A_0}\ B_1\ \overline{B_0} + A_1\ \overline{A_0}\ B_1\ B_0 + \\
&\quad\ A_1\ A_0\ \overline{B_1}\ B_0 + A_1\ A_0\ B_1\ \overline{B_0} + A_1\ A_0\ B_1\ B_0 && (3)
\end{aligned}$$

These three expressions are identical in the sense that they have the same truth tables—they are the same mathematical function. Equation (1) is the form that we gave when we introduced the idea of using logic to calculate overflow. In this form, we were able to explain the terms intuitively. Equation (2) results from distributing the parenthesized OR in Equation (1). Equation (3) is the result of our logical completeness construction.

Since the functions are identical, does the form actually matter at all? Certainly either of the first two forms is easier for us to write than is the third. If we think of the form of an expression as a mapping from the function that we are trying to calculate into the AND, OR, and NOT functions that we use as logical building blocks, we might also say that the first two versions use fewer building blocks. That observation does have some truth, but let's try to be more precise by framing a question. For any given function, there are an infinite number of ways that we can express the function (for example, given one variable A on which the function depends, you can OR together any number of copies of $A\overline{A}$ without changing the function). *What exactly makes one expression better than another?*

In 1952, Edward Veitch wrote an article on simplifying truth functions. In the introduction, he said, "This general problem can be very complicated and difficult. Not only does the complexity increase greatly with the number of inputs and outputs, but the criteria of the best circuit will vary with the equipment involved." Sixty years later, the answer is largely the same: the criteria depend strongly on the underlying technology (the gates and the devices used to construct the gates), and no single **metric**, or way of measuring, is sufficient to capture the important differences between expressions in all cases.

Three high-level metrics commonly used to evaluate chip designs are cost, power, and performance. Cost usually represents the manufacturing cost, which is closely related to the physical silicon area required for the design: the larger the chip, the more expensive the chip is to produce. Power measures energy consumption over time. A chip that consumes more power means that a user's energy bill is higher and, in a portable device, either that the device is heavier or has a shorter battery life. Performance measures the speed at which the design operates. A faster design can offer more functionality, such as supporting the latest games, or can just finish the same work in less time than a slower design. These metrics are sometimes related: if a chip finishes its work, the chip can turn itself off, saving energy.

How do such high-level metrics relate to the problem at hand? Only indirectly in practice. There are too many factors involved to make direct calculations of cost, power, or performance at the level of logic expressions. Finding an **optimal** solution—the best formulation of a specific logic function for a given metric—is often impossible using the computational resources and algorithms available to us. Instead, tools typically use heuristic approaches to find solutions that strike a balance between these metrics. A **heuristic** approach is one that is believed to yield fairly good solutions to a problem, but does not necessarily find an optimal solution. A human engineer can typically impose **constraints**, such as limits on the chip area or limits on the minimum performance, in order to guide the process. Human engineers may also restructure the implementation of a larger design, such as a design to perform floating-point arithmetic, so as to change the logic functions used in the design.

Today, manipulation of logic expressions for the purposes of optimization is performed almost entirely by computers. Humans must supply the logic functions of interest, and must program the acceptable transformations between equivalent forms, but computers do the grunt work of comparing alternative formulations and deciding which one is best to use in context.

Although we believe that hand optimization of Boolean expressions is no longer an important skill for our graduates, we do think that you should be exposed to the ideas and metrics historically used for such optimization. The rationale for retaining this exposure is threefold. First, we believe that you still need to be able to perform basic logic reformulations (slowly is acceptable) and logical equivalence checking (answering the question, "Do two expressions represent the same function?"). Second, the complexity of the problem is a good way to introduce you to real engineering. Finally, the contextual information will help you to develop a better understanding of finite state machines and higher-level abstractions that form the core of digital systems and are still defined directly by humans today.

Towards that end, we conclude this introduction by discussing two metrics that engineers traditionally used to optimize logic expressions. These metrics are now embedded in **computer-aided design** (**CAD**) tools and tuned to specific underlying technologies, but the reasons for their use are still interesting.

The first metric of interest is a heuristic for the area needed for a design. The measurement is simple: count the number of variable occurrences in an expression. Simply go through and add up how many variables you see. Using our example function C, Equation (1) gives a count of 6, Equation (2) gives a count of 8, and Equation (3) gives a count of 24. Smaller numbers represent better expressions, so Equation (1) is the best choice by this metric. Why is this metric interesting? Recall how gates are built from transistors. An N-input gate requires roughly $2N$ transistors, so if you count up the number of variables in the expression, you get an estimate of the number of transistors needed, which is in turn an estimate for the area required for the design.

A variation on variable counting is to add the number of operations, since each gate also takes space for wiring (within as well as between gates). Note that we ignore the number of inputs to the operations, so a 2-input AND counts as 1, but a 10-input AND also counts as 1. We do not usually count complementing variables as an operation for this metric because the complements of variables are sometimes available at no extra cost in gates or wires. If we add the number of operations in our example, we get a count of 10 for Equation (1)—two ANDs, two ORs, and 6 variables, a count of 12 for Equation (2)—three ANDS, one OR, and 8 variables, and a count of 31 for Equation (3)—six ANDs, one OR, and 24 variables. The relative differences between these equations are reduced when one counts operations.

A second metric of interest is a heuristic for the performance of a design. Performance is inversely related to the delay necessary for a design to produce an output once its inputs are available. For example, if you know how many seconds it takes to produce a result, you can easily calculate the number of results that can be produced per second, which measures performance. The measurement needed is the longest chain of operations performed on any instance of a variable. The complement of a variable is included if the variable's complement is not available without using an inverter. The rationale for this metric is that gate outputs do not change instantaneously when their inputs change. Once an input to a gate has reached an appropriate voltage to represent a 0 or a 1, the transistors in the gate switch (on or off) and electrons start to move. Only when the output of the gate reaches the appropriate new voltage can the gates driven by the output start to change. If we count each function/gate as one delay (we call this time a **gate delay**), we get an estimate of the time needed to compute the function. Referring again to our example equations, we find that Equation (1) requires 3 gate delays, Equation (2) requires 2 gate delays, Equation (3) requires 2 or 3 gate delays, depending on whether we have variable complements available. Now Equation (2) looks more attractive: better performance than Equation (1) in return for a small extra cost in area.

Heuristics for estimating energy use are too complex to introduce at this point, but you should be aware that every time electrons move, they generate heat, so we might favor an expression that minimizes the number of bit transitions inside the computation. Such a measurement is not easy to calculate by hand, since you need to know the likelihood of input combinations.

Terminology

We use many technical terms when we talk about simplification of logic expressions, so we now introduce those terms so as to make the description of the tools and processes easier to understand.

Let's assume that we have a logic function $F(A, B, C, D)$ that we want to express concisely. A **literal** in an expression of F refers to either one of the variables or its complement. In other words, for our function F, the following is a complete set of literals: A, $\overline{A}$, B, $\overline{B}$, C, $\overline{C}$, D, and $\overline{D}$.

When we introduced the AND and OR functions, we also introduced notation borrowed from arithmetic, using multiplication to represent AND and addition to represent OR. We also borrow the related terminology, so a **sum** in Boolean algebra refers to a number of terms OR'd together (for example, $A + B$, or $AB + CD$), and a **product** in Boolean algebra refers to a number of terms AND'd together (for example, $A\overline{D}$, or $AB(C + D)$. Note that the terms in a sum or product may themselves be sums, products, or other types of expressions (for example, $A \oplus \overline{B}$).

The construction method that we used to demonstrate logical completeness made use of minterms for each input combination for which the function F produces a 1. We can now use the idea of a literal to give a simpler definition of minterm: a **minterm** for a function on N variables is a product (AND function) of N literals in which each variable or its complement appears exactly once. For our function F, examples of minterms include $ABC\overline{D}$, $A\overline{B}CD$, and $\overline{A}BC\overline{D}$. As you know, a minterm produces a 1 for exactly one combination of inputs.

When we sum minterms for each output value of 1 in a truth table to express a function, as we did to obtain Equation (3), we produce an example of the sum-of-products form. In particular, a **sum-of-products** (**SOP**) is a sum composed of products of literals. Terms in a sum-of-products need not be minterms, however. Equation (2) is also in sum-of-products form. Equation (1), however, is not, since the last term in the sum is not a product of literals.

Analogously to the idea of a minterm, we define a **maxterm** for a function on N variables as a sum (OR function) of N literals in which each variable or its complement appears exactly once. Examples for F

include $(A+B+\overline{C}+D)$, $(A+\overline{B}+\overline{C}+D)$, and $(\overline{A}+\overline{B}+C+\overline{D})$. A maxterm produces a 0 for exactly one combination of inputs. Just as we did with minterms, we can multiply a maxterm corresponding to each input combination for which a function produces 0 (each row in a truth table that produces a 0 output) to create an expression for the function. The resulting expression is in a **product-of-sums** (**POS**) form: a product of sums of literals. The carry out function that we used to produce Equation (3) has 10 input combinations that produce 0, so the expression formed in this way is unpleasantly long:

$$\begin{aligned} C &= (\overline{A_1}+\overline{A_0}+B_1+B_0)(\overline{A_1}+A_0+B_1+\overline{B_0})(\overline{A_1}+A_0+B_1+B_0)(A_1+\overline{A_0}+\overline{B_1}+B_0) \\ &\quad (A_1+\overline{A_0}+B_1+\overline{B_0})(A_1+\overline{A_0}+B_1+B_0)(A_1+A_0+\overline{B_1}+\overline{B_0})(A_1+A_0+\overline{B_1}+B_0) \\ &\quad (A_1+A_0+B_1+\overline{B_0})(A_1+A_0+B_1+B_0) \end{aligned}$$

However, the approach can be helpful with functions that produce mostly 1s. The literals in maxterms are complemented with respect to the literals used in minterms. For example, the maxterm $(\overline{A_1}+\overline{A_0}+B_1+B_0)$ in the equation above produces a zero for input combination $A_1=1$, $A_0=1$, $B_1=0$, $B_0=0$.

An **implicant** G of a function F is defined to be a second function operating on the same variables for which the implication $G \rightarrow F$ is true. In terms of logic functions that produce 0s and 1s, *if G is an implicant of F, the input combinations for which G produces 1s are a subset of the input combinations for which F produces 1s.* Any minterm for which F produces a 1, for example, is an implicant of F.

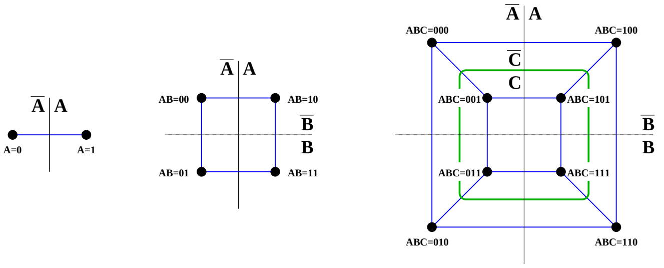
In the context of logic design, *the term implicant is used to refer to a single product of literals.* In other words, if we have a function $F(A,B,C,D)$, examples of possible implicants of F include AB, $B\overline{C}$, $ABC\overline{D}$, and $\overline{A}$. In contrast, although they may technically imply F, we typically do not call expressions such as $(A+B)$, $C(\overline{A}+D)$, nor $A\overline{B}+C$ implicants.

Let's say that we have expressed function F in sum-of-products form. All of the individual product terms in the expression are implicants of F. As a first step in simplification, we can ask: for each implicant, is it possible to remove any of the literals that make up the product? If we have an implicant G for which the answer is no, we call G a **prime implicant** of F. In other words, if one removes any of the literals from a prime implicant G of F, the resulting product is not an implicant of F.

Prime implicants are the main idea that we use to simplify logic expressions, both algebraically and with graphical tools (computer tools use algebra internally—by graphical here we mean drawings on paper).

Veitch Charts and Karnaugh Maps

Veitch's 1952 paper was the first to introduce the idea of using a graphical representation to simplify logic expressions. Earlier approaches were algebraic. A year later, Maurice Karnaugh published a paper showing a similar idea with a twist. The twist makes the use of **Karnaugh maps** to simplify expressions much easier than the use of Veitch charts. As a result, few engineers have heard of Veitch, but everyone who has ever taken a class on digital logic knows how to make use of a **K-map**.



Before we introduce the Karnaugh map, let's think about the structure of the domain of a logic function. Recall that a function's **domain** is the space on which the function is defined, that is, for which the function produces values. For a Boolean logic function on N variables, you can think of the domain as sequences of N bits, but you can also visualize the domain as an N-dimensional hypercube. An ***N*-dimensional hypercube** is the generalization of a cube to N dimensions. Some people only use the term hypercube when $N \geq 4$, since we have other names for the smaller values: a point for $N=0$, a line segment for $N=1$, a square for $N=2$, and a cube for $N=3$. The diagrams above and to the right illustrate the cases that are easily drawn on paper. The black dots represent specific input combinations, and the blue edges connect input combinations that differ in exactly one input value (one bit).

By viewing a function's domain in this way, we can make a connection between a product of literals and the structure of the domain. Let's use the 3-dimensional version as an example. We call the variables A, B, and C, and note that the cube has $2^3 = 8$ corners corresponding to the 2^3 possible combinations of A, B, and C. The simplest product of literals in this case is 1, which is the product of 0 literals. Obviously, the product 1 evaluates to 1 for any variable values. We can thus think of it as covering the entire domain of the function. In the case of our example, the product 1 covers the whole cube. In order for the product 1 to be an implicant of a function, the function itself must be the function 1.

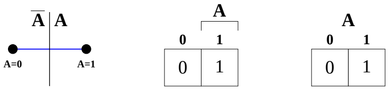
What about a product consisting of a single literal, such as A or $\overline{C}$? The dividing lines in the diagram illustrate the answer: any such product term evaluates to 1 on a face of the cube, which includes $2^2 = 4$ of the corners. If a function evaluates to 1 on any of the six faces of the cube, the corresponding product term (consisting of a single literal) is an implicant of the function.

Continuing with products of two literals, we see that any product of two literals, such as $A\overline{B}$ or $\overline{B}C$, corresponds to an edge of our 3-dimensional cube. The edge includes $2^1 = 2$ corners. And, if a function evaluates to 1 on any of the 12 edges of the cube, the corresponding product term (consisting of two literals) is an implicant of the function.

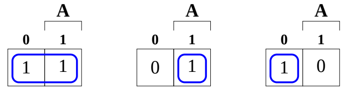
Finally, any product of three literals, such as $\overline{A}B\overline{C}$, corresponds to a corner of the cube. But for a function on three variables, these are just the minterms. As you know, if a function evaluates to 1 on any of the 8 corners of the cube, that minterm is an implicant of the function (we used this idea to construct the function to prove logical completeness).

How do these connections help us to simplify functions? If we're careful, we can map cubes onto paper in such a way that product terms (the possible implicants of the function) usually form contiguous groups of 1s, allowing us to spot them easily. Let's work upwards starting from one variable to see how this idea works. The end result is called a Karnaugh map.

The first drawing shown to the right replicates our view of the 1-dimensional hypercube, corresponding to the domain of a function on one variable, in this case the variable A. To the right of the hypercube (line segment) are two variants of a Karnaugh map on one variable. The middle variant clearly indicates the column corresponding to the product A (the other column corresponds to $\overline{A}$). The right variant simply labels the column with values for A.

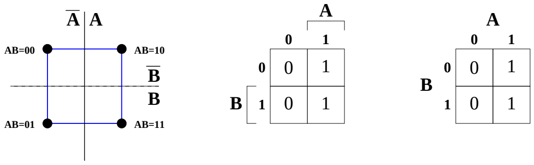


The three drawings shown to the right illustrate the three possible product terms on one variable. *The functions shown in these Karnaugh maps are arbitrary, except that we have chosen them such that each implicant shown is a prime implicant for the illustrated function.*

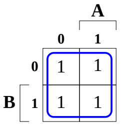


implicant: 1 implicant: A implicant: $\overline{A}$

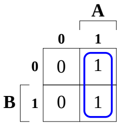
Let's now look at two-variable functions. We have replicated our drawing of the 2-dimensional hypercube (square) to the right along with two variants of Karnaugh maps on two variables. With only two variables (A and B), the extension is fairly straightforward, since we can use the second dimension of the paper (vertical) to express the second variable (B).



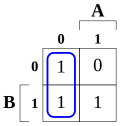
The number of possible products of literals grows rapidly with the number of variables. For two variables, nine are possible, as shown to the right. Notice that all implicants have two properties. First, they occupy contiguous regions of the grid. And, second, their height and width are always powers of two. These properties seem somewhat trivial at this stage, but they are the key to the utility of K-maps on more variables.



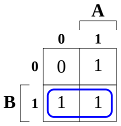
implicant: 1



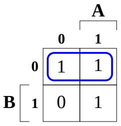
implicant: A



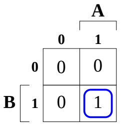
implicant: $\overline{A}$



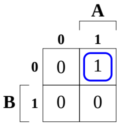
implicant: B



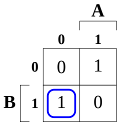
implicant: $\overline{B}$



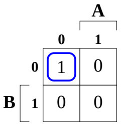
implicant: AB



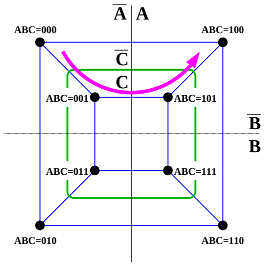
implicant: A$\overline{B}$

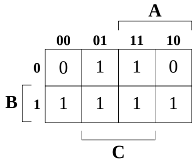


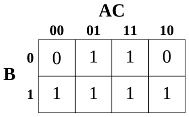
implicant: $\overline{A}$B



implicant: $\overline{A}\overline{B}$

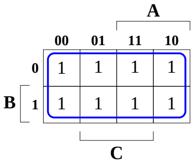




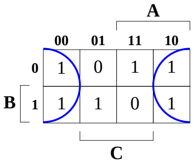


Three-variable functions are next. The cube diagram is again replicated to the right. But now we have a problem: how can we map four points (say, from the top half of the cube) into a line in such a way that any points connected by a blue line are adjacent in the K-map? The answer is that we cannot, but we can preserve most of the connections by choosing an order such as the one illustrated by the arrow. The result is called a Gray code. Two K-map variants again appear to the right of the cube. Look closely at the order of the two-variable combinations along the top, which allows us to have as many contiguous products of literals as possible. Any product of literals that contains $\overline{C}$ but not A nor $\overline{A}$ wraps around the edges of the K-map, so you should think of it as rolling up into a cylinder rather than a grid. Or you can think that we're unfolding the cube to fit the corners onto a sheet of paper, but the place that we split the cube should still be considered to be adjacent when looking for implicants. The use of a Gray code is the one difference between a K-map and a Veitch chart; Veitch used the base 2 order, which makes some implicants hard to spot.

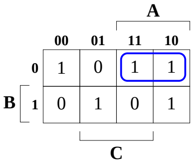
With three variables, we have 27 possible products of literals. You may have noticed that the count scales as 3^N for N variables; can you explain why? We illustrate several product terms below. Note that we sometimes need to wrap around the end of the K-map, but that if we account for wrapping, the squares covered by all product terms are contiguous. Also notice that both the width and the height of all product terms are powers of two. *Any square or rectangle that meets these two constraints corresponds to a product term!* And any such square or rectangle that is filled with 1s is an implicant of the function in the K-map.



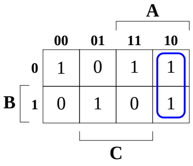
implicant: 1



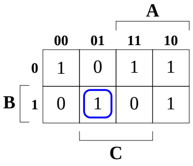
implicant: $\overline{C}$



implicant: A$\overline{B}$

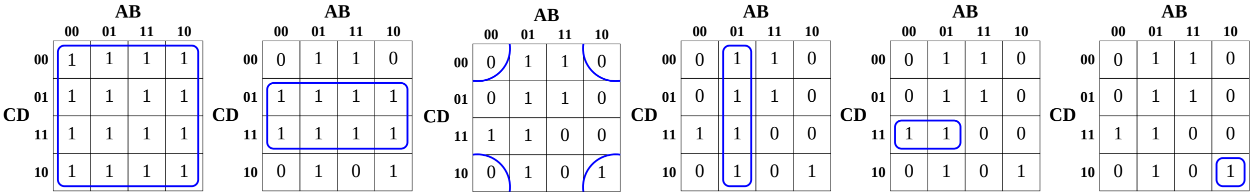


implicant: A$\overline{C}$



implicant: $\overline{A}$BC

Let's keep going. With a function on four variables—A, B, C, and D—we can use a Gray code order on two of the variables in each dimension. Which variables go with which dimension in the grid really doesn't matter, so we'll assign AB to the horizontal dimension and CD to the vertical dimension. A few of the 81 possible product terms are illustrated at the top of the next page. Notice that while wrapping can now occur in both dimensions, we have exactly the same rule for finding implicants of the function: any square or rectangle (allowing for wrapping) that is filled with 1s and has both height and width equal to (possibly different) powers of two is an implicant of the function. *Furthermore, unless such a square or rectangle is part of a larger square or rectangle that meets these criteria, the corresponding implicant is a prime implicant of the function.*

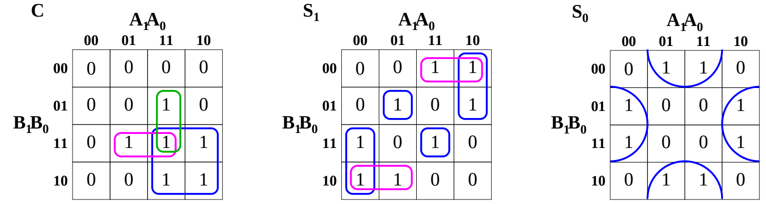


implicant: 1 implicant: D implicant: $\overline{B}\,\overline{D}$ implicant: $\overline{A}B$ implicant: $\overline{A}CD$ implicant: $A\overline{B}C\overline{D}$

Finding a simple expression for a function using a K-map then consists of solving the following problem: pick a minimal set of prime implicants such that every 1 produced by the function is covered by at least one prime implicant. The metric that you choose to minimize the set may vary in practice, but for simplicity, let's say that we minimize the number of prime implicants chosen.

Let's try a few! The table on the left below reproduces (from Notes Set 1.4) the truth table for addition of two 2-bit unsigned numbers, A_1A_0 and B_1B_0, to produce a sum S_1S_0 and a carry out C. K-maps for each output bit appear to the right. The colors are used only to make the different prime implicants easier to distinguish. The equations produced by summing these prime implicants appear below the K-maps.

inputs				outputs		
A_1	A_0	B_1	B_0	C	S_1	S_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0



$$C = A_1\ B_1 + A_1\ A_0\ B_0 + A_0\ B_1\ B_0$$

$$\begin{aligned} S_1 = {} & A_1\ \overline{B_1}\ \overline{B_0} + A_1\ \overline{A_0}\ \overline{B_1} + \overline{A_1}\ \overline{A_0}\ B_1 + \overline{A_1}\ B_1\ \overline{B_0} + \\ & \overline{A_1}\ A_0\ \overline{B_1}\ B_0 + A_1\ A_0\ B_1\ B_0 \end{aligned}$$

$$S_0 = A_0\ \overline{B_0} + \overline{A_0}\ B_0$$

In theory, K-maps extend to an arbitrary number of variables. Certainly Gray codes can be extended. An N**-bit Gray code** is a sequence of N-bit patterns that includes all possible patterns such that any two adjacent patterns differ in only one bit. The code is actually a cycle: the first and last patterns also differ in only one bit. You can construct a Gray code recursively as follows: for an $(N+1)$-bit Gray code, write the sequence for an N-bit Gray code, then add a 0 in front of all patterns. After this sequence, append a second copy of the N-bit Gray code in reverse order, then put a 1 in front of all patterns in the second copy. The result is an $(N+1)$-bit Gray code. For example, the following are Gray codes:

1-bit	0, 1
2-bit	00, 01, 11, 10
3-bit	000, 001, 011, 010, 110, 111, 101, 100
4-bit	0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000

Unfortunately, some of the beneficial properties of K-maps do not extend beyond two variables in a dimension. *Once you have three variables in one dimension*, as is necessary if a function operates on five or more variables, *not all product terms are contiguous in the grid.* The terms still require a total number of rows and columns equal to a power of two, but they don't all need to be a contiguous group. Furthermore, *some contiguous groups of appropriate size do not correspond to product terms.* So you can still make use of K-maps if you have more variables, but their use is a little trickier.

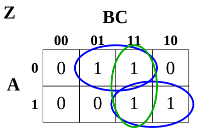
Canonical Forms

What if we want to compare two expressions to determine whether they represent the same logic function? Such a comparison is a test of **logical equivalence**, and is an important part of hardware design. Tools today provide help with this problem, but you should understand the problem.

You know that any given function can be expressed in many ways, and that two expressions that look quite different may in fact represent the same function (look back at Equations (1) to (3) for an example). But what if we rewrite the function using only prime implicants? Is the result unique? Unfortunately, no.

In general, *a sum of products is not unique (nor is a product of sums), even if the sum contains only prime implicants.*

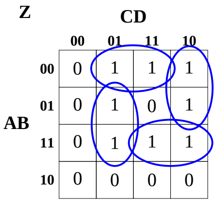
For example, consensus terms may or may not be included in our expressions. (They are necessary for reliable design of certain types of systems, as you will learn in a later ECE class.) The green ellipse in the K-map to the right represents the consensus term BC.



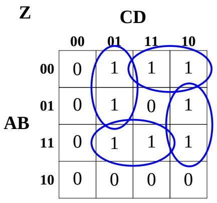
A \ BC	00	01	11	10
0	0	1	1	0
1	0	0	1	1

$$Z = A\,C + \overline{A}\,B + B\,C$$
$$Z = A\,C + \overline{A}\,B$$

Some functions allow several equivalent formulations as sums of prime implicants, even without consensus terms. The K-maps shown to the right, for example, illustrate how one function might be written in either of the following ways:



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



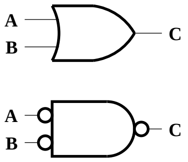
AB \ CD	00	01	11	10
00	0	1	1	1
01	0	1	0	1
11	0	1	1	1
10	0	0	0	0

$$Z = \overline{A}\,\overline{B}\,D + \overline{A}\,C\,\overline{D} + A\,B\,C + B\,\overline{C}\,D$$
$$Z = \overline{A}\,\overline{B}\,C + B\,C\,\overline{D} + A\,B\,D + \overline{A}\,\overline{C}\,D$$

When we need to compare two things (such as functions), we need to transform them into what in mathematics is known as a **canonical form**, which simply means a form that is defined so as to be unique for each thing of the given type. What can we use for logic functions? You already know two answers! The **canonical sum** of a function (sometimes called the **canonical SOP form**) is the sum of minterms. The **canonical product** of a function (sometimes called the **canonical POS form**) is the product of maxterms. These forms technically only meet the mathematical definition of canonical if we agree on an order for the min/maxterms, but that problem is solvable. However, as you already know, the forms are not particularly convenient to use. In practice, people and tools in the industry use more compact approaches when comparing functions, but those solutions are a subject for a later class (such as ECE 462).

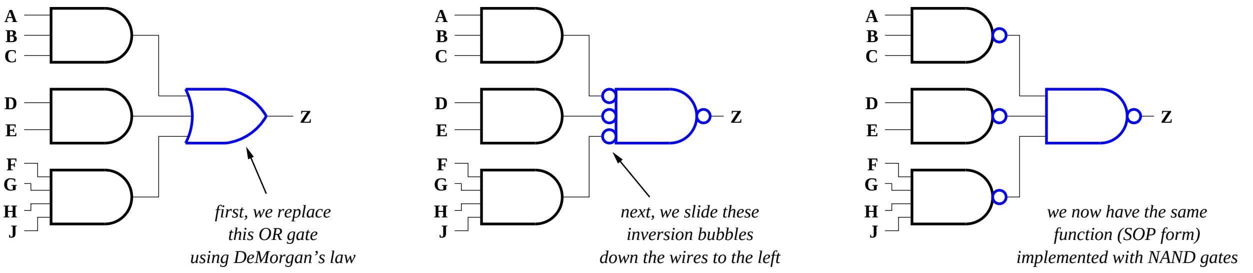
Two-Level Logic

Two-level logic is a popular way of expressing logic functions. The two levels refer simply to the number of functions through which an input passes to reach an output, and both the SOP and POS forms are examples of two-level logic. In this section, we illustrate one of the reasons for this popularity and show you how to graphically manipulate expressions, which can sometimes help when trying to understand gate diagrams.



We begin with one of DeMorgan's laws, which we can illustrate both algebraically and graphically: $C = B + A = \overline{\overline{B}\,\overline{A}}$

Let's say that we have a function expressed in SOP form, such as $Z = ABC + DE + FGHJ$. The diagram on the left below shows the function constructed from three AND gates and an OR gate. Using DeMorgan's law, we can replace the OR gate with a NAND with inverted inputs. But the bubbles that correspond to inversion do not need to sit at the input to the gate. We can invert at any point along the wire, so we slide each bubble down the wire to the output of the first column of AND gates. *Be careful: if the wire splits, which does not happen in our example, you have to replicate the inverter onto the other output paths as you slide past the split point!* The end result is shown on the right: we have not changed the function, but now we use only NAND gates. Since CMOS technology only supports NAND and NOR directly, using two-level logic makes it simple to map our expression into CMOS gates.

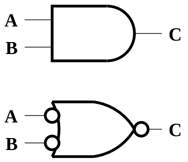


first, we replace this OR gate using DeMorgan's law

next, we slide these inversion bubbles down the wires to the left

we now have the same function (SOP form) implemented with NAND gates

You may want to make use of DeMorgan's other law, illustrated graphically to the right, to perform the same transformation on a POS expression. What do you get?



Multi-Metric Optimization

As engineers, almost every real problem that you encounter will admit multiple metrics for evaluating possible designs. Becoming a good engineer thus requires not only that you be able to solve problems creatively so as to improve the quality of your solutions, but also that you are aware of how people might evaluate those solutions and are able both to identify the most important metrics and to balance your design effectively according to them. In this section, we introduce some general ideas and methods that may be of use to you in this regard. *We will not test you on the concepts in this section.*

When you start thinking about a new problem, your first step should be to think carefully about metrics of possible interest. Some important metrics may not be easy to quantify. For example, compatibility of a design with other products already owned by a customer has frequently defined the success or failure of computer hardware and software solutions. But how can you compute the compability of your approach as a number?

Humans—including engineers—are not good at comparing multiple metrics simultaneously. Thus, once you have a set of metrics that you feel is complete, your next step is to get rid of as many as you can. Towards this end, you may identify metrics that have no practical impact in current technology, set threshold values for other metrics to simplify reasoning about them, eliminate redundant metrics, calculate linear sums to reduce the count of metrics, and, finally, make use of the notion of Pareto optimality. All of these ideas are described in the rest of this section.

Let's start by considering metrics that we can quantify as real numbers. For a given metric, we can divide possible measurement values into three ranges. In the first range, all measurement values are equivalently useful. In the second range, possible values are ordered and interesting with respect to one another. Values in the third range are all impossible to use in practice. Using power consumption as our example, the first range corresponds to systems in which when a processor's power consumption in a digital system is extremely low relative to the power consumption of the system. For example, the processor in a computer might use less than 1% of the total used by the system including the disk drive, the monitor, the power supply, and so forth. One power consumption value in this range is just as good as any another, and no one cares about the power consumption of the processor in such cases. In the second range, power consumption of the processor makes a difference. Cell phones use most of their energy in radio operation, for example, but if you own a phone with a powerful processor, you may have noticed that you can turn off the phone and drain the battery fairly quickly by playing a game. Designing a processor that uses half as much power lengthens the battery life in such cases. Finally, the third region of power consumption measurements is impossible: if you use so much power, your chip will overheat or even burst into flames. Consumers get unhappy when such things happen.

As a first step, you can remove any metrics for which all solutions are effectively equivalent. Until a little less than a decade ago, for example, the power consumption of a desktop processor actually was in the first range that we discussed. Power was simply not a concern to engineers: all designs of interest consumed so little power that no one cared. Unfortunately, at that point, power consumption jumped into the third range rather quickly. Processors hit a wall, and products had to be cancelled. Given that the time spent designing a processor has historically been about five years, a lot of engineering effort was wasted because people had not thought carefully enough about power (since it had never mattered in the past). Today, power is an important metric that engineers must take into account in their designs.

However, in some areas, such as desktop and high-end server processors, other metrics (such as performance) may be so important that we always want to operate at the edge of the interesting range. In such cases, we might choose to treat a metric such as power consumption as a **threshold**: stay below 150 Watts for a desktop processor, for example. One still has to make a coordinated effort to ensure that the system as a whole does not exceed the threshold, but reasoning about threshold values, a form of constraint, is easier than trying to think about multiple metrics at once.

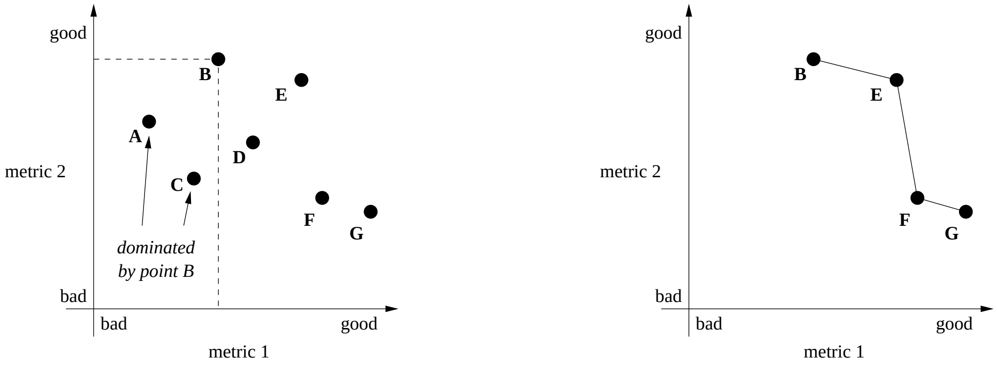
Some metrics may only allow discrete quantification. For example, one could choose to define compatibility with previous processor generations as binary: either an existing piece of software (or operating system) runs out of the box on your new processor, or it does not. If you want people who own that software to make use of your new processor, you must ensure that the value of this binary metric is 1, which can also be viewed as a threshold.

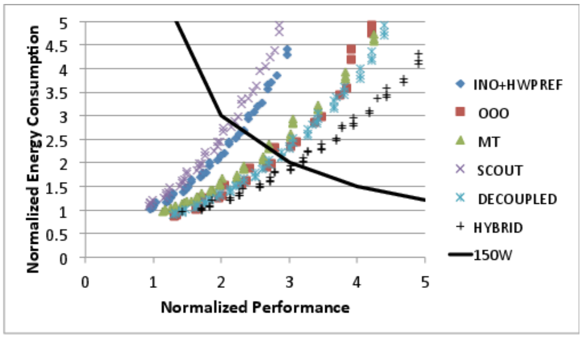
In some cases, two metrics may be strongly **correlated**, meaning that a design that is good for one of the metrics is frequently good for the other metric as well. Chip area and cost, for example, are technically distinct ways to measure a digital design, but we rarely consider them separately. A design that requires a larger chip is probably more complex, and thus takes more engineering time to get right (engineering time costs money). Each silicon wafer costs money to fabricate, and fewer copies of a large design fit on one wafer, so large chips mean more fabrication cost. Physical defects in silicon can cause some chips not to work. A large chip uses more silicon than a small one, and is thus more likely to suffer from defects (and not work). Cost thus goes up again for large chips relative to small ones. Finally, large chips usually require more careful testing to ensure that they work properly (even ignoring the cost of getting the design right, we have to test for the presence of defects), which adds still more cost for a larger chip. All of these factors tend to correlate chip area and chip cost, to the point that most engineers do not consider both metrics.

After you have tried to reduce your set of metrics as much as possible, or simplified them by turning them into thresholds, you should consider turning the last few metrics into a weighted linear sum. All remaining metrics must be quantifiable in this case. For example, if you are left with three metrics for which a given design has values A, B, and C, you might reduce these to one metric by calculating $D = w_A A + w_B B + w_C C$. What are the w values? They are weights for the three metrics. Their values represent the relative importance of the three metrics to the overall evaluation. Here we've assumed that larger values of A, B, and C are either all good or all bad. If you have metrics with different senses, use the reciprocal values. For example, if a large value of A is good, a small value of $1/A$ is also good.

The difficulty with linearizing metrics is that not everyone agrees on the weights. Is using less power more important than having a cheaper chip? The answer may depend on many factors.

When you are left with several metrics of interest, you can use the idea of Pareto optimality to identify interesting designs. Let's say that you have two metrics. If a design D_1 is better than a second design D_2 for both metrics, we say that D_1 **dominates** D_2. A design D is then said to be **Pareto optimal** if no other design dominates D. Consider the figure on the left below, which illustrates seven possible designs measured with two metrics. The design corresponding to point B dominates the designs corresponding to points A and C, so neither of the latter designs is Pareto optimal. No other point in the figure dominates B, however, so that design is Pareto optimal. If we remove all points that do not represent Pareto optimal designs, and instead include only those designs that are Pareto optimal, we obtain the version shown on the right. These are points in a two-dimensional space, not a line, but we can imagine a line going through the points, as illustrated in the figure: the points that make up the line are called a **Pareto curve**, or, if you have more than two metrics, a **Pareto surface**.





As an example of the use of Pareto optimality, consider the figure to the right, which is copied with permission from Neal Crago's Ph.D. dissertation (UIUC ECE, 2012). The figure compares hundreds of thousands of possible designs based on a handful of different core approaches for implementing a processor. The axes in the graph are two metrics of interest. The horizontal axis measures the average performance of a design when executing a set of benchmark applications, normalized to a baseline processor design. The vertical axis measures the energy consumed by a design when executing the same benchmarks, normalized again to the energy consumed by a baseline design. The six sets of points in the graph represent alternative design techniques for the processor, most of which are in commercial use today. The points shown for each set are the subset of many thousands of possible variants that are Pareto optimal. In this case, more performance and less energy consumption are the good directions, so any point in a set for which another point is both further to the right and further down is not shown in the graph. The black line represents an absolute power consumption of 150 Watts, which is a nominal threshold for a desktop environment. Designs above and to the right of that line are not as interesting for desktop use. The **design-space exploration** that Neal reported in this figure was of course done by many computers using many hours of computation, but he had to design the process by which the computers calculated each of the points.

ECE199JL: Introduction to Computer Engineering **Fall 2012**
Notes Set 2.2

Boolean Properties and Don't Care Simplification

This set of notes begins with a brief illustration of a few properties of Boolean logic, which may be of use to you in manipulating algebraic expressions and in identifying equivalent logic functions without resorting to truth tables. We then discuss the value of underspecifying a logic function so as to allow for selection of the simplest possible implementation. This technique must be used carefully to avoid incorrect behavior, so we illustrate the possibility of misuse with an example, then talk about several ways of solving the example correctly. We conclude by generalizing the ideas in the example to several important application areas and talking about related problems.

Logic Properties

Table 1 (on the next page) lists a number of properties of Boolean logic. Most of these are easy to derive from our earlier definitions, but a few may be surprising to you. In particular, in the algebra of real numbers, multiplication distributes over addition, but addition does not distribute over multiplication. For example, $3 \times (4 + 7) = (3 \times 4) + (3 \times 7)$, but $3 + (4 \times 7) \neq (3 + 4) \times (3 + 7)$. In Boolean algebra, both operators distribute over one another, as indicated in Table 1. The consensus properties may also be nonintuitive. Drawing a K-map may help you understand the consensus property on the right side of the table. For the consensus variant on the left side of the table, consider that since either A or $\overline{A}$ must be 0, either B or C or both must be 1 for the first two factors on the left to be 1 when ANDed together. But in that case, the third factor is also 1, and is thus redundant.

As mentioned previously, Boolean algebra has an elegant symmetry known as a duality, in which any logic statement (an expression or an equation) is related to a second logic statement. To calculate the **dual form** of a Boolean expression or equation, replace 0 with 1, replace 1 with 0, replace AND with OR, and replace OR with AND. *Variables are not changed when finding the dual form.* The dual form of a dual form is the original logic statement. Be careful when calculating a dual form: our convention for ordering arithmetic operations is broken by the exchange, so you may want to add explicit parentheses before calculating the dual. For example, the dual of $AB + C$ is not $A + BC$. Rather, the dual of $AB + C$ is $(A + B)C$. *Add parentheses as necessary when calculating a dual form to ensure that the order of operations does not change.*

Duality has several useful practical applications. First, the **principle of duality** states that any theorem or identity has the same truth value in dual form (we do not prove the principle here). The rows of Table 1 are organized according to this principle: each row contains two equations that are the duals of one another. Second, the dual form is useful when designing certain types of logic, such as the networks of transistors connecting the output of a CMOS gate to high voltage and ground. If you look at the gate designs in the textbook (and particularly those in the exercises), you will notice that these networks are duals. A function/expression is not a theorem nor an identity, thus the principle of duality does not apply to the dual of an expression. However, if you treat the value 0 as "true," the dual form of an expression has the same truth values as the original (operating with value 1 as "true"). Finally, you can calculate the complement of a Boolean function (any expression) by calculating the dual form and then complementing each variable.

Choosing the Best Function

When we specify how something works using a human language, we leave out details. Sometimes we do so deliberately, assuming that a reader or listener can provide the details themselves: "Take me to the airport!" rather than "Please bend your right arm at the elbow and shift your right upper arm forward so as to place your hand near the ignition key. Next, ..."

You know the basic technique for implementing a Boolean function using **combinational logic**: use a K-map to identify a reasonable SOP or POS form, draw the resulting design, and perhaps convert to NAND/NOR gates.

$1 + A = 1$	$0 \cdot A = 0$	
$1 \cdot A = A$	$0 + A = A$	
$A + A = A$	$A \cdot A = A$	
$A \cdot \overline{A} = 0$	$A + \overline{A} = 1$	
$\overline{A + B} = \overline{A}\ \overline{B}$	$\overline{AB} = \overline{A} + \overline{B}$	DeMorgan's laws
$(A + B)C = AC + BC$	$A\ B + C = (A + C)(B + C)$	distribution
$(A + B)(\overline{A} + C)(B + C) = (A + B)(\overline{A} + C)$	$A\ B + \overline{A}\ C + B\ C = A\ B + \overline{A}\ C$	consensus

Table 1: Boolean logic properties. The two columns are dual forms of one another.

When we develop combinational logic designs, we may also choose to leave some aspects unspecified. In particular, the value of a Boolean logic function to be implemented may not matter for some input combinations. If we express the function as a truth table, we may choose to mark the function's value for some input combinations as "**don't care**," which is written as "x" (no quotes).

What is the benefit of using "don't care" values? Using "don't care" values allows you to choose from among several possible logic functions, all of which produce the desired results (as well as some combination of 0s and 1s in place of the "don't care" values). Each input combination marked as "don't care" doubles the number of functions that can be chosen to implement the design, often enabling the logic needed for implementation to be simpler.

For example, the K-map to the right specifies a function $F(A, B, C)$ with two "don't care" entries. If you are asked to design combinational logic for this function, you can choose any values for the two "don't care" entries. When identifying prime implicants, each "x" can either be a 0 or a 1.

F		AB			
		00	01	11	10
C	0	0	1	x	x
	1	0	1	1	0

Depending on the choices made for the x's, we obtain one of the following four functions:

$$\begin{aligned} F &= \overline{A}\ B + B\ C \\ F &= \overline{A}\ B + B\ C + A\ \overline{B}\ \overline{C} \\ F &= B \\ F &= B + A\ \overline{C} \end{aligned}$$

F		AB			
		00	01	11	10
C	0	0	1	*1*	*0*
	1	0	1	1	0

Given this set of choices, a designer typically chooses the third: $F = B$, which corresponds to the K-map shown to the right of the equations. The design then produces $F = 1$ when $A = 1$, $B = 1$, and $C = 0$ ($ABC = 110$), and produces $F = 0$ when $A = 1$, $B = 0$, and $C = 0$ ($ABC = 100$). These differences are marked with shading and green italics in the new K-map. No implementation ever produces an "x."

Caring about Don't Cares

What can go wrong? In the context of a digital system, unspecified details may or may not be important. However, *any implementation of a specification implies decisions* about these details, so decisions should only be left unspecified if any of the possible answers is indeed acceptable.

As a concrete example, let's design logic to control an ice cream dispenser. The dispenser has two flavors, lychee and mango, but also allows us to create a blend of the two flavors. For each of the two flavors, our logic must output two bits to control the amount of ice cream that comes out of the dispenser. The two-bit $C_L[1:0]$ output of our logic must specify the number of half-servings of lychee ice cream as a binary number, and the two-bit $C_M[1:0]$ output must specify the number of half-servings of mango ice cream. Thus, for either flavor, 00 indicates none of that flavor, 01 indicates one-half of a serving, and 10 indicates a full serving.

Inputs to our logic will consist of three buttons: an L button to request a serving of lychee ice cream, a B button to request a blend—half a serving of each flavor, and an M button to request a serving of mango ice cream. Each button produces a 1 when pressed and a 0 when not pressed.

Let's start with the assumption that the user only presses one button at a time. In this case, we can treat input combinations in which more than one button is pressed as "don't care" values in the truth tables for the outputs. K-maps for all four output bits appear below. The x's indicate "don't care" values.

$C_L[1]$

M \ LB	00	01	11	10
0	0	0	x	1
1	0	x	x	x

$C_L[0]$

M \ LB	00	01	11	10
0	0	1	x	0
1	0	x	x	x

$C_M[1]$

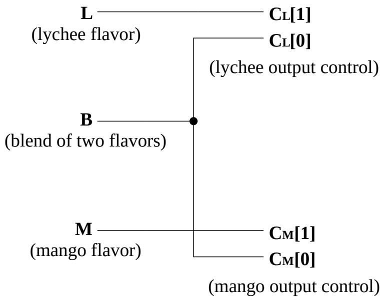
M \ LB	00	01	11	10
0	0	0	x	0
1	1	x	x	x

$C_M[0]$

M \ LB	00	01	11	10
0	0	1	x	0
1	0	x	x	x

When we calculate the logic function for an output, each "don't care" value can be treated as either 0 or 1, whichever is more convenient in terms of creating the logic. In the case of $C_M[1]$, for example, we can treat the three x's in the ellipse as 1s, treat the x outside of the ellipse as a 0, and simply use M (the implicant represented by the ellipse) for $C_M[1]$. The other three output bits are left as an exercise, although the result appears momentarily.

The implementation at right takes full advantage of the "don't care" parts of our specification. In this case, we require no logic at all; we need merely connect the inputs to the correct outputs. Let's verify the operation. We have four cases to consider. First, if none of the buttons are pushed ($LBM = 000$), we get no ice cream, as desired ($C_M = 00$ and $C_L = 00$). Second, if we request lychee ice cream ($LBM = 100$), the outputs are $C_L = 10$ and $C_M = 00$, so we get a full serving of lychee and no mango. Third, if we request a blend ($LBM = 010$), the outputs are $C_L = 01$ and $C_M = 01$, giving us half a serving of each flavor. Finally, if we request mango ice cream ($LBM = 001$), we get no lychee but a full serving of mango.



The K-maps for this implementation appear below. Each of the "don't care" x's from the original design has been replaced with either a 0 or a 1 and highlighted with shading and green italics. Any implementation produces either 0 or 1 for every output bit for every possible input combination.

$C_L[1]$

M \ LB	00	01	11	10
0	0	0	*1*	1
1	0	*0*	*1*	*1*

$C_L[0]$

M \ LB	00	01	11	10
0	0	1	*1*	0
1	0	*1*	*1*	*0*

$C_M[1]$

M \ LB	00	01	11	10
0	0	0	*0*	0
1	1	*1*	*1*	*1*

$C_M[0]$

M \ LB	00	01	11	10
0	0	1	*1*	0
1	0	*1*	*1*	*0*

As you can see, leveraging "don't care" output bits can sometimes significantly simplify our logic. In the case of this example, we were able to completely eliminate any need for gates! Unfortunately, the resulting implementation may sometimes produce unexpected results. Based on the implementation, what happens if a user presses more than one button? The ice cream cup overflows!

Let's see why. Consider the case $LBM = 101$, in which we've pressed both the lychee and mango buttons. Here $C_L = 10$ and $C_M = 10$, so our dispenser releases a full serving of each flavor, or two servings total. Pressing other combinations may have other repercussions as well. Consider pressing lychee and blend ($LBM = 110$). The outputs are then $C_L = 11$ and $C_M = 01$. Hopefully the dispenser simply gives us one and a half servings of lychee and a half serving of mango. However, if the person who designed the dispenser assumed that no one would ever ask for more than one serving, something worse might happen. In other words, giving an input of $C_L = 11$ to the ice cream dispenser may lead to other unexpected behavior if its designer decided that that input pattern was a "don't care."

The root of the problem is that *while we don't care about the value of any particular output marked "x" for any particular input combination, we do actually care about the relationship between the outputs.*

What can we do? When in doubt, it is safest to make choices and to add the new decisions to the specification rather than leaving output values specified as "don't care." For our ice cream dispenser logic, rather than leaving the outputs unspecified whenever a user presses more than one button, we could choose an acceptable outcome for each input combination and replace the x's with 0s and 1s. We might, for example, decide to produce lychee ice cream whenever the lychee button is pressed, regardless of other buttons ($LBM = 1xx$, which means that we don't care about the inputs B and M, so $LBM = 100$, $LBM = 101$, $LBM = 110$, or $LBM = 111$). That decision alone covers three of the four unspecified input patterns. We might also decide that when the blend and mango buttons are pushed together (but without the lychee button, LBM=011), our logic produces a blend. The resulting K-maps are shown below, again with shading and green italics identifying the combinations in which our original design specified "don't care."

$C_L[1]$

M \ LB	00	01	11	10
0	0	0	*1*	1
1	0	*0*	*1*	*1*

$C_L[0]$

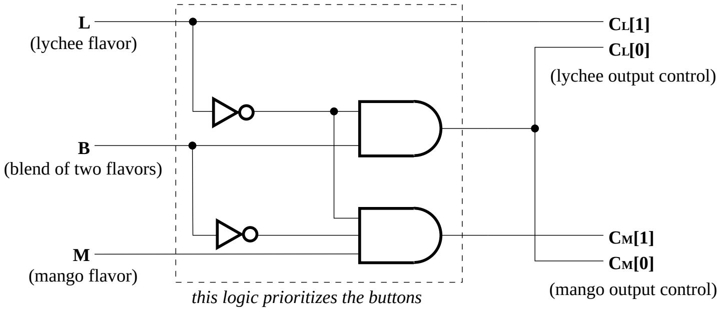
M \ LB	00	01	11	10
0	0	1	*0*	0
1	0	*1*	*0*	*0*

$C_M[1]$

M \ LB	00	01	11	10
0	0	0	*0*	0
1	1	*0*	*0*	*0*

$C_M[0]$

M \ LB	00	01	11	10
0	0	1	*0*	0
1	0	*1*	*0*	*0*

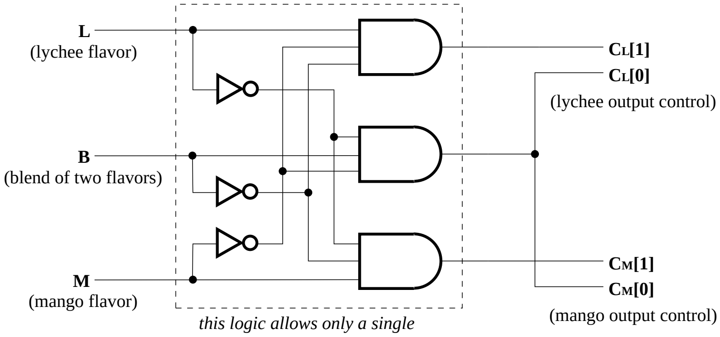


this logic prioritizes the buttons and passes only one at any time

The logic in the dashed box to the right implements the set of choices just discussed, and matches the K-maps above. Based on our additional choices, this implementation enforces a strict priority scheme on the user's button presses. If a user requests lychee, they can also press either or both of the other buttons with no effect. The lychee button has priority. Similarly, if the user does not press lychee, but presses the blend button, pressing the mango button at the same time has no effect. Choosing mango requires that no other buttons be pressed. We have thus chosen a prioritization order for the buttons and imposed this order on the design.

We can view this same implementation in another way. Note the one-to-one correspondence between inputs (on the left) and outputs (on the right) for the dashed box. This logic takes the user's button presses and chooses at most one of the buttons to pass along to our original controller implementation (to the right of the dashed box). In other words, rather than thinking of the logic in the dashed box as implementing a specific set of decisions, we can think of the logic as cleaning up the inputs to ensure that only valid combinations are passed to our original implementation. Once the inputs are cleaned up, the original implementation is acceptable, because input combinations containing more than a single 1 are in fact impossible.

Strict prioritization is one useful way to clean up our inputs. In general, we can design logic to map each of the four undesirable input patterns into one of the permissible combinations (the four that we specified explicitly in our original design, with LBM in the set $\{000, 001, 010, 100\}$). Selecting a prioritization scheme is just one approach for making these choices in a way that is easy for a user to understand and is fairly easy to implement.



this logic allows only a single button to be pressed at any time

A second simple approach is to ignore illegal combinations by mapping them into the "no buttons pressed" input pattern. Such an implementation appears to the right, laid out to show that one can again view the logic in the dashed box either as cleaning up the inputs (by mentally grouping the logic with the inputs) or as a specific set of choices for our "don't care" output values (by grouping the logic with the outputs). In either case, the logic shown enforces our assumptions in a fairly conservative way: if a user presses more than one button, the logic squashes all button presses. Only a single 1 value at a time can pass through to the wires on the right of the figure.

For completeness, the K-maps corresponding to this implementation are given here.

$C_L[1]$		LB			
		00	01	11	10
M	0	0	0	*0*	1
	1	0	*0*	*0*	*0*

$C_L[0]$		LB			
		00	01	11	10
M	0	0	1	*0*	0
	1	0	*0*	*0*	*0*

$C_M[1]$		LB			
		00	01	11	10
M	0	0	0	*0*	0
	1	1	*0*	*0*	*0*

$C_M[0]$		LB			
		00	01	11	10
M	0	0	1	*0*	0
	1	0	*0*	*0*	*0*

Generalizations and Applications

The approaches that we illustrated to clean up the input signals to our design have application in many areas. The ideas in this section are drawn from the field and are sometimes the subjects of later classes, but *are not exam material for our class.*

Prioritization of distinct inputs is used to arbitrate between devices attached to a processor. Processors typically execute much more quickly than do devices. When a device needs attention, the device signals the processor by changing the voltage on an interrupt line (the name comes from the idea that the device interrupts the processor's current activity, such as running a user program). However, more than one device may need the attention of the processor simultaneously, so a priority encoder is used to impose a strict order on the devices and to tell the processor about their needs one at a time. If you want to learn more about this application, take ECE391.

When components are designed together, assuming that some input patterns do not occur is common practice, since such assumptions can dramatically reduce the number of gates required, improve performance, reduce power consumption, and so forth. As a side effect, when we want to test a chip to make sure that no defects or other problems prevent the chip from operating correctly, we have to be careful so as not to "test" bit patterns that should never occur in practice. Making up random bit patterns is easy, but can produce bad results or even destroy the chip if some parts of the design have assumed that a combination produced randomly can never occur. To avoid these problems, designers add extra logic that changes the disallowed patterns into allowed patterns, just as we did with our design. The use of random bit patterns is common in Built-In Self Test (BIST), and so the process of inserting extra logic to avoid problems is called BIST hardening. BIST hardening can add 10-20% additional logic to a design. Our graduate class on digital system testing, ECE543, covers this material, but has not been offered recently.

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.

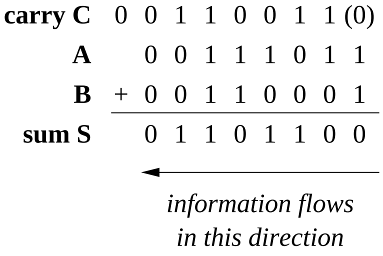
One Bit at a Time

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

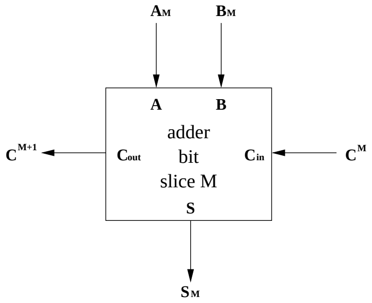
Think about how we as humans add two N-bit numbers, A and B. An illustration appears to the right, using $N = 8$. For now, let's assume that our numbers are stored in an unsigned representation. As you know, addition for 2's complement is identical except for the calculation of overflow. We start adding 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

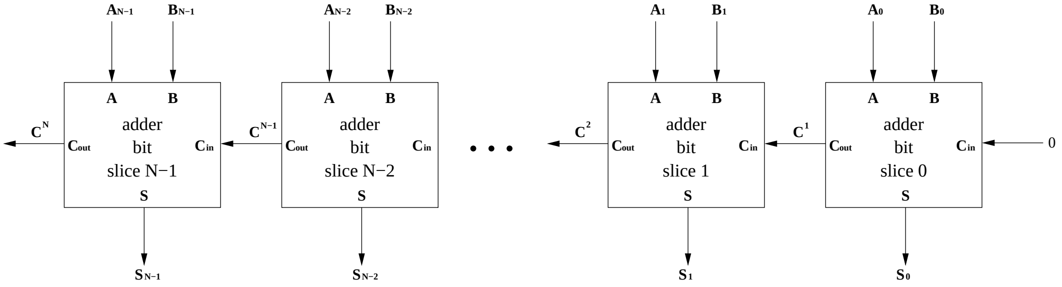
Focus now on the addition of a single column. Except for the first and last bits, which we might choose to handle slightly differently, the addition process is identical for any column. We add a carry in bit (possibly 0) with one bit from each of our numbers to produce a sum bit and a carry out bit for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

an N−bit adder composed of bit slices



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

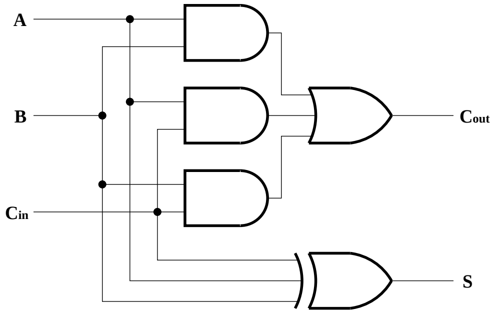
S		AB			
		00	01	11	10
Cin	0	0	1	0	1
	1	1	0	1	0

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

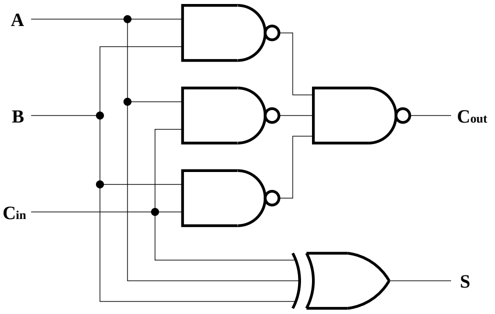
$$\begin{aligned} S &= A\;B\;C_{out} + A\;\overline{B}\;\overline{C_{out}} + \\ &\quad \overline{A}\;B\;\overline{C_{out}} + \overline{A}\;\overline{B}\;C_{out} \\ &= A \oplus B \oplus C_{out} \end{aligned}$$

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.

an adder bit slice (known as a "full adder")



an adder bit slice using NAND gates

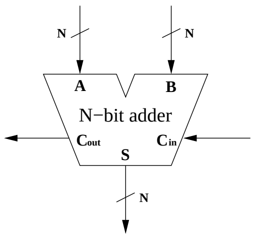


The gate diagrams above implement a single bit slice for an adder. The version on the left uses AND and OR gates (and an XOR for the sum), while the version on the right uses NAND gates, leaving the XOR as an XOR.

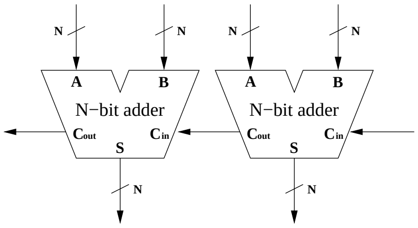
Let's discuss the design in terms of area and speed. As an estimate of area, we can count gates, remembering that we need two transistors per input on a gate. For each bit, we need three 2-input NAND gates, one 3-input NAND gate, and a 3-input XOR gate (a big gate; around 30 transistors). For speed, 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.

ECE199JL: Introduction to Computer Engineering Fall 2012
Notes Set 2.4

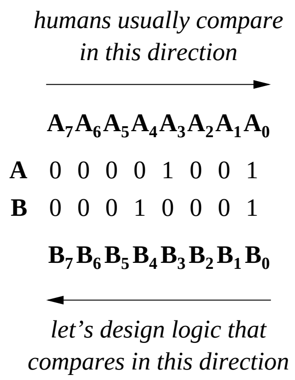
Example: Bit-Sliced Comparison

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.

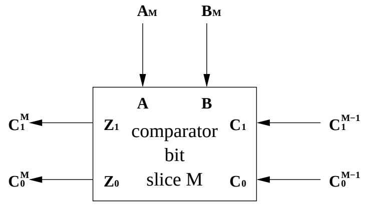

humans usually compare in this direction
A7A6A5A4A3A2A1A0
A 0 0 0 0 1 0 0 1
B 0 0 0 1 0 0 0 1
B7B6B5B4B3B2B1B0
let's design logic that compares in this direction

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

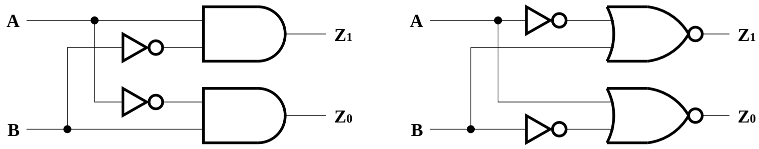
Now we can design the logic for the first bit (bit 0). In keeping with the bit slice philosophy, in practice we simply use another copy of the full bit slice design for bit 0 and attach the C_1C_0 inputs to ground (to denote $A = B$). Here we tackle the simpler problem as a warm-up exercise.

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

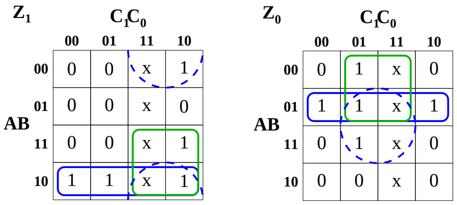
A	B	C_1	C_0	Z_1	Z_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	1	0
x	x	1	1	x	x

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 $ABC_1C_0 = 01\text{xx}$ and output pattern $Z_1Z_0 = 01$. We might also collapse our "don't care" output patterns: whenever the input matches $ABC_1C_0 = \text{xx}11$, we don't care about the output, so $Z_1Z_0 = \text{xx}$. 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:



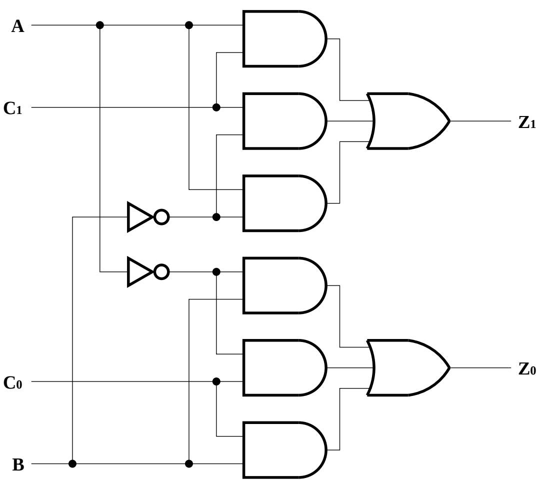
Z_1 (AB \ C_1C_0)	00	01	11	10
00	0	0	x	1
01	0	0	x	0
11	0	0	x	1
10	1	1	x	1

Z_0 (AB \ C_1C_0)	00	01	11	10
00	0	1	x	0
01	1	1	x	1
11	0	1	x	0
10	0	0	x	0

$$
\begin{aligned}
Z_1 &= A\,\overline{B} + A\,C_1 + \overline{B}\,C_1 \\
Z_0 &= \overline{A}\,B + \overline{A}\,C_0 + B\,C_0
\end{aligned}
$$

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.

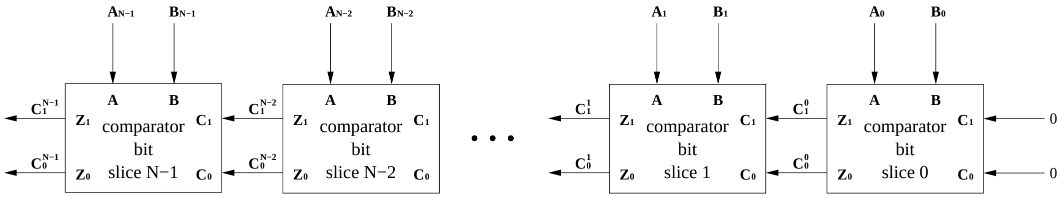
a comparator bit slice (first attempt)



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!

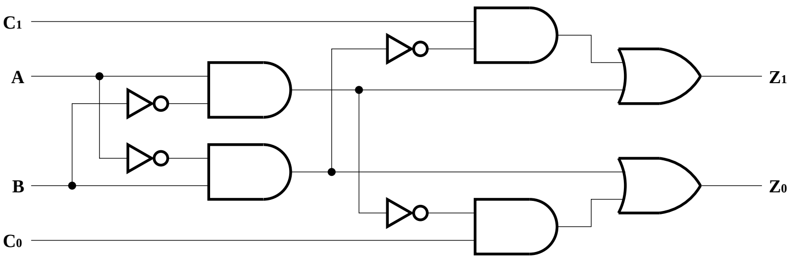
The first step is to manipulate our algebra to expose common terms that occur due to the design's symmetry. Starting with our original equation for Z_1, we have

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

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.

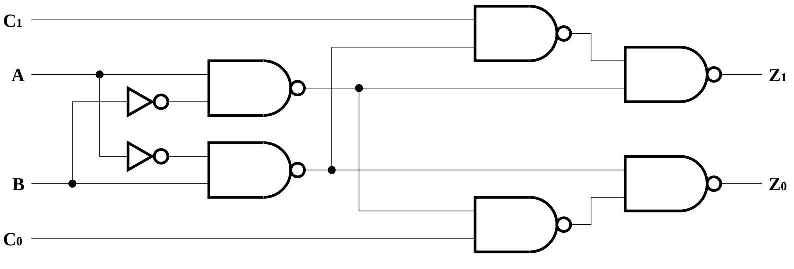
An implementation based on our new algebraic formulation appears to the right. In this form, we seem to have kept the same number of gates, although we have replaced the 3-input gates with inverters. However, the middle inverters disappear when we convert to NAND/NOR form, as shown below to the right. 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)



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.

a comparator bit slice (optimized, NAND/NOR)



Extending to 2's Complement

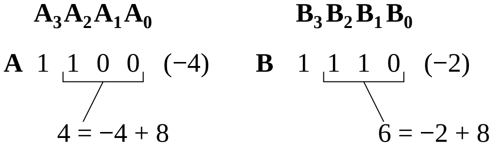
What about comparing 2's complement numbers? Can we make use of the unsigned comparator that we just designed?

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.



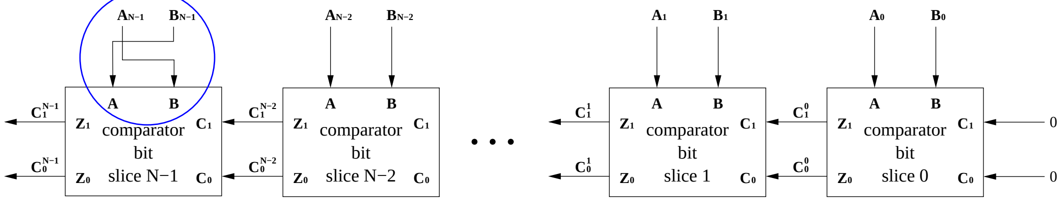
Let's define $A_r = A + 2^{N-1}$ as the value of the remaining bits for A and B_r similarly for B. What happens if we just go ahead and compare A_r and B_r using an $(N-1)$-bit unsigned comparator? If we find that $A_r < B_r$ we know that $A_r - 2^{N-1} < B_r - 2^{N-1}$ as well, but that means $A < B$! 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

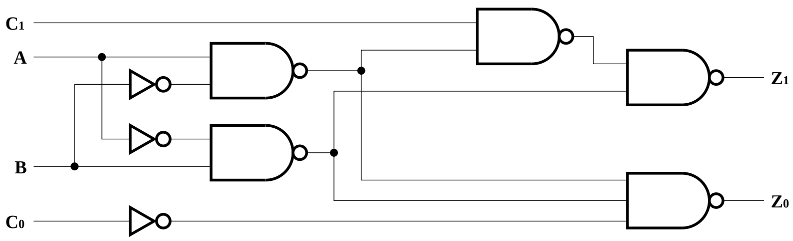
Let's return to the topic of optimization. To what extent did the representation of the three outcomes affect our ability to develop a good bit slice design? Although selecting a good representation can be quite important, for this particular problem most representations lead to similar implementations.

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.

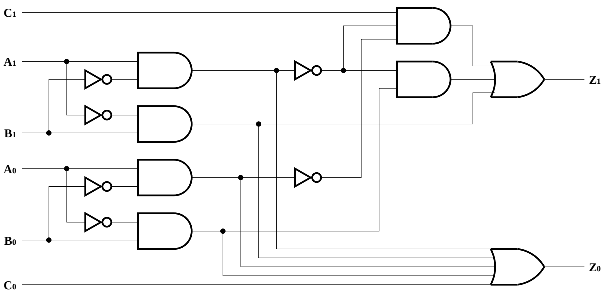
a comparator bit slice (alternate representation)



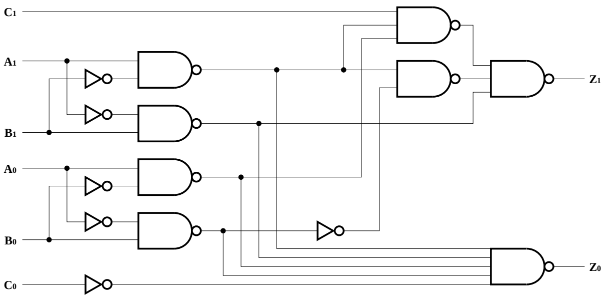
Why didn't it work? Should we consider still other representations? In fact, none of the possible representations that we might choose for a bit slice can cut the delay down to one gate delay per bit. The problem is fundamental, and is related to the nature of CMOS. For a single bit slice, we define 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.

ECE199JL: Introduction to Computer Engineering Fall 2012
Notes Set 2.5

Example: Using Abstraction to Simplify Problems

In this set of notes, we illustrate the use of abstraction to simplify problems. In particular, we show how two specific examples—integer subtraction and identification of upper-case letters in ASCII—can be implemented using logic functions that we have already developed. We also introduce a conceptual technique for breaking functions into smaller pieces, which allows us to solve several simpler problems and then to compose a full solution from these partial solutions.

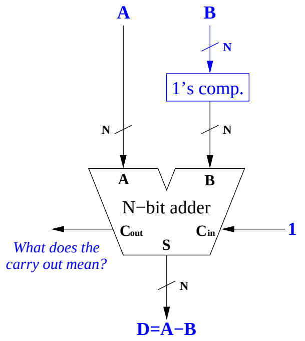
Together with the idea of bit-sliced designs that we introduced earlier, these techniques help to simplify the process of designing logic that operates correctly. The techniques can, of course, lead to less efficient designs, but *correctness is always more important than performance.* The potential loss of efficiency is often acceptable for three reasons. First, as we mentioned earlier, computer-aided design tools for optimizing logic functions are fairly effective, and in many cases produce better results than human engineers (except in the rare cases in which the human effort required to beat the tools is worthwhile). Second, as you know from the design of the 2's complement representation, we may be able to reuse specific pieces of hardware if we think carefully about how we define our problems and representations. Finally, many tasks today are executed in software, which is designed to leverage the fairly general logic available via an instruction set architecture. A programmer cannot easily add new logic to a user's processor. As a result, the hardware used to execute a function typically is not optimized for that function. The approaches shown in this set of notes illustrate how abstraction can be used to design logic.

Subtraction

Our discussion of arithmetic implementation has focused so far on addition. What about other operations, such as subtraction, multiplication, and division? The latter two require more work, and we will not discuss them in detail until later in our class (if at all).

Subtraction, however, can be performed almost trivially using logic that we have already designed. Let's say that we want to calculate the difference D between two N-bit numbers A and B. In particular, we want to find $D = A - B$. For now, think of A, B, and D as 2's complement values. Recall how we defined the 2's complement representation: the N-bit pattern that we use to represent $-B$ is the same as the base 2 bit pattern for $(2^N - B)$, so we can use an adder if we first calculate the bit pattern for $-B$, then add the resulting pattern to A. As you know, our N-bit adder always produces a result that is correct modulo 2^N, so the result of such an operation, $D = 2^N + A - B$, is correct so long as the subtraction does not overflow.

How can we calculate $2^N - B$? The same way that we do by hand! Calculate the 1's complement, $(2^N - 1) - B$, then add 1. The diagram to the right shows how we can use the N-bit adder that we designed in Notes Set 2.3 to build an N-bit subtracter. New elements appear in blue in the figure—the rest of the logic is just an adder. The box labeled "1's comp." calculates the 1's complement of the value B, which together with the carry in value of 1 correspond to calculating $-B$. What's in the "1's comp." box? One inverter per bit in B. That's all we need to calculate the 1's complement. You might now ask: does this approach also work for unsigned numbers? The answer is yes, absolutely. However, the overflow conditions for both 2's complement and unsigned subtraction are different than the overflow condition for either type of addition. What does the carry out of our adder signify, for example? The answer may not be immediately obvious.



Let's start with the overflow condition for unsigned subtraction. Overflow means that we cannot represent the result. With an N-bit unsigned number, we have $A - B \notin [0, 2^N - 1]$. Obviously, the difference cannot be larger than the upper limit, since A is representable and we are subtracting a non-negative (unsigned) value. We can thus assume that overflow occurs only when $A - B < 0$. In other words, when $A < B$.

To calculate the unsigned subtraction overflow condition in terms of the bits, recall that our adder is calculating $2^N + A - B$. The carry out represents the 2^N term. When $A \geq B$, the result of the adder is at least 2^N, and we see a carry out, $C_{out} = 1$. However, when $A < B$, the result of the adder is less than 2^N, and we see no carry out, $C_{out} = 0$. *Overflow for unsigned subtraction is thus inverted from overflow for unsigned addition*: a carry out of 0 indicates an overflow for subtraction.

What about overflow for 2's complement subtraction? We can use arguments similar to those that we used to reason about overflow of 2's complement addition to prove that subtraction of one negative number from a second negative number can never overflow. Nor can subtraction of a non-negative number from a second non-negative number overflow.

If $A \geq 0$ and $B < 0$, the subtraction overflows iff $A - B \geq 2^{N-1}$. Again using similar arguments as before, we can prove that the difference D appears to be negative in the case of overflow, so the product $\overline{A_{N-1}}\ B_{N-1}\ D_{N-1}$ evaluates to 1 when this type of overflow occurs (these variables represent the most significant bits of the two operands and the difference; in the case of 2's complement, they are also the sign bits). Similarly, if $A < 0$ and $B \geq 0$, we have overflow when $A - B < -2^{N-1}$. Here we can prove that $D \geq 0$ on overflow, so $A_{N-1}\ \overline{B_{N-1}}\ \overline{D_{N-1}}$ evaluates to 1.

Our overflow condition for N-bit 2's complement subtraction is thus given by the following:

$$\overline{A_{N-1}}\ B_{N-1}\ D_{N-1} + A_{N-1}\ \overline{B_{N-1}}\ \overline{D_{N-1}}$$

If we calculate all four overflow conditions—unsigned and 2's complement, addition and subtraction—and provide some way to choose whether or not to complement B and to control the C_{in} input, we can use the same hardware for addition and subtraction of either type.

Checking ASCII for Uppercase Letters

Let's now consider how we can check whether or not an ASCII character is an upper-case letter. Let's call the 7-bit letter $C = C_6C_5C_4C_3C_2C_1C_0$ and the function that we want to calculate $L(C)$. The function L should equal 1 whenever C represents an upper-case letter, and 0 whenever C does not.

In ASCII, the 7-bit patterns from 0x41 through 0x5A correspond to the letters A through Z in order. Perhaps you want to draw a 7-input K-map? Get a few large sheets of paper! Instead, imagine that we've written the full 128-row truth table. Let's break the truth table into pieces. Each piece will correspond to one specific pattern of the three high bits $C_6C_5C_4$, and each piece will have 16 entries for the four low bits $C_3C_2C_1C_0$. The truth tables for high bits 000, 001, 010, 011, 110, and 111 are easy: the function is exactly 0. The other two truth tables appear on the left below. We've called the two functions T_4 and T_5, where the subscripts correspond to the binary value of the three high bits of C.

C_3	C_2	C_1	C_0	T_4	T_5
0	0	0	0	0	1
0	0	0	1	1	1
0	0	1	0	1	1
0	0	1	1	1	1
0	1	0	0	1	1
0	1	0	1	1	1
0	1	1	0	1	1
0	1	1	1	1	1
1	0	0	0	1	1
1	0	0	1	1	1
1	0	1	0	1	1
1	0	1	1	1	0
1	1	0	0	1	0
1	1	0	1	1	0
1	1	1	0	1	0
1	1	1	1	1	0

T_4	C_3C_2			
C_1C_0	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

$$T_4 = C_3 + C_2 + C_1 + C_0$$

T_5	C_3C_2			
C_1C_0	00	01	11	10
00	1	1	0	1
01	1	1	0	1
11	1	1	0	0
10	1	1	0	1

$$T_5 = \overline{C_3} + \overline{C_2}\ \overline{C_1} + \overline{C_2}\ \overline{C_0}$$

As shown to the right of the truth tables, we can then draw simpler K-maps for T_4 and T_5, and can solve the K-maps to find equations for each, as shown to the right (check that you get the same answers).

How do we merge these results to form our final expression for L? We AND each of the term functions (T_4 and T_5) with the appropriate minterm for the high bits of C, then OR the results together, as shown here:

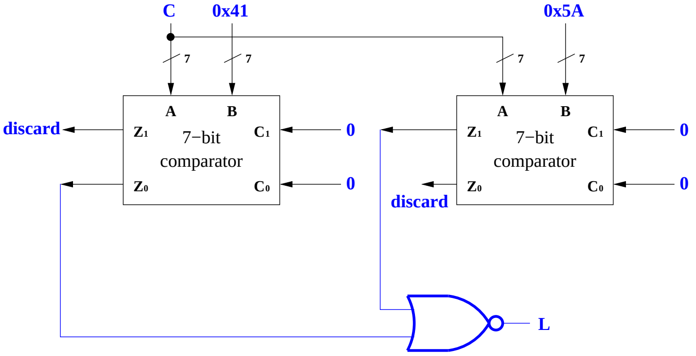
$$\begin{aligned} L &= C_6\ \overline{C_5}\ \overline{C_4}\ T_4 + C_6\ \overline{C_5}\ C_4\ T_5 \\ &= C_6\ \overline{C_5}\ \overline{C_4}\ (C_3 + C_2 + C_1 + C_0) + C_6\ \overline{C_5}\ C_4\ (\overline{C_3} + \overline{C_2}\ \overline{C_1} + \overline{C_2}\ \overline{C_0}) \end{aligned}$$

Rather than trying to optimize by hand, we can at this point let the CAD tools take over, confident that we have the right function to identify an upper-case ASCII letter.

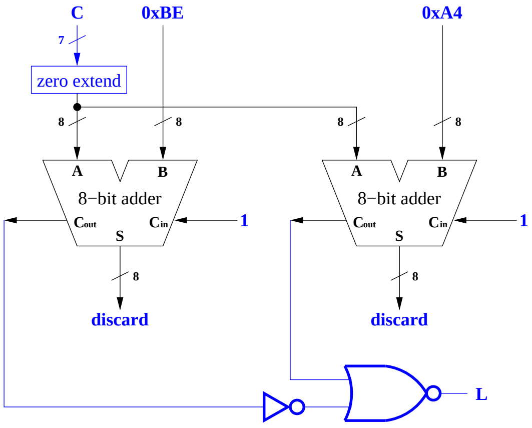
Breaking the truth table into pieces and using simple logic to reconnect the pieces is one way to make use of abstraction when solving complex logic problems. In fact, recruiters for some companies often ask questions that involve using specific logic elements as building blocks to implement other functions. Knowing that you can implement a truth table one piece at a time will help you to solve this type of problem.

Let's think about other ways to tackle the problem of calculating L. In Notes Sets 2.3 and 2.4, we developed adders and comparators. Can we make use of these as building blocks to check whether C represents an upper-case letter? Yes, of course we can: by comparing C with the ends of the range of upper-case letters, we can check whether or not C falls in that range.

The idea is illustrated on the left below using two 7-bit comparators constructed as discussed in Notes Set 2.4. The comparators are the black parts of the drawing, while the blue parts represent our extensions to calculate L. Each comparator is given the value C as one input. The second value to the comparators is either the letter A (0x41) or the letter Z (0x5A). The meaning of the 2-bit input and result to each comparator is given in the table on the right below. The inputs on the right of each comparator are set to 0 to ensure that equality is produced if C matches the second input (B). One output from each comparator is then routed to a NOR gate to calculate L. Let's consider how this combination works. The left comparator compares C with the letter A (0x41). If $C \geq$ 0x41, the comparator produces $Z_0 = 0$. In this case, we may have a letter. On the other hand, if $C <$ 0x41, the comparator produces $Z_0 = 1$, and the NOR gate outputs $L = 0$, since we do not have a letter in this case. The right comparator compares C with the letter Z (0x5A). If $C \leq$ 0x5A, the comparator produces $Z_1 = 0$. In this case, we may have a letter. On the other hand, if $C >$ 0x5A, the comparator produces $Z_1 = 1$, and the NOR gate outputs $L = 0$, since we do not have a letter in this case. Only when 0x41 $\leq C \leq$ 0x51 does $L = 1$, as desired.



Z_1	Z_0	meaning
0	0	$A = B$
0	1	$A < B$
1	0	$A > B$
1	1	not used



What if we have only 8-bit adders available for our use, such as those developed in Notes Set 2.3? Can we still calculate L? Yes. The diagram shown to the right illustrates the approach, again with black for the adders and blue for our extensions. Here we are actually using the adders as subtracters, but calculating the 1's complements of the constant values by hand. The "zero extend" box simply adds a leading 0 to our 7-bit ASCII letter. The left adder subtracts the letter A from C: if no carry is produced, we know that $C <$ 0x41 and thus C does not represent an upper-case letter, and $L = 0$. Similarly, the right adder subtracts 0x5B (the letter Z plus one) from C. If a carry is produced, we know that $C \geq$ 0x5B, and thus C does not represent an upper-case letter, and $L = 0$. With the right combination of carries (1 from the left and 0 from the right), we obtain $L = 1$.

Looking carefully at this solution, however, you might be struck by the fact that we are calculating two sums and then discarding them. Surely such an approach is inefficient?

We offer two answers. First, given the design shown above, a good CAD tool recognizes that the sum outputs of the adders are not being used, and does not generate logic to calculate them. The logic for the two carry bits used to calculate L can then be optimized. Second, the design shown, including the calculation of the sums, is similar in efficiency to what happens at the rate of about 10^{15} times per second, 24 hours a day, seven days a week, inside processors in data centers processing HTML, XML, and other types of human-readable Internet traffic. Abstraction is a powerful tool.

Later in our class, you will learn how to control logical connections between hardware blocks so that you can make use of the same hardware for adding, subtracting, checking for upper-case letters, and so forth.

ECE199JL: Introduction to Computer Engineering Fall 2012
Notes Set 2.6

Sequential Logic

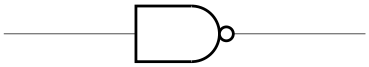
These notes introduce logic components for storing bits, building up from the idea of a pair of cross-coupled inverters through an implementation of a flip-flop, the storage abstractions used in most modern logic design processes. We then introduce simple forms of timing diagrams, which we use to illustrate the behavior of a logic circuit. After commenting on the benefits of using a clocked synchronous abstraction when designing systems that store and manipulate bits, we illustrate timing issues and explain how these are abstracted away in clocked synchronous designs. *Sections marked with an asterisk are provided solely for your interest, but you probably need to learn this material in later classes.*

Storing One Bit

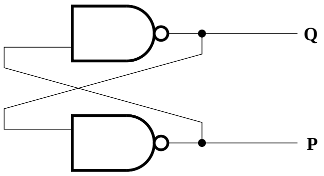
So far we have discussed only implementation of Boolean functions: given some bits as input, how can we design a logic circuit to calculate the result of applying some function to those bits? The answer to such questions is called **combinational logic** (sometimes **combinatorial logic**), a name stemming from the fact that we are combining existing bits with logic, not storing new bits.

You probably already know, however, that combinational logic alone is not sufficient to build a computer. We need the ability to store bits, and to change those bits. Logic designs that make use of stored bits—bits that can be changed, not just wires to high voltage and ground—are called **sequential logic**. The name comes from the idea that such a system moves through a sequence of stored bit patterns (the current stored bit pattern is called the **state** of the system).

Consider the diagram to the right. What is it? A 1-input NAND gate, or an inverter drawn badly? If you think carefully about how these two gates are built, you will realize that they are the same thing. Conceptually, we use two inverters to store a bit, but in most cases we make use of NAND gates to simplify the mechanism for changing the stored bit.



Take a look at the design to the right. Here we have taken two inverters (drawn as NAND gates) and coupled each gate's output to the other's input. What does the circuit do? Let's make some guesses and see where they take us. Imagine that the value at Q is 0. In that case, the lower gate drives P to 1. But P drives the upper gate, which forces Q to 0. In other words, this combination forms a stable state of the system: once the gates reach this state, they continue to hold these values. The first row of the truth table to the right (outputs only) shows this state.



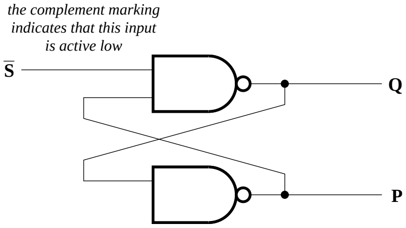
Q	P
0	1
1	0

What if $Q = 1$, though? In this case, the lower gate forces P to 0, and the upper gate in turn forces Q to 1. Another stable state! The $Q = 1$ state appears as the second row of the truth table.

We have identified all of the stable states.[1] Notice that our cross-coupled inverters can store a bit. Unfortunately, we have no way to specify which value should be stored, nor to change the bit's value once the gates have settled into a stable state. What can we do?

[1] Most logic families also allow unstable states in which the values alternate rapidly between 0 and 1. These metastable states are beyond the scope of our class, but ensuring that they do not occur in practice is important for real designs.

Let's add an input to the upper gate, as shown to the right. We call the input $\bar{S}$. The "S" stands for set—as you will see, our new input allows us to **set** our stored bit Q to 1. The use of a complemented name for the input indicates that the input is **active low**. In other words, the input performs its intended task (setting Q to 1) when its value is 0 (not 1).


the complement marking indicates that this input is active low
S̄
Q
P

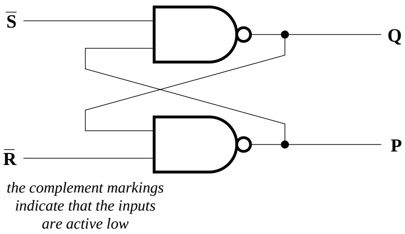
$\bar{S}$	Q	P
1	0	1
1	1	0
0	1	0

Think about what happens when the new input is not active, $\bar{S} = 1$. As you know, ANDing any value with 1 produces the same value, so our new input has no effect when $\bar{S} = 1$. The first two rows of the truth table are simply a copy of our previous table: the circuit can store either bit value when $\bar{S} = 1$. What happens when $\bar{S} = 0$? In that case, the upper gate's output is forced to 1, and thus the lower gate's is forced to 0. This third possibility is reflected in the last row of the truth table.

Now we have the ability to force bit Q to have value 1, but if we want $Q = 0$, we just have to hope that the circuit happens to settle into that state when we turn on the power. What can we do?

As you probably guessed, we add an input to the other gate, as shown to the right. We call the new input $\bar{R}$: the input's purpose is to **reset** bit Q to 0, and the input is active low. We extend the truth table to include a row with $\bar{R} = 0$ and $\bar{S} = 1$, which forces $Q = 0$ and $P = 1$.

an R̄−S̄ latch (stores a single bit)

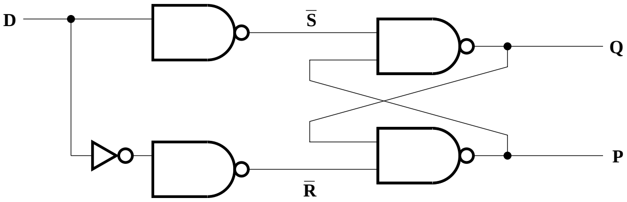

S̄
Q
R̄
P
the complement markings indicate that the inputs are active low

$\bar{R}$	$\bar{S}$	Q	P
1	1	0	1
1	1	1	0
1	0	1	0
0	1	0	1
0	0	1	1

The circuit that we have drawn has a name: an **R̄-S̄ latch**. One can also build R-S latches (with active high set and reset inputs). The textbook also shows an R̄-S̄ latch (labeled incorrectly). Can you figure out how to build an R-S latch yourself?

Let's think a little more about the R̄-S̄ latch. What happens if we set $\bar{S} = 0$ and $\bar{R} = 0$ at the same time? Nothing bad happens immediately. Looking at the design, both gates produce 1, so $Q = 1$ and $P = 1$. The bad part happens later: if we raise both $\bar{S}$ and $\bar{R}$ back to 1 at around the same time, the stored bit may end up in either state.[2]

We can avoid the problem by adding gates to prevent the two control inputs ($\bar{S}$ and $\bar{R}$) from ever being 1 at the same time. A single inverter might technically suffice, but let's build up the structure shown below, noting that the two inverters in sequence connecting D to $\bar{R}$ have no practical effect at the moment. A truth table is shown to the right of the logic diagram. When $D = 0$, $\bar{R}$ is forced to 0, and the bit is reset. Similarly, when $D = 1$, $\bar{S}$ is forced to 0, and the bit is set.

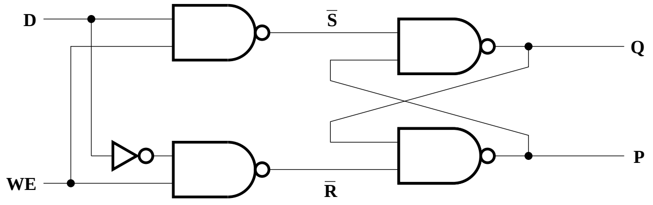


D	$\bar{R}$	$\bar{S}$	Q	P
0	0	1	0	1
1	1	0	1	0

Unfortunately, except for some interesting timing characteristics, the new design has the same functionality as a piece of wire. And, if you ask a circuit designer, thin wires also have some interesting timing characteristics. What can we do? Rather than having Q always reflect the current value of D, let's add some extra inputs to the new NAND gates that allow us to control when the value of D is copied to Q, as shown on the next page.

[2] Or, worse, in a metastable state, as mentioned earlier.

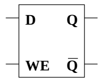
a gated D latch (stores a single bit)



WE	D	$\bar{R}$	$\bar{S}$	Q	P
1	0	0	1	0	1
1	1	1	0	1	0
0	0	1	1	0	1
0	1	1	1	0	1
0	0	1	1	1	0
0	1	1	1	1	0

The WE (write enable) input controls whether or not Q mirrors the value of D. The first two rows in the truth table are replicated from our "wire" design: a value of $WE = 1$ has no effect on the first two NAND gates, and $Q = D$. A value of $WE = 0$ forces the first two NAND gates to output 1, thus $\bar{R} = 1$, $\bar{S} = 1$, and the bit Q can occupy either of the two possible states, regardless of the value of D, as reflected in the lower four lines of the truth table.

gated D latch symbol



P and Q were always opposites, so we now just write $\bar{Q}$ (often omitted entirely in drawings)

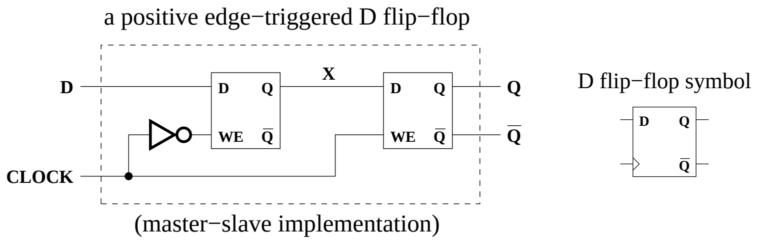
The circuit just shown is called a **gated D latch**, and is an important mechanism for storing state in sequential logic. (Random-access memory uses a slightly different technique to connect the cross-coupled inverters, but latches are used for nearly every other application of stored state.) The "D" stands for "data," meaning that the bit stored is matches the value of the input. Other types of latches (including S-R latches) have been used historically, but D latches are used predominantly today, so we omit discussion of other types. The "gated" qualifier refers to the presence of an enable input (we called it WE) to control when the latch copies its input into the stored bit. A symbol for a gated D latch appears to the right. Note that we have dropped the name P in favor of $\bar{Q}$, since $P = \bar{Q}$ in a gated D latch.

The Clock Abstraction

High-speed logic designs often use latches directly. Engineers specify the number of latches as well as combinational logic functions needed to connect one latch to the next, and the CAD tools optimize the combinational logic. The enable inputs of successive groups of latches are then driven by what we call a clock signal, a single bit line distributed across most of the chip that alternates between 0 and 1 with a regular period. While the clock is 0, one set of latches holds its bit values fixed, and combinational logic uses those latches as inputs to produce bits that are copied into a second set of latches. When the clock switches to 1, the second set of latches stops storing their data inputs and retains their bit values in order to drive other combinational logic, the results of which are copied into a third set of latches. Of course, some of the latches in the first and third sets may be the same.

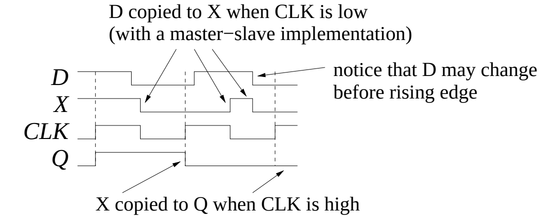
The timing of signals in such designs plays a critical role in their correct operation. Fortunately, we have developed powerful abstractions that allow engineers to ignore much of the complexity while thinking about the Boolean logic needed for a given design.

Towards that end, we make a simplifying assumption for the rest of our class, and for most of your career as an undergraduate: the clock signal is a **square wave** delivered uniformly across a chip. For example, if the period of a clock is 0.5 nanoseconds (2 GHz), the clock signal is a 1 for 0.25 nanoseconds, then a 0 for 0.25 nanoseconds. We assume that the clock signal changes instantaneously and at the same time across the chip. Such a signal can never exist in the real world: voltages do not change instantaneously, and the phrase "at the same time" may not even make sense at these scales. However, circuit designers can usually provide a clock signal that is close enough, allowing us to forget for now that no physical signal can meet our abstract definition.


a positive edge−triggered D flip−flop
D
CLOCK
D
Q
WE
Q̄
X
D
Q
WE
Q̄
Q
Q̄
(master−slave implementation)
D flip−flop symbol
D
Q
Q̄

The device shown to the right is a **master-slave** implementation of a **positive edge-triggered** D flip-flop. As you can see, we have constructed it from two gated D latches with opposite senses of write enable. The "D" part of the name has the same meaning as with a gated D latch: the bit stored is the same as the one delivered to the input. Other variants of flip-flops have also been built, but this type dominates designs today. Most are actually generated automatically from hardware "design" languages (that is, computer programming languages for hardware design).

When the clock is low (0), the first latch copies its value from the flip-flop's D input to the midpoint (marked X in our figure, but not usually given a name). When the clock is high (1), the second latch copies its value from X to the flip-flop's output Q. Since X can not change when the clock is high, the result is that the output changes each time the clock changes from 0 to 1, which is called the **rising edge** or **positive edge** (the derivative) of the clock signal. Hence the qualifier "positive edge-triggered," which describes the flip-flop's behavior. The "master-slave" implementation refers to the use of two latches. In practice, flip-flops are almost never built this way. To see a commercial design, look up 74LS74, which uses six 3-input NAND gates and allows set/reset of the flip-flop (using two extra inputs).


D copied to X when CLK is low
(with a master−slave implementation)
notice that D may change
before rising edge
D
X
CLK
Q
X copied to Q when CLK is high

The **timing diagram** to the right illustrates the operation of our flip-flop. In a timing diagram, the horizontal axis represents (continuous) increasing time, and the individual lines represent voltages for logic signals. The relatively simple version shown here uses only binary values for each signal. One can also draw transitions more realistically (as taking finite time). The dashed vertical lines here represent the times at which the clock rises. To make the example interesting, we have varied D over two clock cycles. Notice that even though D rises and falls during the second clock cycle, its value is not copied to the output of our flip-flop. One can build flip-flops that "catch" this kind of behavior (and change to output 1), but we leave such designs for later in your career.

Circuits such as latches and flip-flops are called **sequential feedback** circuits, and the process by which they are designed is beyond the scope of our course. The "feedback" part of the name refers to the fact that the outputs of some gates are fed back into the inputs of others. Each cycle in a sequential feedback circuit can store one bit. Circuits that merely use latches and flip-flops as building blocks are called **clocked synchronous sequential circuits**. Such designs are still sequential: their behavior depends on the bits currently stored in the latches and flip-flops. However, their behavior is substantially simplified by the use of a clock signal (the "clocked" part of the name) in a way that all elements change at the same time ("synchronously").

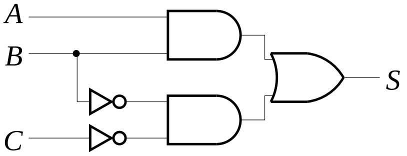
The value of using flip-flops and assuming a square-wave clock signal with uniform timing may not be clear to you yet, but it bears emphasis. With such assumptions, *we can treat time as having discrete values.* In other words, time "ticks" along discretely, like integers instead of real numbers. We can look at the state of the system, calculate the inputs to our flip-flops through the combinational logic that drives their D inputs, and be confident that, when time moves to the next discrete value, we will know the new bit values stored in our flip-flops, allowing us to repeat the process for the next clock cycle without worrying about exactly when things change. Values change only on the rising edge of the clock!

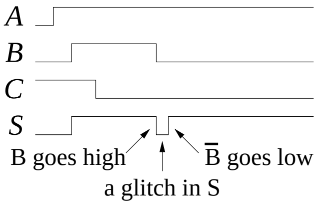
Real systems, of course, are not so simple, and we do not have one clock to drive the universe, so engineers must also design systems that interact even though each has its own private clock signal (usually with different periods).

Static Hazards: Causes and Cures*

Before we forget about the fact that real designs do not provide perfect clocks, let's explore some of the issues that engineers must sometimes face. We discuss these primarily to ensure that you appreciate the power of the abstraction that we use in the rest of our course. In later classes (probably our 298, which will absorb material from 385), you may be required to master this material. *For now, we provide it simply for your interest.*

Consider the circuit shown below, for which the output is given by the equation $S = AB + \bar{B}\bar{C}$.



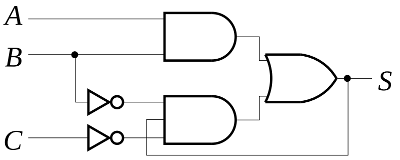


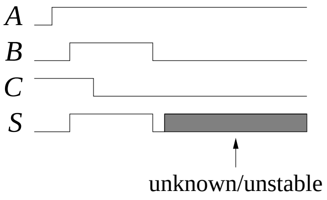
The timing diagram on the right shows a **glitch** in the output when the input shifts from $ABC = 110$ to 100, that is, when B falls. The problem lies in the possibility that the upper AND gate, driven by B, might go low before the lower AND gate, driven by $\bar{B}$, goes high. In such a case, the OR gate output S falls until the second AND gate rises, and the output exhibits a glitch.

A circuit that might exhibit a glitch in an output that functionally remains stable at 1 is said to have a **static-1 hazard**. The qualifier "static" here refers to the fact that we expect the output to remain static, while the "1" refers to the expected value of the output.

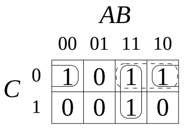
The presence of hazards in circuits can be problematic in certain cases. In domino logic, for example, an output is precharged and kept at 1 until the output of a driving circuit pulls it to 0, at which point it stays low (like a domino that has been knocked over). If the driving circuit contains static-1 hazards, the output may fall in response to a glitch.

Similarly, hazards can lead to unreliable behavior in sequential feedback circuits. Consider the addition of a feedback loop to the circuit just discussed, as shown in the figure below. The output of the circuit is now given by the equation $S^* = AB + \bar{B}\bar{C}S$, where S^* denotes the state after S feeds back through the lower AND gate. In the case discussed previously, the transition from $ABC = 110$ to 100, the glitch in S can break the feedback, leaving S low or unstable. The resulting sequential feedback circuit is thus unreliable.





Eliminating static hazards from two-level circuits is fairly straightforward. The Karnaugh map to the right corresponds to our original circuit; the solid lines indicate the implicants selected by the AND gates. A static-1 hazard is present when two adjacent 1s in the K-map are not covered by a common implicant. Static-0 hazards do not occur in two-level SOP circuits.



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

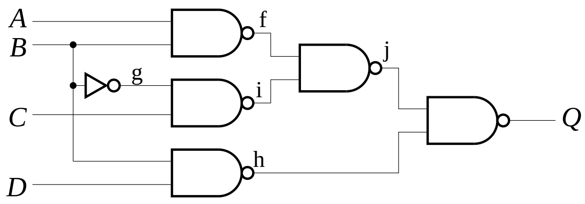
Eliminating static hazards requires merely extending the circuit with consensus terms in order to ensure that some AND gate remains high through every transition between input states with output 1.[3] In the K-map shown, the dashed line indicates the necessary consensus term, $A\bar{C}$.

[3] Hazard elimination is not in general simple; we have considered only two-level circuits.

Dynamic Hazards*

Consider an input transition for which we expect to see a change in an output. Under certain timing conditions, the output may not transition smoothly, but instead bounce between its original value and its new value before coming to rest at the new value. A circuit that might exhibit such behavior is said to contain a **dynamic hazard**. The qualifier "dynamic" refers to the expected change in the output.

Dynamic hazards appear only in more complex circuits, such as the one shown below. The output of this circuit is defined by the equation $Q = \bar{A}B + \bar{A}\bar{C} + \bar{B}\bar{C} + BD$.



Consider the transition from the input state $ABCD = 1111$ to 1011, in which B falls from 1 to 0. For simplicity, assume that each gate has a delay of 1 time unit. If B goes low at time $T = 0$, the table shows the progression over time of logic levels at several intermediate points in the circuit and at the output Q. Each gate merely produces the appropriate output based on its inputs in the previous time step. After one delay, the three gates with B as a direct input change their outputs (to stable, final values). After another delay, at $T = 2$, the other three gates respond to the initial changes and flip their outputs. The resulting changes induce another set of changes at $T = 3$, which in turn causes the output Q to change a final time at $T = 4$.

T	f	g	h	i	j	Q
0	0	0	0	1	1	1
1	1	1	1	1	1	1
2	1	1	1	0	0	0
3	1	1	1	0	1	1
4	1	1	1	0	1	0

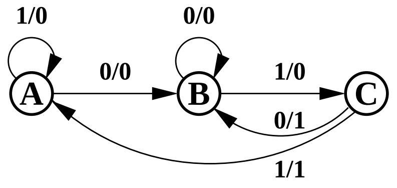
The output column in the table illustrates the possible impact of a dynamic hazard: rather than a smooth transition from 1 to 0, the output drops to 0, rises back to 1, and finally falls to 0 again. The dynamic hazard in this case can be attributed to the presence of a static hazard in the logic that produces intermediate value j.

Essential Hazards*

Essential hazards are inherent to the function of a circuit and may appear in any implementation. In sequential feedback circuit design, they must be addressed at a low level to ensure that variations in logic path lengths (**timing skew**) through a circuit do not expose them. With clocked synchronous circuits, essential hazards are abstracted into a single form: **clock skew**, or disparate clock edge arrival times at a circuit's flip-flops.

An example demonstrates the possible effects: consider the construction of a clocked synchronous circuit to recognize 0-1 sequences on an input IN. Output Q should be held high for one cycle after recognition, that is, until the next rising clock edge. A description of states and a state diagram for such a circuit appear below.

S_1S_0	state	meaning
00	A	nothing, 1, or 11 seen last
01	B	0 seen last
10	C	01 recognized (output high)
11	unused	



For three states, we need two ($= \lceil \log_2 3 \rceil$) flip-flops. Denote the internal state S_1S_0. The specific internal state values for each logical state (A, B, and C) simplify the implementation and the example. A **state table** and K-maps for the next-state logic appear below. The state table uses one line per state with separate columns for each input combination, making the table more compact than one with one line per state/input combination. Each column contains the full next-state information, including output. Using this form of the state table, the K-maps can be read directly from the table.

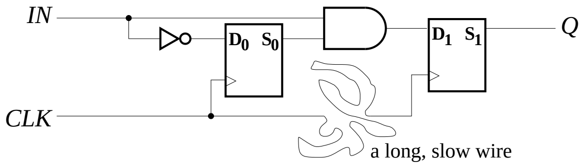
	IN	
S_1S_0	0	1
00	01/0	00/0
01	01/0	10/0
11	x	x
10	01/1	00/1

S_1^+		S_1S_0			
		00	01	11	10
IN	0	0	0	x	0
	1	0	1	x	0

S_0^+		S_1S_0			
		00	01	11	10
IN	0	1	1	x	1
	1	0	0	x	0

Q		S_1S_0			
		00	01	11	10
IN	0	0	0	x	1
	1	0	0	x	1

Examining the K-maps, we see that the excitation and output equations are $S_1^+ = IN \cdot S_0$, $S_0^+ = \overline{IN}$, and $Q = S_1$. An implementation of the circuit using two D flip-flops appears below. Imagine that mistakes in routing or process variations have made the clock signal's path to flip-flop 1 much longer than its path into flip-flop 0, as illustrated.



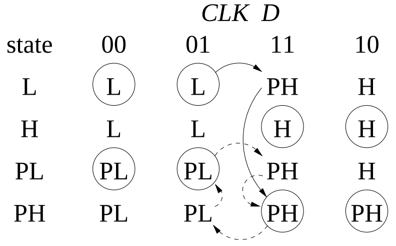
Due to the long delays, we cannot assume that rising clock edges arrive at the flip-flops at the same time. The result is called clock skew, and can make the circuit behave improperly by exposing essential hazards. In the logical B to C transition, for example, we begin in state $S_1S_0 = 01$ with $IN = 1$ and the clock edge rising. Assume that the edge reaches flip-flop 0 at time $T = 0$. After a flip-flop delay ($T = 1$), S_0 goes low. After another AND gate delay ($T = 2$), input D_1 goes low, but the second flip-flop has yet to change state! Finally, at some later time, the clock edge reaches flip-flop 1. However, the output S_1 remains at 0, leaving the system in state A rather than state C.

Fortunately, in clocked synchronous sequential circuits, all essential hazards are related to clock skew. This fact implies that we can eliminate a significant amount of complexity from circuit design by doing a good job of distributing the clock signal. It also implies that, as a designer, you should avoid specious addition of logic in a clock path, as you may regret such a decision later, as you try to debug the circuit timing.

Proof Outline for Clocked Synchronous Design*

This section outlines a proof of the claim made regarding clock skew being the only source of essential hazards for clocked synchronous sequential circuits. A **proof outline** suggests the form that a proof might take and provides some of the logical arguments, but is not rigorous enough to be considered a proof. Here we use a D flip-flop to illustrate a method for identifying essential hazards (*the D flip-flop has no essential hazards, however*), then argue that the method can be applied generally to collections of flip-flops in a clocked synchronous design to show that essential hazards occur only in the form of clock skew.

state		
low	L	clock low, last input low
high	H	clock high, last input low
pulse low	PL	clock low, last input high (output high, too)
pulse high	PH	clock high, last input high (output high, too)



	CLK D			
state	00	01	11	10
L	L	L	PH	H
H	L	L	H	H
PL	PL	PL	PH	H
PH	PL	PL	PH	PH

Consider the sequential feedback state table for a positive edge-triggered D flip-flop, shown above. In designing and analyzing such circuits, we assume that only one input bit changes at a time. The state table consists of one row for each state and one column for each input combination. Within a row, input combinations that have no effect on the internal state of the circuit (that is, those that do not cause any change in the state) are said to be stable; these states are circled. Other states are unstable, and the circuit changes state in response to changes in the inputs.

For example, given an initial state L with low output, low clock, and high input D, the solid arcs trace the reaction of the circuit to a rising clock edge. From the 01 input combination, we move along the column to the 11 column, which indicates the new state, PH. Moving down the column to that state's row, we see that the new state is stable for the input combination 11, and we stop. If PH were not stable, we would continue to move within the column until coming to rest on a stable state.

An essential hazard appears in such a table as a difference between the final state when flipping a bit once and the final state when flipping a bit thrice in succession. The dashed arcs in the figure illustrate the concept: after coming to rest in the PH state, we reset the input to 01 and move along the PH row to find a new state of PL. Moving up the column, we see that the state is stable. We then flip the clock a third time and move back along the row to 11, which indicates that PH is again the next state. Moving down the column, we come again to rest in PH, the same state as was reached after one flip. Flipping a bit three times rather than once evaluates the impact of timing skew in the circuit; if a different state is reached after two more flips, timing skew could cause unreliable behavior. As you can verify from the table, a D flip-flop has no essential hazards.

A group of flip-flops, as might appear in a clocked synchronous circuit, can and usually does have essential hazards, but only dealing with the clock. As you know, the inputs to a clocked synchronous sequential circuit consist of a clock signal and other inputs (either external of fed back from the flip-flops). Changing an input other than the clock can change the internal state of a flip-flop (of the master-slave variety), but flip-flop designs do not capture the number of input changes in a clock cycle beyond one, and changing an input three times is the same as changing it once. Changing the clock, of course, results in a synchronous state machine transition.

The detection of essential hazards in a clocked synchronous design based on flip-flops thus reduces to examination of the state machine. If the next state of the machine has any dependence on the current state, an essential hazard exists, as a second rising clock edge moves the system into a second new state. For a single D flip-flop, the next state is independent of the current state, and no essential hazards are present.

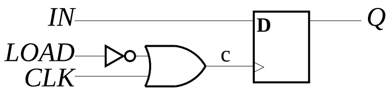
ECE199JL: Introduction to Computer Engineering Fall 2012
Notes Set 2.7

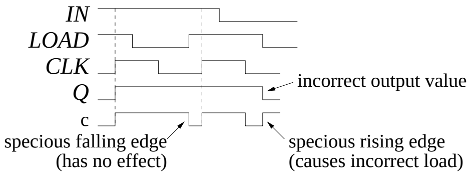
Registers

This set of notes introduces registers, an abstraction used for storage of groups of bits in digital systems. We introduce some terminology used to describe aspects of register design and illustrate the idea of a shift register. The registers shown here are important abstractions for digital system design. *In the Fall 2012 offering of our course, we will cover this material on the third midterm.*

Registers

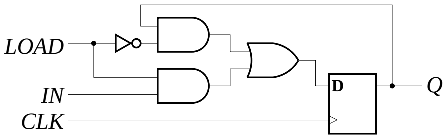
A **register** is a storage element composed from one or more flip-flops operating on a common clock. In addition to the flip-flops, most registers include logic to control the bits stored by the register. For example, the D flip-flops described previously copy their inputs at the rising edge of each clock cycle, discarding whatever bits they have stored during that cycle. To enable a flip-flop to retain its value, we might try to hide the rising edge of the clock from the flip-flop, as shown to the right.



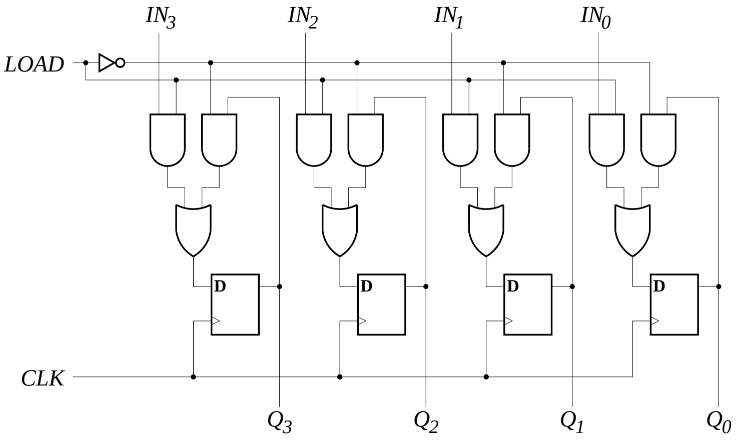


The *LOAD* input controls the clock signals through a method known as **clock gating**. When *LOAD* is high, the circuit reduces to a regular D flip-flop. When *LOAD* is low, the flip-flop clock input, *c*, is held high, and the flip-flop stores its current value. The problems with clock gating are twofold. First, adding logic to the clock path introduces clock skew, which may cause timing problems later in the development process (or, worse, in future projects that use your circuits as components). Second, in the design shown above, the *LOAD* signal can only be lowered while the clock is high to prevent spurious rising edges from causing incorrect behavior, as shown in the timing diagram.

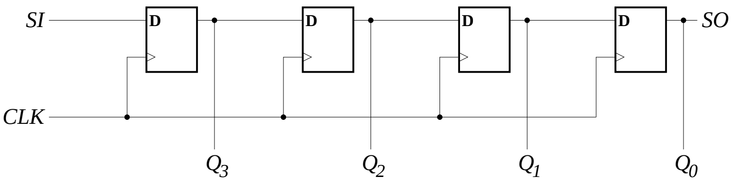
A better approach is to add a feedback loop from the flip-flop's output, as shown in the figure to the right. When *LOAD* is low, the upper AND gate selects the feedback line, and the register reloads its current value. When *LOAD* is high, the lower AND gate selects the *IN* input, and the register loads a new value. We will generalize this type of selection structure, known as a multiplexer, later in our course. The result is similar to a gated D latch with distinct write enable and clock lines.



We can use this extended flip-flop as a bit slice for a multi-bit register. A four-bit register of this type is shown to the right. Four data lines—one for each bit—enter the registers from the top of the figure. When *LOAD* is low, the logic copies each flip-flop's value back to its input, and the *IN* input lines are ignored. When *LOAD* is high, the logic forwards each *IN* line to the corresponding flip-flop's *D* input, allowing the register to load the new 4-bit value. The use of one input line per bit to load a multi-bit register in a single cycle is termed a **parallel load**.



Shift Registers

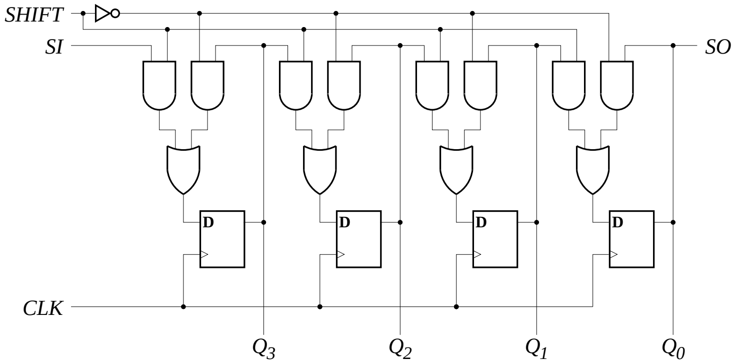


Certain types of registers include logic to manipulate data held within the register. A **shift register** is an important example of this type. The simplest shift register is a series of D flip-flops, with the output of each attached to the input of the next, as shown to the right. In the circuit shown, a serial input SI accepts a single bit of data per cycle and delivers the bit four cycles later to a serial output SO. Shift registers serve many purposes in modern systems, from the obvious uses of providing a fixed delay and performing bit shifts for processor arithmetic to rate matching between components and reducing the pin count on programmable logic devices such as field programmable gate arrays (FPGAs), the modern form of the programmable logic array mentioned in the textbook.

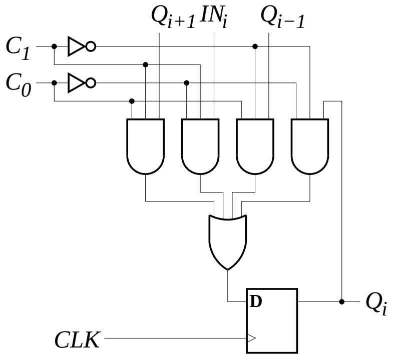
An example helps to illustrate the rate matching problem: historical I/O buses used fairly slow clocks, as they had to drive signals and be arbitrated over relatively long distances. The Peripheral Control Interconnect (PCI) standard, for example, provided for 33 and 66 MHz bus speeds. To provide adequate data rates, such buses use many wires in parallel, either 32 or 64 in the case of PCI. In contrast, a Gigabit Ethernet (local area network) signal travelling over a fiber is clocked at 1.25 GHz, but sends only one bit per cycle. Several layers of shift registers sit between the fiber and the I/O bus to mediate between the slow, highly parallel signals that travel over the I/O bus and the fast, serial signals that travel over the fiber. The latest variant of PCI, PCIe (e for "express"), uses serial lines at much higher clock rates.

Returning to the figure above, imagine that the outputs Q_i feed into logic clocked at $1/4^{th}$ the rate of the shift register (and suitably synchronized). Every four cycles, the flip-flops fill up with another four bits, at which point the outputs are read in parallel. The shift register shown can thus serve to transform serial data to 4-bit-parallel data at one-quarter the clock speed. Unlike the registers discussed earlier, the shift register above does not support parallel load, which prevents it from transforming a slow, parallel stream of data into a high-speed serial stream. The use of **serial load** requires N cycles for an N-bit register, but can reduce the number of wires needed to support the operation of the shift register. How would you add support for parallel load? How many additional inputs would be necessary?

The shift register shown above is also incapable of storing a value rather than continuously shifting. The addition of the same structure that we used to control register loading can be applied to control shifting, as shown to the right.

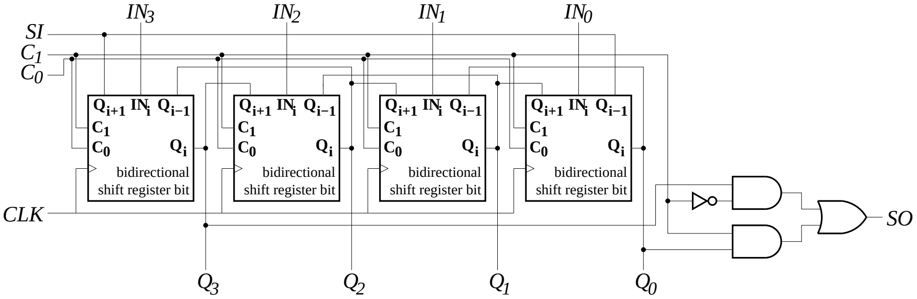


Through the use of more complex input logic, we can construct a shift register with additional functionality. The bit slice shown to the right allows us to build a **bidirectional shift register** with parallel load capability and the ability to retain its value indefinitely. The two control inputs, C_1 and C_0, make use of a representation that we have chosen for the four operations supported by our shift register, as shown in the table to the far right.



C_1	C_0	meaning
0	0	retain current value
0	1	shift left (low to high)
1	0	load new value (from IN)
1	1	shift right (high to low)

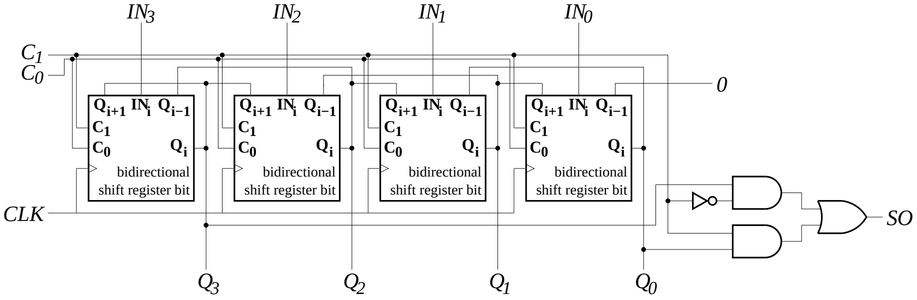
The bit slice allows us to build N-bit shift registers by replicating the slice and adding a fixed amount of "**glue logic**" (for example, the SO output logic). The figure to the right represents a 4-bit bidirectional shift register constructed in this way.



At each rising clock edge, the action specified by C_1C_0 is taken. When $C_1C_0 = 00$, the register holds its current value, with the register value appearing on $Q[3:0]$ and each flip-flop feeding its output back into its input. For $C_1C_0 = 01$, the shift register shifts left: the serial input, SI, is fed into flip-flop 0, and Q_3 is passed to the serial output, SO. Similarly, when $C_1C_0 = 11$, the shift register shifts right: SI is fed into flip-flop 3, and Q_0 is passed to SO. Finally, the case $C_1C_0 = 10$ causes all flip-flops to accept new values from $IN[3:0]$, effecting a parallel load.

Several specialized shift operations are used to support data manipulation in modern processors (CPU's). Essentially, these specializations dictate the form of the glue logic for a shift register as well as the serial input value. The simplest is a **logical shift**, for which SI and SO are hardwired to 0; incoming bits are always 0. A **cyclic shift** takes SI and feeds it back into SO, forming a circle of register bits through which the data bits cycle.

Finally, an **arithmetic shift** treats the shift register contents as a number in 2's complement form. For non-negative numbers and left shifts, an arithmetic shift is the same as a logical shift. When a negative number is arithmetically shifted to the right, however, the sign bit is retained, resulting in a function similar to division by two. The difference lies in the rounding direction. Division by two rounds towards zero in most processors: $-5/2$ gives -2. Arithmetic shift right rounds away from zero for negative numbers (and towards zero for positive numbers): $-5 >> 1$ gives -3. We transform our previous shift register into one capable of arithmetic shifts by eliminating the serial input and feeding the most significant bit, which represents the sign in 2's complement form, back into itself for right shifts, as shown below.



ECE199JL: Introduction to Computer Engineering Fall 2012
Notes Set 2.8

Summary of Part 2 of the Course

The difficulty of learning depends on the type of task involved. Remembering new terminology is relatively easy, while applying the ideas underlying design decisions shown by example to new problems posed as human tasks is relatively hard. In this short summary, we give you lists at several levels of difficulty of what we expect you to be able to do as a result of the last few weeks of studying (reading, listening, doing homework, discussing your understanding with your classmates, and so forth).

We'll start with the easy stuff. You should recognize all of these terms and be able to explain what they mean. For the specific circuits, you should be able to draw them and explain how they work. Actually, we don't care whether you can draw something from memory—a full adder, for example—provided that you know what a full adder does and can derive a gate diagram correctly for one in a few minutes. Higher-level skills are much more valuable. (You may skip the *'d terms in Fall 2012.)

- Boolean functions and logic gates
  - NOT/inverter
  - AND
  - OR
  - XOR
  - NAND
  - NOR
  - XNOR
  - majority function
- specific logic circuits
  - full adder
  - ripple carry adder
  - $\bar{R}$-$\bar{S}$ latch
  - R-S latch
  - gated D latch
  - master-slave implementation of a positive edge-triggered D flip-flop
  - (bidirectional) shift register*
  - register supporting parallel load*
- design metrics
  - metric
  - optimal
  - heuristic
  - constraints
  - power, area/cost, performance
  - computer-aided design (CAD) tools
  - gate delay
- general math concepts
  - canonical form
  - N-dimensional hypercube
- tools for solving logic problems
  - truth table
  - Karnaugh map (K-map)
  - implicant
  - prime implicant
  - bit-slicing
  - timing diagram
- device technology
  - complementary metal-oxide semiconductor (CMOS)
  - field effect transistor (FET)
  - transistor gate, source, drain
- Boolean logic terms
  - literal
  - algebraic properties
  - dual form, principle of duality
  - sum, product
  - minterm, maxterm
  - sum-of-products (SOP)
  - product-of-sums (POS)
  - canonical sum/SOP form
  - canonical product/POS form
  - logical equivalence
- digital systems terms
  - word size
  - N-bit Gray code
  - combinational/combinatorial logic
    - two-level logic
    - "don't care" outputs (x's)
  - sequential logic
    - state
    - active low inputs
    - set a bit (to 1)
    - reset a bit (to 0)
    - master-slave implementation
    - positive edge-triggered
  - clock signal
    - square wave
    - rising/positive clock edge
    - falling/negative clock edge
    - clock gating
  - clocked synchronous sequential circuits
  - parallel/serial load of registers*
  - glue logic*
  - logical/arithmetic/cyclic shift*

We expect you to be able to exercise the following skills:

- Design a CMOS gate from n-type and p-type transistors.
- Apply DeMorgan's laws repeatedly to simplify the form of the complement of a Boolean expression.
- Use a K-map to find a reasonable expression for a Boolean function (for example, in POS or SOP form with the minimal number of terms).
- More generally, translate Boolean logic functions among concise algebraic, truth table, K-map, and canonical (minterm/maxterm) forms.

When designing combinational logic, we expect you to be able to apply the following design strategies:

- Make use of human algorithms (for example, multiplication from addition).
- Determine whether a bit-sliced approach is applicable, and, if so, make use of one.
- Break truth tables into parts so as to solve each part of a function separately.
- Make use of known abstractions (adders, comparators, or other abstractions available to you) to simplify the problem.

And, at the highest level, we expect that you will be able to do the following:

- Understand and be able to reason at a high-level about circuit design tradeoffs between area/cost and performance (and that power is also important, but we haven't given you any quantification methods).
- Understand the tradeoffs typically made to develop bit-sliced designs—typically, bit-sliced designs are simpler but bigger and slower—and how one can develop variants between the extremes of the bit-sliced approach and optimization of functions specific to an N-bit design.
- Understand the pitfalls of marking a function's value as "don't care" for some input combinations, and recognize that implementations do not produce "don't care."
- Understand the tradeoffs involved in selecting a representation for communicating information between elements in a design, such as the bit slices in a bit-sliced design.
- Explain the operation of a latch or a flip-flop, particularly in terms of the bistable states used to hold a bit.
- Understand and be able to articulate the value of the clocked synchronous design abstraction.