ECE 120

Introduction to Computing

Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

The bulk of these notes was developed for the ECE 120 class at the University of Illinois at Urbana-Champaign, but some material was taken from my previous lecture notes for ECE 290 (written around 2000), ECE 190 (written around 2004), and ECE 391 (written around 2004).

Contents

ECE120: Introduction to Computer Engineering

Notes Set 1.1 The Halting Problem

For some of the topics in this course, we plan to cover the material more deeply than does the textbook. We will provide notes in this format to supplement the textbook for this purpose. In order to make these notes more useful as a reference, definitions are highlighted with boldface, and italicization emphasizes pitfalls or other important points. *Sections marked with an asterisk are provided solely for your interest, but you probably need to learn this material in later classes.*

These notes are broken up into four parts, corresponding to the three midterm exams and the final exam. Each part is covered by one examination in our class. *The last section of each of the four parts gives you a summary of material that you are expected to know for the corresponding exam.* Feel free to read it in advance.

As discussed in the textbook and in class, a **universal computational device** (or **computing machine**) is a device that is capable of computing the solution to any problem that can be computed, provided that the device is given enough storage and time for the computation to finish.

One might ask whether we can describe problems that we cannot answer (other than philosophical ones, such as the meaning of life). The answer is yes: there are problems that are provably **undecidable**, for which no amount of computation can solve the problem in general. This set of notes describes the first problem known to be undecidable, the **halting problem**. For our class, you need only recognize the name and realize that one can, in fact, give examples of problems that cannot be solved by computation. In the future, you should be able to recognize this type of problem so as to avoid spending your time trying to solve it.

1.1.1 Universal Computing Machines*

The things that we call computers today, whether we are talking about a programmable microcontroller in a microwave oven or the Blue Waters supercomputer sitting on the south end of our campus (the United States' main resource to support computational science research), are all equivalent in the sense of what problems they can solve. These machines do, of course, have access to different amounts of memory, and compute at different speeds.

The idea that a single model of computation could be described and proven to be equivalent to all other models came out of a 1936 paper by Alan Turing, and today we generally refer to these devices as **Turing machines**. All computers mentioned earlier, as well as all computers with which you are familiar in your daily life, are provably equivalent to Turing machines.

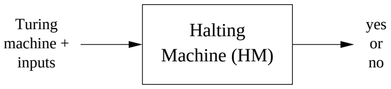
Turing also conjectured that his definition of computable was identical to the "natural" definition (today, this claim is known as the **Church-Turing conjecture**). In other words, a problem that cannot be solved by a Turing machine cannot be solved in any systematic manner, with any machine, or by any person. This conjecture remains unproven! However, neither has anyone been able to disprove the conjecture, and it is widely believed to be true. Disproving the conjecture requires that one demonstrate a systematic technique (or a machine) capable of solving a problem that cannot be solved by a Turing machine. No one has been able to do so to date.

1.1.2 The Halting Problem*

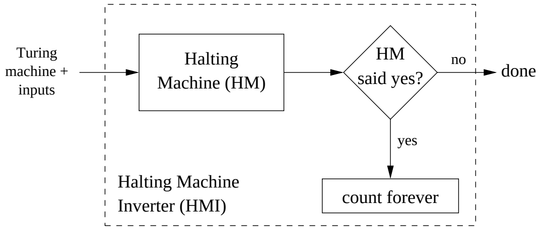
You might reasonably ask whether any problems can be shown to be incomputable. More common terms for such problems—those known to be insolvable by any computer—are **intractable** or undecidable. In the same 1936 paper in which he introduced the universal computing machine, Alan Turing also provided an answer to this question by introducing (and proving) that there are in fact problems that cannot be computed by a universal computing machine. The problem that he proved undecidable, using proof techniques almost identical to those developed for similar problems in the 1880s, is now known as **the halting problem**.

The halting problem is easy to state and easy to prove undecidable. The problem is this: given a Turing machine and an input to the Turing machine, does the Turing machine finish computing in a finite number of steps (a finite amount of time)? In order to solve the problem, an answer, either yes or no, must be given in a finite amount of time regardless of the machine or input in question. Clearly some machines never finish. For example, we can write a Turing machine that counts upwards starting from one.

You may find the proof structure for undecidability of the halting problem easier to understand if you first think about a related problem with which you may already be familiar, the Liar's paradox (which is at least 2,300 years old). In its stengthened form, it is the following sentence: "This sentence is not true."

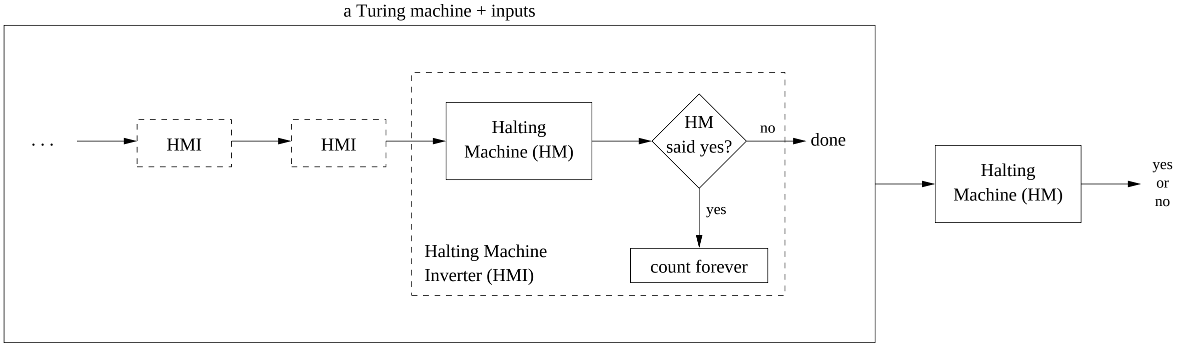


To see that no Turing machine can solve the halting problem, we begin by assuming that such a machine exists, and then show that its existence is self-contradictory. We call the machine the "Halting Machine," or HM for short. HM is a machine that operates on another Turing machine and its inputs to produce a yes or no answer in finite time: either the machine in question finishes in finite time (HM returns "yes"), or it does not (HM returns "no"). The figure illustrates HM's operation.



From HM, we construct a second machine that we call the HM Inverter, or HMI. This machine inverts the sense of the answer given by HM. In particular, the inputs are fed directly into a copy of HM, and if HM answers "yes," HMI enters an infinite loop. If HM answers "no," HMI halts. A diagram appears to the right.

The inconsistency can now be seen by asking HM whether HMI halts when given itself as an input (repeatedly), as shown below. Two copies of HM are thus being asked the same question. One copy is the rightmost in the figure below and the second is embedded in the HMI machine that we are using as the input to the rightmost HM. As the two copies of HM operate on the same input (HMI operating on HMI), they should return the same answer: a Turing machine either halts on an input, or it does not; they are deterministic.



Let's assume that the rightmost HM tells us that HMI operating on itself halts. Then the copy of HM in HMI (when HMI executes on itself, with itself as an input) must also say "yes." But this answer implies that HMI doesn't halt (see the figure above), so the answer should have been no!

Alternatively, we can assume that the rightmost HM says that HMI operating on itself does not halt. Again, the copy of HM in HMI must give the same answer. But in this case HMI halts, again contradicting our assumption.

Since neither answer is consistent, no consistent answer can be given, and the original assumption that HM exists is incorrect. Thus, no Turing machine can solve the halting problem.

ECE120: Introduction to Computer Engineering

Notes Set 1.2 The 2's Complement Representation

This set of notes explains the rationale for using the 2's complement representation for signed integers and derives the representation based on equivalence of the addition function to that of addition using the unsigned representation with the same number of bits.

1.2.1 Review of Bits and the Unsigned Representation

In modern digital systems, we represent all types of information using binary digits, or **bits**. Logically, a bit is either 0 or 1. Physically, a bit may be a voltage, a magnetic field, or even the electrical resistance of a tiny sliver of glass. Any type of information can be represented with an ordered set of bits, provided that *any given pattern of bits corresponds to only one value* and that *we agree in advance on which pattern of bits represents which value.*

For unsigned integers—that is, whole numbers greater or equal to zero—we chose to use the base 2 representation already familiar to us from mathematics. We call this representation the **unsigned representation**. For example, in a 4-bit unsigned representation, we write the number 0 as 0000, the number 5 as 0101, and the number 12 as 1100. Note that we always write the same number of bits for any pattern in the representation: *in a digital system, there is no "blank" bit value.*

1.2.2 Picking a Good Representation

In class, we discussed the question of what makes one representation better than another. The value of the unsigned representation, for example, is in part our existing familiarity with the base 2 analogues of arithmetic. For base 2 arithmetic, we can use nearly identical techniques to those that we learned in elementary school for adding, subtracting, multiplying, and dividing base 10 numbers.

Reasoning about the relative merits of representations from a practical engineering perspective is (probably) currently beyond your ability. Saving energy, making the implementation simple, and allowing the implementation to execute quickly probably all sound attractive, but a quantitative comparison between two representations on any of these bases requires knowledge that you will acquire in the next few years.

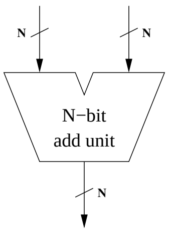
We can sidestep such questions, however, by realizing that if a digital system has hardware to perform operations such as addition on unsigned values, using the same piece of hardware to operate on other representations incurs little or no additional cost. In this set of notes, we discuss the 2's complement representation, which allows reuse of the unsigned add unit (as well as a basis for performing subtraction of either representation using an add unit!). In discussion section and in your homework, you will use the same idea to perform operations on other representations, such as changing an upper case letter in ASCII to a lower case one, or converting from an ASCII digit to an unsigned representation of the same number.

1.2.3 The Unsigned Add Unit

In order to define a representation for signed integers that allows us to reuse a piece of hardware designed for unsigned integers, we must first understand what such a piece of hardware actually does (we do not need to know how it works yet—we'll explore that question later in our class).

The unsigned representation using N bits is not closed under addition. In other words, for any value of N, we can easily find two N-bit unsigned numbers that, when added together, cannot be represented as an N-bit unsigned number. With $N = 4$, for example, we can add 12 (1100) and 6 (0110) to obtain 18. Since 18 is outside of the range $[0, 2^4 - 1]$ representable using the 4-bit unsigned representation, our representation breaks if we try to represent the sum using this representation. We call this failure an **overflow** condition: the representation cannot represent the result of the operation, in this case addition.

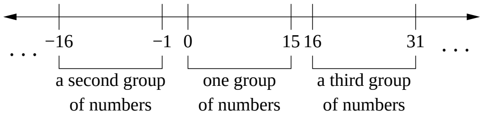
Using more bits to represent the answer is not an attractive solution, since we might then want to use more bits for the inputs, which in turn requires more bits for the outputs, and so on. We cannot build something supporting an infinite number of bits. Instead, we choose a value for N and build an add unit that adds two N-bit numbers and produces an N-bit sum (and some overflow indicators, which we discuss in the next set of notes). The diagram to the right shows how we might draw such a device, with two N-bit numbers entering at from the top, and the N-bit sum coming out from the bottom.



The function used for N-bit unsigned addition is addition modulo 2^N. In a practical sense, you can think of this function as simply keeping the last N bits of the answer; other bits are simply discarded. In the example to the right, we add 12 and 6 to obtain 18, but then discard the extra bit on the left, so the add unit produces 2 (an overflow).

$$\begin{array}{rl} \texttt{1100} & (12) \\ \underline{\texttt{+ 0110}} & (6) \\ \cancel{\texttt{1}}\texttt{0010} & (2) \end{array}$$

Modular arithmetic defines a way of performing arithmetic for a finite number of possible values, usually integers. As a concrete example, let's use modulo 16, which corresponds to the addition unit for our 4-bit examples.



Starting with the full range of integers, we can define equivalence classes for groups of 16 integers by simply breaking up the number line into contiguous groups, starting with the numbers 0 to 15, as shown to the right. The numbers -16 to -1 form a group, as do the numbers from 16 to 31. An infinite number of groups are defined in this manner.

You can think of these groups as defining **equivalence classes** modulo 16. All of the first numbers in the groups are equivalent modulo 16. All of the second numbers in the groups are equivalent modulo 16. And so forth. Mathematically, we say that two numbers A and B are equivalent modulo 16, which we write as

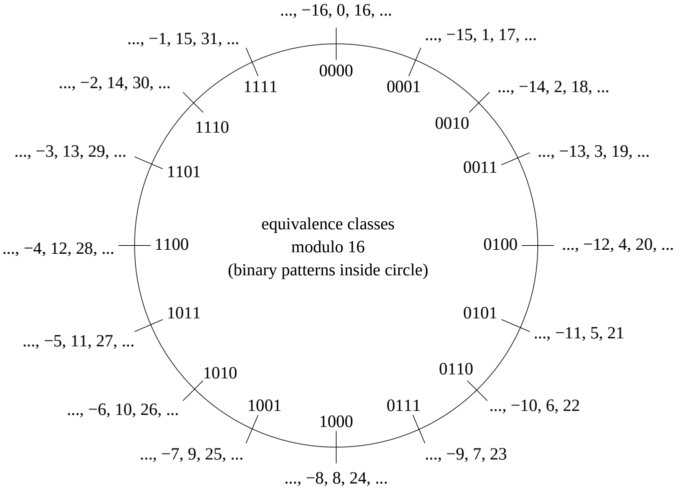
$$(A = B) \bmod 16$$

if and only if $A = B + 16k$ for some integer k.

It is worth noting that equivalence as defined by a particular modulus distributes over addition and multiplication. If, for example, we want to find the equivalence class for $(A+B) \bmod 16$, we can find the equivalence classes for A (call it C) and B (call it D) and then calculate the equivalence class of $(C + D) \bmod 16$. As a concrete example of distribution over multiplication, given $(A = 1,083,102,112 \times 7,323,127) \bmod 10$, find A. For this problem, we note that the first number is equivalent to 2 mod 10, while the second number is equivalent to 7 mod 10. We then write $(A = 2 \times 7) \bmod 10$, and, since $2 \times 7 = 14$, we have $(A = 4) \bmod 10$.

1.2.4 Deriving 2's Complement

Given these equivalence classes, we might instead choose to draw a circle to identify the equivalence classes and to associate each class with one of the sixteen possible 4-bit patterns, as shown to the right. Using this circle representation, we can add by counting clockwise around the circle, and we can subtract by counting in a counterclockwise direction around the circle. With an unsigned representation, we choose to use the group from $[0, 15]$ (the middle group in the diagram markings to the right) as the number represented by each of the patterns. Overflow occurs with unsigned addition (or subtraction) because we can only choose one value for each binary pattern.



In fact, we can choose any single value for each pattern to create a representation, and our add unit will always produce results that are correct modulo 16. Look back at our overflow example, where we added 12 and 6 to obtain 2, and notice that $(2 = 18) \bmod 16$. Normally, only a contiguous sequence of integers makes a useful representation, but we do not have to restrict ourselves to non-negative numbers.

The 2's complement representation can then be defined by choosing a set of integers balanced around zero from the groups. In the circle diagram, for example, we might choose to represent numbers in the range $[-7, 7]$ when using 4 bits. What about the last pattern, 1000? We could choose to represent either -8 or 8. The number of arithmetic operations that overflow is the same with both choices (the choices are symmetric around 0, as are the combinations of input operands that overflow), so we gain nothing in that sense from either choice. If we choose to represent -8, however, notice that all patterns starting with a 1 bit then represent negative numbers. No such simple check arises with the opposite choice, and thus an N-bit 2's complement representation is defined to represent the range $[-2^{N-1}, 2^{N-1}-1]$, with patterns chosen as shown in the circle.

1.2.5 An Algebraic Approach

Some people prefer an algebraic approach to understanding the definition of 2's complement, so we present such an approach next. Let's start by writing $f(A, B)$ for the result of our add unit:

$$f(A, B) = (A + B) \bmod 2^N$$

We assume that we want to represent a set of integers balanced around 0 using our signed representation, and that we will use the same binary patterns as we do with an unsigned representation to represent non-negative numbers. Thus, with an N-bit representation, the patterns in the range $[0, 2^{N-1} - 1]$ are the same as those used with an unsigned representation. In this case, we are left with all patterns beginning with a 1 bit.

The question then is this: given an integer k, $2^{N-1} > k > 0$, for which we want to find a pattern to represent $-k$, and any integer $m \geq 0$ that we might want to add to $-k$, can we find another integer $p > 0$ such that

$$(-k + m = p + m) \bmod 2^N \quad ? \tag{1}$$

If we can, we can use p's representation to represent $-k$ and our unsigned addition unit $f(A, B)$ will work correctly.

To find the value p, start by subtracting m from both sides of Equation (1) to obtain:

$$(-k = p) \bmod 2^N \tag{2}$$

Note that $(2^N = 0) \bmod 2^N$, and add this equation to Equation (2) to obtain

$$(2^N - k = p) \bmod 2^N$$

Let $p = 2^N - k$. For example, if $N = 4$, $k = 3$ gives $p = 16 - 3 = 13$, which is the pattern 1101. With $N = 4$ and $k = 5$, we obtain $p = 16 - 5 = 11$, which is the pattern 1011. In general, since $2^{N-1} > k > 0$, we have $2^{N-1} < p < 2^N$. But these patterns are all unused—they all start with a 1 bit!—so the patterns that we have defined for negative numbers are disjoint from those that we used for positive numbers, and the meaning of each pattern is unambiguous. The algebraic definition of bit patterns for negative numbers also matches our circle diagram from the last section exactly, of course.

1.2.6 Negating 2's Complement Numbers

The algebraic approach makes understanding negation of an integer represented using 2's complement fairly straightforward, and gives us an easy procedure for doing so. Recall that given an integer k in an N-bit 2's complement representation, the N-bit pattern for $-k$ is given by $2^N - k$ (also true for $k = 0$ if we keep only the low N bits of the result). But $2^N = (2^N - 1) + 1$. Note that $2^N - 1$ is the pattern of all 1 bits. Subtracting any value k from this value is equivalent to simply flipping the bits, changing 0s to 1s and 1s to 0s. (This operation is called a **1's complement**, by the way.) We then add 1 to the result to find the pattern for $-k$.

Negation can overflow, of course. Try finding the negative pattern for -8 in 4-bit 2's complement.

Finally, be aware that people often overload the term 2's complement and use it to refer to the operation of negation in a 2's complement representation. In our class, we try avoid this confusion: 2's complement is a representation for signed integers, and negation is an operation that one can apply to a signed integer (whether the representation used for the integer is 2's complement or some other representation for signed integers).

ECE120: Introduction to Computer Engineering

Notes Set 1.3 Overflow Conditions

This set of notes discusses the overflow conditions for unsigned and 2's complement addition. For both types, we formally prove that the conditions that we state are correct. Many of our faculty want our students to learn to construct formal proofs, so we plan to begin exposing you to this process in our classes. Prof. Lumetta is a fan of Prof. George Polya's educational theories with regard to proof techniques, and in particular the idea that one builds up a repertoire of approaches by seeing the approaches used in practice.

1.3.1 Implication and Mathematical Notation

Some of you may not have been exposed to basics of mathematical logic, so let's start with a brief introduction to implication. We'll use variables p and q to represent statements that can be either true or false. For example, p might represent the statement, "Jan is an ECE student," while q might represent the statement, "Jan works hard." The **logical complement** or **negation** of a statement p, written for example as "not p," has the opposite truth value: if p is true, not p is false, and if p is false, not p is true.

An **implication** is a logical relationship between two statements. The implication itself is also a logical statement, and may be true or false. In English, for example, we might say, "If p, q." In mathematics, the same implication is usually written as either "q if p" or "$p \rightarrow q$," and the latter is read as, "p implies q." Using our example values for p and q, we can see that $p \rightarrow q$ is true: "Jan is an ECE student" does in fact imply that "Jan works hard!"

The implication $p \rightarrow q$ is only considered false if p is true and q is false. In all other cases, the implication is true. This definition can be a little confusing at first, so let's use another example to see why. Let p represent the statement "Entity X is a flying pig," and let q represent the statement, "Entity X obeys air traffic control regulations." Here the implication $p \rightarrow q$ is again true: flying pigs do not exist, so p is false, and thus "$p \rightarrow q$" is true—for any value of statement q!

Given an implication "$p \rightarrow q$," we say that the **converse** of the implication is the statement "$q \rightarrow p$," which is also an implication. In mathematics, the converse of $p \rightarrow q$ is sometimes written as "q only if p." The converse of an implication may or may not have the same truth value as the implication itself. Finally, we frequently use the shorthand notation, "p if and only if q," (or, even shorter, "p iff q") to mean "$p \rightarrow q$ *and* $q \rightarrow p$." This last statement is true only when both implications are true.

1.3.2 Overflow for Unsigned Addition

Let's say that we add two N-bit unsigned numbers, A and B. The N-bit unsigned representation can represent integers in the range $[0, 2^N - 1]$. Recall that we say that the addition operation has overflowed if the number represented by the N-bit pattern produced for the sum does not actually represent the number $A + B$.

For clarity, let's name the bits of A by writing the number as $a_{N-1}a_{N-2}...a_1a_0$. Similarly, let's write B as $b_{N-1}b_{N-2}...b_1b_0$. Name the sum $C = A + B$. The sum that comes out of the add unit has only N bits, but recall that we claimed in class that the overflow condition for unsigned addition is given by the **carry** out of the most significant bit. So let's write the sum as $c_Nc_{N-1}c_{N-2}...c_1c_0$, realizing that c_N is the carry out and not actually part of the sum produced by the add unit.

Theorem: Addition of two N-bit unsigned numbers $A = a_{N-1}a_{N-2}...a_1a_0$ and $B = b_{N-1}b_{N-2}...b_1b_0$ to produce sum $C = A + B = c_Nc_{N-1}c_{N-2}...c_1c_0$, overflows if and only if the carry out c_N of the addition is a 1 bit.

Proof: Let's start with the "if" direction. In other words, $c_N = 1$ implies overflow. Recall that unsigned addition is the same as base 2 addition, except that we discard bits beyond c_{N-1} from the sum C. The bit c_N has place value 2^N, so, when $c_N = 1$ we can write that the correct sum $C \geq 2^N$. But no value that large can be represented using the N-bit unsigned representation, so we have an overflow.

The other direction ("only if") is slightly more complex: we need to show that overflow implies that $c_N = 1$. We use a range-based argument for this purpose. Overflow means that the sum C is outside the representable range $[0, 2^N - 1]$. Adding two non-negative numbers cannot produce a negative number, so the sum can't be smaller than 0. Overflow thus implies that $C \geq 2^N$.

Does that argument complete the proof? No, because some numbers, such as 2^{N+1}, are larger than 2^N, but do not have a 1 bit in the Nth position when written in binary. We need to make use of the constraints on A and B implied by the possible range of the representation.

In particular, given that A and B are represented as N-bit unsigned values, we can write

$$\begin{array}{rcl} 0 \leq & A & \leq 2^N - 1 \\ 0 \leq & B & \leq 2^N - 1 \end{array}$$

We add these two inequalities and replace $A + B$ with C to obtain

$$0 \leq \quad C \quad \leq 2^{N+1} - 2$$

Combining the new inequality with the one implied by the overflow condition, we obtain

$$2^N \leq \quad C \quad \leq 2^{N+1} - 2$$

All of the numbers in the range allowed by this inequality have $c_N = 1$, completing our proof.

1.3.3 Overflow for 2's Complement Addition

Understanding overflow for 2's complement addition is somewhat trickier, which is why the problem is a good one for you to think about on your own first. Our operands, A and B, are now two N-bit 2's complement numbers. The N-bit 2's complement representation can represent integers in the range $[-2^{N-1}, 2^{N-1} - 1]$. Let's start by ruling out a case that we can show never leads to overflow.

Lemma: Addition of two N-bit 2's complement numbers A and B does not overflow if one of the numbers is negative and the other is not.

Proof: We again make use of the constraints implied by the fact that A and B are represented as N-bit 2's complement values. We can assume **without loss of generality**[1], or **w.l.o.g.**, that $A < 0$ and $B \geq 0$.

Combining these constraints with the range representable by N-bit 2's complement, we obtain

$$\begin{array}{rcl} -2^{N-1} \leq & A & < 0 \\ 0 \leq & B & < 2^{N-1} \end{array}$$

We add these two inequalities and replace $A + B$ with C to obtain

$$-2^{N-1} \leq \quad C \quad < 2^{N-1}$$

But anything in the range specified by this inequality can be represented with N-bit 2's complement, and thus the addition does not overflow.

[1] This common mathematical phrasing means that we are using a problem symmetry to cut down the length of the proof discussion. In this case, the names A and B aren't particularly important, since addition is commutative ($A + B = B + A$). Thus the proof for the case in which A is negative (and B is not) is identical to the case in which B is negative (and A is not), except that all of the names are swapped. The term "without loss of generality" means that we consider the proof complete even with additional assumptions, in our case that $A < 0$ and $B \geq 0$.

We are now ready to state our main theorem. For convenience, let's use different names for the actual sum $C = A + B$ and the sum S returned from the add unit. We define S as the number represented by the bit pattern produced by the add unit. When overflow occurs, $S \neq C$, but we always have $(S = C) \bmod 2^N$.

Theorem: Addition of two N-bit 2's complement numbers A and B overflows if and only if one of the following conditions holds:

1. $A < 0$ and $B < 0$ and $S \geq 0$
2. $A \geq 0$ and $B \geq 0$ and $S < 0$

Proof: We once again start with the "if" direction. That is, if condition 1 or condition 2 holds, we have an overflow. The proofs are straightforward. Given condition 1, we can add the two inequalities $A < 0$ and $B < 0$ to obtain $C = A + B < 0$. But $S \geq 0$, so clearly $S \neq C$, thus overflow has occurred.

Similarly, if condition 2 holds, we can add the inequalities $A \geq 0$ and $B \geq 0$ to obtain $C = A + B \geq 0$. Here we have $S < 0$, so again $S \neq C$, and we have an overflow.

We must now prove the "only if" direction, showing that any overflow implies either condition 1 or condition 2. By the **contrapositive**[2] of our Lemma, we know that if an overflow occurs, either both operands are negative, or they are both positive.

Let's start with the case in which both operands are negative, so $A < 0$ and $B < 0$, and thus the real sum $C < 0$ as well. Given that A and B are represented as N-bit 2's complement, they must fall in the representable range, so we can write

$$\begin{array}{rcl} -2^{N-1} \leq & A & < 0 \\ -2^{N-1} \leq & B & < 0 \end{array}$$

We add these two inequalities and replace $A + B$ with C to obtain

$$-2^N \leq \quad C \quad < 0$$

Given that an overflow has occurred, C must fall outside of the representable range. Given that $C < 0$, it cannot be larger than the largest possible number representable using N-bit 2's complement, so we can write

$$-2^N \leq \quad C \quad < -2^{N-1}$$

We now add 2^N to each part to obtain

$$0 \leq \quad C + 2^N \quad < 2^{N-1}$$

This range of integers falls within the representable range for N-bit 2's complement, so we can replace the middle expression with S (equal to C modulo 2^N) to find that

$$0 \leq \quad S \quad < 2^{N-1}$$

Thus, if we have an overflow and both $A < 0$ and $B < 0$, the resulting sum $S \geq 0$, and condition 1 holds.

The proof for the case in which we observe an overflow when both operands are non-negative ($A \geq 0$ and $B \geq 0$) is similar, and leads to condition 2. We again begin with inequalities for A and B:

$$\begin{array}{rcl} 0 \leq & A & < 2^{N-1} \\ 0 \leq & B & < 2^{N-1} \end{array}$$

We add these two inequalities and replace $A + B$ with C to obtain

$$0 \leq \quad C < \quad 2^N$$

[2] If we have a statement of the form (p implies q), its contrapositive is the statement (not q implies not p). Both statements have the same truth value. In this case, we can turn our Lemma around as stated.

Given that an overflow has occurred, C must fall outside of the representable range. Given that $C \geq 0$, it cannot be smaller than the smallest possible number representable using N-bit 2's complement, so we can write

$$2^{N-1} \leq \quad C \quad < 2^N$$

We now subtract 2^N to each part to obtain

$$-2^{N-1} \leq \quad C - 2^N \quad < 0$$

This range of integers falls within the representable range for N-bit 2's complement, so we can replace the middle expression with S (equal to C modulo 2^N) to find that

$$-2^{N-1} \leq \quad S \quad < 0$$

Thus, if we have an overflow and both $A \geq 0$ and $B \geq 0$, the resulting sum $S < 0$, and condition 2 holds.

Thus overflow implies either condition 1 or condition 2, completing our proof.

ECE120: Introduction to Computer Engineering

Notes Set 1.4 Logic Operations

This set of notes briefly describes a generalization to truth tables, then introduces Boolean logic operations as well as notational conventions and tools that we use to express general functions on bits. We illustrate how logic operations enable us to express functions such as overflow conditions concisely, then show by construction that a small number of logic operations suffices to describe any operation on any number of bits. We close by discussing a few implications and examples.

1.4.1 Truth Tables

You have seen the basic form of truth tables in the textbook and in class. Over the semester, we will introduce several extensions to the basic concept, mostly with the goal of reducing the amount of writing necessary when using truth tables. For example, the truth table to the right uses two generalizations to show the carry out C (also the unsigned overflow indicator) and the sum S produced by adding two 2-bit unsigned numbers. First, rather than writing each input bit separately, we have grouped pairs of input bits into the numbers A and B. Second, we have defined multiple output columns so as to include both bits of S as well as C in the same table. Finally, we have grouped the two bits of S into one column.

inputs		outputs	
A	B	C	S
00	00	0	00
00	01	0	01
00	10	0	10
00	11	0	11
01	00	0	01
01	01	0	10
01	10	0	11
01	11	1	00
10	00	0	10
10	01	0	11
10	10	1	00
10	11	1	01
11	00	0	11
11	01	1	00
11	10	1	01
11	11	1	10

Keep in mind as you write truth tables that only rarely does an operation correspond to a simple and familiar process such as addition of base 2 numbers. We had to choose the unsigned and 2's complement representations carefully to allow ourselves to take advantage of a familiar process. In general, for each line of a truth table for an operation, you may need to make use of the input representation to identify the input values, calculate the operation's result as a value, and then translate the value back into the correct bit pattern using the output representation. Signed magnitude addition, for example, does not always correspond to base 2 addition: when the signs of the two input operands differ, one should instead use base 2 subtraction. For other operations or representations, base 2 arithmetic may have no relevance at all.

1.4.2 Boolean Logic Operations

In the middle of the 19th century, George Boole introduced a set of logic operations that are today known as **Boolean logic** (also as **Boolean algebra**). These operations today form one of the lowest abstraction levels in digital systems, and an understanding of their meaning and use is critical to the effective development of both hardware and software.

You have probably seen these functions many times already in your education—perhaps first in set-theoretic form as Venn diagrams. However, given the use of common English words *with different meanings* to name some of the functions, and the sometimes confusing associations made even by engineering educators, we want to provide you with a concise set of definitions that generalizes correctly to more than two operands. You may have learned these functions based on truth values (true and false), but we define them based on bits, with 1 representing true and 0 representing false.

Table 1 on the next page lists logic operations. The first column in the table lists the name of each function. The second provides a fairly complete set of the notations that you are likely to encounter for each function, including both the forms used in engineering and those used in mathematics. The third column defines the function's value for two or more input operands (except for NOT, which operates on a single value). The last column shows the form generally used in logic schematics/diagrams and mentions the important features used in distinguishing each function (in pictorial form usually called a **gate**, in reference to common physical implementations) from the others.

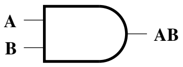
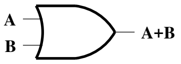
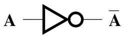
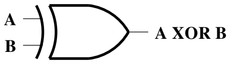
Function	Notation	Explanation	Schematic
AND	A AND B AB $A \cdot B$ $A \times B$ $A \wedge B$	the "all" function: result is 1 iff **all** input operands are equal to 1	 flat input, round output
OR	A OR B $A + B$ $A \vee B$	the "any" function: result is 1 iff **any** input operand is equal to 1	 round input, pointed output
NOT	NOT A A' $\overline{A}$ $\neg A$	logical complement/negation: NOT 0 is 1, and NOT 1 is 0	 triangle and circle
XOR exclusive OR	A XOR B $A \oplus B$	the "odd" function: result is 1 iff an **odd** number of input operands are equal to 1	 OR with two lines on input side
English "or"	A, B, or C	the "one of" function: result is 1 iff exactly **one of** the input operands is equal to 1	(not used)

Table 1: Boolean logic operations, notation, definitions, and symbols.

The first function of importance is **AND**. Think of **AND** as the "all" function: given a set of input values as operands, AND evaluates to 1 if and only if *all* of the input values are 1. The first notation line simply uses the name of the function. In Boolean algebra, AND is typically represented as multiplication, and the middle three forms reflect various ways in which we write multiplication. The last notational variant is from mathematics, where the AND function is formally called **conjunction**.

The next function of importance is **OR**. Think of **OR** as the "any" function: given a set of input values as operands, OR evaluates to 1 if and only if *any* of the input values is 1. The actual number of input values equal to 1 only matters in the sense of whether it is at least one. The notation for OR is organized in the same way as for AND, with the function name at the top, the algebraic variant that we will use in class—in this case addition—in the middle, and the mathematics variant, in this case called **disjunction**, at the bottom.

The definition of Boolean OR is not the same as our use of the word "or" in English. For example, if you are fortunate enough to enjoy a meal on a plane, you might be offered several choices: "Would you like the chicken, the beef, or the vegetarian lasagna today?" Unacceptable answers to this English question include: "Yes," "Chicken and lasagna," and any other combination that involves more than a single choice!

You may have noticed that we might have instead mentioned that AND evaluates to 0 if any input value is 0, and that OR evaluates to 0 if all input values are 0. These relationships reflect a mathematical duality underlying Boolean logic that has important practical value in terms of making it easier for humans to digest complex logic expressions. We will talk more about duality later in the course, but you should learn some of the practical value now: if you are trying to evaluate an AND function, look for an input with value 0; if you are trying to evaluate an OR function, look for an input with value 1. If you find such an input, you know the function's value without calculating any other input values.

We next consider the **logical complement** function, **NOT**. The NOT function is also called **negation**. Unlike our first two functions, NOT accepts only a single operand, and reverses its value, turning 0 into 1 and 1 into 0. The notation follows the same pattern: a version using the function name at the top, followed by two variants used in Boolean algebra, and finally the version frequently used in mathematics. For the NOT gate, or **inverter**, the circle is actually the important part: the triangle by itself merely copies the input. You will see the small circle added to other gates on both inputs and outputs; in both cases the circle implies a NOT function.

Last among the Boolean logic functions, we have the **XOR**, or **exclusive OR** function. Think of XOR as the "odd" function: given a set of input values as operands, XOR evaluates to 1 if and only if *an odd number* of the input values are 1. Only two variants of XOR notation are given: the first using the function name, and the second used with Boolean algebra. Mathematics rarely uses this function.

Finally, we have included the meaning of the word "or" in English as a separate function entry to enable you to compare that meaning with the Boolean logic functions easily. Note that many people refer to English' use of the word "or" as "exclusive" because one true value excludes all others from being true. Do not let this human language ambiguity confuse you about XOR! For all logic design purposes, *XOR is the odd function.*

The truth table to the right provides values illustrating these functions operating on three inputs. The AND, OR, and XOR functions are all associative—(A op B) op C = A op (B op C)—and commutative—A op B = B op A, as you may have already realized from their definitions.

inputs			outputs			
A	B	C	ABC	$A+B+C$	$\overline{A}$	$A \oplus B \oplus C$
0	0	0	0	0	1	0
0	0	1	0	1	1	1
0	1	0	0	1	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	1
1	0	1	0	1	0	0
1	1	0	0	1	0	0
1	1	1	1	1	0	1

1.4.3 Overflow as Logic Expressions

In the last set of notes, we discussed overflow conditions for unsigned and 2's complement representations. Let's use Boolean logic to express these conditions.

We begin with addition of two 1-bit unsigned numbers. Call the two input bits A_0 and B_0. If you write a truth table for this operation, you'll notice that overflow occurs only when all (two) bits are 1. If either bit is 0, the sum can't exceed 1, so overflow cannot occur. In other words, overflow in this case can be written using an AND operation:

$$A_0B_0$$

The truth table for adding two 2-bit unsigned numbers is four times as large, and seeing the structure may be difficult. One way of writing the expression for overflow of 2-bit unsigned addition is as follows:

$$A_1B_1 + (A_1 + B_1)A_0B_0$$

This expression is slightly trickier to understand. Think about the place value of the bits. If both of the most significant bits—those with place value 2—are 1, we have an overflow, just as in the case of 1-bit addition. The A_1B_1 term represents this case. We also have an overflow if one or both (the OR) of the most significant bits are 1 and the sum of the two next significant bits—in this case those with place value 1—generates a carry.

The truth table for adding two 3-bit unsigned numbers is probably not something that you want to write out. Fortunately, a pattern should start to become clear with the following expression:

$$A_2B_2 + (A_2 + B_2)A_1B_1 + (A_2 + B_2)(A_1 + B_1)A_0B_0$$

In the 2-bit case, we mentioned the "most significant bit" and the "next most significant bit" to help you see the pattern. The same reasoning describes the first two product terms in our overflow expression for 3-bit unsigned addition (but the place values are 4 for the most significant bit and 2 for the next most significant bit). The last term represents the overflow case in which the two least significant bits generate a carry which then propagates up through all of the other bits because at least one of the two bits in every position is a 1.

$$\begin{array}{rl} & A_{N-1}A_{N-2}\ldots A_2A_1A_0 \\ + & B_{N-1}B_{N-2}\ldots B_2B_1B_0 \\ \hline & S_{N-1}\,S_{N-2}\ldots S_2\,S_1\,S_0 \end{array}$$

The overflow condition for addition of two N-bit 2's complement numbers can be written fairly concisely in terms of the first bits of the two numbers and the first bit of the sum. Recall that overflow in this case depends only on whether the three numbers are negative or non-negative, which is given by the most significant bit. Given the bit names as shown to the right, we can write the overflow condition as follows:

$$A_{N-1}\;B_{N-1}\;\overline{S_{N-1}} + \overline{A_{N-1}}\;\overline{B_{N-1}}\;S_{N-1}$$

The overflow condition does of course depend on all of the bits in the two numbers being added. In the expression above, we have simplified the form by using S_{N-1}. But S_{N-1} depends on the bits A_{N-1} and B_{N-1} as well as the carry out of bit $(N-2)$.

Later in this set of notes, we present a technique with which you can derive an expression for an arbitrary Boolean logic function. As an exercise, after you have finished reading these notes, try using that technique to derive an overflow expression for addition of two N-bit 2's complement numbers based on A_{N-1}, B_{N-1}, and the carry out of bit $(N-2)$ (and into bit $(N-1)$), which we might call C_{N-1}. You might then calculate C_{N-1} in terms of the rest of the bits of A and B using the expressions for unsigned overflow just discussed. In the next month or so, you will learn how to derive more compact expressions yourself from truth tables or other representations of Boolean logic functions.

1.4.4 Logical Completeness

Why do we feel that such a short list of functions is enough? If you think about the number of possible functions on N bits, you might think that we need many more functions to be able to manipulate bits. With 10 bits, for example, there are 2^{1024} such functions. Obviously, some of them have never been used in any computer system, but maybe we should define at least a few more logic operations? In fact, we do not even need XOR. The functions AND, OR, and NOT are sufficient, even if we only allow two input operands for AND and OR!

The theorem below captures this idea, called **logical completeness**. In this case, we claim that the set of functions {AND, OR, NOT} is sufficient to express any operation on any finite number of variables, where each variable is a bit.

Theorem: Given enough 2-input AND, 2-input OR, and 1-input NOT functions, one can express any Boolean logic function on any finite number of variables.

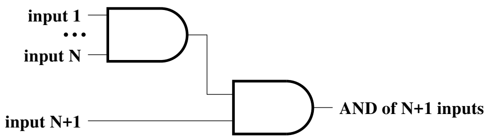
The proof of our theorem is **by construction**. In other words, we show a systematic approach for transforming an arbitrary Boolean logic function on an arbitrary number of variables into a form that uses only AND, OR, and NOT functions on one or two operands. As a first step, we remove the restriction on the number of inputs for the AND and OR functions. For this purpose, we state and prove two **lemmas**, which are simpler theorems used to support the proof of a main theorem.

Lemma 1: Given enough 2-input AND functions, one can express an AND function on any finite number of variables.

Proof: We prove the Lemma **by induction**.[3] Denote the number of inputs to a particular AND function by N.

The base case is $N = 2$. Such an AND function is given.

To complete the proof, we need only show that, given any number of AND functions with up to N inputs, we can express an AND function with $N+1$ inputs. To do so, we need merely use one 2-input AND function to join together the result of an N-input AND function with an additional input, as illustrated to the right.



[3] We assume that you have seen proof by induction previously.

Lemma 2: Given enough 2-input OR functions, one can express an OR function on any finite number of variables.

Proof: The proof of Lemma 2 is identical in structure to that of Lemma 1, but uses OR functions instead of AND functions.

Let's now consider a small subset of functions on N variables. For any such function, you can write out the truth table for the function. The output of a logic function is just a bit, either a 0 or a 1. Let's consider the set of functions on N variables that produce a 1 for exactly one combination of the N variables. In other words, if you were to write out the truth table for such a function, exactly one row in the truth table would have output value 1, while all other rows had output value 0.

Lemma 3: Given enough AND functions and 1-input NOT functions, one can express any Boolean logic function that produces a 1 for exactly one combination of any finite number of variables.

Proof: The proof of Lemma 3 is by construction. Let N be the number of variables on which the function operates. We construct a **minterm** on these N variables, which is an AND operation on each variable or its complement. The minterm is specified by looking at the unique combination of variable values that produces a 1 result for the function. Each variable that must be a 1 is included as itself, while each variable that must be a 0 is included as the variable's complement (using a NOT function). The resulting minterm produces the desired function exactly. When the variables all match the values for which the function should produce 1, the inputs to the AND function are all 1, and the function produces 1. When any variable does not match the value for which the function should produce 1, that variable (or its complement) acts as a 0 input to the AND function, and the function produces a 0, as desired.

The table below shows all eight minterms for three variables.

inputs			outputs							
A	B	C	$\overline{A}\,\overline{B}\,\overline{C}$	$\overline{A}\,\overline{B}\,C$	$\overline{A}\,B\,\overline{C}$	$\overline{A}\,B\,C$	$A\,\overline{B}\,\overline{C}$	$A\,\overline{B}\,C$	$A\,B\,\overline{C}$	$A\,B\,C$
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

We are now ready to prove our theorem.

Proof (of Theorem): Any given function on N variables produces the value 1 for some set of combinations of inputs. Let's say that M such combinations produce 1. Note that $M \leq 2^N$. For each combination that produces 1, we can use Lemma 1 to construct an N-input AND function. Then, using Lemma 3, we can use as many as M NOT functions and the N-input AND function to construct a minterm for that input combination. Finally, using Lemma 2, we can construct an M-input OR function and OR together all of the minterms. The result of the OR is the desired function. If the function should produce a 1 for some combination of inputs, that combination's minterm provides a 1 input to the OR, which in turn produces a 1. If a combination should produce a 0, its minterm does not appear in the OR; all other minterms produce 0 for that combination, and thus all inputs to the OR are 0 in such cases, and the OR produces 0, as desired.

The construction that we used to prove logical completeness does not necessarily help with efficient design of logic functions. Think about some of the expressions that we discussed earlier in these notes for overflow conditions. How many minterms do you need for N-bit unsigned overflow? A single Boolean logic function can be expressed in many different ways, and learning how to develop an efficient implementation of a function as well as how to determine whether two logic expressions are identical without actually writing out truth tables are important engineering skills that you will start to learn in the coming months.

1.4.5 Implications of Logical Completeness

If logical completeness doesn't really help us to engineer logic functions, why is the idea important? Think back to the layers of abstraction and the implementation of bits from the first couple of lectures. Voltages are real numbers, not bits. *The device layer implementations of Boolean logic functions must abstract away the analog properties of the physical system.* Without such abstraction, we must think carefully about analog issues such as noise every time we make use of a bit! Logical completeness assures us that no matter what we want to do with bits, implementating a handful of operations correctly is enough to guarantee that we never have to worry.

A second important value of logical completeness is as a tool in screening potential new technologies for computers. If a new technology does not allow implementation of a logically complete set of functions, the new technology is extremely unlikely to be successful in replacing the current one.

That said, {AND, OR, and NOT} is not the only logically complete set of functions. In fact, our current complementary metal-oxide semiconductor (CMOS) technology, on which most of the computer industry is now built, does not directly implement these functions, as you will see later in our class.

The functions that are implemented directly in CMOS are NAND and NOR, which are abbreviations for AND followed by NOT and OR followed by NOT, respectively. Truth tables for the two are shown to the right.

inputs		outputs	
		$\overline{AB}$	$\overline{A+B}$
A	B	A NAND B	A NOR B
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

Either of these functions by itself forms a logically complete set. That is, both the set {NAND} and the set {NOR} are logically complete. For now, we leave the proof of this claim to you. Remember that all you need to show is that you can implement any set known to be logically complete, so in order to prove that {NAND} is logically complete (for example), you need only show that you can implement AND, OR, and NOT using only NAND.

1.4.6 Examples and a Generalization

Let's use our construction to solve a few examples. We begin with the functions that we illustrated with the first truth table from this set of notes, the carry out C and sum S of two 2-bit unsigned numbers. Since each output bit requires a separate expression, we now write S_1S_0 for the two bits of the sum. We also need to be able to make use of the individual bits of the input values, so we write these as A_1A_0 and B_1B_0, as shown on the left below. Using our construction from the logical completeness theorem, we obtain the equations on the right. You should verify these expressions yourself.

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

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

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

$$\begin{aligned}
S_0 &= \overline{A_1}\ \overline{A_0}\ \overline{B_1}\ B_0 + \overline{A_1}\ \overline{A_0}\ B_1\ B_0 + \overline{A_1}\ A_0\ \overline{B_1}\ \overline{B_0} + \\
&\quad \overline{A_1}\ A_0\ B_1\ \overline{B_0} + A_1\ \overline{A_0}\ \overline{B_1}\ B_0 + A_1\ \overline{A_0}\ B_1\ B_0 + \\
&\quad A_1\ A_0\ \overline{B_1}\ \overline{B_0} + A_1\ A_0\ B_1\ \overline{B_0}
\end{aligned}$$

Now let's consider a new function. Given an 8-bit 2's complement number, $A = A_7A_6A_5A_4A_3A_2A_1A_0$, we want to compare it with the value -1. We know that we can construct this function using AND, OR, and NOT, but how? We start by writing the representation for -1, which is 11111111. If the number A matches that representation, we want to produce a 1. If the number A differs in any bit, we want to produce a 0. The desired function has exactly one combination of inputs that produces a 1, so in fact we need only one minterm! In this case, we can compare with -1 by calculating the expression:

$$A_7 \cdot A_6 \cdot A_5 \cdot A_4 \cdot A_3 \cdot A_2 \cdot A_1 \cdot A_0$$

Here we have explicitly included multiplication symbols to avoid confusion with our notation for groups of bits, as we used when naming the individual bits of A.

In closing, we briefly introduce a generalization of logic operations to groups of bits. Our representations for integers, real numbers, and characters from human languages all use more than one bit to represent a given value. When we use computers, we often make use of multiple bits in groups in this way. A **byte**, for example, today means an ordered group of eight bits. We can extend our logic functions to operate on such groups by pairing bits from each of two groups and performing the logic operation on each pair. For example, given $A = A_7A_6A_5A_4A_3A_2A_1A_0 = 01010101$ and $B = B_7B_6B_5B_4B_3B_2B_1B_0 = 11110000$, we calculate A AND B by computing the AND of each pair of bits, A_7 AND B_7, A_6 AND B_6, and so forth, to produce the result 01010000, as shown to the right. In the same way, we can extend other logic operations, such as OR, NOT, and XOR, to operate on bits of groups.

$$\begin{array}{rr}
 & A\ 01010101 \\
\text{AND} & B\ 11110000 \\
\hline
 & 01010000
\end{array}$$

ECE120: Introduction to Computer Engineering

Notes Set 1.5 Programming Concepts and the C Language

This set of notes introduces the C programming language and explains some basic concepts in computer programming. Our purpose in showing you a high-level language at this early stage of the course is to give you time to become familiar with the syntax and meaning of the language, not to teach you how to program. Throughout this semester, we will use software written in C to demonstrate and validate the digital system design material in our course. Towards the end of the semester, you will learn to program computers using instructions and assembly language. In ECE 220, you will make use of the C language to write programs, at which point already being familiar with the language will make the material easier to master. These notes are meant to complement the introduction provided by Patt and Patel.

After a brief introduction to the history of C and the structure of a program written in C, we connect the idea of representations developed in class to the data types used in high-level languages. We next discuss the use of variables in C, then describe some of the operators available to the programmer, including arithmetic and logic operators. The notes next introduce C functions that support the ability to read user input from the keyboard and to print results to the monitor. A description of the structure of statements in C follows, explaining how programs are executed and how a programmer can create statements for conditional execution as well as loops to perform repetitive tasks. The main portion of the notes concludes with an example program, which is used to illustrate both the execution of C statements as well as the difference between variables in programs and variables in algebra.

The remainder of the notes covers more advanced topics. First, we describe how the compilation process works, illustrating how a program written in a high-level language is transformed into instructions. You will learn this process in much more detail in ECE 220. Second, we briefly introduce the C preprocessor. Finally, we discuss implicit and explicit data type conversion in C. *Sections marked with an asterisk are provided solely for your interest, but you probably need to learn this material in later classes.*

1.5.1 The C Programming Language

Programming languages attempt to bridge the semantic gap between human descriptions of problems and the relatively simple instructions that can be provided by an instruction set architecture (ISA). Since 1954, when the Fortran language first enabled scientists to enter FORmulae symbolically and to have them TRANslated automatically into instructions, people have invented thousands of computer languages.

The C programming language was developed by Dennis Ritchie at Bell Labs in order to simplify the task of writing the Unix operating system. The C language provides a fairly transparent mapping to typical ISAs, which makes it a good choice both for system software such as operating systems and for our class. The **syntax** used in C—that is, the rules that one must follow to write valid C programs—has also heavily influenced many other more recent languages, such as C++, Java, and Perl.

```
int
main ()
{
    int answer = 42;        /* the Answer! */

    printf ("The answer is %d.\n", answer);

    /* Our work here is done.
       Let's get out of here!  */
    return 0;
}
```

For our purposes, a C program consists of a set of **variable declarations** and a sequence of **statements**. Both of these parts are written into a single C function called `main`, which executes when the program starts. A simple example appears to the right. The program uses one variable called `answer`, which it initializes to the value 42. The program prints a line of output to the monitor for the user, then terminates using the `return` statement. **Comments** for human readers begin with the characters `/*` (a slash followed by an asterisk) and end with the characters `*/` (an asterisk followed by a slash). The C language ignores white space in programs, so we encourage you to use blank lines and extra spacing to make your programs easier to read.

The variables defined in the `main` function allow a programmer to associate arbitrary **symbolic names** (sequences of English characters, such as "sum" or "product" or "highScore") with specific types of data, such as a 16-bit unsigned integer or a double-precision floating-point number. In the example program above, the variable `answer` is declared to be a 32-bit 2's complement number.

Those with no programming experience may at first find the difference between variables in algebra and variables in programs slightly confusing. *As a program executes, the values of variables can change from step to step of execution.*

The statements in the `main` function are executed one by one until the program terminates. Programs are not limited to simple sequences of statements, however. Some types of statements allow a programmer to specify conditional behavior. For example, a program might only print out secret information if the user's name is "lUmeTTa." Other types of statements allow a programmer to repeat the execution of a group of statements until a condition is met. For example, a program might print the numbers from 1 to 10, or ask for input until the user types a number between 1 and 10. The order of statement execution is well-defined in C, but the statements in `main` do not necessarily make up an algorithm: *we can easily write a C program that never terminates.*

If a program terminates, the `main` function returns an integer to the operating system, usually by executing a `return` statement, as in the example program. By convention, returning the value 0 indicates successful completion of the program, while any non-zero value indicates a program-specific error. However, `main` is not necessarily a function in the mathematical sense because *the value returned from* `main` *is not necessarily unique for a given set of input values to the program.* For example, we can write a program that selects a number from 1 to 10 at random and returns the number to the operating system.

1.5.2 Data Types

As you know, modern digital computers represent all information with binary digits (0s and 1s), or **bits**. Whether you are representing something as simple as an integer or as complex as an undergraduate thesis, the data are simply a bunch of 0s and 1s inside a computer. For any given type of information, a human selects a data type for the information. A **data type** (often called just a **type**) consists of both a size in bits and a representation, such as the 2's complement representation for signed integers, or the ASCII representation for English text. A **representation** is a way of encoding the things being represented as a set of bits, with each bit pattern corresponding to a unique object or thing.

A typical ISA supports a handful of data types in hardware in the sense that it provides hardware support for operations on those data types. The arithmetic logic units (ALUs) in most modern processors, for example, support addition and subtraction of both unsigned and 2's complement representations, with the specific data type (such as 16- or 64-bit 2's complement) depending on the ISA. Data types and operations not supported by the ISA must be handled in software using a small set of primitive operations, which form the **instructions** available in the ISA. Instructions usually include data movement instructions such as loads and stores and control instructions such as branches and subroutine calls in addition to arithmetic and logic operations. The last quarter of our class covers these concepts in more detail and explores their meaning using an example ISA from the textbook.

In class, we emphasized the idea that digital systems such as computers do not interpret the meaning of bits. Rather, they do exactly what they have been designed to do, even if that design is meaningless. If, for example, you store a sequence of ASCII characters in a computer's memory as and then write computer instructions to add consecutive groups of four characters as 2's complement integers and to print the result to the screen, the computer will not complain about the fact that your code produces meaningless garbage.

In contrast, high-level languages typically require that a programmer associate a data type with each datum in order to reduce the chance that the bits making up an individual datum are misused or misinterpreted accidentally. Attempts to interpret a set of bits differently usually generate at least a warning message, since such re-interpretations of the bits are rarely intentional and thus rarely correct. A compiler—a program that transforms code written in a high-level language into instructions—can also generate the proper type conversion instructions automatically when the transformations are intentional, as is often the case with arithmetic.

Some high-level languages, such as Java, prevent programmers from changing the type of a given datum. If you define a type that represents one of your favorite twenty colors, for example, you are not allowed to turn a color into an integer, despite the fact that the color is represented as a handful of bits. Such languages are said to be **strongly typed**.

The C language is not strongly typed, and programmers are free to interpret any bits in any manner they see fit. Taking advantage of this ability in any but a few exceptional cases, however, results in arcane and non-portable code, and is thus considered to be bad programming practice. We discuss conversion between types in more detail later in these notes.

Each high-level language defines a number of **primitive data types**, which are always available. Most languages, including C, also provide ways of defining new types in terms of primitive types, but we leave that part of C for ECE 220. The primitive data types in C include signed and unsigned integers of various sizes as well as single- and double-precision IEEE floating-point numbers.

	2's complement	unsigned
8 bits	`char`	`unsigned char`

16 bits	`short`
	`short int`
		`unsigned short`
		`unsigned short int`

32 bits	`int`	`unsigned`
		`unsigned int`

32 or 64 bits	`long`
	`long int`
		`unsigned long`
		`unsigned long int`

64 bits	`long long`
	`long long int`
		`unsigned long long`
		`unsigned long long int`

The primitive integer types in C include both unsigned and 2's complement representations. These types were originally defined so as to give reasonable performance when code was ported. In particular, the `int` type is intended to be the native integer type for the target ISA. Using data types supported directly in hardware is faster than using larger or smaller integer types. When C was standardized in 1989, these types were defined so as to include a range of existing C compilers rather than requiring all compilers to produce uniform results. At the time, most workstations and mainframes were 32-bit machines, while most personal computers were 16-bit machines, thus flexibility was somewhat desirable. For the GCC compiler on Linux, the C integer data types are defined in the table above. Although the `int` and `long` types are usually the same, there is a semantic difference in common usage. In particular, on most architectures and most compilers, a `long` has enough bits to identify a location in the computer's memory, while an `int` may not. When in doubt, the **size in bytes** of any type or variable can be found using the built-in C function `sizeof`.

	2's complement	unsigned
8 bits	`int8_t`	`uint8_t`
16 bits	`int16_t`	`uint16_t`
32 bits	`int32_t`	`uint32_t`
64 bits	`int64_t`	`uint64_t`

Over time, the flexibility of size in C types has become less important (except for the embedded markets, where one often wants even more accurate bit-width control), and the fact that the size of an `int` can vary from machine to machine and compiler to compiler has become more a source of headaches than a helpful feature. In the late 1990s, a new set of fixed-size types were recommended for inclusion in the C library, reflecting the fact that many companies had already developed and were using such definitions to make their programs platform-independent. We encourage you to make use of these types, which are shown in the table above. In Linux, they can be made available by including the `stdint.h` header file.

Floating-point types in C include `float` and `double`, which correspond respectively to single- and double-precision IEEE floating-point values. Although the 32-bit `float` type can save memory compared with use of 64-bit `double` values, C's math library works with double-precision values, and single-precision data are uncommon in scientific and engineering codes. In contrast, single-precision floating-point operations dominated the graphics industry until recently, and are still well-supported even on today's graphics processing units.

1.5.3 Variable Declarations

The function `main` executed by a program begins with a list of **variable declarations**. Each declaration consists of two parts: a data type specification and a comma-separated list of variable names. Each variable declared can also be **initialized** by assigning an initial value. A few examples appear below. Notice that one can initialize a variable to have the same value as a second variable.

```
int    x = 42;              /* a 2's complement variable, initially equal to 42          */
int    y = x;               /* a second 2's complement variable, initially equal to x     */
int    z;                   /* a third 2's complement variable with unknown initial value */
double a, b, c, pi = 3.1416; /*
                             * four double-precision IEEE floating-point variables
                             * a, b, and c are initially of unknown value, while pi is
                             * initially 3.1416
                             */
```

What happens if a programmer declares a variable but does not initialize it? Remember that bits can only be 0 or 1. An uninitialized variable does have a value, but its value is unpredictable. The compiler tries to detect uses of uninitialized variables, but sometimes it fails to do so, so *until you are more familiar with programming, you should always initialize every variable.*

Variable names, also called **identifiers**, can include both letters and digits in C. Good programming style requires that programmers select variable names that are meaningful and are easy to distinguish from one another. Single letters are acceptable in some situations, but longer names with meaning are likely to help people (including you!) understand your program. Variable names are also case-sensitive in C, which allows programmers to use capitalization to differentiate behavior and meaning, if desired. Some programs, for example, use identifiers with all capital letters to indicate variables with values that remain constant for the program's entire execution. However, the fact that identifiers are case-sensitive also means that a programmer can declare distinct variables named `variable`, `Variable`, `vaRIable`, `vaRIabLe`, and `VARIABLE`. We strongly discourage you from doing so.

1.5.4 Expressions and Operators

The `main` function also contains a sequence of statements. A statement is a complete specification of a single step in the program's execution. We explain the structure of statements in the next section. Many statements in C include one or more **expressions**, which represent calculations such as arithmetic, comparisons, and logic operations. Each expression is in turn composed of **operators** and **operands**. Here we give only a brief introduction to some of the operators available in the C language. We deliberately omit operators with more complicated meanings, as well as operators for which the original purpose was to make writing common operations a little shorter. For the interested reader, both the textbook and ECE 220 give more detailed introductions. The table to the right gives examples for the operators described here.

```
int i = 42, j = 1000;
/* i = 0x0000002A, j = 0x000003E8 */

        i + j   → 1042
    i - 4 * j   → -3958
           -j   → -1000
        j / i   → 23
        j % i   → 42
        i & j   → 40       /* 0x00000028 */
        i | j   → 1002     /* 0x000003EA */
        i ∧ j   → 962      /* 0x000003C2 */
           ∼i   → -43      /* 0xFFFFFFD5 */
    (∼i) >> 2   → -11      /* 0xFFFFFFF5 */
∼((∼i) >> 4)    → 2        /* 0x00000002 */
       j >> 4   → 62       /* 0x0000003E */
       j << 3   → 8000     /* 0x00001F40 */
        i > j   → 0
       i <= j   → 1
       i == j   → 0
        j = i   → 42       /* ...and j is changed!  */
```

Arithmetic operators in C include addition (+), subtraction (-), negation (a minus sign not preceded by another expression), multiplication (*), division (/), and modulus (%). No exponentiation operator exists; instead, library routines are defined for this purpose as well as for a range of more complex mathematical functions.

C also supports **bitwise operations** on integer types, including AND (&), OR (|), XOR (^), NOT (~), and left (<<) and right (>>) bit shifts. Right shifting a signed integer results in an **arithmetic right shift** (the sign bit is copied), while right shifting an unsigned integer results in a **logical right shift** (0 bits are inserted).

A range of **relational** or **comparison operators** are available, including equality (==), inequality (!=), and relative order (<, <=, >=, and >). All such operations evaluate to 1 to indicate a true relation and 0 to indicate a false relation. Any non-zero value is considered to be true for the purposes of tests (for example, in an `if` statement or a `while` loop) in C—these statements are explained later in these notes.

Assignment of a new value to a variable uses a single equal sign (=) in C. For example, the expression `A = B` copies the value of variable `B` into variable `A`, overwriting the bits representing the previous value of `A`. *The use of two equal signs for an equality check and a single equal sign for assignment is a common source of errors,* although modern compilers generally detect and warn about this type of mistake. Assignment in C does not solve equations, even simple equations. Writing "`A-4=B`", for example, generates a compiler error. You must solve such equations yourself to calculate the desired new value of a single variable, such as "`A=B+4`." For the purposes of our class, you must always write a single variable on the left side of an assignment, and can write an arbitrary expression on the right side.

Many operators can be combined into a single expression. When an expression has more than one operator, which operator is executed first? The answer depends on the operators' **precedence**, a well-defined order on operators that specifies how to resolve the ambiguity. In the case of arithmetic, the C language's precedence specification matches the one that you learned in elementary school. For example, `1+2*3` evaluates to 7, not to 9, because multiplication has precedence over addition. For non-arithmetic operators, or for any case in which you do not know the precedence specification for a language, *do not look it up—other programmers will not remember the precedence ordering, either!* Instead, add parentheses to make your expressions clear and easy to understand.

1.5.5 Basic I/O

The `main` function returns an integer to the operating system. Although we do not discuss how additional functions can be written in our class, we may sometimes make use of functions that have been written in advance by making **calls** to those functions. A **function call** is type of expression in C, but we leave further description for ECE 220. In our class, we make use of only two additional functions to enable our programs to receive input from a user via the keyboard and to write output to the monitor for a user to read.

Let's start with output. The `printf` function allows a program to print output to the monitor using a programmer-specific format. The "f" in `printf` stands for "formatted."[4] When we want to use `printf`, we write a expression with the word `printf` followed by a parenthesized, comma-separated list of expressions. The expressions in this list are called the **arguments** to the `printf` function.

The first argument to the `printf` function is a format string—a sequence of ASCII characters between quotation marks—which tells the function what kind of information we want printed to the monitor as well as how to format that information. The remaining arguments are C expressions that give `printf` a copy of any values that we want printed.

How does the format string specify the format? Most of the characters in the format string are simply printed to the monitor. In the first example shown to on the next page, we use `printf` to print a hello message followed by an ASCII newline character to move to the next line on the monitor.

[4] The original, unformatted variant of printing was never available in the C language. Go learn Fortran.

The percent sign—“%”—is used as an **escape character** in the `printf` function. When “%” appears in the format string, the function examines the next character in the format string to determine which format to use, then takes the next expression from the sequence of arguments and prints the value of that expression to the monitor. Evaluating an expression generates a bunch of bits, so it is up to the programmer to ensure that those bits are not misinterpreted. In other words, the programmer must make sure that the number and types of formatted values match the number and types of arguments passed to `printf` (not counting the format string itself). The `printf` function returns the number of characters printed to the monitor.

```
        printf ("Hello, world!\n");
output: Hello, world! [and a newline]

        printf ("To %x or not to %d...\n", 190, 380 / 2);
output: To be or not to 190... [and a newline]

        printf ("My favorite number is %c%c.\n", 0x34, ’0’+2);
output: My favorite number is 42. [and a newline]

        printf ("What is pi?  %f or %e?\n", 3.1416, 3.1416);
output: What is pi?  3.141600 or 3.141600e+00? [and a newline]
```

escape sequence	`printf` function’s interpretation of expression bits
`%c`	2’s complement integer printed as an ASCII character
`%d`	2’s complement integer printed as decimal
`%e`	double printed in decimal scientific notation
`%f`	double printed in decimal
`%u`	unsigned integer printed as decimal
`%x`	integer printed as hexadecimal (lower case)
`%X`	integer printed as hexadecimal (upper case)
`%%`	a single percent sign

A program can read input from the user with the `scanf` function. The user enters characters in ASCII using the keyboard, and the `scanf` function converts the user’s input into C primitive types, storing the results into variables. As with `printf`, the `scanf` function takes a format string followed by a comma-separated list of arguments. Each argument after the format string provides `scanf` with the memory address of a variable into which the function can store a result.

How does `scanf` use the format string? For `scanf`, the format string is usually just a sequence of conversions, one for each variable to be typed in by the user. As with `printf`, the conversions start with “%” and are followed by characters specifying the type of conversion to be performed. The first example shown to the right reads two integers. The conversions in the format string can be separated by spaces for readability, as shown in the example. The spaces are ignored by `scanf`. However, *any non-space characters in the format string must be typed exactly by the user!*

```
        int      a, b; /* example variables */
        char     c;
        unsigned u;
        double   d;
        float    f;

        scanf ("%d%d", &a, &b);   /* These have the */
        scanf ("%d %d", &a, &b);  /* same effect.   */
effect: try to convert two integers typed in decimal to
        2’s complement and store the results in a and b

        scanf ("%c%x %lf", &c, &u, &d);
effect: try to read an ASCII character into c, a value
        typed in hexadecimal into u, and a double-
        precision floating-point number into d

        scanf ("%lf %f", &d, &f);
effect: try to read two real numbers typed as decimal,
        convert the first to double-precision and store it
        in d, and convert the second to single-precision
        and store it in f
```

escape sequence	`scanf` function’s conversion to bits
`%c`	store one ASCII character (as `char`)
`%d`	convert decimal integer to 2’s complement
`%f`	convert decimal real number to float
`%lf`	convert decimal real number to double
`%u`	convert decimal integer to unsigned int
`%x`	convert hexadecimal integer to unsigned int
`%X`	(as above)

The remaining arguments to `scanf` specify memory addresses where the function can store the converted values. The ampersand (“&”) in front of each variable name in the examples is an operator that returns the address of a variable in memory. For each conversion in the format string, the `scanf` function tries to convert input from the user into the appropriate result, then stores the result in memory at the address given by the next argument. The programmer is responsible for ensuring that the number of conversions in the format string matches the number of arguments provided (not counting the format string itself). The programmer must also ensure that the type of information produced by each conversion can be stored at the address passed for that conversion—in other words, the address of a variable with the correct type must be provided. Modern compilers often detect missing `&` operators and incorrect variable types, but many only give warnings to the programmer. The `scanf` function itself cannot tell whether the arguments given to it are valid or not.

If a conversion fails—for example, if a user types "hello" when `scanf` expects an integer—`scanf` does not overwrite the corresponding variable and immediately stops trying to convert input. The `scanf` function returns the number of successful conversions, allowing a programmer to check for bad input from the user.

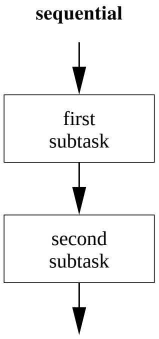
1.5.6 Types of Statements in C

Each statement in a C program specifies a complete operation. There are three types of statements, but two of these types can be constructed from additional statements, which can in turn be constructed from additional statements. The C language specifies no bound on this type of recursive construction, but code readability does impose a practical limit.

The three types are shown to the right. They are the **null statement**, **simple statements**, and **compound statements**. A null statement is just a semicolon, and a compound statement is just a sequence of statements surrounded by braces.

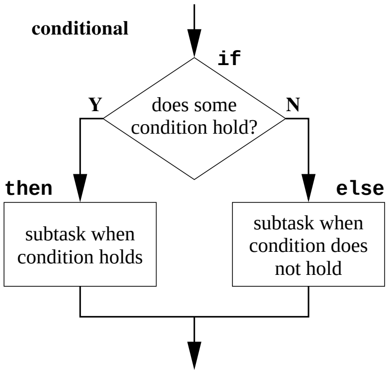
```
;                  /* a null statement (does nothing) */

A = B;             /* examples of simple statements   */
printf ("Hello, world!\n");

{                  /* a compound statement            */
    C = D;         /* (a sequence of statements       */
    N = 4;         /*  between braces)                */
    L = D - N;
}
```

Simple statements can take several forms. All of the examples shown to the right, including the call to `printf`, are simple statements consisting of a C expression followed by a semicolon. Simple statements can also consist of conditionals or iterations, which we introduce next.



Remember that after variable declarations, the `main` function contains a sequence of statements. These statements are executed one at a time in the order given in the program, as shown to the right for two statements. We say that the statements are executed in sequential order.

A program must also be able to execute statements only when some condition holds. In the C language, such a condition can be an arbitrary expression. The expression is first evaluated. If the result is 0, the condition is considered to be false. Any result other than 0 is considered to be true. The C statement for conditional execution is called an `if` statement. Syntactically, we put the expression for the condition in parentheses after the keyword `if` and follow the parenthesized expression with a compound statement containing the statements that should be executed when the condition is true. Optionally, we can append the keyword `else` and a second compound statement containing statements to be executed when the condition evaluates to false. The corresponding flow chart is shown to the right.


conditional
if
Y
does some condition hold?
N
then
else
subtask when condition holds
subtask when condition does not hold

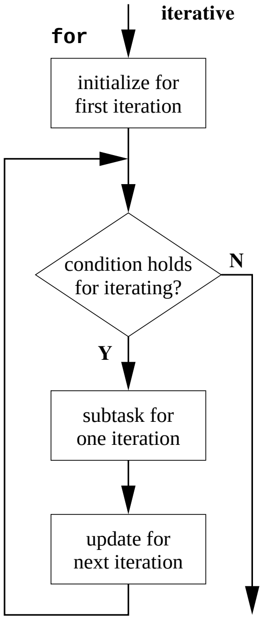
```
/* Set the variable y to the absolute value of variable x. */
if (0 <= x) {   /* Is x greater or equal to 0? */
    y = x;      /* Then block: assign x to y. */
} else {
    y = -x;     /* Else block: assign negative x to y. */
}
```

If instead we chose to assign the absolute value of variable x to itself, we can do so without an else block:

```
/* Set the variable x to its absolute value. */
if (0 > x) {    /* Is x less than 0? */
    x = -x;     /* Then block: assign negative x to x. */
}               /* No else block is given--no work is needed. */
```

Finally, we sometimes need to repeatedly execute a set of statements, either a fixed number of times or so long as some condition holds. We refer to such repetition as an **iteration** or a **loop**. In our class, we make use of C's for loop when we need to perform such a task. A for loop is structured as follows:

```
for ([initialization] ; [condition] ; [update]) {
    [subtask to be repeated]
}
```


iterative
for
initialize for first iteration
condition holds for iterating?
N
Y
subtask for one iteration
update for next iteration

A flow chart corresponding to execution of a for loop appears to the right. First, any initialization is performed. Then the condition—again an arbitrary C expression—is checked. If the condition evaluates to false (exactly 0), the loop is done. Otherwise, if the condition evaluates to true (any non-zero value), the statements in the compound statement, the subtask or **loop body**, are executed. The loop body can contain anything: a sequence of simple statements, a conditional, another loop, or even just an empty list. Once the loop body has finished executing, the for loop update rule is executed. Execution then checks the condition again, and this process repeats until the condition evaluates to 0. The for loop below, for example, prints the numbers from 1 to 42.

```
/* Print the numbers from 1 to 42.  */
for (i = 1; 42 >= i; i = i + 1) {
    printf ("%d\n", i);
}
```

1.5.7 Program Execution

We are now ready to consider the execution of a simple program, illustrating how variables change value from step to step and determine program behavior.

Let's say that two numbers are "friends" if they have at least one 1 bit in common when written in base 2. So, for example, 100_2 and 111_2 are friends because both numbers have a 1 in the bit with place value $2^2 = 4$. Similarly, 101_2 and 010_2 are not friends, since no bit position is 1 in both numbers.

The program to the right prints all friendships between numbers in the interval $[0, 7]$.

```
int
main ()
{
    int check;    /* number to check for friends                   */
    int friend;   /* a second number to consider as check's friend */

    /* Consider values of check from 0 to 7.  */
    for (check = 0; 8 > check; check = check + 1) {

        /* Consider values of friend from 0 to 7.  */
        for (friend = 0; 8 > friend; friend = friend + 1) {

            /* Use bitwise AND to see if the two share a 1 bit.  */
            if (0 != (check & friend)) {

                /* We have friendship!  */
                printf ("%d and %d are friends.\n", check, friend);
            }
        }
    }
}
```

The program uses two integer variables, one for each of the numbers that we consider. We use a `for` loop to iterate over all values of our first number, which we call `check`. The loop initializes `check` to 0, continues until check reaches 8, and adds 1 to check after each loop iteration. We use a similar `for` loop to iterate over all possible values of our second number, which we call `friend`. For each pair of numbers, we determine whether they are friends using a bitwise AND operation. If the result is non-zero, they are friends, and we print a message. If the two numbers are not friends, we do nothing, and the program moves on to consider the next pair of numbers.

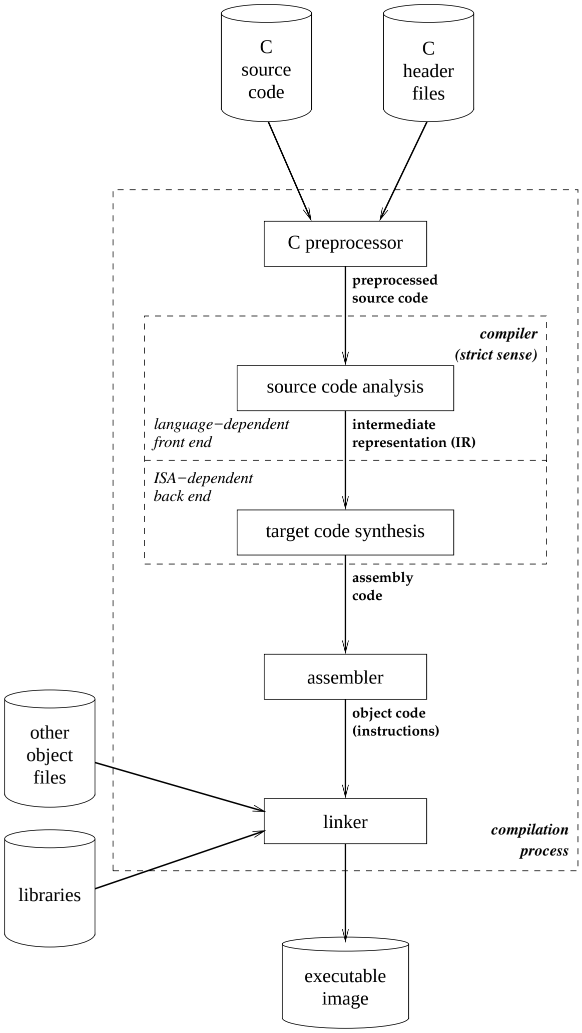
after executing...	`check` is...	and `friend` is...
(variable declarations)	unpredictable bits	unpredictable bits
`check = 0`	0	unpredictable bits
`8 > check`	0	unpredictable bits
`friend = 0`	0	0
`8 > friend`	0	0
`if (0 != (check & friend))`	0	0
`friend = friend + 1`	0	1
`8 > friend`	0	1
`if (0 != (check & friend))`	0	1
`friend = friend + 1`	0	2
(repeat last three lines six more times; number 0 has no friends!)		
`8 > friend`	0	8
`check = check + 1`	1	8
`8 > check`	1	8
`friend = 0`	1	0
`8 > friend`	1	0
`if (0 != (check & friend))`	1	0
`friend = friend + 1`	1	1
`8 > friend`	1	1
`if (0 != (check & friend))`	1	1
`printf ...`	1	1
(our first friend!?)		

Now let's think about what happens when this program executes. When the program starts, both variables are filled with random bits, so their values are unpredictable. The first step is the initialization of the first `for` loop, which sets `check` to 0. The condition for that loop is `8 > check`, which is true, so execution enters the loop body and starts to execute the first statement, which is our second `for` loop. The next step is then the initialization code for the second `for` loop, which sets `friend` to 0. The condition for the second loop is `8 > friend`, which is true, so execution enters the loop body and starts to execute the first statement, which is the `if` statement. Since both variables are 0, the `if` condition is false, and nothing is printed. Having finished the loop body for the inner loop (on `friend`), execution continues with the update rule for that loop—`friend = friend + 1`—then returns to check the loop's condition again. This process repeats, always finding that the number 0 (in `check`) is not friends (0 has no friends!) until `friend` reaches 8, at which point the inner loop condition becomes false. Execution then moves to the update rule for the first `for` loop, which increments `check`. Check is then compared with 8 to see if the loop is done. Since it is not, we once again enter the loop body and start the second `for` loop over. The initialization code again sets `friend` to 0, and we move forward as before. As you see above, the first time that we find our `if` condition to be true is when both `check` and `friend` are equal to 1.

Is that result what you expected? To learn that the number 1 is friends with itself? If so, the program works. If you assumed that numbers could not be friends with themselves, perhaps we should fix the bug? We could, for example, add another `if` statement to avoid printing anything when `check == friend`.

Our program, you might also realize, prints each pair of friends twice. The numbers 1 and 3, for example, are printed in both possible orders. To eliminate this redundancy, we can change the initialization in the second `for` loop, either to `friend = check` or to `friend = check + 1`, depending on how we want to define friendship (the same question as before: can a number be friends with itself?).

1.5.8 Compilation and Interpretation*



Many programming languages, including C, can be **compiled**, which means that the program is converted into instructions for a particular ISA before the program is run on a processor that supports that ISA. The figure to the right illustrates the compilation process for the C language. In this type of figure, files and other data are represented as cylinders, while rectangles represent processes, which are usually implemented in software. In the figure to the right, the outer dotted box represents the full compilation process that typically occurs when one compiles a C program. The inner dotted box represents the work performed by the **compiler** software itself. The cylinders for data passed between the processes that compose the full compilation process have been left out of the figure; instead, we have written the type of data being passed next to the arrows that indicate the flow of information from one process to the next.

The C preprocessor (described later in these notes) forms the first step in the compilation process. The preprocessor operates on the program's **source code** along with **header files** that describe data types and operations. The preprocessor merges these together into a single file of preprocessed source code. The preprocessed source code is then analyzed by the front end of the compiler based on the specific programming language being used (in our case, the C language), then converted by the back end of the compiler into instructions for the desired ISA. The output of a compiler is not binary instructions, however, but is instead a human-readable form of instructions called **assembly code**, which we cover in the last quarter of our class. A tool called an assembler then converts these human-readable instructions into bits that a processor can understand. If a program consists of multiple source files, or needs to make use of additional pre-programmed operations (such as math functions, graphics, or sound), a tool called a linker merges the object code of the program with those additional elements to form the final **executable image** for the program. The executable image is typically then stored on a disk, from which it can later be read into memory in order to allow a processor to execute the program.

Some languages are difficult or even impossible to compile. Typically, the behavior of these languages depends on input data that are only available when the program runs. Such languages can be **interpreted**: each step of the algorithm described by a program is executed by a software interpreter for the language. Languages such as Java, Perl, and Python are usually interpreted. Similarly, when we use software to simulate one ISA using another ISA, as we do at the end of our class with the LC-3 ISA described by the textbook, the simulator is a form of interpreter. In the lab, you will use a simulator compiled into and executing as x86 instructions in order to interpret LC-3 instructions. While a program is executing in an interpreter, enough information is sometimes available to compile part or all of the program to the processor's ISA as the program runs, a technique known as **"just in time" (JIT) compilation**.

1.5.9 The C Preprocessor*

The C language uses a preprocessor to support inclusion of common information (stored in header files) into multiple source files. The most frequent use of the preprocessor is to enable the unique definition of new data types and operations within header files that can then be included by reference within source files that make use of them. This capability is based on the **include directive**, `#include`, as shown here:

```
#include <stdio.h>       /* search in standard directories                    */
#include "my_header.h"   /* search in current followed by standard directories */
```

The preprocessor also supports integration of compile-time constants into source files before compilation. For example, many software systems allow the definition of a symbol such as `NDEBUG` (no debug) to compile without additional debugging code included in the sources. Two directives are necessary for this purpose: the **define directive**, `#define`, which provides a text-replacement facility, and **conditional inclusion** (or exclusion) of parts of a file within `#if`/`#else`/`#endif` directives. These directives are also useful in allowing a single header file to be included multiple times without causing problems, as C does not allow redefinition of types, variables, and so forth, even if the redundant definitions are identical. Most header files are thus wrapped as shown to the right.

```
#if !defined(MY_HEADER_H)
#define MY_HEADER_H
/* actual header file material goes here */
#endif /* MY_HEADER_H */
```

The preprocessor performs a simple linear pass on the source and does not parse or interpret any C syntax. Definitions for text replacement are valid as soon as they are defined and are performed until they are undefined or until the end of the original source file. The preprocessor does recognize spacing and will not replace part of a word, thus "`#define i 5`" will not wreak havoc on your `if` statements, but will cause problems if you name any variable `i`.

Using the text replacement capabilities of the preprocessor does have drawbacks, most importantly in that almost none of the information is passed on for debugging purposes.

1.5.10 Changing Types in C*

Changing the type of a datum is necessary from time to time, but sometimes a compiler can do the work for you. The most common form of **implicit type conversion** occurs with binary arithmetic operations. Integer arithmetic in C always uses types of at least the size of `int`, and all floating-point arithmetic uses `double`. If either or both operands have smaller integer types, or differ from one another, the compiler implicitly converts them before performing the operation, and the type of the result may be different from those of both operands. In general, the compiler selects the final type according to some preferred ordering in which floating-point is preferred over integers, unsigned values are preferred over signed values, and more bits are preferred over fewer bits. The type of the result must be at least as large as either argument, but is also at least as large as an `int` for integer operations and a `double` for floating-point operations.

Modern C compilers always extend an integer type's bit width before converting from signed to unsigned. The original C specification interleaved bit width extensions to `int` with sign changes, thus *older compilers may not be consistent, and implicitly require both types of conversion in a single operation may lead to portability bugs.*

The implicit extension to `int` can also be confusing in the sense that arithmetic that seems to work on smaller integers fails with larger ones. For example, multiplying two 16-bit integers set to 1000 and printing the result works with most compilers because the 32-bit `int` result is wide enough to hold the right answer. In contrast, multiplying two 32-bit integers set to 100,000 produces the wrong result because the high bits of the result are discarded before it can be converted to a larger type. For this operation to produce the correct result, one of the integers must be converted explicitly (as discussed later) before the multiplication.

Implicit type conversions also occur due to assignments. Unlike arithmetic conversions, the final type must match the left-hand side of the assignment (for example, a variable to which a result is assigned), and the compiler simply performs any necessary conversion. *Since the desired type may be smaller than the type of the value assigned, information can be lost.* Floating-point values are truncated when assigned to integers, and high bits of wider integer types are discarded when assigned to narrower integer types. *Note that a positive number may become a negative number when bits are discarded in this manner.*

Passing arguments to functions can be viewed as a special case of assignment. Given a function prototype, the compiler knows the type of each argument and can perform conversions as part of the code generated to pass the arguments to the function. Without such a prototype, or for functions with variable numbers of arguments, the compiler lacks type information and thus cannot perform necessary conversions, leading to unpredictable behavior. By default, however, the compiler extends any integer smaller than an `int` to the width of an `int` and converts `float` to `double`.

Occasionally it is convenient to use an **explicit type cast** to force conversion from one type to another. *Such casts must be used with caution, as they silence many of the warnings that a compiler might otherwise generate when it detects potential problems.* One common use is to promote integers to floating-point before an arithmetic operation, as shown to the right.

```
int
main ()
{
    int numerator = 10;
    int denominator = 20;

    printf ("%f\n", numerator / (double)denominator);
    return 0;
}
```

The type to which a value is to be converted is placed in parentheses in front of the value. In most cases, additional parentheses should be used to avoid confusion about the precedence of type conversion over other operations.

ECE120: Introduction to Computer Engineering

Notes Set 1.6 Summary of Part 1 of the Course

This short summary provides a list of both terms that we expect you to know and and skills that we expect you to have after our first few weeks together. The first part of the course is shorter than the other three parts, so the amount of material is necessarily less. These notes supplement the Patt and Patel textbook, so you will also need to read and understand the relevant chapters (see the syllabus) in order to master this material completely.

According to educational theory, 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).

This time, we'll list the skills first and leave the easy stuff for the next page. We expect you to be able to exercise the following skills:

- Represent decimal numbers with unsigned, 2's complement, and IEEE floating-point representations, and be able to calculate the decimal value represented by a bit pattern in any of these representations.
- Be able to negate a number represented in the 2's complement representation.
- Perform simple arithmetic by hand on unsigned and 2's complement representations, and identify when overflow occurs.
- Be able to write a truth table for a Boolean expression.
- Be able to write a Boolean expression as a sum of minterms.
- Know how to declare and initialize C variables with one of the primitive data types.

At a more abstract level, we expect you to be able to:

- Understand the value of using a common mathematical basis, such as modular arithmetic, in defining multiple representations (such as unsigned and 2's complement).
- Write Boolean expressions for the overflow conditions on both unsigned and 2's complement addition.
- Be able to write single `if` statements and `for` loops in C in order to perform computation.
- Be able to use `scanf` and `printf` for basic input and output in C.

And, at the highest level, we expect that you will be able to reason about and analyze problems in the following ways:

- Understand the tradeoffs between integer and floating-point representations for numbers.
- Understand logical completeness and be able to prove or disprove logical completeness for sets of logic functions.
- Understand the properties necessary in a representation: no ambiguity in meaning for any bit pattern, and agreement in advance on the meanings of all bit patterns.
- Analyze a simple, single-function C program and be able to explain its purpose.

You should recognize all of these terms and be able to explain what they mean. Note that we are not saying that you should, for example, be able to write down the ASCII representation from memory. In that example, knowing that it is a 7-bit representation used for English text is sufficient. You can always look up the detailed definition in practice.

- universal computational devices / computing machines
  - undecidable
  - the halting problem
- information storage in computers
  - bits
  - representation
  - data type
  - unsigned representation
  - 2's complement representation
  - IEEE floating-point representation
  - ASCII representation
- operations on bits
  - 1's complement operation
  - carry (from addition)
  - overflow (on any operation)
  - Boolean logic and algebra
  - logic functions/gates
  - truth table
  - AND/conjunction
  - OR/disjunction
  - NOT/logical complement/ (logical) negation/inverter
  - XOR
  - logical completeness
  - minterm
- mathematical terms
  - modular arithmetic
  - implication
  - contrapositive
  - proof approaches: by construction, by contradiction, by induction
  - without loss of generality (w.l.o.g.)
- high-level language concepts
  - syntax
  - variables
    - declaration
    - primitive data types
    - symbolic name/identifier
    - initialization
  - expression
  - statement
- C operators
  - operands
  - arithmetic
  - bitwise
  - comparison/relational
  - assignment
  - address
  - arithmetic shift
  - logical shift
  - precedence
- functions in C
  - `main`
  - function call
  - arguments
  - `printf` and `scanf`
    - format string
    - escape character
  - `sizeof`
- transforming tasks into programs
  - flow chart
  - sequential construct
  - conditional construct
  - iterative construct/iteration/loop
  - loop body
- C statements
  - statement: null, simple, compound
  - `if` statement
  - `for` loop
  - `return` statement

ECE120: Introduction to Computer Engineering

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.

2.1.1 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 $B = B_1B_0$.

$$\begin{aligned} C &= A_1B_1 + (A_1 + B_1)A_0B_0 && (3)\\ &= A_1B_1 + A_1A_0B_0 + A_0B_1B_0 && (4)\\ &= \overline{A_1}\ A_0\ B_1\ B_0 + A_1\ \overline{A_0}\ B_1\ \overline{B_0} + A_1\ \overline{A_0}\ B_1\ B_0 + \\ &\quad A_1\ A_0\ \overline{B_1}\ B_0 + A_1\ A_0\ B_1\ \overline{B_0} + A_1\ A_0\ B_1\ B_0 && (5)\end{aligned}$$

These three expressions are identical in the sense that they have the same truth tables—they are the same mathematical function. Equation (3) 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 (4) results from distributing the parenthesized OR in Equation (3). Equation (5) 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 (3) gives a count of 6, Equation (4) gives a count of 8, and Equation (5) gives a count of 24. Smaller numbers represent better expressions, so Equation (3) 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 (3)—two ANDs, two ORs, and 6 variables, a count of 12 for Equation (4)—three ANDS, one OR, and 8 variables, and a count of 31 for Equation (5)—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 (3) requires 3 gate delays, Equation (4) requires 2 gate delays, Equation (5) requires 2 or 3 gate delays, depending on whether we have variable complements available. Now Equation (4) looks more attractive: better performance than Equation (3) 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.

2.1.2 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 (5), 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 (4) is also in sum-of-products form. Equation (3), 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 (5) has 10 input combinations that produce 0, so the expression formed in this way is unpleasantly long:

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

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 $G \to F$ 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.

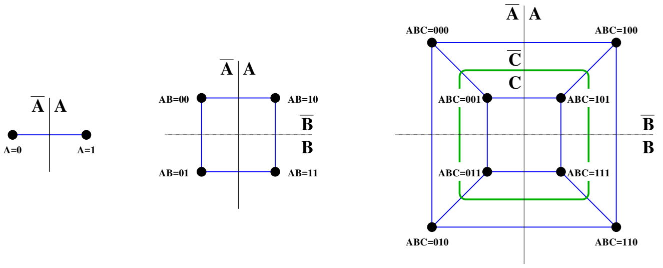
Let's say that we have expressed function F in sum-of-products form. All of the individual product terms in the expression are implicants of F. As a first step in simplification, we can ask: for each implicant, is it possible to remove any of the literals that make up the product? If we have an implicant 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).

2.1.3 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).



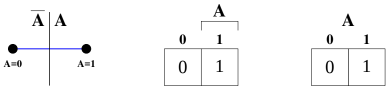
By viewing a function's domain in this way, we can make a connection between a product of literals and the structure of the domain. Let's use the 3-dimensional version as an example. We call the variables A, B, and C, and note that the cube has $2^3 = 8$ corners corresponding to the 2^3 possible combinations 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.

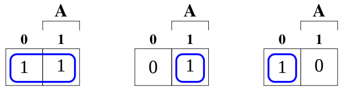
Continuing with products of two literals, we see that any product of two literals, such as $A\overline{B}$ or $\overline{B}C$, corresponds to an edge of our 3-dimensional cube. The edge includes $2^1 = 2$ corners. And, if a function evaluates to 1 on any of the 12 edges of the cube, the corresponding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

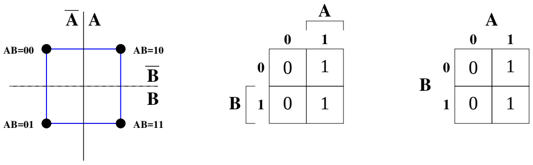


The first drawing shown to the right replicates our view of the 1-dimensional hypercube, corresponding to the domain of a function on one variable, in this case the variable A. To the right of the hypercube (line segment) are two variants of a Karnaugh map on one variable. The middle variant clearly 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.



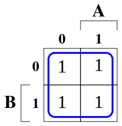
implicant: 1 implicant: A implicant: $\overline{A}$

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.*

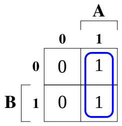


Let's now look at two-variable functions. We have replicated our drawing of the 2-dimensional hypercube (square) to the right along with two variants of Karnaugh maps on two variables. With only two variables (A and B), the extension is fairly straightforward, since we can use the second dimension of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

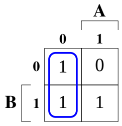
The number of possible products of literals grows rapidly with the number of variables. For two variables, nine are possible, as shown to the right. Notice that all implicants have two properties. First, they occupy contiguous regions of the grid. And, second, their height and width are always powers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



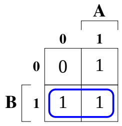
implicant: 1



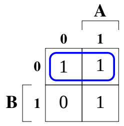
implicant: A



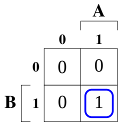
implicant: $\overline{A}$



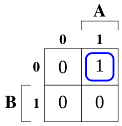
implicant: B



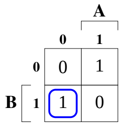
implicant: $\overline{B}$



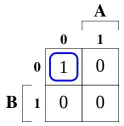
implicant: AB



implicant: $A\overline{B}$

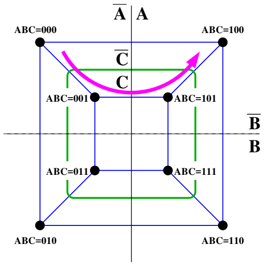


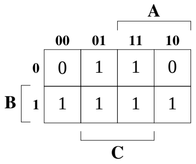
implicant: $\overline{A}B$

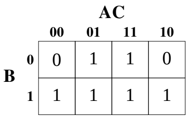


implicant: $\overline{A}\overline{B}$

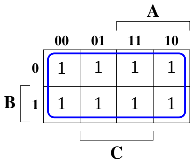
Three-variable functions are next. The cube diagram is again replicated to the right. But now we have a problem: how can we map four points (say, from the top half of the cube) into a line in such a way that any points connected by a blue line are adjacent in the K-map? The answer is that we cannot, 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.



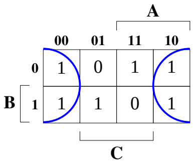




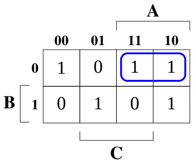
With three variables, we have 27 possible products of literals. You may have noticed that the count scales as 3^N for N variables; can you explain why? We illustrate several product terms below. Note that we sometimes need to wrap around the end of the K-map, but that if we account for wrapping, the 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.



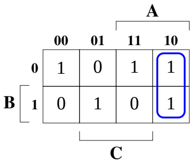
implicant: 1



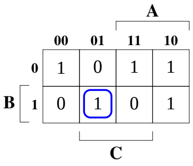
implicant: $\overline{C}$



implicant: $A\overline{B}$

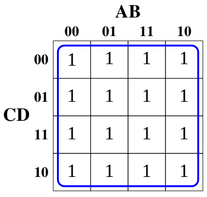


implicant: $A\overline{C}$

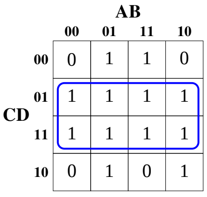


implicant: $\overline{A}BC$

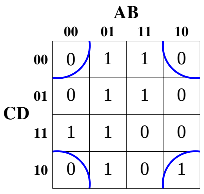
Let's keep going. With a function on four variables—A, B, C, and D—we can use a Gray code order on two of the variables in each dimension. Which variables go with which dimension in the grid really doesn't matter, so we'll assign AB to the horizontal dimension and CD to the vertical dimension. A few 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.*



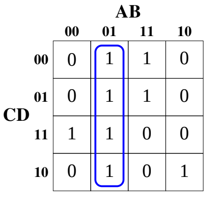
implicant: 1



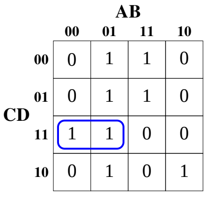
implicant: D



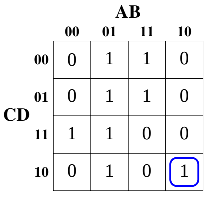
implicant: $\overline{B}\overline{D}$



implicant: $\overline{A}B$



implicant: $\overline{A}CD$

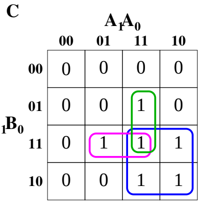


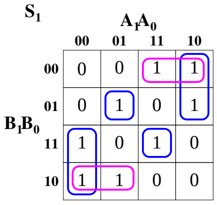
implicant: $A\overline{B}C\overline{D}$

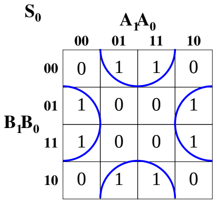
Finding a simple expression for a function using a K-map then consists of solving the following problem: pick a minimal set of prime implicants such that every 1 produced by the function is covered by at least one prime implicant. The metric that you choose to minimize the set may vary in practice, but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

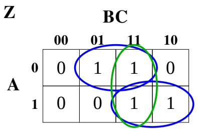
2.1.4 Canonical Forms

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 (3) to (5) for an example). But what if we rewrite the function using only prime implicants? Is the result unique? Unfortunately, no.

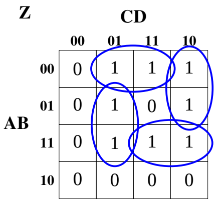
In general, *a sum of products is not unique (nor is a product of sums), even if the sum contains only prime implicants.*

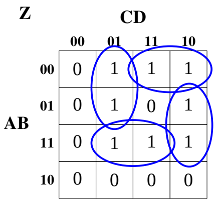
For example, consensus terms may or may not be included in our expressions. (They are necessary for reliable design of certain types of systems, as you will learn in a later ECE class.) The green ellipse in the K-map to the right represents the consensus term BC.



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

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:



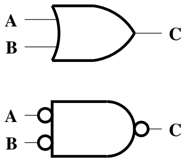


$$\begin{aligned} Z &= \overline{A}\,\overline{B}\,D + \overline{A}\,C\,\overline{D} + A\,B\,C + B\,\overline{C}\,D \\ Z &= \overline{A}\,\overline{B}\,C + B\,C\,\overline{D} + A\,B\,D + \overline{A}\,\overline{C}\,D \end{aligned}$$

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).

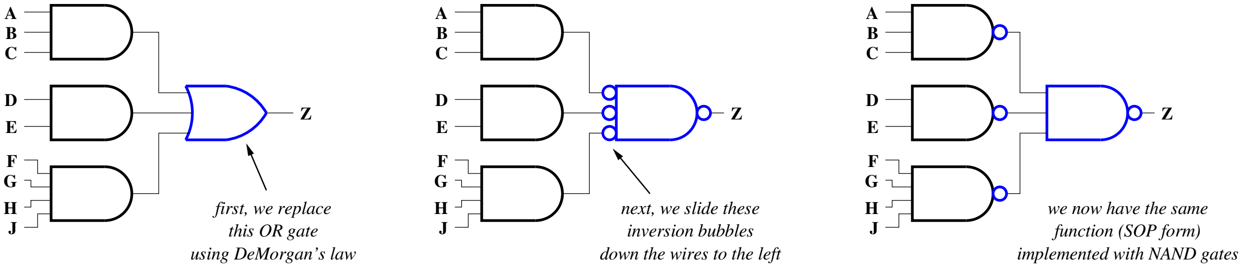
2.1.5 Two-Level Logic

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.

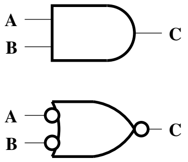


first, we replace this OR gate using DeMorgan's law

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?



2.1.6 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.

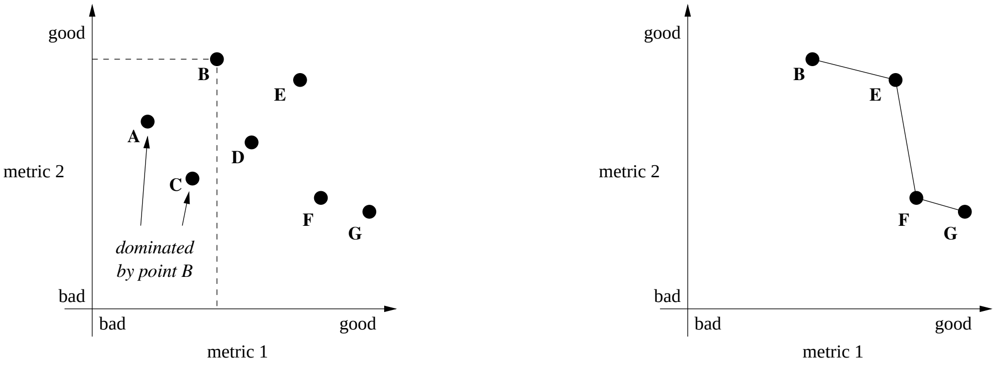
Some metrics may only allow discrete quantification. For example, one could choose to define compatibility with previous processor generations as binary: either an existing piece of software (or operating system) runs out of the box on your new processor, or it does not. If you want people who own that 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.

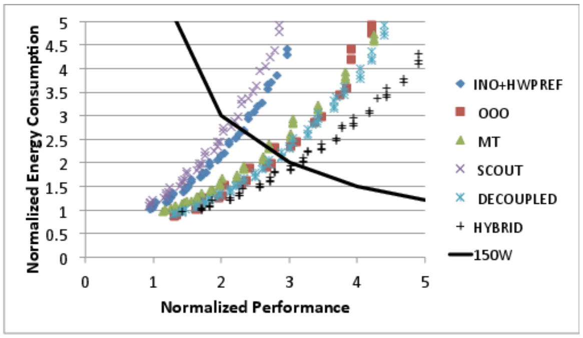
After you have tried to reduce your set of metrics as much as possible, or simplified them by turning them into thresholds, you should consider turning the last few metrics into a weighted linear sum. All remaining metrics must be quantifiable in this case. For example, if you are left with three metrics 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.



ECE120: Introduction to Computer Engineering

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.

2.2.1 Logic Properties

Table 2 (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 2. 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 2 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.

2.2.2 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 2: 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."

2.2.3 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.

$C_L[1]$ LB

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

$C_L[0]$ LB

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

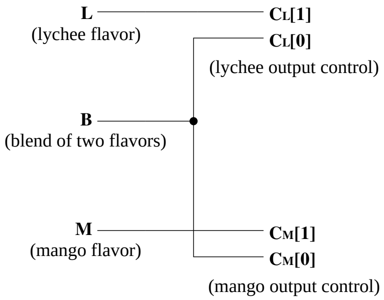
$C_M[1]$ LB

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

$C_M[0]$ LB

M	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.

$C_L[1]$ LB

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

$C_L[0]$ LB

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

$C_M[1]$ LB

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

$C_M[0]$ LB

M	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.*

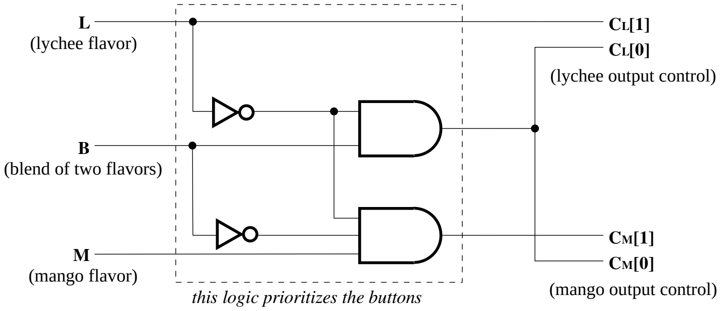
What can we do? When in doubt, it is safest to make choices and to add the new decisions to the specification rather than leaving output values specified as "don't care." For our ice cream dispenser logic, rather than leaving the outputs unspecified whenever a user presses more than one button, we could 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."

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

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

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

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

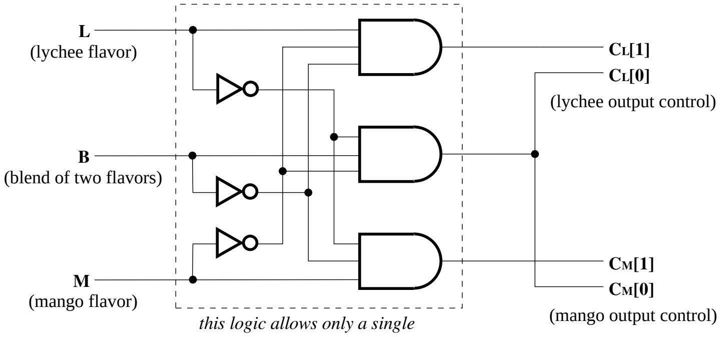


this logic prioritizes the buttons and passes only one at any time

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*

2.2.4 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.

ECE120: Introduction to Computer Engineering

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.

2.3.1 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.

2.3.2 Abstracting the Human Process

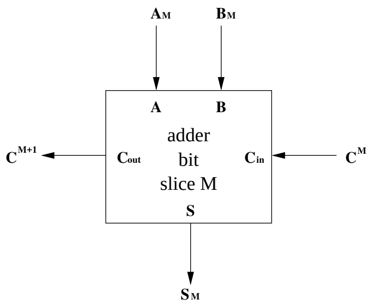
Think about how we as humans add two N-bit numbers, A and B. An illustration appears to the right, using $N = 8$. For now, let's assume that our numbers are stored in an unsigned representation. As you know, addition for 2's complement is identical except for the calculation of overflow. We start adding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

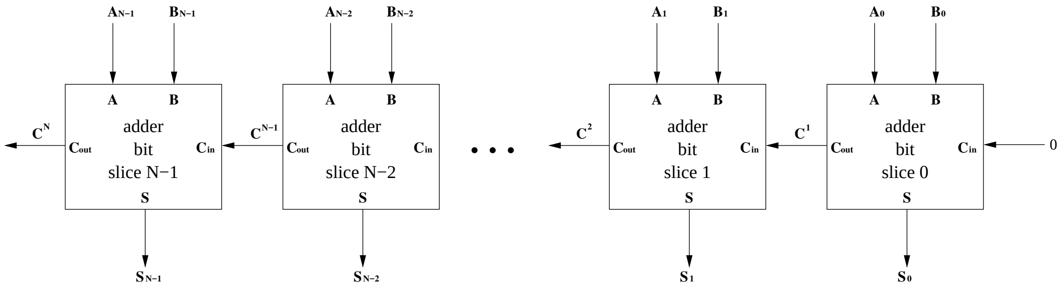
Focus now on the addition of a single column. Except for the first and last bits, which we might choose to handle slightly differently, the addition process is identical for any column. We add a carry in bit (possibly 0) with one bit from each of our numbers to produce a sum bit and a carry out bit for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



2.3.3 Designing the Logic

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

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

Cout

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

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

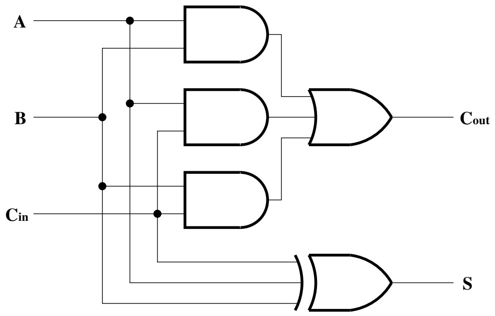
S

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

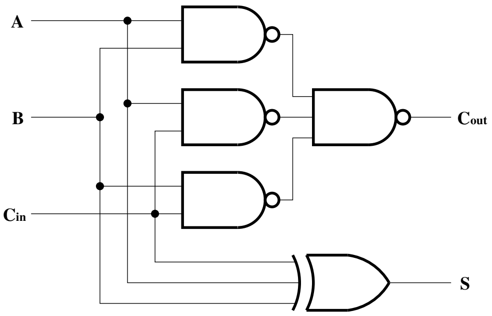
$$\begin{aligned} S &= A\,B\,C_{out} + A\,\overline{B}\,\overline{C_{out}} + \\ &\quad \overline{A}\,B\,\overline{C_{out}} + \overline{A}\,\overline{B}\,C_{out} \\ &= A \oplus B \oplus C_{out} \end{aligned}$$

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

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



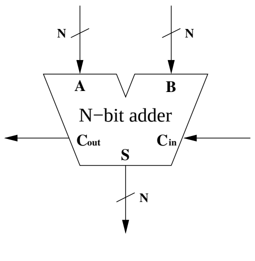
an adder bit slice using NAND gates



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.

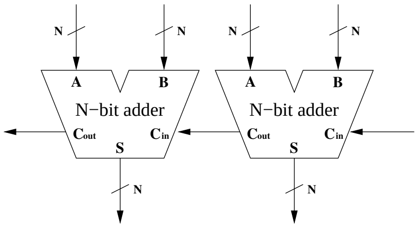
2.3.4 Adders and Word Size



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.

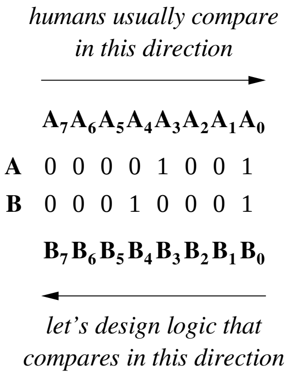
ECE120: Introduction to Computer Engineering

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.

2.4.1 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.


humans usually compare in this direction
$A_7A_6A_5A_4A_3A_2A_1A_0$
A 0 0 0 0 1 0 0 1
B 0 0 0 1 0 0 0 1
$B_7B_6B_5B_4B_3B_2B_1B_0$
let's design logic that compares in this direction

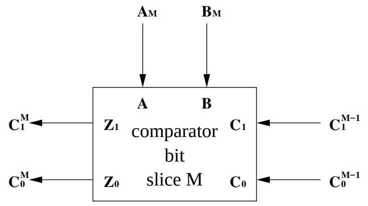
As humans, we typically start comparing at the most significant bit. After all, if we find a difference in that bit, we are done, saving ourselves some time. In the example to the right, we know that $A < B$ as soon as we reach bit 4 and observe that $A_4 < B_4$. If we instead start from the least significant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

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

2.4.2 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.

2.4.3 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.

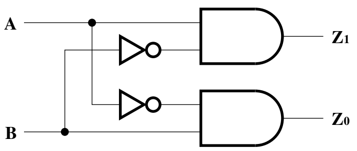
The truth table for bit 0 appears to the right (recall that we use Z_1 and Z_0 for the output names). Note that the bit 0 function has only two meaningful inputs—there is no bit to the right of bit 0. If the two inputs A and B are the same, we output equality. Otherwise, we do a 1-bit comparison and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

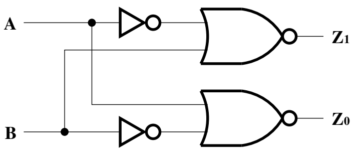
A	B	Z_1	Z_0
0	0	0	0
0	1	0	1
1	0	1	0
1	1	0	0

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

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

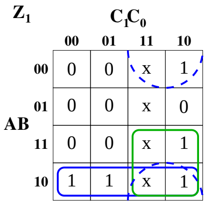
A	B	C_1	C_0	Z_1	Z_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	1	0
other				x	x

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.

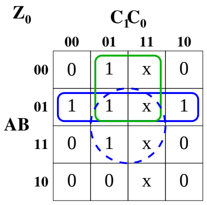
It is tempting to shorten the full truth table by replacing groups of rows. For example, if $AB = 01$, we know that $A < B$, so the less significant bits (for which the result is represented by the C_1C_0 inputs) don't matter. We could write one row with input pattern $ABC_1C_0 = 01\text{xx}$ and output 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:

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



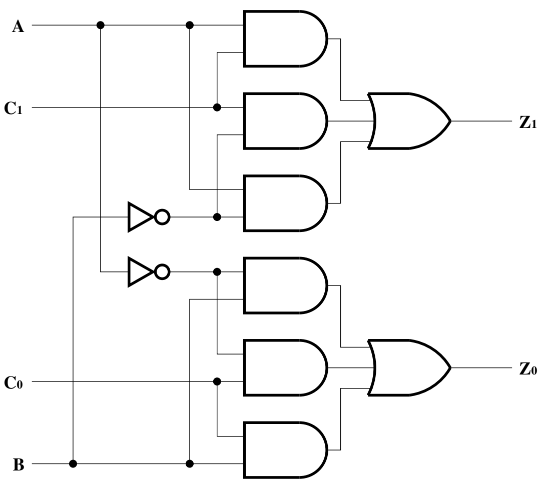
Z_1 AB \ C_1C_0	00	01	11	10
00	0	0	x	1
01	0	0	x	0
11	0	0	x	1
10	1	1	x	1



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

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.

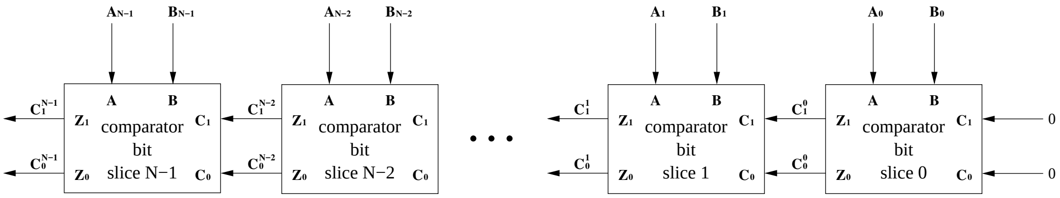
a comparator bit slice (first attempt)



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



2.4.4 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!

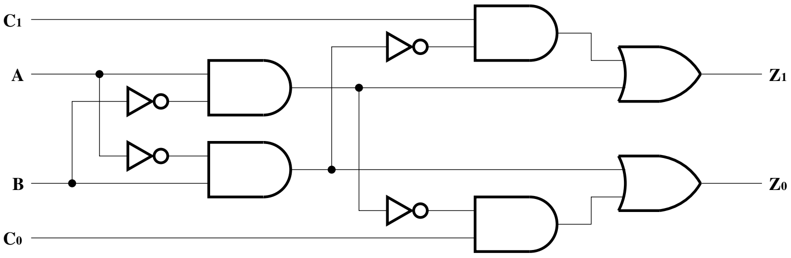
The first step is to manipulate our algebra to expose common terms that occur due to the design's symmetry. Starting with our original equation for Z_1, we have

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

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.

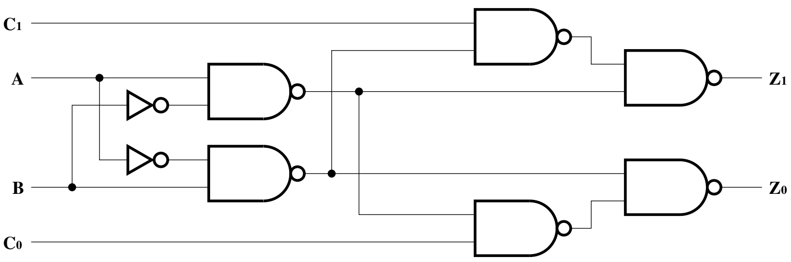
An implementation based on our new algebraic formulation appears to the right. In this form, we seem to have kept the same number of gates, although we have replaced the 3-input gates with inverters. However, the middle inverters disappear when we convert to NAND/NOR form, as shown below to the right.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



2.4.5 Extending to 2's Complement

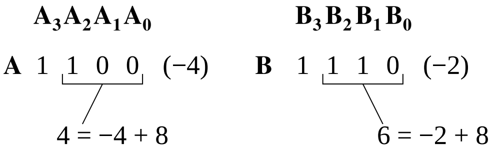
What about comparing 2's complement numbers? Can we make use of the unsigned comparator that we just designed?

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.



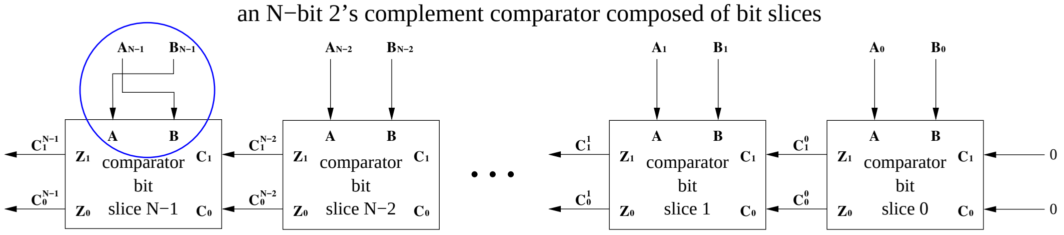
Let's define $A_r = A + 2^{N-1}$ as the value of the remaining bits for A and B_r similarly for B. What happens if we just go ahead and compare A_r and B_r using an $(N-1)$-bit unsigned comparator? If we find that $A_r < B_r$ we know that $A_r - 2^{N-1} < B_r - 2^{N-1}$ as well, but that means $A < B$!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



2.4.6 Further Optimization

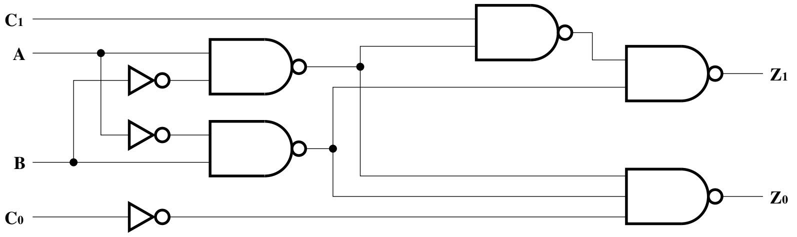
Let's return to the topic of optimization. To what extent did the representation of the three outcomes affect our ability to develop a good bit slice design? Although selecting a good representation can be quite important, for this particular problem most representations lead to similar implementations.

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.

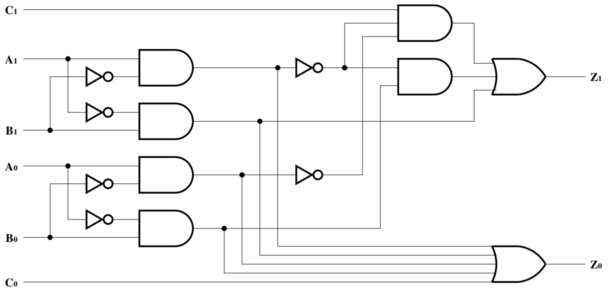
a comparator bit slice (alternate representation)



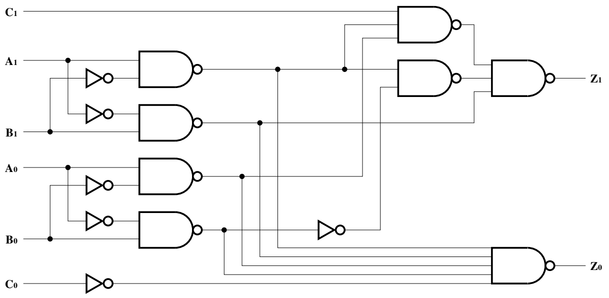
Why didn't it work? Should we consider still other representations? In fact, none of the possible representations that we might choose for a bit slice can cut the delay down to one gate delay per bit. The problem is fundamental, and is related to the nature of CMOS. For a single bit slice, we define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

ECE120: Introduction to Computer Engineering

Notes Set 2.5 Using Abstraction to Simplify Problems

In this set of notes, we illustrate the use of abstraction to simplify problems, then introduce a component called a multiplexer that allows selection among multiple inputs. We begin by showing how two specific examples—integer subtraction and identification of 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.

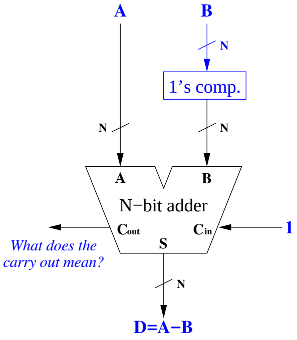
Together with the idea of bit-sliced designs that we introduced earlier, these techniques help to simplify the process of designing logic that operates correctly. The techniques can, of course, lead to less efficient designs, but *correctness is always more important than performance.* The potential 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.

2.5.1 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.

2.5.2 Checking ASCII for Upper-case 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 $U(C)$. The function U should equal 1 whenever C represents an upper-case letter, and should equal 0 whenever C does not.

In ASCII, the 7-bit patterns from 0x41 through 0x5A correspond to the letters A through Z in alphabetic 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_1C_0 \ C_3C_2	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

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

T_5

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

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

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 U? 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:

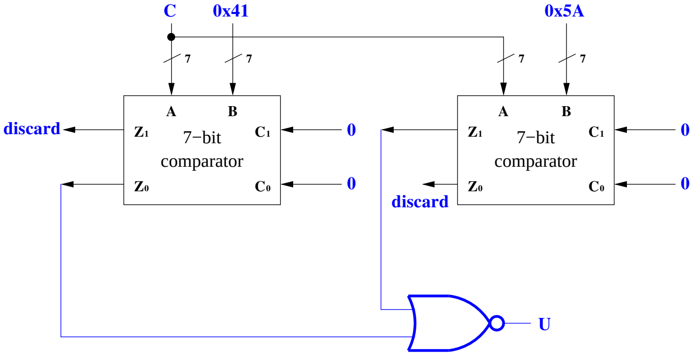
$$\begin{aligned} U &= C_6\,\overline{C_5}\,\overline{C_4}\,T_4 + C_6\,\overline{C_5}\,C_4\,T_5 \\ &= C_6\,\overline{C_5}\,\overline{C_4}\,(C_3 + C_2 + C_1 + C_0) + C_6\,\overline{C_5}\,C_4\,(\overline{C_3} + \overline{C_2}\,\overline{C_1} + \overline{C_2}\,\overline{C_0}) \end{aligned}$$

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.

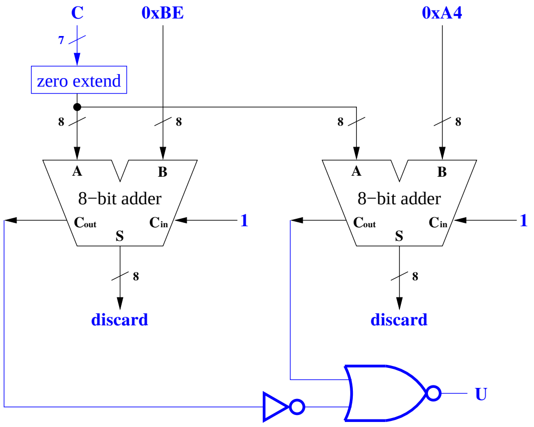
Let's think about other ways to tackle the problem of calculating U. In Notes Sets 2.3 and 2.4, we developed adders and comparators. Can we make use of these components as building blocks to check whether C represents an upper-case letter? Yes, of course we can: by comparing C with the ends of the range 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 U. 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 to and result of 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 U. 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 $U = 0$, 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 $U = 0$, since we do not have a letter in this case. Only when 0x41 $\leq C \leq$ 0x5A does $U = 1$, 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 U? 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 $U = 0$. 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 $U = 0$. With the right combination of carries (1 from the left and 0 from the right), we obtain $U = 1$.



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?

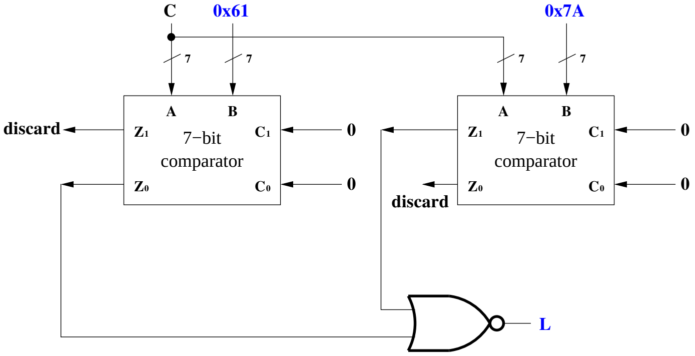
We offer two answers. First, given the design shown above, a good CAD tool recognizes that the sum outputs of the adders are not being used, and does not generate logic to calculate them. The logic for the two carry bits used to calculate U can then be optimized. Second, the design shown, including the 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.

2.5.3 Checking ASCII for Lower-case Letters

Having developed several approaches for checking for an upper-case letter, the task of checking for a lower-case letter should be straightforward. In ASCII, lower-case letters are represented by the 7-bit patterns from 0x61 through 0x7A. One can now easily see how more abstract designs make solving similar tasks easier. If we have designed our upper-case checker with 7-variable K-maps, we must start again with new K-maps for the lower-case checker. If instead we have taken the approach of designing logic for the upper and lower bits of the ASCII character, we can reuse most of that logic, since the functions T_4 and T_5 are identical when checking for a lower-case character. Recalling the algebraic form of $U(C)$, we can then write a function $L(C)$ (a lower-case checker) as shown on the left below.

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



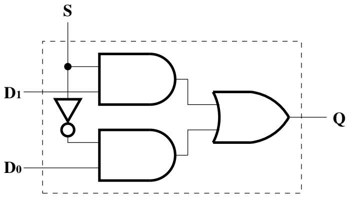
Finally, if we have used a design based on comparators or adders, the design of a lower-case checker becomes trivial: simply change the numbers that we input to these components, as shown in the figure on the right above for the comparator-based design. The only changes from the upper-case checker design are the inputs to the comparators and the output produced, highlighted with blue text in the figure.

2.5.4 The Multiplexer

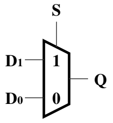
Using the more abstract designs for checking ranges of ASCII characters, we can go a step further and create a checker for both upper- and lower-case letters. To do so, we add another input S that allows us to select the function that we want—either the upper-case checker $U(C)$ or the lower-case checker $L(C)$.

For this purpose, we make use of a logic block called a **multiplexer**, or **mux**. Multiplexers are an important abstraction for digital logic. In general, a multiplexer allows us to use one digital signal to select which of several others is forwarded to an output.

The simplest form of the multiplexer is the 2-to-1 multiplexer shown to the right. The logic diagram illustrates how the mux works. The block has two inputs from the left and one from the top. The top input allows us to choose which of the left inputs is forwarded to the output. When the input $S = 0$, the upper AND gate outputs 0, and the lower AND gate outputs the value of D_0. The OR gate then produces $Q = 0 + D_0 = D_0$. Similarly, when input $S = 1$, the upper AND gate outputs D_1, and the lower AND gate outputs 0. In this case, the OR gate produces $Q = D_1 + 0 = D_1$.



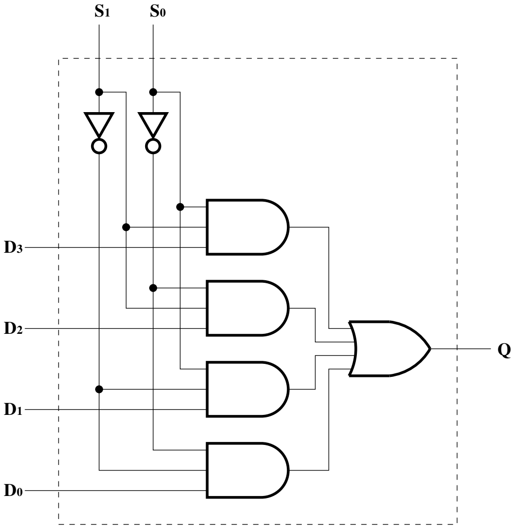
a 2−to−1 multiplexer
(logic diagram)



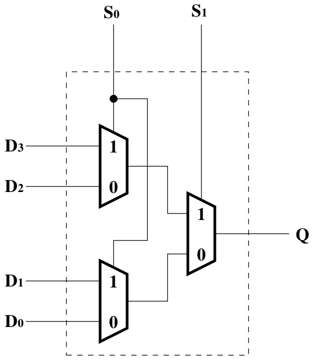
a 2−to−1 multiplexer
(symbolic form)

The symbolic form of the mux is a trapezoid with data inputs on the larger side, an output on the smaller side, and a select input on the angled part of the trapezoid. The labels inside the trapezoid indicate the value of the select input S for which the adjacent data signal, D_1 or D_0, is copied to the output Q.

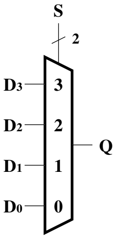
We can generalize multiplexers in two ways. First, we can extend the single select input to a group of select inputs. An N-bit select input allows selection from amongst 2^N inputs. A 4-to-1 multiplexer is shown below, for example. The logic diagram on the left shows how the 4-to-1 mux operates. For any combination of S_1S_0, three of the AND gates produce 0, and the fourth outputs the D input corresponding to the interpretation of S as an unsigned number. Given three zeroes and one D input, the OR gate thus reproduces one of the D's. When $S_1S_0 = 10$, for example, the third AND gate copies D_2, and $Q = D_2$.



a 4−to−1 multiplexer
(logic diagram)



a 4−to−1 multiplexer
built from 2−to−1
multiplexers



a 4−to−1 multiplexer
(symbolic form)

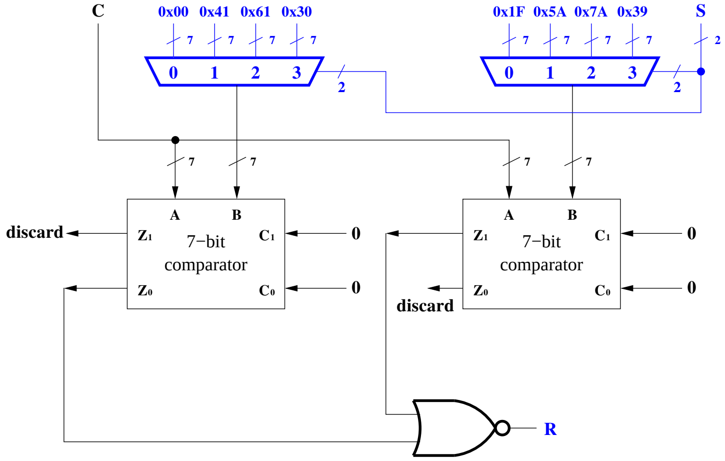
As shown in the middle figure, a 4-to-1 mux can also be built from three 2-to-1 muxes. Finally, the symbolic form of a 4-to-1 mux appears on the right in the figure.

The second way in which we can generalize multiplexers is by using several multiplexers of the same type and using the same signals for selection. For example, we might use a single select bit T to choose between any number of paired inputs. Denote input pair by i D_1^i and D_0^i. For each pair, we have an output Q_i. When $T = 0$, $Q_i = D_0^i$ for each value of i. And, when $T = 1$, $Q_i = D_1^i$ for each value of i. Each value of i requires a 2-to-1 mux with its select input driven by the global select signal T.

Returning to the example of the upper- and lower-case checker, we can make use of two groups of seven 2-to-1 muxes, all controlled by a single bit select signal S, to choose between the inputs needed for an upper-case checker and those needed for a lower-case checker.

Specific configurations of multiplexers are often referred to as N**-to-**M **multiplexers**. Here the value N refers to the number of inputs, and M refers to the number of outputs. The number of select bits can then be calculated as $\log_2(N/M)$—N/M is generally a power of two—and one way to build such a multiplexer is to use M copies of an (N/M)-to-1 multiplexer.

Let's extend our upper- and lower-case checker to check for four different ranges of ASCII characters, as shown below. This design uses two 28-to-7 muxes to create a single checker for the four ranges. Each of the muxes in the figure logically represents seven 4-to-1 muxes.



The table to the right describes the behavior of the checker. When the select input S is set to 00, the left mux selects the value 0x00, and the right mux selects the value 0x1F, which checks whether the ASCII character represented by C is a control character. When the select input $S = 01$, the muxes produce the values needed to check whether C is an upper-case letter. Similarly, when the select input $S = 10$, the muxes produce the values needed to check whether C is a lower-case letter. Finally, when the select input $S = 11$, the left mux selects the value 0x30, and the right mux selects the value 0x39, which checks whether the ASCII character represented by C is a digit (0 to 9).

S_1S_0	left comparator input	right comparator input	$R(C)$ produced
00	0x00	0x1F	control character?
01	0x41	0x5A	upper-case letter?
10	0x61	0x7A	lower-case letter?
11	0x30	0x39	numeric digit?

ECE120: Introduction to Computer Engineering

Notes Set 2.6 Sequential Logic

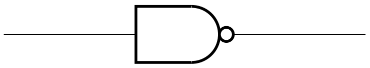
These notes introduce logic components for storing bits, building up from the idea of a pair of cross-coupled inverters through an implementation of a flip-flop, the storage abstractions used in most modern logic design processes. We then introduce simple forms of timing diagrams, which we use to illustrate 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.*

2.6.1 Storing One Bit

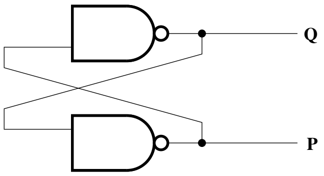
So far we have discussed only implementation of Boolean functions: given some bits as input, how can we design a logic circuit to calculate the result of applying some function to those bits? The answer to such questions is called **combinational logic** (sometimes **combinatorial logic**), a name stemming 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.



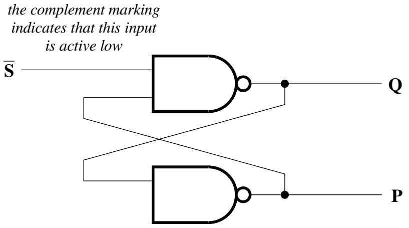
Q	P
0	1
1	0

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.[5] 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?

[5] 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

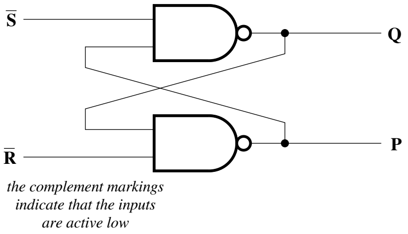
$\bar{S}$	Q	P
1	0	1
1	1	0
0	1	0

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)

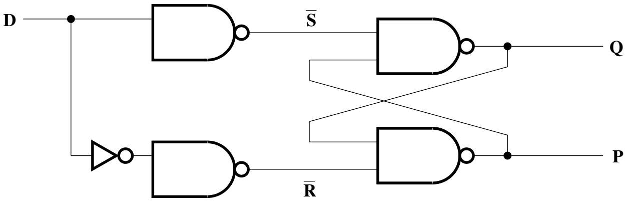

S̄
R̄
Q
P
the complement markings indicate that the inputs are active low

$\bar{R}$	$\bar{S}$	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.[6]

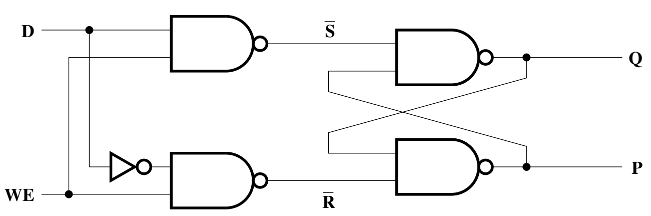
We can avoid the problem by adding gates to prevent the two control inputs ($\bar{S}$ and $\bar{R}$) from ever being 1 at the same time. A single inverter might technically suffice, but let's build up the structure shown below, noting that the two inverters in sequence connecting D to $\bar{R}$ have 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.

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

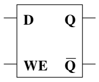


a gated D latch (stores a single bit)

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)

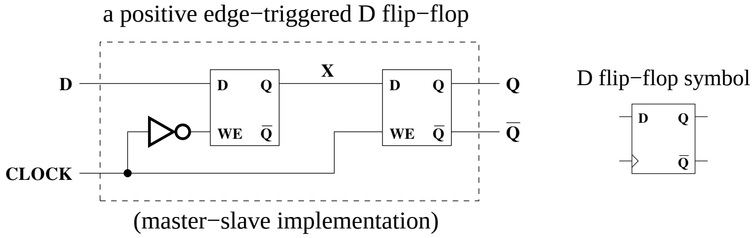
The circuit just shown is called a **gated D latch**, and is an important mechanism for storing state in sequential logic. (Random-access memory uses a slightly different technique to connect the cross-coupled inverters, but latches are used for nearly every other application of stored state.) The "D" 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.

2.6.2 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.

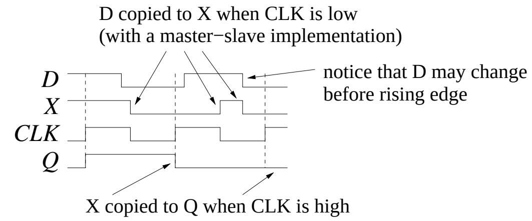
The timing of signals in such designs plays a critical role in their correct operation. Fortunately, we have developed powerful abstractions that allow engineers to ignore much of the complexity while thinking about the Boolean logic needed for a given design.

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).



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").

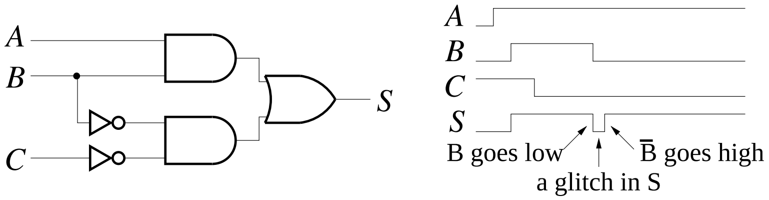
The value of using flip-flops and assuming a square-wave clock signal with uniform timing may not be clear to you yet, but it bears emphasis. With such assumptions, *we can treat time as having discrete values.* In other words, time "ticks" along discretely, like integers instead of real numbers. We 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).

2.6.3 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 $S = AB + \bar{B}\bar{C}$.

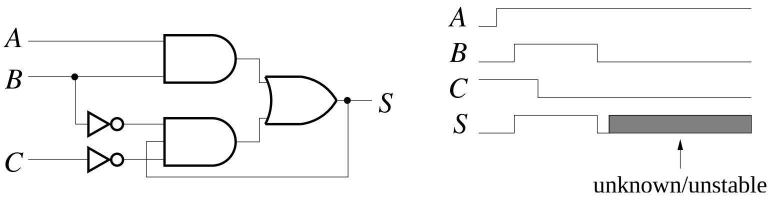


The timing diagram on the right shows a **glitch** in the output when the input shifts from $ABC = 110$ to 100, that is, when B falls. The problem lies in the possibility that the upper AND gate, driven by B, might go low before the lower AND gate, driven by $\bar{B}$, goes high. In such a case, the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

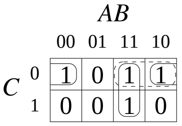
A circuit that might exhibit a glitch in an output that functionally remains stable at 1 is said to have a **static-1 hazard**. The qualifier "static" here refers to the fact that we expect the output to remain static, while the "1" refers to the expected value of the output.

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.



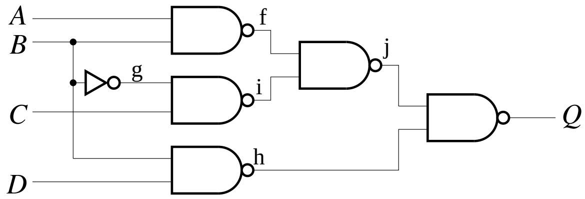
Eliminating static hazards requires merely extending the circuit with consensus terms in order to ensure that some AND gate remains high through every transition between input states with output 1.[7] In the K-map shown, the dashed line indicates the necessary consensus term, $A\bar{C}$.

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

2.6.4 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

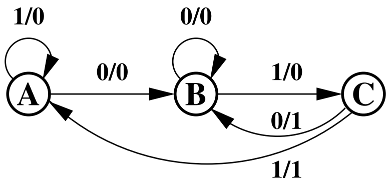
The output column in the table illustrates the possible impact of a dynamic hazard: rather than a smooth transition from 1 to 0, the output drops to 0, rises back to 1, and finally falls to 0 again. The dynamic hazard in this case can be attributed to the presence of a static hazard in the logic that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

2.6.5 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.

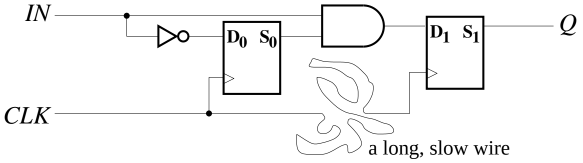
	IN	
S_1S_0	0	1
00	01/0	00/0
01	01/0	10/0
11	x	x
10	01/1	00/1

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.



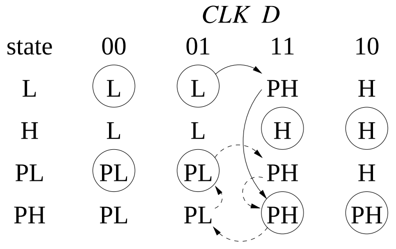
Due to the long delays, we cannot assume that rising clock edges arrive at the flip-flops at the same time. The result is called clock skew, and can make the circuit behave improperly by exposing essential hazards. In the logical B to C transition, for example, we begin in state $S_1S_0 = 01$ with $IN = 1$ 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.

2.6.6 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.

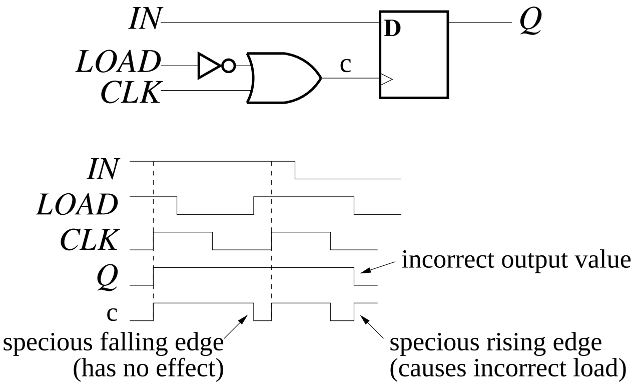
The detection of essential hazards in a clocked synchronous design based on flip-flops thus reduces to examination of the state machine. If the next state of the machine has any dependence on the current state, an essential hazard exists, as a second rising clock edge moves the system into a second new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

ECE120: Introduction to Computer Engineering

Notes Set 2.7 Registers

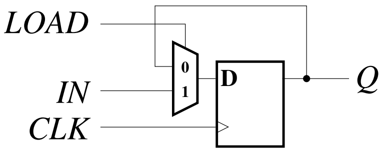
This set of notes introduces registers, an abstraction used for storage of groups of bits in digital systems. We introduce some terminology used to describe aspects of register design and illustrate the idea of a shift register. The registers shown here are important abstractions for digital system design.

2.7.1 Registers

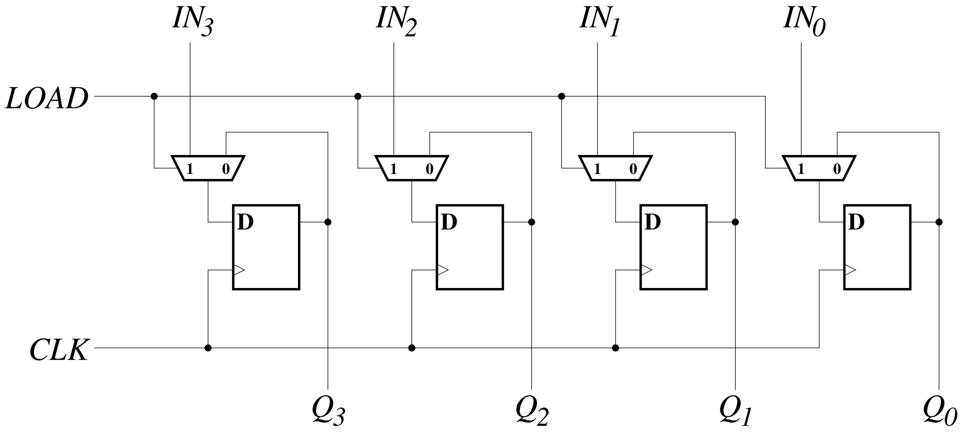


A **register** is a storage element composed from one or more flip-flops operating on a common clock. In addition to the flip-flops, most registers include logic to control the bits stored by the register. For example, D flip-flops copy their inputs at the rising edge of each clock cycle, discarding whatever bits they have stored before the rising edge (in the previous clock 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.

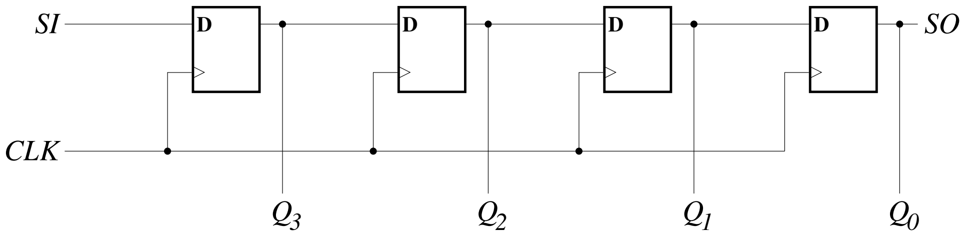


A better approach is to use a mux and a feedback loop from the flip-flop's output, as shown in the figure to the right. When *LOAD* is low, the mux selects the feedback line, and the register reloads its current value. When *LOAD* is high, the mux selects the *IN* input, and the register loads a new value. 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 muxes forward 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**.

2.7.2 Shift Registers

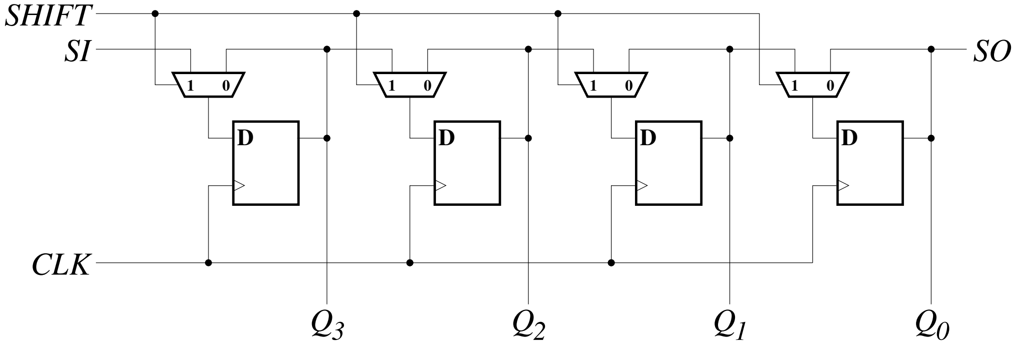


Certain types of registers include logic to manipulate data held within the register. A **shift register** is an important example of this type. The simplest shift register is a series of D flip-flops, with the output of each attached to the input of the next, as shown to the right above. In the circuit 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.

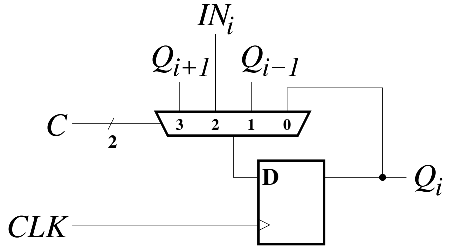
Returning to the figure above, imagine that the outputs Q_i feed into logic clocked at $1/4^{th}$ the rate of the shift register (and suitably synchronized). Every four cycles, the flip-flops fill up with another four bits, at which point the outputs are read in parallel. The shift register shown can 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 above also shifts continuously, and cannot store a value. A set of muxes, analogous to those that we used to control register loading, can be applied to control shifting, as shown below.

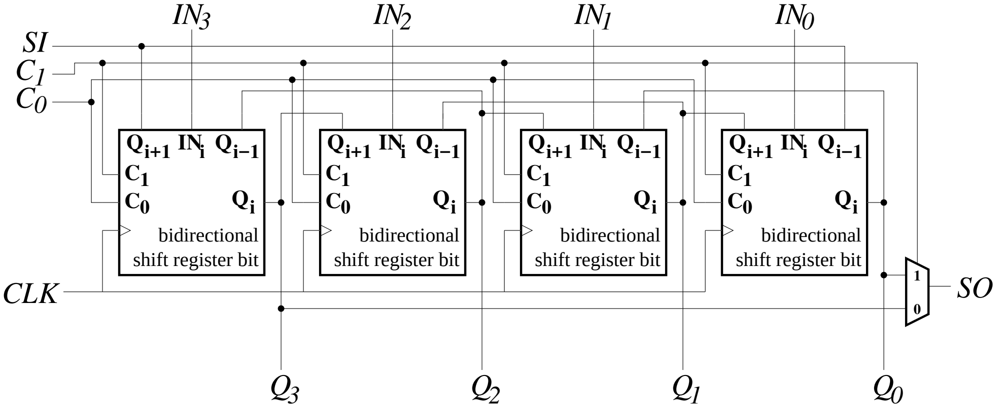


Using a 4-to-1 mux, we can construct a shift register with additional functionality. The bit slice at the top of the next page allows us to build a **bidirectional shift register** with parallel load capability and the ability to retain its value indefinitely. The two-bit control input C uses a representation that we have chosen for the four operations supported by our shift register, as shown in the table below the bit slice design.

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 figure below represents a 4-bit bidirectional shift register constructed in this way. The mux used for the SO output logic is the glue logic needed in addition to the four bit slices. 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.

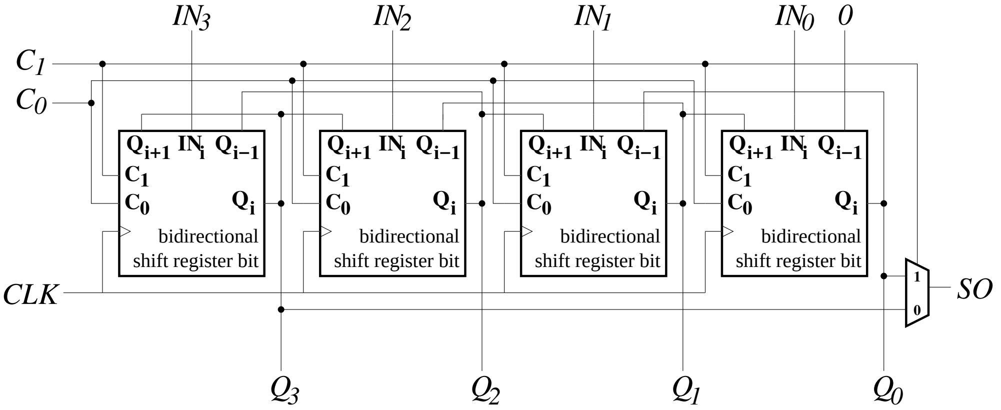


C_1C_0	meaning
00	retain current value
01	shift left (low to high)
10	load new value (from IN)
11	shift right (high to low)



Several specialized shift operations are used to support data manipulation in modern processors (CPUs). Essentially, these specializations dictate the glue logic for a shift register as well as the serial input value. The simplest is a **logical shift**, for which SI is hardwired to 0: incoming bits are always 0. A **cyclic shift** takes SO and feeds it back into SI, 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. The bit shifted in for left shifts has been hardwired to 0.



ECE120: Introduction to Computer Engineering

Notes Set 2.8 Summary of Part 2 of the Course

These notes supplement the Patt and Patel textbook, so you will also need to read and understand the relevant chapters (see the syllabus) in order to master this material completely.

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 skills, and leave the easy stuff for the next page. We expect you to be able to exercise the following skills:

- Design a CMOS gate for a simple Boolean function 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, muxes, 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 to know 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.

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.

- Boolean functions and logic gates
    - NOT/inverter
    - AND
    - OR
    - XOR
    - NAND
    - NOR
    - XNOR
    - majority function
- specific logic circuits
    - full adder
    - ripple carry adder
    - N-to-M multiplexer (mux)
    - N-to-2N decoder
    - $\bar{\text{R}}$-$\bar{\text{S}}$ 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 input
        - 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 circuit
    - parallel/serial load of register
    - logical/arithmetic/cyclic shift

ECE120: Introduction to Computer Engineering

Notes Set 3.1 Serialization and Finite State Machines

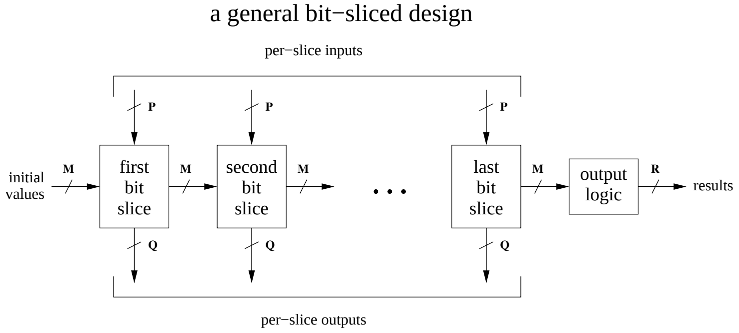
The third part of our class builds upon the basic combinational and sequential logic elements that we developed in the second part. After discussing a simple application of stored state to trade between area and performance, we introduce a powerful abstraction for formalizing and reasoning about digital systems, the Finite State Machine (FSM). General FSM models are broadly applicable in a range of engineering contexts, including not only hardware and software design but also the design of control systems and distributed systems. We limit our model so as to avoid circuit timing issues in your first exposure, but provide some amount of discussion as to how, when, and why you should eventually learn the more sophisticated models. Through development a range of FSM examples, we illustrate important design issues for these systems and motivate a couple of more advanced combinational logic devices that can be used as building blocks. Together with the idea of memory, another form of stored state, these elements form the basis for development of our first computer. At this point we return to the textbook, in which Chapters 4 and 5 provide a solid introduction to the von Neumann model of computing systems and the LC-3 (Little Computer, version 3) instruction set architecture. By the end of this part of the course, you will have seen an example of the boundary between hardware and software, and will be ready to write some instructions yourself.

In this set of notes, we cover the first few parts of this material. We begin by describing the conversion of bit-sliced designs into serial designs, which store a single bit slice's output in flip-flops and then feed the outputs back into the bit slice in the next cycle. As a specific example, we use our bit-sliced comparator to discuss tradeoffs in area and performance. We introduce Finite State Machines and some of the tools used to design them, then develop a handful of simple counter designs. Before delving too deeply into FSM design issues, we spend a little time discussing other strategies for counter design and placing the material covered in our course in the broader context of digital system design. Remember that *sections marked with an asterisk are provided solely for your interest,* but you may need to learn this material in later classes.

3.1.1 Serialization: General Strategy

In previous notes, we discussed and illustrated the development of bit-sliced logic, in which one designs a logic block to handle one bit of a multi-bit operation, then replicates the bit slice logic to construct a design for the entire operation. We developed ripple carry adders in this way in Notes Set 2.3 and both unsigned and 2's complement comparators in Notes Set 2.4.

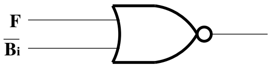
Another interesting design strategy is **serialization**: rather than replicating the bit slice, we can use flip-flops to store the bits passed from one bit slice to the next, then present the stored bits *to the same bit slice* in the next cycle. Thus, in a serial design, we only need one copy of the bit slice logic! The area needed for a serial design is usually much less than for a bit-sliced design, but such a design is also usually slower. After illustrating the general design strategy, we'll consider these tradeoffs more carefully in the context of a detailed example.



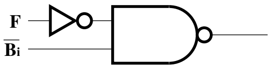
Recall the general bit-sliced design approach, as illustrated to the right. Some number of copies of the logic for a single bit slice are connected in sequence. Each bit slice accepts P bits of operand input and produces Q bits of external output. Adjacent bit slices receive an additional M bits of information from the previous bit slice and pass along M bits to the next bit slice, generally using some representation chosen by the designer. The first bit slice is initialized by passing in constant values, and some calculation may be performed on the final bit slice's results to produce R bits more external output.

We can transform this bit-sliced design to a serial design with a single copy of the bit slice logic, $M + Q$ flip-flops, and M gates (and sometimes an inverter). The strategy is illustrated on the right below. A single copy of the bit slice operates on one set of P external input bits and produces one set of Q external output bits each clock cycle. In the design shown, these output bits are available during the next cycle, after they have been stored in the flip-flops. The M bits to be passed to the "next" bit slice are also stored in flip-flops, and in the next cycle are provided back to the same physical bit slice as inputs. The first cycle of a multi-cycle operation must be handled slightly differently, so we add selection logic and an control signal, F. For the first cycle, we apply $F = 1$, and the initial values are passed into the bit slice. For all other bits, we apply $F = 0$, and the values stored in the flip-flops are returned to the bit slice's inputs. After all bits have passed through the bit slice—after N cycles for an N-bit design—the final M bits are stored in the flip-flops, and the results are calculated by the output logic.

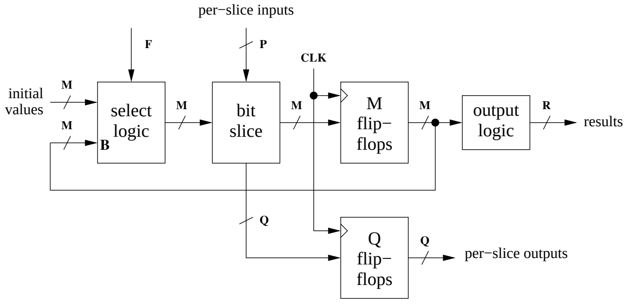
initialize to 0 when F=1



initialize to 1 when F=1



a serialized bit−sliced design

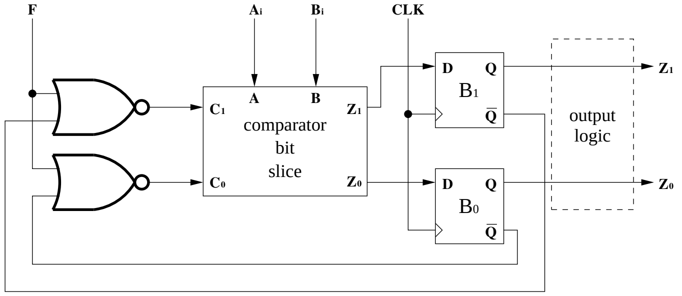


The selection logic merits explanation. Given that the original design initialized the bits to constant values (0s or 1s), we need only simple logic for selection. The two drawings on the left above illustrate how $\overline{B_i}$, the complemented flip-flop output for a bit i, can be combined with the first-cycle signal F to produce an appropriate input for the bit slice. Selection thus requires one extra gate for each of the M inputs, and we need an inverter for F if any of the initial values is 1.

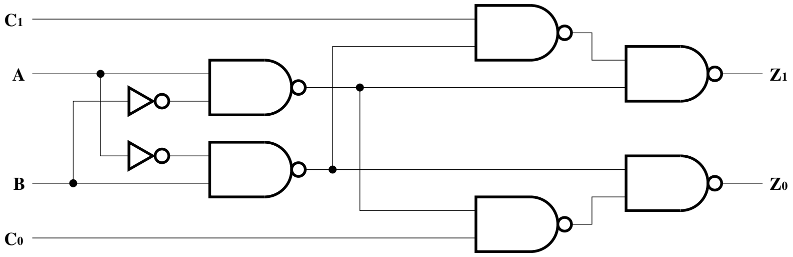
3.1.2 Serialization: Comparator Example

We now apply the general strategy to a specific example, our bit-sliced unsigned comparator from Notes Set 2.4. The result is shown to the right. In terms of the general model, the single comparator bit slice accepts $P = 2$ bits of inputs each cycle, in this case a single bit from each of the two numbers being compared, presented to the bit slice in increasing order of significance. The bit slice produces no external output other than the final result ($Q = 0$). Two bits ($M = 2$) are produced each cycle by the bit slice and stored into flip flops B_1 and B_0. These bits represent the relationship between the two numbers compared so far (including only the bit already seen by the comparator bit slice). On the first cycle, when the least significant bits of A and B are being fed into the bit slice, we set $F = 1$, which forces the C_1 and C_0 inputs of the bit slice to 0 independent of the values stored in the flip-flops. In all other cycles, $F = 0$, and the NOR gates set $C_1 = B_1$ and $C_0 = B_0$. Finally, after N cycles for an N-bit comparison, the output logic—in this case simply wires, as shown in the dashed box—places the result of the comparison on the Z_1 and Z_0 outputs ($R = 2$ in the general model). The result is encoded in the representation defined for constructing the bit slice (see Notes Set 2.4, but the encoding does not matter here).

a serial unsigned comparator



a comparator bit slice (optimized, NAND/NOR)



How does the serial design compare with the bit-sliced design? As an estimate of area, let's count gates. Our optimized design is replicated to the right for convenience. Each bit slice requires six 2-input gates and two inverters. Assume that each flip-flop requires eight 2-input gates and two inverters, so the serial design overall requires 24 gates and six inverters to handle any number of bits. Thus, for any number of bits $N \geq 4$, the serial design is smaller than the bit-sliced design, and the benefit grows with N.

What about performance? In Notes Set 2.4, we counted gate delays for our bit-sliced design. The path from A or B to the outputs is four gate delays, but the C to Z paths are all two gate delays. Overall, then, the bit-sliced design requires $2N + 2$ gate delays for N bits. What about the serial design?

The performance of the serial design is likely to be much worse for three reasons. First, all paths in the design matter, not just the paths from bit slice to bit slice. None of the inputs can be assumed to be available before the start of the clock cycle, so we must consider all paths from input to output. Second, we must also count gate delays for the selection logic as well as the gates embedded in the flip-flops. Finally, the result of these calculations may not matter, since the clock speed may well be limited by other logic elsewhere in the system. If we want a common clock for all of our logic, the clock must not go faster than the slowest element in the entire system, or some of our logic will not work properly.

What is the longest path through our serial comparator? Let's assume that the path through a flip-flop is eight gate delays, with four on each side of the clock's rising edge. The inputs A and B are likely to be driven by flip-flops elsewhere in our system, so we conservatively count four gate delays to A and B and five gate delays to C_1 and C_0 (the extra one comes from the selection logic). The A and B paths thus dominate inside the bit slice, adding four more gate delays to the outputs Z_1 and Z_0. Finally, we add the last four gate delays to flip the first latch in the flip-flops for a total of 12 gate delays. If we assume that our serial comparator limits the clock frequency (that is, if everything else in the system can use a faster clock), we take 12 gate delays per cycle, or $12N$ gate delays to compare two N-bit numbers.

You might also notice that adding support for 2's complement is no longer free. We need extra logic to swap the A and B inputs in the cycle corresponding to the sign bits of A and B. In other cycles, they must remain in the usual order. This extra logic is not complex, but adds further delay to the paths.

The bit-sliced and serial designs represent two extreme points in a broad space of design possibilities. Optimization of the entire N-bit logic function (for any metric) represents a third extreme. As an engineer, you should realize that you can design systems anywhere in between these points as well. At the end of Notes Set 2.4, for example, we showed a design for a logic slice that compares two bits at a time. In general, we can optimize logic for any number of bits and then apply multiple copies of the resulting logic in space (a generalization of the bit-sliced approach), or in time (a generalization of the serialization approach), or in a combination of the two. Sometimes these tradeoffs may happen at a higher level. As mentioned in Notes Set 2.3, computer software uses the carry out of an adder to perform addition of larger groups of bits (over multiple clock cycles) than is supported by the processor's adder hardware. In computer system design, engineers often design hardware elements that are general enough to support this kind of extension in software.

As a concrete example of the possible tradeoffs, consider a serial comparator design based on the 2-bit slice variant. This approach leads to a serial design with 24 gates and 10 inverters, which is not much larger than our earlier serial design. In terms of gate delays, however, the new design is identical, meaning that we finish a comparison in half the time. More realistic area and timing metrics show slightly more difference between the two designs. These differences can dominate the results if we blindly scale the idea to handle more bits without thinking carefully about the design. Neither many-input gates nor gates driving many outputs work well in practice.

3.1.3 Finite State Machines

A **finite state machine** (or **FSM**) is a model for understanding the behavior of a system by describing the system as occupying one of a finite set of states, moving between these states in response to external inputs, and producing external outputs. In any given state, a particular input may cause the FSM to move to another state; this combination is called a **transition rule**. An FSM comprises five parts: a finite set of states, a set of possible inputs, a set of possible outputs, a set of transition rules, and methods for calculating outputs.

When an FSM is implemented as a digital system, all states must be represented as patterns using a fixed number of bits, all inputs must be translated into bits, and all outputs must be translated into bits. For a digital FSM, transition rules must be **complete**; in other words, given any state of the FSM, and any pattern of input bits, a transition must be defined from that state to another state (transitions from a state to itself, called **self-loops**, are acceptable). And, of course, calculation of outputs for a digital FSM reduces to Boolean logic expressions. In this class, we focus on clocked synchronous FSM implementations, in which the FSM's internal state bits are stored in flip-flops.

In this section, we introduce the tools used to describe, develop, and analyze implementations of FSMs with digital logic. In the next few weeks, we will show you how an FSM can serve as the central control logic in a computer. At the same time, we will illustrate connections between FSMs and software and will make some connections with other areas of interest in ECE, such as the design and analysis of digital control systems.

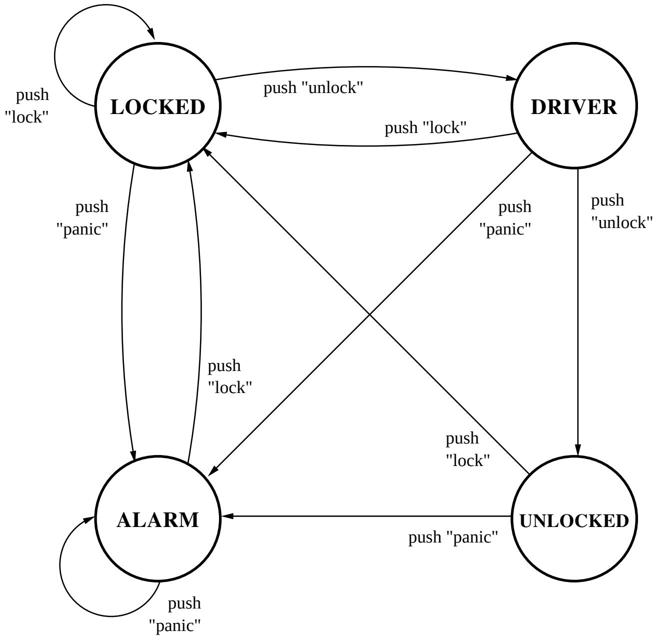
The table below gives a **list of abstract states** for a typical keyless entry system for a car. In this case, we have merely named the states rather than specifying the bit patterns to be used for each state—for this reason, we refer to them as abstract states. The description of the states in the first column is an optional element often included in the early design stages for an FSM, when identifying the states needed for the design. A list may also include the outputs for each state. Again, in the list below, we have specified these outputs abstractly. By including outputs for each state, we implicitly assume that outputs depend only on the state of the FSM. We discuss this assumption in more detail later in these notes (see "Machine Models"), but will make the assumption throughout our class.

meaning	state	driver's door	other doors	alarm on
vehicle locked	LOCKED	locked	locked	no
driver door unlocked	DRIVER	unlocked	locked	no
all doors unlocked	UNLOCKED	unlocked	unlocked	no
alarm sounding	ALARM	locked	locked	yes

Another tool used with FSMs is the **next-state table** (sometimes called a **state transition table**, or just a **state table**), which maps the current state and input combination into the next state of the FSM. The abstract variant shown below outlines desired behavior at a high level, and is often ambiguous, incomplete, and even inconsistent. For example, what happens if a user pushes two buttons? What happens if they push unlock while the alarm is sounding? These questions should eventually be considered. However, we can already start to see the intended use of the design: starting from a locked car, a user can push "unlock" once to gain entry to the driver's seat, or push "unlock" twice to open the car fully for passengers. To lock the car, a user can push the "lock" button at any time. And, if a user needs help, pressing the "panic" button sets off an alarm.

state	action/input	next state
LOCKED	push "unlock"	DRIVER
DRIVER	push "unlock"	UNLOCKED
(any)	push "lock"	LOCKED
(any)	push "panic"	ALARM

A **state transition diagram** (or **transition diagram**, or **state diagram**), as shown to the right, illustrates the contents of the next-state table graphically, with each state drawn in a circle, and arcs between states labeled with the input combinations that cause these transitions from one state to another.



Putting the FSM design into this graphical form does not solve the problems with the abstract model. The questions that we asked in regard to the next-state table remain unanswered.

Implementing an FSM using digital logic requires that we translate the design into bits, eliminate any ambiguity, and complete the specification. How many internal bits should we use? What are the possible input values, and how are their meanings represented in bits? What are the possible output values, and how are their meanings represented in bits? We will consider these questions for several examples in the coming weeks.

For now, we simply define answers for our example design, the keyless entry system. Given four states, we need at least $\lceil \log_2(4) \rceil = 2$ bits of internal state, which we store in two flip-flops and call S_1S_0. The table below lists input and output signals and defines their meaning.

outputs	D	driver door; 1 means unlocked
	R	other doors (remaining doors); 1 means unlocked
	A	alarm; 1 means alarm is sounding
inputs	U	unlock button; 1 means it has been pressed
	L	lock button; 1 means it has been pressed
	P	panic button; 1 means it has been pressed

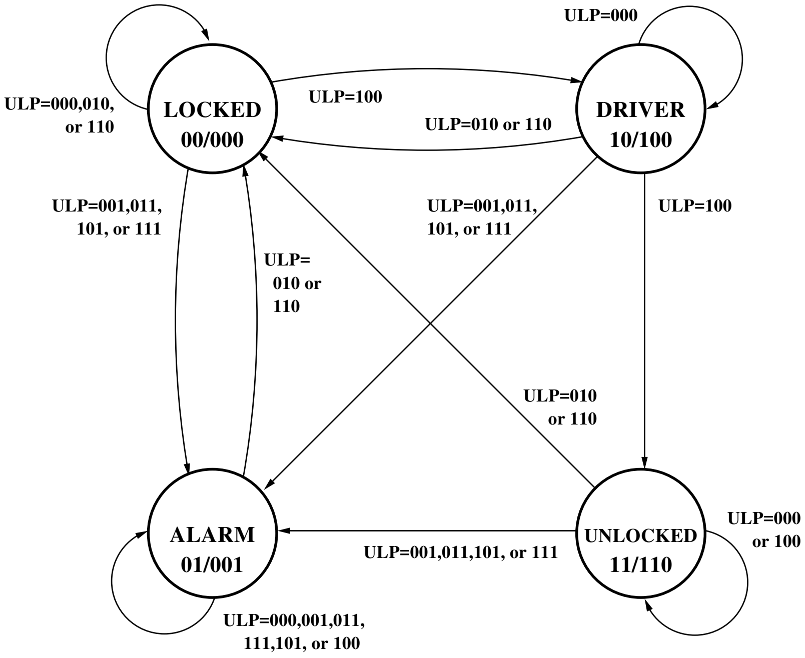
We can now choose a representation for our states and rewrite the list of states, using bits both for the states and for the outputs. We also include the meaning of each state for clarity in our example. Note that we can choose the internal representation in any way. Here we have matched the D and R outputs when possible to simplify the output logic needed for the implementation. The order of states in the list is not particularly important, but should be chosen for convenience and clarity (including transcribing bits into to K-maps, for example).

			driver's door	other doors	alarm on
meaning	state	S_1S_0	D	R	A
vehicle locked	LOCKED	00	0	0	0
driver door unlocked	DRIVER	10	1	0	0
all doors unlocked	UNLOCKED	11	1	1	0
alarm sounding	ALARM	01	0	0	1

We can also rewrite the next-state table in terms of bits. We use Gray code order on both axes, as these orders make it more convenient to use K-maps. The values represented in this table are the next FSM state given the current state S_1S_0 and the inputs U, L, and P. Our symbols for the next-state bits are S_1^+ and S_0^+. The "+" superscript is a common way of expressing the next value in a discrete series, here induced by the use of clocked synchronous logic in implementing the FSM. In other words, S_1^+ is the value of S_1 in the next clock cycle, and S_1^+ in an FSM implemented as a digital system is a Boolean expression based on the current state and the inputs. For our example problem, we want to be able to write down expressions for $S_1^+(S_1, S_0, U, L, P)$ and $S_1^+(S_1, S_0, U, L, P)$, as well as expressions for the output logic $U(S_1, S_0)$, $L(S_1, S_0)$, and $P(S_1, S_0)$.

current state	ULP							
S_1S_0	000	001	011	010	110	111	101	100
00	00	01	01	00	00	01	01	10
01	01	01	01	00	00	01	01	01
11	11	01	01	00	00	01	01	11
10	10	01	01	00	00	01	01	11

In the process of writing out the next-state table, we have made decisions for all of the questions that we asked earlier regarding the abstract state table. These decisions are also reflected in the complete state transition diagram shown to the right. The states have been extended with state bits and output bits, as S_1S_0/DRA. You should recognize that we can also leave some questions unanswered by placing x's (don't cares) into our table. However, you should also understand at this point that any implementation will produce bits, not x's, so we must be careful not to allow arbitrary choices unless any of the choices allowed is indeed acceptable for our FSM's purpose. We will discuss this process and the considerations necessary as we cover more FSM design examples.



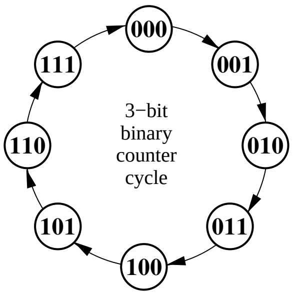
We have deliberately omitted calculation of expressions for the next-state variables S_1^+ and S_0^+, and for the outputs U, L, and P. We expect that you are able to do so from the detailed state table above, and may assign such an exercise as part of your homework.

3.1.4 Synchronous Counters

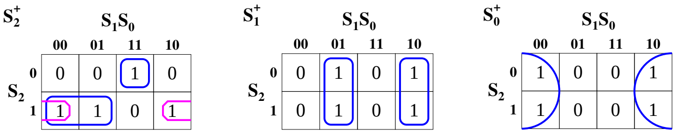
A **counter** is a clocked sequential circuit with a state diagram consisting of a single logical cycle. Not all counters are synchronous. In other words, not all flip-flops in a counter are required to use the same clock signal. A counter in which all flip-flops do utilize the same clock signal is called a **synchronous counter**. Except for a brief introduction to other types of counters in the next section, our class focuses entirely on clocked synchronous designs, including counters.

The design of synchronous counter circuits is a fairly straightforward exercise given the desired cycle of output patterns. The task can be more complex if the internal state bits are allowed to differ from the output bits, so for now we assume that output Z_i is equal to internal bit S_i. Note that distinction between internal states and outputs is necessary if any output pattern appears more than once in the desired cycle.

The cycle of states shown to the right corresponds to the states of a 3-bit binary counter. The numbers in the states represent both internal state bits $S_2S_1S_0$ and output bits $Z_2Z_1Z_0$. We transcribe this diagram into the next-state table shown on the left below. We then write out K-maps for the next state bits S_2^+, S_1^+, and S_0^+, as shown to the right, and use the K-maps to find expressions for these variables in terms of the current state.



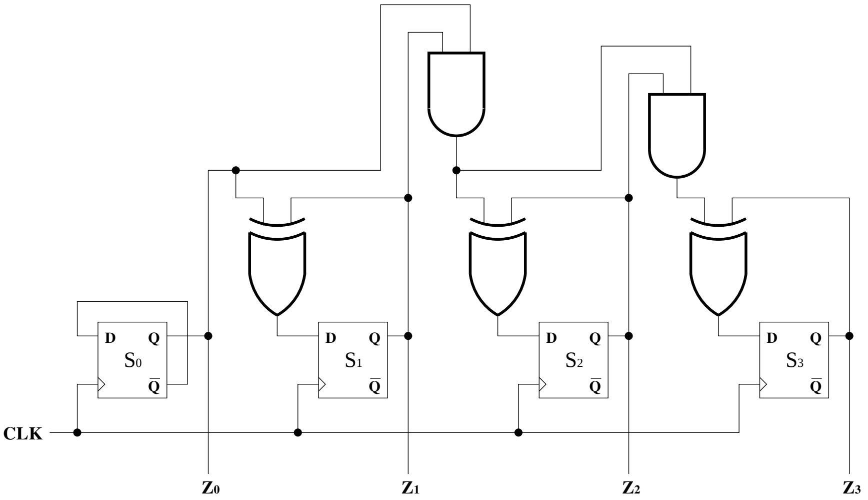
S_2	S_1	S_0	S_2^+	S_1^+	S_0^+
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0



$$
\begin{aligned}
S_2^+ &= \bar{S}_2S_1S_0 + S_2\bar{S}_1 + S_2\bar{S}_0 &&= S_2 \oplus (S_1S_0) \\
S_1^+ &= S_1\bar{S}_0 + \bar{S}_1S_0 &&= S_1 \oplus S_0 \\
S_0^+ &= \bar{S}_0 &&= S_0 \oplus 1
\end{aligned}
$$

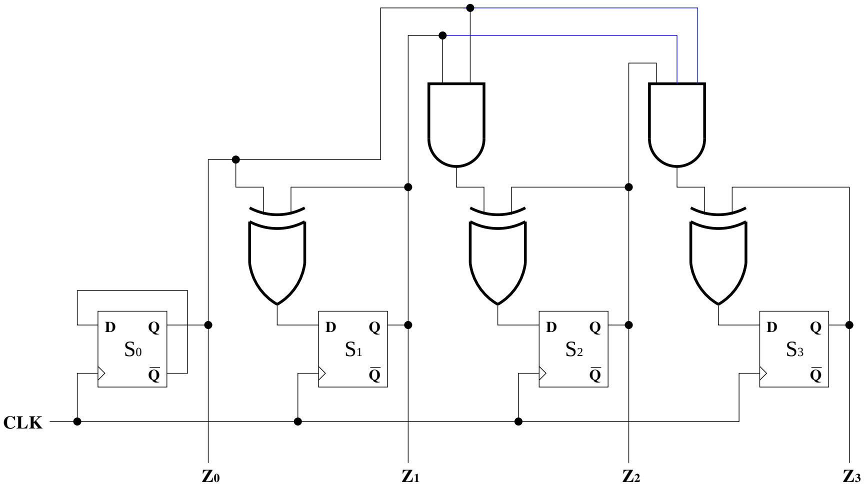
The first form of the expression for each next-state variable is taken directly from the corresponding K-map. We have rewritten each expression to make the emerging pattern more obvious. We can also derive the pattern intuitively by asking the following: given a binary counter in state $S_{N-1}S_{N-2}\ldots S_{j+1}S_jS_{j-1}\ldots S_1S_0$, when does S_j change in the subsequent state? The answer, of course, is that S_j changes when all of the bits below S_j are 1. Otherwise, S_j remains the same in the next state. We thus write $S_j^+ = S_j \oplus (S_{j-1}\ldots S_1S_0)$ and implement the counter as shown below for a 4-bit design. Note that the usual order of output bits along the bottom is reversed in the figure, with the most significant bit at the right rather than the left.

a 4−bit synchronous binary counter with serial gating



The calculation of the left inputs to the XOR gates in the counter shown above is performed with a series of two-input AND gates. Each of these gates AND's another flip-flop value into the product. This approach, called **serial gating**, implies that an N-bit counter requires more than $N-2$ gate delays to settle into the next state. An alternative approach, called **parallel gating**, calculates each input independently with a single logic gate, as shown below. The blue inputs to the AND gate for S_3 highlight the difference from the previous figure (note that the two approaches differ only for bits S_3 and above). With parallel gating, the **fan-in** of the gates (the number of inputs) and the **fan-out** of the flip-flop outputs (number of other gates into which an output feeds) grow with the size of the counter. In practice, large counters use a combination of these two approaches.

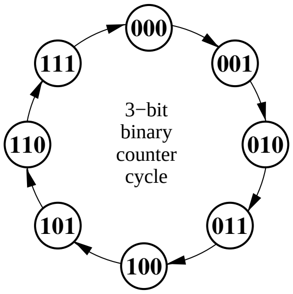
a 4−bit synchronous binary counter with parallel gating



3.1.5 Ripple Counters

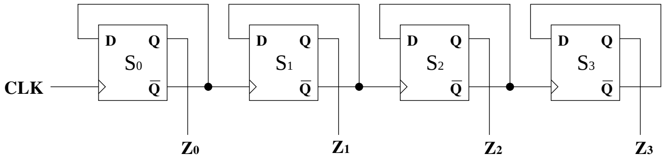
A second class of counter drives some of its flip-flops with a clock signal and feeds flip-flop outputs into the clock inputs of its remaining flip-flops, possibly through additional logic. Such a counter is called a **ripple counter**, because the effect of a clock edge ripples through the flip-flops. The delay inherent to the ripple effect, along with the complexity of ensuring that timing issues do not render the design unreliable, are the major drawbacks of ripple counters. Compared with synchronous counters, however, ripple counters consume less energy, and are sometimes used for devices with restricted energy supplies.

General ripple counters can be tricky because of timing issues, but certain types are easy. Consider the design of binary ripple counter. The state diagram for a 3-bit binary counter is replicated to the right. Looking at the states, notice that the least-significant bit alternates with each state, while higher bits flip whenever the next smaller bit (to the right) transitions from one to zero. To take advantage of these properties, we use positive edge-triggered D flip-flops with their complemented ($\bar{Q}$) outputs wired back to their inputs. The clock input is fed only into the first flip-flop, and the complemented output of each flip-flop is also connected to the clock of the next.



An implementation of a 4-bit binary ripple counter appears to the right. The order of bits in the figure matches the order used for our synchronous binary counters: least significant on the left, most significant on the right. As you can see from the figure, the technique generalizes to arbitrarily large binary ripple counters, but the time required for the outputs to settle after a clock edge scales with the number of flip-flops in the counter. On the other hand, an average of only two flip-flops see each clock edge $(1 + 1/2 + 1/4 + \ldots)$, which reduces the power requirements.[8]

a 4−bit binary ripple counter



[8] Recall that flip-flops record the clock state internally. The logical activity required to record such state consumes energy.

Beginning with the state 0000, at the rising clock edge, the left (S_0) flip-flop toggles to 1. The second (S_1) flip-flop sees this change as a falling clock edge and does nothing, leaving the counter in state 0001. When the next rising clock edge arrives, the left flip-flop toggles back to 0, which the second flip-flop sees as a rising clock edge, causing it to toggle to 1. The third (S_2) flip-flop sees the second flip-flop's change as a falling edge and does nothing, and the state settles as 0010. We leave verification of the remainder of the cycle as an exercise.

3.1.6 Timing Issues*

Ripple counters are a form of a more general strategy known as clock gating.[9] **Clock gating** uses logic to control the visibility of a clock signal to flip-flops (or latches). Historically, digital system designers rarely used clock gating techniques because of the complexity introduced for the circuit designers, who must ensure that clock edges are delivered with little skew along a dynamically changing set of paths to flip-flops. Today, however, the power benefits of hiding the clock signal from flip-flops have made clock gating an attractive strategy. Nevertheless, digital logic designers and computer architects still almost never use clock gating strategies directly. In most of the industry, CAD tools insert logic for clock gating automatically. A handful of companies (such as Intel and Apple/Samsung) design custom circuits rather than relying on CAD tools to synthesize hardware designs from standard libraries of elements. In these companies, clock gating is used widely by the circuit design teams, and some input is occasionally necessary from the higher-level designers.

More aggressive gating strategies are also used in modern designs, but these usually require more time to transition between the on and off states and can be more difficult to get right automatically (with the tools), hence hardware designers may need to provide high-level information about their designs. A flip-flop that does not see any change in its clock input still has connections to high voltage and ground, and thus allows a small amount of **leakage current**. In contrast, with **power gating**, the voltage difference is removed, and the circuit uses no power at all. Power gating can be tricky—as you know, for example, when you turn the power on, you need to make sure that each latch settles into a stable state. Latches may need to be initialized to guarantee that they settle, which requires time after the power is restored.

If you want a deeper understanding of gating issues, take ECE482, Digital Integrated Circuit Design, or ECE527, System-on-a-Chip Design.

3.1.7 Machine Models

Before we dive fully into FSM design, we must point out that we have placed a somewhat artificial restriction on the types of FSMs that we use in our course. Historically, this restriction was given a name, and machines of the type that we have discussed are called Moore machines. However, outside of introductory classes, almost no one cares about this name, nor about the name for the more general model used almost universally in hardware design, Mealy machines.

What is the difference? In a **Moore machine**, outputs depend only on the internal state bits of the FSM (the values stored in the flip-flops). In a **Mealy machine**, outputs may be expressed as functions both of internal state and FSM inputs. As we illustrate shortly, the benefit of using input signals to calculate outputs (the Mealy machine model) is that input bits effectively serve as additional system state, which means that the number of internal state bits can be reduced. The disadvantage of including input signals in the expressions for output signals is that timing characteristics of input signals may not be known, whereas an FSM designer may want to guarantee certain timing characteristics for output signals.

In practice, when such timing guarantees are needed, the designer simply adds state to the FSM to accommodate the need, and the problem is solved. The coin-counting FSM that we designed for our class' lab assignments, for example, required that we use a Moore machine model to avoid sending the servo controlling the coin's path an output pulse that was too short to enforce the FSM's decision about which way to send the coin. By adding more states to the FSM, we were able to hold the servo in place, as desired.

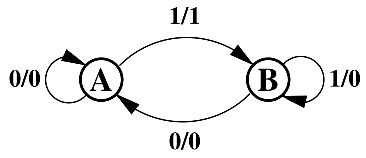
[9] Fall 2012 students: This part may seem a little redundant, but we're going to remove the earlier mention of clock gating in future semesters.

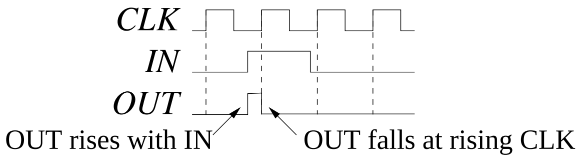
Why are we protecting you from the model used in practice? First, timing issues add complexity to a topic that is complex enough for an introductory course. And, second, most software FSMs are Moore machines, so the abstraction is a useful one in that context, too.

In many design contexts, the timing issues implied by a Mealy model can be relatively simple to manage. When working in a single clock domain, all of the input signals come from flip-flops in the same domain, and are thus stable for most of the clock cycle. Only rarely does one need to keep additional state to improve timing characteristics in these contexts. In contrast, when interacting across clock domains, more care is sometimes needed to ensure correct behavior.

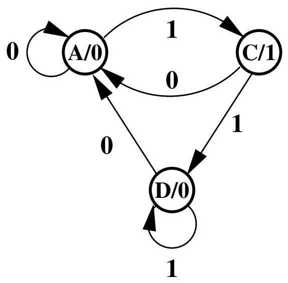
We now illustrate the state reduction benefit of the Mealy machine model with a simple example, an FSM that recognizes the pattern of a 0 followed by a 1 on a single input and outputs a 1 when it observes the pattern. As already mentioned, Mealy machines often require fewer flip-flops. Intuitively, the number of combinations of states and inputs is greater than the number of combinations of states alone, and allowing a function to depend on inputs reduces the number of internal states needed.

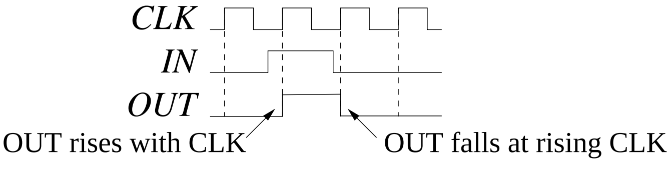
A Mealy implementation of the FSM appears on the left below, and an example timing diagram illustrating the FSM's behavior is shown on the right. The machine shown below occupies state A when the last bit seen was a 0, and state B when the last bit seen was a 1. Notice that the transition arcs in the state diagram are labeled with two values instead of one. Since outputs can depend on input values as well as state, transitions in a Mealy machine are labeled with input/output combinations, while states are labeled only with their internal bits (or just their names, as shown below). Labeling states with outputs does not make sense for a Mealy machine, since outputs may vary with inputs. Notice that the outputs indicated on any given transition hold only until that transition is taken (at the rising clock edge), as is apparent in the timing diagram. When inputs are asynchronous, that is, not driven by the same clock signal, output pulses from a Mealy machine can be arbitrarily short, which can lead to problems.





For a Moore machine, we must create a special state in which the output is high. Doing so requires that we split state B into two states, a state C in which the last two bits seen were 01, and a state D in which the last two bits seen were 11. Only state C generates output 1. State D also becomes the starting state for the new state machine. The state diagram on the left below illustrates the changes, using the transition diagram style that we introduced earlier to represent Moore machines. Notice in the associated timing diagram that the output pulse lasts a full clock cycle.





ECE120: Introduction to Computer Engineering

Notes Set 3.2 Finite State Machine Design Examples, Part I

This set of notes uses a series of examples to illustrate design principles for the implementation of finite state machines (FSMs) using digital logic. We begin with an overview of the design process for a digital FSM, from the development of an abstract model through the implementation of functions for the next-state variables and output signals. Our first few examples cover only the concrete aspects: we implement several counters, which illustrate the basic process of translating a concrete and complete state transition diagram into an implementation based on flip-flops and logic gates. We next consider a counter with a number of states that is not a power of two, with which we illustrate the need for FSM initialization.

We then consider the design process as a whole through a more general example of a counter with multiple inputs to control its behavior. We work from an abstract model down to an implementation, illustrating how semantic knowledge from the abstract model can be used to simplify the implementation. Finally, we illustrate how the choice of representation for the FSM's internal state affects the complexity of the implementation. Fortunately, designs that are more intuitive and easier for humans to understand also typically make the best designs in terms of other metrics, such as logic complexity.

3.2.1 Steps in the Design Process

Before we begin exploring designs, let's talk briefly about the general approach that we take when designing an FSM. We follow a six-step process:

1. develop an abstract model
2. specify I/O behavior
3. complete the specification
4. choose a state representation
5. calculate logic expressions
6. implement with flip-flops and gates

In Step 1, we translate our description in human language into a model with states and desired behavior. At this stage, we simply try to capture the intent of the description and are not particularly thorough nor exact.

Step 2 begins to formalize the model, starting with its input and output behavior. If we eventually plan to develop an implementation of our FSM as a digital system (which is not the only choice, of course!), all input and output must consist of bits. Often, input and/or output specifications may need to match other digital systems to which we plan to connect our FSM. In fact, *most problems in developing large digital systems today arise because of incompatibilities when composing two or more separately designed pieces* (or **modules**) into an integrated system.

Once we know the I/O behavior for our FSM, in Step 3 we start to make any implicit assumptions clear and to make any other decisions necessary to the design. Occasionally, we may choose to leave something undecided in the hope of simplifying the design with "don't care" entries in the logic formulation.

In Step 4, we select an internal representation for the bits necessary to encode the state of our FSM. In practice, for small designs, this representation can be selected by a computer in such a way as to optimize the implementation. However, for large designs, such as the LC-3 instruction set architecture that we study later in this class, humans do most of the work by hand. In the later examples in this set of notes, we show how even a small design can leverage meaningful information from the design when selecting the representation, leading to an implementation that is simpler and is easier to build correctly. We also show how one can use abstraction to simplify an implementation.

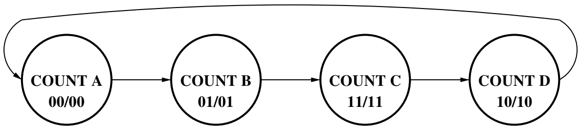
By Step 5, our design is a complete specification in terms of bits, and we need merely derive logic expressions for the next-state variables and the output signals. This process is no different than for combinational logic, and should already be fairly familiar to you.

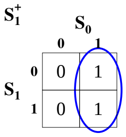
Finally, in Step 6, we translate our logic expressions into gates and use flip-flops (or registers) to hold the internal state bits of the FSM. In later notes, we use more complex building blocks when implementing an FSM, building up abstractions in order to simplify the design process in much the same way that we have shown for combinational logic.

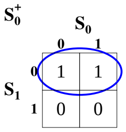
3.2.2 Example: A Two-Bit Gray Code Counter

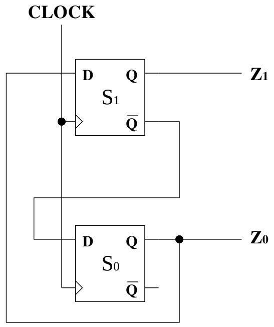
Let's begin with a two-bit Gray code counter with no inputs. As we mentioned in Notes Set 2.1, a Gray code is a cycle over all bit patterns of a certain length in which consecutive patterns differ in exactly one bit. For simplicity, our first few examples are based on counters and use the internal state of the FSM as the output values. You should already know how to design combinational logic for the outputs if it were necessary. The inputs to a counter, if any, are typically limited to functions such as starting and stopping the counter, controlling the counting direction, and resetting the counter to a particular state.

A fully-specified transition diagram for a two-bit Gray code counter appears below. With no inputs, the states simply form a loop, with the counter moving from one state to the next each cycle. Each state in the diagram is marked with the internal state value S_1S_0 (before the "/") and the output Z_1Z_0 (after the "/"), which are always equal for this counter. Based on the transition diagram, we can fill in the K-maps for the next-state values S_1^+ and S_0^+ as shown to the right of the transition diagram, then derive algebraic expressions in the usual way to obtain $S_1^+ = S_0$ and $S_0^+ = \overline{S_1}$. We then use the next-state logic to develop the implementation shown on the far right, completing our first counter design.





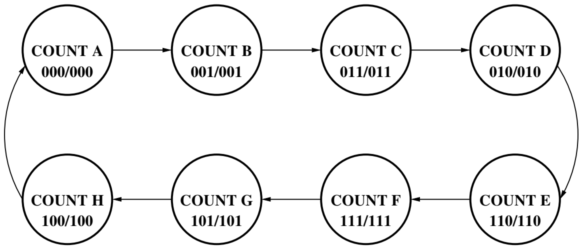




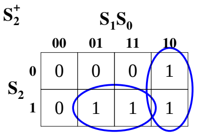
a two−bit Gray code counter

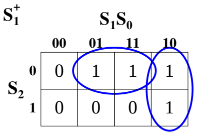
3.2.3 Example: A Three-Bit Gray Code Counter

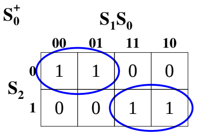
Now we'll add a third bit to our counter, but again use a Gray code as the basis for the state sequence. A fully-specified transition diagram for such a counter appears to the right. As before, with no inputs, the states simply form a loop, with the counter moving from one state to the next each cycle. Each state in the diagram is marked with the internal state value $S_2S_1S_0$ (before "/") and the output $Z_2Z_1Z_0$ (after "/").



Based on the transition diagram, we can fill in the K-maps for the next-state values S_2^+, S_1^+, and S_0^+ as shown to the right, then derive algebraic expressions. The results are more complex this time.



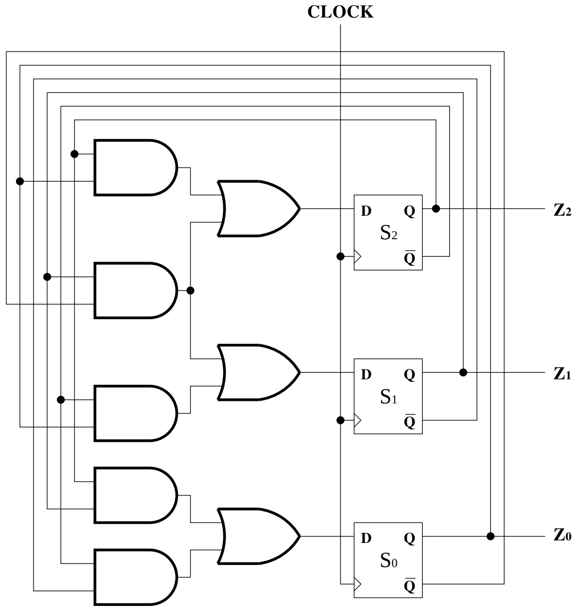




For our next-state logic, we obtain:

$$
\begin{aligned}
S_2^+ &= S_2\ S_0 + S_1\ \overline{S_0} \\
S_1^+ &= \overline{S_2}\ S_0 + S_1\ \overline{S_0} \\
S_0^+ &= \overline{S_2}\ \overline{S_1} + S_2\ S_1
\end{aligned}
$$

Notice that the equations for S_2^+ and S_1^+ share a common term, $S_1\overline{S_0}$. This design does not allow much choice in developing good equations for the next-state logic, but some designs may enable you to reduce the design complexity by explicitly identifying and making use of common algebraic terms and sub-expressions for different outputs. In modern design processes, identifying such opportunities is generally performed by a computer program, but it's important to understand how they arise. Note that the common term becomes a single AND gate in the implementation of our counter, as shown to the right.



a three−bit Gray code counter

Looking at the counter's implementation diagram, notice that the vertical lines carrying the current state values and their inverses back to the next state logic inputs have been carefully ordered to simplify understanding the diagram. In particular, they are ordered from left to right (on the left side of the figure) as $\overline{S_0}S_0\overline{S_1}S_1\overline{S_2}S_2$. When designing any logic diagram, be sure to make use of a reasonable order so as to make it easy for someone (including yourself!) to read and check the correctness of the logic.

3.2.4 Example: A Color Sequencer

Early graphics systems used a three-bit red-green-blue (RGB) encoding for colors. The color mapping for such a system is shown to the right.

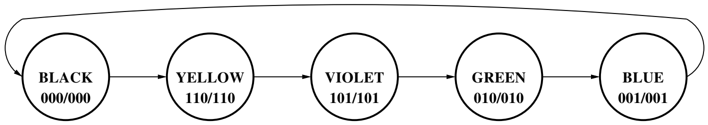
RGB	color
000	black
001	blue
010	green
011	cyan
100	red
101	violet
110	yellow
111	white

Imagine that you are charged with creating a counter to drive a light through a sequence of colors. The light takes an RGB input as just described, and the desired pattern is

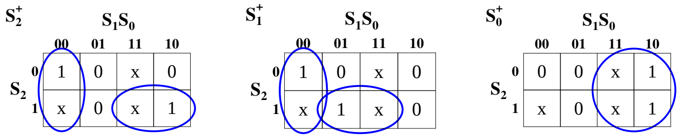
off (black) yellow violet green blue

You immediately recognize that you merely need a counter with five states. How many flip-flops will we need? At least three, since $\lceil \log_2 (5) \rceil = 3$. Given that we need three flip-flops, and that the colors we need to produce as outputs are all unique bit patterns, we can again choose to use the counter's internal state directly as our output values.

A fully-specified transition diagram for our color sequencer appears to the right. The states again form a loop, and are marked with the internal state value $S_2S_1S_0$ and the output RGB.



As before, we can use the transition diagram to fill in K-maps for the next-state values S_2^+, S_1^+, and S_0^+, as shown to the right. For each of the three states not included in our transition diagram, we have inserted x's into the K-maps to indicate "don't care." As you know, we can treat each x as either a 0 or a 1, whichever produces better results (where "better" usually means simpler equations). The terms that we have chosen for our algebraic equations are illustrated in the K-maps. The x's within the ellipses become 1s in the implementation, and the x's outside of the ellipses become 0s.



S_2^+	S_1S_0 00	01	11	10
S_2 0	1	0	x	0
1	x	0	x	1

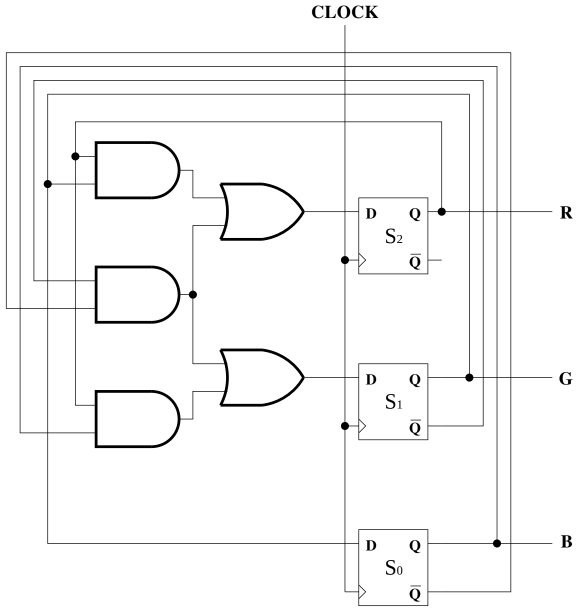
S_1^+	S_1S_0 00	01	11	10
S_2 0	1	0	x	0
1	x	1	x	0

S_0^+	S_1S_0 00	01	11	10
S_2 0	0	0	x	1
1	x	0	x	1

For our next-state logic, we obtain:

$$\begin{aligned} S_2^+ &= S_2\ S_1 + \overline{S_1}\ \overline{S_0} \\ S_1^+ &= S_2\ S_0 + \overline{S_1}\ \overline{S_0} \\ S_0^+ &= S_1 \end{aligned}$$

Again our equations for S_2^+ and S_1^+ share a common term, which becomes a single AND gate in the implementation shown to the right.



an RGB color sequencer

3.2.5 Identifying an Initial State

Let's say that you go the lab and build the implementation above, hook it up to the light, and turn it on. Does it work? Sometimes. Sometimes it works perfectly, but sometimes the light glows cyan or red briefly first. At other times, the light is an unchanging white.

What could be going wrong?

Let's try to understand. We begin by deriving K-maps for the implementation, as shown to the right. In these K-maps, each of the x's in our design has been replaced by either a 0 or a 1. These entries are highlighted with green italics.

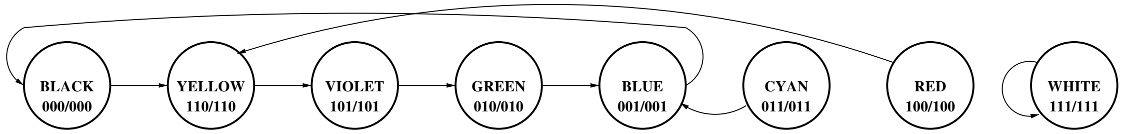
S_2^+	S_1S_0			
S_2	00	01	11	10
0	1	0	*0*	0
1	*1*	0	*1*	1

S_1^+	S_1S_0			
S_2	00	01	11	10
0	1	0	*0*	0
1	*1*	1	*1*	0

S_0^+	S_1S_0			
S_2	00	01	11	10
0	0	0	*1*	1
1	*0*	0	*1*	1

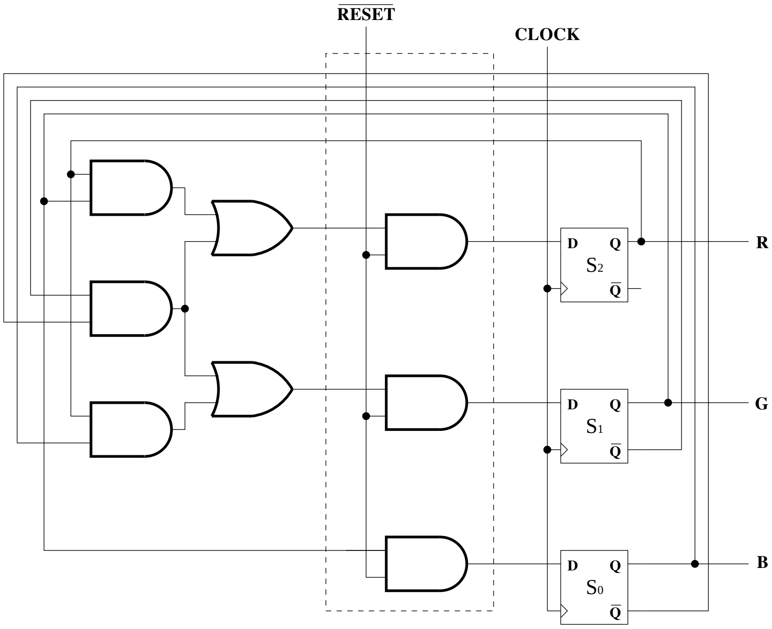
Now let's imagine what might happen if somehow our FSM got into the $S_2S_1S_0 = 111$ state. In such a state, the light would appear white, since $RGB = S_2S_1S_0 = 111$. What happens in the next cycle? Plugging into the equations or looking into the K-maps gives (of course) the same answer: the next state is the $S_2^+S_1^+S_0^+ = 111$ state. In other words, the light stays white indefinitely! As an exercise, you should check what happens if the light is red or cyan.

We can extend the transition diagram that we developed for our design with the extra states possible in the implementation, as shown below. As with the five states in the design, the extra states are named with the color of light that they produce.



Notice that the FSM does not move out of the WHITE state (ever). You may at this point wonder whether more careful decisions in selecting our next-state expressions might address this issue. To some extent, yes. For example, if we replace the S_2S_1 term in the equation for S_2^+ with $S_2\overline{S_0}$, a decision allowed by the "don't care" boxes in the K-map for our design, the resulting transition diagram does not suffer from the problem that we've found. However, even if we do change our implementation slightly, we need to address another aspect of the problem: how can the FSM ever get into the unexpected states?

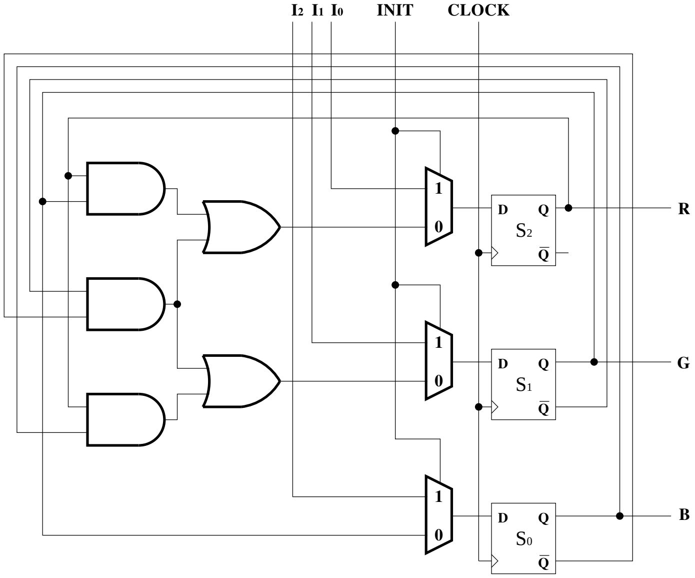
What is the initial state of the three flip-flops in our implementation? *The initial state may not even be 0s and 1s unless we have an explicit mechanism for initialization.* Initialization can work in two ways. The first approach makes use of the flip-flop design. As you know, a flip-flop is built from a pair of latches, and we can make use of the internal reset lines on these latches to force each flip-flop into the 0 state (or the 1 state) using an additional input.



an RGB color sequencer with reset

Alternatively, we can add some extra logic to our design. Consider adding a few AND gates and a $\overline{RESET}$ input (active low), as shown in the dashed box in the figure to the right. In this case, when we assert $\overline{RESET}$ by setting it to 0, the FSM moves to state 000 in the next cycle, putting it into the BLACK state. The approach taken here is for clarity; one can optimize the design, if desired. For example, we could simply connect $\overline{RESET}$ as an extra input into the three AND gates on the left rather than adding new ones, with the same effect.

We may sometimes want a more powerful initialization mechanism—one that allows us to force the FSM into any specific state in the next cycle. In such a case, we can add multiplexers to each of our flip-flop inputs, allowing us to use the $INIT$ input to choose between normal operation ($INIT = 0$) of the FSM and forcing the FSM into the next state given by $I_2I_1I_0$ (when $INIT = 1$).



an RGB color sequencer with arbitrary initialization

3.2.6 Developing an Abstract Model

We are now ready to discuss the design process for an FSM from start to finish. For this first abstract FSM example, we build upon something that we have already seen: a two-bit Gray code counter. We now want a counter that allows us to start and stop the count. What is the mechanism for stopping and starting? To begin our design, we could sketch out an abstract next-state table such as the one shown to the right above. In this form of the table, the first column lists the states, while each of the other columns lists states to which the FSM transitions after a clock cycle for a particular input combination. The table contains two states, counting and halted, and specifies that the design uses two distinct buttons to move between the states.

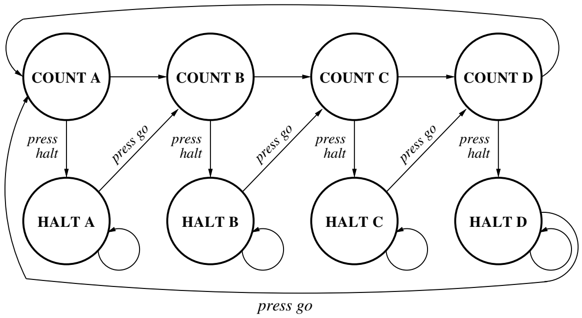
state	no input	halt button	go button
counting	counting	halted	
halted	halted		counting

A counter with a single counting state, of course, does not provide much value. We extend the table with four counting states and four halted states, as shown to the right. This version of the table also introduces more formal state names, for which these notes use all capital letters.

state	no input	halt button	go button
COUNT A	COUNT B	HALT A	
COUNT B	COUNT C	HALT B	
COUNT C	COUNT D	HALT C	
COUNT D	COUNT A	HALT D	
HALT A	HALT A		COUNT B
HALT B	HALT B		COUNT C
HALT C	HALT C		COUNT D
HALT D	HALT D		COUNT A

The upper four states represent uninterrupted counting, in which the counter cycles through these states indefinitely. A user can stop the counter in any state by pressing the "halt" button, causing the counter to retain its current value until the user presses the "go" button.

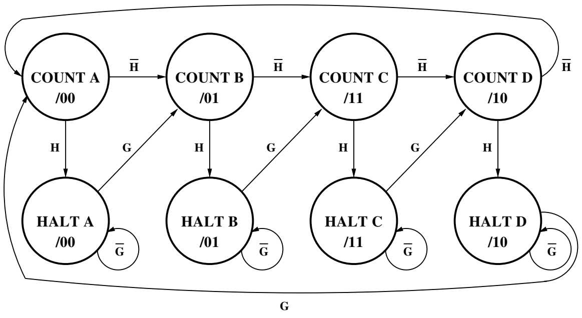
Below the state table is an abstract transition diagram, which provides exactly the same information in graphical form. Here circles represent states (as labeled) and arcs represent transitions from one state to another based on an input combination (which is used to label the arc).



We have already implicitly made a few choices about our counter design. First, the counter shown retains the current state of the system when "halt" is pressed. We could instead reset the counter state whenever it is restarted, in which case we need only five states: four for counting and one more for a halted counter. Second, we've designed the counter to stop when the user presses "halt" and to resume counting when the user presses "go." We could instead choose to delay these effects by a cycle. For example, pressing "halt" in state COUNT B could take the counter to state HALT C, and pressing "go" in state HALT C could take the system to state COUNT C. In these notes, we implement only the diagrams shown.

3.2.7 Specifying I/O Behavior

We next start to formalize our design by specifying its input and output behavior digitally. Each of the two control buttons provides a single bit of input. The "halt" button we call H, and the "go" button we call G. For the output, we use a two-bit Gray code. With these choices, we can redraw the transition diagram as show to the right.



In this figure, the states are marked with output values Z_1Z_0 and transition arcs are labeled in terms of our two input buttons, G and H. The uninterrupted counting cycle is labeled with $\overline{H}$ to indicate that it continues until we press H.

3.2.8 Completing the Specification

Now we need to think about how the system should behave if something outside of our initial expectations occurs. Having drawn out a partial transition diagram can help with this process, since we can use the diagram to systematically consider all possible input conditions from all possible states. The state table form can make the missing parts of the specification even more obvious.

For our counter, the symmetry between counting states makes the problem substantially simpler. Let's write out part of a list of states and part of a state table with one counting state and one halt state, as shown to the right. Four values of the inputs HG are possible (recall that N bits allow 2^N possible patterns). We list the columns in Gray code order, since we may want to transcribe this table into K-maps later.

state		description
first counting state	COUNT A	counting, output $Z_1Z_0 = 00$
first halted state	HALT A	halted, output $Z_1Z_0 = 00$

	HG			
state	00	01	11	10
COUNT A	COUNT B	unspecified	unspecified	HALT A
HALT A	HALT A	COUNT B	unspecified	unspecified

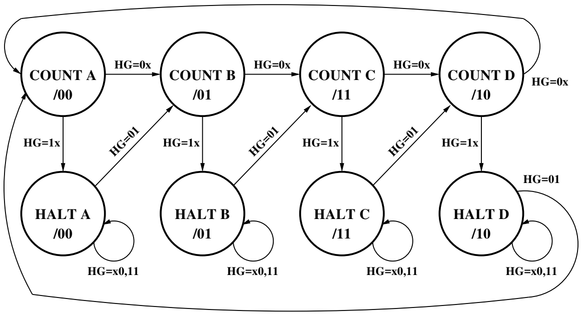
Let's start with the COUNT A state. We know that if neither button is pressed ($HG = 00$), we want the counter to move to the COUNT B state. And, if we press the "halt" button ($HG = 10$), we want the counter to move to the HALT A state. What should happen if a user presses the "go" button ($HG = 01$)? Or if the user presses both buttons ($HG = 11$)? Answering these questions is part of fully specifying our design. We can choose to leave some parts unspecified, but *any implementation of our system will imply answers*, and thus we must be careful. We choose to ignore the "go" button while counting, and to have the "halt" button override the "go" button. Thus, if $HG = 01$ when the counter is in state COUNT A, the counter moves to state COUNT B. And, if $HG = 11$, the counter moves to state HALT A.

Use of explicit bit patterns for the inputs HG may help you to check that all four possible input values are covered from each state. If you choose to use a transition diagram instead of a state table, you might even want to add four arcs from each state, each labeled with a specific value of HG. When two arcs connect the same two states, we can either use multiple labels or can indicate bits that do not matter using a **don't-care** symbol, x. For example, the arc from state COUNT A to state COUNT B could be labeled $HG = 00, 01$ or $HG = 0x$. The arc from state COUNT A to state HALT A could be labeled $HG = 10, 11$ or $HG = 1x$. We can also use logical expressions as labels, but such notation can obscure unspecified transitions.

Now consider the state HALT A. The transitions specified so far are that when we press "go" ($HG = 01$), the counter moves to the COUNT B state, and that the counter remains halted in state HALT A if no buttons are pressed ($HG = 00$). What if the "halt" button is pressed ($HG = 10$), or both buttons are pressed ($HG = 11$)? For consistency, we decide that "halt" overrides "go," but does nothing special if it alone is pressed while the counter is halted. Thus, input patterns $HG = 10$ and $HG = 11$ also take state HALT A back to itself. Here the arc could be labeled $HG = 00, 10, 11$ or, equivalently, $HG = 00, 1x$ or $HG = x0, 11$.

To complete our design, we apply the same decisions that we made for the COUNT A state to all of the other counting states, and the decisions that we made for the HALT A state to all of the other halted states. If we had chosen not to specify an answer, an implementation could produce different behavior from the different counting and/or halted states, which might confuse a user.

The resulting design appears to the right.

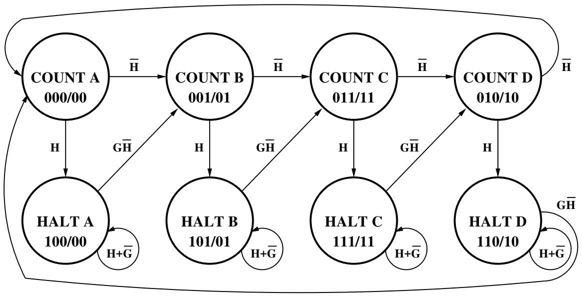


3.2.9 Choosing a State Representation

Now we need to select a representation for the states. Since our counter has eight states, we need at least three ($\lceil \log_2 \ (8) \rceil = 3$) state bits $S_2S_1S_0$ to keep track of the current state. As we show later, *the choice of representation for an FSM's states can dramatically affect the design complexity.* For a design as simple as our counter, you could just let a computer implement all possible representations (there aren't more than 840, if we consider simple symmetries) and select one according to whatever metrics are interesting. For bigger designs, however, the number of possibilities quickly becomes impossible to explore completely.

Fortunately, *use of abstraction in selecting a representation also tends to produce better designs* for a wide variety of metrics (such as design complexity, area, power consumption, and performance). The right strategy is thus often to start by selecting a representation that makes sense to a human, even if it requires more bits than are strictly necessary. The resulting implementation will be easier to design and to debug than an implementation in which only the global behavior has any meaning.

Let's return to our specific example, the counter. We can use one bit, S_2, to record whether or not our counter is counting ($S_2 = 0$) or halted ($S_2 = 1$). The other two bits can then record the counter state in terms of the desired output. Choosing this representation implies that only wires will be necessary to compute outputs Z_1 and Z_0 from the internal state: $Z_1 = S_1$ and $Z_0 = S_0$. The resulting design, in which states are now labeled with both internal state and outputs ($S_2S_1S_0/Z_1Z_0$) appears to the right. In this version, we have changed the arc labeling to use logical expressions, which can sometimes help us to think about the implementation.

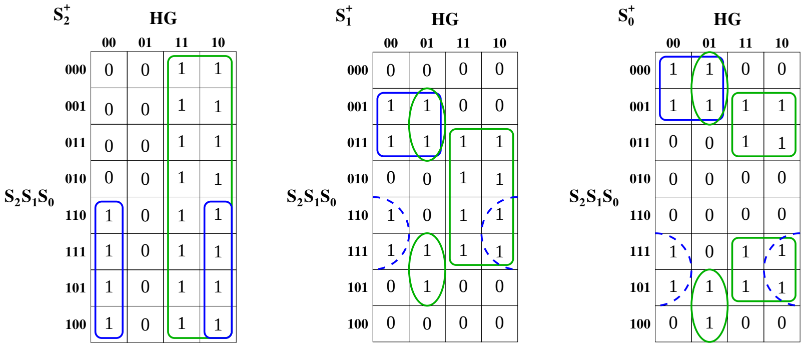


The equivalent state listing and state table appear below. We have ordered the rows of the state table in Gray code order to simplify transcription of K-maps.

state	$S_2S_1S_0$	description
COUNT A	000	counting, output $Z_1Z_0 = 00$
COUNT B	001	counting, output $Z_1Z_0 = 01$
COUNT C	011	counting, output $Z_1Z_0 = 11$
COUNT D	010	counting, output $Z_1Z_0 = 10$
HALT A	100	halted, output $Z_1Z_0 = 00$
HALT B	101	halted, output $Z_1Z_0 = 01$
HALT C	111	halted, output $Z_1Z_0 = 11$
HALT D	110	halted, output $Z_1Z_0 = 10$

		HG			
state	$S_2S_1S_0$	00	01	11	10
COUNT A	000	001	001	100	100
COUNT B	001	011	011	101	101
COUNT C	011	010	010	111	111
COUNT D	010	000	000	110	110
HALT D	110	110	000	110	110
HALT C	111	111	010	111	111
HALT B	101	101	011	101	101
HALT A	100	100	001	100	100

Having chosen a representation, we can go ahead and implement our design in the usual way. As shown to the right, K-maps for the next-state logic are complicated, since we have five variables and must consider implicants that are not contiguous in the K-maps. The S_2^+ logic is easy enough: we only need two terms, as shown.



S_2^+

$S_2S_1S_0$ \ HG	00	01	11	10
000	0	0	1	1
001	0	0	1	1
011	0	0	1	1
010	0	0	1	1
110	1	0	1	1
111	1	0	1	1
101	1	0	1	1
100	1	0	1	1

S_1^+

$S_2S_1S_0$ \ HG	00	01	11	10
000	0	0	0	0
001	1	1	0	0
011	1	1	1	1
010	0	0	1	1
110	1	0	1	1
111	1	1	1	1
101	0	1	0	0
100	0	0	0	0

S_0^+

$S_2S_1S_0$ \ HG	00	01	11	10
000	1	1	0	0
001	1	1	1	1
011	0	0	1	1
010	0	0	0	0
110	0	0	0	0
111	1	0	1	1
101	1	1	1	1
100	0	1	0	0

Notice that we have used color and line style to distinguish different implicants in the K-maps. Furthermore, the symmetry of the design produces symmetry in the S_1^+ and S_0^+ formula, so we have used the same color and line style for analogous terms in these two K-maps. For S_1^+, we need four terms. The green ellipses in the $HG = 01$ column are part of the same term, as are the two halves of the dashed blue circle. In S_0^+, we still need four terms, but three of them are split into two pieces in the K-map. As you can see, the utility of the K-map is starting to break down with five variables.

3.2.10 Abstracting Design Symmetries

Rather than implementing the design as two-level logic, let's try to take advantage of our design's symmetry to further simplify the logic (we reduce gate count at the expense of longer, slower paths).

Looking back to the last transition diagram, in which the arcs were labeled with logical expressions, let's calculate an expression for when the counter should retain its current value in the next cycle. We call this variable $HOLD$. In the counting states, when $S_2 = 0$, the counter stops (moves into a halted state without changing value) when H is true. In the halted states, when $S_2 = 1$, the counter stops (stays in a halted state) when $H + \overline{G}$ is true. We can thus write

$$\begin{aligned} HOLD &= \overline{S_2} \cdot H + S_2 \cdot (H + \overline{G}) \\ HOLD &= \overline{S_2}H + S_2H + S_2\overline{G} \\ HOLD &= H + S_2\overline{G} \end{aligned}$$

In other words, the counter should hold its current value (stop counting) if we press the "halt" button or if the counter was already halted and we didn't press the "go" button. As desired, the current value of the counter (S_1S_0) has no impact on this decision. You may have noticed that the expression we derived for $HOLD$ also matches S_2^+, the next-state value of S_2 in the K-map on the previous page.

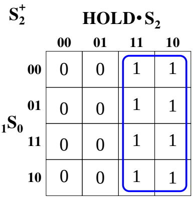
Now let's re-write our state transition table in terms of $HOLD$. The left version uses state names for clarity; the right uses state values to help us transcribe K-maps.

		$HOLD$	
state	$S_2S_1S_0$	0	1
COUNT A	000	COUNT B	HALT A
COUNT B	001	COUNT C	HALT B
COUNT C	011	COUNT D	HALT C
COUNT D	010	COUNT A	HALT D
HALT A	100	COUNT B	HALT A
HALT B	101	COUNT C	HALT B
HALT C	111	COUNT D	HALT C
HALT D	110	COUNT A	HALT D

		$HOLD$	
state	$S_2S_1S_0$	0	1
COUNT A	000	001	100
COUNT B	001	011	101
COUNT C	011	010	111
COUNT D	010	000	110
HALT A	100	001	100
HALT B	101	011	101
HALT C	111	010	111
HALT D	110	000	110

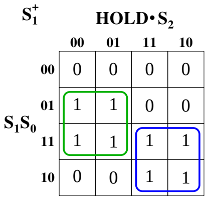
The K-maps based on the $HOLD$ abstraction are shown to the right. As you can see, the necessary logic has been simplified substantially, requiring only two terms each for both S_1^+ and S_0^+. Writing the next-state logic algebraically, we obtain

$$\begin{aligned} S_2^+ &= HOLD \\ S_1^+ &= \overline{HOLD} \cdot S_0 + HOLD \cdot S_1 \\ S_0^+ &= \overline{HOLD} \cdot \overline{S_1} + HOLD \cdot S_0 \end{aligned}$$



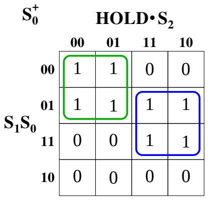
S_2^+

S_1S_0 \ $HOLD \cdot S_2$	00	01	11	10
00	0	0	1	1
01	0	0	1	1
11	0	0	1	1
10	0	0	1	1



S_1^+

S_1S_0 \ $HOLD \cdot S_2$	00	01	11	10
00	0	0	0	0
01	1	1	0	0
11	1	1	1	1
10	0	0	1	1



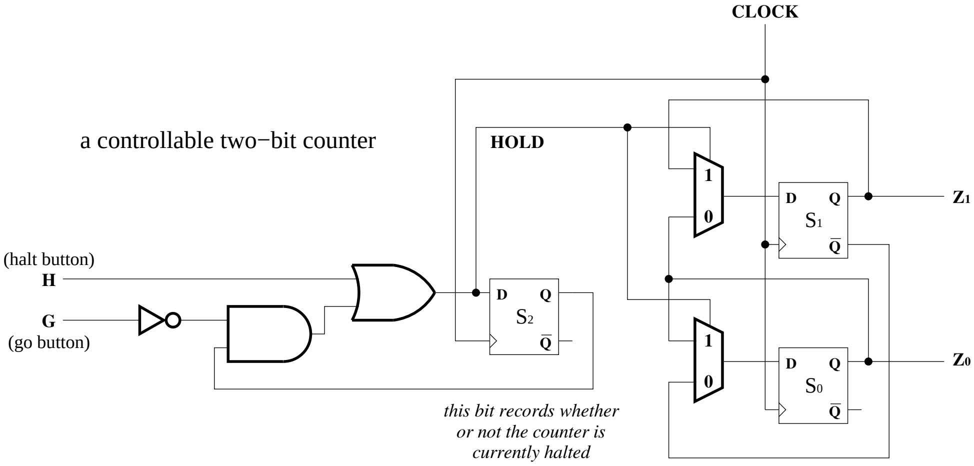
S_0^+

S_1S_0 \ $HOLD \cdot S_2$	00	01	11	10
00	1	1	0	0
01	1	1	1	1
11	0	0	1	1
10	0	0	0	0

Notice the similarity between the equations for $S_1^+S_0^+$ and the equations for a 2-to-1 mux: when $HOLD = 1$, the counter retains its state, and when $HOLD = 0$, it counts.

An implementation appears below. By using semantic meaning in our choice of representation—in particular the use of S_2 to record whether the counter is currently halted ($S_2 = 1$) or counting ($S_2 = 0$)—we have enabled ourselves to separate out the logic for deciding whether to advance the counter fairly cleanly from the logic for advancing the counter itself. Only the $HOLD$ bit in the diagram is used to determine whether or not the counter should advance in the current cycle.

Let's check that the implementation matches our original design. Start by verifying that the $HOLD$ variable is calculated correctly, $HOLD = H + S_2\overline{G}$, then look back at the K-map for S_2^+ in the low-level design to verify that the expression we used does indeed match.



Next, check the mux abstraction. When $HOLD = 1$, the next-state logic for S_1^+ and S_0^+ reduces to $S_1^+ = S_1$ and $S_0^+ = S_0$; in other words, the counter stops counting and simply stays in its current state. When $HOLD = 0$, these equations become $S_1^+ = S_0$ and $S_0^+ = \overline{S_1}$, which produces the repeating sequence for S_1S_0 of 00, 01, 11, 10, as desired. You may want to look back at our two-bit Gray code counter design to compare the next-state equations.

We can now verify that the implementation produces the correct transition behavior. In the counting states, $S_2 = 0$, and the $HOLD$ value simplifies to $HOLD = H$. Until we push the "halt" button, S_2 remains 0, and and the counter continues to count in the correct sequence. When $H = 1$, $HOLD = 1$, and the counter stops at its current value ($S_2^+S_1^+S_0^+ = 1S_1S_0$, which is shorthand for $S_2^+ = 1$, $S_1^+ = S_1$, and $S_0^+ = S_0$).

In any of the halted states, $S_2 = 1$, and we can reduce $HOLD$ to $HOLD = H + \overline{G}$. Here, so long as we press the "halt" button or do not press the "go" button, the counter stays in its current state, because $HOLD = 1$. If we release "halt" and press "go," we have $HOLD = 0$, and the counter resumes counting ($S_2^+S_1^+S_0^+ = 0S_0\overline{S_1}$, which is shorthand for $S_2^+ = 0$, $S_1^+ = S_0$, and $S_0^+ = \overline{S_1}$). We have now verified the implementation.

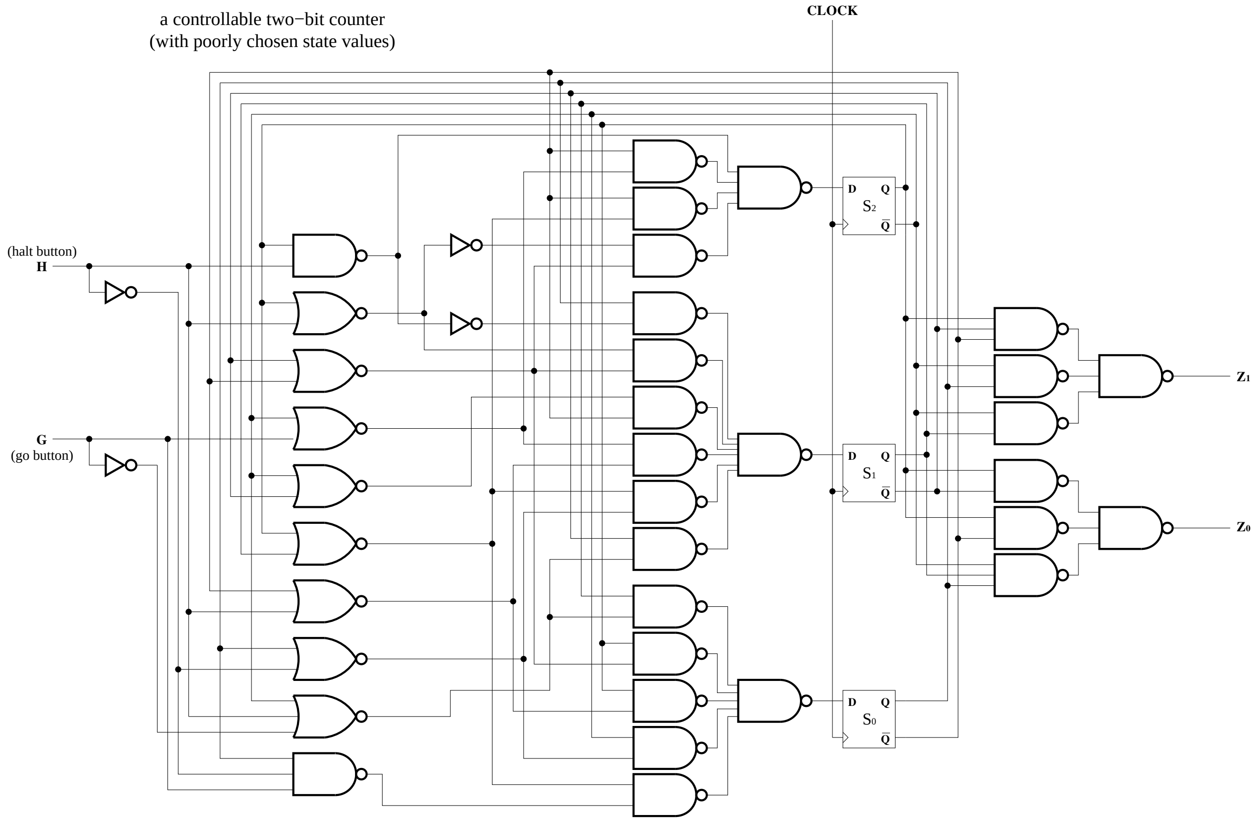
What if you wanted to build a three-bit Gray code counter with the same controls for starting and stopping? You could go back to basics and struggle with six-variable K-maps. Or you could simply copy the $HOLD$ mechanism from the two-bit design above, insert muxes between the next state logic and the flip-flops of the three-bit Gray code counter that we designed earlier, and control the muxes with the $HOLD$ bit. Abstraction is a powerful tool.

3.2.11 Impact of the State Representation

What happens if we choose a bad representation? For the same FSM—the two-bit Gray code counter with start and stop inputs—the table below shows a poorly chosen mapping from states to internal state representation. Below the table is a diagram of an implementation using that representation. Verifying that the implementation's behavior is correct is left as an exercise for the determined reader.

state	$S_2S_1S_0$
COUNT A	000
COUNT B	101
COUNT C	011
COUNT D	010

state	$S_2S_1S_0$
HALT A	111
HALT B	110
HALT C	100
HALT D	001


a controllable two−bit counter
(with poorly chosen state values)
CLOCK
(halt button)
H
G
(go button)
D
Q
S2
Q̄
D
Q
S1
Q̄
D
Q
S0
Q̄
Z1
Z0

ECE120: Introduction to Computer Engineering

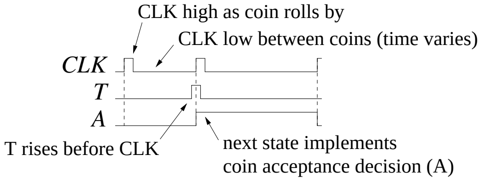
Notes Set 3.3 Design of the Finite State Machine for the Lab

This set of notes explains the process that Prof. Jones used to develop the FSM for the lab. The lab simulates a vending machine mechanism for automatically identifying coins (dimes and quarters only), tracking the amount of money entered by the user, accepting or rejecting coins, and emitting a signal when a total of 35 cents has been accepted. In the lab, we will only drive a light with the "paid in full" signal. Sorry, no candy nor Dew will be distributed!

3.3.1 Physical Design, Sensors, and Timing

The physical elements of the lab were designed by Prof. Chris Schmitz and constructed with some help from the ECE shop. A user inserts a coin into a slot at one end of the device. The coin then rolls down a slope towards a gate controlled by a servo. The gate can be raised or lowered, and determines whether the coin exits from the other side or the bottom of the device. As the coin rolls, it passes two optical sensors.[10] One of these sensors is positioned high enough above the slope that a dime passes beneath the sesnor, allowing the signal T produced by the sensor to tell us whether the coin is a dime or a quarter. The second sensor is positioned so that all coins pass in front of it. The sensor positions are chosen carefully to ensure that, in the case of a quarter, the coin is still blocking the first sensor when it reaches the second sensor. Blocked sensors give a signal of 1 in this design, so the rising edge the signal from the second sensor can be used as a "clock" for our FSM. When the rising edge occurs, the signal T from the first sensor indicates whether the coin is a quarter ($T = 1$) or a dime ($T = 0$).

A sample timing diagram for the lab appears to the right. The clock signal generated by the lab is not only not a square wave—in other words, the high and low portions are not equal—but is also unlikely to be periodic. Instead, the "cycle" is defined by the time between coin insertions. The T signal serves as the single input to our FSM. In the timing diagram, T is shown as rising and falling before the clock edge. We use positive edge-triggered flip-flops to implement our FSM, thus the aspect of the relative timing that matters to our design is that, when the clock rises, the value of T is stable and indicates the type of coin entered. The signal T may fall before or after the clock does—the two are equivalent for our FSM's needs.


CLK high as coin rolls by
CLK low between coins (time varies)
CLK
T
A
T rises before CLK
next state implements
coin acceptance decision (A)

The signal A in the timing diagram is an output from the FSM, and indicates whether or not the coin should be accepted. This signal controls the servo that drives the gate, and thus determines whether the coin is accepted ($A = 1$) as payment or rejected ($A = 0$) and returned to the user.

Looking at the timing diagram, you should note that our FSM makes a decision based on its current state and the input T and enters a new state at the rising clock edge. The value of A in the next cycle thus determines the position of the gate when the coin eventually rolls to the end of the slope. As we said earlier, our FSM is thus a Moore machine: the output A does not depend on the input T, but only on the current internal state bits of the the FSM. However, you should also now realize that making A depend on T is not adequate for this lab. If A were to rise with T and fall with the rising clock edge (on entry to the next state), or even fall with the falling edge of T, the gate would return to the reject position by the time the coin reached the gate, regardless of our FSM's decision!

[10] The full system actually allows four sensors to differentiate four types of coins, but our lab uses only two of these sensors.

3.3.2 An Abstract Model

state	dime ($T = 0$)	quarter ($T = 1$)	accept? (A)	paid? (P)
START	DIME	QUARTER		no
DIME		PAID	yes	no
QUARTER	PAID		yes	no
PAID			yes	yes

We start by writing down states for a user's expected behavior. Given the fairly tight constraints that we have placed on our lab, few combinations are possible. For a total of 35 cents, a user should either insert a dime followed by a quarter, or a quarter followed by a dime. We begin in a START state, which transitions to states DIME or QUARTER when the user inserts the first coin. With no previous coin, we need not specify a value for A. No money has been deposited, so we set output $P = 0$ in the START state. We next create DIME and QUARTER states corresponding to the user having entered one coin. The first coin should be accepted, but more money is needed, so both of these states output $A = 1$ and $P = 0$. When a coin of the opposite type is entered, each state moves to a state called PAID, which we use for the case in which a total of 35 cents has been received. For now, we ignore the possibility that the same type of coin is deposited more than once. Finally, the PAID state accepts the second coin ($A = 1$) and indicates that the user has paid the full price of 35 cents ($P = 1$).

state	dime ($T = 0$)	quarter ($T = 1$)	accept? (A)	paid? (P)
START	DIME	QUARTER		no
DIME	REJECTD	PAID	yes	no
REJECTD	REJECTD	PAID	no	no
QUARTER	PAID	REJECTQ	yes	no
REJECTQ	PAID	REJECTQ	no	no
PAID			yes	yes

We next extend our design to handle user mistakes. If a user enters a second dime in the DIME state, our FSM should reject the coin. We create a REJECTD state and add it as the next state from DIME when a dime is entered. The REJECTD state rejects the dime ($A = 0$) and continues to wait for a quarter ($P = 0$). What should we use as next states from REJECTD? If the user enters a third dime (or a fourth, or a fifth, and so on), we want to reject the new dime as well. If the user enters a quarter, we want to accept the coin, at which point we have received 35 cents (counting the first dime). We use this reasoning to complete the description of REJECTD. We also create an analogous state, REJECTQ, to handle a user who inserts more than one quarter.

What should happen after a user has paid 35 cents and bought one item? The FSM at that point is in the PAID state, which delivers the item by setting $P = 1$. Given that we want the FSM to allow the user to purchase another item, how should we choose the next states from PAID? The behavior that we want from PAID is identical to the behavior that we defined from START. The 35 cents already deposited was used to pay for the item delivered, so the machine is no longer holding any of the user's money. We can thus simply set the next states from PAID to be DIME when a dime is inserted and QUARTER when a quarter is inserted.

state	dime ($T = 0$)	quarter ($T = 1$)	accept? (A)	paid? (P)
PAID	DIME	QUARTER	yes	yes
DIME	REJECTD	PAID	yes	no
REJECTD	REJECTD	PAID	no	no
QUARTER	PAID	REJECTQ	yes	no
REJECTQ	PAID	REJECTQ	no	no

At this point, we make a decision intended primarily to simplify the logic needed to build the lab. Without a physical item delivery mechanism with a specification for how its input must be driven, the behavior of the output signal P can be fairly flexible. For example, we could build a delivery mechanism that used the rising edge of P to open a chute. In this case, the output $P = 0$ in the start state is not relevant, and we can merge the state START with the state PAID. The way that we handle P in the lab, we might find it strange to have a "paid" light turn on before inserting any money, but keeping the design simple enough for a first lab exercise is more important. Our final abstract state table appears above.

3.3.3 Picking the Representation

We are now ready to choose the state representation for the lab FSM. With five states, we need three bits of internal state. Prof. Jones decided to leverage human meaning in assigning the bit patterns, as follows:

S_2	type of last coin inserted (0 for dime, 1 for quarter)
S_1	more than one quarter inserted? (1 for yes, 0 for no)
S_0	more than one dime inserted? (1 for yes, 0 for no)

These meanings are not easy to apply to all of our states. For example, in the PAID state, the last coin inserted may have been of either type, or of no type at all, since we decided to start our FSM in that state as well. However, for the other four states, the meanings provide a clear and unique set of bit pattern assignments, as shown to the right. We can choose any of the remaining four bit patterns (010, 011, 101, or 111) for the PAID state. In fact, *we can choose all of the remaining patterns* for the PAID state. We can always represent any state with more than one pattern if we have spare patterns available. Prof. Jones used this freedom to simplify the logic design.

state	$S_2S_1S_0$
PAID	???
DIME	000
REJECTD	001
QUARTER	100
REJECTQ	110

This particular example is slightly tricky. The four free patterns do not share any single bit in common, so we cannot simply insert x's into all K-map entries for which the next state is PAID. For example, if we insert an x into the K-map for S_2^+, and then choose a function for S_2^+ that produces a value of 1 in place of the don't care, we must also produce a 1 in the corresponding entry of the K-map for S_0^+. Our options for PAID include 101 and 111, but not 100 nor 110. These latter two states have other meanings.

Let's begin by writing a next-state table consisting mostly of bits, as shown to the right. We use this table to write out a K-map for S_2^+ as follows: any of the patterns that may be used for the PAID state obey the next-state rules for PAID. Any next-state marked as PAID is marked as don't care in the K-map, since we can choose patterns starting with either or both values to represent our PAID state. The resulting K-map appears to the far right. As shown, we simply set $S_2^+ = T$, which matches our original "meaning" for S_2. That is, S_2 is the type of the last coin inserted.

		$S_2^+S_1^+S_0^+$	
state	$S_2S_1S_0$	$T=0$	$T=1$
PAID	PAID	000	100
DIME	000	001	PAID
REJECTD	001	001	PAID
QUARTER	100	PAID	110
REJECTQ	110	PAID	110

S_2^+		S_2S_1		
S_0T	00	01	11	10
00	0	0	x	x
01	x	1	1	1
11	x	1	1	1
10	0	0	0	0

Based on our choice for S_2^+, we can rewrite the K-map as shown to the right, with green italics and shading marking the values produced for the x's in the specification. Each of these boxes corresponds to one transition into the PAID state. By specifying the S_2 value, we cut the number of possible choices from four to two in each case. For those combinations in which the implementation produces $S_2^+ = 0$, we must choose $S_1^+ = 1$, but are still free to leave S_0^+ marked as a don't care. Similarly, for those combinations in which the implementation produces $S_2^+ = 1$, we must choose $S_0^+ = 1$, but are still free to leave S_1^+ marked as a don't care.

S_2^+		S_2S_1		
S_0T	00	01	11	10
00	0	0	*0*	*0*
01	*1*	1	1	1
11	*1*	1	1	1
10	0	0	0	0

The K-maps for S_1^+ and S_0^+ are shown to the right. We have not given algebraic expressions for either, but have indicated our choices by highlighting the resulting replacements of don't care entries with the values produced by our expressions. At this point, we can review the state patterns actually produced by each of the four next-state transitions into the PAID state. From the DIME state, we move into the 101 state when the user inserts a quarter. The result is the same from the REJECTD state. From the QUARTER state, however, we move into the 010 state when the user inserts a dime. The result is the same from the REJECTQ state. We must thus classify both patterns, 101 and 010, as PAID states. The remaining two patterns, 011 and 111, cannot

S_1^+		S_2S_1		
S_0T	00	01	11	10
00	0	0	1	1
01	*0*	0	1	1
11	*0*	0	0	0
10	0	0	0	0

S_0^+		S_2S_1		
S_0T	00	01	11	10
00	1	0	*0*	*0*
01	1	0	0	0
11	1	0	0	0
10	1	0	0	0

be reached from any of the states in our design. We might then try to leverage the fact that the next-state patterns from these two states are not relevant (recall that we fixed the next-state patterns for all four of the possible PAID states) to further simplify our logic, but doing so does not provide any advantage (you may want to check our claim).

		$S_2^+S_1^+S_0^+$			
state	$S_2S_1S_0$	$T=0$	$T=1$	A	P
PAID1	010	000	100	1	1
PAID2	101	000	100	1	1
DIME	000	001	101	1	0
REJECTD	001	001	101	0	0
QUARTER	100	010	110	1	0
REJECTQ	110	010	110	0	0
EXTRA1	011	000	100	x	x
EXTRA2	111	000	100	x	x

The final state table is shown to the right. We have included the extra states at the bottom of the table. We have specified the next-state logic for these states, but left the output bits as don't cares. A state transition diagram appears at the bottom of this page.

3.3.4 Testing the Design

Having a complete design on paper is a good step forward, but humans make mistakes at all stages. How can we know that a circuit that we build in the lab correctly implements the FSM that we have outlined in these notes?

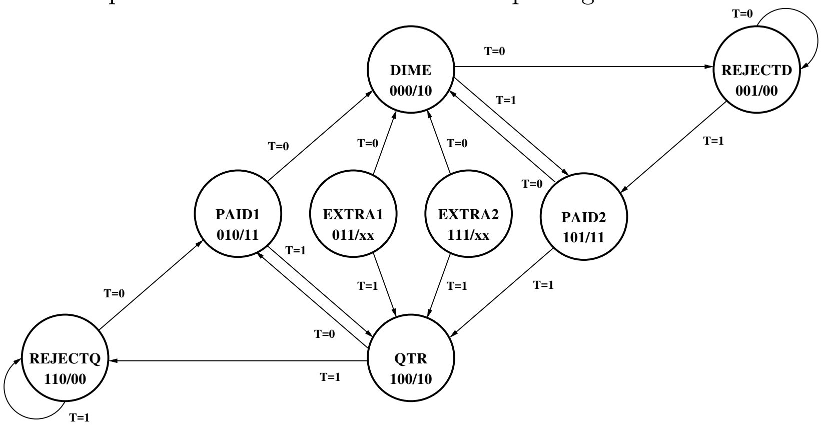
For the lab design, we have two problems to solve. First, we have not specified an initialization scheme for the FSM. We may want the FSM to start in one of the PAID states, but adding initialization logic to the design may mean requiring you to wire together significantly more chips. Second, we need a sequence of inputs that manages to test that all of the next-state and output logic implementations are correct.

Testing sequential logic, including FSMs, is in general extremely difficult. In fact, large sequential systems today are generally converted into combinational logic by using shift registers to fill the flip-flops with a particular pattern, executing the logic for one clock cycle, and checking that the resulting pattern of bits in the flip-flops is correct. This approach is called **scan-based testing**, and is discussed in ECE 543. You will make use of a similar approach when you test your combinational logic in the second week of the lab, before wiring up the flip-flops.

We have designed our FSM to be easy to test (even small FSMs may be challenging) with a brute force approach. In particular, we identify two input sequences that together serve both to initialize and to test a correctly implemented variant of our FSM. Our initialization sequence forces the FSM into a specific state regardless of its initial state. And our test sequence crosses every transition arc leaving the six valid states.

In terms of T, the coin type, we initialize the FSM with the input sequence 001. Notice that such a sequence takes any initial state into PAID2.

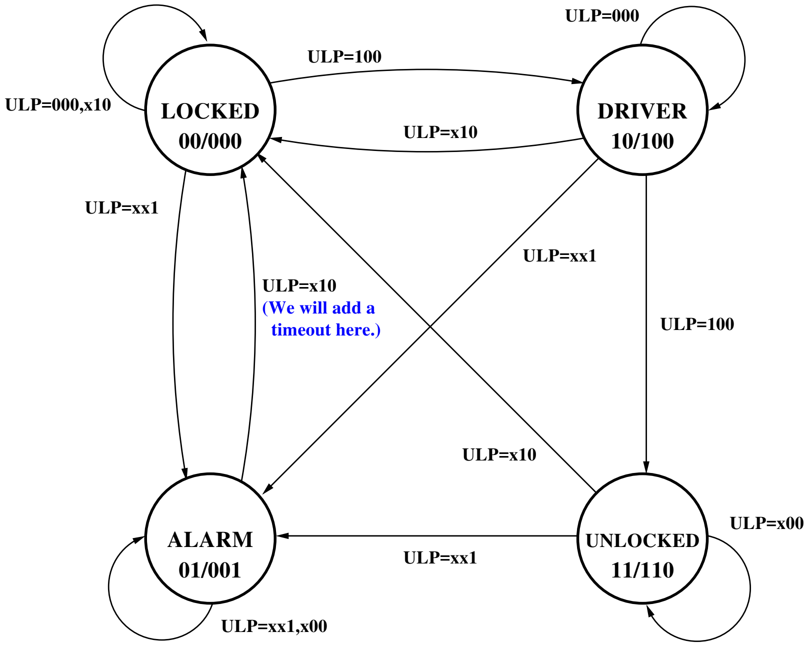
For testing, we use the input sequence 111010010001. You should trace this sequence, starting from PAID2, on the diagram below to see how the test sequence covers all of the possible arcs. As we test, we need also to observe the A and P outputs in each state to check the output logic.



ECE120: Introduction to Computer Engineering

Notes Set 3.4 Extending Keyless Entry with a Timeout

This set of notes builds on the keyless entry control FSM that we designed earlier. In particular, we use a counter to make the alarm time out, turning itself off after a fixed amount of time. The goal of this extension is to illustrate how we can make use of components such as registers and counters as building blocks for our FSMs without fully expanding the design to explicitly illustrate all possible states.



To begin, let's review the FSM that we designed earlier for keyless entry. The state transition diagram for our design is replicated to the right. The four states are labeled with state bits and output bits, S_1S_0/DRA, where D indicates that the driver's door should be unlocked, R indicates that the rest of the doors should be unlocked, and A indicates that the alarm should be on. Transition arcs in the diagram are labeled with concise versions of the inputs ULP (using don't cares), where U represents an unlock button, L represents a lock button, and P represents a panic button.

In this design, once a user presses the panic button P, the alarm sounds until the user presses the lock button L to turn it off. Instead of sounding the alarm indefinitely, we might want to turn the alarm off after a fixed amount of time. In other words, after the system has been in the ALARM state for, say, thirty or sixty seconds, we might want to move back to the LOCKED state even if the user has not pushed the lock button. The blue annotation in the diagram indicates the arc that we must adjust. But thirty or sixty seconds is a large number of clock cycles, and our FSM must keep track of the time. Do we need to draw all of the states?

Instead of following the design process that we outlined earlier, let's think about how we can modify our existing design to incorporate the new functionality. In order to keep track of time, we use a binary counter. Let's say that we want our timeout to be T cycles. When we enter the alarm state, we want to set the counter's value to $T-1$, then let the counter count down until it reaches 0, at which point a timeout occurs. To load the initial value, our counter should have a parallel load capability that sets the counter value when input $LD = 1$. When $LD = 0$, the counter counts down. The counter also has an output Z that indicates that the counter's value is currently zero, which we can use to indicate a timeout on the alarm. You should be able to build such a counter based on what you have learned earlier in the class. Here, we will assume that we can just make use of it.

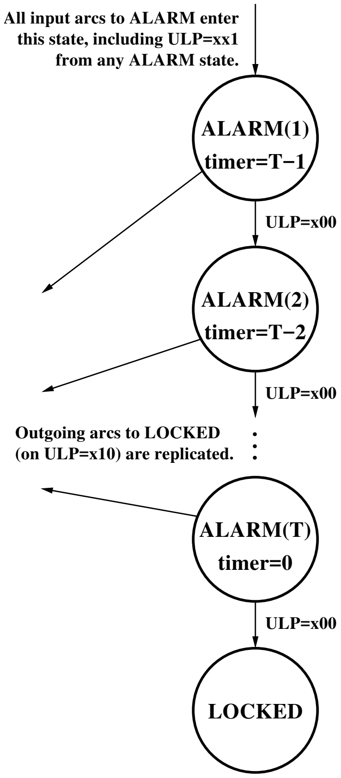
How many bits do we need in our counter? The answer depends on T. If we add the counter to our design, the state of the counter is technically part of the state of our FSM, but we can treat it somewhat abstractly. For example, we only plan to make use of the counter value in the ALARM state, so we ignore the counter bits in the three other states. In other words, $S_1S_0 = 10$ means that the system is in the LOCKED state regardless of the counter's value.

We expand the ALARM state into T separate states based on the value of the counter. As shown to the right, we name the states ALARM(1) through ALARM(T). All of these alarm states use $S_1S_0 = 01$, but they can be differentiated using a "timer" (the counter value).

We need to make design decisions about how the arcs entering and leaving the ALARM state in our original design should be used once we have incorporated the timeout. As a first step, we decide that all arcs entering ALARM from other states now enter ALARM(1). Similarly, if the user presses the panic button P in any of the ALARM(t) states, the system returns to ALARM(1). Effectively, pressing the panic button resets the timer.

The only arc leaving the ALARM state goes to the LOCKED state on $ULP = x10$. We replicate this arc for all ALARM(t) states: the user can push the lock button at any time to silence the alarm.

Finally, the self-loop back to the ALARM state on $ULP = x00$ becomes the countdown arcs in our expanded states, taking ALARM(t) to ALARM(t+1), and ALARM(T) to LOCKED.


All input arcs to ALARM enter this state, including ULP=xx1 from any ALARM state.
ALARM(1) timer=T−1
ULP=x00
ALARM(2) timer=T−2
ULP=x00
Outgoing arcs to LOCKED (on ULP=x10) are replicated.
ALARM(T) timer=0
ULP=x00
LOCKED

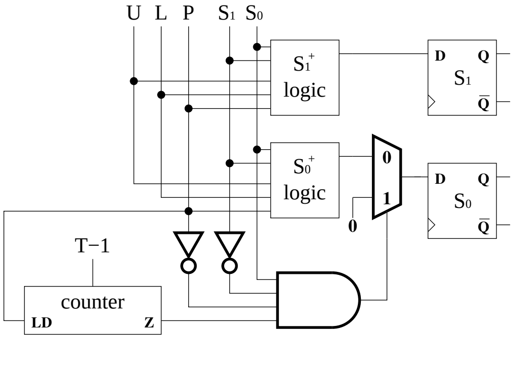
Now that we have a complete specification for the extended design, we can implement it. We want to reuse our original design as much as possible, but we have three new features that must be considered. First, when we enter the ALARM(1) state, we need to set the counter value to $T - 1$. Second, we need the counter value to count downward while in the ALARM state. Finally, we need to move back to the LOCKED state when a timeout occurs—that is, when the counter reaches zero.

The first problem is fairly easy. Our counter supports parallel load, and the only value that we need to load is $T - 1$, so we apply the constant bit pattern for $T - 1$ to the load inputs and raise the LD input whenever we enter the ALARM(1) state. In our original design, we chose to enter the ALARM state whenever the user pressed P, regardless of the other buttons. Hence we can connect P directly to our counter's LD input.

The second problem is handled by the counter's countdown functionality. In the ALARM(t) states, the counter will count down each cycle, moving the system from ALARM(t) to ALARM(t+1).

The last problem is slightly trickier, since we need to change S_1S_0. Notice that $S_1S_0 = 01$ for the ALARM state and $S_1S_0 = 00$ for the LOCKED state. Thus, we need only force S_0 to 0 when a timeout occurs. We can use a single 2-to-1 multiplexer for this purpose. The "0" input of the mux comes from the original S_0^+ logic, and the "1" input is a constant 0. All other state logic remains unchanged. When does a timeout occur? First, we must be in the ALARM(T) state, so $S_1S_0 = 01$ and the counter's Z output is raised. Second, the input combination must be $ULP = xx0$—notice that both $ULP = x00$ and $ULP = x10$ return to LOCKED from ALARM(T). A single, four-input AND gate thus suffices to obtain the timeout signal, $\bar{S_1}S_0Z\bar{P}$, which we connect to the select input of the mux between the S_0^+ logic and the S_0 flip-flop.

The extension thus requires only a counter, a mux, and a gate, as shown below.



ECE120: Introduction to Computer Engineering

Notes Set 3.5 Finite State Machine Design Examples, Part II

This set of notes provides several additional examples of FSM design. We first design an FSM to control a vending machine, introducing encoders and decoders as components that help us to implement our design. We then design a game controller for a logic puzzle implemented as a children's game. Finally, we analyze a digital FSM designed to control the stoplights at the intersection of two roads.

3.5.1 Design of a Vending Machine

For the next example, we design an FSM to control a simple vending machine. The machine accepts U.S. coins[11] as payment and offers a choice of three items for sale.

What states does such an FSM need? The FSM needs to keep track of how much money has been inserted in order to decide whether a user can purchase one of the items. That information alone is enough for the simplest machine, but let's create a machine with adjustable item prices. We can use registers to hold the item prices, which we denote P_1, P_2, and P_3.

Technically, the item prices are also part of the internal state of the FSM. However, we leave out discussion (and, indeed, methods) for setting the item prices, so no state with a given combination of prices has any transition to a state with a different set of item prices. In other words, any given combination of item prices induces a subset of states that operate independently of the subset induced by a distinct combination of item prices. By abstracting away the prices in this way, we can focus on a general design that allows the owner of the machine to set the prices dynamically.

coin type	value	# of nickels
nickel	$0.05	1
dime	$0.10	2
quarter	$0.25	5
half dollar	$0.50	10
dollar	$1.00	20

Our machine will not accept pennies, so let's have the FSM keep track of how much money has been inserted as a multiple of 5 cents (one nickel). The table to the right shows five types of coins, their value in dollars, and their value in terms of nickels.

The most expensive item in the machine might cost a dollar or two, so the FSM must track at least 20 or 40 nickels of value. Let's decide to use six bits to record the number of nickels, which allows the machine to keep track of up to $3.15 (63 nickels). We call the abstract states **STATE00** through **STATE63**, and refer to a state with an inserted value of N nickels as **STATE**<N>.

Let's now create a next-state table, as shown at the top of the next page. The user can insert one of the five coin types, or can pick one of the three items. What should happen if the user inserts more money than the FSM can track? Let's make the FSM reject such coins. Similarly, if the user tries to buy an item without inserting enough money first, the FSM must reject the request. For each of the possible input events, we add a condition to separate the FSM states that allow the input event to be processed as the user desires from those states that do not. For example, if the user inserts a quarter, those states with $N < 59$ transition to states with value $N + 5$ and accept the quarter. Those states with $N \geq 59$ reject the coin and remain in **STATE**<N>.

[11] Most countries have small bills or coins in demoninations suitable for vending machine prices, so think about some other currency if you prefer.

			final state		
				accept	release
initial state	input event	condition	state	coin	product
STATE<N>	no input	always	**STATE**<N>	—	none
STATE<N>	nickel inserted	$N < 63$	**STATE**<$N+1$>	yes	none
STATE<N>	nickel inserted	$N = 63$	**STATE**<N>	no	none
STATE<N>	dime inserted	$N < 62$	**STATE**<$N+2$>	yes	none
STATE<N>	dime inserted	$N \geq 62$	**STATE**<N>	no	none
STATE<N>	quarter inserted	$N < 59$	**STATE**<$N+5$>	yes	none
STATE<N>	quarter inserted	$N \geq 59$	**STATE**<N>	no	none
STATE<N>	half dollar inserted	$N < 54$	**STATE**<$N+10$>	yes	none
STATE<N>	half dollar inserted	$N \geq 54$	**STATE**<N>	no	none
STATE<N>	dollar inserted	$N < 44$	**STATE**<$N+20$>	yes	none
STATE<N>	dollar inserted	$N \geq 44$	**STATE**<N>	no	none
STATE<N>	item 1 selected	$N \geq P_1$	**STATE**<$N-P_1$>	—	1
STATE<N>	item 1 selected	$N < P_1$	**STATE**<N>	—	none
STATE<N>	item 2 selected	$N \geq P_2$	**STATE**<$N-P_2$>	—	2
STATE<N>	item 2 selected	$N < P_2$	**STATE**<N>	—	none
STATE<N>	item 3 selected	$N \geq P_3$	**STATE**<$N-P_3$>	—	3
STATE<N>	item 3 selected	$N < P_3$	**STATE**<N>	—	none

We can now begin to formalize the I/O for our machine. Inputs include insertion of coins and selection of items for purchase. Outputs include a signal to accept or reject an inserted coin as well as signals to release each of the three items.

coin type	$C_2C_1C_0$
none	110
nickel	010
dime	000
quarter	011
half dollar	001
dollar	111

For input to the FSM, we assume that a coin inserted in any given cycle is classified and delivered to our FSM using the three-bit representation shown to the right. For item selection, we assume that the user has access to three buttons, B_1, B_2, and B_3, that indicate a desire to purchase the corresponding item.

For output, the FSM must produce a signal A indicating whether a coin should be accepted. To control the release of items that have been purchased, the FSM must produce the signals R_1, R_2, and R_3, corresponding to the release of each item. Since outputs in our class depend only on state, we extend the internal state of the FSM to include bits for each of these output signals. The output signals go high in the cycle after the inputs that generate them. Thus, for example, the accept signal A corresponds to a coin inserted in the previous cycle, even if a second coin is inserted in the current cycle. This meaning must be made clear to whomever builds the mechanical system to return coins.

Now we are ready to complete the specification. How many states does the FSM have? With six bits to record money inserted and four bits to drive output signals, we have a total of 1,024 (2^{10}) states! Six different coin inputs are possible, and the selection buttons allow eight possible combinations, giving 48 transitions from each state. Fortunately, we can use the meaning of the bits to greatly simplify our analysis.

First, note that the current state of the coin accept bit and item release bits—the four bits of FSM state that control the outputs—have no effect on the next state of the FSM. Thus, we can consider only the current amount of money in a given state when thinking about the transitions from the state. As you have seen, we can further abstract the states using the number N, the number of nickels currently held by the vending machine.

We must still consider all 48 possible transitions from **STATE**<N>. Looking back at our abstract next-state table, notice that we had only eight types of input events (not counting "no input"). If we strictly prioritize these eight possible events, we can safely ignore combinations. Recall that we adopted a similar strategy for several earlier designs, including the ice cream dispenser in Notes Set 2.2 and the keyless entry system developed in Notes Set 3.1.3.

We choose to prioritize purchases over new coin insertions, and to prioritize item 3 over item 2 over item 1. These prioritizations are strict in the sense that if the user presses B_3, both other buttons are ignored, and any coin inserted is rejected, regardless of whether or not the user can actually purchase item 3 (the machine may not contain enough money to cover the item price). With the choice of strict prioritization, all transitions from all states become well-defined. We apply the transition rules in order of decreasing priority, with conditions, and with don't-cares for lower-priority inputs. For example, for any of the 16 **STATE50**'s (remember that the four current output bits do not affect transitions), the table below lists all possible transitions assuming that $P_3 = 60$, $P_2 = 10$, and $P_1 = 35$.

					next state				
initial state	B_3	B_2	B_1	$C_2C_1C_0$	state	A	R_3	R_2	R_1
STATE50	1	x	x	xxx	**STATE50**	0	0	0	0
STATE50	0	1	x	xxx	**STATE40**	0	0	1	0
STATE50	0	0	1	xxx	**STATE15**	0	0	0	1
STATE50	0	0	0	010	**STATE51**	1	0	0	0
STATE50	0	0	0	000	**STATE52**	1	0	0	0
STATE50	0	0	0	011	**STATE55**	1	0	0	0
STATE50	0	0	0	001	**STATE60**	1	0	0	0
STATE50	0	0	0	111	**STATE50**	0	0	0	0
STATE50	0	0	0	110	**STATE50**	0	0	0	0

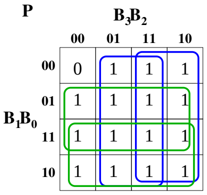
Next, we need to choose a state representation. But this task is essentially done: each output bit (A, R_1, R_2, and R_3) is represented with one bit in the internal representation, and the remaining six bits record the number of nickels held by the vending machine using an unsigned representation.

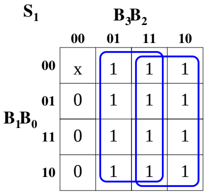
The choice of a numeric representation for the money held is important, as it allows us to use an adder to compute the money held in the next state.

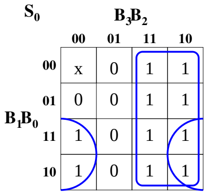
3.5.2 Encoders and Decoders

Since we chose to prioritize purchases, let's begin by building logic to perform state transitions for purchases. Our first task is to implement prioritization among the three selection buttons. For this purpose, we construct a 4-input **priority encoder**, which generates a signal P whenever any of its four input lines is active and encodes the index of the highest active input as a two-bit unsigned number S. A truth table for our priority encoder appears on the left below, with K-maps for each of the output bits on the right.

B_3	B_2	B_1	B_0	P	S
1	x	x	x	1	11
0	1	x	x	1	10
0	0	1	x	1	01
0	0	0	1	1	00
0	0	0	0	0	xx





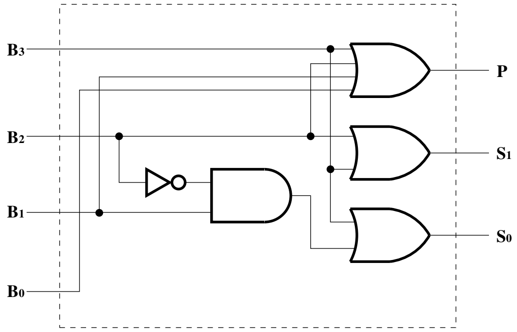


From the K-maps, we extract the following equations:

$$\begin{aligned} P &= B_3 + B_2 + B_1 + B_0 \\ S_1 &= B_3 + B_2 \\ S_0 &= B_3 + \overline{B_2} B_1 \end{aligned}$$

which allow us to implement our encoder as shown to the right.

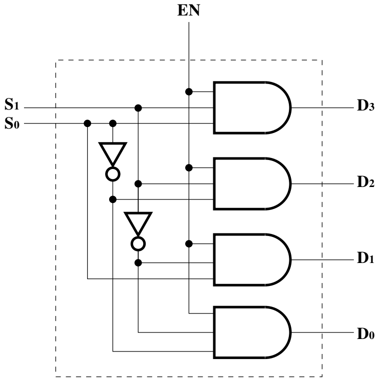
If we connect our buttons B_1, B_2, and B_3 to the priority encoder (and feed 0 into the fourth input), it produces a signal P indicating that the user is trying to make a purchase and a two-bit signal S indicating which item the user wants.



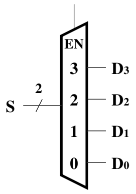
a 4−input priority encoder
(logic diagram)

We also need to build logic to control the item release outputs R_1, R_2, and R_3. An item should be released only when it has been selected (as indicated by the priority encoder signal S) and the vending machine has enough money. For now, let's leave aside calculation of the item release signal, which we call R, and focus on how we can produce the correct values of R_1, R_2, and R_3 from S and R.

The component to the right is a **decoder** with an enable input. A decoder takes an input signal—typically one coded as a binary number—and produces one output for each possible value of the signal. You may notice the similarity with the structure of a mux: when the decoder is enabled ($EN = 1$), each of the AND gates produces one minterm of the input signal S. In the mux, each of the inputs is then included in one minterm's AND gate, and the outputs of all AND gates are ORd together. In the decoder, the AND gate outputs are the outputs of the decoder. Thus, when enabled, the decoder produces exactly one 1 bit on its outputs. When not enabled ($EN = 0$), the decoder produces all 0 bits.



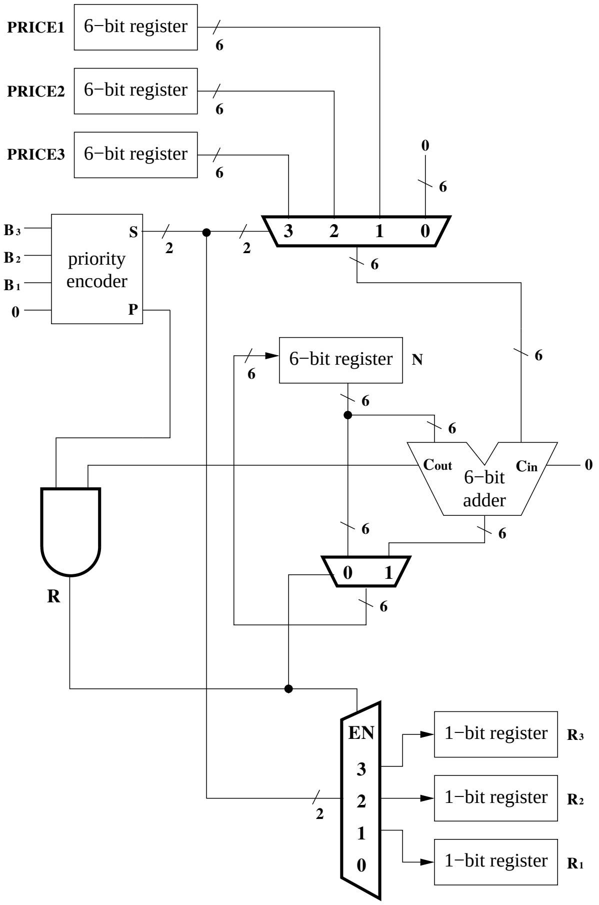
a 2−to−4 decoder with enable
(logic diagram)



a 2−to−4 decoder
with enable
(symbolic form)

We use a decoder to generate the release signals for the vending machine by connecting the signal S produced by the priority encoder to the decoder's S input and connecting the item release signal R to the decoder's EN input. The outputs D_1, D_2, and D_3 then correspond to the individual item release signals R_1, R_2, and R_3 for our vending machine.

3.5.3 Vending Machine Implementation



We are now ready to implement the FSM to handle purchases, as shown to the right. The current number of nickels, N, is stored in a register in the center of the diagram. Each cycle, N is fed into a 6-bit adder, which subtracts the price of any purchase requested in that cycle. Recall that we chose to record item prices in registers. We avoid the need to negate prices before adding them by storing the negated prices in our registers. Thus, the value of register PRICE1 is $-P_1$, the the value of register PRICE2 is $-P_2$, and the the value of register PRICE3 is $-P_3$. The priority encoder's S signal is then used to select the value of one of these three registers (using a 24-to-6 mux) as the second input to the adder.

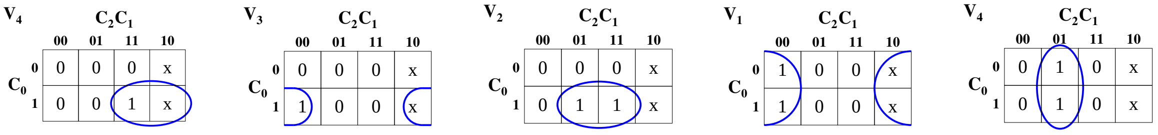
We use the adder to execute a subtraction, so the carry out C_{out} is 1 whenever the value of N is at least as great as the amount being subtracted. In that case, the purchase is successful. The AND gate on the left calculates the signal R indicating a successful purchase, which is then used to select the next value of N using the 12-to-6 mux below the adder. When no item selection buttons are pushed, P and thus R are both 0, and the mux below the adder keeps N unchanged in the next cycle. Similarly, if $P = 1$ but N is insufficient, C_{out} and thus R are both 0, and again N does not change. Only when $P = 1$ and $C_{out} = 1$ is the purchase successful, in which case the price is subtracted from N in the next cycle.

The signal R is also used to enable a decoder that generates the three individual item release outputs. The correct output is generated based on the decoded S signal from the priority encoder, and all three output bits are latched into registers to release the purchased item in the next cycle.

One minor note on the design so far: by hardwiring C_{in} to 0, we created a problem for items that cost nothing (0 nickels): in that case, C_{out} is always 0. We could instead store $-P_1 - 1$ in PRICE1 (and so forth) and feed P in to C_{in}, but maybe it's better not to allow free items.

How can we support coin insertion? Let's use the same adder to add each inserted coin's value to N. The table at the right shows the value of each coin as a 5-bit unsigned number of nickels. Using this table, we can fill in K-maps for each bit of V, as shown below. Notice that we have marked the two undefined bit patterns for the coin type C as don't cares in the K-maps.

coin type	$C_2C_1C_0$	$V_4V_3V_2V_1V_0$
none	110	00000
nickel	010	00001
dime	000	00010
quarter	011	00101
half dollar	001	01010
dollar	111	10100



V_4

C_0 \ C_2C_1	00	01	11	10
0	0	0	0	x
1	0	0	1	x

V_3

C_0 \ C_2C_1	00	01	11	10
0	0	0	0	x
1	1	0	0	x

V_2

C_0 \ C_2C_1	00	01	11	10
0	0	0	0	x
1	0	1	1	x

V_1

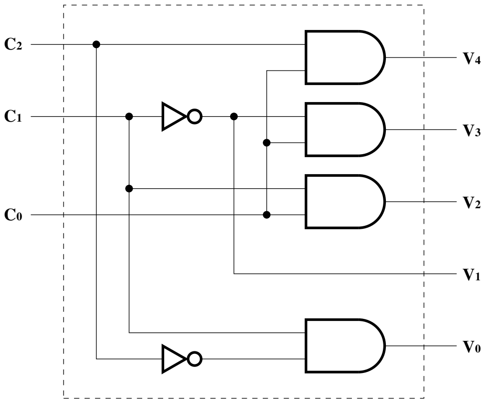
C_0 \ C_2C_1	00	01	11	10
0	1	0	0	x
1	1	0	0	x

V_4

C_0 \ C_2C_1	00	01	11	10
0	0	1	0	x
1	0	1	0	x

Solving the K-maps gives the following equations, which we implement as shown to the right.

$$\begin{aligned} V_4 &= C_2C_0 \\ V_3 &= \overline{C_1}C_0 \\ V_2 &= C_1C_0 \\ V_1 &= \overline{C_1} \\ V_0 &= \overline{C_2}C_1 \end{aligned}$$

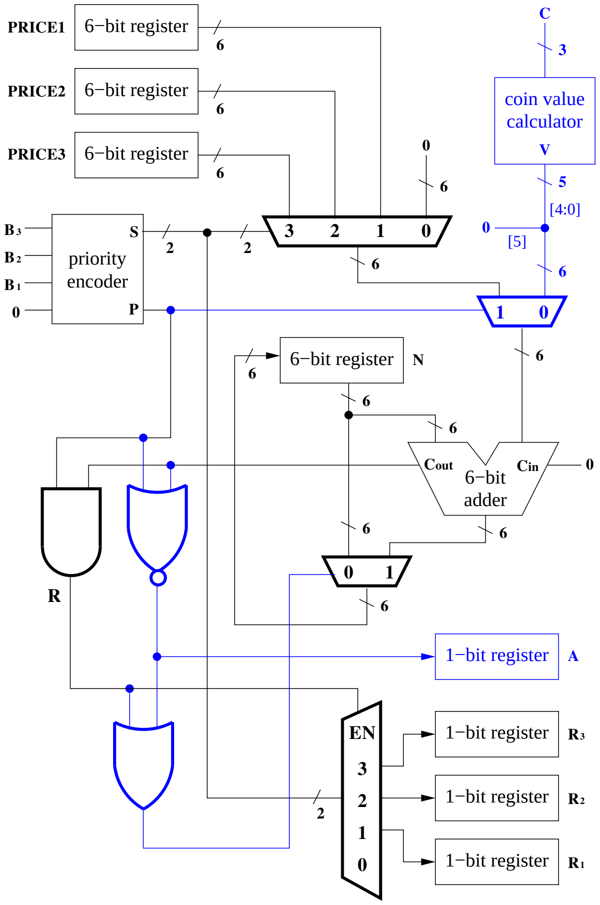


coin value calculator

Now we can extend the design to handle coin insertion, as shown to the right with new elements highlighted in blue. The output of the coin value calculator is extended with a leading 0 and then fed into a 12-to-6 mux. When a purchase is requested, $P = 1$ and the mux forwards the item price to the adder—recall that we chose to give purchases priority over coin insertion. When no purchase is requested, the value of any coin inserted (or 0 when no coin is inserted) is passed to the adder.

Two new gates have been added on the lower left. First, let's verify that purchases work as before. When a purchase is requested, $P = 1$, so the NOR gate outputs 0, and the OR gate simple forwards R to control the mux that decides whether the purchase was successful, just as in our original design.

When no purchase is made ($P = 0$, and $R = 0$), the adder adds the value of any inserted coin to N. If the addition overflows, $C_{out} = 1$, and the output of the NOR gate is 0. Note that the NOR gate output is stored as the output A in the next cycle, so a coin that causes overflow in the amount of money stored is rejected. The OR gate also outputs 0, and N remains unchanged. If the addition does not overflow, the NOR gate outputs a 1, the coin is accepted ($A = 1$ in the next cycle), and the mux allows the sum $N+V$ to be written back as the new value of N.



The tables at the top of the next page define all of the state variables, inputs, outputs, and internal signals used in the design, and list the number of bits for each variable.

		FSM state
PRICE1	6	negated price of item 1 ($-P_1$)
PRICE2	6	negated price of item 2 ($-P_2$)
PRICE3	6	negated price of item 3 ($-P_3$)
N	6	value of money in machine (in nickels)
A	1	stored value of accept coin output
R_1	1	stored value of release item 1 output
R_2	1	stored value of release item 2 output
R_3	1	stored value of release item 3 output
		internal signals
V	5	inserted coin value in nickels
P	1	purchase requested (from priority encoder)
S	2	item # requested (from priority encoder)
R	1	release item (purchase approved)

		inputs
B_1	1	item 1 selected for purchase
B_2	1	item 2 selected for purchase
B_3	1	item 3 selected for purchase
C	3	coin inserted (see earlier table for meaning)
		outputs
A	1	accept inserted coin (last cycle)
R_1	1	release item 1
R_2	1	release item 2
R_3	1	release item 3
Note that outputs correspond one-to-one with four bits of FSM state.		

3.5.4 Design of a Game Controller

For the next example, imagine that you are part of a team building a game for children to play at Engineering Open House. The game revolves around an old logic problem in which a farmer must cross a river in order to reach the market. The farmer is traveling to the market to sell a fox, a goose, and some corn. The farmer has a boat, but the boat is only large enough to carry the fox, the goose, or the corn along with the farmer. The farmer knows that if he leaves the fox alone with the goose, the fox will eat the goose. Similarly, if the farmer leaves the goose alone with the corn, the goose will eat the corn. How can the farmer cross the river?

Your team decides to build a board illustrating the problem with a river from top to bottom and lights illustrating the positions of the farmer (always with the boat), the fox, the goose, and the corn on either the left bank or the right bank of the river. Everything starts on the left bank, and the children can play the game until they win by getting everything to the right bank or until they make a mistake. As the ECE major on your team, you get to design the FSM!

Since the four entities (farmer, fox, goose, and corn) can be only on one bank or the other, we can use one bit to represent the location of each entity. Rather than giving the states names, let's just call a state $FXGC$. The value of F represents the location of the farmer, either on the left bank ($F = 0$) or the right bank ($F = 1$). Using the same representation (0 for the left bank, 1 for the right bank), the value of X represents the location of the fox, G represents the location of the goose, and C represents the location of the corn.

We can now put together an abstract next-state table, as shown to the right. Once the player wins or loses, let's have the game indicate their final status and stop accepting requests to have the farmer cross the river. We can use a reset button to force the game back into the original state for the next player.

initial state	input event	condition	final state
$FXGC$	no input	always	$FXGC$
$FXGC$	reset	always	0000
$FXGC$	cross alone	always	$\bar{F}XGC$
$FXGC$	cross with fox	$F = X$	$\bar{F}\bar{X}GC$
$FXGC$	cross with goose	$F = G$	$\bar{F}X\bar{G}C$
$FXGC$	cross with corn	$F = C$	$\bar{F}XG\bar{C}$

Note that we have included conditions for some of the input events, as we did previously with the vending machine design. The conditions here require that the farmer be on the same bank as any entity that the player wants the farmer to carry across the river.

Next, we specify the I/O interface. For input, the game has five buttons. A reset button R forces the FSM back into the initial state. The other four buttons cause the farmer to cross the river: B_F crosses alone, B_X with the fox, B_G with the goose, and B_C with the corn.

For output, we need position indicators for the four entities, but let's assume that we can simply output the current state $FXGC$ and have appropriate images or lights appear on the correct banks of the river. We also need two more indicators: W for reaching the winning state, and L for reaching a losing state.

Now we are ready to complete the specification. We could use a strict prioritization of input events, as we did with earlier examples. Instead, in order to vary the designs a bit, we use a strict prioritization among allowed inputs. The reset button R has the highest priority, followed by B_F, B_C, B_G, and finally B_X. However, only those buttons that result in an allowed move are considered when selecting one button among several pressed in a single clock cycle.

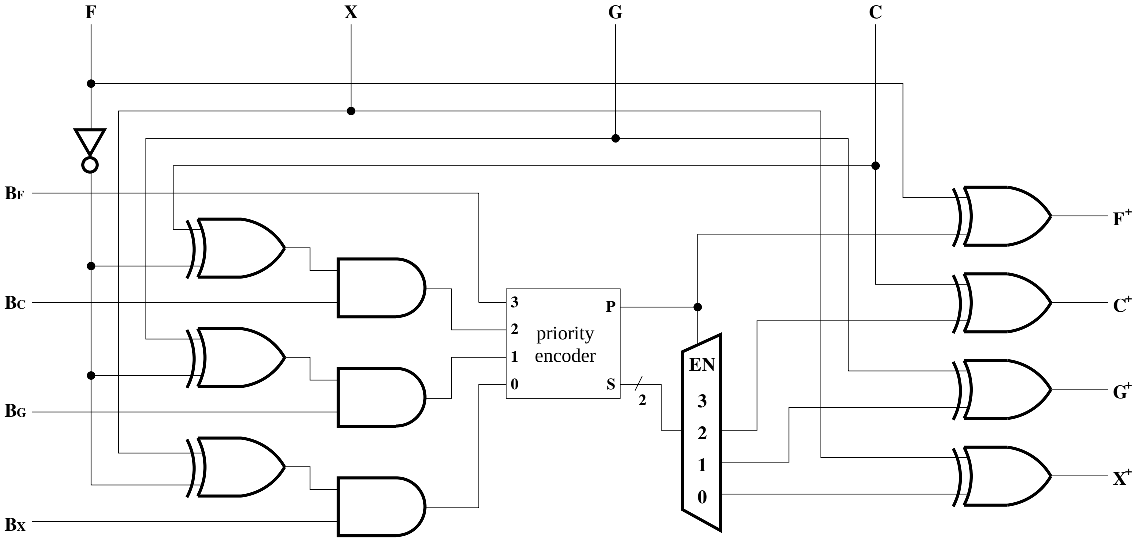
As an example, consider the state $FXGC = 0101$. The farmer is not on the same bank as the fox, nor as the corn, so the B_X and B_C buttons are ignored, leading to the next-state table to the right. Notice that B_G is accepted even if B_C is pressed because the farmer is not on the same bank as the corn. As shown later, this approach to prioritization of inputs is equally simple in terms of implementation.

$FXGC$	R	B_F	B_X	B_G	B_C	$F^+X^+G^+C^+$
0101	1	x	x	x	x	0000
0101	0	1	x	x	x	1101
0101	0	0	x	1	x	1111
0101	0	0	x	0	x	0101

Recall that we want to stop the game when the player wins or loses. In these states, only the reset button is accepted. For example, the state $FXGC = 0110$ is a losing state because the farmer has left the fox with the goose on the opposite side of the river. In this case, the player can reset the game, but other buttons are ignored.

$FXGC$	R	B_F	B_X	B_G	B_C	$F^+X^+G^+C^+$
0110	1	x	x	x	x	0000
0110	0	x	x	x	x	0110

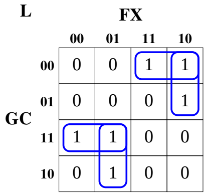
As we have already chosen a representation, we now move on to implement the FSM. Let's begin by calculating the next state ignoring the reset button and the winning and losing states, as shown in the logic diagram below.



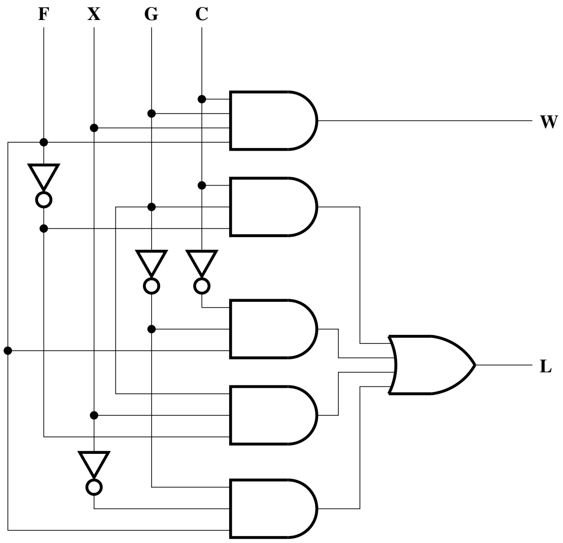
The left column of XOR gates determines whether the farmer is on the same bank as the corn (top gate), goose (middle gate), and fox (bottom gate). The output of each gate is then used to mask out the corresponding button: only when the farmer is on the same bank are these buttons considered. The adjusted button values are then fed into a priority encoder, which selects the highest priority input event according to the scheme that we outlined earlier (from highest to lowest, B_F, B_C, B_G, and B_X, ignoring the reset button).

The output of the priority encoder is then used to drive another column of XOR gates on the right in order to calculate the next state. If any of the allowed buttons is pressed, the priority encoder outputs $P = 1$, and the farmer's bank is changed. If B_C is allowed and selected by the priority encoder (only when B_F is not pressed), both the farmer and the corn's banks are flipped. The goose and the fox are handled in the same way.

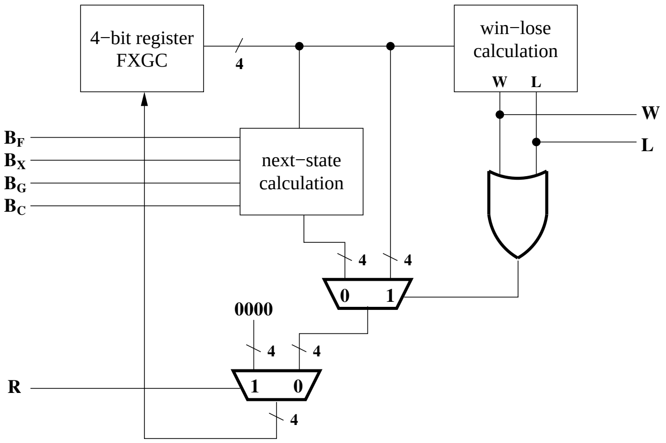
Next, let's build a component to produce the win and lose signals. The one winning state is $FXGC = 1111$, so we simply need an AND gate. For the lose signal L, we can fill in a K-map and derive an expression, as shown to the right, then implement as shown in the logic diagram to the far right. For the K-map, remember that the player loses whenever the fox and the goose are on the same side of the river, but opposite from the farmer, or whenever the goose and the corn are on the same side of the river, but opposite from the farmer.



$$L = F\bar{X}\bar{G} + \bar{F}XG + F\bar{G}\bar{C} + \bar{F}GC$$



Finally, we complete our design by integrating the next-state calculation and the win-lose calculation with a couple of muxes, as shown to the right. The lower mux controls the final value of the next state: note that it selects between the constant state $FXGC = 0000$ when the reset button R is pressed and the output of the upper mux when $R = 0$. The upper mux is controlled by $W + L$, and retains the current state whenever either signal is 1. In other words, once the player has won or lost, the upper mux prevents further state changes until the reset button is pressed. When R, W, and L are all 0, the next state is calculated according to whatever buttons have been pressed.



3.5.5 Analysis of a Stoplight Controller

In this example, we begin with a digital FSM design and analyze it to understand how it works and to verify that its behavior is appropriate. The FSM that we analyze has been designed to control the stoplights at the intersection of two roads. For naming purposes, we assume that one of the roads runs East and West (EW), and the second runs North and South (NS).

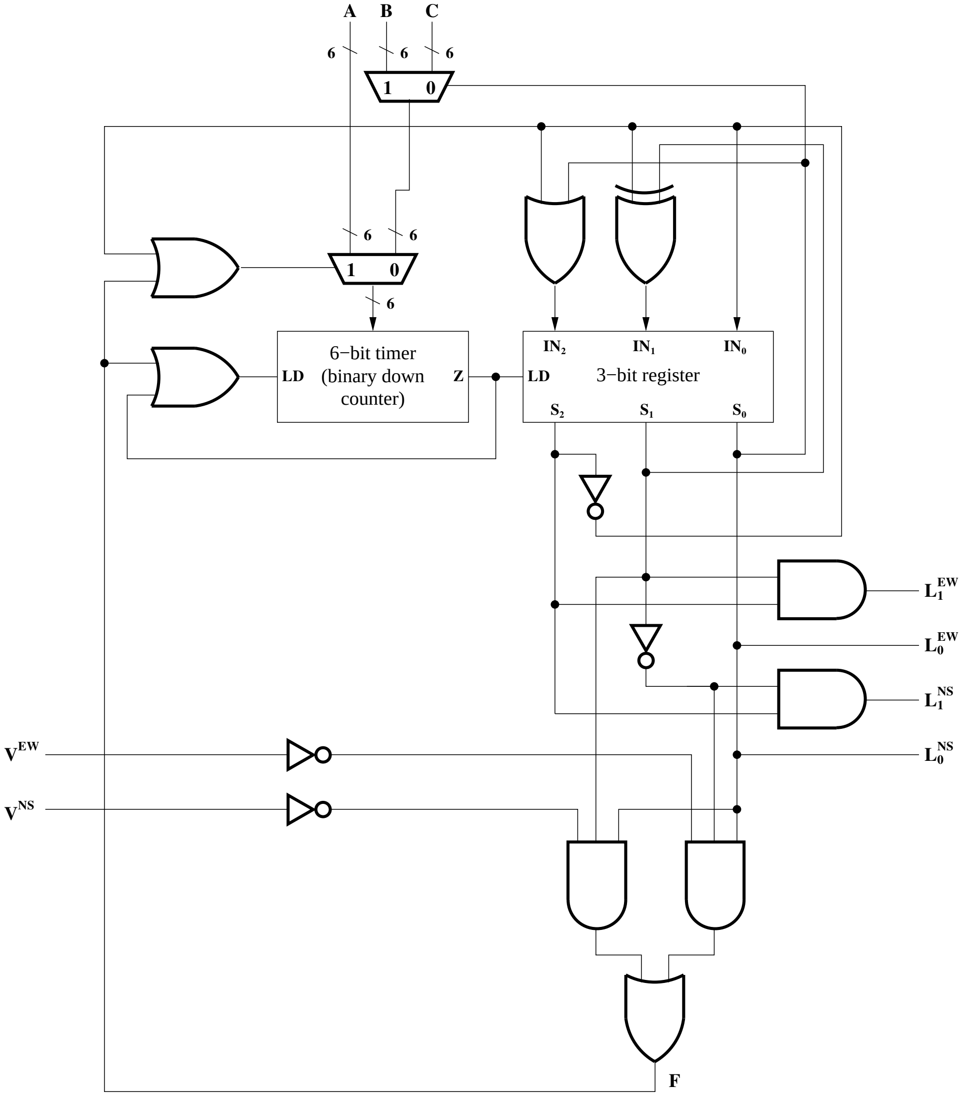
The stoplight controller has two inputs, each of which senses vehicles approaching from either direction on one of the two roads. The input $V^{EW} = 1$ when a vehicle approaches from either the East or the West, and the input $V^{NS} = 1$ when a vehicle approaches from either the North or the South. These inputs are also active when vehicles are stopped waiting at the corresponding lights. Another three inputs, A, B, and C, control the timing behavior of the system; we do not discuss them here except as variables.

The outputs of the controller consist of two 2-bit values, L^{EW} and L^{NS}, that specify the light colors for the two roads. In particular, L^{EW} controls the lights facing East and West, and L^{NS} controls the lights facing North and South. The meaning of these outputs is given in the table to the right.

L	light color
0x	red
10	yellow
11	green

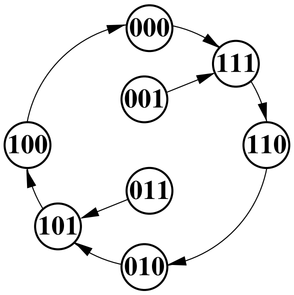
Let's think about the basic operation of the controller. For safety reasons, the controller must ensure that the lights on one or both roads are red at all times. Similarly, if a road has a green light, the controller should show a yellow light before showing a red light to give drivers some warning and allow them to slow down. Finally, for fairness, the controller should alternate green lights between the two roads.

Now take a look at the logic diagram below. The state of the FSM has been split into two pieces: a 3-bit register S and a 6-bit timer. The timer is simply a binary counter that counts downward and produces an output of $Z = 1$ when it reaches 0. Notice that the register S only takes a new value when the timer reaches 0, and that the Z signal from the timer also forces a new value to be loaded into the timer in the next cycle. We can thus think of transitions in the FSM on a cycle by cycle basis as consisting of two types. The first type simply counts downward for a number of cycles while holding the register S constant, while the second changes the value of S and sets the timer in order to maintain the new value of S for some number of cycles.



Let's look at the next-state logic for S, which feeds into the IN inputs on the 3-bit register ($S_2^+ = IN_2$ and so forth). Notice that none of the inputs to the FSM directly affect these values. The states of S thus act like a counter. By examining the connections, we can derive equations for the next state and draw a transition diagram, as shown to the right. As the figure shows, there are six states in the loop defined by the next-state logic, with the two remaining states converging into the loop after a single cycle.

$$\begin{aligned} S_2^+ &= \overline{S_2} + S_0 \\ S_1^+ &= \overline{S_2} \oplus S_1 \\ S_0^+ &= \overline{S_2} \end{aligned}$$



Let's now examine the outputs for each state in order to understand how the stoplight sequencing works. We derive equations for the outputs that control the lights, as shown to the right, then calculate values and colors for each state, as shown to the far right. For completeness, the table includes the states outside of the desired loop. The lights are all red in both of these states, which is necessary for safety.

$$\begin{aligned} L_1^{EW} &= S_2 S_1 \\ L_0^{EW} &= S_0 \\ L_1^{NS} &= S_2 \overline{S_1} \\ L_0^{NS} &= S_0 \end{aligned}$$

			EW	NS
			light	light
S	L^{EW}	L^{NS}	color	color
000	00	00	red	red
111	11	01	green	red
110	10	00	yellow	red
010	00	00	red	red
101	01	11	red	green
100	00	10	red	yellow
001	01	01	red	red
011	01	01	red	red

Now let's think about how the timer works. As we already noted, the timer value is set whenever S enters a new state, but it can also be set under other conditions—in particular, by the signal F calculated at the bottom of the FSM logic diagram.

For now, assume that $F = 0$. In this case, the timer is set only when the state S changes, and we can find the duration of each state by analyzing the muxes. The bottom mux selects A when $S_2 = 0$, and selects the output of the top mux when $S_2 = 1$. The top mux selects B when $S_0 = 1$, and selects C when $S_0 = 0$. Combining these results, we can calculate the duration of the next states of S when $F = 0$, as shown in the table to the right. We can then combine the next state duration with our previous calculation of the state sequencing (also the order in the table) to obtain the durations of each state, also shown in the rightmost column of the table.

	EW	NS	next	current
	light	light	state	state
S	color	color	duration	duration
000	red	red	A	C
111	green	red	B	A
110	yellow	red	C	B
010	red	red	A	C
101	red	green	B	A
100	red	yellow	C	B
001	red	red	A	—
011	red	red	A	—

What does F do? Analyzing the gates that produce it gives $F = S_1 S_0 \overline{V^{NS}} + \overline{S_1} S_0 \overline{V^{EW}}$. If we ignore the two states outside of the main loop for S, the first term is 1 only when the lights are green on the East and West roads and the detector for the North and South roads indicates that no vehicles are approaching. Similarly, the second term is 1 only when the lights are green on the North and South roads and the detector for the East and West roads indicates that no vehicles are approaching.

What happens when $F = 1$? First, the OR gate feeding into the timer's LD input produces a 1, meaning that the timer loads a new value instead of counting down. Second, the OR gate controlling the lower mux selects the A input. In other words, the timer is reset to A cycles, corresponding to the initial value for the green light states. In other words, the light stays green until vehicles approach on the other road, plus A more cycles.

Unfortunately, the signal F may also be 1 in the unused states of S, in which case the lights on both roads may remain red even though cars are waiting on one of the roads. To avoid this behavior, we must be sure to initialize the state S to one of the six states in the desired loop.

ECE120: Introduction to Computer Engineering

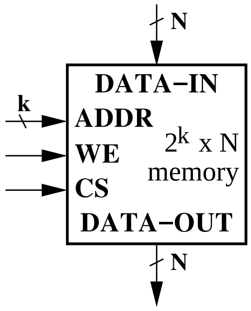
Notes Set 3.6 Random Access Memories

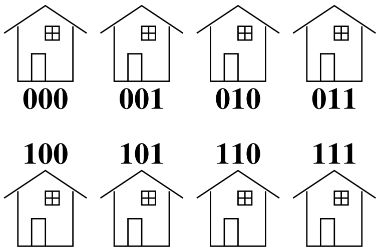
This set of notes describes random access memories (RAMs), providing slightly more detail than is available in the textbook. We begin with a discussion of the memory abstraction and the types of memory most commonly used in digital systems, then examine how one can build memories (static RAMs) using logic. We next introduce tri-state buffers as a way of simplifying ouput connections, and illustrate how memory chips can be combined to provide larger and wider memories. A more detailed description of dynamic RAMs finishes this set. *Sections marked with an asterisk are provided solely for your interest, but you probably need to learn this material in later classes.*

3.6.1 Memory

A computer **memory** is a group of storage elements and the logic necessary to move data in and out of the elements. The size of the elements in a memory—called the **addressability** of the memory—varies from a single binary digit, or **bit**, to a **byte** (8 bits) or more. Typically, we refer to data elements larger than a byte as **words**, but the size of a word depends on context.

Each element in a memory is assigned a unique name, called an **address**, that allows an external circuit to identify the particular element of interest. These addresses are not unlike the street addresses that you use when you send a letter. Unlike street addresses, however, memory addresses usually have little or no redundancy; each possible combination of bits in an address identifies a distinct set of bits in the memory. The figure on the right below illustrates the concept. Each house represents a storage element and is associated with a unique address.





The memories that we consider in this class have several properties in common. These memories support two operations: **write** places a word of data into an element, and **read** retrieves a copy of a word of data from an element. The memories are also **volatile**, which means that the data held by a memory are erased when electrical power is turned off or fails. **Non-volatile** forms of memory include magnetic and optical storage media such as DVDs, CD-ROMs, disks, and tapes, capacitive storage media such as Flash drives, and some programmable logic devices. Finally, the memories considered in this class are **random access memories (RAMs)**, which means that the time required to access an element in the memory is independent of the element being accessed. In contrast, **serial memories** such as magnetic tape require much less time to access data near the current location in the tape than data far away from the current location.

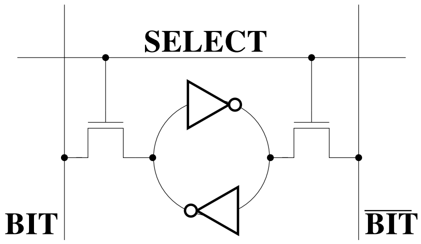
The figure on the left above shows a generic RAM structure. The memory contains 2^k elements of N bits each. A k-bit address input, $ADDR$, identifies the memory element of interest for any particular operation. The write enable input, WE, selects the operation to be performed: if WE is high, the operation is a write; if it is low, the operation is a read. Data to be written into an element are provided through N inputs at the top, and data read from an element appear on N outputs at the bottom. Finally, a **chip select** input, CS, functions as an enable control for the memory; when CS is low, the memory neither reads nor writes any location.

Random access memory further divides into two important types: **static RAM**, or **SRAM**, and **dynamic RAM**, or **DRAM**. SRAM employs active logic in the form of a two-inverter loop to maintain stored values.

DRAM uses a charged capacitor to store a bit; the charge drains over time and must be replaced, giving rise to the qualifier "dynamic." "Static" thus serves only to differentiate memories with active logic elements from those with capacitive elements. Both types are volatile, that is, both lose all data when the power supply is removed. We consider both SRAM and DRAM in this course, but the details of DRAM operation are beyond our scope.

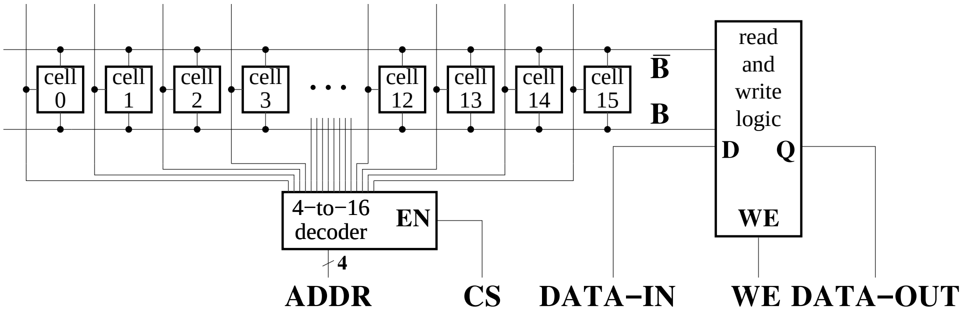
3.6.2 Static Random Access Memory

Static random access memory is used for high-speed applications such as processor caches and some embedded designs. As SRAM bit density—the number of bits in a given chip area—is significantly lower than DRAM bit density, most applications with less demanding speed requirements use DRAM. The main memory in most computers, for example, is DRAM, whereas the memory on the same chip as a processor is SRAM.[12] DRAM is also unavailable when recharging its capacitors, which can be a problem for applications with stringent real-time needs.



A diagram of an SRAM **cell** (a single bit) appears to the right. A dual-inverter loop stores the bit, and is connected to opposing BIT lines through transistors controlled by a $SELECT$ line. The cell works as follows. When $SELECT$ is high, the transistors connect the inverter loop to the bit lines. When writing a cell, the bit lines are held at opposite logic values, forcing the inverters to match the values on the lines and storing the value from the BIT input. When reading a cell, the bit lines are disconnected from other logic, allowing the inverters to drive the lines with their current outputs. The value stored previously is thus copied onto the BIT line as an output, and the opposite value is placed on the $\overline{BIT}$ line. When $SELECT$ is low, the transistors disconnect the inverters from the bit lines, and the cell holds its current value until $SELECT$ goes high again.

The actual operation of an SRAM cell is more complicated than we have described. For example, when writing a bit, the BIT lines can temporarily connect high voltage to ground (a short). The circuit must be designed carefully to minimize the power consumed during this process. When reading a bit, the BIT lines are pre-charged halfway between high-voltage and ground, and analog devices called sense amplifiers are used to detect the voltage changes on the BIT lines (driven by the inverter loop) as quickly as possible. These analog design issues are outside of the scope of our class.



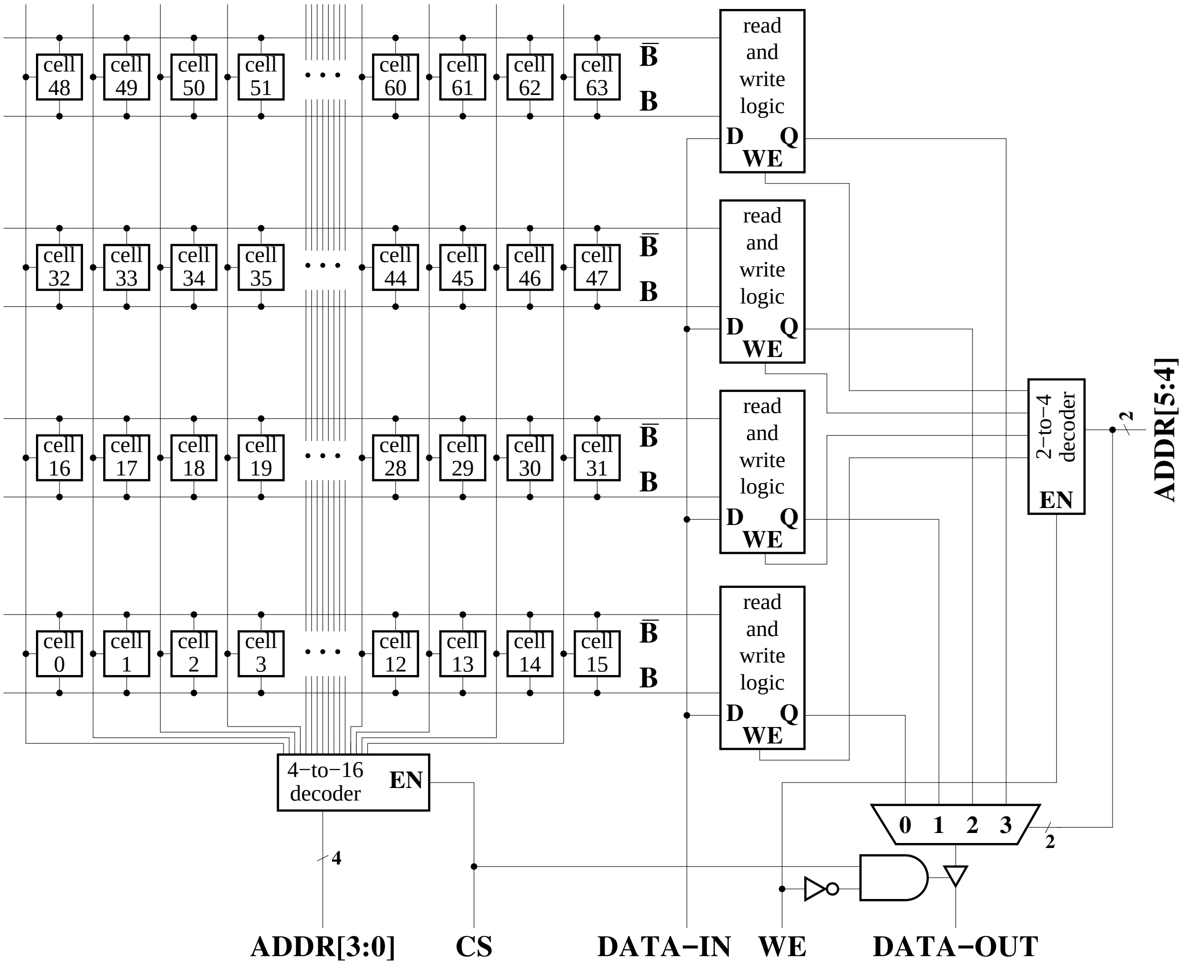
A number of cells are combined into a **bit slice**, as shown to the right. The labels along the bottom of the figure are external inputs to the bit slice, and match the labels for the abstract memory discussed earlier. The bit slice in the figure can be thought of as a 16-address, 1-bit-addressable memory ($2^4 \times 1$b).

The cells in a bit slice share bit lines and analog read and write logic, which appears to the right in the figure. Based on the $ADDR$ input, a decoder sets one cell's $SELECT$ line high to enable a read or write operation to the cell. The chip select input CS drives the enable input of the decoder, so none of the memory cells is active when chip select is low ($CS = 0$), and exactly one of the memory cells is active when chip select is high ($CS = 1$). Actual bit slices can contain many more cells than are shown in the figure—more cells means less extra logic per cell, but slower memory, since longer wires have higher capacitance.

[12] Chips combining both DRAM and processor logic are available, and are used by some processor manufacturers (such as IBM). Research is underway to couple such logic types more efficiently by building 3D stacks of chips.

A read operation is performed as follows. We set $CS = 1$ and $WE = 0$, and place the address of the cell to be read on the $ADDR$ input. The decoder outputs a 1 on the appropriate cell's $SELECT$ line, and the read logic reads the bit from the cell and delivers it to its Q output, which is then available on the bit slice's $DATA\text{-}OUT$ output.

For a write operation, we set $CS = 1$ and $WE = 1$. We again place the address of the cell to be written on the $ADDR$ input and set the value of the bit slice's $DATA\text{-}IN$ input to the value to be written into the memory cell. When the decoder activates the cell's $SELECT$ line, the write logic writes the new value from its D input into the memory cell. Later reads from that cell then produce the new value.



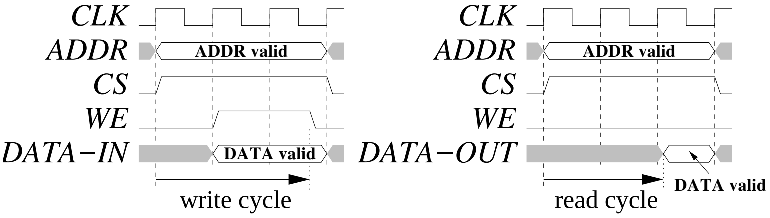
The outputs of the cell selection decoder can be used to control multiple bit slices, as shown in the figure above of a $2^6 \times 1$b memory. Selection between bit slices is then based on other bits from the address ($ADDR$). In the figure above, a 2-to-4 decoder is used to deliver write requests to one of four bit slices, and a 4-to-1 mux is used to choose the appropriate output bit for read requests.

The 4-to-16 decoder now activates one cell in each of the four bit slices. For a read operation, $WE = 0$, and the 2-to-4 decoder is not enabled, so it outputs all 0s. All four bit slices thus perform reads, and the desired result bit is forwarded to $DATA\text{-}OUT$ by the 4-to-1 mux. The tri-state buffer between the mux and $DATA\text{-}OUT$ is explained in a later section. For a write operation, exactly one of the bit slices has its WE input set to 1 by the 2-to-4 decoder. That bit slice writes the bit value delivered to all bit slices from $DATA\text{-}IN$. The other three bit slices perform reads, but their results are simply discarded.

The approach shown above, in which a cell is selected through a two-dimensional indexing scheme, is known as **coincident selection**. The qualifier "coincident" arises from the notion that the desired cell coincides with the intersection of the active row and column outputs from the decoders.

The benefit of coincident selection is easily calculated in terms of the number of gates required for the decoders. Decoder complexity is roughly equal to the number of outputs, as each output is a minterm and requires a unique gate to calculate it. Consider a 1M×8b RAM chip. The number of addresses is 2^{20}, and the total number of memory cells is 8,388,608 (2^{23}). One option is to use eight bit slices and a 20-to-1,048,576 decoder, or about 2^{20} gates. Alternatively, we can use 8,192 bit slices of 1,024 cells. For the second implementation, we need two 10-to-1024 decoders, or about 2^{11} gates. As chip area is roughly proportional to the number of gates, the savings are substantial. Other schemes are possible as well: if we want a more square chip area, we might choose to use 4,096 bit slices of 2,048 cells along with one 11-to-2048 decoder and one 9-to-512 decoder. This approach requires roughly 25% more decoder gates than our previous example, but is still far superior to the eight-bit-slice implementation.

Memories are typically unclocked devices. However, as you have seen, the circuits are highly structured, which enables engineers to cope with the complexity of sequential feedback design. Devices used to control memories are typically clocked, and the interaction between the two can be fairly complex.



Timing diagrams for reads and writes to SRAM are shown to the right. A write operation appears on the left. In the first cycle, the controller raises the chip select signal and places the memory address to be written on the address inputs. Once the memory has had time to set up the appropriate select lines internally, the WE input is raised, and data are placed on the data inputs. The delay, which is specified by the memory manufacturer, is necessary to avoid writing data to the incorrect element within the memory. The timing shown in the figure rounds this delay up to a single clock cycle, but the actual delay needed depends on the clock speed and the memory's specification. At some point after new data have been delivered to the memory, the write operation completes within the memory. The time from the application of the address until the (worst-case) completion of the write operation is called the **write cycle** of the memory, and is also specified by the memory manufacturer. Once the write cycle has passed, the controlling logic lowers WE, waits for the change to settle within the memory, then removes the address and lowers the chip select signal. The reason for the delay between these signal changes is the same: to avoid mistakenly overwriting another memory location.

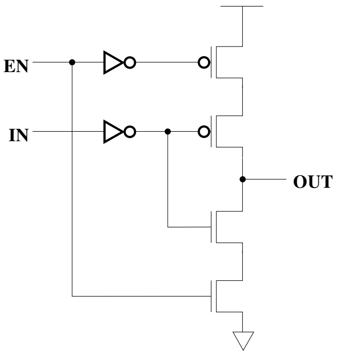
A read operation is quite similar. As shown on the right, the controlling logic places the address on the input lines and raises the chip select signal. No races need be considered, as read operations on SRAM do not affect the stored data. After a delay called the **read cycle**, the data can be read from the data outputs. The address can then be removed and the chip select signal lowered.

For both reads and writes, the number of cycles required for an operation depends on a combination of the clock cycle of the controller and the cycle time of the memory. For example, with a 25 nanosecond write cycle and a 10 nanosecond clock cycle, a write requires three cycles. In general, the number of cycles required is given by the formula $\lceil$memory cycle time/clock cycle time$\rceil$.

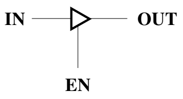
3.6.3 Tri-State Buffers and Combining Chips

Recall the buffer symbol—a triangle like an inverter, but with no inversion bubble—between the mux and the *DATA-OUT* signal of the $2^6 \times 1$b memory shown earlier. This **tri-state buffer** serves to disconnect the memory logic from the output line when the memory is not performing a read.

An implementation diagram for a tri-state buffer appears to the right along with the symbolic form and a truth table. The "Z" in the truth table output means high impedance (and is sometimes written "hi-Z"). In other words, there is effectively no electrical connection between the tri-state buffer and the output OUT.



tri−state buffer
(logic diagram)



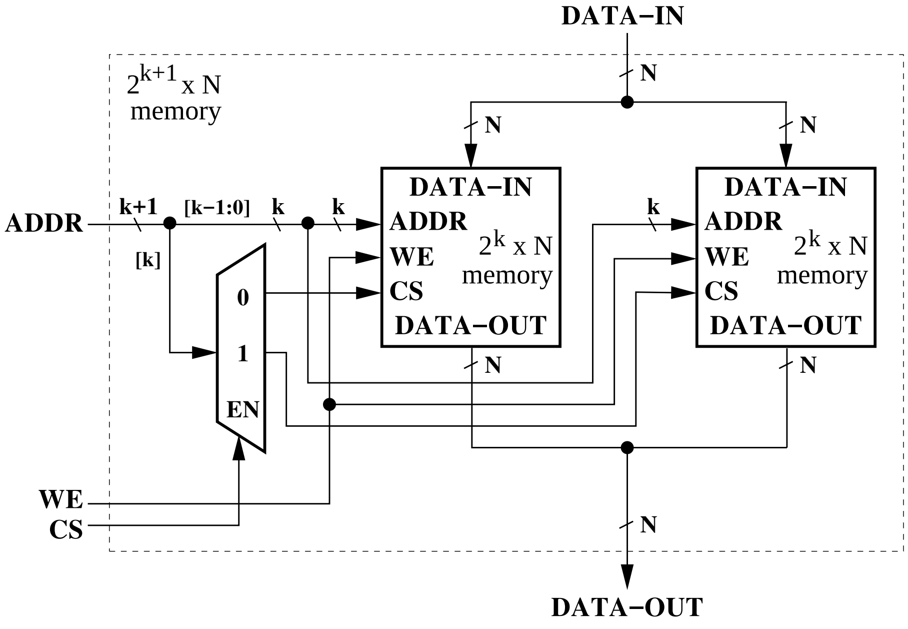
tri−state buffer
(symbolic form)

EN	IN	OUT
0	x	Z
1	0	0
1	1	1

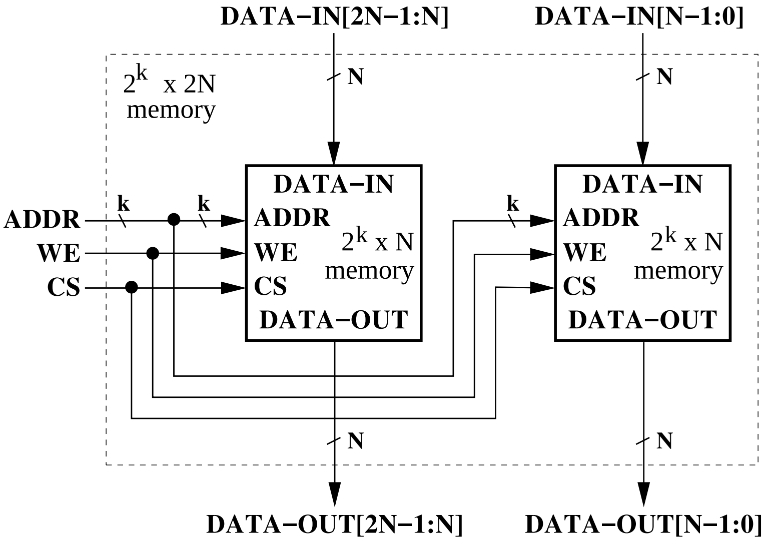
This logical disconnection is achieved by using the outer (upper and lower) pair of transistors in the logic diagram. When $EN = 0$, both transistors turn off, meaning that regardless of the value of IN, OUT is connected neither to high voltage nor to ground. When $EN = 1$, both transistors turn on, and the tri-state buffer acts as a pair of back-to-back inverters, copying the signal from IN to OUT, as shown in the truth table.

What benefit does this logical disconnection provide? So long as only one memory's chip select input is high at any time, the same output line can be shared by more than one memory without the need for additional multiplexers. Memory chips were often combined in this way to produce larger memories.

The figure to the right illustrates how larger memories can be constructed using multiple chips. In the case shown, two $2^k \times N$-bit memories are used to implement a $2^{k+1} \times N$-bit memory. One of the address bits—in the case shown, the most significant bit—is used to drive a decoder that determines which of the two chips is active ($CS = 1$). The decoder is enabled with the chip select signal for the larger memory, so neither chip is enabled when the external CS is low, as desired. The rest of the address bits, as well as the external data inputs and write enable signal, are simply delivered to both memories. The external data outputs are also connected to both memories. Ensuring that at most one chip select signal is high at any time guarantees that at most one of the two memory chips drives logic values on the data outputs.



Multiple chips can also be used to construct wider memories, as shown to the right. In the case shown, two $2^k \times N$-bit memories are used to implement a $2^k \times 2N$-bit memory. Both chips are either active or inactive at the same time, so the external address, write enable, and chip select inputs are routed to both chips. In contrast, the data inputs and outputs are separate: the left chip handles the high N bits of input on writes and produces the high N bits of output on reads, while the right chip handles the low N bits of input and produces the low N bits of output.

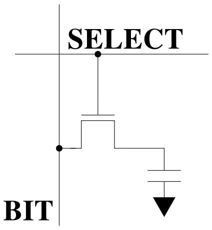


Historically, tri-state buffers were also used to reduce the number of pins needed on chips. Pins have long been a scarce resource, and the amount of data that can cross a chip's pins in a second (the product of the number of pins and the data rate per pin) has not grown nearly as rapidly as the number of transistors packed into a fixed area. By combining inputs and outputs, chip designers were able to halve the number of pins needed. For example, data inputs and outputs of memory were often combined into a single set of data wires, with bidirectional signals. When performing a read from a memory chip, the memory chip drove the data pins with the bits being read (tri-state buffers on the memory chip were enabled). When performing a write, other logic such as a processor wrote the value to be stored onto the data pins (tri-state buffers were not enabled).

3.6.4 Dynamic Random Access Memory*

Dynamic random access memory, or DRAM, is used for main memory in computers and for other applications in which size is more important than speed. While slower than SRAM, DRAM is denser (has more bits per chip area). A substantial part of DRAM density is due to transistor count: typical SRAM cells use six transistors (two for each inverter, and two more to connect the inverters to the bit lines), while DRAM cells use only a single transistor. However, memory designers have also made significant advances in further miniaturizing DRAM cells to improve density beyond the benefit available from simple transistor count.

A diagram of a DRAM cell appears to the right. DRAM storage is capacitive: a bit is stored by charging or not charging a capacitor. The capacitor is attached to a BIT line through a transistor controlled by a $SELECT$ line. When $SELECT$ is low, the capacitor is isolated and holds its charge. However, the transistor's resistance is finite, and some charge leaks out onto the bit line. Charge also leaks into the substrate on which the transistor is constructed. After some amount of time, all of the charge dissipates, and the bit is lost. To avoid such loss, the cell must be **refreshed** periodically by reading the contents and writing them back with active logic.

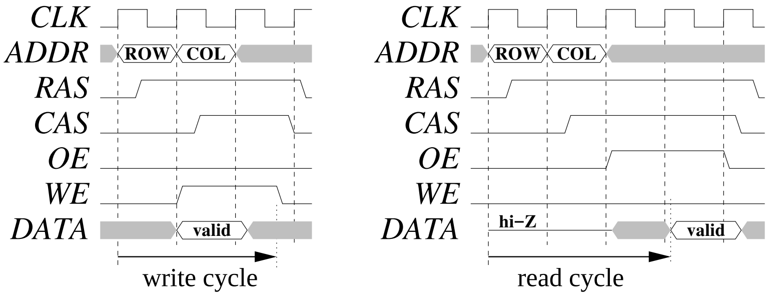


When the $SELECT$ line is high during a write operation, logic driving the bit line forces charge onto the capacitor or removes all charge from it. For a read operation, the bit line is first brought to an intermediate voltage level (a voltage level between 0 and 1), then $SELECT$ is raised, allowing the capacitor to either pull a small amount of charge from the bit line or to push a small amount of charge onto the bit line. The resulting change in voltage is then detected by a **sense amplifier** at the end of the bit line. A sense amp is analogous to a marble on a mountaintop: a small push causes the marble to roll rapidly downhill in the direction of the push. Similarly, a small change in voltage causes a sense amp's output to move rapidly to a logical 0 or 1, depending on the direction of the small change. As mentioned earlier, sense amplifiers also appear in SRAM implementations. While not technically necessary, as they are with DRAM, the use of a sense amp to react to small changes in voltage makes reads faster.

Each read operation on a DRAM cell brings the voltage on its capacitor closer to the intermediate voltage level, in effect destroying the data in the cell. DRAM is thus said to have **destructive reads**. To preserve data during a read, the bits must be written back into the cells after a read. For example, the output of the sense amplifiers can be used to drive the bit lines, rewriting the cells with the appropriate data.

At the chip level, typical DRAM inputs and outputs differ from those of SRAM. Due to the large size and high density of DRAM, addresses are split into row and column components and provided through a common set of pins. The DRAM stores the components in registers to support this approach. Additional inputs, known as the **row** and **column address strobes**—RAS and CAS, respectively—are used to indicate when address components are available. As you might guess from the structure of coincident selection, DRAM refresh occurs on a row-by-row basis (across bit slices—on columns rather than rows in the figures earlier in these notes, but the terminology of DRAM is a row). Raising the $SELECT$ line for a row destructively reads the contents of all cells on that row, forcing the cells to be rewritten and effecting a refresh. The row is thus a natural basis for the refresh cycle. The DRAM data pins provide bidirectional signals for reading and writing elements of the DRAM. An **output enable** input, OE, controls tri-state buffers with the DRAM to determine whether or not the DRAM drives the data pins. The WE input, which controls the type of operation, is also present.

Timing diagrams for writes and reads on a historical DRAM implementation appear to the right. In both cases, the row component of the address is first applied to the address pins, then RAS is raised. In the next cycle of the controlling logic, the column component is applied to the address pins, and CAS is raised.



For a write, as shown on the left, the WE signal and the data can also be applied in the second cycle. The DRAM has internal timing and control logic that prevent races from overwriting an incorrect element (remember that the row and column addresses have to be stored in registers). The DRAM again specifies a write cycle, after which the operation is guaranteed to be complete. In order, the WE, CAS, and RAS signals are then lowered.

For a read operation, the output enable signal, OE, is raised after CAS is raised. The $DATA$ pins, which should be floating (in other words, not driven by any logic), are then driven by the DRAM. After the read cycle, valid data appear on the $DATA$ pins, and OE, CAS, and RAS are lowered in order after the data are read.

Modern DRAM chips are substantially more sophisticated than those discussed here, and many of the functions that used to be provided by external logic are now integrated onto the chips themselves. As an example of modern DRAMs, one can obtain the data sheet for Micron Semiconductor's 8Gb ($2^{31} \times 4$b, for example) DDR4 SDRAM, which is 366 pages long as of 11 May 2016.

The ability to synchronize to an external clock has become prevalent in the industry, leading to the somewhat confusing term SDRAM, which stands for **synchronous DRAM**. The memory structures themselves are still unclocked, but logic is provided on the chip to synchronize accesses to the external clock without the need for additional logic. The clock provided to the Micron chip just mentioned can be as fast as 1.6 GHz, and data can be transferred on both the rising and falling edges of the clock (hence the name DDR, or **double data rate**).

In addition to row and column components of the address, these chips further separate cells into **banks** and groups of banks. These allow a user to exploit parallelism by starting reads or writes to separate banks at the same time, thus improving the speed at which data can move in and out of the memory. For the $2^{31} \times 4$b version of the Micron chip, the cells are structured into 4 groups of 4 banks (16 banks total), each with 131,072 rows and 1,024 columns.

DRAM implementations provide interfaces for specifying refresh operations in addition to reads and writes. Managing refresh timing and execution is generally left to an external DRAM controller. For the Micron chip, refresh commands must be issued every 7.8 microseconds at normal temperatures. Each command refreshes about 2^{20} cells, so 8,192 commands refresh the whole chip in less than 64 milliseconds. Alternatively, the chip can handle refresh on-chip in order to maintain memory contents when the rest of the system is powered down.

ECE120: Introduction to Computer Engineering

Notes Set 3.7 From FSM to Computer

The FSM designs we have explored so far have started with a human-based design process in which someone writes down the desired behavior in terms of states, inputs, outputs, and transitions. Such an approach makes it easier to build a digital FSM, since the abstraction used corresponds almost directly to the implementation.

As an alternative, one can start by mapping the desired task into a high-level programming language, then using components such as registers, counters, and memories to implement the variables needed. In this approach, the control structure of the code maps into a high-level FSM design. Of course, in order to implement our FSM with digital logic, we eventually still need to map down to bits and gates.

In this set of notes, we show how one can transform a piece of code written in a high-level language into an FSM. This process is meant to help you understand how we can design an FSM that executes simple pieces of a flow chart such as assignments, `if` statements, and loops. Later, we generalize this concept and build an FSM that allows the pieces to be executed to be specified after the FSM is built—in other words, the FSM executes a program specified by bits stored in memory. This more general model, as you might have already guessed, is a computer.

3.7.1 Specifying the Problem

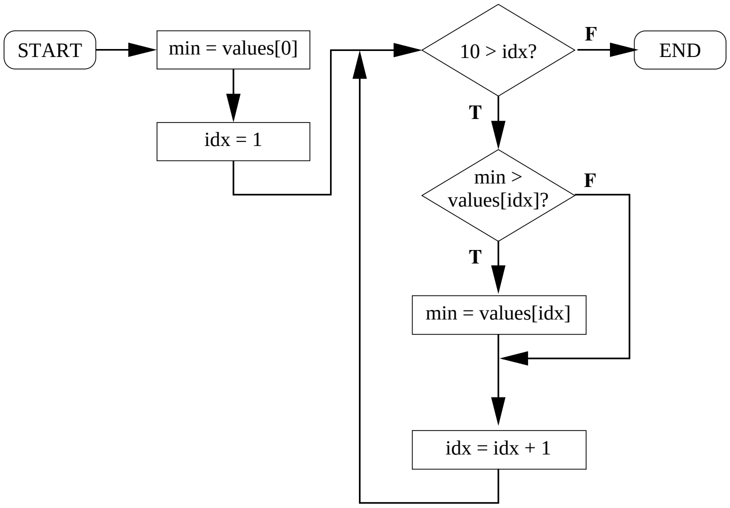
Let's begin by specifying the problem that we want to solve. Say that we want to find the minimum value in a set of 10 integers. Using the C programming language, we can write the following fragment of code:

```
int values[10];    /* 10 integers--filled in by other code */
int idx;
int min

min = values[0];
for (idx = 1; 10 > idx; idx = idx + 1) {
    if (min > values[idx]) {
        min = values[idx];
    }
}
/* The minimum value from array is now in min.  */
```

The code uses array notation, which we have not used previously in our class, so let's first discuss the meaning of the code.

The code uses three variables. The variable `values` represents the 10 values in our set. The suffix "[10]" after the variable name tells the compiler that we want an array of 10 integers (`int`) indexed from 0 to 9. These integers can be treated as 10 separate variables, but can be accessed using the single name "`values`" along with an index (again, from 0 to 9 in this case). The variable `idx` holds a loop index that we use to examine each of the values one by one in order to find the minimum value in the set. Finally, the variable `min` holds the smallest known value as the program examines each of the values in the set.

The program body consists of two statements. We assume that some other piece of code—one not shown here—has initialized the 10 values in our set before the code above executes. The first statement initializes the minimum known value (`min`) to the value stored at index 0 in the array (`values[0]`). The second statement is a loop in which the variable `index` takes on values from 1 to 9. For each value, an `if` statement compares the current known minimum with the value stored in the array at index given by the `idx` variable. If the stored value is smaller, the current known value (again, `min`) is updated to reflect the program's having found a smaller value. When the loop finishes all nine iterations, the variable `min` holds the smallest value among the set of 10 integers stored in the `values` array.



As a first step towards designing an FSM to implement the code, we transform the code into a flow chart, as shown to the right. The program again begins with initialization, which appears in the second column of the flow chart. The loop in the program translates to the third column of the flow chart, and the `if` statement to the middle comparison and update of `min`.

Our goal is now to design an FSM to implement the flow chart. In order to do so, we want to leverage the same kind of abstraction that we used earlier, when extending our keyless entry system with a timer. Although the timer's value was technically also part of the FSM's state, we treated it as data and integrated it into our next-state decisions in only a couple of cases.

For our minimum value problem, we have two sources of data. First, an external program supplies data in the form of a set of 10 integers. If we assume 32-bit integers, these data technically form 320 input bits! Second, as with the keyless entry system timer, we have data used internally by our FSM, such as the loop index and the current minimum value. These are technically state bits. For both types of data, we treat them abstractly as values rather than thinking of them individually as bits, allowing us to develop our FSM at a high-level and then to implement it using the components that we have developed earlier in our course.

3.7.2 Choosing Components and Identifying States

Now we are ready to design an FSM that implements the flow chart. What components do we need, other than our state logic? We use registers and counters to implement the variables `idx` and `min` in the program. For the array `values`, we use a 16×32-bit memory.[13] We need a comparator to implement the test for the `if` statement. We choose to use a serial comparator, which allows us to illustrate again how one logical high-level state can be subdivided into many actual states. To operate the serial comparator, we make use of two shift registers that present the comparator with one bit per cycle on each input, and a counter to keep track of the comparator's progress.

How do we identify high-level states from our flow chart? Although the flow chart attempts to break down the program into 'simple' steps, one step of a flow chart may sometimes require more than one state in an FSM. Similarly, one FSM state may be able to implement several steps in a flow chart, if those steps can be performed simultaneously. Our design illustrates both possibilities.

How we map flow chart elements into FSM states also depends to some degree on what components we use, which is why we began with some discussion of components. In practice, one can go back and forth between the two, adjusting components to better match the high-level states, and adjusting states to better match the desired components.

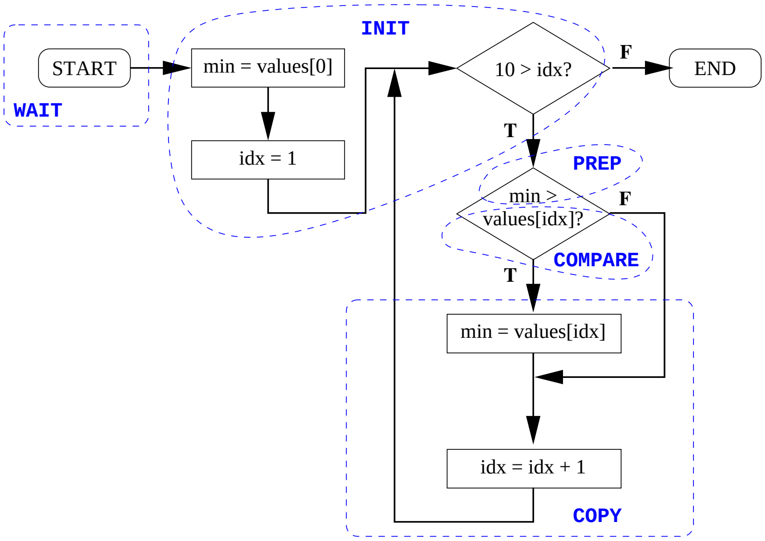
Finally, note that we are only concerned with high-level states, so we do not need to provide details (yet) down to the level of individual clock cycles, but we do want to define high-level states that can be implemented in a fixed number of cycles, or at least a controllable number of cycles. If we cannot specify clearly when transitions occur from an FSM state, we may not be able to implement the state.

[13] We technically only need a 10×32-bit memory, but we round up the size of the address space to reflect more realistic memory designs; one can always optimize later.

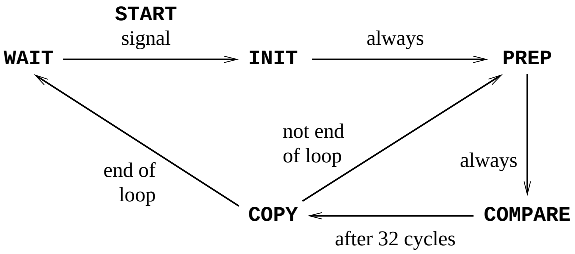
Now let's go through the flow chart and identify states. Initialization of `min` and `idx` need not occur serially, and the result of the first comparison between `idx` and the constant 10 is known in advance, so we can merge all three operations into a single state, which we call `INIT`.

We can also merge the updates of `min` and `idx` into a second FSM state, which we call `COPY`. However, the update to `min` occurs only when the comparison (`min > value[idx]`) is true. We can use logic to predicate execution of the update. In other words, we can use the output of the comparator, which is available after the comparator has finished comparing the two values (in a high-level FSM state that we have yet to define), to determine whether or not the register holding `min` loads a new value in the `COPY` state.

Our model of use for this FSM involves external logic filling the memory (the array of integer values), executing the FSM "code," and then checking the answer. To support this use model, we create a FSM state called `WAIT` for cycles in which the FSM has no work to do. Later, we also make use of an external input signal `START` to start the FSM execution. The `WAIT` state logically corresponds to the "START" bubble in the flow chart.



Only the test for the `if` statement remains. Using a serial comparator to compare two 32-bit values requires 32 cycles. However, we need an additional cycle to move values into our shift registers so that the comparator can see the first bit. Thus our single comparison operation breaks into two high-level states. In the first state, which we call `PREP`, we copy `min` to one of the shift registers, copy `values[idx]` to the other shift register, and reset the counter that measures the cycles needed for our serial comparator. We then move to a second high-level state, which we call `COMPARE`, in which we feed one bit per cycle from each shift register to the serial comparator. The `COMPARE` state executes for 32 cycles, after which the comparator produces the one-bit answer that we need, and we can move to the `COPY` state. The association between the flow chart and the high-level FSM states is illustrated in the figure shown to the right above.



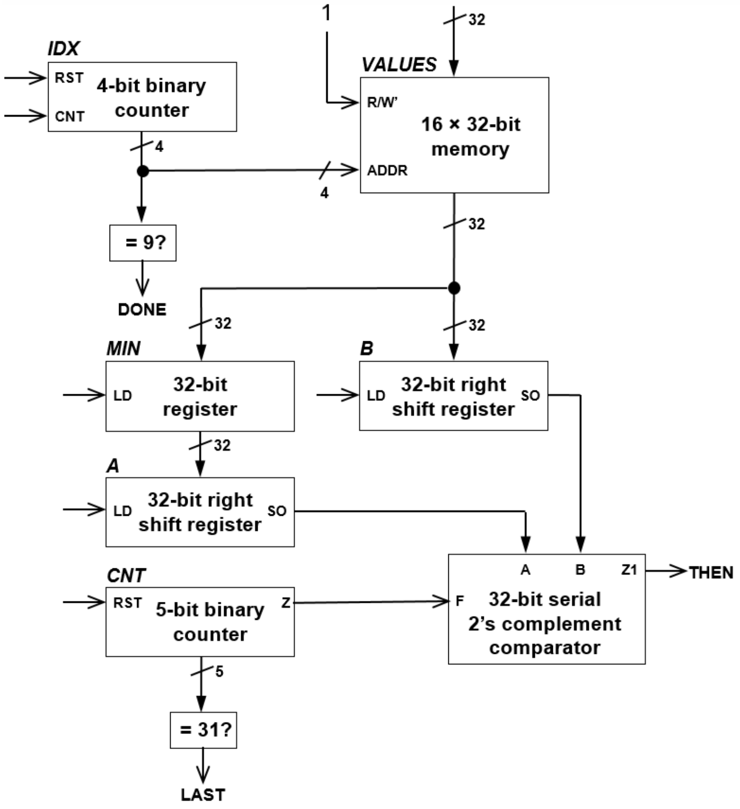
We can now also draw an abstract state diagram for our FSM, as shown to the right. The FSM begins in the `WAIT` state. After external logic fills the `values` array, it signals the FSM to begin by raising the `START` signal. The FSM transitions into the `INIT` state, and in the next cycle into the `PREP` state. From `PREP`, the FSM always moves to `COMPARE`, where it remains for 32 cycles while the serial comparator executes a comparison. After `COMPARE`, the FSM moves to the `COPY` state, where it remains for one cycle. The transition from `COPY` depends on how many loop iterations have executed. If more loop iterations remain, the FSM moves to `PREP` to execute the next iteration. If the loop is done, the FSM returns to `WAIT` to allow external logic to read the result of the computation.

3.7.3 Laying Out Components

Our high-level FSM design tells us what our components need to be able to do in any given cycle. For example, when we load new values into the shift registers that provide bits to the serial comparator, we always copy `min` into one shift register and `values[idx]` into the second. Using this information, we can put together our components and simplify our design by fixing the way in which bits flow between them.

The figure at the right shows how we can organize our components. Again, in practice, one goes back and forth thinking about states, components, and flow from state to state. In these notes, we present only a completed design.

Let's take a detailed look at each of the components. At the upper left of the figure is a 4-bit binary counter called `IDX` to hold the `idx` variable. The counter can be reset to 0 using the `RST` input. Otherwise, the `CNT` input controls whether or not the counter increments its value. With this counter design, we can force `idx` to 0 in the `WAIT` state and then count upwards in the `INIT` and `COPY` states.



A memory labeled `VALUES` to hold the array `values` appears in the upper right of the figure. The read/write control for the memory is hardwired to 1 (read) in the figure, and the data input lines are unattached. To integrate with other logic that can operate our FSM, we need to add more control logic to allow writing into the memory and to attach the data inputs to something that provides the data bits. The address input of the memory comes always from the `IDX` counter value; in other words, whenever we access this memory by making use of the data output lines, we read `values[idx]`.

In the middle left of the figure is a 32-bit register for the `min` variable. It has a control input `LD` that determines whether or not it loads a new value at the end of the clock cycle. If a new value is loaded, the new value always corresponds to the output of the `VALUES` memory, `values[idx]`. Recall that `min` always changes in the `INIT` state, and may change in the `COPY` state. But the new value stored in `min` is always `values[idx]`. Note also that when the FSM completes its task, the result of the computation is left in the `MIN` register for external logic to read (connections for this purpose are not shown in the figure).

Continuing downward in the figure, we see two right shift registers labeled `A` and `B`. Each has a control input `LD` that enables a parallel load. Register `A` loads from register `MIN`, and register `B` loads from the memory data output (`values[idx]`). These loads are needed in the `PREP` state of our FSM. When `LD` is low, the shift registers simply shifts to the right. The serial output `SO` makes the least significant bit of each shift register available. Shifting is necessary to feed the serial comparator in the `COMPARE` state.

Below register `A` is a 5-bit binary counter called `CNT`. The counter is used to control the serial comparator in the `COMPARE` state. A reset input `RST` allows it to be forced to 0 in the `PREP` state. When the counter value is exactly zero, the output `Z` is high.

The last major component is the serial comparator, which is based on the design developed in Notes Set 3.1. The two bits to be compared in a cycle come from shift registers A and B. The first bit indicator comes from the zero indicator of counter CNT. The comparator actually produces two outputs (Z1 and Z0), but the meaning of the Z1 output by itself is A > B. In the diagram, this signal has been labeled THEN.

There are two additional elements in the figure that we have yet to discuss. Each simply compares the value in a register with a fixed constant and produces a 1-bit signal. When the FSM finishes an iteration of the loop in the COPY state, it must check the loop condition (10 > idx) and move either to the PREP state or, when the loop finishes, to the WAIT state to let the external logic read the answer from the MIN register. The loop is done when the current iteration count is nine, so we compare IDX with nine to produce the DONE signal. The other constant comparison is between the counter CNT and the value 31 to produce the LAST signal, which indicates that the serial comparator is on its last cycle of comparison. In the cycle after LAST is high, the THEN output of the comparator indicates whether or not A > B.

3.7.4 Control and Data

One can think of the components and the interconnections between them as enabling the movement of data between registers, while the high-level FSM controls which data move from register to register in each cycle. With this model in mind, we call the components and interconnections for our design the **datapath**—a term that we will see again when we examine the parts of a computer in the coming weeks. The datapath requires several inputs to control the operation of the components—these we can treat as outputs of the FSM. These signals allow the FSM to control the motion of data in the datapath, so we call them **control signals**. Similarly, the datapath produces several outputs that we can treat as inputs to the FSM. The tables below summarize the control signals (left) and outputs (right) from the datapath for our FSM.

datapath input	meaning
IDX.RST	reset IDX counter to 0
IDX.CNT	increment IDX counter
MIN.LD	load new value into MIN register
A.LD	load new value into shift register A
B.LD	load new value into shift register B
CNT.RST	reset CNT counter

datapath output	meaning	based on
DONE	last loop iteration finished	IDX = 9
LAST	serial comparator executing last cycle	CNT = 31
THEN	if statement condition true	A > B

Using the datapath controls signals and outputs, we can now write a more formal state transition table for the FSM, as shown below. The "actions" column of the table lists the changes to register and counter values that are made in each of the FSM states. The notation used to represent the actions is called **register transfer language** (**RTL**). The meaning of an individual action is similar to the meaning of the corresponding statement from our C code or from the flow chart. For example, in the WAIT state, "IDX ← 0" means the same thing as "idx = 0;". In particular, both mean that the value currently stored in the IDX counter is overwritten with the number 0 (all 0 bits).

state	actions (simultaneous)	condition	next state
WAIT	IDX ← 0 (to read VALUES[0] in INIT)	START $\overline{\text{START}}$	INIT WAIT

INIT	MIN ← VALUES[IDX] (IDX is 0 in this state)
	IDX ← IDX + 1
		(always)	PREP

PREP	A ← MIN
	B ← VALUES[IDX]
	CNT ← 0
		(always)	COMPARE

COMPARE	run serial comparator	LAST $\overline{\text{LAST}}$	COPY COMPARE

COPY	THEN: MIN ← VALUES[IDX]
	IDX ← IDX + 1
		DONE
		$\overline{\text{DONE}}$
			WAIT
			PREP

The meaning of RTL is slightly different from the usual interpretation of high-level programming languages, however, in terms of when the actions happen. A list of C statements is generally executed one at a time. In contrast, the entire list of RTL actions for an FSM state is executed simultaneously, at the end of the clock cycle. As you know, an FSM moves from its current state into a new state at the end of every clock cycle, so actions during different cycles usually are associated with different states. We can, however, change the value in more than one register at the end of the same clock cycle, so we can execute more than one RTL action in the same state, so long as the actions do not exceed the capabilities of our datapath (the components must be able to support the simultaneous execution of the actions). Some care must be taken with states that execute for more than one cycle to ensure that repeating the RTL actions is appropriate. In our design, only the `WAIT` and `COMPARE` states execute for more than one cycle. The `WAIT` state resets the `IDX` counter repeatedly, which causes no problems. The `COMPARE` statement has no RTL actions—all of the shifting, comparison, and counting activity needed to do its work occurs within the datapath itself.

One additional piece of RTL syntax needs explanation. In the `COPY` state, the first action begins with "`THEN`:," which means that the prefixed RTL action occurs only when the `THEN` signal is high. Recall that the `THEN` signal indicates that the comparator has found `A` > `B`, so the equivalent C code is "`if (A > B) {min = values[idx]}`".

3.7.5 State Representation and Logic Expressions

Let's think about the representation for the FSM states. The FSM has five states, so we could use as few as three flip-flops. Instead, we choose to use a **one-hot encoding**, in which any valid bit pattern has exactly one 1 bit. In other words, we use five flip-flops instead of three, and our states are represented with the bit patterns 10000, 01000, 00100, 00010, and 00001.

The table below shows the mapping from each high-level state to both the five-bit encoding for the state as well as the six control signals needed for the datapath. For each state, the values of the control signals can be found by examining the actions necessary in that state.

state	$S_4S_3S_2S_1S_0$	IDX.RST	IDX.CNT	MIN.LD	A.LD	B.LD	CNT.RST
WAIT	1 0 0 0 0	1	0	0	0	0	0
INIT	0 1 0 0 0	0	1	1	0	0	0
PREP	0 0 1 0 0	0	0	0	1	1	1
COMPARE	0 0 0 1 0	0	0	0	0	0	0
COPY	0 0 0 0 1	0	1	THEN	0	0	0

The `WAIT` state needs to set `IDX` to 0 but need not affect other register or counter values, so `WAIT` produces a 1 only for `IDX.RST`. The `INIT` state needs to load `values[0]` into the `MIN` register while simultaneously incrementing the `IDX` counter (from 0 to 1), so `INIT` produces 1s for `IDX.CNT` and `MIN.LD`. The `PREP` state loads both shift registers and resets the counter `CNT` by producing 1s for `A.LD`, `B.LD`, and `CNT.RST`. The `COMPARE` state does not change any register values, so it produces all 0s. Finally, the `COPY` state increments the `IDX` counter while simultaneously loading a new value into the `MIN` register. The `COPY` state produces 1 for `IDX.CNT`, but must use the signal `THEN` coming from the datapath to decide whether or not `MIN` is loaded.

The advantage of a one-hot encoding becomes obvious when we write equations for the six control signals and the next-state logic, as shown to the right. Implementing the logic to complete our design now requires only a handful of small logic gates.

$$\begin{aligned} \texttt{IDX.RST} &= S_4 \\ \texttt{IDX.CNT} &= S_3 + S_0 \\ \texttt{MIN.LD} &= S_3 + S_0 \cdot \texttt{THEN} \\ \texttt{A.LD} &= S_2 \\ \texttt{B.LD} &= S_2 \\ \texttt{CNT.RST} &= S_2 \end{aligned}$$

$$\begin{aligned} S_4^+ &= S_4 \cdot \overline{\texttt{START}} + S_0 \cdot \texttt{DONE} \\ S_3^+ &= S_4 \cdot \texttt{START} \\ S_2^+ &= S_3 + S_0 \cdot \overline{\texttt{DONE}} \\ S_1^+ &= S_2 + S_1 \cdot \overline{\texttt{LAST}} \\ S_0^+ &= S_1 \cdot \texttt{LAST} \end{aligned}$$

Notice that the terms in each control signal can be read directly from the rows of the state table and OR'd together. The terms in each of the next-state equations represent the incoming arcs for the corresponding state. For example, the `WAIT` state has one self-loop (the first term) and a transition arc coming from the `COPY` state when the loop is done. These expressions complete our design.

ECE120: Introduction to Computer Engineering

Notes Set 3.8 Summary of Part 3 of the Course

Students often find this part of the course more challenging than the earlier parts of the course. In addition to these notes, you should read Chapters 4 and 5 of the Patt and Patel textbook, which cover the von Neumann model, instruction processing, and ISAs.

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 mux, for example—provided that you know what a mux does and can derive a gate diagram correctly for one in a few minutes. Higher-level skills are much more valuable.

- digital systems terms
  - module
  - fan-in
  - fan-out
  - machine models: Moore and Mealy
- simple state machines
  - synchronous counter
  - ripple counter
  - serialization (of bit-sliced design)
- finite state machines (FSMs)
  - states and state representation
  - transition rule
  - self-loop
  - next state (+) notation
  - meaning of don't care in input combination
  - meaning of don't care in output
  - unused states and initialization
  - completeness (with regard to FSM specification)
  - list of (abstract) states
  - next-state table/state transition table/state table
  - state transition diagram/transition diagram/state diagram
- memory
  - number of addresses
  - addressability
  - read/write logic
  - serial/random access memory (RAM)
  - volatile/non-volatile (N-V)
  - static/dynamic RAM (SRAM/DRAM)
  - SRAM cell
  - DRAM cell
  - design as a collection of cells
  - coincident selection
- von Neumann model
  - processing unit
    - register file
    - arithmetic logic unit (ALU)
    - word size
  - control unit
    - program counter (PC)
    - instruction register (IR)
    - implementation as FSM
  - input and output units
  - memory
    - memory address register (MAR)
    - memory data register (MDR)
  - processor datapath
  - bus
  - control signal
- tri-state buffer
  - meaning of Z/hi-Z output
  - use in distributed mux
- instruction processing
  - fetch
  - decode
  - execute
  - register transfer language (RTL)
- Instruction Set Architecture (ISA)
  - instruction encoding
  - field (of an encoded instruction)
  - operation code (opcode)
  - types of instructions
    - operations
    - data movement
    - control flow
  - addressing modes
    - immediate
    - register
    - PC-relative
    - indirect
    - base + offset

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

- Transform a bit-sliced design into a serial design, and explain the tradeoffs involved in terms of area and time required to compute a result.
- Based on a transition diagram, implement a synchronous counter from flip-flops and logic gates.
- Implement a binary ripple counter (but not necessarily a more general type of ripple counter) from flip-flops and logic gates.
- Given an FSM implemented as digital logic, analyze the FSM to produce a state transition diagram.
- Design an FSM to meet an abstract specification for a task, including production of specified output signals, and possibly including selection of appropriate inputs.
- Complete the specification of an FSM by ensuring that each state includes a transition rule for every possible input combination.
- Compose memory chips into larger memory systems, using additional decoders when necessary.
- Encode LC-3 instructions into machine code.
- Read and understand programs written in LC-3 assembly/machine code.

At a higher level, we expect that you understand the concepts and ideas sufficiently well to do the following:

- Abstract design symmetries from an FSM specification in order to simplify the implementation.
- Make use of a high-level state design, possibly with many sub-states in each high-level state, to simplify the implementation.
- Use counters to insert time-based transitions between states (such as timeouts).
- Implement an FSM using logic components such as registers, counters, comparators, and adders as building blocks.
- Explain the basic organization of a computer's microarchitecture as well as the role played by elements of a von Neumann design in the processing of instructions.
- Identify the stages of processing an instruction (such as fetch, decode, getting operands, execution, and writing back results) in a processor control unit state machine diagram.

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

- Explain the difference between the Moore and Mealy machine models, as well as why you might find each of them useful when designing an FSM.
- Understand the need for initialization of an FSM, be able to analyze and identify potential problems arising from lack of initialization, and be able to extend an implementation to include initialization to an appropriate state when necessary.
- Understand how the choice of internal state bits for an FSM can affect the complexity of the implementation of next-state and output logic, and be able to select a reasonable state assignment.
- Identify and fix design flaws in simple FSMs by analyzing an existing implementation, comparing it with the specification, and removing any differences by making any necessary changes to the implementation.

ECE120: Introduction to Computer Engineering

Notes Set 4.1 Control Unit Design

Appendix C of the Patt and Patel textbook describes a microarchitecture for the LC-3 ISA, including a control unit implementation. In this set of notes, we introduce a few concepts and strategies for control unit design, using the textbook's LC-3 microarchitecture to help illustrate them. Several figures from the textbook are reproduced with permission in these notes as an aid to understanding.

The control unit of a computer based on the von Neumann model can be viewed as an FSM that fetches instructions from memory and executes them. Many possible implementations exist both for the control unit itself and for the resources that it controls, the other components in the von Neumann model, which we collectively call the **datapath**. In this set of notes, we discuss two strategies for structured control unit design and introduce the idea of using memories to encode logic functions.

Let's begin by recalling that the control unit consists of three parts: a high-level FSM that controls instruction processing, a program counter (PC) register that holds the address of the next instruction to be executed, and an instruction register (IR) that holds the current instruction as it executes.

Other von Neumann components provide inputs to the control unit. The memory unit, for example, contains the instructions and data on which the program executes. The processing unit contains a register file and condition codes (N, Z, and P for the LC-3 ISA). The outputs of the control unit are signals that control operation of the datapath: the processing unit, the memory, and the I/O interfaces. The basic problem that we must solve, then, for control unit design, is to map instruction processing and the state of the FSM (including the PC and the IR) into appropriate sequences of **control signals** for the datapath.

4.1.1 LC-3 Datapath Control Signals

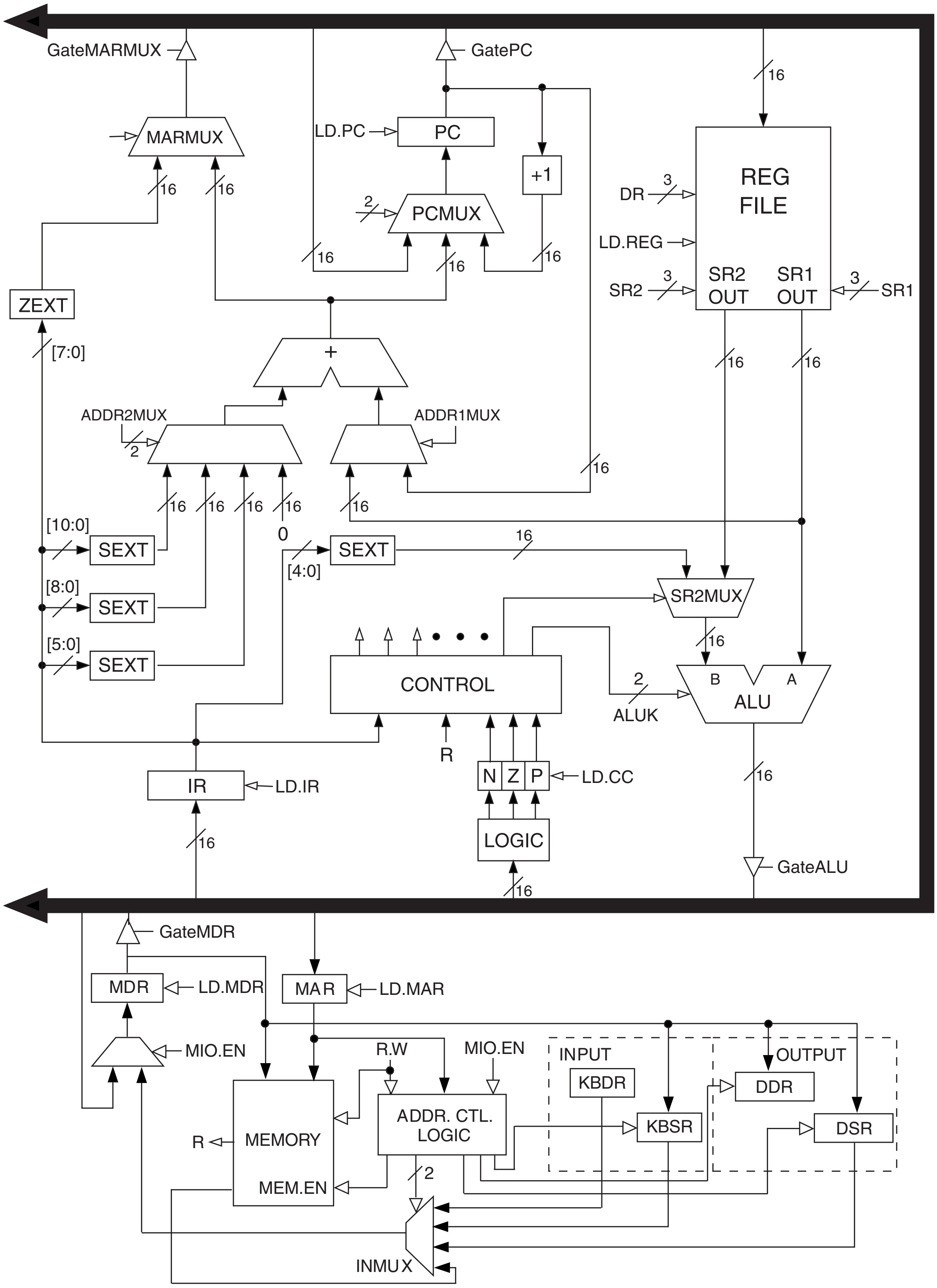
As we have skipped over the implementation details of interrupts and privilege of the LC-3 in our class, let's consider a microarchitecture and datapath without those capabilities. The figure on the next page (Patt and Patel Figure C.3) shows an LC-3 datapath and control signals without support for interrupts and privilege.

Some of the datapath control signals mentioned in the textbook are no longer necessary in the simplified design. Let's discuss the signals that remain and give some examples of how they are used. A list appears to the right.

signal	meaning
LD.MAR	load new value into memory address register
LD.MDR	load new value into memory data register
LD.IR	load new value into instruction register
LD.BEN	load new value into branch enable register
LD.REG	load new value into register file
LD.CC	load new values into condition code registers (N,Z,P)
LD.PC	load new value into program counter
GatePC	write program counter value onto bus
GateMDR	write memory data register onto bus
GateALU	write arithmetic logic unit result onto bus
GateMARMUX	write memory address register mux output onto bus
PCMUX	select value to write to program counter (2 bits)
DRMUX	select value to write to destination register (2 bits)
SR1MUX	select register to read from register file (2 bits)
ADDR1MUX	select register component of address (1 bit)
ADDR2MUX	select offset component of address (2 bits)
MARMUX	select type of address generation (1 bit)
ALUK	select arithmetic logic unit operation (2 bits)
MIO.EN	enable memory
R.W	read or write from memory

First, we have a set of seven 1-bit control signals (starting with "LD.") that specifies whether registers in the datapath load new values.

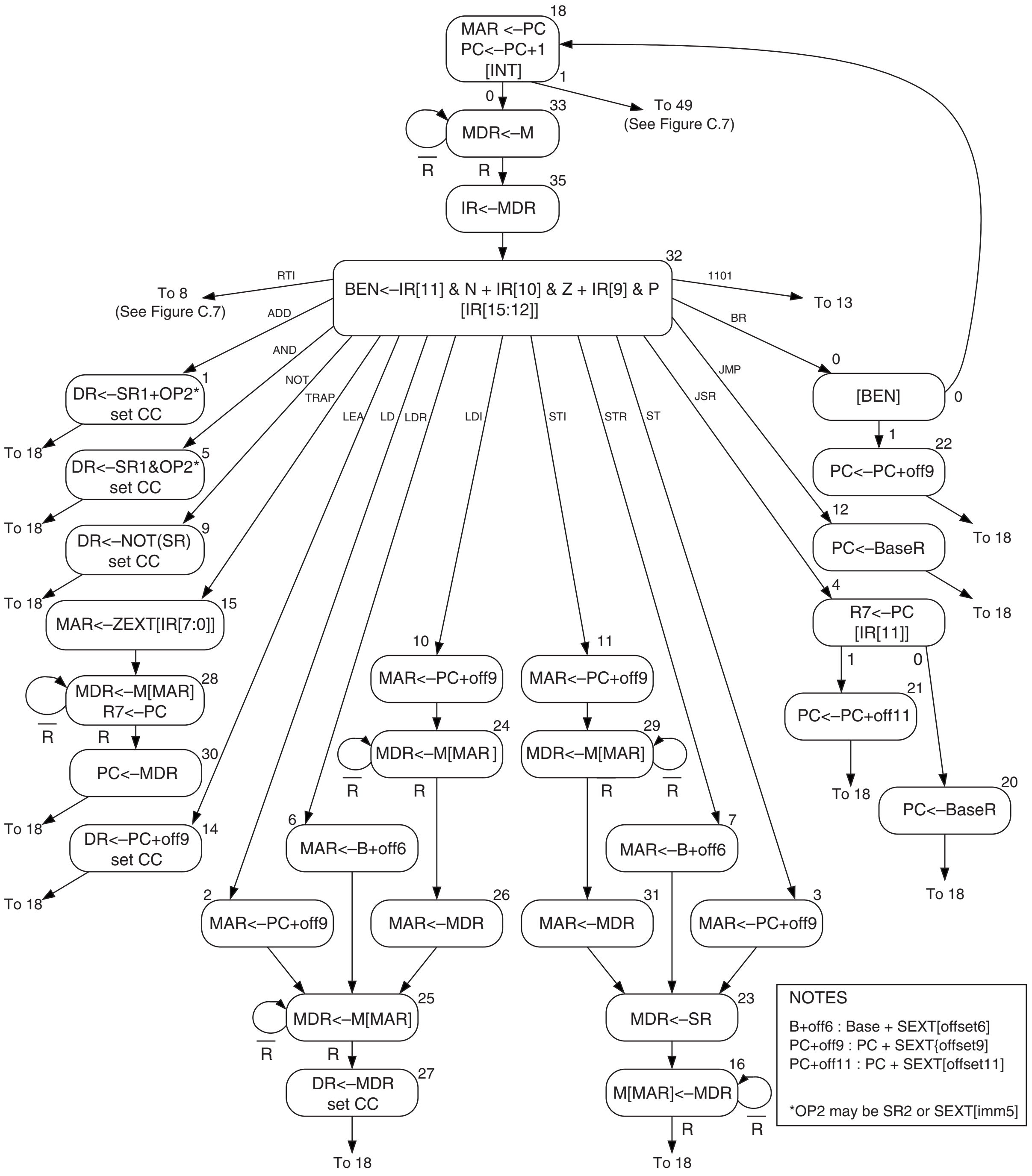
Next, there are four 1-bit signals (starting with "Gate") for tri-state buffers that control access to the bus. These four implement a distributed mux for the bus. Only one value can appear on the bus in any cycle, so at most one of these signals can be 1; others must all be 0 to avoid creating a short.


GateMARMUX
GatePC
16
MARMUX
LD.PC
PC
+1
16
16
REG
FILE
DR
3
2
PCMUX
LD.REG
16
16
16
SR2
3
SR2
OUT
SR1
OUT
3
SR1
ZEXT
[7:0]
+
16
16
ADDR2MUX
ADDR1MUX
2
16
16
16
16
16
16
[10:0]
SEXT
0
SEXT
16
[4:0]
[8:0]
SEXT
SR2MUX
16
[5:0]
SEXT
CONTROL
2
B
A
ALU
ALUK
R
IR
LD.IR
N
Z
P
LD.CC
16
16
LOGIC
GateALU
16
GateMDR
MDR
LD.MDR
MAR
LD.MAR
MIO.EN
R.W
MIO.EN
INPUT
OUTPUT
KBDR
DDR
ADDR. CTL.
LOGIC
KBSR
DSR
R
MEMORY
MEM.EN
2
INMUX

The third group (ending with "MUX") of signals controls multiplexers in the datapath. The number of bits for each depends on the number of inputs to the mux; the total number of signals is 10. The last two groups of signals control the ALU and the memory, requiring a total of 4 more signals. The total of all groups is thus 25 control signals for the datapath without support for privilege and interrupts.

4.1.2 Example Control Word: ADD

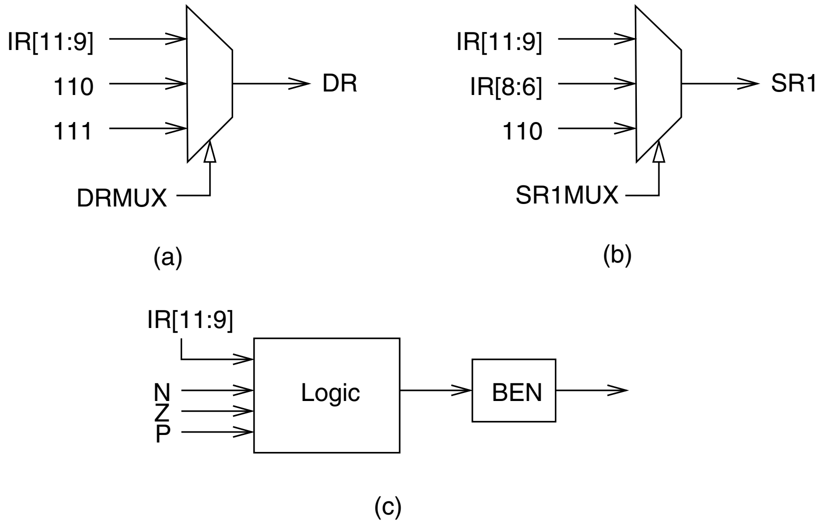
Before we begin to discuss control unit design in more detail, let's work through a couple of examples of implementing specific RTL with the control signals available. The figure below (Patt and Patel Figure C.2) shows a state machine for the LC-3 ISA (again without detail on interrupts and privilege).



Consider now the state that implements the ADD instruction—state number 1 in the figure on the previous page, just below and to the left of the decode state. The RTL for the state is: DR ← SR + OP2, set CC.

We can think of the 25 control signals that implement the desired RTL as a **control word** for the datapath. Let's begin with the register load control signals. The RTL writes to two types of registers: the register file and the condition codes. To accomplish these simultaneous writes, LD.REG and LD.CC must be high. No other registers in the datapath should change, so the other five LD signals—LD.MAR, LD.MDR, LD.IR, LD.BEN, and LD.PC—should be low.

What about the bus? In order to write the result of the add operation into the register file, the control signals must allow the ALU to write its result on to the bus. So we need GateALU=1. And the other Gate signals—GatePC, GateMDR, and GateMARMUX—must all be 0. The condition codes are also calculated from the value on the bus, but they are calculated on the same value as is written to the register file (by the definition of the LC-3 ISA). If the RTL for an FSM state implicitly requires more than one value to appear on the bus in the same cycle, that state is impossible to implement using the given datapath. Either the datapath or the state machine must be changed in such a case. The textbook's design has been fairly thoroughly tested and debugged.



The earlier figure of the datapath does not show all of the muxes. The remaining muxes appear in the figure to the right (Patt and Patel Figure C.6).

Some of the muxes in the datapath must be used to enable the addition needed for ADD to occur. The DRMUX must select its IR[11:9] input in order to write to the destination register specified by the ADD instruction. Similarly, the SR1MUX must select its IR[8:6] input in order to pass the first source register specified by the ADD to the ALU as input A (see the datapath figure). SR2MUX is always controlled by the mode bit IR[5], so the control unit does not need to generate anything (note that this signal was not in the list given earlier). The rest of the muxes in the datapath—PCMUX, ADDR1MUX, ADDR2MUX, and MARMUX—do not matter, and the signals controlling them are don't cares. For example, since the PC does not change, the output of the PCMUX is simply discarded, thus which input the PCMUX forwards to its output cannot matter.

The ALU must perform an addition, so we must set the operation type ALUK appropriately. And memory should not be enabled (MIO.EN=0), in which case the read/write control for memory, R.W, is a don't care. These 25 signal values together (including seven don't cares) implement the RTL for the single state of execution for an ADD instruction.

4.1.3 Example Control Word: LDR

As a second example, consider the first state in the sequence that implements the LDR instruction—state number 6 in the figure on the previous page. The RTL for the state is: MAR ← BaseR + off6, but BaseR is abbreviated to "B" in the state diagram.

What is the control word for this state? Let's again begin with the register load control signals. Only the MAR is written by the RTL, so we need LD.MAR=1 and the other load signals all equal to 0.

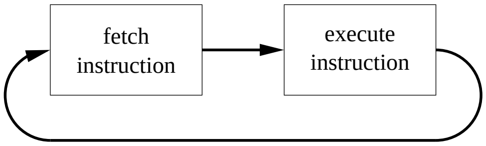
The address (BaseR + off6) is generated by the address adder, then passes through the MARMUX to the bus, from which it can be written into the MAR. To allow the MARMUX to write to the bus, we set GateMARMUX high and set the other three Gate control signals low.

More of the muxes are needed for this state's RTL than we needed for the ADD execution state's RTL. The SR1MUX must again select its IR[8:6] input, this time in order to pass the BaseR specified by the instruction to ADDR1MUX. ADDR1MUX must then select the output of the register file in order to pass the BaseR to the address adder. The other input of the address adder should be off6, which corresponds to the sign-extended version of IR[5:0]. ADDR2MUX must select this input to pass to the address adder. Finally, the MARMUX must select the output of the address adder. The PCMUX and DRMUX do not matter for this case, and can be left as don't cares. Neither the PC nor any register in the register file is written.

The output of the ALU is not used, so which operation it performs is irrelevant, and the ALUK controls are also don't cares. Memory is also not used, so again we set MIO.EN=0. And, as before, the R.W control for memory is a don't care. These 25 signal values together (again including seven don't cares) implement the RTL for the first state of LDR execution.

4.1.4 Hardwired Control

Now we are ready to think about how control signals can be generated. As illustrated to the right, instruction processing consists of two steps repeated infinitely: fetch an instruction, then execute the instruction.



Let's say that we choose a fixed number of cycles for each of these two steps. We can then control our system with a counter, using the counter's value and the IR to generate the control signals through combinational logic. The PC is used only as data and has little or no direct effect on how the system fetches and processes instructions.[14] This approach in general is called **hardwired control**.

How many cycles do we need for instruction fetch? How many cycles do we need for instruction processing? The answers depend on the factors: the complexity of the ISA, and the complexity of the datapath.

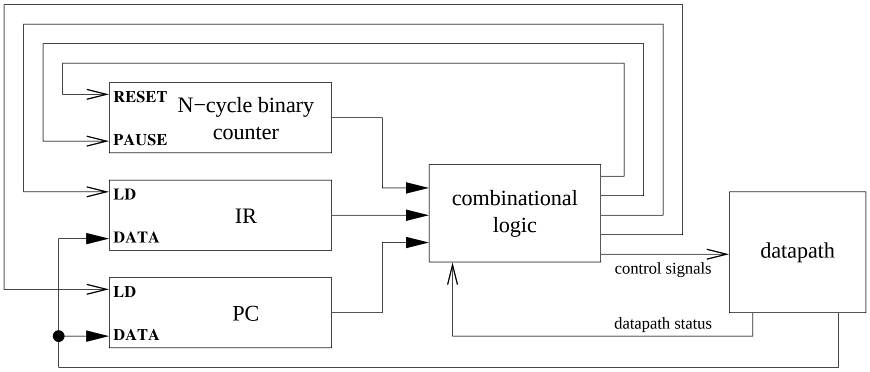
Given a simple ISA, we can design a datapath that is powerful enough to process any instruction in a single cycle. The control unit for such a design is an example of **single-cycle, hardwired control**. While this approach simplifies the control unit, the cycle time for the clock is limited by the slowest instruction. The clock must also be slow enough to let memory operations complete in single cycle, both for instruction fetch and for instructions that require a memory access. Such a low clock rate is usually not acceptable.

More generally, we can use a simpler datapath and break both instruction fetch and instruction processing into multiple steps. Using the datapath in the figure from Patt and Patel, for example, instruction fetch requires three steps, and the number of steps for instruction processing depends on the type of instruction being processed. The state diagram shown earlier illustrates the steps for each opcode.

Although the datapath is not powerful enough to complete instructions in a single cycle, we can build a **multi-cycle, hardwired control** unit (one based on combinational logic). In fact, this type of control unit is not much more complex than the single-cycle version. For the control unit's FSM, we can use a binary counter to enumerate first the steps of fetch, then the steps of processing. The counter value along with the IR register can drive combinational logic to produce control signals. And, to avoid processing at the speed of the slowest instruction (the opcode that requires the largest number of steps; LDI and STI in the LC-3 state diagram), we can add a reset signal to the counter to force it back to instruction fetch. The FSM counter reset signal is simply another control signal. Finally, we can add one more signal that pauses the counter while waiting for a memory operation to complete. The system clock can then run at the speed of the logic rather than at the speed of memory. The control unit implementation discussed in Appendix C of Patt and Patel is not hardwired, but it does make use of a memory ready signal to achieve this decoupling between the clock speed of the processor and the access time of the memory.

[14] Some ISAs do split the address space into privileged and non-privileged regions, but we ignore that possibility here.

The figure to the right illustrates a general multi-cycle, hardwired control unit. The three blocks on the left are the control unit state. The combinational logic in the middle uses the control unit state along with some datapath status bits to compute the control signals for the datapath and the extra controls needed for the FSM counter, IR, and PC. The datapath appears to the right in the figure.



How complex is the combinational logic? As mentioned earlier, we assume that the PC does not directly affect control. But we still have 16 bits of IR, the FSM counter state, and the datapath status signals. Perhaps we need 24-variable K-maps? Here's where engineering and human design come to the rescue: by careful design of the ISA and the encoding, the authors have made many of the datapath control signals for the LC-3 ISA quite simple. For example, the register that appears on the register file's SR2 output is always specified by IR[2:0]. The SR1 output requires a mux, but the choices are limited to IR[11:9] and IR[8:6] (and R6 in the design with support for interrupts). Similarly, the destination register in the register file is always R7 or IR[11:9] (or, again, R6 when supporting interrupts). The control signals for an LC-3 datapath depend almost entirely on the state of the control unit FSM (counter bits in a hardwired design) and the opcode IR[15:12]. The control signals are thus reduced to fairly simple functions.

Let's imagine building a hardwired control unit for the LC-3. Let's start by being more precise about the number of inputs to the combinational logic. Although most decisions are based on the opcode, the datapath and state diagram shown earlier for the LC-3 ISA do have one instance of using another instruction bit to determine behavior. Specifically, the JSR instruction has two modes, and the control unit uses IR[11] to choose between them. So we need to have five bits of IR instead of four as input to our logic.

How many datapath status signals are needed? When the control unit accesses memory, it must wait until the memory finishes the access, as indicated by a memory ready signal R. And the control unit must implement the conditional part of conditional branches, for which it uses the datapath's branch enable signal BEN. These two datapath status signals suffice for our design.

How many bits do we need for the counter? Instruction fetch requires three cycles: one to move the PC to the MAR and increment the PC, a second to read from memory into MDR, and a third to move the instruction bits across the bus from MDR into IR. Instruction decoding in a hardwired design is implicit and requires no cycles: since all of our control signals can depend on the IR, we do not need a cycle to change the FSM state to reflect the opcode. Looking at the LC-3 state diagram, we see that processing an instruction requires at most five cycles. In total, at most eight steps are needed to fetch and process any LC-3 instruction, so we can use a 3-bit binary counter.

We thus have a total of ten bits of input: IR[15:11], R, BEN, and a 3-bit counter. Adding the RESET and PAUSE controls for our FSM counter to the 25 control signals listed earlier, we need to find 27 functions on 10 variables. That's still a lot of big K-maps to solve. Is there an easier way?

4.1.5 Using a Memory for Logic Functions

Consider a case in which you need to compute many functions on a small number of bits, such as we just described for the multi-cycle, hardwired control unit. One strategy is to use a memory (possibly a read-only memory). A $2^m \times N$ memory can be viewed as computing N arbitrary functions on m variables. The functions to be computed are specified by filling in the bits of the memory. So long as the value of m is fairly small, the memory (especially SRAM) can be fast.

Synthesis tools (or hard work) can, of course, produce smaller designs that use fewer gates. Actually, tools may be able to optimize a fixed design expressed as read-only memory, too. But designing the functions with a memory makes them easier to modify later. If we make a mistake, for example, in computing one of the functions, we need only change a bit or two in the memory instead of solving equations and reoptimizing and replacing logic gates. We can also extend our design if we have space remaining (that is, if the functions are undefined for some combinations of the m inputs). The Cray T3D supercomputer, for example, used a similar approach to add new instructions to the Alpha processors on which it was based.

This strategy is effective in many contexts, so let's briefly discuss two analogous cases. In software, a memory becomes a lookup table. Before handheld calculators, lookup tables were used by humans to compute transcendental functions such as sines, cosines, logarithms. Computer graphics hardware and software used a similar approach for transcendental functions in order to reduce cost and improve speed. Functions such as counting 1 bits in a word are useful for processor scheduling and networking, but not all ISAs provide this type of instruction. In such cases, lookup tables in software are often the best solution.

In programmable hardware such as Field Programmable Gate Arrays (FPGAs), lookup tables (called LUTs in this context) have played an important role in implementing arbitrary logic functions. The FPGA is the modern form of the programmable logic array (PLA) mentioned in the textbook, and will be your main tool for developing digital hardware in ECE385. For many years, FPGAs served as a hardware prototyping platform, but many companies today ship their first round products using designs mapped to FPGAs. Why? Chips are more and more expensive to design, and mistakes are costly to fix. In contrast, while companies pay more to buy an FPGA than to produce a chip (after the first chip!), errors in the design can usually be fixed by sending customers a new version through the Internet.

Let's return to our LC-3 example. Instead of solving the K-maps, we can use a small memory: $2^{10} \times 27$ bits (27,648 bits total). We just need calculate the bits, put them into the memory, and use the memory to produce the control signals. The "address" input to the memory are the same 10 bits that we needed for our combinational logic: IR[15:11], R, BEN, and the FSM counter. The data outputs of the memory are the control signals and the RESET and PAUSE inputs to the FSM counter. And we're done.

We can do a little better, though. The datapath in the textbook was designed to work with the textbook's control unit. If we add a little logic, we can significantly simplify our memory-based, hardwired implementation. For example, we only need to pause the FSM counter when waiting for memory. If we can produce a control signal that indicates a need to wait for memory, say WAIT-MEM, we can use a couple of gates to compute the FSM counter's PAUSE signal as WAIT-MEM AND (NOT R). Making this change shrinks our memory to $2^9 \times 27$ bits. The extra two control signals in this case are RESET and WAIT-MEM.

Next, look at how BEN is used in the state diagram: the only use is to terminate the processing of branch instructions when no branch should occur (when BEN=0). We can fold that functionality into the FSM counter's RESET signal by producing a branch reset signal, BR-RESET, to reset the counter to end a branch and a second signal, INST-DONE, when an instruction is done. The RESET input for the FSM counter is then (BR-RESET AND (NOT BEN)) OR INST-DONE. And our memory further shrinks to $2^8 \times 28$ bits, where the extra three control signals are WAIT-MEM, BR-RESET, and INST-DONE.

Finally, recall that the only need for IR[11] is to implement the two forms of JSR. But we can add wires to connect SR1 to PCMUX's fourth input, then control the PCMUX output selection using IR[11] when appropriate (using another control signal). With this extension, we can implement both forms with a single state, writing to both R7 and PC in the same cycle. Our final memory can then be $2^7 \times 29$ bits (3,712 bits total), which is less than one-seventh the number of bits that we needed before modifying the datapath.

4.1.6 Microprogrammed Control

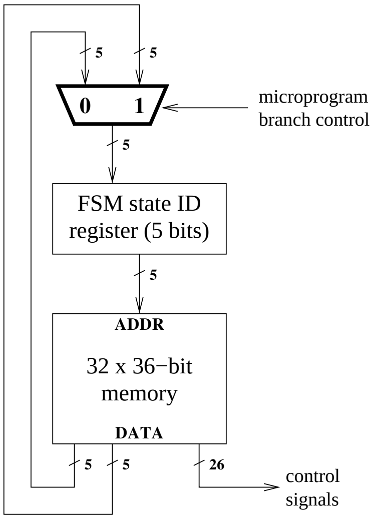
We are now ready to discuss the second approach to control unit design. Take another look at the state diagram for the the LC-3 ISA. Does it remind you of anything? Like a flowchart, it has relatively few arcs leaving each state—usually only one or two.

What if we treat the state diagram as a program? We can use a small memory to hold **microinstructions** (another name for control words) and use the FSM state number as the memory address. Without support for interrupts or privilege, and with the datapath extension for JSR mentioned for hardwired control, the LC-3 state machine requires fewer than 32 states. The datapath has 25 control signals, but we need one more for the datapath extension for JSR. We thus start with 5-bit state number (in a register) and a $2^5 \times 26$ bit memory, which we call our control ROM (read-only memory) to distinguish it from the big, slow, von Neumann memory. Each cycle, the **microprogrammed control** unit applies the FSM state number to the control ROM (no IR bits, just the state number), gets back a set of control signals, and uses them to drive the datapath.

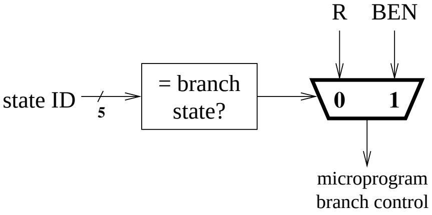
To write our microprogram, we need to calculate the control signals for each microinstruction and put them in the control ROM, but we also need to have a way to decide which microinstruction should execute next. We call the latter problem **sequencing** or microsequencing.

Notice that most of the time there's no choice: we have only *one* next microinstruction. One simple approach is then to add the address (the 5-bit state ID) of the next microinstruction to the control ROM. Instead of 26 bits per FSM state, we now have 31 bits per FSM state.

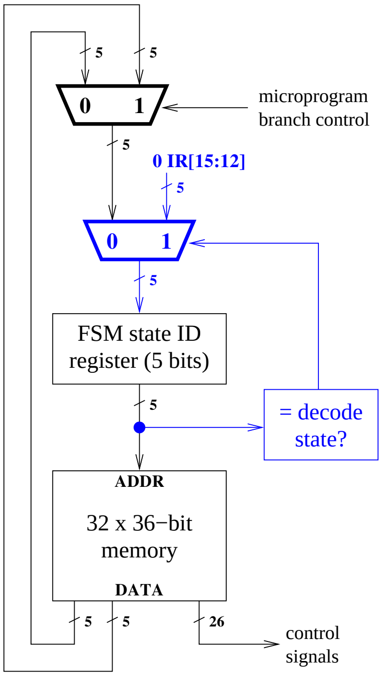
Sometimes we do need to have two possible next states. When waiting for memory (the von Neumann memory, not the control ROM) to finish an access, for example, we want our FSM to stay in the same state, then move to the next state when the access completes. Let's add a second address to each microinstruction, and add a branch control signal for the microprogram to decide whether we should use the first address or the second for the next microinstruction. This design, using a $2^5 \times 36$ bit memory (1,152 bits total), appears to the right.



The microprogram branch control signal is a Boolean logic expression based on the memory ready signal R and IR[11]. We can implement it with a state ID comparison and a mux, as shown to the right. For the branch instruction execution state, the mux selects input 1, BEN. For all other states, the mux selects input 0, R. When an FSM state has only a single next state, we set both IDs in the control ROM to the ID for that state, so the value of R has no effect.



What's missing? Decode! We do have one FSM state in which we need to be able to branch to one of sixteen possible next states, one for each opcode. Let's just add another mux and choose the state IDs for starting to process each opcode in an easy way, as shown to the right with extensions highlighted in blue. The textbook assigns the first state for processing each opcode the IDs IR[15:12] preceeded by two 0s (the textbook's design requires 6-bit state IDs). We adopt the same strategy. For example, the first FSM state for processing an ADD is 00001, and the first state for a TRAP is 01111. In each case, the opcode encoding specifies the last four bits.



Now we can pick the remaining state IDs arbitrarily and fill in the control ROM with control signals that implement each state's RTL and the two possible next states. Transitions from the decode state are handled by the extra mux.

The microprogrammed control unit implementation in Appendix C of Patt and Patel is similar to the one that we have developed here, but is slightly more complex so that it can handle interrupts and privilege.

ECE120: Introduction to Computer Engineering

Notes Set 4.2 Redundancy and Coding

This set of notes introduces the idea of using sparsely populated representations to protect against accidental changes to bits. Today, such representations are used in almost every type of storage system, from bits on a chip to main memory to disk to archival tapes. We begin our discussion with examples of representations in which some bit patterns have no meaning, then consider what happens when a bit changes accidentally. We next outline a general scheme that allows a digital system to detect a single bit error. Building on the mechanism underlying this scheme, we describe a distance metric that enables us to think more broadly about both detecting and correcting such errors, and then show a general approach that allows correction of a single bit error. We leave discussion of more sophisticated schemes to classes on coding and information theory.

4.2.1 Sparse Representations

Representations used by computers must avoid ambiguity: a single bit pattern in a representation cannot be used to represent more than one value. However, the converse need not be true. A representation can have several bit patterns representing the same value, and not all bit patterns in a representation need be used to represent values.

Let's consider a few example of representations with unused patterns. Historically, one common class of representations of this type was those used to represent individual decimal digits. We examine three examples from this class.

The first is Binary-coded Decimal (BCD), in which decimal digits are encoded individually using their representations in the unsigned (binary) representation. Since we have 10 decimal digits, we need 10 patterns, and four bits for each digit. But four bits allow $2^4 = 16$ bit patterns. In BCD, the patterns $1010, 1011, \ldots, 1111$ are unused. It is important to note that BCD is not the same as the unsigned representation. The decimal number 732, for example, requires 12 bits when encoded as BCD: 0111 0011 0010. When written using a 12-bit unsigned representation, 732 is written 001011011100. Operations on BCD values were implemented in early processors, including the 8086, and are thus still available in the x86 instruction set architecture today!

The second example is an Excess-3 code, in which each decimal digit d is represented by the pattern corresponding to the 4-bit unsigned pattern for $d + 3$. For example, the digit 4 is represented as 0111, and the digit 7 is represented as 1010. The Excess-3 code has some attractive aspects when using simple hardware. For example, we can use a 4-bit binary adder to add two digits c and d represented in the Excess-3 code, and the carry out signal produced by the adder is the same as the carry out for the decimal addition, since $c+d \geq 10$ is equivalent to $(c + 3) + (d + 3) \geq 16$.

The third example of decimal digit representations is a 2-out-of-5 code. In such a code, five bits are used to encode each digit. Only patterns with exactly two 1s are used. There are exactly ten such patterns, and an example representation is shown to the right (more than one assignment of values to patterns has been used in real systems).

digit	a 2-out-of-5 representation
1	00011
2	00101
3	00110
4	01001
5	01010
6	01100
7	10001
8	10010
9	10100
0	11000

4.2.2 Error Detection

Errors in digital systems can occur for many reasons, ranging from cosmic ray strikes to defects in chip fabrication to errors in the design of the digital system. As a simple model, we assume that an error takes the form of changes to some number of bits. In other words, a bit that should have the value 0 instead has the value 1, or a bit that should have the value 1 instead has the value 0. Such an error is called a **bit error**.

Digital systems can be designed with or without tolerance to errors. When an error occurs, no notification nor identification of the error is provided. Rather, if error tolerance is needed, the system must be designed to be able to recognize and identify errors automatically. Often, we assume that each of the bits may be in error independently of all of the others, each with some low probability. With such an assumption, multiple bit errors are much less likely than single bit errors, and we can focus on designs that tolerate a single bit error. When a bit error occurs, however, we must assume that it can happen to any of the bits.

The use of many patterns to represent a smaller number of values, as is the case in a 2-out-of-5 code, enables a system to perform **error detection**. Let's consider what happens when a value represented using a 2-out-of-5 code is subjected to a single bit error. Imagine that we have the digit 7. In the table on the previous page, notice that the digit 7 is represented with the pattern 10001.

As we mentioned, we must assume that the bit error can occur in any of the five bits, thus we have five possible bit patterns after the error occurs. If the error occurs in the first bit, we have the pattern 00001. If the error occurs in the second bit, we have the pattern 11001. The complete set of possible error patterns is 00001, 11001, 10101, 10011, and 10000.

Notice that none of the possible error patterns has exactly two 1s, and thus none of them is a meaningful pattern in our 2-out-of-5 code. In other words, whenever a digital system represents the digit 7 and a single bit error occurs, the system will be able to detect that an error has occurred.

What if the system needs to represent a different digit? Regardless of which digit is represented, the pattern with no errors has exactly two 1s, by the definition of our representation. If we then flip one of the five bits by subjecting it to a bit error, the resulting error pattern has either one 1 (if the bit error changes a 1 to a 0) or three 1s (if the bit error changes a 0 to a 1). In other words, regardless of which digit is represented, and regardless of which bit has an error, the resulting error pattern never has a meaning in the 2-out-of-5 code. So this representation enables a digital system to detect any single bit error!

4.2.3 Parity

The ability to detect any single bit error is certainly useful. However, so far we have only shown how to protect ourselves when we want to represent decimal digits. Do we need to develop a separate error-tolerant representation for every type of information that we might want to represent? Or can we instead come up with a more general approach? The answer to the second question is yes: we can, in fact, systematically transform any representation into a representation that allows detection of a single bit error. The key to this transformation is the idea of **parity**.

Consider an arbitrary representation for some type of information. For each pattern used in the representation, we can count the number of 1s. The resulting count is either odd or even. By adding an extra bit—called a **parity bit**—to the representation, and selecting the parity bit's value appropriately for each bit pattern, we can ensure that the count of 1s is odd (called **odd parity**) or even (called **even parity**) for all values represented. The idea is illustrated in the table to the right for the 3-bit unsigned representation. The parity bits are shown in bold.

value represented	3-bit unsigned	number of 1s	with odd parity	with even parity
0	000	0	000**1**	000**0**
1	001	1	001**0**	001**1**
2	010	1	010**0**	010**1**
3	011	2	011**1**	011**0**
4	100	1	100**0**	100**1**
5	101	2	101**1**	101**0**
6	110	2	110**1**	110**0**
7	111	3	111**0**	111**1**

Either approach to selecting the parity bits ensures that any single bit error can be detected. For example, if we choose to use odd parity, a single bit error changes either a 0 into a 1 or a 1 into a 0. The number of 1s in the resulting error pattern thus differs by exactly one from the original pattern, and the parity of the error pattern is even. But all valid patterns have odd parity, so any single bit error can be detected by simply counting the number of 1s.

4.2.4 Hamming Distance

Next, let's think about how we might use representations—we might also think of them as **codes**—to protect a system against multiple bit errors. As we have seen with parity, one strategy that we can use to provide such error tolerance is the use of representations in which only some of the patterns actually represent values. Let's call such patterns **code words**. In other words, the code words in a representation are those patterns that correspond to real values of information. Other patterns in the representation have no meaning.

As a tool to help us understand error tolerance, let's define a measure of the distance between code words in a representation. Given two code words X and Y, we can calculate the number $N_{X,Y}$ of bits that must change to transform X into Y. Such a calculation merely requires that we compare the patterns bit by bit and count the number of places in which they differ. Notice that this relationship is symmetric: the same number of changes are required to transform Y into X, so $N_{Y,X} = N_{X,Y}$. We refer to this number $N_{X,Y}$ as the **Hamming distance** between code word X and code word Y. The metric is named after Richard Hamming, a computing pioneer and an alumnus of the UIUC Math department.

The Hamming distance between two code words tells us how many bit errors are necessary in order for a digital system to mistake one code word for the other. Given a representation, we can calculate the minimum Hamming distance between any pair of code words used by the representation. The result is called the **Hamming distance of the representation**, and represents the minimum of bit errors that must occur before a system might fail to detect errors in a stored value.

The Hamming distance for nearly all of the representations that we introduced in earlier sections is 1. Since more than half of the patterns (and often all of the patterns!) correspond to meaningful values, some pairs of code words must differ in only one bit, and these representations cannot tolerate any errors. For example, the decimal value 42 is stored as 101010 using a 6-bit unsigned representation, but any bit error in that pattern produces another valid pattern corresponding to one of the following decimal numbers: 10, 58, 34, 46, 40, 43. Note that the Hamming distance between any two patterns is not necessarily 1. Rather, the Hamming distance of the unsigned representation, which corresponds to the minimum between any pair of valid patterns, is 1.

In contrast, the Hamming distance of the 2-out-of-5 code that we discussed earlier is 2. Similarly, the Hamming distance of any representation extended with a parity bit is at least 2.

Now let's think about the problem slightly differently. Given a particular representation, how many bit errors can we detect in values using that representation? *A representation with Hamming distance d can detect up to $d-1$ bit errors.* To understand this claim, start by selecting a code word from the representation and changing up to $d-1$ of the bits. No matter how one chooses to change the bits, these changes cannot result in another code word, since we know that any other code word has to require at least d changes from our original code word, by the definition of the representation's Hamming distance. A digital system using the representation can thus detect up to $d-1$ errors. However, if d or more errors occur, the system might sometimes fail to detect any error in the stored value.

4.2.5 Error Correction

Detection of errors is important, but may sometimes not be enough. What can a digital system do when it detects an error? In some cases, the system may be able to find the original value elsewhere, or may be able to re-compute the value from other values. In other cases, the value is simply lost, and the digital system may need to reboot or even shut down until a human can attend to it. Many real systems cannot afford such a luxury. Life-critical systems such as medical equipment and airplanes should not turn themselves off and wait for a human's attention. Space vehicles face a similar dilemma, since no human may be able to reach them.

Can we use a strategy similar to the one that we have developed for error detection in order to try to perform **error correction**, recovering the original value? Yes, but the overhead—the extra bits that we need to provide such functionality—is higher.

Let's start by thinking about a code with Hamming distance 2, such as 4-bit 2's complement with odd parity. We know that such a code can detect one bit error. Can it correct such a bit error, too?

Imagine that a system has stored the decimal value 6 using the pattern 0110**1**, where the last bit is the odd parity bit. A bit error occurs, changing the stored pattern to 0111**1**, which is not a valid pattern, since it has an even number of 1s. But can the system know that the original value stored was 6? No, it cannot. The original value may also have been 7, in which case the original pattern was 0111**0**, and the bit error occurred in the final bit. The original value may also have been -1, 3, or 5. The system has no way of resolving this ambiguity. The same problem arises if a digital system uses a code with Hamming distance d to detect up to $d-1$ errors.

Error correction is possible, however, if we assume that fewer bit errors occur (or if we instead use a representation with a larger Hamming distance). As a simple example, let's create a representation for the numbers 0 through 3 by making three copies of the 2-bit unsigned representation, as shown to the right. The Hamming distance of the resulting code is 3, so any two bit errors can be detected. However, this code also enables us to correct a single bit error. Intuitively, think of the three copies as voting on the right answer. Since a single bit error can only corrupt one copy, a majority vote always gives the right answer! Tripling the number of bits needed in a representation is not a good general strategy, however. Notice also that "correcting" a pattern with two bit errors can produce the wrong result.

value represented	three-copy code
0	000000
1	010101
2	101010
3	111111

Let's think about the problem in terms of Hamming distance. Assume that we use a code with Hamming distance d and imagine that up to k bit errors affect a stored value. The resulting pattern then falls within a neighborhood of distance k from the original code word. This neighborhood contains all bit patterns within Hamming distance k of the original pattern. We can define such a neighborhood around each code word. Now, since d bit errors are needed to transform a code word into any other code word, these neighborhoods are disjoint so long as $2k \leq d-1$. In other words, if the inequality holds, any bit pattern in the representation can be in at most one code word's neighborhood. The digital system can then correct the errors by selecting the unique value identified by the associated neighborhood. Note that patterns encountered as a result of up to k bit errors always fall within the original code word's neighborhood; the inequality ensures that the neighborhood identified in this way is unique. We can manipulate the inequality to express the number of errors k that can be corrected in terms of the Hamming distance d of the code. *A code with Hamming distance d allows up to $\lfloor \frac{d-1}{2} \rfloor$ errors to be corrected*, where $\lfloor x \rfloor$ represents the integer floor function on x, or rounding x down to the nearest integer.

4.2.6 Hamming Codes

Hamming also developed a general and efficient approach for extending an arbitrary representation to allow correction of a single bit error. The approach yields codes with Hamming distance 3. To understand how a **Hamming code** works, think of the bits in the representation as being numbered starting from 1. For example, if we have seven bits in the code, we might write a bit pattern X as $x_7x_6x_5x_4x_3x_2x_1$.

The bits with indices that are powers of two are parity check bits. These include x_1, x_2, x_4, x_8, and so forth. The remaining bits can be used to hold data. For example, we could use a 7-bit Hamming code and map the bits from a 4-bit unsigned representation into bits x_7, x_6, x_5, and x_3. Notice that Hamming codes are not so useful for small numbers of bits, but require only logarithmic overhead for large numbers of bits. That is, in an N-bit Hamming code, only $\lceil \log_2(N+1) \rceil$ bits are used for parity checks.

How are the parity checks defined? Each parity bit is used to provide even parity for those bits with indices for which the index, when written in binary, includes a 1 in the single position in which the parity bit's index contains a 1. The x_1 bit, for example, provides even parity on all bits with odd indices. The x_2 bit provides even parity on x_2, x_3, x_6, x_7, x_{10}, and so forth.

In a 7-bit Hamming code, for example, x_1 is chosen so that it has even parity together with x_3, x_5, and x_7. Similarly, x_2 is chosen so that it has even parity together with x_3, x_6, and x_7. Finally, x_4 is chosen so that it has even parity together with x_5, x_6, and x_7.

The table to the right shows the result of embedding a 4-bit unsigned representation into a 7-bit Hamming code.

	4-bit				7-bit
value	unsigned				Hamming
represented	($x_7x_6x_5x_3$)	x_4	x_2	x_1	code
0	0000	0	0	0	0000000
1	0001	0	1	1	0000111
2	0010	1	0	1	0011001
3	0011	1	1	0	0011110
4	0100	1	1	0	0101010
5	0101	1	0	1	0101101
6	0110	0	1	1	0110011
7	0111	0	0	0	0110100
8	1000	1	1	1	1001011
9	1001	1	0	0	1001100
10	1010	0	1	0	1010010
11	1011	0	0	1	1010101
12	1100	0	0	1	1100001
13	1101	0	1	0	1100110
14	1110	1	0	0	1111000
15	1111	1	1	1	1111111

A Hamming code provides a convenient way to identify which bit should be corrected when a single bit error occurs. Notice that each bit is protected by a unique subset of the parity bits corresponding to the binary form of the bit's index. Bit x_6, for example, is protected by bits x_4 and x_2, because the number 6 is written 110 in binary. If a bit is affected by an error, the parity bits that register the error are those corresponding to 1s in the binary number of the index. So if we calculate check bits as 1 to represent an error (odd parity) and 0 to represent no error (even parity), then concatenate those bits into a binary number, we obtain the binary value of the index of the single bit affected by an error (or the number 0 if no error has occurred).

Let's do a couple of examples based on the pattern for the decimal number 9, 1001100. First, assume that no error occurs. We calculate check bit c_4 by checking whether x_4, x_5, x_6, and x_7 together have even parity. Since no error occurred, they do, so $c_4 = 0$. Similarly, for c_2 we consider x_2, x_3, x_6, and x_7. These also have even parity, so $c_2 = 0$. Finally, for c_1, we consider x_1, x_3, x_5, and x_7. As with the others, these together have even parity, so $c_1 = 0$. Writing $c_4c_2c_1$, we obtain 000, and conclude that no error has occurred.

Next assume that bit 3 has an error, giving us the pattern 1001000. In this case, we have again that $c_4 = 0$, but the bits corresponding to both c_2 and c_1 have odd parity, so $c2 = 1$ and $c_1 = 1$. Now when we write the check bits $c_4c_2c_1$, we obtain 011, and we are able to recognize that bit 3 has been changed.

A Hamming code can only correct one bit error, however. If two bit errors occur, correction will produce the wrong answer. Let's imagine that both bits 3 and 5 have been flipped in our example pattern for the decimal number 9, producing the pattern 1011000. Calculating the check bits as before and writing them as $c_4c_2c_1$, we obtain 110, which leads us to incorrectly conclude that bit 6 has been flipped. As a result, we "correct" the pattern to 1111000, which represents the decimal number 14.

4.2.7 SEC-DED Codes

We now consider one final extension of Hamming codes to enable a system to perform single error correction while also detecting any two bit errors. Such codes are known as **Single Error Correction, Double Error Detection (SEC-DED)** codes. Creating such a code from a Hamming code is trivial: add a parity bit covering the entire Hamming code. The extra parity bit increases the Hamming distance to 4. A Hamming distance of 4 still allows only single bit error correction, but avoids the problem of Hamming distance 3 codes when two bit errors occur, since patterns at Hamming distance 2 from a valid code word cannot be within distance 1 of another code word, and thus cannot be "corrected" to the wrong result.

In fact, one can add a parity bit to any representation with an odd Hamming distance to create a new representation with Hamming distance one greater than the original representation. To proof this convenient fact, begin with a representation with Hamming distance d, where d is odd. If we choose two code words from the representation, and their Hamming distance is already greater than d, their distance in the new representation will also be greater than d. Adding a parity bit cannot decrease the distance. On the other hand, if the two code words are exactly distance d apart, they must have opposite parity, since they differ by an odd number of bits. Thus the new parity bit will be a 0 for one of the code words and a 1 for the other, increasing the Hamming distance to $d + 1$ in the new representation. Since all pairs of code words have Hamming distance of at least $d + 1$, the new representation also has Hamming distance $d + 1$.

ECE120: Introduction to Computer Engineering

Notes Set 4.3 Instruction Set Architecture*

This set of notes discusses tradeoffs and design elements of instruction set architectures (ISAs). *The material is beyond the scope of our class, and is provided purely for your interest.* Those who find these topics interesting may also want to read the ECE391 notes, which describe similar material with a focus on the x86 ISA.

As you know, the ISA defines the interface between software and hardware, abstracting the capabilities of a computer's datapath and standardizing the format of instructions to utilize those capabilities. Successful ISAs are rarely discarded, as success implies the existence of large amounts of software built to use the ISA. Rather, they are extended, and their original forms must be supported for decades (consider, for example, the IBM 360 and the Intel x86). Employing sound design principles is thus imperative in an ISA.

4.3.1 Formats and Fields*

The LC-3 ISA employs fixed-length instructions and a load-store architecture, two aspects that help to reduce the design space to a manageable set of choices. In a general ISA design, many other options exist for instruction formats.

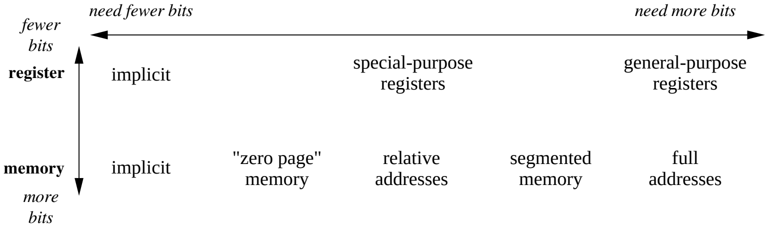
Recall the idea of separating the bits of an instruction into (possibly non-contiguous) fields. One of the fields must contain an opcode, which specifies the type of operation to be performed by the instruction. In the LC-3 ISA, most opcodes specify both the type of operation and the types of arguments to the operation. More generally, many addressing modes are possible for each operand, and we can think of the bits that specify the addressing mode as a separate field, known as the **mode** field. As a simple example, the LC-3's ADD and AND instructions contain a 1-bit mode field that specifies whether the second operand of the ADD/AND comes from a register or is an immediate value.

Several questions must be answered in order to define the possible instruction formats for an ISA. First, are instructions fixed-length or variable-length? Second, how many addresses are needed for each instruction, and how many of the addresses can be memory addresses? Finally, what forms of addresses are possible for each operand? For example, can one use full memory addresses or only limited offsets relative to a register?

The answer to the first question depends on many factors, but several clear advantages exist for both answers. **Fixed-length instructions** are easy to fetch and decode. A processor knows in advance how many bits must be fetched to fetch a full instruction; fetching the opcode and mode fields in order to decide how many more bits are necessary to complete the instruction may require more than one cycle. Fixing the time necessary for instruction fetch also simplifies pipelining. Finally, fixed-length instructions simplify the datapath by restricting instructions to the size of the bus and always fetching properly aligned instructions. As an example of this simplification, note that the LC-3 ISA does not support addressing for individual bytes, only for 16-bit words.

Variable-length instructions also have benefits, however. Variable-length encodings allow more efficient encodings, saving both memory and disk space. A register transfer operation, for example, clearly requires fewer bits than addition of values at two direct memory addresses for storage at a third. Fixed-length instructions must be fixed at the length of the longest possible instruction, whereas variable-length instructions can use lengths appropriate to each mode. The same tradeoff has another form in the sense that fixed-length ISAs typically eliminate many addressing modes in order to limit the size of the instructions. Variable-length instructions thus allow more flexibility; indeed, extensions to a variable-length ISA can incorporate new addressing modes that require longer instructions without affecting the original ISA. For example, the maximum length of x86 instructions has grown from six bytes in 1978 (the 8086 ISA) to fifteen bytes in today's version of the ISA.

Moving to the last of the three questions posed for instruction format definition, operand address specification, we explore a range of answers developed over the last few decades. Answers are usually chosen based on the number of bits necessary, and we use this metric to organize the possibilities. The figure below separates approaches into two dimensions: the vertical dimension divides addressing into registers and memory, and the horizontal dimension into varieties within each type.



As a register file contains fewer registers than a memory does words, the use of register operands rather than memory addresses reduces the number of bits required to specify an operand. The LC-3 ISA uses a restricted set of addressing modes to stay within the limit imposed by the use of 16-bit instructions. Both register and memory addresses, however, admit a wide range of implementations.

Implicit operands of either type require no additional bits for the implicit address. The LC-3 procedure call instruction, JSR, for example, stores the return address in R7. No bits in the JSR encoding name the R7 register; R7 is used implicitly for every JSR executed. Similarly, the procedure call instructions in many ISAs push the return address onto a stack using an implicit register for the top of stack pointer. Memory addresses can also be implicitly equated to other memory addresses. An increment instruction operating on a memory address, for example, implicitly writes the result back to the same address.

At the opposite extreme, an instruction may include a full address, either to any register in the register file or to any address in the memory. The term **general-purpose registers** indicates that registers are used in any operation. **Special-purpose registers**, in contrast, split the register file and allow only certain registers to be used in each operation. For example, the Motorola 680x0 series, used in early Apple Macintosh computers, provides distinct sets of address and data registers. Loads and stores use the address registers; arithmetic, logic, and shift operations use the data registers. As a result, each instruction selects from a smaller set of registers and thus requires fewer bits in the instruction to name the register for use.

As full memory addresses require many more bits than full register addresses, a wider range of techniques has been employed to reduce the length. "Zero page" addresses, as defined in the 6510 (6502) ISA used by Commodore PET's,[15] C64's,[16] and VIC 20's, prefixed a one-byte address with a zero byte, allowing shorter instructions when memory addresses fell within the first 256 memory locations. Assembly and machine language programmers made heavy use of these locations to produce shorter programs.

Relative addressing is quite common in the LC-3 ISA, in which many addresses are PC-relative. Typical commerical ISAs also make use of relative addressing. The Alpha ISA, for example, has a PC-relative form of procedure call with a 21-bit offset (plus or minus a megabyte), and the x86 ISA has a "short" form of branch instructions that uses an 8-bit offset.

Segmented memory is a form of relative addressing that uses a register (usually implicit) to provide the high bits of an address and an explicit memory address (or another register) to provide the low bits. In the early x86 ISAs, for example, 20-bit addresses are found by adding a 16-bit segment register extended with four zero bits to a 16-bit offset.

[15] My computer in junior high school.
[16] My computer in high school.

4.3.2 Addressing Architectures*

One question remains for the definition of instruction formats: how many addresses are needed for each instruction, and how many of the addresses can be memory addresses? The first part of this question usually ranges from zero to three, and is rarely allowed to go beyond three. The answer to the second part determines the **addressing architecture** implemented by an ISA. We now illustrate the tradeoffs between five distinct addressing architectures through the use of a running example, the assignment $X = AB + C/D$.

A binary operator requires two source operands and one destination operand, for a total of three addresses. The ADD instruction, for example, has a **3-address** format:

```
      ADD   A,B,C          ; M[A] ← M[B] + M[C]
or    ADD   R1,R2,R3       ; R1 ← R2 + R3
```

If all three addresses can be memory addresses, the ISA is dubbed a **memory-to-memory architecture**. Such architectures may have small register sets or even lack a register file completely. To implement the assignment, we assume the availability of two memory locations, T1 and T2, for temporary storage:

```
MUL   T1,A,B         ; T1 ← M[A] * M[B]
DIV   T2,C,D         ; T2 ← M[C]/M[D]
ADD   X,T1,T2        ; X ← M[T1] + M[T2]
```

The assignment requires only three instructions to implement, but each instruction contains three full memory addresses, and is thus quite long.

At the other extreme is the **load-store architecture** used by the LC-3 ISA. In a load-store architecture, only loads and stores can use memory addresses; all other operations use only registers. As most instructions use only registers, this type of addressing architecture is also called a **register-to-register architecture**. The example assignment translates to the code shown below, which assumes that R1, R2, and R3 are free for use (the instructions are **NOT** LC-3 instructions, but rather a generic assembly language for a load-store architecture).

```
LD    R1,A           ; R1 ← M[A]
LD    R2,B           ; R2 ← M[B]
MUL   R1,R1,R2       ; R1 ← R1 * R2
LD    R2,C           ; R2 ← M[C]
LD    R3,D           ; R3 ← M[D]
DIV   R2,R2,R3       ; R2 ← R2/R3
ADD   R1,R1,R2       ; R1 ← R1 + R2
ST    R1,X           ; M[X] ← R1
```

Eight instructions are necessary, but no instruction requires more than one full memory address, and several use only register addresses, allowing the use of shorter instructions. The need to move data in and out of memory explicitly, however, also requires a reasonably large register set, as is available in the ARM, Sparc, Alpha, and IA-64 ISAs.

Architectures that use other combinations of memory and register addresses with 3-address formats are not named. Unary operators and transfer operators require only one source operand, thus can use a 2-address format (for example, NOT A,B). Binary operations can also use **2-address** format if one operand is implicit, as in the following instructions:

```
      ADD   A,B            ; M[A] ← M[A] + M[B]
or    ADD   R1,B           ; R1 ← R1 + M[B]
```

The second instruction, in which one address is a register and the second is a memory address, defines a **register-memory architecture**. As shown by the code on the next page, such architectures strike a balance between the two architectures just discussed.

```
LD    R1,A       ; R1 ← M[A]
MUL   R1,B       ; R1 ← R1 ∗ M[B]
LD    R2,C       ; R2 ← M[C]
DIV   R2,D       ; R2 ← R2/M[D]
ADD   R1,R2      ; R1 ← R1 + R2
ST    R1,X       ; M[X] ← R1
```

The assignment now requires six instructions using at most one memory address each; like memory-to-memory architectures, register-memory architectures use relatively few registers. Note that two-register operations are also allowed. Intel's x86 ISA is a register-memory architecture.

Several ISAs of the past[17] used a special-purpose register called the accumulator for ALU operations, and are called **accumulator architectures**. The accumulator in such architectures is implicitly both a source and the destination for any such operation, allowing a **1-address** format for instructions, as shown below.

```
    ADD  B       ; ACC ← ACC + M[B]
or  ST   E       ; M[E] ← ACC
```

Accumulator architectures strike the same balance as register-memory architectures, but use fewer registers. Note that memory location X is used as a temporary storage location as well as the final storage location in the following code:

```
LD    A          ; ACC ← M[A]
MUL   B          ; ACC ← ACC ∗ M[B]
ST    X          ; M[X] ← ACC
LD    C          ; ACC ← M[C]
DIV   D          ; ACC ← ACC/M[D]
ADD   X          ; ACC ← ACC + M[X]
ST    X          ; M[X] ← ACC
```

The last addressing architecture that we discuss is rarely used for modern general-purpose processors, but may be familiar to you because of its historical use in scientific and engineering calculators. A **stack architecture** maintains a stack of values and draws all ALU operands from this stack, allowing these instructions to use a **0-address** format. A special-purpose stack pointer (SP) register points to the top of the stack in memory, and operations analogous to load (**push**) and store (**pop**) are provided to move values on and off the stack. To implement our example assignment, we first transform it into postfix notation (also called reverse Polish notation):

```
A B * C D / +
```

The resulting sequence of symbols transforms on a one-to-one basis into instructions for a stack architecture:

```
PUSH  A   ; SP ← SP − 1, M[SP] ← M[A]                    A
PUSH  B   ; SP ← SP − 1, M[SP] ← M[B]                    B        A
MUL       ; M[SP + 1] ← M[SP + 1] ∗ M[SP], SP ← SP + 1   AB
PUSH  C   ; SP ← SP − 1, M[SP] ← M[C]                    C        AB
PUSH  D   ; SP ← SP − 1, M[SP] ← M[D]                    D        C        AB
DIV       ; M[SP + 1] ← M[SP + 1]/M[SP], SP ← SP + 1     C/D      AB
ADD       ; M[SP + 1] ← M[SP + 1] + M[SP], SP ← SP + 1   AB+C/D
POP   X   ; M[X] ← M[SP], SP ← SP + 1
```

The values to the right are the values on the stack, starting with the top value on the left and progressing downwards, *after the completion of each instruction.*

[17] The 6510/6502 as well, if memory serves, as the 8080, Z80, and Z8000, which used to drive parlor video games.

4.3.3 Common Special-Purpose Registers*

This section illustrates the uses of special-purpose registers through a few examples.

The **stack pointer (SP)** points to the top of the stack in memory. Most older architectures support push and pop operations that implicitly use the stack pointer. Modern architectures assign a general-purpose register to be the stack pointer and reference it explicitly, although an assembler may support instructions that appear to use implicit operands but in fact translate to machine instructions with explicit reference to the register defined to be the SP.

The **program counter (PC)** points to the next instruction to be executed. Some modern architectures expose it as a general-purpose register, although its distinct role in the implementation keeps such a model from becoming as common as the use of a general-purpose register for the SP.

The **processor status register (PSR)**, also known as the **processor status word (PSW)**, contains all status bits as well as a mode bit indicating whether the processor is operating in user mode or privileged (operating system) mode. Having a register with this information allows more general access than is possible solely through the use of control flow instructions.

The **zero register** appears in modern architectures of the RISC variety (defined in the next section of these notes). The register is read-only and serves both as a useful constant and as a destination for operations performed only for their side-effects (for example, setting status bits). The availability of a zero register also allows certain opcodes to serve double duty. A register-to-register add instruction becomes a register move instruction when one source operand is zero. Similarly, an immediate add instruction becomes an immediate load instruction when one source operand is zero.

4.3.4 Reduced Instruction Set Computers*

By the mid-1980's, the VAX architecture dominated the workstation and minicomputer markets, which included most universities. Digital Equipment Corporation, the creator of the VAX, was second only to IBM in terms of computer sales. VAXen, as the machines were called, used microprogrammed control units and supported numerous addressing modes as well as complex instructions ranging from "square root" to "find roots of polynomial equation."

The impact of increasingly dense integrated circuit technology had begun to have its effect, however, and in view of increasing processor clock speeds, more and more programmers were using high-level languages rather than writing assembly code. Although assembly programmers often made use of the complex VAX instructions, compilers were usually unable to recognize the corresponding high-level language constructs and thus were unable to make use of the instructions.

Increasing density also led to rapid growth in memory sizes, to the point that researchers began to question the need for variable-length instructions. Recall that variable-length instructions allow shorter codes by providing more efficient instruction encodings. With the trend toward larger memories, code length was less important. The performance advantage of fixed-length instructions, which simplifies the datapath and enables pipelining, on the other hand, was attractive.

Researchers leveraged these ideas, which had been floating around the research community (and had appeared in some commercial architectures) to create **reduced instruction set computers**, or **RISC** machines. The competing VAXen were labeled **CISC** machines, which stands for **complex instruction set computers**.

RISC machines employ fixed-length instructions and a load-store architecture, allowing only a few addressing modes and small offsets. This combination of design decisions enables deep pipelines and multiple instruction issues in a single cycle (termed superscalar implementations), and for years, RISC machines were viewed by many researchers as the proper design for future ISAs. However, companies such as Intel soon learned to pipeline microoperations after decoding instructions, and CISC architectures now offer competitive if not superior performance in comparison with RISC machines. The VAXen are dead, of course,[18] having been replaced by the Alpha, which in turn fell to x86, which is now struggling with ARM to enter the mobile market.

[18] Unless you talk with customer support employees, for whom no machine ever dies.

4.3.5 Procedure and System Calls*

A **procedure** is a sequence of instructions that executes a particular task. Procedures are used as building blocks for multiple, larger tasks. The concept of a procedure is fundamental to programming, and appears in some form in every high-level language as well as in most ISAs. For our purposes, the terms procedure, subroutine, function, and method are synonymous, although they usually have slightly different meanings from the linguistic point of view. Procedure calls are supported through **call** and **return** control flow instructions. The first instruction in the code below, for example, transfers control to the procedure "DoSomeWork," which presumably does some work, then returns control to the instruction following the call.

```
loop:           CALL   DoSomeWork
                CMP    R6,#1          ; compare return value in R6 to 1
                BEQ    loop           ; keep doing work until R6 is not 1

DoSomeWork:     ···                   ; set R6 to 0 when all work is done, 1 otherwise
                RETN
```

The procedure also places a return value in R6, which the instruction following the call compares with immediate value 1. Until the two are not equal (when all work is done), the branch returns control to the call and executes the procedure again.

As you may recall, the call and return use the stack pointer to keep track of nested calls. Sample RTL for these operations appears below.

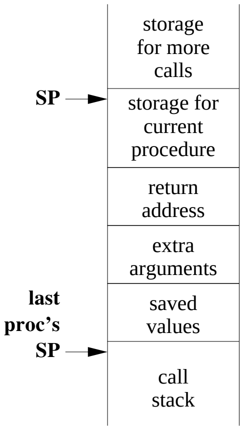
call RTL $SP \leftarrow SP - 1$
$M[SP] \leftarrow PC$
$PC \leftarrow$ procedure start

return RTL $PC \leftarrow M[SP]$
$SP \leftarrow SP + 1$

While an ISA provides the call and return instructions necessary to support procedures, it does not specify how information is passed to or returned from a procedure. A standard for such decisions is usually developed and included in descriptions of the architecture, however. This **calling convention** specifies how information is passed between a caller and a callee. In particular, it specifies the following: where arguments must be placed, either in registers or in specific stack memory locations; which registers can be used or changed by the procedure; and where any return value must be placed.

The term "calling convention" is also used in the programming language community to describe the convention for deciding what information is passed for a given call operation. For example, are variables passed by value, by pointers to values, or in some other way? However, once the things to be sent are decided, the architectural calling convention that we discuss here is used to determine where to put the data in order for the callee to be able to find it.

Calling conventions for architectures with large register sets typically pass arguments in registers, and nearly all conventions place the return value in a register. A calling convention also divides the register set into **caller-saved** and **callee-saved** registers. Caller-saved registers can be modified arbitrarily by the called procedure, whereas any value in a callee-saved register must be preserved. Similarly, before calling a procedure, a caller must preserve the values of any caller saved registers that are needed after the call. Registers of both types usually saved on the stack by the appropriate code (caller or callee).



A typical stack structure appears in the figure to the right. In preparation for a call, a caller first stores any caller-saved registers on the stack. Arguments to the procedure to be called are pushed next. The procedure is called next, implicitly pushing the return address (the address of the instruction following the call instruction). Finally, the called procedure may allocate space on the stack for storage of callee-saved registers as well as local variables.

As an example, the following calling convention can be applied to an 8-register load-store architecture similar to the LC-3 ISA: the first three arguments must be placed in R0 through R2 (in order), with any remaining arguments on the stack; the return value must be placed in R6; R0 through R2 are caller-saved, as is R6, while R3 through R5 are callee-saved; R7 is used as the stack pointer. The code fragments below use this calling convention to implement a procedure and a call of that procedure.

```
int add3 (int n1, int n2, int n3) {      add3:  ADD    R0,R0,R1
    return (n1 + n2 + n3);                      ADD    R6,R0,R2
}                                               RETN
. . .                                           . . .
printf (“%d”, add3 (10, 20, 30));               PUSH   R1         ; save the value in R1
                                                LDI    R0,#10     ; marshal arguments
                                                LDI    R1,#20
                                                LDI    R2,#30
by convention:                                  CALL   add3
     n1 is in R0                                MOV    R1,R6      ; return value becomes 2nd argument
     n2 is in R1                                LDI    R0,“%d”    ; load a pointer to the string
     n3 is in R2                                CALL   printf
     return value is in R6                      POP    R1         ; restore R1
```

The add3 procedure takes three integers as arguments, adds them together, and returns the sum. The procedure is called with the constants 10, 20, and 30, and the result is printed. By the calling convention, when the call is made, R0 must contain the value 10, R1 the value 20, and R2 the value 30. We assume that the caller wants to preserve the value of R1, but does not care about R3 or R5. In the assembly language version on the right, R1 is first saved to the stack, then the arguments are marshaled into position, and finally the call is made. The procedure itself needs no local storage and does not change any callee-saved registers, thus must simply add the numbers together and place the result in R6. After add3 returns, its return value is moved from R6 to R1 in preparation for the call to printf. After loading a pointer to the format string into R0, the second call is made, and R1 is restored, completing the translation.

System calls are almost identical to procedure calls. As with procedure calls, a calling convention is used: before invoking a system call, arguments are marshaled into the appropriate registers or locations in the stack; after a system call returns, any result appears in a pre-specified register. The calling convention used for system calls need not be the same as that used for procedure calls. Rather than a call instruction, system calls are usually initiated with a **trap** instruction, and system calls are also known as traps. With many architectures, a system call places the processor in privileged or kernel mode, and the instructions that implement the call are considered to be part of the operating system. The term system call arises from this fact.

4.3.6 Interrupts and Exceptions*

Unexpected processor interruptions arise both from interactions between a processor and external devices and from errors or unexpected behavior in the program being executed. The term **interrupt** is reserved for asynchronous interruptions generated by other devices, including disk drives, printers, network cards, video cards, keyboards, mice, and any number of other possibilities. **Exceptions** occur when a processor encounters an unexpected opcode or operand. An undefined instruction, for example, gives rise to an exception, as does an attempt to divide by zero. Exceptions usually cause the current program to terminate, although many operating systems will allow the program to catch the exception and to handle it more intelligently. The table below summarizes the characteristics of the two types and compares them to system calls.

type	generated by	example	asynchronous	unexpected
interrupt	external device	packet arrived at network card	yes	yes
exception	invalid opcode or operand	divide by zero	no	yes
trap/system call	deliberate, via trap instruction	print character to console	no	no

Interrupts occur asynchronously with respect to the program. Most designs only recognize interrupts between instructions. In other words, the presence of interrupts is checked only after completing an instruction rather than in every cycle. In pipelined designs, however, instructions execute simultaneously, and the decision as to which instructions occur "before" an interrupt and which occur "after" must be made by the processor. Exceptions are not asynchronous in the sense that they occur for a particular instruction, thus no decision need be made as to instruction ordering. After determining which instructions were before an interrupt, a pipelined processor discards the state of any partially executed instructions that occur "after" the interrupt and completes all instructions that occur "before." The terminated instructions are simply restarted after the interrupt completes. Handling the decision, the termination, and the completion, however, significantly increases the design complexity of the system.

The code associated with an interrupt, an exception, or a system call is a form of procedure called a **handler**, and is found by looking up the interrupt number, exception number, or trap number in a table of functions called a **vector table**. Vector tables for each type (interrupts, exceptions, and system calls) may be separate, or may be combined into a single table. Interrupts and exceptions share a need to save all registers and status bits before execution of the corresponding handler code (and to restore those values afterward). Generally, the values—including the status word register—are placed on the stack. With system calls, saving and restoring any necessary state is part of the calling convention. A special return from interrupt instruction is used to return control from the interrupt handler to the interrupted code; a similar instruction forces the processor back into user mode when returning from a system call.

Interrupts are also interesting in the sense that typical computers often have many interrupt-generating devices but only a few interrupts. Interrupts are prioritized by number, and only an interrupt with higher priority can interrupt another interrupt. Interrupts with equal or lower priority are blocked while an interrupt executes. Some interrupts can also be blocked in some architectures by setting bits in a special-purpose register called an interrupt mask. While an interrupt number is masked, interrupts of that type are blocked, and can not occur.

As several devices may generate interrupts with the same interrupt number, interrupt handlers can be **chained** together. Each handler corresponds to a particular device. When an interrupt occurs, control is passed to the handler for the first device, which accesses device registers to determine whether or not that device generated an interrupt. If it did, the appropriate service is provided. If not, or after the service is complete, control is passed to the next handler in the chain, which handles interrupts from the second device, and so forth until the last handler in the chain completes. At this point, registers and processor state are restored and control is returned to the point at which the interrupt occurred.

4.3.7 Control Flow Conditions*

Control flow instructions may change the PC, loading it with an address specified by the instruction. Although any addressing mode can be supported, the most common specify an address directly in the instruction, use a register as an address, or use an address relative to a register.

Unconditional control flow instructions typically provided by an ISA include procedure calls and returns, traps, and jumps. Conditional control flow instructions are branches, and are logically based on status bits set by two types of instructions: **comparisons** and **bit tests**. Comparisons subtract one value from another to set the status bits, whereas bit tests use an AND operation to check whether certain bits are set or not in a value.

Many ISAs implement status bits as special-purpose registers and implicitly set them when executing certain instructions. A branch based on R2 being less or equal to R3 can then be written as shown below. The status bits are set by subtracting R3 from R2 with the ALU.

```
CMP     R2,R3       ; R2 < R3 : CNZ ← 110, R2 = R3 : CNZ ← 001,
                    ;     R2 > R3 : CNZ ← 000
BLE     R1          ; Z XOR C = 1 : PC ← R1
```

The status bits are not always implemented as special-purpose registers; instead, they may be kept in general-purpose registers or not kept at all. For example, the Alpha ISA stores the results of comparisons in general-purpose registers, and the same branch is instead implemented as follows:

```
CMPLE   R4,R2,R3    ; R2 ≤ R3 : R4 ← 1, R2 > R3 : R4 ← 0
BNE     R4,R1       ; R4 ≠ 0 : PC ← R1
```

Finally, status bits can be calculated, used, and discarded within a single instruction, in which case the branch is written as follows:

```
BLE     R1,R2,R3    ; R2 ≤ R3 : PC ← R1
```

The three approaches have advantages and disadvantages similar to those discussed in the section on addressing architectures: the first has the shortest instructions, the second is the most general and simplest to implement, and the third requires the fewest instructions.

4.3.8 Stack Operations*

Two types of stack operations are commonly supported. Push and pop are the basic operations in many older architectures, and values can be placed upon or removed from the stack using these instructions. In more modern architectures, in which the SP becomes a general-purpose register, push and pop are replaced with indexed loads and stores, that is, loads and stores using the stack pointer and an offset as the address for the memory operation. Stack updates are performed using the ALU, subtracting and adding immediate values from the SP as necessary to allocate and deallocate local storage.

Stack operations serve three purposes in a typical architecture. The first is to support procedure calls, as illustrated in a previous section. The second is to provide temporary storage during interrupts, which was also mentioned earlier.

The third use of stack operations is to support **spill code** generated by compilers. Compilers first translate high-level languages into an intermediate representation much like assembly code but with an extremely large (theoretically infinite) register set. The final translation step translates this intermediate representation into assembly code for the target architecture, assigning architectural registers as necessary. However, as real ISAs support only a finite number of registers, the compiler must occasionally spill values into memory. For example, if ten values are in use at some point in the code, but the architecture has only eight registers, spill code must be generated to store the remaining two values on the stack and to restore them when they are needed.

4.3.9 I/O*

As a final topic, we now consider how a processor connects to other devices to allow input and output. We have already discussed interrupts, which are a special form of I/O in which only the signal requesting attention is conveyed to the processor. Communication of data occurs through instructions similar to loads and stores. A processor is designed with a number of **I/O ports**—usually read-only or write-only registers to which devices can be attached with opposite semantics. That is, a port is usually written by the processor and read by a device or written by a device and read by the processor.

The question of exactly how I/O ports are accessed is an interesting one. One option is to create special instructions, such as the **in** and **out** instructions of the x86 architecture. Port addresses can then be specified in the same way that memory addresses are specified, but use a distinct address space. Just as two sets of special-purpose registers can be separated by the ISA, such an **independent I/O** system separates I/O ports from memory addresses by using distinct instructions for each class of operation.

Alternatively, device registers can be accessed using the same load and store instructions as are used to access memory. This approach, known as **memory-mapped I/O**, requires no new instructions for I/O, but demands that a region of the memory address space be set aside for I/O. The memory words with those addresses, if they exist, can not be accessed during normal processor operations.

ECE120: Introduction to Computer Engineering

Notes Set 4.4 Summary of Part 4 of the Course

With the exception of control unit design strategies and redundancy and coding, most of the material in this part of the course is drawn from Patt and Patel Chapters 4 through 7. You may also want to read Patt and Patel's Appendix C for details of their control unit design.

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.

- von Neumann elements
    - program counter (PC)
    - instruction register (IR)
    - memory address register (MAR)
    - memory data register (MDR)
    - processor datapath
    - bus
    - control signal
    - instruction processing
- Instruction Set Architecture (ISA)
    - instruction encoding
    - field (in an encoded instruction)
    - operation code (opcode)
- assemblers and assembly code
    - opcode mnemonic
      (such as ADD, JMP)
    - two-pass process
    - label
    - symbol table
    - pseudo-op / directive
- systematic decomposition
    - sequential
    - conditional
    - iterative
- control unit design strategies
    - control word / microinstruction
    - sequencing / microsequencing
    - hardwired control
        - single-cycle
        - multi-cycle
    - microprogrammed control
- error detection and correction
    - code/sparse representation
    - code word
    - bit error
    - odd/even parity bit
    - Hamming distance between code words
    - Hamming distance of a code
    - Hamming code
    - SEC-DED

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

- Map RTL (register transfer language) operations into control words for a given processor datapath.
- Systematically decompose a (simple enough) problem to the level of LC-3 instructions.
- Encode LC-3 instructions into machine code.
- Read and understand programs written in LC-3 assembly/machine code.
- Test and debug a small program in LC-3 assembly/machine code.
- Be able to calculate the Hamming distance of a code/representation.
- Know the relationships between Hamming distance and the abilities to detect and to correct bit errors.

We expect that you will understand the concepts and ideas to the extent that you can do the following:

- Explain the role of different types of instructions in allowing a programmer to express a computation.
- Explain the importance of the three types of subdivisions in systematic decomposition (sequential, conditional, and iterative).
- Explain the process of transforming assembly code into machine code (that is, explain how an assembler works, including describing the use of the symbol table).
- Be able to use parity for error detection, and Hamming codes for error correction.

At the highest level, we hope that, while you do not have direct substantial experience in this regard from our class (and should not expect to be tested on these skills), that you will nonetheless be able to begin to do the following when designing combinational logic:

- Design and compare implementations using gates, decoders, muxes, and/or memories as appropriate, and including reasoning about the relevant design tradeoffs in terms of area and delay.
- Design and compare implementation as a bit-sliced, serial, pipelined, or tree-based design, again including reasoning about the relevant design tradeoffs in terms of area and delay.
- Design and compare implementations of processor control units using both hardwired and microprogrammed strategies, and again including reasoning about the relevant design tradeoffs in terms of area and delay.
- Understand basic tradeoffs in the sparsity of code words with error detection and correction capabilities.