Introduction to Digital Systems

0. Acknowledgments

Many thanks to Prof. Bernhard Boser and National Instruments for funding this project in the Summer of 2007. Ferenc Kovac has been (and will continue to be) an excellent mentor. Winthrop Williams designed the strain gauge lab (a paradigm of the K.I.S.S – Keep it Simple Stupid – philosophy). We are using this as a sensor for interfacing to our digital system (a PIC microcontroller). Tho Nguyen and Karen Tran were the brave souls who first tried out the PIC microcontroller. Kevin Mullaly's staff installed the microcontroller development tools.

Last but not the least, a shout-out to the authors on the internet who made this document possible: Tony R. Kuphaldt's free description of digital electronics (http://www.ibiblio.org/obp/electricCircuits/Digital/index.html) forms the crux of this document. Combined with Richard Bowles' excellent description of feedback in digital systems (http://richardbowles.tripod.com/dig_elec/chapter1/chapter1.htm), it is hoped that this document will serve as a self-contained introduction to digital systems.

Go Bears!

1. Introduction

Logic circuits are the basis for modern digital computer systems. To appreciate how computer systems operate you will need to understand digital logic and Boolean algebra. This chapter provides only a basic introduction to Boolean algebra – describing it in its entirety would take up an entire textbook. I chose to concentrate on the basics of Boolean algebra, rather than on optimizing concepts like Karnaugh Maps.

First we start out with the concept of digital vs. analog.

2. Digital vs. Analog [2]

The term *digital* refers to the fact that the signal is limited to only a few possible values. In general, digits signals are represented by only two possible voltages on a wire - 0 volts (which we called "binary 0", or just "0") and 5 volts (which we call "binary 1", or just "1"). We sometimes call these values "low" and "high", or "false" and "true".

More complicated signals can be constructed from 1s and 0s by stringing them end-to-end, like a necklace. If we put three binary digits end-to-end, we have eight possible combinations: 000, 001, 010, 011, 100, 101, 110 and 111. In principle, there is no limit to how many binary digits we can use in a signal, so signals can be as complicated as you like. The figure below shows a typical digital signal, firstly represented as a series of voltage levels that change as time goes on, and then as a series of 1s and 0s.

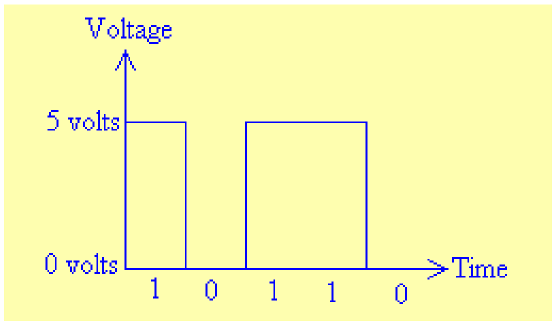


Figure 1. A digital signal

Analog electronics uses voltages that can be any value (within limits, of course - it's difficult to imagine a radio with voltages of a million volts!) The voltages often change smoothly from one value to the next, like gradually turning a light dimmer switch up or down. The figure below shows an analog signal that changes with time.

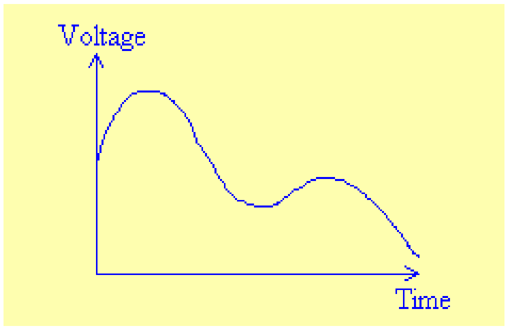


Figure 2. An analog signal

3. Number Systems [1]

a. The Binary Number System

The binary number system is a natural choice for representing the behavior of circuits that operate in one of two states (on or off, 1 or 0). For instance, we studied a diode logic gate (refer to the Diodes and Transistors handout online) when we discussed diode circuits. But before we study logic gates, you need to be intimately familiar with the binary number system – the system used by computers for counting.

Lets count from zero to twenty using the decimal number system and the binary number system.

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111

8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111
16	10000
17	10001
18	10010
19	10011
20	10100

Notice, though, how much shorter decimal notation is over binary notation, for the same number of quantities. What takes five bits in binary notation only takes two digits in decimal notation. An interesting footnote for this topic is the one of the first electronic digital computers, the ENIAC. The designers of the ENIAC chose to represent numbers in decimal form, digitally, using a series of circuits called "ring counters" instead of just going with the binary numeration system, in an effort to minimize the number of circuits required to represent and calculate very large numbers. This approach turned out to be counter-productive, and virtually all digital computers since then have been purely binary in design. This is intuitively due to that fact that a binary number directly maps to the “on” and “off” state in digital systems.

Notice that the binary number system and digital logic are actually two different concepts. A binary number is a number in base-2, it is independent of the concept of digital logic. However, the computer revolution is attributed to the very simple fact that mathematics in digital electronics can be represented by binary numbers. This is the number system that we will primarily study, along with the hexadecimal (base-16) system for convenience of representing large digits.

To convert a number in binary numeration to its equivalent in decimal form, all you have to do is calculate the sum of all the products of bits with their respective place-weight constants. To illustrate:

Convert 11001101_2 to decimal form:
```
bits =         1  1  0  0  1  1  0  1
.              -  -  -  -  -  -  -  -
weight =       1  6  3  1  8  4  2  1
(in decimal    2  4  2  6
notation)      8
```

The bit on the far right side is called the Least Significant Bit (LSB), because it stands in the place of the lowest weight (the one's place). The bit on the far left side is called the Most Significant Bit (MSB), because it stands in the place of the highest weight (the one hundred twenty-eight's place). Remember, a bit value of "1" means that the respective place weight gets added to the total value, and a bit value of "0" means that the respective place weight **does not** get added to the total value. With the above example, we have:

$128_{10} + 64_{10} + 8_{10} + 4_{10} + 1_{10} = 205_{10}$

If we encounter a binary number with a dot (.), called a "binary point" instead of a decimal point, we follow the same procedure, realizing that each place weight to the right of the point is one-half the value of the one to the left of it (just as each place weight to the right of a **decimal** point is one-tenth the weight of the one to the left of it). For example:

```
Convert 101.011₂  to decimal form:
.
bits =         1  0  1  .  0  1  1
.              -  -  -  -  -  -  -
weight =       4  2  1     1  1  1
(in decimal                /  /  /
notation)                  2  4  8


4₁₀  + 1₁₀  + 0.25₁₀  + 0.125₁₀  = 5.375₁₀
```

b. The Hexadecimal Number System

Because binary numeration requires so many bits to represent relatively small numbers compared to the economy of the decimal system, analyzing the numerical states inside of digital electronic circuitry can be a tedious task. Computer programmers who design sequences of number codes instructing a computer what to do would have a very difficult task if they were forced to work with nothing but long strings of 1's and 0's, the "native language" of any digital circuit. To make it easier for human engineers, technicians, and programmers to "speak" this language of the digital world, other systems of place-weighted numeration have been made which are very easy to convert to and from binary.

One of those numeration systems is called **octal,** because it is a place-weighted system with a base of eight. We won't discuss this base system in this document, rather we will concentrate on the hexadecimal system.

The hexadecimal system is a place-weighted system with a base of sixteen. Valid ciphers include the normal decimal symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, plus six alphabetical characters A, B, C, D, E, and F, to make a total of sixteen. As you might have guessed already, each place weight differs from the one before it by a factor of sixteen.

Let's count again from zero to twenty using decimal, binary and hexadecimal to contrast these systems of numeration:

Number	Decimal	Binary	Hexadecimal
Zero	0	0	0
One	1	1	1
Two	2	10	2
Three	3	11	3
Four	4	100	4
Five	5	101	5
Six	6	110	6
Seven	7	111	7
Eight	8	1000	8
Nine	9	1001	9
Ten	10	1010	A
Eleven	11	1011	B
Twelve	12	1100	C
Thirteen	13	1101	D
Fourteen	14	1110	E
Fifteen	15	1111	F

Sixteen	16	10000	10
Seventeen	17	10001	11
Eighteen	18	10010	12
Nineteen	19	10011	13
Twenty	20	10100	14

The hexadecimal numeration system would be pointless if not for the ability to be easily converted to and from binary notation. The primary purpose of the hexadecimal system is to serve as a "shorthand" method of denoting a number represented electronically in binary form. Because the hexadecimal (sixteen) system is an eve multiple of binary's base (two), binary bits can be grouped together and directly converted to or from hexadecimal digits: the binary bits are grouped in four's (because $2^4 = 16$):

```
BINARY TO HEXADECIMAL CONVERSION
Convert 10110111.1₂  to hexadecimal:
.
.                                        implied zeros
.                                             |||
.                              1011   0111   1000
Convert each group of bits     ----   ---- . ----
to its hexadecimal equivalent:   B      7      8
.
Answer:     10110111.1₂  =  B7.8₁₆
```

We had to group the bits in four's, from the binary point left, and from the binary point right, adding (implied) zeros as necessary to make complete 4-bit groups. Likewise, the conversion from hexadecimal to binary is done by taking each hexadecimal digit and converting it to its equivalent binary (4 bit) group, then putting all the binary bit groups together.

Another reason why the hexadecimal notation is more popular: binary bit groupings in digital equipment are commonly multiples of eight (8, 16, 32, 64, and 128 bit), which are also multiples of 4.

c. Binary Arithmetic

Now that we know what the binary number system is, lets take the next step: operations on binary numbers.

It is imperative to understand that the type of numeration system used to represent numbers has no impact upon the outcome of any arithmetical function (addition, subtraction, multiplication, division, roots, powers, or logarithms). A number is a number is a number; one plus one will always equal two (so long as we're dealing with **real** numbers), no matter how you symbolize one, one, and two. A prime number in decimal form is still prime if it's shown in binary form or hexadecimal. π is still the ratio between the circumference and diameter of a circle, no matter what symbol(s) you use to denote its value. The essential functions and interrelations of mathematics are unaffected by the particular system of symbols we might choose to represent quantities. This distinction between **numbers** and **systems of numeration** is critical to understand.

The essential distinction between the two is much like that between an object and the spoken word(s) we associate with it. A house is still a house regardless of whether we call it by its

English name **house** or its Spanish name **casa**. The first is the actual thing, while the second is merely the symbol for the thing.

That being said, performing a simple arithmetic operation such as addition (longhand) in binary form can be confusing to a person accustomed to working with decimal numeration only. In this lesson, we'll explore the techniques used to perform simple arithmetic functions on binary numbers, since these techniques will be employed in the design of electronic circuits to do the same. You might take longhand addition and subtraction for granted, having used a calculator for so long, but deep inside that calculator's circuitry all those operations are performed "longhand," using binary numeration. To understand how that's accomplished, we need to review to the basics of arithmetic.

Adding binary numbers is a very simple task, and very similar to the longhand addition of decimal numbers. As with decimal numbers, you start by adding the bits (digits) one column, or place weight, at a time, from right to left. Unlike decimal addition, there is little to memorize in the way of rules for the addition of binary bits:

```
0 + 0 = 0
1 + 0 = 1
0 + 1 = 1
1 + 1 = 10
1 + 1 + 1 = 11
```

Just as with decimal addition, when the sum in one column is a two-bit (two-digit) number, the least significant figure is written as part of the total sum and the most significant figure is "carried" to the next left column. Consider the following examples:

```
.                          11  1 <--- Carry bits ----->  11
.     1001101             1001001                     1000111
.   + 0010010           + 0011001                   + 0010110
.   ---------           ---------                   ---------
.     1011111             1100010                     1011101
```

The addition problem on the left did not require any bits to be carried, since the sum of bits in each column was either 1 or 0, not 10 or 11. In the other two problems, there definitely were bits to be carried, but the process of addition is still quite simple.

As we'll see later, there are ways that electronic circuits can be built to perform this very task of addition, by representing each bit of each binary number as a voltage signal (either "high," for a 1; or "low" for a 0). This is the very foundation of all the arithmetic which modern digital computers perform.

With addition being easily accomplished, we can perform the operation of subtraction with the same technique simply by making one of the numbers negative. For example, the subtraction problem of 7 - 5 is essentially the same as the addition problem 7 + (-5). Since we already know how to represent positive numbers in binary, all we need to know now is how to represent their negative counterparts and we'll be able to subtract.

Usually we represent a negative decimal number by placing a minus sign directly to the left of the most significant digit, just as in the example above, with -5. However, the whole purpose of using binary notation is for constructing on/off circuits that can represent bit values in terms of voltage (2 alternative values: either "high" or "low"). In this context, we don't have the luxury of a third symbol such as a "minus" sign, since these circuits can only be on or off (two possible states). One solution is to reserve a bit (circuit) that does nothing but represent the mathematical sign:

```
.                        101₂ = 5₁₀    (positive)
.
.  Extra bit, representing sign (0=positive, 1=negative)
.                       |
.                       0101₂ = 5₁₀    (positive)
.
.  Extra bit, representing sign (0=positive, 1=negative)
.                       |
.                       1101₂ = -5₁₀   (negative)
```

As you can see, we have to be careful when we start using bits for any purpose other than standard place-weighted values. Otherwise, 1101_2 could be misinterpreted as the number thirteen when in fact we mean to represent negative five. To keep things straight here, we must first decide how many bits are going to be needed to represent the largest numbers we'll be dealing with, and then be sure not to exceed that bit field length in our arithmetic operations. For the above example, I've limited myself to the representation of numbers from negative seven (1111_2) to positive seven (0111_2), and no more, by making the fourth bit the "sign" bit. Only by first establishing these limits can I avoid confusion of a negative number with a larger, positive number.

Representing negative five as 1101_2 is an example of the **sign-magnitude** system of negative binary numeration. By using the leftmost bit as a sign indicator and not a place-weighted value, I am sacrificing the "pure" form of binary notation for something that gives me a practical advantage: the representation of negative numbers. The leftmost bit is read as the sign, either positive or negative, and the remaining bits are interpreted according to the standard binary notation: left to right, place weights in multiples of two.

As simple as the sign-magnitude approach is, it is not very practical for arithmetic purposes. For instance, how do I add a negative five (1101_2) to any other number, using the standard technique for binary addition? I'd have to invent a new way of doing addition in order for it to work, and if I do that, I might as well just do the job with longhand subtraction; there's no arithmetical advantage to using negative numbers to perform subtraction through addition if we have to do it with sign-magnitude numeration, and that was our goal!

There's another method for representing negative numbers which works with our familiar technique of longhand addition, and also happens to make more sense from a place-weighted numeration point of view, called **complementation**. With this strategy, we assign the leftmost bit to serve a special purpose, just as we did with the sign-magnitude approach, defining our number limits just as before. However, this time, the leftmost bit is more than just a sign bit; rather, it possesses a negative place-weight value. For example, a value of negative five would be represented as such:

```
Extra bit, place weight = negative eight
.                    |
.                    1011₂ = 5₁₀   (negative)
.
.          (1 x -8₁₀)  +  (0 x 4₁₀)  +  (1 x 2₁₀)  +  (1 x 1₁₀)  =  -5₁₀
```

With the right three bits being able to represent a magnitude from zero through seven, and the leftmost bit representing either zero or negative eight, we can successfully represent any integer number from negative seven ($1001_2 = -8_{10} + 7_{10} = -1_{10}$) to positive seven ($0111_2 = 0_{10} + 7_{10} = 7_{10}$).

Representing positive numbers in this scheme (with the fourth bit designated as the negative weight) is no different from that of ordinary binary notation. However, representing negative numbers is not quite as straightforward:

zero	0000		
positive one	0001	negative one	1111
positive two	0010	negative two	1110
positive three	0011	negative three	1101
positive four	0100	negative four	1100
positive five	0101	negative five	1011
positive six	0110	negative six	1010
positive seven	0111	negative seven	1001

Note that the negative binary numbers in the right column, being the sum of the right three bits' total plus the negative eight of the leftmost bit, don't "count" in the same progression as the positive binary numbers in the left column. Rather, the right three bits have to be set at the proper value to equal the desired (negative) total when summed with the negative eight place value of the leftmost bit.

Those right three bits are referred to as the **two's complement** of the corresponding positive number. Consider the following comparison:

positive number	two's complement
001	111
010	110
011	101
100	100
101	011
110	010
111	001

In this case, with the negative weight bit being the fourth bit (place value of negative eight), the two's complement for any positive number will be whatever value is needed to add to negative eight to make that positive value's negative equivalent. Thankfully, there's an easy way to figure out the two's complement for any binary number: simply invert all the bits of that number, changing all 1's to 0's and visa-versa (to arrive at what is called the **one's complement**) and then add one! For example, to obtain the two's complement of five (101_2), we would first invert all the bits to obtain 010_2 (the "one's complement"), then add one to obtain 011_2, or -5_{10} in three-bit, two's complement form.

Interestingly enough, generating the two's complement of a binary number works the same if you manipulate **all** the bits, including the leftmost (sign) bit at the same time as the magnitude bits. Let's try this with the former example, converting a positive five to a negative five, but performing the complementation process on all four bits. We must be sure to include the 0 (positive) sign bit on the original number, five (0101_2). First, inverting all bits to obtain the one's complement: 1010_2. Then, adding one, we obtain the final answer: 1011_2, or -5_{10} expressed in four-bit, two's complement form.

It is critically important to remember that the place of the negative-weight bit must be already determined before any two's complement conversions can be done. If our binary numeration field were such that the eighth bit was designated as the negative-weight bit (10000000_2), we'd have to determine the two's complement based on all seven of the other bits. Here, the two's complement of five (0000101_2) would be 1111011_2. A positive five in this system would be represented as 00000101_2, and a negative five as 11111011_2.

We can subtract one binary number from another by using the standard techniques adapted for decimal numbers (subtraction of each bit pair, right to left, "borrowing" as needed from bits to the left). However, if we can leverage the already familiar (and easier) technique of binary addition to subtract, that would be better. As we just learned, we can represent negative binary numbers by using the "two's complement" method and a negative place-weight bit. Here, we'll use those negative binary numbers to subtract through addition. Here's a sample problem:

Subtraction: $7_{10} - 5_{10}$ Addition equivalent: $7_{10} + (-5_{10})$

If all we need to do is represent seven and negative five in binary (two's complemented) form, all we need is three bits plus the negative-weight bit:

positive seven = 0111_2
negative five = 1011_2

Now, let's add them together:

```
.                    1111  <--- Carry bits
.                     0111
.                   + 1011
.                   ------
.                    10010
.                    |
.             Discard extra bit
.
.            Answer = 0010₂
```

Since we've already defined our number bit field as three bits plus the negative-weight bit, the fifth bit in the answer (1) will be discarded to give us a result of 0010_2, or positive two, which is the correct answer.

Another way to understand why we discard that extra bit is to remember that the leftmost bit of the lower number possesses a negative weight, in this case equal to negative eight. When we add these two binary numbers together, what we're actually doing with the MSBs is subtracting the lower number's MSB from the upper number's MSB. In subtraction, one never "carries" a digit or bit on to the next left place-weight.

Let's try another example, this time with larger numbers. If we want to add -25_{10} to 18_{10}, we must first decide how large our binary bit field must be. To represent the largest (absolute value) number in our problem, which is twenty-five, we need at least five bits, plus a sixth bit for the negative-weight bit. Let's start by representing positive twenty-five, then finding the two's complement and putting it all together into one numeration:

```
+25₁₀  = 011001₂ (showing all six bits)
One's complement of 11001₂ = 100110₂
One's complement + 1 = two's complement = 100111₂
-25₁₀ = 100111₂
```

Essentially, we're representing negative twenty-five by using the negative-weight (sixth) bit with a value of negative thirty-two, plus positive seven (binary 111_2).

Now, let's represent positive eighteen in binary form, showing all six bits:

```
.              18₁₀  = 010010₂
.
.       Now, let's add them together and see what we get:
.
.                    11   <--- Carry bits
.                  100111
.                + 010010
.                --------
.                  111001
```

Since there were no "extra" bits on the left, there are no bits to discard. The leftmost bit on the answer is a 1, which means that the answer is negative, in two's complement form, as it should be. Converting the answer to decimal form by summing all the bits times their respective weight values, we get:

```
(1 x -32₁₀)  +  (1 x 16₁₀)  +  (1 x 8₁₀)  +  (1 x 1₁₀)  = -7₁₀
```

Indeed -7_{10} is the proper sum of -25_{10} and 18_{10}.

d. Concept of Overflow

One caveat with signed binary numbers is that of **overflow**, where the answer to an addition or subtraction problem exceeds the magnitude which can be represented with the allotted number of bits. Remember that the place of the sign bit is fixed from the beginning of the problem. With the last example problem, we used five binary bits to represent the magnitude of the number, and the left-most (sixth) bit as the negative-weight, or sign, bit. With five bits to represent magnitude, we have a representation range of 2^5, or thirty-two integer steps from 0 to maximum. This means that we can represent a number as high as $+31_{10}$ (011111_2), or as low as -32_{10} (100000_2). If we set up an addition problem with two binary numbers, the sixth bit used for sign, and the result either exceeds $+31_{10}$ or is less than -32_{10}, our answer will be incorrect. Let's try adding 17_{10} and 19_{10} to see how this overflow condition works for excessive positive numbers:

```
.       17₁₀  = 10001₂            19₁₀  = 10011₂
.
.                            1  11  <--- Carry bits
.    (Showing sign bits)    010001
.                         + 010011
.                         --------
.                           100100
```

The answer (100100_2), interpreted with the sixth bit as the -32_{10} place, is actually equal to -28_{10}, not $+36_{10}$ as we should get with $+17_{10}$ and $+19_{10}$ added together! Obviously, this is not correct. What went wrong? The answer lies in the restrictions of the six-bit number field within which we're working Since the magnitude of the true and proper sum (36_{10}) exceeds the allowable limit for our designated bit field, we have an **overflow error**. Simply put, six places doesn't give enough bits to represent the correct sum, so whatever figure we obtain using the strategy of discarding the left-most "carry" bit will be incorrect.

A similar error will occur if we add two negative numbers together to produce a sum that is too low for our six-bit binary field. Let's try adding -17_{10} and -19_{10} together to see how this works (or doesn't work, as the case may be!):

```
.      $-17_{10}$  = $101111_2$           $-19_{10}$  = $101101_2$
.
.                           1 1111  <--- Carry bits
.    (Showing sign bits)     101111
.                          + 101101
.                          --------
.                           1011100
.                           |
.                   Discard extra bit
.
FINAL ANSWER:   $011100_2$   = $+28_{10}$
```

The (incorrect) answer is a **positive** twenty-eight. The fact that the real sum of negative seventeen and negative nineteen was too low to be properly represented with a five bit magnitude field and a sixth sign bit is the root cause of this difficulty.

Let's try these two problems again, except this time using the seventh bit for a sign bit, and allowing the use of 6 bits for representing the magnitude:

```
.         $17_{10}$ + $19_{10}$                      ($-17_{10}$) + ($-19_{10}$)
.
.          1  11                           11 1111
.         0010001                            1101111
.       + 0010011                          + 1101101
.       ---------                          ---------
.         $0100100_2$                          $11011100_2$
.                                           |
.                                     Discard extra bit
.
. ANSWERS:  $0100100_2$ = $+36_{10}$
.           $1011100_2$ = $-36_{10}$
```

By using bit fields sufficiently large to handle the magnitude of the sums, we arrive at the correct answers.

In these sample problems we've been able to detect overflow errors by performing the addition problems in decimal form and comparing the results with the binary answers. For example, when adding $+17_{10}$ and $+19_{10}$ together, we knew that the answer was *supposed* to be $+36_{10}$, so when the binary sum checked out to be -28_{10}, we knew that something had to be wrong. Although this is a valid way of detecting overflow, it is not very efficient. After all, the whole idea of complementation is to be able to reliably add binary numbers together and not have to double-check the result by adding the same numbers together in decimal form! This is especially true for the purpose of building electronic circuits to add binary quantities together:

the circuit has to be able to check itself for overflow without the supervision of a human being who already knows what the correct answer is.

What we need is a simple error-detection method that doesn't require any additional arithmetic. Perhaps the most elegant solution is to check for the **sign** of the sum and compare it against the signs of the numbers added. Obviously, two positive numbers added together should give a positive result, and two negative numbers added together should give a negative result. Notice that whenever we had a condition of overflow in the example problems, the sign of the sum was always **opposite** of the two added numbers: $+17_{10}$ plus $+19_{10}$ giving -28_{10}, or -17_{10} plus -19_{10} giving $+28_{10}$. By checking the signs alone we are able to tell that something is wrong.

But what about cases where a positive number is added to a negative number? What sign should the sum be in order to be correct. Or, more precisely, what sign of sum would necessarily indicate an overflow error? The answer to this is equally elegant: there will **never** be an overflow error when two numbers of opposite signs are added together! The reason for this is apparent when the nature of overflow is considered. Overflow occurs when the magnitude of a number exceeds the range allowed by the size of the bit field. The sum of two identically-signed numbers may very well exceed the range of the bit field of those two numbers, and so in this case overflow is a possibility. However, if a positive number is added to a negative number, the sum will always be closer to zero than either of the two added numbers: its magnitude *must* be less than the magnitude of either original number, and so overflow is impossible.

Fortunately, this technique of overflow detection is easily implemented in electronic circuitry, and it is a standard feature in digital adder circuits. However, a discussion of this circuit is beyond the scope of this document. Nevertheless, you can use the basic concepts studied in the next section to try and design such a circuit. Before we jump to that section on digital logic gates, there is one important point to consider: Bit groupings.

e. Bit Groupings

The singular reason for learning and using the binary numeration system in electronics is to understand how to design, build, and troubleshoot circuits that represent and process numerical quantities in digital form. Since the bivalent (two-valued) system of binary bit numeration lends itself so easily to representation by "on" and "off" transistor states (recall saturation and cutoff, respectively in the case of the BJT), it makes sense to design and build circuits leveraging this principle to perform binary calculations.

If we were to build a circuit to represent a binary number, we would have to allocate enough transistor circuits to represent as many bits as we desire. In other words, in designing a digital circuit, we must first decide how many bits (maximum) we would like to be able to represent, since each bit requires one on/off circuit to represent it. This is analogous to designing an abacus to digitally represent decimal numbers: we must decide how many digits we wish to handle in this primitive "calculator" device, for each digit requires a separate rod with its own beads. Refer to figure 3 below.

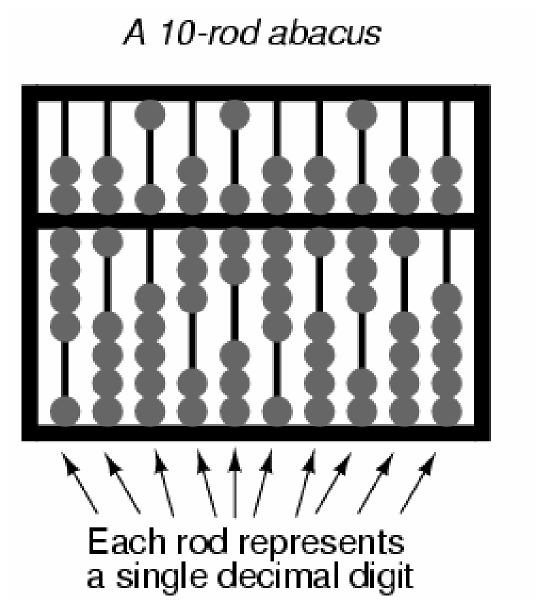


Figure 3. The Abacus, the first "digital computer"

A ten-rod abacus would be able to represent a ten-digit decimal number, or a maximum value of 9,999,999,999. If we wished to represent a larger number on this abacus, we would be unable to, unless additional rods could be added to it.

In digital, electronic computer design, it is common to design the system for a common "bit width:" a maximum number of bits allocated to represent numerical quantities. Early digital computers handled bits in groups of four or eight. More modern systems handle numbers in clusters of 32 bits or more. To more conveniently express the "bit width" of such clusters in a digital computer, specific labels were applied to the more common groupings.

Eight bits, grouped together to form a single binary quantity, is known as a **byte**. Four bits, grouped together as one binary number, is known by the humorous title of **nibble**, often spelled as **nybble**.

A multitude of terms have followed byte and nibble for labeling specific groupings of binary bits. Most of the terms shown here are informal, and have not been made "authoritative" by any standards group or other sanctioning body. However, their inclusion into this chapter is warranted by their occasional appearance in technical literature, as well as the levity they add to an otherwise dry subject:

- **Bit**: A single, bivalent unit of binary notation. Equivalent to a decimal "digit."
- **Crumb**, **Tydbit**, or **Tayste**: Two bits.
- **Nibble**, or **Nybble**: Four bits.
- **Nickle**: Five bits.
- **Byte**: Eight bits.
- **Deckle**: Ten bits.
- **Playte**: Sixteen bits.
- **Dynner**: Thirty-two bits.
- **Word**: (system dependent).

The most ambiguous term by far is **word**, referring to the standard bit-grouping within a particular digital system. For a computer system using a 32 bit-wide "data path," a "word" would mean 32 bits. If the system used 16 bits as the standard grouping for binary quantities, a "word" would mean 16 bits. The terms **playte** and **dynner** (hardly used), by contrast, always refer to 16 and 32 bits, respectively, regardless of the system context in which they are used.

4. Digital Logic Gates [1]

While the binary numeration system is an interesting mathematical abstraction, we haven't yet seen its practical application to electronics. This section is devoted to just that: practically applying the concept of binary bits to circuits. What makes binary numeration so important to the application of digital electronics is the ease in which bits may be represented in physical terms. Because a binary bit can only have one of two different values, either 0 or 1, any physical medium capable of switching between two saturated states may be used to represent a bit. Consequently, any physical system capable of representing binary bits is able to represent numerical quantities, and potentially has the ability to manipulate those numbers. This is the basic concept underlying digital computing.

The actual circuitry inside a gate is beyond the scope of this document, however you have seen one possibility in the Diodes and Transistors handout: a diode AND gate. Nevertheless, logic circuits today are built from MOS transistors, a topic again beyond the scope of this document. You will only deal with gate level abstraction in this class: NOT, AND and OR gates being the most important.

a. The NOT gate

The gate shown here is known as an **inverter**, or NOT gate, because it outputs the exact opposite digital signal as what is input. For convenience, gate circuits are generally represented by their own symbols rather than by their constituent transistors and resistors. The following is the symbol for an inverter:

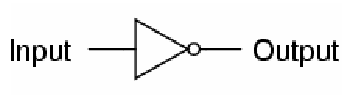
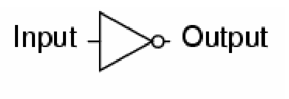


Figure 4. The NOT gate

One common way to express the particular function of a gate circuit is called a **truth table**. Truth tables show all combinations of input conditions in terms of logic level states (either "high" or "low," "1" or "0," for each input terminal of the gate), along with the corresponding output logic level, either "high" or "low." For the inverter, or NOT, circuit just illustrated, the truth table is very simple indeed:



Input	Output
0	1
1	0

Figure 5. NOT gate truth table

Truth tables for more complex gates are, of course, larger than the one shown for the NOT gate. A gate's truth table must have as many rows as there are possibilities for unique input combinations. For a single-input gate like the NOT gate, there are only two possibilities, 0 and 1. For a two input gate, there are **four** possibilities (00, 01, 10, and 11), and thus four rows to the corresponding truth table. For a three-input gate, there are **eight** possibilities (000, 001, 010, 011, 100, 101, 110, and 111), and thus a truth table with eight rows are needed. The mathematically inclined will realize that the number of truth table rows needed for a gate is equal to 2 raised to the power of the number of input terminals.

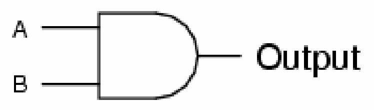
REVIEW:

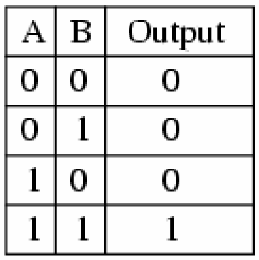
- In digital circuits, binary bit values of 0 and 1 are represented by voltage signals measured in reference to a common circuit point called **ground**. An absence of voltage represents a binary "0" and the presence of full DC supply voltage represents a binary "1."
- A **logic gate**, or simply **gate**, is a special circuit designed to input and output **logic level** voltages (voltages intended to represent binary bits). Gate circuits are most commonly represented in a schematic by their own unique symbols rather than by their constituent transistors and resistors.
- Just as with operational amplifiers, the power supply connections to gates are often omitted in schematic diagrams for the sake of simplicity.
- A **truth table** is a standard way of representing the input/output relationships of a gate circuit, listing all the possible input logic level combinations with their respective output logic levels.

Let us finish examining the other common gates.

b. The AND Gate

Figure 6 shows the schematic symbol and the truth table for a 2-input AND gate.





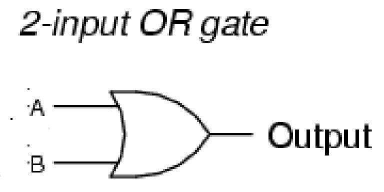
A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

Figure 6. AND gate schematic symbol and truth table

You can see the reason why it is called "AND" gate: the output will be 1 if and only if all inputs are 1. AND gates can be made with more than three inputs, but they are far less common.

c. The OR gate

Our next gate to investigate is the OR gate, so-called because the output of this gate will be "high" (1) if *any* of the inputs (first input *or* the second input *or* . . .) are "high" (1). The output of an OR gate goes "low" (0) if and only if all inputs are "low" (0). Figure 7 shows the schematic symbol and truth table of an OR gate.



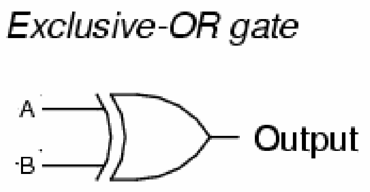
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Figure 7. OR gate schematic symbol and truth table

d. The XOR gate

The three previous gates are fairly direct variations on basic functions: NOT, AND and OR. The Exclusive-OR (XOR) gate is something quite different.

Exclusive-OR gates output a "high" (1) logic level if the inputs are at *different* logic levels, either 0 and 1 or 1 and 0. Conversely, they output a "low" (0) logic level if the inputs are at the *same* logic levels. The Exclusive-OR (sometimes called XOR) gate has both a symbol and a truth table pattern that is unique, refer to figure 8.

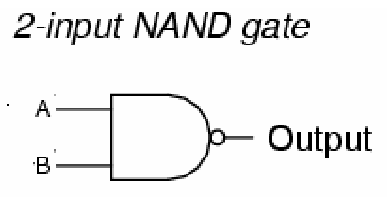


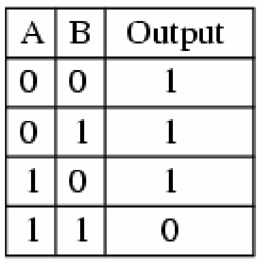
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Figure 8. XOR gate schematic symbol and truth table

e. The NAND gate

A variation on the idea of the AND gate is called the NAND gate. The word "NAND" is a verbal contraction of the words NOT and AND. Essentially, a NAND gate behaves the same as an AND gate with a NOT (inverter) gate connected to the output terminal. To symbolize this output signal inversion, the NAND gate symbol has a bubble on the output line. The truth table for a NAND gate is as one might expect, exactly opposite as that of an AND gate. Refer to figure 9.





A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

Figure 9. NAND gate schematic symbol and truth table

In practice, NAND gates are easier to realize than AND gates. Similar to the NAND gate, you can have a NOR gate and an XNOR gate.

5. Boolean Algebra

You have seen logic gates and how they can operate on binary numbers. Now, we need a mathematical framework for describing the relationship between logic gates and binary numbers. That framework is Boolean Algebra. This document of course provides only an introduction to Boolean algebra, refer to dedicated texts for a detailed discussion of the subject.

The English mathematician George Boole (1815-1864) sought to give symbolic form to Aristotle's system of logic. Boole wrote a treatise on the subject in 1854, titled *An Investigation of the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities*, which codified several rules of relationship between mathematical quantities limited to one of two possible values: true or false, 1 or 0. His mathematical system became known as Boolean algebra.

All arithmetic operations performed with Boolean quantities have but one of two possible outcomes: either 1 or 0. There is no such thing as "2" or "-1" or "1/2" in the Boolean world. It is a world in which all other possibilities are invalid by fiat. As one might guess, this is not the kind of math you want to use when balancing a checkbook or calculating current through a resistor. However, Claude Shannon of MIT fame recognized how Boolean algebra could be applied to on-and-off circuits, where all signals are characterized as either "high" (1) or "low" (0). His 1938 thesis, titled *A Symbolic Analysis of Relay and Switching Circuits*, put Boole's theoretical work to use in a way Boole never could have imagined, giving us a powerful mathematical tool for designing and analyzing digital circuits.

In this section, you will find a lot of similarities between Boolean algebra and "normal" algebra, the kind of algebra involving so-called real numbers. Just bear in mind that the system of numbers defining Boolean algebra is severely limited in terms of scope, and that there can only be one of two possible values for any Boolean variable: 1 or 0. Consequently, the "Laws" of Boolean algebra often differ from the "Laws" of real-number algebra, making possible such statements as 1 + 1 = 1, which would normally be considered absurd. Once you comprehend the premise of all quantities in Boolean algebra being limited to the two possibilities of 1 and 0, and the general philosophical principle of Laws depending on quantitative definitions, the "nonsense" of Boolean algebra disappears.

a. Boolean Arithmetic

Let us begin our exploration of Boolean algebra by adding numbers together:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 1$$

The first three sums make perfect sense to anyone familiar with elementary addition. The last sum, though, is quite possibly responsible for more confusion than any other single statement in digital electronics, because it seems to run contrary to the basic principles of mathematics. Well, it *does* contradict principles of addition for real numbers, but not for Boolean numbers. Remember that in the world of Boolean algebra, there are only two possible values for any quantity and for any arithmetic operation: 1 or 0. There is no such thing as "2" within the scope of Boolean values. Since the sum "1 + 1" certainly isn't 0, it must be 1 by process of elimination.

It does not matter how many or few terms we add together, either. Consider the following sums:

$$0 + 1 + 1 = 1$$

$$1 + 1 + 1 = 1$$

$$0 + 1 + 1 + 1 = 1$$

$$1 + 0 + 1 + 1 + 1 = 1$$

Take a close look at the two-term sums in the first set of equations. Does that pattern look familiar to you? It should! It is the same pattern of 1's and 0's as seen in the truth table for an OR gate. In other words, Boolean addition corresponds to the logical function of an "OR" gate, as shown in figure 10 below.

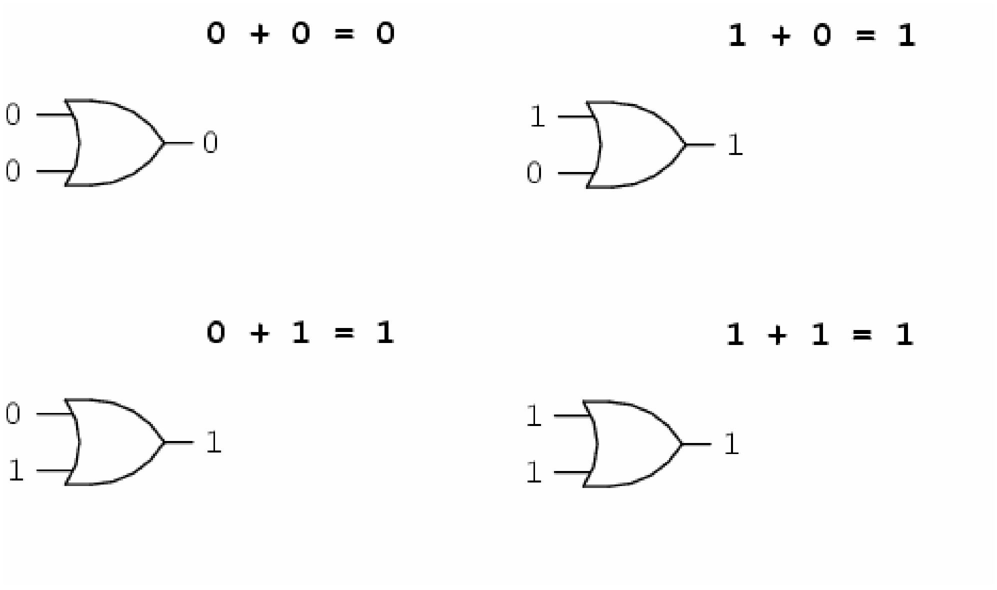


Figure 10. Boolean addition and the OR gate

There is no such thing as subtraction in the realm of Boolean mathematics. Subtraction implies the existence of negative numbers: 5 - 3 is the same thing as 5 + (-3), and in Boolean algebra negative quantities are forbidden. There is no such thing as division in Boolean mathematics, either, since division is really nothing more than compounded subtraction, in the same way that multiplication is compounded addition.

Multiplication is valid in Boolean algebra, and thankfully it is the same as in real-number algebra: anything multiplied by 0 is 0, and anything multiplied by 1 remains unchanged:

$$0 \times 0 = 0$$
$$0 \times 1 = 0$$
$$1 \times 0 = 0$$
$$1 \times 1 = 1$$

This set of equations should also look familiar to you: it is the same pattern found in the truth table for an AND gate. In other words, Boolean multiplication corresponds to the logical function of an "AND" gate.

Like "normal" algebra, Boolean algebra uses alphabetical letters to denote variables. Unlike "normal" algebra, though, Boolean variables are always UPPERCASE letters, never lower-case. Because they are allowed to possess only one of two possible values, either 1 or 0, each and every variable has a **complement**: the opposite of its value. For example, if variable "A" has a value of 0, then the complement of A has a value of 1. Boolean notation uses a bar above the variable character to denote complementation, like this.

If: $A=0$

Then: $\overline{A}=1$

If: $A=1$

Then: $\overline{A}=0$

In written form, the complement of "A" denoted as "A-not" or "A-bar". Sometimes a "prime" symbol is used to represent complementation (A'). Boolean complementation finds equivalency in the form of the NOT gate.

The basic definition of Boolean quantities has led to the simple rules of addition and multiplication, and has excluded both subtraction and division as valid arithmetic operations. We have symbols for denoting Boolean variables, and their complements. In the next section we will proceed to develop Boolean identities.

b. Boolean Algebraic Identities

In mathematics, an **identity** is a statement true for all possible values of its variable or variables. The algebraic identity of x + 0 = x tells us that anything (x) added to zero equals the original "anything," no matter what value that "anything" (x) may be. Like ordinary algebra, Boolean algebra has its own unique identities based on the bivalent states of Boolean variables. I just list the identities here, for detailed descriptions refer to [1]. If A is a Boolean variables, figure 11 below shows the basic Boolean algebraic identities.

Additive	Multiplicative
$A + 0 = A$	$0A = 0$
$A + 1 = 1$	$1A = A$
$A + A = A$	$AA = A$
$A + \overline{A} = 1$	$A\overline{A} = 0$

Figure 11. Basic Boolean algebraic identities

c. Boolean Algebraic Properties

Another type of mathematical identity, called a "property" or a "law," describes how differing variables relate to each other in a system of numbers. Assuming A and B are Boolean numbers, figure 12 lists the Boolean algebraic properties.

Additive	Multiplicative
A + B = B + A	AB = BA
A + (B + C) = (A + B) + C	A(BC) = (AB)C

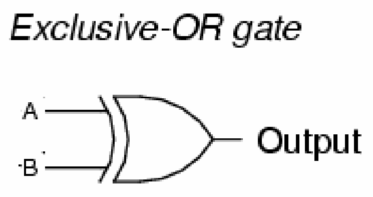
Distributive A(B + C) = AB + AC

Figure 12. Basic Boolean algebraic properties of additive association, multiplicative association and distribution

d. Translating truth tables into Boolean expressions

In designing digital circuits, the designer often begins with a truth table describing what the circuit should do. The design task is largely to determine what type of circuit will perform the function described in the truth table. While some people seem to have a natural ability to look at a truth table and immediately envision the necessary logic gate or relay logic circuitry for the task, there are procedural techniques available for the rest of us. Here, Boolean algebra proves its utility in a most dramatic way

As an example let us take a look at the exclusive-OR (XOR) gate. Figure 8 is repeated below as figure 13 for convenience.



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Figure 13. XOR gate schematic symbol and truth table revisited

It is not necessarily obvious what kind of logic circuit would satisfy the truth table. However, a simple method for designing such a circuit is found in a standard form of Boolean expression called the **Sum-Of-Products**, or **SOP**, form. As you might suspect, a Sum-Of-Products Boolean expression is literally a set of Boolean terms added (**summed**) together, each term being a multiplicative (**product**) combination of Boolean variables. An example of an SOP expression would be something like this: ABC + BC + DF, the sum of products "ABC," "BC," and "DF."

Sum-Of-Products expressions are easy to generate from truth tables. All we have to do is examine the truth table for any rows where the output is "high" (1), and write a Boolean product term that would equal a value of 1 given those input conditions. In figure 13, rows 2 and 3 have output high. The product term corresponding to row 2 would be A'B since the term would have a value of 1 if and only if A = 0 and B = 1. Similarly, the product term corresponding to row 3 would be AB'. Now, we join our Boolean products together by addition to create a single Boolean expression for the truth table as a whole. The ⊕ is the symbol for XOR.

$$A \oplus B = A'B + AB'$$

Now, we can easily translate the right hand-side of the equation above into a circuit, refer to figure 14.

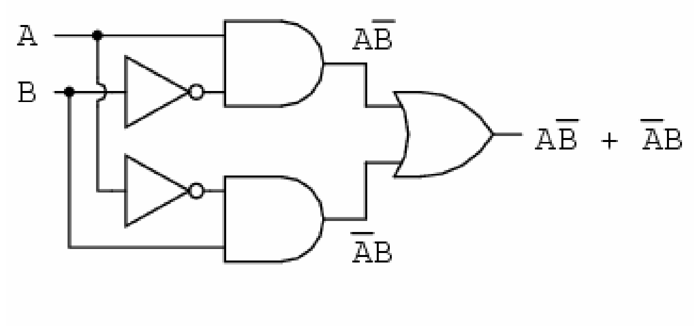


Figure 14. Simplified gate-level schematic of XOR

An alternative to generating a Sum-Of-Products expression to account for all the "high" (1) output conditions in the truth table is to generate a **Product-Of-Sums**, or **POS**, expression, to account for all the "low" (0) output conditions instead. For the XOR gate above, a POS expression is (A' + B').(A + B).

Both the Sum-Of-Products and Products-Of-Sums standard Boolean forms are powerful tools when applied to truth tables. They allow us to derive a Boolean expression -- and ultimately, an actual logic circuit -- from nothing but a truth table, which is a written specification for what we want a logic circuit to do. To be able to go from a written specification to an actual circuit using simple, deterministic procedures means that it is possible to automate the design process for a digital circuit. In other words, a computer could be programmed to design a custom logic circuit from a truth table specification! The steps to take from a truth table to the final circuit are so unambiguous and direct that it requires little, if any, creativity or other original thought to execute them.

e. Boolean Rules for Simplification

Boolean algebra finds its most practical use in the simplification of logic circuits. If we apply certain algebraic rules to a Boolean equation resulting from a truth table, we will get a simpler equation. The simplified equation may be translated into circuit form for a logic circuit performing the same function with fewer components. If equivalent function may be achieved with fewer components, the result will be increased reliability and decreased cost of manufacture. A few of the Boolean rules for simplification are shown below.

$$A + AB = A$$
$$A + \overline{A}B = A + B$$
$$(A + B)(A + C) = A + BC$$

Figure 15. Useful Boolean rules for simplification

Proving the rules above requires the use of concepts learned in sections 5 (a), (b) and (c). I prove the first two below and leave the third as an exercise.

A + AB = A.(1 + B) [Distributive property]
= A

A + A'B = (A + AB) + A'B [From rule above: A = A+B]
= A + (AB + A'B) [Additive association property]
= A + B.(A + A') [Distributive property]
= A + B

Now, we will encapsulate what we learned about the binary number systems, logic gates and Boolean algebra by designing a few common building blocks of digital systems.

6. Examples of Digital Systems

a. Adder [3]

For single bit adders, there are two general types. A **half-adder** has two inputs, generally labeled A and B and two outputs, the sum S and carry C. The truth table of a half-adder circuit is shown in figure 16.

Input		Output	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Figure 16. Truth table for the half-adder

A very important yet subtle point: the truth table above is **adding binary numbers.** This operation is **different** from the **addition operation** we defined for **Boolean algebra**. Recall from section 5 (a), the rules of addition for Boolean algebra:

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 1$$

Figure 16 is a truth table representation of binary addition. This is not the same as addition for Boolean algebra. We can **use** Boolean algebra to represent truth tables. In this case, we will use the SOP form to write the Boolean expressions for C and S as shown below:

$$C = A.B$$
$$S = A'.B + A.B' = A \oplus B$$

Hence, the circuit realization for the half-adder is shown in figure 17 below.

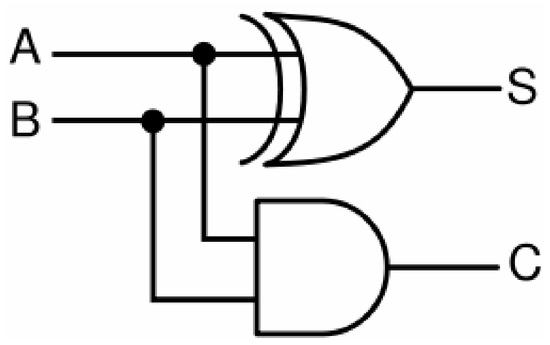


Figure 17. The half-adder

Next, let us realize a **full-adder**. This circuit takes into account a Carry-In (C*i*) as well. The truth table is shown in figure 18.

Input			Output	
A	B	C_i	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 18. The full-adder truth table. C_o is the Carry-out.

Again, we can use Boolean algebra (and corresponding simplification techniques) to write the Boolean expressions for C_o and S.

$$\begin{aligned} C_o &= A'.B.C_i + A.B'.C_i + A.B.C_i' + A.B.C_i \\ &= A'.B.C_i + A.B'.C_i + A.B \\ &= (A\oplus B).C_i + A.B \\ S &= A\oplus B\oplus C_i \end{aligned}$$

You can easily synthesize a gate level realization of C_o and S. Figure 19 shows a block diagram view of a 1-bit full adder.

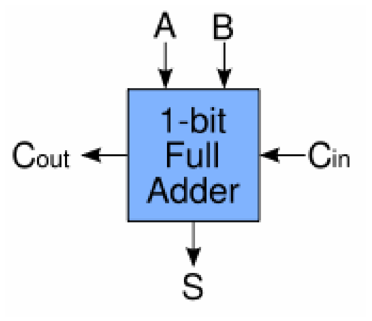


Figure 19. 1-bit full adder

Notice that we can chain carry-in and carry-outs together for multi-bit addition. This is called as a **ripple-carry adder** (figure 20) because the correct value of the carry bit ripples from one bit to the next. They layout of a ripple carry adder is simple, which allows for fast design time. However, the ripple-carry adder is relatively slow, since each full adder must wait for the carry bit to be calculated from the previous full adder. More complex adders like Carry look-ahead adders are beyond the scope of this document.

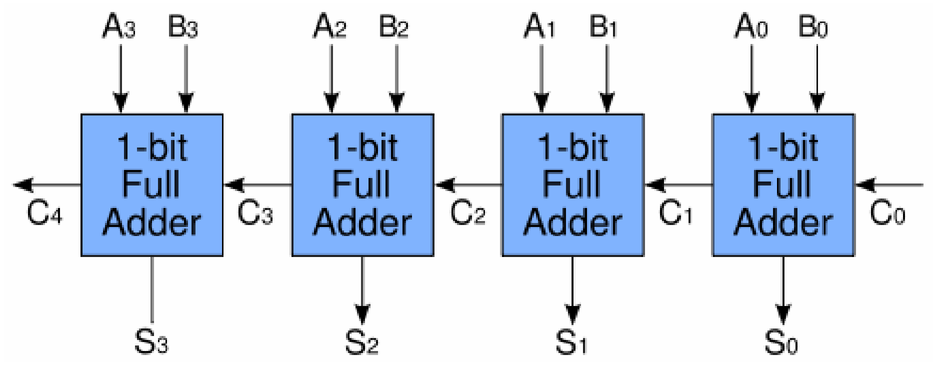
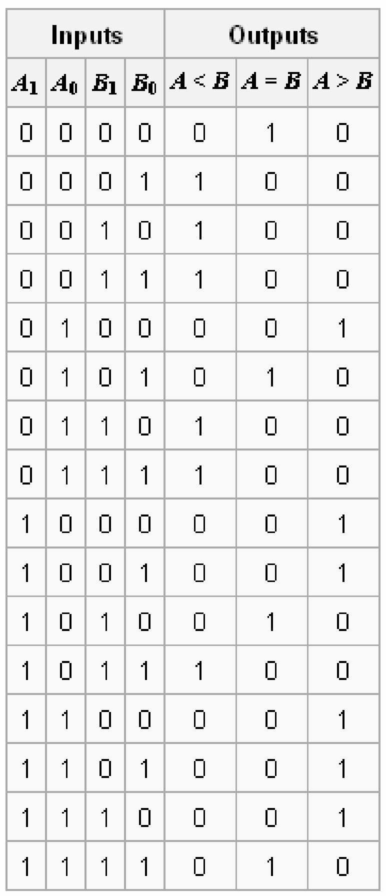


Figure 20. 4-bit Ripple-Carry adder

b. Comparator [4] [5]

A digital comparator is a hardware electronic device that compares two numbers in binary form and generates a one or a zero at its output depending on whether they are the same or not. Figure 21 shows the truth table for a 2-bit comparator.



Inputs				Outputs		
A_1	A_0	B_1	B_0	$A < B$	$A = B$	$A > B$
0	0	0	0	0	1	0
0	0	0	1	1	0	0
0	0	1	0	1	0	0
0	0	1	1	1	0	0
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	1	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	1	0	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	0	1	0

Figure 21. 2-bit comparator truth table

A block diagram view of the two bit comparator is shown below.

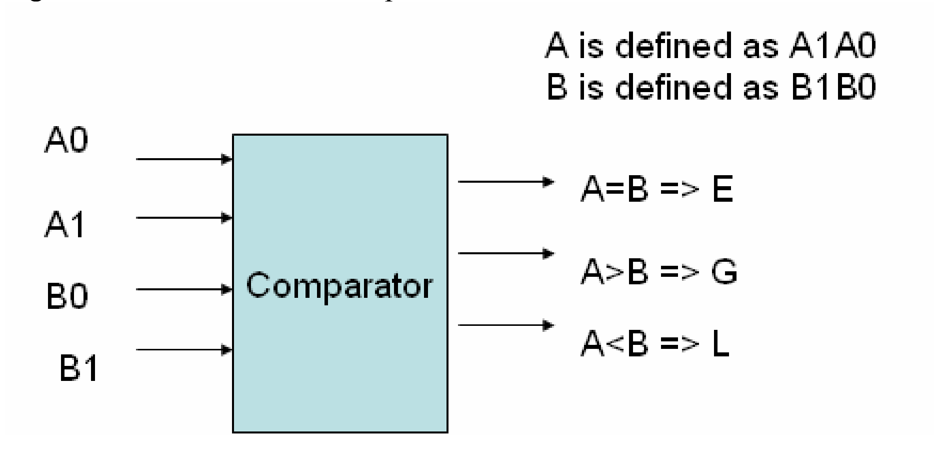


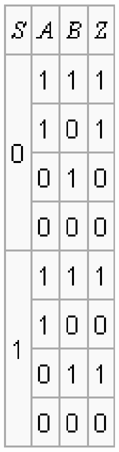
Figure 22. 2-bit comparator block diagram

Simplified Boolean expressions for the three outputs are shown below.

$E = ((A0 \oplus B0) + (A1 \oplus B1))'$
$G = A1B1' + A0B1'B0' + A1A0B0'$
$L = A1'B1+A1'A0'B0+A0'B1B0$

c. Multiplexer [6]

A multiplexer (mux) is a digital system that selects one out of possible 2^n inputs depending on n select bit(s). For instance, the truth table and schematic symbol for a 2-to-1 mux are shown below.



S	A	B	Z
0	1	1	1
	1	0	1
	0	1	0
	0	0	0
1	1	1	1
	1	0	0
	0	1	1
	0	0	0

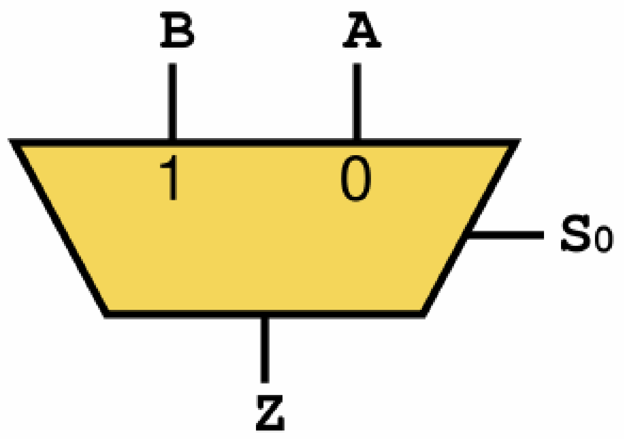


Figure 23. Truth table and schematic symbol for a 2-to-1 mux

Examining the truth table closely we see that a logic value of 0 on the select bit (S_0) would connect A to the output while a logic value of 1 would connect B to the output. The Boolean expression for the output (Z) in terms of inputs A, B and S is:

$$Z = (A.S') + (B.S)$$

Larger multiplexers are also common, if you have 4 inputs then you need 2 select bits. This is the reason for the n-select bits mapping 2^n inputs to one output.

d. Arithmetic Logic Unit (ALU)

We can combine the digital systems from (a), (b) and (c) to create a very simple Arithmetic Logic Unit (ALU)[1]. Refer to figure 24.

[1] Our design procedure is called "bottom-up" design

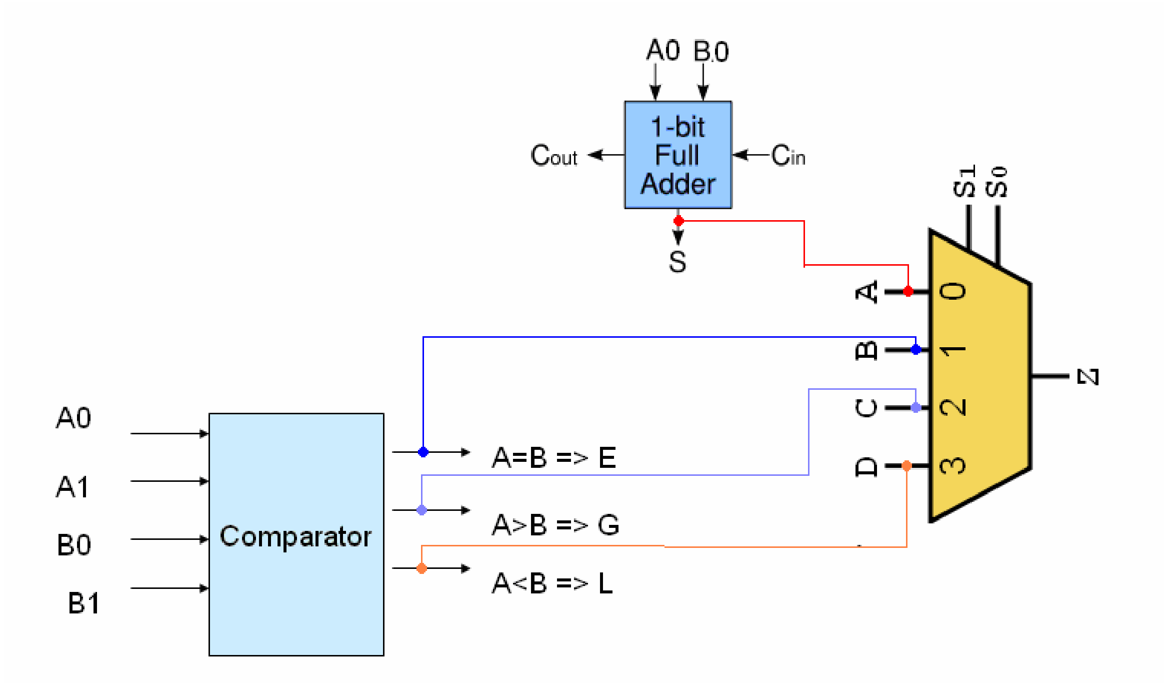


Figure 24. A very simple ALU

We select the output we want depending on S_1 and S_0. For instance, $S_1S_0 = 10$ selects G (A>B).

7. The Concept of Memory: Feedback in Digital Systems [2]

So far we have seen digital circuits with no feedback – **combinational logic** – circuits. In this section we will take a very brief look at feedback in digital systems (**sequential logic**). Consider figure 25.

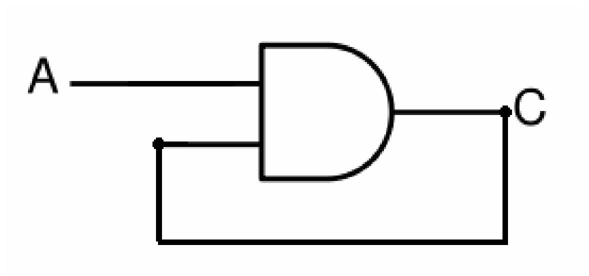


Figure 25. Simple memory unit

In order to calculate the output of the AND gate, we have to make some assumption about what the output already is and follow the argument through to its logical conclusion. For instance, lets suppose the output and the input are both 1. This is a valid assumption since if you feedback 1 to the input you get 1.1 = 1 and the system is consistent. Using this argument, the table below summarizes the behavior of the circuit.

A (Input)	Assumed state of the output	New state of the output
0	0	0
0	1	**Invalid assumption.** Output GOES TO 0
1	0	0
1	1	1

Table 1. "Truth-table" for the behavior of the circuit in figure 25

Let us use a different notation to represent the table above. Let us call the output at any particular instance Q_n, then the output at the next instant of time is represented by Q_{n+1}. Using this notation, we have:

Input	Q_{n+1}
0	0
1	Q_n

Table 2. Simplified truth table for the memory element in figure 25

This means that if the input is 0, then the output of the circuit is 0. If the input is 1, then the output is the same as it was one instant ago. If it was 0, then it stays 0. If it was 1 then is stays 1. The point to note here is the apparent[2] simplicity of this circuit as compared to its analog counterpart: the op-amp latch from your EE100 homework. You didn't need any external inductor to make a memory element out of a digital system. All you needed to add was feedback.

Such a simple circuit would not be used as a proper memory element. For a start, how would you ensure that the output of the circuit was 1 to begin with? You would have to connect the AND gate up so as to produce a 1 and then rewire the circuit without losing that output. This led engineers to develop ingenious ways of designing practical memory elements. To learn more take CS150 and/or EECS 141 at the University of California, Berkeley.

8. Conclusion

This is a long chapter! The point to notice here is how we were able to realize complex digital systems using elementary gates and Boolean algebra. This should remind you of how we were able to realize complex analog systems using KCL, KVL and I-V relationships.

The beauty of electronics is that a digital system is fundamentally analog, but the **digital abstraction** enabled us to design systems ranging from a "simple" desktop PC to a supercomputer and thus heralded the computer revolution.

9. References

[1] *Lessons in Electric Circuits – Volume IV (Digital)*. Kuphaldt, Tony R. Last accessed July 28th 2007. Online at: http://www.ibiblio.org/obp/electricCircuits/Digital/index.html

[2] I say "apparent" because if we analyze the transistor level implementation of a logic gate, it turns out that capacitors are really responsible for storing charge and hence enable this device to function as a memory element. But that concept is again beyond the scope of this present document.

[2] *Introduction to Digital Electronics – Lesson 7*. Bowles, Richard. Last accessed July 28th 2007. Online at: http://richardbowles.tripod.com/dig_elec/chapter7/chapter7.htm
[3] *Adder Wiki*. Last accessed July 28th 2007. Online at: http://en.wikipedia.org/wiki/Adder_(electronics)
[4] *Comparator Wiki*. Last accessed July 28th 2007. Online at: http://en.wikipedia.org/wiki/Digital_comparator
[5] *Combinational Circuits*. Abhari, Abdolreza. Ryerson University lecture slides. Last accessed July 28th 2007. Online at: http://www.scs.ryerson.ca/~aabhari/cps213Chapter4.ppt
[6] *Multiplexer Wiki*. Last accessed July 28th 2007. Online at: http://en.wikipedia.org/wiki/Multiplexer

10. REVISION HISTORY

Date/Author	Revision comments
Summer 2007/Bharathwaj Muthuswamy	Typed up initial doc.