

Review of ECE 230 Material

Prof. A. Mason, Michigan State University

Preface

This document was developed for students taking ECE 331 to review material covered in ECE 230. It will be assumed that ECE 331 students already know all of this material, so please review it thoroughly and contact your instructor if you have any questions. Experience has shown that the following topics are particularly important for students to study:

- 2's complement form
- Boolean arithmetic using 2's complements
- DeMorgan relations
- Karnaugh maps.

Contents

- | | |
|---|------------------------------------|
| 1. Number Systems | 3.1.2. DeMorgan relations |
| 1.1. Digital Bases | 3.1.3. Tristate, XOR, MUX, Decoder |
| 1.2. Base Conversions | 3.2. Sequential Logic Circuits |
| 1.3. Boolean Addition | 3.2.1. Flip flops |
| 1.4. Boolean Subtraction (2's complement) | 3.2.2. Registers, Shifters |
| 2. Logic Expressions and Minimization | 3.2.3. Counters |
| 2.1. Boolean arithmetic properties | 3.3. Complete Set Concept |
| 2.2. Min/Max term notation | 4. Appendices |
| 2.3. Reducing logic expressions | Base Values |
| 3. Digital Logic | Powers of 2 |
| 3.1. Combinational Logic Gates | Signed Value Forms |
| 3.1.1. INV, AND, OR | |

1. Number Systems

1.1 Digital Bases

Number bases of interest to digital logic:

- **Decimal (Dec)** Base 10 "normal" numbers
- **Binary (Bin)** Base 2 standard digital base
- **Hexadecimal (Hex)** Base 16 shorthand for binary; used often in programming

Any number can be expressed as $N_r = A_{n-1}r^{n-1} + A_{n-2}r^{n-2} + \dots + A_0$, where

- **A** \equiv coefficient (varies with number base)
 - Base 10: **A** \in {0,1,...9}
 - Base 2: **A** \in {0,1}
 - Base 16: **A** \in {0,1,...9, A, B, C, D, E, F}
 - where A has a value of 10, B a value of 11, ... and F a value of 15
- **r** \equiv radix (function of number base; base 10, , base 2, **r**=2, etc.)
 - Base 10: **r** = 10
 - Base 2: **r** = 2
 - Base 16: **r** = 16
- **n** \equiv position

EXAMPLE: Expressing decimal number with multiple digits

For Base 10: $934_{10} = 9 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$

Note: The subscripts after the numbers identify the number base, e.g., 934_{10} is a Base 10 number. The Decimal, Binary, and Hexadecimal representations of (base 10) values 0 – 15 are shown in Appendix A, Table A.1.

Thought Exercise

How many binary digits are needed to represent the values {0, 1, ... 15}?

1.2 Base Conversions

A value in any number base can readily be converted to any other number base. Because we are accustomed to working with decimal numbers, conversion to base 10 is often the easiest.

EXAMPLE: Converting Bin and Hex to Dec

1. Base 2: $1001_2 = 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 9_{10}$
 8 4 2 1

2. Base 16: $4AC_{16} = 4 \times 16^2 + A \times 16^1 + C \times 16^0 = 4 (256) + 10 (16) + 12 (1) = 1196_{10}$
 256 16 1

Conversion between binary (base 2) and hexadecimal (base 16) is also straightforward. Hexadecimal (Hex) is considered to be a shorthand for binary (Bin) because 4 bits (digits) of a Bin number can be represented by 1 bit (digit) in Hex. ($2^4 = 16^1$)

EXAMPLE: Converting Bin to Hex

$10111111_2 = ?_{16}$

Step 1: Group by 4 bits, starting with lowest bits

$10111111 \rightarrow 1011\ 1111$

Step 2: Convert each 4-bit Bin value into a Hex value

$1011 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11_{10} = B_{16}$

$1111 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 15_{10} = F_{16}$

Step 3: Represent Hex value

$10111111_2 = BF_{16}$

See Textbook Appendix D for more examples.

EXAMPLE: Converting Bin to Hex

$$1000101111000111_2 = ?_{16}$$

Step 1: Group by 4 bits, starting with lowest bits

$$1000101111000111 \rightarrow 1000\ 1011\ 1100\ 0111$$

Step 2: Convert each 4-bit Bin value into a Hex value

$$1000 = 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 8_{10} = 8_{16}$$

$$1011 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11_{10} = B_{16}$$

$$1100 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 12_{10} = C_{16}$$

$$0111 = 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 7_{10} = 7_{16}$$

Step 3: Represent Hex value

$$1000101111000111_2 = 8BC7_{16}$$

Conversion from Decimal to Bin or Hex can be more mathematically complicated, involving a lot of division. It's often easier to utilize calculators for number base conversions. However, you are expected to be able to perform number conversions without a calculator. So **practice!**

EXAMPLE: Converting Dec to Hex

$$1592_{10} = ?_{16}$$

Step 1: Find the highest Base 16 digit in the value

$$16^3 = 4096, \text{ which is greater than } 1592, \text{ so there will be no } 16^3 \text{ digit}$$

$$16^2 = 256 \rightarrow 1592 / 256 = 6 \text{ (and some decimals)} \text{ [Remember this digit: } \underline{6} \times 16^2 \text{]}$$

Step 2: Subtract value of highest Base 16 digit from the number

$$1592 - 6 \times 16^2 = 56, \text{ the remaining value to represent with remaining Base 16 digits}$$

Step 3: Repeat steps 1 and 2, operating on the remaining value from step 2.

$$16^1 = 16 \rightarrow 56 / 16 = 3 \text{ (and some decimals)} \text{ [Remember this digit: } \underline{3} \times 16^1 \text{]}$$

$$56 - 3 \times 16^1 = 8$$

and the final digit...

$$16^0 = 1 \rightarrow 8 / 1 = 8 \text{ (with no decimals)} \text{ [Remember this digit: } \underline{8} \times 16^0 \text{]}$$

Step 4: Represent Hex value

$$1592_{10} = 638_{16}$$

Practice Exercise

Find the Binary representation of 1592_{10} . This should require a lot of Bin digits (bits) and a lot of divisions!

1.3 Boolean Addition

Addition: Bin and Hex numbers can be added exactly the same way as Dec numbers.

EXAMPLE: Addition in different number bases

Decimal		Binary		Hexadecimal
7	$7 \rightarrow 0111$	0111	$0111_2 \rightarrow 7_{16}$	7
+ 4	$4 \rightarrow 0100$	+ 0100	$0100_2 \rightarrow 4_{16}$	+ 4
<hr/> 11 ₁₀		<hr/> 1011 ₂		<hr/> B ₁₆

EXAMPLE: Addition in Hex and Bin

Add $A3_{16} + 2C_{16}$ in Hex. Then check by performing addition in Bin and Dec.

Step 1: $A3$

$$\begin{array}{r} + 2C \\ \hline CF_{16} \end{array}$$

Step 2: (in Bin) $A3_{16} \rightarrow 1010\ 0011_2$

$$+ 2C_{16} \rightarrow \underline{0010\ 1100}_2$$

$1100\ 1111_2$ (just add vertically as you would with decimal numbers)

↑ (carry any values greater than 1 to the next digit)

$1010\ 1111_2 \rightarrow CF_{16}$, so the Bin result checks with the Hex result.

Step 3: (in Dec) $A3_{16} \rightarrow A \times 16^1 + 3 \times 16^0 = 10 \times 16 + 3 = 163_{10}$

$$\begin{aligned} + 2C_{16} &\rightarrow 2 \times 16^1 + C \times 16^0 = 2 \times 16 + 12 = \underline{44}_{10} \\ &= 207_{10} \end{aligned}$$

$CF_{16} = C \times 16^1 + F \times 16^0 = 12 \times 16 + 15 = 207_{10}$ so the Dec result checks with the Hex result.

Overflow (out of range): In digital hardware, values are generally represented by a finite number of bits (or digits). As a result, some values are **out of range**, or greater than the maximum value that can be represented by the specified number of bits. Similarly, when values are added, the result can be greater than the maximum representable number. When a value is out of range, we say that an **overflow** has occurred. This overflow bit is also called a **carry out** bit and is generally a standard output signal from a CPU. To avoid confusion with 2's complement overflow, which is described below, we will refer to an overflow bit from addition as **carry out overflow**.

EXAMPLE: Addition with overflow

Decimal		Binary	
7	$7 \rightarrow 0111$	0111	The circled digit is an out of range overflow because it requires more than 4 bits to represent the value. Looking at the decimal value (18) we should expect this, because $18 > 15 = 2^4 - 1$
+11	$11 \rightarrow 1011$	+ 1011	
18_{10}		$\textcircled{1}0010_2$	

Maximum representable value: An n -bit binary number can represent values ranging from **0 to $2^n - 1$** . For example, a 4-bit Bin number can have a max value of $2^4 - 1 = 15$. More bits would be needed to represent any value greater than 15. And any operation (e.g., addition) resulting in a value requiring more than 4-bits would generate an overflow. More generally, for an arbitrary number base, r , using n digits to represent a value:

- $r^n \equiv$ number of representable values
- $r^n - 1 \equiv$ maximum representable value (range = 0 to $r^n - 1$)

Table A.2 in Appendix A shows the decimal value for n -bit binary numbers up to $n=30$ (1 Gigabit).

1.4 Boolean Subtraction

Signed 2's Complement Form: In digital hardware, additions are much easier to perform than subtractions. But subtraction can be achieved by adding a negative value. For example,

- $X - Y = X + (-Y)$

so rather than subtracting Y from X , we are adding $-Y$ to X . In digital hardware, numbers are saved (in memory) using a specific number of binary bits. So we can't just stick a negative sign in front of the

number. We have to find a way to store numbers that can be either positive or negative. The Signed 2's Complement form is essentially a method for encoding a number so that both negative and positive values can be stored. For binary numbers, each additional bit doubles the number of values that can be represented (4 values for 2 bits, 8 values for 3 bits, etc.). So we can dedicate the most significant bit (left-most) as a sign bit and represent $\frac{1}{2}$ of the normally possible values using the remaining bits.

There are two common forms for storing "signed" binary values. The simplest (but less often used) is called sign-magnitude form. Here, the most significant bit is a sign bit, where 0 = positive and 1 = negative, and the remaining bits are the numeric value of the number. The signed 2's complement (S2C) form is very similar; it uses the same sign bit and positive values are stored just like in sign-magnitude form, but negative numbers are *encoded* in a way that facilitates subtraction operations. Recall that subtraction can be performed by addition with a negated value

- $X - Y = X + (-Y)$

However, for finite value numbers, such as those stored within a specific number of bits (e.g., a two digit decimal number can have a maximum value of 99, or a four bit binary number can have a maximum value of 7)

- $X - Y = X + [Y]^*$, where $[Y]^*$ is the 2's complement of Y .

Factoid!?!

The 2's complement is a specific designation for binary numbers. More generally, it can be referred to as an **additive inverse**. Thanks Wikipedia!

In S2C form, positive values are stored as their normal value and negative values are stored as the 2's complement value. For example, consider an 8-bit binary value described by $A_7A_6A_5A_4A_3A_2A_1A_0$. If this value were in S2C form:

- $A^7 \equiv$ sign bit {1 = negative, 0 = positive}
 - if $A^7 = 0$, value = $A_6A_5A_4A_3A_2A_1A_0$
 - if $A^7 = 1$, value = $[A_6A_5A_4A_3A_2A_1A_0]^* \leftarrow$ 2's complement

Thus, when a number is in S2C form, we must first look at the sign bit. If the sign bit is 0, the value is simply that represented by all digits after the sign. If the sign bit is 1, the correct unsigned value is obtained by performing a 2's complement operation on the set of digits that follow the sign.

We will rarely use sign-magnitude form in ECE331, but for completeness, an 8-bit binary value in sign-magnitude form would be described by

- $A^7 \equiv$ sign bit {1 = negative, 0 = positive}
- $A_6-A_0 =$ 7-bit binary value

The difference between S2C and sign-magnitude forms is solely in how the value is represented by A_6-A_0 . Table A.3 in Appendix A shows the 4-bit representations of numbers using S2C and sign-magnitude form. Notice that sign-magnitude results in two values for zero and an inability to represent -8, which can be represented using S2C form.

2's Complement Operation: To represent negative values in S2C form, we must perform the 2's complement (2C) operation on the number. There are two methods to perform this operation. First is the more general approach for an additive inverse that can be used for any number base. Let $[X]^*$ represent the 2C of X . Then

- $[X]^* = r^n - X$, where $r =$ radix, $n =$ number of bits (digits)

Thus, for a 4-bit base 2 number, $[X]^* = 2^4 - X = 16 - X$ (in decimal) or $10000 - X$ (in binary).

For binary numbers, there is an alternative method that you are probably more familiar with.

- $[X]^* = \overline{X} + 1$, where \overline{X} is the binary complement of X , ($0 \rightarrow 1, 1 \rightarrow 0$)

Because it is very easy to perform the binary complement, both in hardware and by hand, you will probably use this method most of the time.

But why, Professor?

If you want to just accept that 2's complement is a magical form that permits $X - Y = X + [Y]^*$ you can skip this and continue on. If you want to understand the magic, consider this base 10 example that bypasses the unfamiliarity of binary numbers:

Assume we have two 1-digit decimal (base 10) numbers. The maximum value representable by a 1-digit decimal is 9. Now let's evaluate

- $7 - 2$

We can easily see the result is 5, but try it using 2's complement (or more strictly speaking the additive inverse).

- $7 - 2 \rightarrow 7 + [2]^*$

What is $[2]^*$? Use the general definition for 2's complement, $[X]^* = r^n - X$. Here, $r^n = 10^1 = 10$. So

- $10 - 2 = 8$

Now comes the magic!

- $7 + [2]^* = 7 + 8 = 15$

But, we only have 1-digit numbers, so only the least significant digit would be retained. Thus, $15 \rightarrow 5$, and we get the result we expected! $7 + [2]^* = 5$.

DON'T GET CONFUSED!

There is a difference between the "signed 2's complement form" (a noun) and "taking the 2's complement" (a verb) of a number. This difference often confuses students, so please think this through. Just because a number is in S2C form does not necessarily mean you need to perform a 2C operation in order to obtain its decimal value. If the S2C value is positive, no 2C operations are needed. Note also that the 2C operation is generally performed on the set of bits after the sign bit. This set of bits may or may not begin with a 1, since these bits represent the value of the number and not the sign. You can perform a 2C operation on any number, but not all numbers in S2C form need to be "decoded" to find their actual value.

EXAMPLE: 2's Complement Operation

1. What is the 2's complement of 1001_2 ?

binary complement of 1001 is 0110, so

$$[1001]^* = 0110 + 1 = 0111$$

2. What is the 2's complement of 0110101_2 ?

binary complement of 0110101 is 1001010, so

$$[0110101]^* = 1001010 + 1 = 1001011$$

Value Range in Signed 2's Complement Form: As described above, all numbers have a maximum representable value determined by the radix and number of digits (bits). Binary S2C form numbers represent both positive and negative values and thus have different value range than unsigned numbers. The value range of a S2C form number, N , is given by:

- $-2^{n-1} \leq N \leq 2^{n-1} - 1$, where n = number of bits (digits)

For example, a 4-bit S2C Bin number has a range of: $-2^3 \leq N \leq 2^3 - 1 \rightarrow -8 \leq N \leq 7$. Numbers > 7 or < -8 can not be represented and are thus out of range. This is also called **2C Overflow** and is discussed further below.

It is important to know the possible range of values when converting to/from S2C form because it will determine the number of bits required. You need to be able to identify when values are out of range.

EXAMPLE: Converting to signed 2's complement form

1. What is the 4-bit signed 2's complement form of the decimal number 5_{10} ?

This is a positive number so the sign bit is 0 and we just need to convert to binary

$$5_{10} = 4 + 1 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 101$$

Thus the S2C form of 5_{10} is 0101_2

2. What is the 6-bit signed 2's complement form of the decimal number -28_{10} ?

This is a negative number so the sign bit is 1. To find the other 5 bits, we must perform the 2C operation of 28_{10} and we just need to convert to binary

$$28_{10} = 11100_2, \text{ using a calculator}$$

$$[11100]^* = 00011 + 1 = 00100$$

Thus after adding the sign bit the S2C form of -28_{10} is 100100_2

EXAMPLE: Converting from signed 2's complement form

1. What is the 2's complement of 1001_2 ? (Value not specified as being in SC2 form).

binary complement of 1001 is 0110, so

$$[1001]^* = 0110 + 1 = 0111 \rightarrow 7_{10}$$

2. What is the decimal value of the signed 2's complement number 1001_2 ?

Because the value is in SC2 form, the first bit is a sign bit, so 1001 represents $-[001]^*$

$$[001]^* = 110 + 1 = 111 = 7_{10}, \text{ so } 1001_2 = -7_{10}$$

3. What is the 2's complement of 0110101_2 ? (Value not specified as being in SC2 form).

binary complement of 0110101 is 1001010, so

$$[0110101]^* = 1001010 + 1 = 1001011 \rightarrow 75_{10}$$

4. What is the decimal value of the signed 2's complement number 0110101_2 ?

Because the value is in SC2 form, the first bit is a sign bit, so 0110101 represents $+ [110101]$

because this is a positive value, no 2C operation is performed and

$$0110101 = + 110101 = +35_{16} = 3 \times 16 + 5 = 53_{10}$$

Subtraction: In ECE331, we will always perform binary subtraction using the 2's complement method. That is, when asked to perform $X - Y$, you should always evaluate $X + [Y]^*$. Binary addition is very easy, and performing the 2's complement operation is very easy. The only things that can be tricky are recognizing when to perform the 2C operation, determining if values are in the proper range for 2C operation, and identifying overflow in the result. Let's look at each of those.

When should you perform 2C operations? Most obviously, any time you are directly asked to find the 2's complement of a number, for example in a homework or test problem. In this case, just evaluate $[X]^* = \overline{X} + 1$. It does not matter if the number is in S2C form or not; if the instruction is to perform 2C operation, just complement the number and add 1. Similarly, if asked to evaluate $X - Y$ for binary numbers, simply determine $[Y]^*$ and evaluate $X + [Y]^*$. It does not matter if Y was negative or whether it was in S2C form or not. Just complement Y, add 1, and add it to X. If Y happened to have been a S2C negative number, then you would have transformed it to a positive, like $X - (-Y) \rightarrow X + Y$, as desired.

The only time you should need to ask yourself, “should I perform 2C?” is when you are asked to find a **numeric value**, or decimal value. In that case, you have to know if the value is in S2C form. If not, just do a standard Bin to Dec conversion. If it is in S2C form and it’s a positive value (sign bit = 0), then you also just do a standard Bin to Dec conversion. Only when you have a S2C negative number will you need to perform a 2C operation in order to find the numeric value. Then you should perform the 2C operation on the entire number (including the sign bit which just changes to a zero), do a Bin to Dec conversion on the 2C result, and put a minus sign in front of it. Got it? If not, read though this again. This is one of the most common problems ECE331 students have had in the past.

Are values in the proper range? As described above, S2C form numbers can not represent values as large and unsigned numbers. Thus, you need to check that values you are working with are not too large to be converted to signed numbers without adding additional bits. For example, a 4-bit binary number has a maximum value of 15_{10} (1111_2), but the maximum value for a signed 4-bit binary number is 7_{10} (0111_2). If you are asked to evaluate $12 - 9$ using binary arithmetic, you must use 5-bit binary numbers, even though both 12 and 9 could be represented by 4-bit unsigned numbers.

Is there overflow in the result? It is easy to recognize if there is overflow (out of range) in an unsigned binary addition operation because you simply end up with a result that has more bits than the addends. For example, $1011 + 0111 = 10010$. In subtraction, or S2C addition, overflow is harder to evaluate. This is the subject of the next section below.

EXAMPLE: Binary subtraction

Assume all numbers are 4-bit S2C binary numbers.

1. Evaluate 0011 - 1100

First, find the 2C of 1100, $[1100]^* = 0011 + 1 = 0100$

Now add: $0011 + 0100 = 0111$

Finally, it’s always good to check by comparing with decimal evaluation.

$0011_2 = 3_{10}$ and $1100_2 = -4_{10}$ (remember, all numbers are assumed in S2C form)

$3 - (-4) = 7 = 0111_2$, so it checks.

2. Evaluate 0111 - 0101

First, find the 2C of 0101, $[0101]^* = 1010 + 1 = 1011$

Now add: $0111 + 1011 = 10010$, but this has 5 bits! Is it wrong? Is it overflow? See next section.

Truncate to 4 bits: $10010 \rightarrow 0010$, the final answer

Now let’s check with decimal values. $0111_2 = 7_{10}$ and $0101_2 = 5_{10}$

$7 - 5 = 2 = 0010_2$, so it checks. This verifies we were right to truncate the result to 4 bits.

It is important to note that you can perform both addition and subtraction operations using S2C form numbers. In fact, that is one of the main benefits to S2C form. Once values are in this notation, the arithmetic unit (hardware) does not have to “think” at all; it simply evaluates $A+B$ for additions and $A+[B]^*$ (or $A+B'+1$) for subtractions.

2’s Complement Overflow: Recall that an n-bit S2C form number, N, can represent values in the range of

- $-2^{n-1} \leq N \leq 2^{n-1} - 1$

Values outside of the representable range for a given number of bits are said to generate 2’s *complement overflow*. For example, consider a 4-bit S2C binary number which can represent values between -8 and 7. What if the result of an operation were 11? The number 11 can be represented by 4 bits, but it is outside the range of 4-bit S2C numbers. A result of $11 = 1011_2$, would end up looking like a negative number, -5 in this case. Certainly $11 \neq -5$, but how can we detect that there was a 2’s complement overflow error? More importantly, what algorithm could be used so this detection could be achieved in hardware?

Important Notes:

- 2C overflow can only occur when operating on signed 2's complement form numbers.
- Unsigned numbers can generate a carry out overflow (often just called "carry out") but 2C overflow should be ignored for unsigned numbers.
- A carry out, or extra bit, should be ignored when S2C numbers are used, as demonstrated in the 2's complement overflow examples below.
- 2C overflow can occur in either addition or subtraction (see examples below) and should always be checked with S2C numbers are used.

EXAMPLE: Arithmetic with 2C overflow

Assume all numbers are 4-bit S2C binary numbers.

1. Evaluate 1110 - 0111

First, find the 2C of 0111, $[0111]^* = 1000 + 1 = 1001$

Now add: $1110 + 1001 = 10111$. Again we have 5 bits! As discussed below, this 5th carryout bit should be ignored. The result is then 0111.

Now let's check with decimal values. $1110_2 = -2_{10}$ and $0111_2 = 7_{10}$

$-2 - 7 = -9 \neq 0111_2$, because -9 is out of range. A 2C overflow must have occurred.

2. Evaluate 0111 + 0110 (notice this is addition)

Because this is addition, we do not take the 2C of 0110

Add: $0111 + 0110 = 1101$. No 5th carryout bit this time.

Check with decimal values. $0111_2 = 7_{10}$ and $0110_2 = 6_{10}$

$7 + 6 = 13$. Because we are using S2C numbers, $1101 = -3 \neq 13$.

A 2C overflow must have occurred because 13 is out of range for 4-bit signed binary numbers.

Detecting 2's Complement Overflow: 2C overflow can be detected by looking only at the sign bits of the two operands and the result. Consider two numbers A and B added to form sum S.

$$\begin{array}{r} A_{n-1}A_{n-2}\dots A_0 \\ + B_{n-1}B_{n-2}\dots B_0 \\ \hline S_{n-1}S_{n-2}\dots S_0 \end{array}$$

The sign bits are A_{n-1} , B_{n-1} , and S_{n-1} . The algorithm for detecting 2C overflow can be described several different ways. See which one makes the most sense to you.

2C overflow rule: A 2C overflow will occur only when 1) the sign of both operands are the same ($A_{n-1} = B_{n-1}$) and 2) the sign of the operands are different from the sign of the sum ($A_{n-1} = B_{n-1} \neq S_{n-1}$).

Pseudo code algorithm: The pseudo code for detecting 2C overflow is:

```
if  $A_{n-1}$  not same as  $B_{n-1}$ ; if sign bits of operands are different
    2C_overflow = false; then no 2C overflow
else if  $S_{n-1}$  same as  $A_{n-1}$ ; if sign of sum is same as sign of operators
    2C_overflow = false; then no 2C overflow
else 2C_overflow = false; if sum sign not same as operators sign
```

Logic expression: The following logic expression is evaluated in hardware to detect overflow.

- $Overflow = \overline{A_{n-1}} \cdot \overline{B_{n-1}} \cdot S_{n-1} + A_{n-1} \cdot B_{n-1} \cdot \overline{S_{n-1}}$

Practice Exercise

Prepare a truth table showing which combinations of A_{n-1} , B_{n-1} and C_{n-1} generate 2C overflow.

Important Notes:

- Numbers must be in S2C form
- Operand sign bits must be checked immediately *before the addition operation*. That is, if the operation is subtraction, sign bits should not be checked until the 2C operation is complete.
- If the addition operation results in a carry out overflow (extra bit in result), the carry out bit must be ignored and should not be confused with the sum carry bit.

EXAMPLE: Detecting 2C overflow

Determine if 2C overflow occurred in the examples above.

1. Does 1110 – 0111 produce 2C overflow?

From the examples above,

Now add: $1110 + 1001 = 10111$. Again we have 5 bits! As discussed below, this 5th carryout bit should be ignored. The result is then 0111.

1110	1) operand sign bits are the same (both 1)
+ 1001	2) operand sign does not match sum sign
10111	Thus, 2C overflow has occurred.

↑ sum sign bit

↑ carry out bit, ignore

2. Does 0111 + 0110 produce 2C overflow?

From examples above, $0111 + 0110 = 1101$

Operand sign bits match (both 0) and do not match sum sign (1), so **2C overflow has occurred.**

2C overflow in these problems is confirmed by knowing their results are out of range for 4 bits.

2. Logic Expression and Minimization

2.1 Boolean Arithmetic Properties

The following properties of Boolean logic are essential for manipulating and minimizing logic expressions. You should memorize these properties.

1 and 0 properties	$1 + x = 1$ $1 \cdot x = x$ $0 + x = x$ $0 \cdot x = 0$	self/inverse operand properties	$x + \bar{x} = 1$ $x + x = x$ $x \cdot \bar{x} = 0$ $x \cdot x = x$
distributive properties	$a \cdot b + a \cdot c = a \cdot (b + c)$ $(a + b) \cdot (a + c) = a + b \cdot c$	derivable properties	$a + \bar{a} \cdot b = a + b$ $a + ab + ac = a$

Logic Operator Priority: Standard arithmetic obeys the following rules of operator priority. 1) operations in parentheses, 2) multiple or divide, 2) add or subtract. Boolean logic follows the same priority with multiply represent by AND and add represented by OR.

2.2 Logic Expressions and Minimization

Minterm Notation: A logic expression of n variables (literals) can be expressed as a **sum of product** terms. The product terms, also referred to as **minterms**, represent all possible “product” (AND) combinations of the variables. For n variables, there are 2^n minterms.

EXAMPLE: Minterms

Find the minterms for a logic expression with 3 variables X, Y, and Z.

Since $n=3$, there are $2^3 = 8$ minterms.

$$\text{minterm0: } \bar{X} \cdot \bar{Y} \cdot \bar{Z} = m_0$$

$$\text{minterm1: } \bar{X} \cdot \bar{Y} \cdot Z = m_1$$

$$\text{minterm2: } \bar{X} \cdot Y \cdot \bar{Z} = m_2$$

$$\text{minterm3: } \bar{X} \cdot Y \cdot Z = m_3$$

$$\text{minterm4: } X \cdot \bar{Y} \cdot \bar{Z} = m_4$$

$$\text{minterm5: } X \cdot \bar{Y} \cdot Z = m_5$$

$$\text{minterm6: } X \cdot Y \cdot \bar{Z} = m_6$$

$$\text{minterm7: } X \cdot Y \cdot Z = m_7$$

Any logic expression can be expanded to a sum of products (SoP) form, where the “sum” is accomplished by OR operations and the “product” terms use AND operations, as in the minterms above. For example, we could write

$$\bullet \quad F = \bar{X} \cdot \bar{Y} \cdot Z + X \cdot \bar{Y} \cdot Z + X \cdot Y \cdot \bar{Z}$$

which is in SOP form. **Sigma notation** is an alternative SoP form that replaces the product terms with their minterm designations. For example

$$\bullet \quad F = m_1 + m_5 + m_6 = \sum_{XYZ} m_1, m_5, m_6 = \sum_{XYZ} (1,5,6)$$

Maxterm Notation: A logic expression of n variables (literals) can be expressed as a **product of sum** terms. The sum terms, also referred to as **maxterms**, represent all possible “sum” (OR) combinations of the variables. For n variables, there are 2^n maxterms.

EXAMPLE: Maxterms

Find the maxterms for a logic expression with 3 variables X, Y, and Z.

Since $n=3$, there are $2^3 = 8$ maxterms.

$$\text{maxterm7: } \overline{X} + \overline{Y} + \overline{Z} = M_7$$

$$\text{maxterm6: } \overline{X} + \overline{Y} + Z = M_6$$

$$\text{maxterm5: } \overline{X} + Y + \overline{Z} = M_5$$

$$\text{maxterm4: } \overline{X} + Y + Z = M_4$$

$$\text{maxterm3: } X + \overline{Y} + \overline{Z} = M_3$$

$$\text{maxterm2: } X + \overline{Y} + Z = M_2$$

$$\text{maxterm1: } X + Y + \overline{Z} = M_1$$

$$\text{maxterm0: } X + Y + Z = M_0$$

Notice that the maxterm designations are in the opposite order of minterm designations, e.g., the term with all complemented variables is maxterm7 but minterm0.

Any logic expression can be expanded to a product of sums (PoS) form, where the “product” is accomplished by AND operations and the “sum” terms use OR operations, as in the maxterms above. For example, we could write

$$F = (\overline{X} + Y + Z) \cdot (X + \overline{Y} + \overline{Z}) \cdot (X + Y + Z)$$

which is in PoS form. **Pi notation** is an alternative PoS form that replaces the sum terms with their maxterm designations. For example

$$F = M_4 + M_3 + M_0 = \prod_{XYZ} M_4, M_3, M_0 = \prod_{XYZ} (4,3,0)$$

Reduced Form: Note that SoP and PoS forms can be used to describe any Boolean logic expression of binary variables and are often useful. However, they are not necessarily the most reduced expression. We define reduced (or minimal) form as the expression with the fewest possible operations. Reduction is achieved using the Boolean logic properties described in section 3.1. For example

$$F = \overline{X} \cdot \overline{Y} \cdot Z + \overline{X} \cdot \overline{Y} \cdot \overline{Z} = \overline{X} \cdot \overline{Y} \cdot (Z + \overline{Z}) = \overline{X} \cdot \overline{Y}$$

where the leftmost expression is in SoP form with 5 operations and rightmost expression is the equivalent reduced form with only 1 operation.

Conversion Between Sigma and Pi Notation: There is a complementary relationship between minterm and maxterm expressions in sigma and pi notation, respectively. A sum of minterms expression is equal to a product of “missing” maxterms expression. In other words, the minterms that are missing from an expression, e.g., m_2, m_5, m_7 , represent the maxterms needed to form an equivalent expression. This is best illustrated by examples.

$$\sum_{ABC} m_1, m_3, m_4 = \prod_{ABC} M_0, M_2, M_5, M_6, M_7$$

$$\sum_{XY} m_1, m_2 = \prod_{XY} M_0, M_3$$

More generally, we can write $m(x) = \overline{M(x)}, M(x) = \overline{m(x)}$.

2.3 Reducing Logic Expressions

It is often desirable to obtain the minimal form of an expression. The two most common techniques for reducing logic expression to the fewest possible operations are the mathematical approach using Boolean arithmetic and logic properties and, the always popular, Karnaugh maps. This review assumes that you are familiar with both Boolean arithmetic and Karnaugh maps and will illustrate the process of reducing logic expressions through several examples.

EXAMPLE: Reducing logic expressions

1. Express $F = AB + A'$ in minimal reduced form.

Let's try a minterm approach.

Step 1: expand into minterm SoP form.

$$F = AB + A'(B+B) = AB + A'B' + A'B$$

Step 2: assign minterms, $F = m_0 + m_1 + m_3$, which is the canonical SoP form.

But, we can reduce this taking the complementary PoS form.

$$F = m_0 + m_1 + m_3 = M_2 \rightarrow F = A'+B, \text{ which is the most reduced form.}$$

2. Minimize $F = AB + A'$ using a Karnaugh map.

Here's the color-coded K-map for $F = AB + A'$

		A	
		0	1
B	0	1	0
	1	1	1

From the K-map we can see that $F = A'+B$, which matches the result above.

EXAMPLE: Reducing logic expressions

Find the minimal SoP expression for $F = \sum_{XYZ} (1,2,5,7)$.

Here's the K-map where each true (1) cell shows the relevant minterm. All other cells are false (0).

		XY			
		00	01	11	10
Z	0		m_2		
	1	m_1		m_7	m_5

m_7 and m_5 group to form XZ . m_5 and m_1 group to the term $Y'Z$. m_2 ($X'YZ'$) can't be reduced. Thus, $F = XZ + Y'Z + X'YZ'$ is the reduced SoP form, but we can use Boolean arithmetic to reduce further.

$$F = XZ + Y'Z + X'YZ' = Z(X+Y') + X'YZ' \text{ is the minimal form.}$$

4. Appendix

Table A1. Basis Values

Hex	Binary	Unsigned Decimal	Signed 2C* Decimal	BCD
0	0000	0	0	0000
1	0001	1	1	0001
2	0010	2	2	0010
3	0011	3	3	0011
4	0100	4	4	0100
5	0101	5	5	0101
6	0110	6	6	0110
7	0111	7	7	0111
8	1000	8	-8	1000
9	1001	9	-7	1001
A	1010	10	-6	x
B	1011	11	-5	x
C	1100	12	-4	x
D	1101	13	-3	x
E	1110	14	-2	x
F	1111	15	-1	x

x = don't care
2C = 2's complement form

Table A3. Signed Value Forms

Decimal Value	Signed 2C Binary	Sign-Magnitude Binary
-8	1000	n/a
-7	1001	1111
-6	1010	1110
-5	1011	1101
-4	1100	1100
-3	1101	1011
-2	1110	1010
-1	1111	1001
0	0000	0000, 1000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	0101
6	0110	0110
7	0111	0111

Table A2. Powers of 2

n	2 ⁿ	Shorthand
0	1	K=1024
1	2	M=1024K
2	4	G=1024M
3	8	
4	16	
5	32	
6	64	
7	128	
8	256	
9	512	
10	1,024	1K
11	2,048	2K
12	4,096	4K
13	8,192	8K
14	16,384	16K
15	32,768	32K
16	65,536	64K
17	131,072	128K
18	262,144	256K
19	524,288	512K
20	1,048,576	1M
21	2,097,152	2M
22	4,194,304	4M
23	8,388,608	8M
24	16,777,216	16M
26	67,108,864	65M
28	268,435,456	256M
30	1,073,741,824	1G