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

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 $N_r = A_{n-1}r^{n-1} + A_{n-2}r^{n-2} + ... + A_0$, where

- ***A*** ≡ coefficient (varies with number base)
    - Base 10: ***A*** ∈ {0,1,…9}
    - Base 2: ***A*** ∈ {0,1}
    - Base 16: ***A*** ∈ {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*** ≡ 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*** ≡ position

© 2012, Andrew J. Mason

EXAMPLE: Expressing decimal number with multiple digits

For Base 10: $934_{10} = 9\times10^2 + 2\times10^1 + 4\times10^0$

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: $1001_2 = 1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0 = 9_{10}$

8 4 2 1

2. Base 16: $4AC_{16} = 4\times16^2 + A\times16^1 + C\times16^0 = 4\,(256) + 10\,(16) + 12\,(1) = 1196_{10}$

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 → 1011 1111

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

$1011 = 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 = 11_{10} = B_{16}$

$1111 = 1\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0 = 15_{10} = F_{16}$

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 → 1000 1011 1100 0111

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

$1000 = 1\times2^3 + 0\times2^2 + 0\times2^1 + 0\times2^0 = 8_{10} = 8_{16}$

$1011 = 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 = 11_{10} = B_{16}$

$1100 = 1\times2^3 + 1\times2^2 + 0\times2^1 + 0\times2^0 = 12_{10} = C_{16}$

$0111 = 0\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0 = 7_{10} = 7_{16}$

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

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

$16^2 = 256$ → 1592 / 256 = 6 (and some decimals) [Remember this digit: $\underline{6}\times16^2$]

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

$1592 - 6\times16^2 = 56$, 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.

$16^1 = 16$ → 56 / 16 = 3 (and some decimals) [Remember this digit: $\underline{3}\times16^1$]

$56 - 3\times16^1 = 8$

and the final digit…

$16^0 = 1$ → 8 / 1 = 8 (with no decimals) [Remember this digit: $\underline{8}\times16^0$]

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 → 0111	0111	0111_2 → 7_{16}	7
+ 4	4 → 0100	+ 0100	0100_2 → 4_{16}	+ 4
11_{10}		1011_2		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
+ 2C
CF_{16}

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) $A3_{16} \rightarrow A\times16^1 + 3\times16^0 = 10\times16+3 = 163_{10}$

$+ 2C_{16} \rightarrow 2\times16^1 + C\times16^0 = 2\times16+12 = \underline{\ 44_{10}}$

$= 207_{10}$

$CF_{16} = C\times16^1 + F\times16^0 = 12\times16+15 = 207_{10}$ 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 → 0111	0111	The circled digit is an out of range overflow because it requires more than 4 bits to represent the value.
+11	11 → 1011	+ 1011	
18_{10}		$\boxed{1}0010_2$	Looking at the decimal value (18) we should expect this, because $18 > 15 = 2^4 - 1$

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 represetable 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 ½ 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 → 1, 1 → 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-digital 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 → 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 → 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:

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

For example, a 4-bit S2C Bin number has a range of: $-2^3 \le N \le 2^3-1$ → $-8 \le N \le 7$. 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

$28_{10} = 11100_2$, 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 → 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}, 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 → 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 = $+35_{16}$ = 3x16 + 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.

<u>When should you perform 2C operations?</u> 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) → 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 → 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

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

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?

 © 2012, Andrew J. Mason

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*.

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

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 An-1 not same as Bn-1; if sign bits of operands are different
     2C_overflow = false; then no 2C overflow
else if Sn-1 same as An-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.

- 2C $Overflow = \overline{A_{n-1}} \cdot \overline{B_{n-1}} \cdot S_{n-1} + A_{n-1} \cdot B_{n-1} \cdot \overline{S_{n-1}}$

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	$x+\overline{x}=1$ $x+x=x$ $x\cdot\overline{x}=0$ $x\cdot x=x$
distributive properties	$\mathrm{a}\cdot\mathrm{b}+\mathrm{a}\cdot\mathrm{c}=\mathrm{a}\cdot(\mathrm{b}+\mathrm{c})$ $(\mathrm{a}+\mathrm{b})\cdot(\mathrm{a}+\mathrm{c})=\mathrm{a}+\mathrm{b}\cdot\mathrm{c}$	derivable properties	$a+\overline{a}\cdot b=a+b$ $a+ab+ac=a$

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.
>
> $\text{minterm0}: \overline{X}\cdot\overline{Y}\cdot\overline{Z}=m_0$
> $\text{minterm1}: \overline{X}\cdot\overline{Y}\cdot Z=m_1$
> $\text{minterm2}: \overline{X}\cdot Y\cdot\overline{Z}=m_2$
> $\text{minterm3}: \overline{X}\cdot Y\cdot Z=m_3$
> $\text{minterm4}: X\cdot\overline{Y}\cdot\overline{Z}=m_4$
> $\text{minterm5}: X\cdot\overline{Y}\cdot Z=m_5$
> $\text{minterm6}: X\cdot Y\cdot\overline{Z}=m_6$
> $\text{minterm7}: X\cdot Y\cdot Z=m_7$

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

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

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

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

© 2012, Andrew J. Mason

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.
>
> $$\begin{aligned}
> &\text{maxterm7}: \overline{X}+\overline{Y}+\overline{Z}=M_7\\
> &\text{maxterm6}: \overline{X}+\overline{Y}+Z=M_6\\
> &\text{maxterm5}: \overline{X}+Y+\overline{Z}=M_5\\
> &\text{maxterm4}: \overline{X}+Y+Z=M_4\\
> &\text{maxterm3}: X+\overline{Y}+\overline{Z}=M_3\\
> &\text{maxterm2}: X+\overline{Y}+Z=M_2\\
> &\text{maxterm1}: X+Y+\overline{Z}=M_1\\
> &\text{maxterm0}: X+Y+Z=M_0
> \end{aligned}$$

Notice that the maxterm designations are in the opposite order of minterm designations, e.g., the term will 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

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

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

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

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.

$F = m_0 + m_1 + m_3 = M_2 \rightarrow F = A'+B$, 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' is the minimal form.

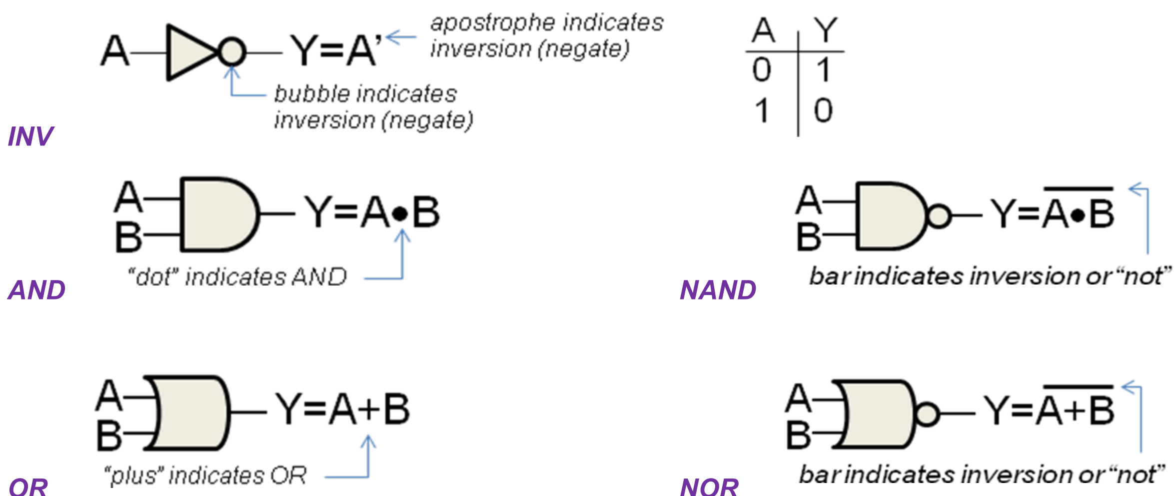
 © 2012, Andrew J. Mason

3. Digital Logic Review

3.1 Combinational Logic Gates

INV, AND, OR, NAND, NOR

The INV (invert), AND, OR, NAND, NOR logic gates form the basic logic functions for all digital circuits. In fact, it is important to note that all digital logic circuits, from flip flops to microprocessors, can be formed from only INV, NAND and NOR gates. Gate symbols and logic truth tables for the INV (invert), AND, OR, NAND, NOR are shown below.



inputs		AND	NAND	OR	NOR
A	B	A•B	$\overline{A•B}$	A+B	$\overline{A+B}$
0	0	0	1	0	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	0	1	0

Truth table for AND, NAND, OR, and NOR logic functions.

DeMorgan's Relations

An important relationship between AND, OR, NAND and NOR logic functions are described by the DeMorgan's relations described and illustrated below. These equivalencies can be used to simplify logic expressions using Boolean algebra or to simplify logic circuits by swapping one gate for an equivalent gate.

NAND-OR rule: "An AND with inverted output (NAND) is the same as an OR with inverted inputs"

$$\overline{A \bullet B} = \overline{A} + \overline{B}$$

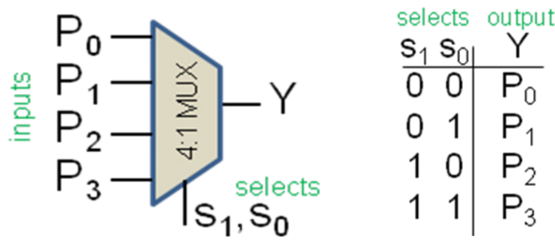
NOR-AND rule: "An OR with inverted output (NOR) is the same as an AND with inverted inputs"

$$\overline{A + B} = \overline{A} \bullet \overline{B}$$

MUX

© 2012, Andrew J. Mason

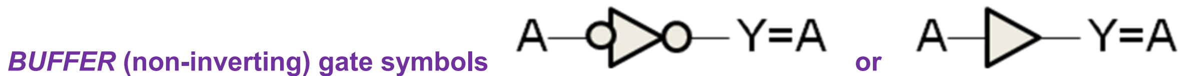
A multiplexer (MUX) routes multiple inputs signals into fewer (or a single) output using digital select lines. *n* select lines can multiplex 2^n inputs.



4:1 MUX, 4 (inputs) to 1 (output) MUX: symbol and truth table

BUFFER

A buffer is used to increase the drive strength of a signal and/or to isolate output signals from the input signal. An inverter is a type of buffer; however we can also use non-inverting buffers.



BUFFER **(non-inverting) gate symbols**

TRI-STATE BUFFER

Often in digital circuits is it desired to disable a signal so that it is neither 1 (hi) or 0 (low). In this case, the signal is essentially disconnected and said to be in an “open circuit” or “high impedance” (generally called “hi Z” because Z is the symbol for impedance) state. A basic circuit for achieving a “high Z” state is the tri-state buffer, which is an inverter with a third state where the output is disconnected or at high impedance. This third state is controlled by an enable signal where enable = false generally activates the high impedance state.

Tri-state buffer gate symbol and truth table.

Other variations of tri-state buffer include the inverting tri-state (basically an inverter with an enable), and active low tri-state buffers and inverters that are high impedance when *en = 1*.

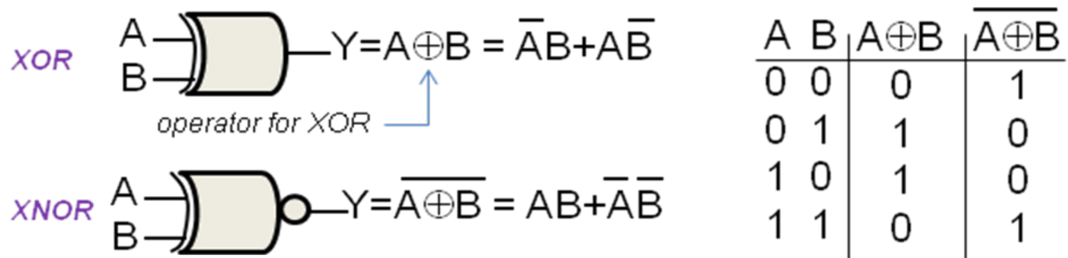
Tri-state inverter symbol

Active low tri-state inverter

XOR and XNOR

Exclusive OR (XOR) sets the output true (hi, 1) when the inputs are different. That is the output is true if *either* input A or input B, but not both, are true.

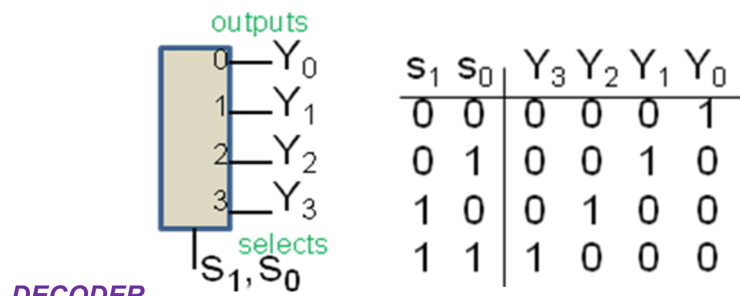
Exclusive NOR (XNOR) sets the output true when the inputs are the same. XNOR is the inverse of XOR.



XOR and XNOR gate symbols and truth tables.

Decoder

Decoder circuits implement "one hot" code, where only one of many outputs is active-high (true) at a time. Decoders can also be implemented as active low circuits, where only the active output is low and all others are high. Decoders are commonly found in memory circuits to select which of many memory cells is active. The inputs to a decoder are binary encoded select lines as shown below.



4-output decoder: gate symbol and truth table.

3.2 Sequential Logic Circuits

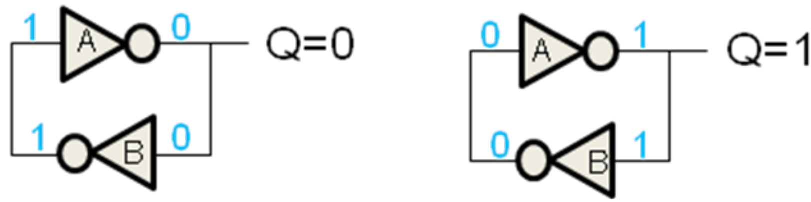
In the combination logic circuits shown above, the outputs are determined entirely by the current inputs. In contrast, there are many useful digital circuits with outputs that are determined (in part) by outputs from prior states (at an earlier time than present). These circuits are generally referred to as sequential logic circuits because their output response can be described by a time sequence pattern.

Sequential logic circuits are said to have "memory" because their outputs are based on states from an earlier time. The ability of sequential circuits to "store" a state for use in the future is critical to the operation of many common digital circuits. Unlike combinational circuits that are described by truth tables, sequential circuit often use state tables to describe what the next state will be for all possible input (and prior state) combinations.

Latch

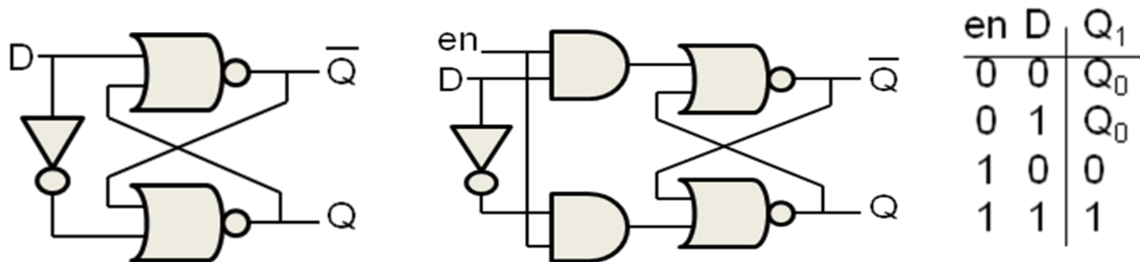
A latch is a circuit that has two stable states and can be used to store state information at one or two outputs. The latch can change states by applying signals to one or more control inputs. Because the state is actively stored by the latch circuit, the output value (state) will be maintained as long as power is applied to the circuit. Latches are often incorporated in static (will not change state over time) memory and are a building block in clock-triggered flip flops.

One of the most basic latch structures is the static digital latch that can be formed by *cross coupling* two inverters in a positive feedback arrangement as shown below. This circuit forces itself to maintain the data value at the output, Q, as follows: If the input to inverter A is a 1, it generates a 0 at its output, which is also the input of inverter B. Inverter B in turn generates an output of 1, which is also the input of inverter A and the circuit achieves a stable state shown in the figure below. Alternatively, if the input to inverter A is a 0, the logic is reversed an a second stable state is achieved. Because this circuit has two stable states, it is often called a bistable circuit.



Circuit schematic for static digital latch (or bistable circuit). The two possible states are shown.

The static digital latch shown above is of limited practical use because it does not provide a means to change the state. To provide access to the latch to set the value it will store, a modified circuit is needed with an input that can set the latch value. The D-latch is a simple circuit that allows the output Q to be set by input D. Functionally, the D-latch is simply an inverter, but positive feedback within the latch structure allows the output to hold its value even if the input is removed, something an inverter cannot do. A D-latch can be implemented using multiple configurations of combinational logic gates, and a simple NOR-based D-latch is shown below. To improve the utility of the D-latch, an enable input can be added that permits the latch to either a) set the output *Q* based on input *D* or b) hold *Q* to the internal value using positive feedback, in which case input *D* is disabled and will not affect the output. This is generally called a gated D-latch because the enable acts as a gate to let *D* pass or not.

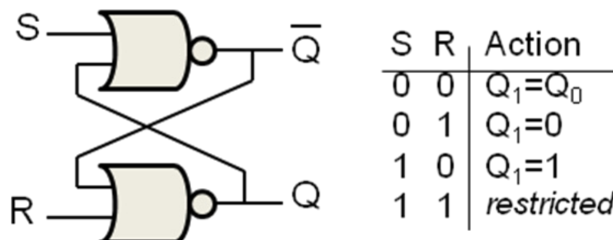


en	D	Q_1
0	0	Q_0
0	1	Q_0
1	0	0
1	1	1

Circuit schematic for a D-latch (left) and gated D-latch (center), and state table for gated D-latch (right).

The state table for the D-latch with enable is shown above where the next-state output, Q_1, is defined in terms of the last-state output, Q_0. When *en* = 0, the circuit is in hold mode and the next-state output is held to the last-state value. When *en* = 1, the input is enabled and the output is equal to input *D*.

Another common latch is the SR latch, which has the storage capability inherent to latches but operates very differently from the D-latch. Here, S stands for *set* and R stands for *reset* and the SR latch has three states, a hold state, a set state, and a reset state. When inputs *S* and *R* are both low, the latch is in hold state and the output remains constant. If *S* goes hi (while *R* is low) the output is set to 1, and if *R* goes hi (while *S* is low) the output is reset to 0. The SR latch can be formed using only two NOR gates as shown below. Having similar operation, an S'R' latch can be formed using only two NAND gates with the set and reset functions becoming active low. The problem with both SR and S'R' latches is that they have a forth "restricted state" that generates improper outputs, and this state must be avoided when using these latches, limiting their practical value.



S	R	Action
0	0	$Q_1=Q_0$
0	1	$Q_1=0$
1	0	$Q_1=1$
1	1	*restricted*

Circuit schematic and operation table for a SR latch.

Notice that the D-latch has the same feedback structure as the SR latch. It avoids the restricted state of the SR latch by forcing the two inputs to the NOR feedback structure to opposite values, but does not exhibit the set/reset capability of the SR latch.

 © 2012, Andrew J. Mason

But why, Professor?

Why does the SR latch have a restricted state? It won't blow up or anything will it?

If you take a moment to observe how these latches work, you'll find you can figure out their state tables on your own as long as you know how simple INV, AND and OR gates work.

Let's start with the SR latch and begin by assuming S, R, and Q are 0. Recall that the NOR gate has a low output unless both inputs are low, in which case the output is hi. So, R and Q low force Q' to hi, and with S low and Q' hi, Q is low thus holding its original low value. Now let Q be hi and start over and you'll find Q' is low and Q stay hi, again holding the state. This confirms the action of the S=R=0 state.

Now look at R hi while S is low. Because R is hi, Q has to be low (NOR output is always low unless both inputs are hi). With Q and S low, Q' is hi. This is the 'reset' state, where Q is forced to 0.

Similarly, when S is hi while R is low, Q' is forced to low and thus Q is hi. This is the 'set' state, where Q is forced to 1.

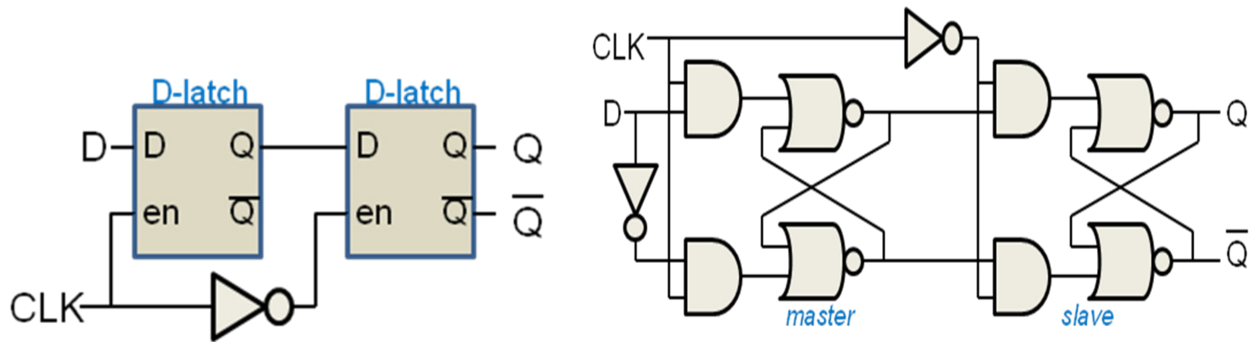
Finally, there is the restricted state. If both S and R are hi, the output of both NOR gates must be low, so both Q and Q' would be low. Because Q' is no longer the inverse of Q, this is not a reliable state. But the circuit is logically stable and nothing should explode!

Flip Flop

Latches are referred to as level-triggered because their outputs can transition any time the input level (logic value) changes. It is often desirable in digital circuit design to use circuits that only evaluate their outputs during the transition of an input signal. Such edge-triggered circuits often use a clock signal to determine when their output should be evaluated, and the output is set immediately after the edge-triggering event and held at that value until the next edge-trigger event, regardless of changes in any other input levels. A flip flop is an edge-triggered storage circuit that behaves similar to a latch but only changes input in the rising, or falling, edge of a triggering input. When multiple circuits are triggered by the same signal, they become synchronized, i.e. their outputs change at the same time. This synchronizing trigger signal is generally referred to as a clock, which is a periodic square wave that sets the speed of operation in most sequential digital circuits.

Flip flops (FF) are data storage elements for synchronous circuits that have circuit blocks triggered simultaneously and periodically by a clock signal. FFs are used to store the logic state of a signal until the next rising (or falling) edge of the clock signal when the output is reevaluated and stored again. They are typically controlled by one or two input signals and a clock, and they typically have differential (inverted) outputs *Q* and *Q'*. Some FFs include asynchronous set or reset input signals that immediately force the output hi or low, respectively, regardless of the state of the clock.

The better understand the operation of a FF, consider the master-slave D-type FF composed of two cascaded gated D-type latches. The output of the master block is connected to the input of the slave block, and the enable signal, which we'll now consider to be a clock (*CLK*), is inverted between the master and slave blocks. Each of the latches is individually level-trigged so that when the master is enabled the slave is not (is in hold state), and when the master is not enabled (is in hold state), the salve is enabled. Thus, when *CLK* is HI, input *D* can pass through the master block but will be held from entering the slave block. Then the *CLK* goes LO, and inputs are blocked from the master while the master's output is enabled to pass through the slave. Notice that the output *Q* (of the slave latch) only changes as *CLK* goes LO; because input *D* is blocked at the master stage while *CLK* is low, the input to the slave cannot change while *CLK* is low and output *Q* is thus constant until the next low-going clock edge. Because of the master-slave interactions, the output is only triggered by the low-going, or falling, edge of the clock, and this FF is considered to be *falling edge triggered*. Similarly, by inverting the clock (or moving the *CLK* input inverter from the slave to the master stage), a *rising edge triggered* FF can be implemented.



Block-level (left) and gate-level (right) schematic for D-type master-slave flip flop.

Just as there are a variety of latches (e.g., D-latch, SR latch), several variations of FFs are available. In a D-type FF, the value of input D is transferred to the output Q at the rising (or falling) clock edge and held there until the next rising (or falling) edge. In contrast, a JK FF implements different functions at the rising (or falling) clock edge based on the values of inputs *J* and *K* and then holds the output value until the next rising (or falling) edge. Similar to the SR latch, the JK FF has hold, set, and reset states. However, rather than having a useless (and troublesome) restricted state like the SR latch, the JK FF implements a forth state that toggles (or inverts) the output. The last FF described here is the T-type FF, or toggle FF. This can be considered as a simplified version of the JK FF that will either a) hold the output or b) toggle (invert) the output. It cannot hold/store any specific input value like the DFF; it can only hold the prior value or invert it, which can be useful in some digital circuits.

D	Q_1
0	0
1	1

J	K	Q_1
0	0	Q_0
0	1	0
1	0	1
1	1	$\overline{Q_0}$

T	Q_1
0	Q_0
1	$\overline{Q_0}$

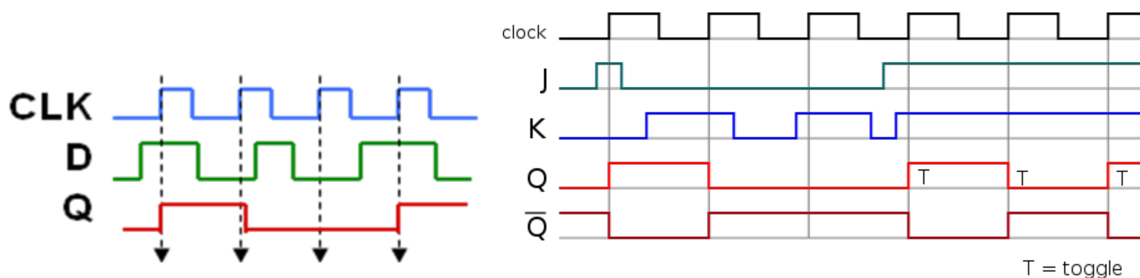
State tables for D-type, JK, and T-type flip flops.

In some ways the JK FF can "do more" than the D-type FF; it has 4 operations while the DFF only has one. However, the JK FF cannot directly store/hold an input value like a DFF, which is the most commonly required function of a FF in most digital circuits. For this reason, the DFF is the most common FF used in VLSI circuits.

It should be noted that, by setting K to J', the JK FF can implement the function of a D-type FF. Similarly, by adding an AND and two OR gates to a DFF, the function of a JK FF can be realized. Also, a JK FF (or a DFF converted to JK FF function) can realize the function of a T-type FF by simply connecting the J and K inputs together. Thus, it is possible to implement the functions of D, JK, and T flip flops with either a JK FF or a DFF (and some additional logic).

Timing Diagrams

It is often necessary to interpret timing diagrams that show the state of inputs and outputs over time. For FF circuits and registers (described below), this typically involves observing how the outputs change at the rising or falling edge of a clock. For rising-edge-triggered FFs, the outputs can only change on the high-going edge of the clock signal; the resulting value will then be held until the next rising edge when the outputs should be checked again. Similarly, falling-edge-triggered circuits are checked at the low-going clock edge. To check the outputs, simply follow the state transition rules of the state table for whatever type of FF is being observed. The examples below show timing diagrams for D-type and JK FFs. For ECE331, **you should be capable of finding the output logic levels over time for any given set of FF inputs**.



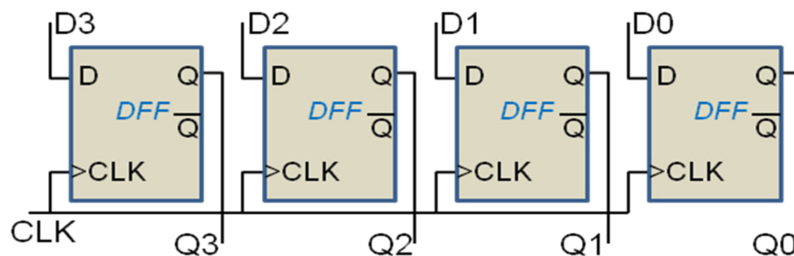
Timing diagrams for rising-edge triggered (left) D-type FF and (right) JK FF. Note that the output only changes when the clock is going high. Any transitions of inputs between clocks should be ignored.

Factoid!?!

The rising edge and falling edges of a clock are often referred to as the positive and negative edges. Don't let the terminology confuse you!

3.3 Data Registers

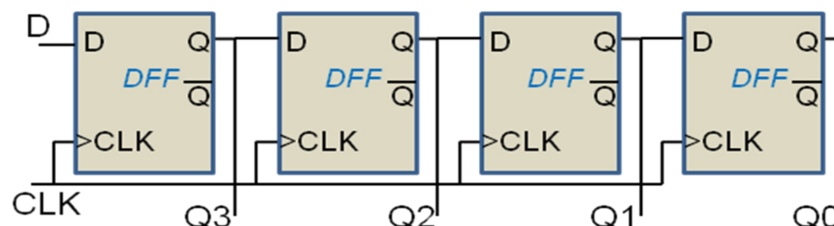
A register is a circuit with a bank of memory elements that is used to store (or manipulate) blocks of binary data. Registers typically have 8, 16 or 32 memory elements to store bytes or words of digital data. Registers are often described by whether their inputs/outputs are written/read parallel (all at the same time) or serially (one bit at a time in sequence). The example below shows a simple 4-bit data register formed from four DFF gates. The inputs D0, D1, D2, D3, often written D(3:1), go into the four FFs in parallel, and the outputs Q(3:1) come out in parallel. The clock signal is shared by all bits of the register so that they all read in new values and write out resulting outputs simultaneously. Whether this data register is positive or negative edge triggered depends on the triggering of the FF blocks.



Block diagram of a 4-bit parallel-in, parallel-out data register formed with DFF gates.

Shift Register

Consider what would happen if we modified the 4-bit register above so that the input to each block was taken from the output of the register to its left, as shown in the diagram below. Bit 2 (Q2) would now get its input from the output Q3, and bit 1 (Q1) would come from Q2, etc. Thus, data entering at D (bit 3) would be shifted through the register from left to right. This would be described as a serial-in, parallel out data register. Because data is shifted from left to right, it could also be called a shift-right register. With some additional logic, the register could be modified to be capable of shifting data both left and right.



Block diagram of a 4-bit serial-in, parallel-out shift register that can only shift right.

© 2012, Andrew J. Mason

> ***Thought Exercise***
> In the 4-bit serial-in shift register shown above, to load in a 4-bit word, which data bit should be input first, D0 or D3?

Shift and Rotate Functions

Common functions implemented within the CPU of a microprocessor are the ability to shift and rotate data to the left and to the right. In the shift and rotate functions, all bit values are moved by one (or more) places to the left or right. However, this leaves a void where the first bit was moved from. For example, assume the shift register above is loaded with 4-bits of data and disconnected from input D. What happens then when the data is shifted to the right? Bit 3 goes to 2, bit 2 goes to 1, and bit 1 goes to 0, but what happens to bit 0 and what comes into bit 3? In a shift function, the bits shift as just described, bits shifted out the front end (bit 0 in this example) are lost (ignored), and a preset value (normally 0) is loaded into the back end (the new bit 3 in this example). In contrast, for a rotate function, the bits going out the front end are rotated back into the opposite end, so no bits are lost they are just cycled around. In most CPUs, both shift and rotate functions can execute multiple-bit cycling at once. For example, shift left by 2, or rotate right by 4.

Illustrations of shift right and rotate left functions.

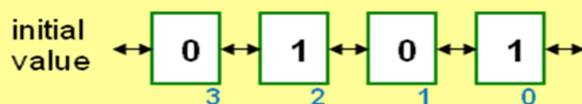
- ***Shift*** move each bit (left or right) to adjacent register, load in preset value (normally 0) into open registers
- ***Rotate*** move each bit (left or right) to adjacent register, rotate exiting bits back into other side of register

It is worth noting that shifting a data byte left by 1 is equivalent to multiplying by 2 (and shifting by 2 is like multiplying by 4, etc). Similarly, shifting a data byte right by 1 is equivalent to dividing by 2.

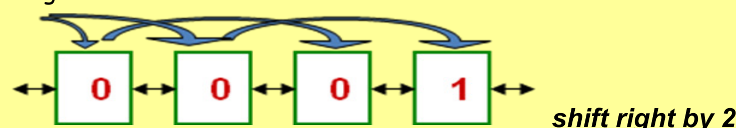
EXAMPLE: Shift and Rotate functions

Assume you have a 4-bit shift register loaded with the initial values below. What values would be in each bit of the register after

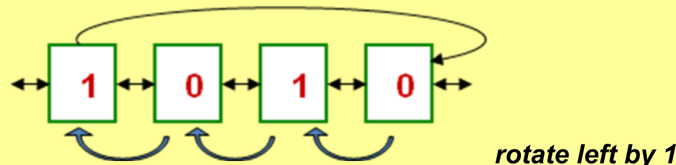
a) shift right by 2, b) rotate left by 1



a) For shift right by 2, bits would be moved to the right by 2 places and 0 values (assumed) would be loaded into any registered emptied by the shift. Thus, position 3 would move to 1, 2 would move to 0, and 1 and 0 would be lost. Positions 3 and 2 would get a new 0 value.



b) For rotate left by 1, bits would be moved to the left by 1 place and the left-most bit (3) would be rotated into the bit 0. Thus, bit 0 goes to 1, bit 1 goes to 2, bit 2 goes to 3, and bit 3 goes to position 0.

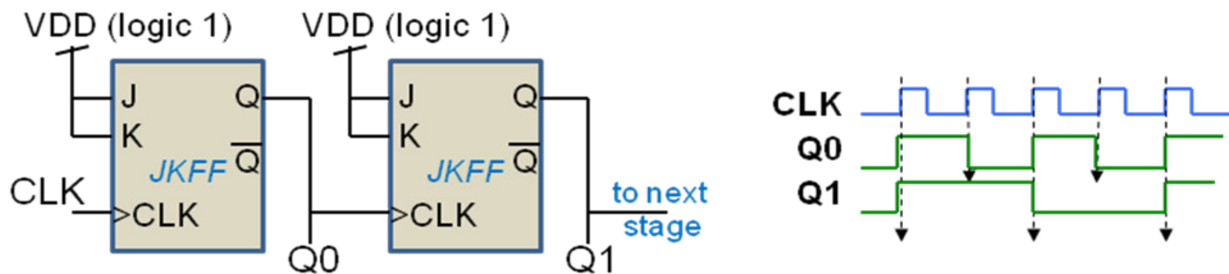


rotate left by 1

Counters

A counter is a special type of register that counts the number of times an input signal changes value, from 1 to 0 or 0 to 1 depending on the polarity of the counter. Counters are commonly employed to count clock cycles in timer hardware. For a periodic input like a clock, each successive bit of the counter will toggle (change state) at ½ the frequency of its input bit. Thus, counters can also be used to divide higher frequency clocks down to lower frequencies.

There are many ways to implement counters using various circuit configurations with flip flops. In a binary ripple counter, a T-type (toggle) FF is used as the first stage and set to toggle mode. Because the output will toggle value only when the input clock shows a rising (or falling) edge, the output would be ½ the frequency of the input clock. That output is then used to clock another T-type FF so that its output changes at ½ its input frequency, which would be ¼ the frequency of the main clock. The structure continues through as many states as needed to count the desired duration or generate the desired frequency. In a synchronous counter, all FFs are clocked at the same time which allows more precise timing. Logic gates are included between bits to implement the desired counting function. In a counter-by-n counter, *m* FFs are used to count from 0 to 2^m-1 before the counter resets to zero and starts over.



A 2-bit binary ripple counter and associated timing diagram showing frequency decreasing.

> *EXAMPLE: Counters*
>
> **How many counter bits (flip flops) would be needed to count for 1ms if the input clock is 1MHz?**
>
> A 1MHz clock changes state every 1us, so 1000 clocks would be needed to count to 1ms. Since 2^{10} is 1024, anything less than 10 bits would not be able to count to 1ms. Thus, **10 counters bits are needed.**

3.4 Complete Set Concept

As noted at the beginning of section 3, all digital logic functions can be implemented with only INV, NAND, and NOR gates, and thus these gate are the primary tools in any VLSI designers toolbox. It is a cornerstone of VLSI design that all complex digital circuits are composed of more simple base gates. In practice, to optimize speed and power, many "building block" logic circuits like flip flops and adders are designed at the transistor level rather than at the gate-level, but it is possible to construct all logic functions from simple gates like NANDs and NORs. In fact, the complete set concept states that all logic functions can be formed using only NAND gates or only NOR gates. An entire microprocessor can be built using only NAND (or only NOR) gates! As proof of this concept, note that registers are composed of flip flops and, as shown above, a D-type flip flops can be constructed from only INV, AND, and NOR gates. Thus, if one could show that INV, AND, and NOR could be implemented using only NAND gates it would prove that data registers could be implemented using only NAND gates. This proof will be explored in homework (aren't you excited!). But to get you started, here's an illustration showing how a NAND gate can implement the INV function.

A — $\bar{A}$ = A — $\overline{A \bullet A} = \bar{A}$

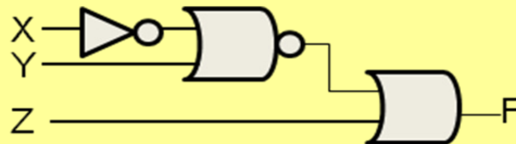
Bubble Pushing Technique

The circle symbols at the end of an INV or NAND gate are referred to as "bubbles". They represent an inversion (negation) operation. And just as the mathematical product of two negatives is a positive, two bubbles can be combined to remove them from a schematic. It is just like A = (A')' = A. Similarly, two bubbles can be added to a single node in a schematic without changing it functionally. These options may seem useless, but they can be used to simply a digital logic schematic. Recall that DeMorgan's relations show how a gate with an inverted output can be replaced by a gate with inverted inputs. Combining DeMorgan's equivalent circuits with bubble pushing techniques can often allow you to manipulate logic at the schematic level the same way you might use Boolean algebra principles to manipulate a logic expression. The following example helps to illustrate this concept.

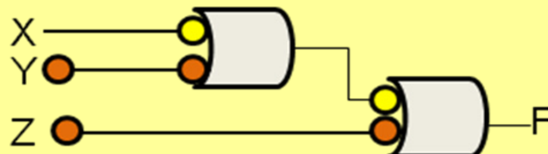
EXAMPLE: Complete Set Concept

This example illustrates the use of DeMorgan's relations and the bubble pushing technique to modify the gates used in a circuit so that it follows the complete set concept. The result is not necessarily the optimal circuit with the best performance, but it exercises techniques useful in the design of digital circuits.

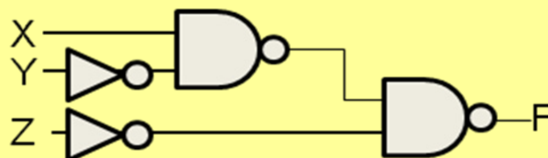
Manipulate the logic circuit below using DeMorgan's relations and bubble pushing to realize the same logic function using only NAND gates.



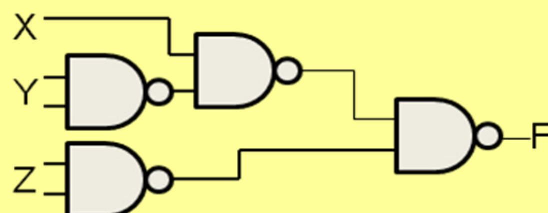
Step 1: By DeMorgan's, NAND is equivalent to OR with inverted inputs, so let's first try to get this circuit into only NAND and OR gates. We can slide the bubble on the NOR up to the input of the OR and replace the INV with a bubble at the input of the NOR. Then we can add bubble pairs to node Y and Z. The result looks like this, with yellow bubbles that were moved and orange that were added.



Step 2: The only logic gates remaining are ORs with inverted inputs, so by DeMorgans we can replace them in NANDs. That just leaves the bubbles at the Y and Z input, which we'll replace for now with INV gates.



Step 3: As shown in the Complete Set Concept section above, INV can be replaced by NAND with the inputs tied together. This gives the **final circuit using only NAND gates**.



 © 2012, Andrew J. Mason

4. Appendix

Table A1. Bases 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 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

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