DIGITAL HARDWARE AND MICROCONTROLLERS

Introduction

Digital hardware can perform a variety of functions in mechatronic system. Signal acquisition and processing, system monitoring and control, switching, and information display are such functions. Hardware implementation involves carrying out simple functions, actions, and tasks without any form of programmability. More complex and variable mechatronic tasks may require programmable digital devices (embedded digital computers) that are known as Microcontrollers; this is called software implementation. Of course, software implementation also requires digital logic hardware and some aspects of hardware implementation.

Hardware implementation has the following advantages:

- High speed
- Simplicity
- Low cost when produced in mass
- Smaller size

However, hardware implementation has a fixed function and this makes it to lack flexibility but software implementation on the other hand provides flexibility that comes with programmability and capability of implementing very complex task. Some of the disadvantages of software implementation are:

- Relatively low speed
- Larger size
- Higher cost

Digital devices, especially digital computers use digits (according to some codes) to represent information and some logic to process such information. In binary (or, two-state) logic, a variable can take one of two discrete states: True (T) or false (F). In binary number system, each digit can assume one of only two values: 0 or 1. A digital device may have to process both logical quantities and numerical data. The purpose of a digital circuit might be to turn on or off a device depending on some logical conditions. In some application, a

digital circuit might have to perform numerical computations on a measured signal (available in digital form) and then generate a control signal (in digital form). A digital device can perform such numerical functions as well, using the binary number system where each digit can assume one of only two values: 0 or 1

A digital circuit converts digital inputs into digital outputs. There are two types of logic devices, classified as either **Combinatorial logic devices** or **Sequential Logic devices**. **Combinatorial logic devices** are static devices where the present inputs completely (and uniquely) determine the present outputs without using any past information (history) or memory. In contrast, the outputs of **sequential logic devices** depend on the past values of the inputs as well as the present values. In other words, they depend on the time sequence of the input data, and hence some form of memory would be needed.

Microcontrollers are miniature digital computers of somewhat limited functionality that can be embedded in various locations of a mechatronic system.

Number Systems and Codes

The base or the radix (denoted by R) of a number is the maximum number of discrete values each digit of a number can assume. This is also equal to the maximum number of different characters (symbols) that are needed to represent any number in that system. For decimal or denary, R = 10, for binary system R = 2, for octal system R = 8, and for the hexadecimal system R= 16. We are quite familiar with the decimal system. The origin of this system is perhaps linked to the fact that a human being has 10 fingers. Also, 10 is a convenient and moderate number, which is neither too large nor too small. However, the binary number is what is natural for digital logic devices and digital computers.

Binary — Base 2

Octal — Base 8

Hexadecimal — Base 16

Conversion from Denary to Binary

Change the following number to binary

(a) 11 (b) 18 (c) 9.25 (d) 12.125

(a) Solution

2 |	11
2 |	5 + 1
2 |	2 + 1
2 |	1 + 0
	0 + 1 ↑

$\therefore 11 \equiv 1011_2$

(b)

2 |	18
2 |	9 + 0
2 |	4 + 1
2 |	2 + 0
2 |	1 + 0
	0 + 1 ↑

$\therefore 18 \equiv 10010_2$

(c) $9.25 \equiv 9 + 0.25$

2 |	9
2 |	4 + 1
2 |	2 + 0
2 |	1 + 0
	0 + 1 ↑

$9 \equiv 1001$

	0.25
0 |	2
	0.50
1 |	0.50
	2
	1.00
	0.00

$\therefore 0.25 \equiv .01$

$\therefore 9.25 \equiv 1001.01$

(d) $12.175 = 12 + 0.125$

2 |	12
2 |	6 + 0
2 |	3 + 0
2 |	1 + 1
	0 + 1 ↑

$\therefore 12 \equiv 1100_2$

	0.125
0 |	2
	0.250
0 |	0.250
	2
	0.500
1 |	0.500
	2
	1.000

$0.125 \equiv .001$

$\therefore 12.125 = 1100.001_2$

Conversion from Binary to Denary

Change the following to Denary

(a) 1011_2 (b) 10010_2 (c) 1001.01

(d) 1100.001_2

Solution

(a) $1011_2 = 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$

$= 8 + 0 + 2 + 1$

$\therefore 1011_2 \equiv 11$

(b) $10010_2 = 1\times2^4 + 0\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0$

$= 16 + 0 + 0 + 2 + 0$

$\therefore 10010_2 \equiv 18$

(c) $1001.01 = 1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0 + 0\times2^{-1} + 1\times2^{-2}$

$= 8 + 0 + 0 + 1 + 0 + \frac{1}{4}$

$\therefore 1001.01_2 \equiv 9.25$

(d) 1100.001_2

$1\times2^3 + 1\times2^2 + 0\times2^1 + 0\times2^0 + 0\times2^{-1} + 0\times2^{-2} + 1\times2^{-3}$

$8 + 4 + 0 + 0 + 0 + 0 + \frac{1}{8}$

$\therefore 1100.001_2 \equiv 12.125$

Addition and Subtraction of Binary Numbers

(a) $1101_2 + 1111_2 + 1011_2$

(b) $1011_2 + 1110_2 + 1111_2 + 1111_2$

Solution

(a)

$$\begin{array}{r} 1101 \\ 1111 \\ 1011 \\ \hline 100111_2 \end{array}$$

(b)

$$\begin{array}{r} 1011 \\ 1110 \\ 1111 \\ 1111 \\ \hline 110111_2 \end{array}$$

(c) $1101_2 - 111_2$

(d) $1110 - 101_2$

Solution

(c)
$$\begin{array}{r} 1101 \\ 111 \\ \hline 110_2 \end{array}$$

(d)
$$\begin{array}{r} 1110 \\ 101 \\ \hline 1001_2 \end{array}$$

Representation of Negative Number

Negative numbers can be represented in three ways

(i) Sign magnitude representation

(ii) One's complement

(iii) Two's complement

Sign Magnitude

CSE 271 — Introduction to Digital Systems
Supplementary Reading
Representation of Signed Numbers

There are many ways to represent signed numbers. Typically the MSB of a bit string is used to represent the sign (the sign bit). Since the MSB is used to indicate the sign (0=plus, 1=minus), an n-bit number can only represent nonnegative numbers from 0 to $2^{n-1}-1$ (instead of 0 to 2^n-1 as for unsigned numbers).

To ease the implementation of subtraction using digital circuits, we would also impose

Requirement A: The subtraction $N1-N2$ can be carried out by the addition of the two numbers $N1$ and $(-N2)$. □

Here the addition is carried out similarly to that of unsigned numbers. If the Requirement A is satisfied by the representation, then in designing a digital system, subtraction circuits need not be separately designed once the addition circuits are available.

Let us look at the following candidate representations for signed numbers.

Signed-Magnitude Representation

In the signed-magnitude representation, a number consists of a magnitude string and a symbol indicating the sign of the number. The sign symbol is at the MSB. The rest of the bits form the magnitude and are interpreted similarly to unsigned numbers. For example, the 4-bit words $0110_2 = 6_{10}$, $1101_2 = -5_{10}$. Now consider $6-5$. Direct subtraction yields $0110-0101=0001$. However, if we express it as $6+(-5)$ and carry out the addition, we have $0110+1101=10011$ and so the 4-bit sum word is 0011 (due to the 4-bit word length). Since $0011 \neq 0001$, the Requirement A is not satisfied.

1's Complement Representation

In the 1's complement representation, a nonnegative number is represented in the same manner as an unsigned number. A negative number $(-N)$ is represented by the 1's complement of the positive number N. The 1's complement of an n-bit number N is obtained by complementing each bit of N (or equivalently, by subtracting it from 2^n-1). For example, the 4-bit words $0110_2 = 6_{10}$, $0101_2 = 5_{10}$, and $1010_2 = -5_{10}$. Now consider $6-5$. Direct subtraction yields $0110-0101=0001$. However, if we express it as $6+(-5)$ and carry out the addition, we have $0110+1010=10000$ and so the 4-bit sum word is 0000 (due to the 4-bit word length). Since $0000 \neq 0001$, the Requirement A is not satisfied.

2's Complement Representation

Now we introduce the 2's complement representation which satisfies the Requirement A. Due to this reason, it is the most commonly used representation for signed binary numbers. In the the 2's complement number system, we have the following representations.

Nonnegative Numbers: Represented in the same manner as an unsigned number.

Negative Numbers: A negative number $(-N)$ is represented by 2's complement of the positive number N.

The 2's complement of an n-bit number N is obtained by subtracting it from 2^n. Note $2^n - N = \big[(2^n-1)-N\big]+1$ and the operation $\big[(2^n-1)-N\big]$ entails the complementing of each bit of N. So the 2's complement of N can simply be obtained by *complementing each bit of N and then adding* 1. The followings are some examples of 2's complement representations.

Examples. The 2's complement representation of the decimal number 6 is 0110. The 2's complement representation of -6 is obtained by the following procedure.

$$\begin{array}{rl} 6_{10} = & 0110 \\ & 1001 \quad \text{complement bits} \\ + & \quad 1 \\ \hline & 1010 = -6_{10} \end{array}$$

Note that the MSB 1 indicates that 1010 represents a negative number. □

In fact, the 2's complement number system negates a number by taking its 2's complement. So the complement operation can also be applied to a negative number representation to obtain the corresponding positive number representation[1]. For example

$$\begin{array}{rl} -6_{10} = & 1010 \\ & 0101 \quad \text{complement bits} \\ + & \quad 1 \\ \hline & 0110 = 6_{10} \end{array}$$

Remarks.

(a) Given a word size of n bits, the range of 2's complement binary numbers is -2^{n-1} through $2^{n-1}-1$.

(b) The 2's complement of an n-bit all 0 string is itself.

(c) The 2's complement of an n-bit string with all 0's except for the MSB being 1 is itself. For example, the complement of a 4-bit word 1000 is 1000 and it represents $-2^3 = -8$ and has no positive counterpart (since 8 is not with the range). □

Decimal Equivalent Values for 2's Complement Binary Numbers. Given a binary number in 2's complement representation, there are two methods for determining its decimal equivalent value.

Method 1: If the MSB is 0, then the number is nonnegative and its value can be determined similarly to an unsigned number. If the MSB is 1, then the number is negative and its absolute value can be determined by taking the 2's complement of the given negative number. For example, given a 4-bit number $N = 1101$. We apply the following procedure to determine the 2's complement of N (i.e., the negation of N).

$$\begin{array}{rl} \mathrm{N} = & 1101 \\ & 0010 \quad \text{complement bits} \\ + & \quad 1 \\ \hline & 0011 = 3_{10} \ \text{(2's complement of N)} \end{array}$$

Hence the decimal value of N is -3_{10}, i.e., $1101_2 = -3_{10}$.

Method 2: The decimal value for an n-bit 2's complement binary number is computed the same way as for an unsigned number using the formula of weighted summation of powers of 2, except that the power term corresponding to the MSB is (-2^{n-1}) instead of 2^{n-1}. For example, the decimal value of the 4-bit number $N = 1101$ can be computed as

$$N = 1 \times (-2^3) + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = -3_{10}$$

The justification of Method 2 is given in the footnote[2].

[1] To justify this, note that the 2's complement representation of an n-bit negative number $(-N)$ is given by $2^n - N$. Now take the 2's complement of $2^n - N$ yields $2^n - (2^n - N) = N$. This is equivalent to saying that the 2's complement of the representation of $(-N)$ gives us the representation of the corresponding positive number N.

[2] For positive numbers, the MSB is 0 so it has no contribution in evaluating the value. While for a negative number $(-N)$, its 2's complement representation is the unsigned binary representation for $D = 2^n - N$. Since the decimal value for D as an unsigned number is $D = \sum_{i=0}^{n-1} d_i \cdot 2^i$ with $d_{n-1} = 1$, we then have $-N = D - 2^n = 1 \times (-2^{n-1}) + \sum_{i=0}^{n-2} d_i \cdot 2^i$.

Sign Extensions. When dealing with hardware, we often need to increase the number of bits required to represent a signed number. In general, to extend an n-bit number to an m-bit number ($m > n$) which has the same decimal value, we simply pad the given n-bit number with $(m - n)$ copies of its MSB to its left to form the corresponding m-bit number. For example, given a 4-bit number $1110_2 = -2_{10}$, we can extend it to an equivalent 8-bit number $11111110_2 = -2_{10}$. Similarly, we can extend $0011_2 = 3_{10}$ to $00000011_2 = 3_{10}$.

Comparison of Different Representations

Now let us compare the aforementioned three representations by studying the following table for 4-bit numbers.

Decimal	2's Complement	1's Complement	Signed-Magnitude
7	0111	0111	0111
6	0110	0110	0110
5	0101	0101	0101
4	0100	0100	0100
3	0011	0011	0011
2	0010	0010	0010
1	0001	0001	0001
0	0000	0000 or 1111	0000 or 1000
-1	1111	1110	1001
-2	1110	1101	1010
-3	1101	1100	1011
-4	1100	1011	1100
-5	1011	1010	1101
-6	1010	1001	1110
-7	1001	1000	1111
-8	1000	—	—

From the above table, it can be observed why the 2's complement is preferred for arithmetic operations. If we start with 1000_2 (-8_{10}) and count up, we see that each successive 2's complement number all the way to 0111_2 (7_{10}) can be obtained by adding 1 to the previous one, ignoring any carries beyond the fourth bit position. The same cannot be said of signed-magnitude and 1's complement numbers. Because ordinary addition is just an extension of counting, 2's complement numbers can thus be added by ordinary binary addition, ignoring any carries beyond the MSB. The result will always be the correct sum as long as the range of the number system is not exceeded. This helps explain why the Requirement A is satisfied by 2's complement numbers. Moreover, note that the range of 2's complement numbers is larger than that of signed-magnitude and 1's complement (for which 0_{10} has 2 representations).

2's Complement Addition and Subtraction

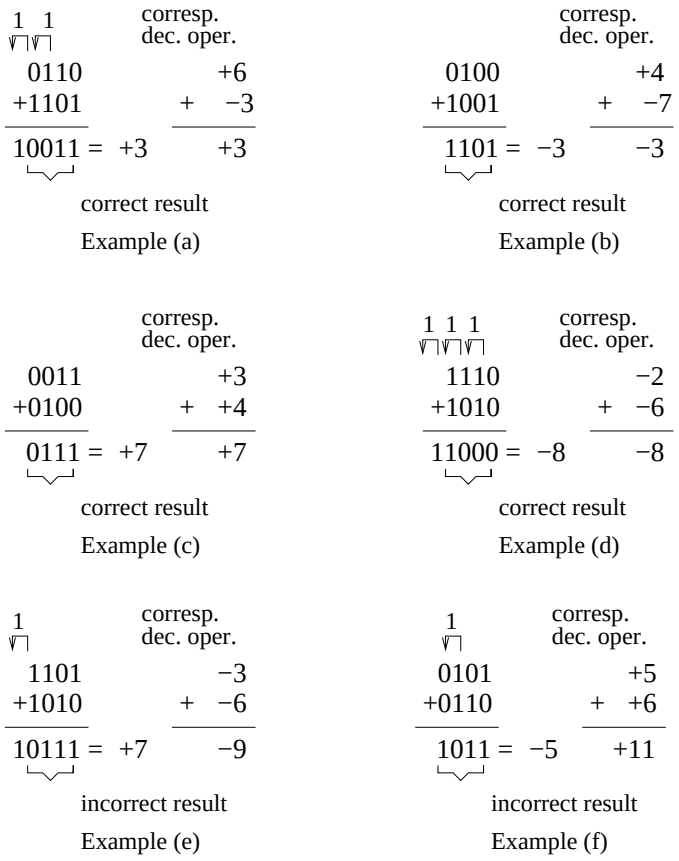
Since 2's complement numbers satisfy the Requirement A mentioned at the beginning of this handout, we only need to consider the addition of 2's complement numbers. As we have mentioned, 2's complement numbers can thus be added by ordinary binary addition. Some examples are given on the next page.

In Examples (e) and (f), the result is incorrect since the decimal value of the sum exceeds the range of 4-bit 2's complement number system. In such cases, *overflow* is said to occur. As can be observed from the examples, in general, the addition of 2's complement numbers has the following properties.

(a) Addition of two numbers with different signs can never produce overflow and thus the result is always correct (ignoring the carries beyond the MSB). Such is the case for Examples (a) and (b).

(b) An addition overflows if the two addends' signs are the same, but the sum's sign is different from the addends. Such is the case for Examples (e) and (f). In Examples (c) and (d), the sums have the same sign as the addends, so the results are correct.

(c) Finally, here is *an easy method to determine whether overflow occurs: overflow occurs if and only if the carry bit c_{in} into and c_{out} out of the sign position (i.e., the MSB) are different.* Such is the case for Examples (e) and (f).



Two's Complement

How can negative numbers be represented using only binary 0's and 1's so that a computer can "read" them accurately?

The concept is this: Consider the binary numbers from 0000 to 1111 (i.e., 0 to 15 in base ten).

$\underline{\underline{0}}001 \rightarrow \underline{\underline{0}}111$ will represent the positive numbers $1 \rightarrow 7$ respectfully

and, $\underline{\underline{1}}001 \rightarrow \underline{\underline{1}}111$ will represent the negative numbers $-7 \rightarrow -1$, respectfully.

In a computer, numbers are stored in **registers** where there is reserved a designated number of bits for the storage of numbers in binary form. Registers come in different sizes. This handout will assume a register of size 8 for each example.

It is easy to change a negative integer in base ten into binary form using the method of two's complement.

First make sure you choose a register that is large enough to accommodate all of the bits needed to represent the number.

Step 1: Write the absolute value of the given number in binary form. Prefix this number with 0 indicate that it is positive.
Step 2: Take the complement of each bit by changing zeroes to ones and ones to zero.
Step 3: Add 1 to your result. This is the two's complement representation of the negative integer.

EXAMPLE: *Find the two's complement of –17*

Step 1: $17_{10} = 0001\ 0001_2$

Step 2: Take the complement: 1110 1110

Step 3: Add 1: 1110 1110 + 1 = 1110 1111.

Thus the two's complement for -17 is $1110\ 1111_2$. It begins on the left with a 1, therefore we know it is negative.

Now you try some:

Find the two's complement for

a. –11

b. –43

c. –123

To translate a number in binary back to base ten, the steps are reversed:

Step 1: Subtract 1: ∴ 1110 1111 – 1 = 1110 1110

Step 2: Take the complement of the complement: 0001 0001

Step 3: Change from base 2 back to base 10 ∴16 + 1 = 17

Step 4: Rewrite this as a negative integer: –17

This suggests a new way to subtract in binary due to the fact that subtraction is defined in the following manner:

X – Y = X + (-Y)

EXAMPLE 1: *Subtract 17 from 23, as a computer would, using binary code.*

Given a register of size 6, 23 – 17 = 23 + (-17) becomes

0001 0111 + 1110 1111 = 10000 0110. (Verify both the binary form of 23 and the addition.) Since this result has 9 bits, which is too large for the register chosen, the leftmost bit is truncated, resulting in the binary representation of the *positive* (it starts with a 0) integer 00000110. When this is changed to a decimal number, note that 4 + 2 = 6 which is the answer expected.

Note that a register of size eight can only represent decimal integers between $-2^{(8-1)}$ and $+2^{(8-1)}$ and, in general, a register of size *n* will be able to represent decimal integers between $-2^{(n-1)}$ and $+2^{(n-1)}$

EXAMPLE 2: *Subtract 29 from 23, as a computer would, using binary code.*

Again we use a register of size 8, so that 23 – 29 = 23 + (-29) becomes

0001 0111 + 1110 0011 = 1111 1010. (Verify both the binary form of −29 and the addition.) Note that no truncation of the leftmost bit is necessary here. The result is the *negative* (it starts with a 1) integer 1111 1010. This needs to be "translated" to change it back to a decimal (see the steps on how to do this in the box above). Hence, going backwards, 1111 1010 – 1 = 1111 1001. The complement of which is 0000 0110 which is 6 in decimal. Negating this we get -6 as expected.

Now you try some:

Subtract each, as a computer out, using binary code using registers of size 8.

a) 26 – 15

b) −31 – 6

c) 144 – 156

d) Make up your own exercises as needed.

ANSWERS

$-11 = 1111\ 0101_2$

$-43 = 1101\ 0101_2$

$-123 = 1000\ 0101_2$

26 – 15 = 26 + (-15) = 0001 1010 + 1111 0001 = 10000 1011, and truncating the leftmost 1 to remain within a register of 8, the answer is $0000\ 1011_2$

$-31 – 6 = (-31) + (-6) = 1110\ 0001 + 1111\ 1010 = 11101\ 1011$, and truncating the leftmost 1 to remain within a register of 8, the answer is $1101\ 1011_2$

144 – 156 = 144 + (–156) = 1001 0000 + 0110 0100 = 1111 0100, which remains within the register of 8 bits (so nothing gets truncated), thus the answer is $1111\ 0100_2$.