MICROCONTROLLER SYSTEM : 68HC11

Taek Mu Kwon, Ph.D
Department of Electrical and Computer Engineering
University of Minnesota, Duluth
271 MWAH, 10 University Dr.
Duluth, MN 55812

1. Introduction

1.1 Background

The early days of computer evolution took place in the direction of designing and building high performance mainframes. The brain of these mainframes was called a Central Processing Unit (CPU) and designed using many high performance discrete components in the form of single or multiple circuit boards. However, with the advances in IC technologies in the early 1970's, engineers began building single-chip CPUs called microprocessors which house a significantly reduced capability of main frame level CPUs at that time. The classes of early microprocessors include Motorola 6800 family, Intel 8085 family, and Zilog Z80, which were evolved to MC680x0, Intel xxx86, and Z8000 family. The creation of microprocessors embarked a new era of personal computers that drastically influenced computer industry and the rest of the world. More and more powerful microcomputers and personal computers were rapidly developed. Today many microprocessors have not only 32-bit data-bus but also have on-chip floating point processors, cache and virtual memory, which have traditionally been the unique properties of mini and mainframe level computers. Some of recent Reduced Instruction Set Computers (RISCs) such as DEC's Alpha-chip has now a 64-bit data bus with the basic clock rate exceeding 500MHz and approaching towards GHz. Such computational power is actually superior to the old multi million dollar supercomputers such as Cray-1.

While microprocessors were rapidly evolving, engineers observed that many computer applications such as instrumentation and control applications do not really require powerful number crunching microprocessors, rather they require small cost effective processors with many convenient I/O (Input Output) functions. For example, the controller for a microwave oven may only require a simple microprocessor with multiple I/O (Input Output) control functions that can turn on and off different switches with simple timed combination. There are perhaps thousands of such examples: TV and VCR controllers, telephone answering machines, automobiles, elevator controllers, gas pump controller, multi-meter, laundry machine controllers, etc. This type of applications typically requires a simple processor, small amount of memory, serial and parallel I/O ports, and timers. Most importantly, the cost of processors must be very low (typically less than few dollars per chip in a large quantity) in order to cut down the final cost of the application products. Hence, another breed of computers called **micro controllers** were born to provide an easy and efficient design of many I/O control applications. With the existing IC technology, integrating a microprocessor with memory (RAM and ROM) and I/O ports into a single chip is not too difficult. Today many different types of micro controllers are available on the market for different capability requirements of application.

1.2 Microcontroller

A microcontroller is a single chip microcomputer that is specifically designed for dedicated embedded applications, and generally includes the following elements in a single chip:

- Microprocessor,
- Parallel I/O ports,
- Serial ports,
- RAM,
- ROM,
- Timers,
- A/D converter,
- Prioritized interrupt control.

The advantages of putting all of the above components into a single chip are numerous. Since most of I/O functions are already available on the microcontroller chip, the design of an application requires a small number of additional chips significantly saving the chip area, power consumption, and cost. Due to minimal chip counts, application designs are relatively simple and the chances of design mistakes are greatly reduced. Moreover, the on-chip memory and I/O ports provide a very high reliability because there exist almost no chances of loose connections or bad contacts. Finally, one of the most prominent characteristics is the low power consumption in which microcontrollers can be used for battery powered applications.

Today, most semiconductor companies produce their own line of microcontrollers, ranging variety of capabilities of chips from 4-bit to 32-bit processors combined with different types of I/O control functions. Thus, a designer must carefully choose a proper microcontroller to optimize the cost, performance, and other design requirements, which adds complicated decision making process. In this course, we will be focusing on studying Motorola 68HC11 family microcontrollers which are most widely used in the industry. Since this microcontroller includes typical capabilities of today's microcontrollers and simple microprocessor systems, it serves as a tool for learning fundamentals of today's microcomputers.

2. Number Systems and Arithmetic

Many different number systems perhaps from the prehistoric era have been developed and evolved. Among them, binary number system is one of the simplest and effective number systems, and has been extensively used in digital computers. Studying number systems can help understanding the basic computing processes and structure of computers.

This section describes positional number systems. The topics include conversion of number systems, negative number representations, arithmetic overflow mechanism, and basic computer arithmetic.

2.1 Positional Number Systems

A good example of positional number system is the decimal which we use in our daily lives. Another example is a binary system which is used as the basic number system for all computers. In positional number systems, a number is represented by a string of digits where the position of each digit is associated with a weight. In general, a positional number is expressed as:

$$d_{m-1}d_{m-2}\cdots d_1 d_0 . d_{-1}d_{-2}\cdots d_{-n}$$

where d_{m-1} is referred to as the *most significant digit* (MSD) and d_{-n} as the *least significant digit* (LSD). Each digit position has an associated weight b^i where *b* is called *the base* or *radix*. The point in the middle is referred to as a *radix point* and is used to separate the fractional part of a number (which is in the right side of the radix point) from the integer part (which is in the left side of the radix point). Fraction is a portion of magnitude of a number which is less than unit (e.g. *fraction* < 1) and thus it is called a *fraction*. Let *D* denote the value (or magnitude) of a positional number, then *D* can be calculated using:

$$D = \sum_{i=-n}^{m-1} d_i \cdot b^i \qquad (1)$$

Example 2.1.1: Calculate the value of 245.37_8

$$\begin{aligned} D &= 2\cdot 8^2 + 4\cdot 8^1 + 5\cdot 8^0 + 3\cdot 8^{-1} + 7\cdot 8^{-2} \\ &= 165.484375_{10} \end{aligned}$$

A binary (base=2) number system is a special case of the positional number system and used in almost all digital systems and computers. In this number system, the allowable digits are only 0 and 1 which are called "bits". Therefore, the leftmost digit of a binary number is called the *most significant bit* (MSB) and the rightmost is called the *least significant bit* (LSB). Because the base of binary numbers is two, bit b_i is associated with weight 2^i.

Example 2.1.1: Magnitude of Binary number

$$11010010_2 = 1\cdot 2^7 + 1\cdot 2^6 + 1\cdot 2^5 + 1\cdot 2^4 + 1\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0$$

$$1101.0011_2 = 1\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 + 0\cdot 2^{-1} + 0\cdot 2^{-2} + 1\cdot 2^{-3} + 1\cdot 2^{-4}$$

If base of a number system is larger than ten, the digits exceeding 9 are expressed using alphabets as a convention. For example, hexadecimal uses 1-9 and A-F; base 32 number uses 1-9 and A-V. This example is shown in Table 1. One may then wonder how a large-base number system such as a base-64 is expressed. Fortunately, we rarely use such a high-base number system because we find no real advantagea of using them in applications. Moreover, we can always convert them from any high-base number system to any other base number systems, which is the subject of the next section.

2.2 Conversion between 2^k bases

The number systems with 2^k bases have an interesting property in that the conversion between them can be achieved without the computation of Eq. (1). Such number systems include binary, octal, hex, and base-32 number systems. Note that since these number systems possess base 2^k, all numbers within these systems can be uniquely represented by k binary bits. For example, octal numbers can be represented by three bits; hex numbers can be represented by four bits, etc. This relation allows us to easily convert these number systems by simply grouping their binary representation with k bits. Two examples are given in Example 2.2.1.

Since the binary representation of 2^k base numbers can be directly associated by simple grouping of k digit strings, the conversion from octal to hex or vice versa can be easily achieved through binary conversion. Example 2.2.2 illustrates this conversion steps.

Table 1. Decimal, binary, hexadecimal, and base-32 Number Systems

Decimal	Binary	Octal	Hexadecimal	Base-32
0	00000	0	0	0
1	00001	1	1	1
2	00010	2	2	2
3	00011	3	3	3
4	00100	4	4	4
5	00101	5	5	5
6	00110	6	6	6
7	00111	7	7	7
8	01000	10	8	8
9	01001	11	9	9
10	01010	12	A	A
11	01011	13	B	B
12	01100	14	C	C
13	01101	15	D	D
14	01110	16	E	E
15	01111	17	F	F
16	10000	20	10	G
17	10001	21	11	H
18	10010	22	12	I
19	10011	23	13	J
20	10100	24	14	K
21	10101	25	15	L
22	10110	26	16	M
23	10111	27	17	N
24	11000	30	18	O
25	11001	31	19	P
26	11010	32	1A	Q
27	11011	33	1B	R
28	11100	34	1C	S
29	11101	35	1D	T
30	11110	36	1E	U
31	11111	37	1F	V

Example 2.2.1: Binary to hexadecimal or octal conversion

$$11010110_2 = \overbrace{011}\overbrace{010}\overbrace{110}_2 = 326_8$$

$$= \overbrace{1101}\overbrace{0110}_2 = D6_{16}$$

$$11010010.10110_2 = \overbrace{11}\overbrace{010}\overbrace{010}.\overbrace{101}\overbrace{100}_2 = 322.5_8$$

$$= \overbrace{1101}\overbrace{0010}.\overbrace{1011}_2 = D2.B_{16}$$

Example 2.2.2: octal to hexadecimal or vice versa

$$273_8 = \overbrace{010}\overbrace{111}\overbrace{011}_2$$

$$= \overbrace{1011}\overbrace{1011}_2$$

$$= BB_{16}$$

We have seen that the conversion between numbers with power of radix 2 can be readily achieved through binary expression and regrouping of bits. This convenience led to utilization of hexadecimal (or octal) numbers in representing binary numbers for many computer architecture related issues. For example, the instruction LDAA (Load Accumulator A) of 68HC11 is encoded as the binary number 10000110_2, but for convenience of writing and reading it is usually expressed in hexadecimal 86_{16}, from which we save time and spaces. Very often, hexadecimal, octal and binary numbers are interchangeably used in the computer architecture or microprocessor related fields.

2.3 General Position Number System Conversion

This section discusses conversion of numbers from any base to any other base. Due to our familiarity and representation of decimal, a convenient way of base-conversion is performed *through the use of decimal*. That is, for the conversion from base-k to base-p, we first convert a base-k number to a decimal, then convert the decimal to a base-p number.

Using Eq. (1) we can easily convert from any base to decimal by simply expressing the digits and weights using decimal as shown in Example 2.3.1. Therefore, this issue will not be discussed any further.

Example 2.3.1: Base-k to decimal conversion

$$1bE8_{16} = 1\cdot 16^3 + 11\cdot 16^2 + 14\cdot 16^1 + 8\cdot 16^0 = 7144_{10}$$
$$437.5_8 = 4\cdot 8^2 + 3\cdot 8^1 + 7\cdot 8^0 + 5\cdot 8^{-1} = 287.625_{10}$$

Next, we consider conversion from a decimal to a base-p number system. This process usually requires more computation. Let a general position number be denoted as an addition of integer and fractional part:

$$D = I + F. \qquad (2)$$

Then we may express the integer part as

$$I = ((\cdots((d_{p-1})\cdot b + d_{p-2})\cdot b + \cdots)\cdot b + d_1)\cdot b + d_0 \qquad (3)$$

Although this formula looks complicated in a first glance, its structure is exactly the same as the integer part of Eq. (2) except that the weights b^k is now expressed as $b^1 b^2 \cdots b^k$. Example 2.3.2 illustrates conversion of a decimal number 5432_{10} into the form given by Eq. (3).

Example 2.3.2: Integer expressions of positional numbers

$$54_{10} = 5\cdot 10 + 4$$
$$543_{10} = (5\cdot 10 + 4)\cdot 10 + 3$$
$$5432_{10} = ((5\cdot 10 + 4)10 + 3)\cdot 10 + 2$$

In this example, notice that if the last expression is divided by 10, the remainder is the least significant digit 2 and the quotient is $((5\cdot 10+4)\cdot 10+3)$. The next significant digit can be obtained by dividing 535_{10} again. Due to this relation, the conversion to an arbitrary base number can be obtained by repeated division of quotient and collection of remainders. A simple hand-calculation method can be devised using the above relation. Let's express the integer division by

Divisor)Dividend
Quotient.....Remainder

Using this expression, Example 2.3.3 shows conversion from a decimal to a binary.

Example 2.3.3: Convert 179_{10} to a binary.

2	)179	
2	)89	.....1 LSB
2	)44	1
2	)22	0
2	)11	0
2	)5	1
2	)2	1
	1	0
	MSB	

The final conversion result reads

$$179_{10} = 10110011_2.$$

It should be noted that the above method can be extended to conversion from any base to any base. For example, consider that we wish to convert a hexadecimal number to a base-5 number. Then, the base-5 number can be directly converted by repeated division by 5. However, this direct division means, you must divide the base-16 number by 5, which is not a simple calculation. Thus, it is essentially wise to first convert the hexadecimal to a decimal, and then convert it to base-5.

Similarly to the expression of integer part in Eq. (3), fractional part can be written in the following form:

$$F = (d_{-1} + (d_{-2} + (\cdots b^{-1}d_{-n}\cdots)b^{-1})b^{-1})b^{-1} \qquad (4)$$

Note that multiplying b to F in Eq. (4) produces d_{-1} as an part of the product. This representation of number system is illustrated using Example 2.3.3.

Example 2.3.3: Fraction expression of positional numbers

$$0.12_{10} = (1+2\cdot 10^{-1})10^{-1}$$

$$0.123_{10} = (1+(2+3\cdot 10^{-1})10^{-1})10^{-1}$$

$$0.1234_{10} = (1+(2+(3+4\cdot 10^{-1})10^{-1})10^{-1})10^{-1}$$

$$0.F1_{16} = (15+1\cdot 16^{-1})16^{-1}$$

$$0.F1A_{16} = (15+1+10\cdot 16^{-1})16^{-1})16^{-1}$$

$$0.F1AC_{16} = (15+1+(10+12\cdot 16^{-1})16^{-1})16^{-1})16^{-1}$$

Due to the structure described above, a fractional number expressed by decimal can be converted into a base "b" number by positioning the integer part from left to right after each multiplication by b, i.e., see Example 2.3.4.

Example 2.3.4: Convert a decimal number 0.625 to binary.

0.625	0.250	0.5
× 2	× 2	× 2
1.250	0.5	1.0
$d_{-1} = 1$	$d_{-2} = 0$	$d_{-3} = 1$

Thus,

$$0.625_{10} = 0.101_2$$

One should be careful to note that a closed form fraction in one number system does not always lead to a closed form in other number system. This case is illustrated in Example 2.3.4.

Example 2.3.4: Decimal to base-x conversion: Convert 0.7_{10} to a binary.

0.7	0.4	0.8	0.6	0.2	0.4
x 2	x 2	x 2	x 2	x 2	x 2
1.4	0.8	1.6	1.2	0.4	0.8

$d_{-1} = 1$ $d_{-2} = 0$ $d_{-3} = 1$ $d_{-4} = 1$ $d_{-5} = 0$ $d_{-6} = 0$

Thus,

$$0.7_{10} = 0.1011001100110\cdots_2$$

This example implies that the base conversion of fractions can introduce some errors by the conversion process itself. A fraction which has a repeating sequence is referred to as a repeating fraction.

2.4 Negative Numbers

2.4.1 Signed Magnitude Number System

Negative numbers can be represented in many ways. In our daily transactions, a **signed magnitude system** is used, where a number consists of a magnitude and a symbol indicating whether the magnitude is positive or negative. For example,

$$-57_{10}, +98_{10}, +1000.1267_{10}, -345.345_{10}.$$

In the above example, the symbols "+" and "-" were used to represent the sign of a number. An alternative is instead of introducing new symbols adding an extra digit to represent positive and negative. This technique is frequently used in binary number system, e.g., bit "1" is appended at MSB to represent *negative* and bit ``0'' for *positive*. Example 2.4.1 illustrates this relation by 8-bit numbers with 7-bit magnitude and one sign-bit.

Example 2.4.1: Signed magnitude binary numbers

$$00101101 = +2D_{16}$$
$$10101100 = -2D_{16}$$
$$01111111 = +7F_{16}$$
$$11111111 = -7F_{16}$$

2.4.2 Complement Number System

In this number system, a negative number is determined by taking its complement as defined by the system. Radix complement and diminished-radix complement are the two basic methods in this system.

i) Radix complement: The complement of an n-digit number is obtained by subtracting it from b^n. See Example 2.4.2

Example 2.4.2: Radix complements

$$\text{10's complement: } 1849_{10} \Rightarrow 10000_{10} - 1849_{10} = 8151_{10}$$
$$\text{8's complement: } 1547_{8} \Rightarrow 10000_{8} - 1547_{8} = 6231_{8}$$
$$\text{4's complement: } 1320_{4} \Rightarrow 10000_{4} - 1320_{4} = 2020_{4}$$
$$\text{2's complement: } 1010_{2} \Rightarrow 10000_{2} - 1010_{2} = 0110_{2}$$

As in the above example, direct subtraction from b^n is inconvenient or cumbersome to calculate. A simpler and easy way can be derived by modifying the subtraction as:

$$b^n - D = (b^n - 1) - D + 1$$

Notice that $b^n - 1$ has the form that all digits are consist of the highest digits in the number system. For example, in decimal $10^4_{10} - 1 = 9999_{10}$, in octal $10000_8 - 1 = 7777_8$, in binary $10000_2 - 1 = 1111_2$, etc.

ii) Diminished-Radix complement:

The complement of an n-digit number *D* is obtained by substituting it from $b^n - 1$. This can be accomplished by complementing the individual digits of *D* without adding 1.

Example 2.4.3: 9's complement
In decimal, the diminished-radix complement is called the 9's complement because the complement is obtained by independently subtracting each digit from 9.

$$\text{Complement of } 1849_{10} \Rightarrow 9999_{10} - 1849_{10} = 8150_{10} = -1849_{10}$$
$$\text{Complement of } 7932_{10} \Rightarrow 9999_{10} - 7932_{10} = 2067_{10} = -7932_{10}$$
$$\text{Complement of } 0007_{10} \Rightarrow 9999_{10} - 0007_{10} = 9992_{10} = -0007_{10}$$

Example 2.4.4: 1's complement
Similarly to the decimal case, the diminished-radix complement of a binary number is called 1's complement because the complement is obtained by subtracting each digit from 1.

$$\text{Complement of } 1011_2 \Rightarrow 1111_2 - 1011_2 = 0100_2 = -1011_2$$
$$\text{Complement of } 0101_2 \Rightarrow 1111_2 - 0101_2 = 1010_2 = -0101_2$$
$$\text{Complement of } 0000_2 \Rightarrow 1111_2 - 0000_2 = 1111_2 = -0000_2$$

Note from Example 2.4.4 that 1's complement is simply obtained by inverting each digit, i.e. $1 \rightarrow 0$ and $0 \rightarrow 1$. Thus, the main advantages of 1's-complement system are its simplicity of conversion and symmetry of complements. However, this symmetry causes the existence of two zeros, i.e., a positive zero $00\cdots00$ and a negative zero $11\cdots11$. Hence implementing addition of 1's complement numbers to digital computer system leads to significant inefficiency because the system must check for both representations of zeros or it must convert one to another zero. This is the main reason why 2's complement number system is used for all today's digital computers, which has a unique zero ($00\cdots00$).

Table 2.2 4-bit Numbers in Different Signed Systems

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

2.5 Signed Addition/Subtraction

In signed computation, subtraction is achieved by adding the negated (i.e. complemented) subtrahend to the minuend. In hardware implementation this means that computers need only adders but not subtractors. Of course the flexibility of using negative numbers provides convenience in a variety of computational applications, in addition to the savings in hardware. Another important aspect of addition/subtraction in computer systems is the overflow errors, which are caused by the limited bit-width of the data path in a computer. A detailed treatment of overflow conditions is discussed in this section.

2.5.1 Signed Overflow

If an addition or a subtraction produces a result that exceeds the range of the number system (the data

width allocated to the result), overflow is said to occur. Overflow is essentially an error condition that requires a special treatment in order to make the current result valid. A simple rule exists for detection of overflow. *Addition of two numbers with different signs can never produce overflow, but addition of two numbers of like sign can*. This simple rule can be used for screening the candidates of overflow condition. As the next step one of the following two rules can be applied, if the two addends have the same sign.

1. An addition (same if subtraction is done by adding the complemented number) overflows if the signs of the addends are the same and the sign of the sum is different from the addends' sign.
2. An addition overflows if the carry bits into and out of the sign position are different.

The overflow detection rule is often built into a piece of hardware called an arithmetic logic unit (ALU) inside the computer. The status register of ALU almost always includes a bit called the overflow-bit which indicates detection of an overflow condition whenever it is set. The following example illustrates overflowed computation for 4-bit arithmetic. Keep in mind that the range of 4-bit number can represent is from -8 to +7. Exceeding this range causes the overflow.

Example 2.5.1: Overflow examples in 4-bit computation

(-3_{10})	1101_2	$(+5_{10})$	0101_2
$+(-6_{10})$	$+1010_2$	$+(6_{10})$	$+0110_2$
-9_{10}	10111_2	$+11_{10}$	1010_2

(-8_{10})	1101_2	$(+7_{10})$	0111_2
$+(-8_{10})$	$+1010_2$	$+(+7_{10})$	$+0111_2$
-16_{10}	10000_2	$+14_{10}$	1110_2

2.5.2 Signed subtraction

Signed subtraction in most computers is done by taking 2's complement of the subtrahend and then adding it to the minuend following the normal rules of addition. Overflow condition must be checked after the addition in order to obtain the correct computational result. If no overflow condition is detected, the correct answer of the subtraction is obtained from the result by simply discarding the carry-out bit of

the MSB if a carry-out bit exists. If an overflow condition is detected, there are two ways of dealing with this error. The first approach is simply reporting an error message that indicates the overflow condition. Most computers use this approach and leave the responsibility of handling the error to the user. The second approach is modifying the result to a correct one by allocating more bits to the addends. Whenever an overflow occurs, only one more bit extension to operands is needed to express the overflowed number. However, due to the fixed data width of computers, the data width is usually extended twice of the data width, i.e., if a single precision computation is overflowed, a double precision (twice the data width) is used to correct the error.

Example 2.5.2: Signed subtraction with no overflow
Compute 0100-0011=?

i) Compute the 2's complement of 0011 : 1100 + 1 = 1101
ii) Add the complemented number to the minuend:

```
  0100
+1101
10001
```

⇒ No overflow. Discard the MSB carry-out bit
Correct Answer: 0001

Example 2.5.3: Signed subtraction with overflow
Compute 0110 - 1101 = ?

i) Compute the 2's complement of 1101 : 0010 + 1 = 0011
ii) Add the complemented number 0011 to the minuend:

```
  0110
+0011
  1001    | Overflow. Report an overflow error message.
```

If a correct answer is wished to be obtained instead of just giving an overflow error message, one can redo the operation by allocating extended bits to operands. In this example, we shall extend the computation to a double-precision (8-bit in this case) arithmetic. That is, compute 00000110 - 11111101 = ?. Notice that the positive number is extended by appending 0's, while the negative number is extended by appending 1's to the MSB of the number. This is because we want to preserve the sign and magnitude of the original number when bits are extended.

i) Compute the 2's complement of 11111101 : 00000010 + 1 = 00000011
ii) Add the complemented number to the minuend:

```
  00000110
+ 00000011
  00001001   | No overflow.
```

Correct answer: 00001001

In Example 2.5.3, the correct result was obtained by extending operands and recalculating them after detecting an overflow condition. In reality, this recalculation is not necessary. The result of operation can be corrected by recognizing the signs of two addends (i.e., step ii)). In Example 2.5.3, since the two addends are both positive, the correct answer is obtained by appending zeros to the MSB side until all the extended bits are filled. If both addends are negative, the correct answer is obtained by appending ones to the MSB side until all the extended bits are filled. An example for this case is illustrated in Example 2.5.4.

Example 2.5.4: Signed subtraction with overflow correction

Compute 1101 - 0111 = ?

i) Compute the 2's complement of 0111 : 1000 + 1 = 1001
ii) Add the complemented number to the minuend:

```
  1101
+ 1001
 10110 |   Overflow Error.
```

Since the final two addends are negative, the correct result is obtained by appending four ones to the MSB side of the 4-bit result 0110.
Correct answer: 11110110. □

2.6 Unsigned Addition/Subtraction

In an unsigned number system, all numbers are considered positive. For instance, four bits in binary represent positive numbers from 0_{10} to 15_{10}. This approach uses the single bit assigned for sign representation as a part of the magnitude, and thus twice the magnitude of the signed representation is achieved.

2.6.1 Unsigned Addition

Since all numbers are positive in unsigned numbers, the two addends are always positive. Hence, an

unsigned overflow condition occurs only if the computation produces a carry-out at the MSB of the allocated bit. The computation must be carried out using normal addition rules, but if an unsigned-overflow condition is detected, the correct answer is obtained by simply appending zeros to the MSB side of extended bits.

Example 6.1: Unsigned addition
Compute 1100 + 1001

```
  1101
+1001
 10110 |   Carry-out exists. An unsigned-overflow has occurred.
```

Correct answer: 00010110

Example 6.2: Unsigned addition
Compute 0110 + 0101

```
  0110
+0101
  1011 |    No carry-out exists. The result is correct.
           Note: It causes an overflow error if it was signed addition.
```

Correct answer: $1011_2 = 11_{10}$

6.2 Unsigned Subtraction

In unsigned subtraction, the minuend must be larger than the subtrahend. Otherwise, the result would become negative, which violates the definition of unsigned computation. If the subtracted result is actually negative, an occurrence of error should be indicated. In a computer implementation, this error condition is shown through a borrow bit. If the borrow bit is set, it means that the minuend is smaller than the subtrahend and needs a borrow to correct the error.

Example 6.3: Unsigned subtraction Compute 1001 - 1100

```
  1001
-1101
 ????
```

Answer | Minuend is bigger than subtrahend. A borrow error (or unsigned-underflow error) has occurred.

Chapter References

[1] J. F. Wakerly, *Microcomputer Architecture and Programming*, John Wiley \& Sons, Inc., 1989.
[2] D. Knuth, *Seminumerical Algorithms*, Addison-Wesley, 1969,
[3] A. Kaufmann and M. Gupta, *Introduction to Fuzzy Arithmetic*: *Theory and Applications*}, Van Nostrand Reinhold Company, New York, 1984.