Module 9 – Microprocessor operation

Overview

In module 6 we performed basic complements arithmetic on signed binary numbers. However, we now know that a typical microprocessor has registers in which all binary numbers are processed and interpreted, so the size of those registers becomes important when manipulating binary numbers.

Microcomputer operations (concept map)

Objectives

At the completion of this module you will be able to:

- describe how binary numbers are processed and interpreted by a CPU in both sign and magnitude
- describe and explain the concepts involved with overflow and carry conditions
- explain the status register and its use by a CPU to interpret special conditions occurring in the accumulator of a CPU.

9.1 Interpretation of binary number sign

A typical microprocessor register may hold 8 bits of data so we refer to an '8 bit binary word'. The maximum number that can be represented in 8 bits is 255_{10}, giving a modulus of 256.

An 8 bit register is usually represented as:

Weighting Value	→	128	64	32	16	8	4	2	1
Register	→								
Bit Position Number	→	7	6	5	4	3	2	1	0
		MSB							LSB

The modular number system was introduced in module 6, and the importance of this number system will become clear when performing arithmetic operations using positive and negative 8-bit binary numbers.

In decimal arithmetic it is easy for us to indicate if a number is negative or positive by the appendix + or – to a number of any magnitude. A microprocessor cannot recognise such signs, nor can it handle numbers of any magnitude so a method to identify and handle negative numbers in a modular number system is required.

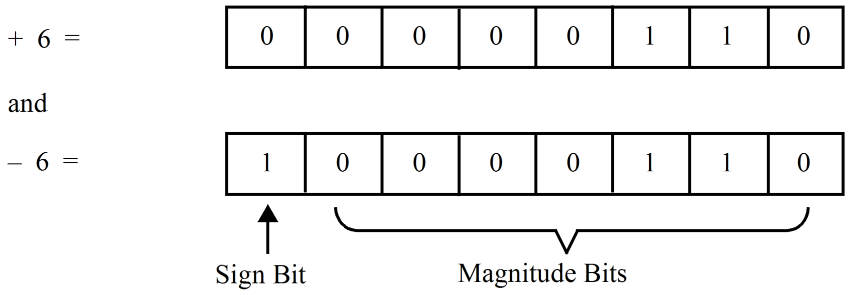
There are three methods which could be used depending upon the application.

9.1.1 Method 1, sign-magnitude system

This method uses the most significant bit (MSB) of the binary number to indicate its sign. For example,

	if the MSB = 0,	the number is positive
and	if the MSB = 1,	the number is negative.

The remaining bits in the number indicate its magnitude. Consider registers containing the decimal number 6.



This method uses a fair amount of hardware to implement, is difficult to do arithmetic with and it has a serious problem as shown. In the decimal system we know that:

$$-6 + 6 = 0$$

In this system we have:

- 6	=	10000110
+ 6	=	00000110
Sum	=	$\overline{\underline{10001100}}$

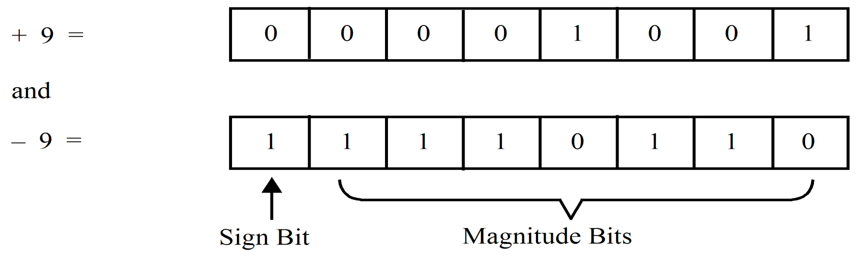
which is –12 in decimal!

Clearly a problem exists with this ‘simple system’. Notice the magnitude bits were not complemented for the negative number.

9.1.2 Method 2, ones complement representation

This method identifies the negative of a number as its ones complement. Positive numbers in this system are the same as in the sign-magnitude system. As seen in module 6 when doing arithmetic with this system, sometimes the sign bits (defined in 9.1.1 above) when added cause a carry, which is added as if it were a magnitude bit. This was called end-around-carry.

In this system we note that:



Note: the 7 magnitude bits do not give the correct magnitude directly – they need re complementing.

Also in this case –9 + 9 equal:

1	1	1	1	1	1	1	1

which is minus zero! – not a useful number.

9.1.3 Method 3, twos complement representation

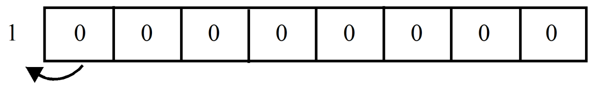
Positive numbers in this system are identical with the other two systems however negative numbers are stored as twos complement numbers as done in module 6. The following table shows that the MSB indicates the sign and the remaining 7 bits represent the magnitude of the number in twos complement form. In twos complement arithmetic carries are ignored as far as affecting the magnitude is concerned.

In this system we note that:

+ 12 =	0	0	0	0	1	1	0	0
and								
- 12 =	1	1	1	1	0	1	0	0

Note: the 7 magnitude bits again require **true** complementing to obtain the correct magnitude.

Also in this case –12 + 12 equal:



The carry is ignored and the correct answer of plus zero is obtained.

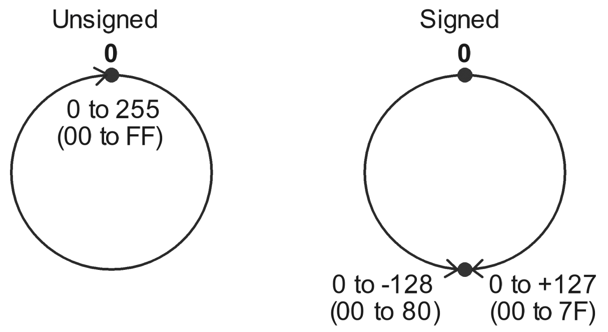
For an eight bit register you can now see that a number could be stored under various conditions. The programmer **must** be aware of the system he/she is using **and** that of the computer being used.

A table of numbers appropriate for an 8 bit register is now shown for the twos complement system which is widely used.

Bit Pattern	Unsigned Value	Signed Value (2's Complement)
00000000	0	0
00000001	+ 1	+ 1
00000010	+ 2	+ 2
↓	↓	↓
01111111	+ 127	+ 127
10000000	+ 128	– 128
10000001	+ 129	– 127
¦	¦	¦
11111111	+ 255	– 1

It can be seen that with the twos complement system there is only one zero and all numbers from 0 to +127 and –128 can be represented without ambiguity.

The following circular representation helps to illustrate a modulo 256 number system. The diagrams are based on an 8-bit binary word (00_{16} to FF_{16}) and can be used to represent both unsigned and signed numbers shown in the above table. Positive numbers are represented by moving clockwise around the circle, while negative numbers are represented by moving anti-clockwise. So unsigned numbers from 0 to 255 can be represented, as well as 2's complement signed numbers from 0 to +127 (00_{16} to $7F_{16}$) and 0 to -128 (00_{16} to 80_{16}).



To sum signed numbers place an arc segment around the circle to represent the first number beginning at 0, then place another arc segment representing the second number on the end of the first. For a valid answer, the result of the addition must be less than or equal to +127 for positive numbers or less than or equal to -128 for negative numbers. If these values are exceeded an overflow is said to have occurred.

Examples:	35 + 65 = 100	the result is less than +127 therefore it is valid
	35 + 105 = 140	the result exceeds +127 therefore is not valid.

Expressed in 2's complement form:

35 $\rightarrow$ 00100011

+65$\rightarrow$ 01000001

$\overline{01100100}$ $= +100_{10}$

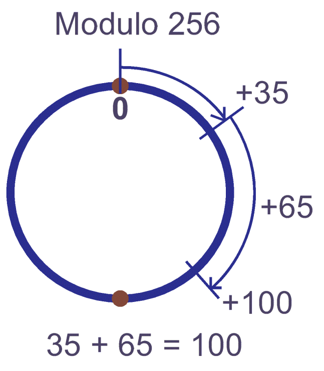
For the 8-bit signed 2's complement system the result is correct for both sign and magnitude (the MSB being the sign).

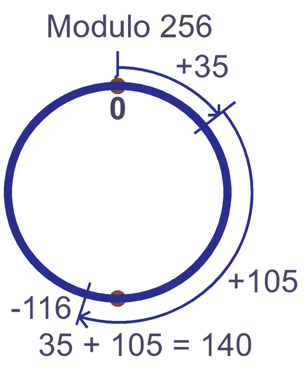
35 $\rightarrow$ 00100011

+105$\rightarrow$ 01101001

$\overline{10001100}$ $= -116_{10}$

For the 8-bit signed 2's complement system the result is incorrect. The result should have been +140, but this is not possible in this system as it exceeds +127.





35 + (-65) = -30 the result is less than -128 therefore it is valid

(-35) + (-105) = -140 the result exceeds -128 therefore is not valid.

Expressed in 2's complement form:

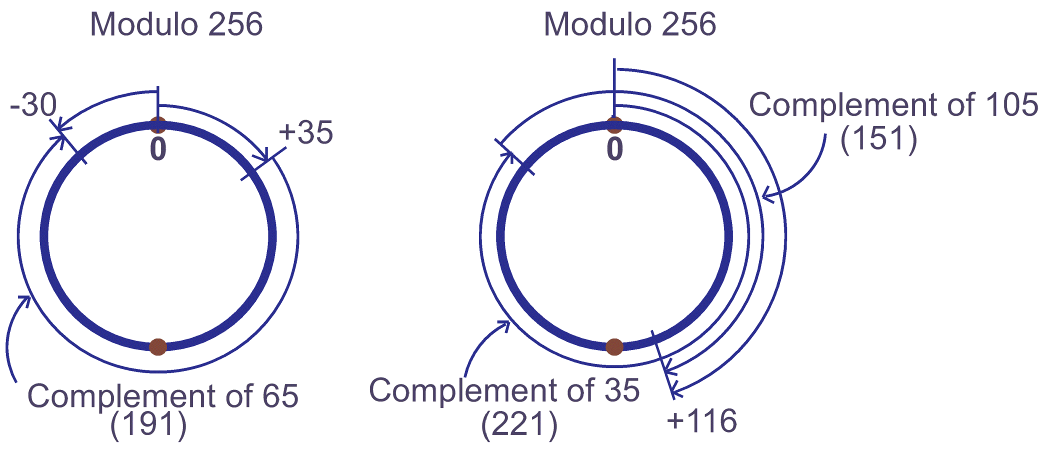
35 →	00100011	
+(-65) →	10111111	(2's complement of 65)
	11100010	= -30_{10}

For the 8-bit signed 2's complement system the result is correct for both sign and magnitude.

Note, being a negative result the magnitude is in 2's complement form.

-35 →	11011101	(2's complement of 35)
+(-105) →	10010111	(2's complement of 105)
	01110100	= $+116_{10}$

For the 8-bit signed 2's complement system the result is incorrect. The result should have been -140, but this is not possible in this system as it exceeds -128.



9.2 Overflow conditions

In the previous section we saw that in a 8-bit twos complement system, the largest positive number which can be stored in a register is 127 and the largest negative number is 128.

Consider, again, the case where 2 signed numbers of the correct size to fit into a **word-size** register are added and the result exceeds the allowable size limit for that register.

9.2.1 Overflow examples

Consider 2 positive numbers A and B:

$$\begin{array}{lllllll} \text{where} & A & = & 01100000 & = & 96_{10} \\ & B & = & 01011001 & = & 89_{10} \\ & A+B & = & \overline{\underline{10111001}} & & 185_{10} \end{array}$$

The answer A + B should be **positive** yet a one bit is held in the MSB indicating it is **negative**.

The result exceeds the allowable limit for positive numbers in a signed system (i.e. its > +127).

Consider now 2 negative numbers A and B:

$$\begin{array}{rcr} -A & = & 10000000 \\ -B & = & 10000000 \\ \hline -A+(-B) & = & 100000000 \\ \text{Carry} & \nearrow & \overline{} \end{array}$$

The carry is ignored however a zero in the most significant bit indicates a **positive** answer **because the answer is too large to fit** in a 8 bit register using a signed number system. That is the number is **more negative** than –128.

In both of these cases an **overflow** is said to have occurred.

Rule: *An overflow occurs if the sign bits of A and B are both positive and the result of an addition is negative or if the sign bits are both negative and the result of an addition is positive.*

In microprocessor hardware, the overflow is determined by XORing the carry into bit 7 with the carry out of bit 7. A logic 1 indicates that an overflow has occurred.

9.3 The status register

Most computers have a special register to note special events occurring in the accumulator. It is called the **status register** or **condition code register**. It is an 8 bit register which has its bits set independently according to particular result of a test done on the accumulator.

One such test is the **overflow** and you can identify this register in figure 8.1 of module 8. If after an instruction as been executed in the accumulator an overflow occurs, then a special bit in the status register called the ‘overflow bit’ or ‘flag’ will be set to a logic 1 accordingly.

The purpose of remembering the status of a previous operation is to allow **branching** to another part of a program to occur. For example if the overflow flag = 1, **branch** to another part of the program. If not, carry on normally. An instruction satisfying this criteria is **branch if overflow set (BVS).**

9.4 Carry conditions

Another bit in the status register is set to a logic 1 if there is a ‘carry’ from a previous operation in the accumulator.

9.4.1 Carry examples

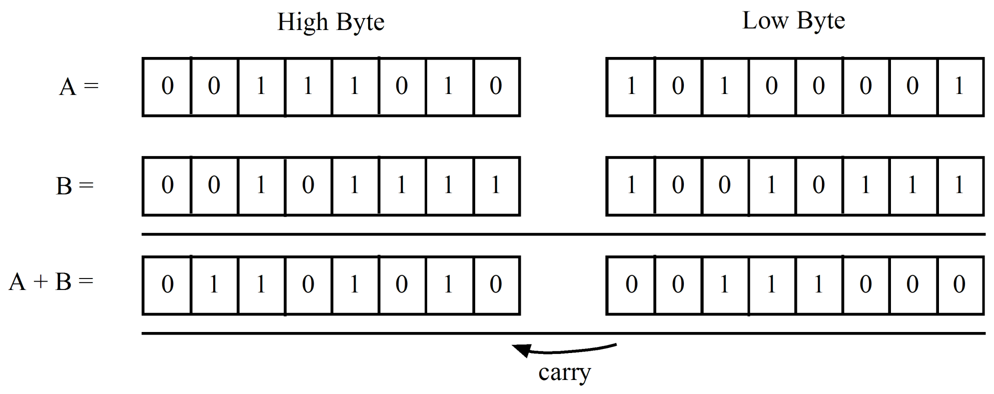
Consider the addition of two correct size negative numbers:

$$\begin{array}{lccccccccc}
 & & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\
+ & & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\text{Carry} \longrightarrow & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
\hline
\end{array}$$

Since the answer must be stored in an 8 bit register, the carry bit can be stored in the status register. It is like storing a ninth bit. A good example considers the addition of two multiple-word length numbers.

Let	A	=	3AA1
and	B	=	2F97

Note that both A and B are sixteen bit hexademical words and they will be held in 2 x 8 bit registers each. The most significant 8 bits is called the **high byte** and the least significant 8 bits is called the **low byte**.



Answer 6A38$_{16}$

To program this, the 2 **low** bytes are added first giving 38$_{16}$. A special instruction ‘add with carry’ is then used to add the two high bytes plus the carry giving 6A. The complete answer is then 6A38$_{16}$.

Note that this technique of storing 16 bit numbers across two registers is also done for numbers having a decimal point. A computer cannot recognise a decimal point and so it is up to the programmer to keep track of the split up across registers.

For the number 13.47_{BCD} we have

0	0	0	1	0	0	1	1

Register 1

Integer byte

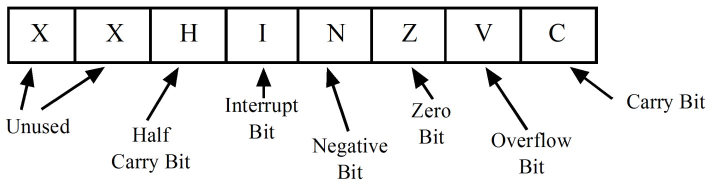
0	1	0	0	0	1	1	1

Register 2

Fractional byte

9.5 Other status register conditions

The status register has 8 bits as previously mentioned. So far we have mentioned overflow and carry. Other bits are set if the accumulator result is negative or if it is zero. Of the remaining four bits, two are unused and two will be explained later. The register pattern is:



Consider a CPU with an 8-bit word size. Any 8 bit number may be used **by the programmer** as:

- A twos complement number, in which case the MSB indicates the sign and the largest positive number is 127 and larger negative is 128.

or

- An 8 bit unsigned number up to a maximum size of 255.

Note: Although the programmer may regard numbers as signed or unsigned, the CPU follows 2 rigid rules always:

- When doing subtraction it always forms the twos complement and adds.

and

- It sets the status or condition code register bits according to a **signed number system**. That is the MSB of result determines the N bit of the status register etc.

9.5.1 Addition example

Consider the following example:

$$64 + 127 = 191$$

64	=		0	1	0	0	0	0	0	0
127	=		0	1	1	1	1	1	1	1
Addition	=	0	1	0	1	1	1	1	1	1

Carry = 0

Note: No carry occurred **but** we say the carry equals zero.

The answer is correct for **unsigned** numbers but the CPU will interpret the result according to a **signed number system** and set the appropriate flags. In this case, the condition code flags are:

C (carry)	=	0	–	**Not set** because its an addition with no carry.
V (overflow)	=	1	–	**Set** because the result has exceeded the signed number system as 191 >127. Also, an overflow occurs if the sign bits of A and B are both positive and the result of an addition is negative.
N (negative)	=	1	–	**Set** because a 1 is in MSB.
Z (zero)	=	0	–	**Not set** because the result is not zero.

Note: It's up to the programmer to decide if he is working with **signed** or **unsigned** numbers.

This is done by choosing the appropriate **branch** instruction which acts after checking the status bits. For example, branch if minus or plus or zero etc.

9.5.2 Subtraction example

Consider the following example:

$$2 - 3 = -1$$

3	=	0 0 0 0 0 0 1 1
2's comp is	=	1 1 1 1 1 1 0 1
2	=	0 0 0 0 0 0 1 0
Addition	=	0 1 1 1 1 1 1 1 1

Carry = 0

MSB = 1
∴ Negative number set status register N bit.

Magnitude of result is twos complement i.e. 0000001.

Answer = –1

Note: When a CPU performs a subtraction using twos complement arithmetic, if there is:

- a 0 carry from MSB ⇒ **set** the **carry** (= 1) (indicates a borrow)
- a 1 carry from MSB ⇒ **reset** the carry (= 0)

This is performed in circuit by **inverting** the carry bit whenever a subtraction is executed.

In this example: C = 1, V = 0, N = 1, Z = 0

C = 1,	V = 0,	N = 1,	Z = 0
because there was a carry of zero	because the answer fits in 7 bits	because the answer is negative MSB = 1	because the answer is not zero

Activity 9.1

Perform the following binary arithmetic examples, using the 8-bit signed 2's complement system, and determine the status register word in each case (consider ONLY the status register low byte):

1. 49 – 52
2. 52 – 49
3. 237 – 116
4. 237 – 85
5. 85 – 237

Solution to activity 9.1

1. N=1, Z=0, V=0, C=1, or 9_{10}
2. 0_{10}
3. 2_{10}
4. 8_{10}
5. 1_{10}