國立成功大學
National Cheng Kung University

Chapter 4
Programming Languages

Chien-Chung Ho (何建忠)

Computer Programming Languages

- Three types of computer programming languages: machine language, assembly language, and high-level language.
- A machine language program consists of either binary or hexadecimal op-codes. Programming a computer with either one is relatively difficult, because one must deal only with numbers. The CPU architecture of the computer determines all its instructions. These instructions are called the computer's *instruction set*.

Computer Programming Languages

- Programs in assembly and high-level languages are represented by instructions that use English-language-type statements. The programmer finds it relatively more convenient to write programs in assembly or high-level language than in machine language.

Computer Programming Languages

- An assembler translates a program written in assembly language into a machine language program. A compiler or interpreter converts a high-level language program into a machine language program. Assembly or high-level language programs are called source codes. Machine language programs are known as object codes.

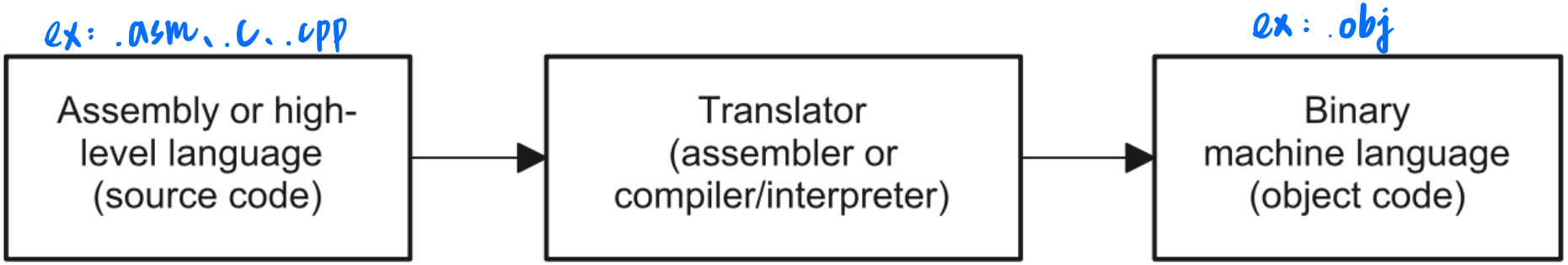
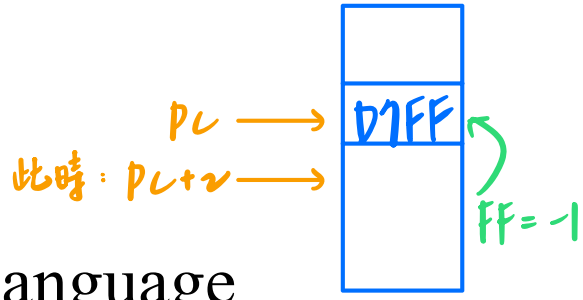


FIGURE 4.1 Translating assembly or high-level language into binary machine language.

Machine Language



- A microcontroller has a unique set of machine language instructions defined by its manufacturer. The Microchip Technology's PIC18F microcontroller uses the code $D7FF_{16}$ for the assembly language statement "HERE BRA HERE" (branch always to HERE), whereas the Motorola/Freescale HC11 microcontroller uses the code $20FE_{16}$ for the same statement with its BRA instruction.

Machine Language

At the most elementary level, a microcontroller program can be written using its instruction set in binary machine language. As an example, the following program adds two numbers using the PIC18F machine language:

```
0000111000000010
0000111100000011
0110111001000000
1110111100000011
```

To increase the programmer's efficiency in writing a machine language program, hexadecimal numbers rather than binary numbers are used. The following is the same addition program in hexadecimal using the PIC18F instruction set:

```
0E02
0F03
6E40
EF03
```

Assembly Language

- Each line in an assembly language program includes four fields:
  - Label field
  - Instruction, mnemonic, or op-code field
  - Operand field
  - Comment field

op code　　註解

Label	Mnemonic	Operand	Comment
	MOVLW	1	; Move 1 into accumulator
	ADDLW	2	; Add 2 with 1, store result in accumulator
	SLEEP		; Halt

Assembly Language

- An assembly language program is translated into binary by an assembler. The assembler program reads each assembly instruction of a program as ASCII characters and translates them into the respective binary op-codes.

TABLE 4.1 Conversion of 0PIC18F SLEEP instruction into its Binary Op-Code

Assembly Code	Binary Form of ASCII Codes as seen by the Assembler	Binary Op-Code Created by the MPLAB PIC18F Assembler
S	0101 0011	
L	0100 1000	0000 0000 0000 0011
E	0100 0101	
E	0100 0101	
P	0101 0000	

確定為SLEEP，產生opcode

Assembly Language

- An advantage of the assembler is address computation. Most programs use addresses within the program as data storage or as targets for jumps or calls. When programming in machine language, these addresses must be calculated by the programmer. The assembler solves this problem by allowing the programmer to assign a symbol to an address. The programmer may then reference that address elsewhere by using the symbol.

↳可直接跳至指定label

Types of Assemblers

- One-Pass Assembler. This assembler goes through an assembly language program once and translates it into a machine language program. This assembler has the problem of defining forward references. This means that a JUMP instruction using an address that appears later in the program must be defined by the programmer after the program is assembled.

Types of Assemblers

- Two-Pass Assembler. This assembler scans an assembly language program twice. In the first pass, this assembler creates a symbol table. This way, labels can be used for JUMP statements and no address calculation has to be done by the user. On the second pass, the assembler translates the assembly language program into machine code.
- Note that MPLAB PIC18F assembler is a two-pass assembler.

subroutine：有return，compile時會跳至其他位置

mavro：沒有return，compile時會直接取代mavro部份(先寫好一份重複用)

Types of Assemblers

- Macroassembler. This assembler lets the programmer define all instruction sequences using macros. By using macros, the programmer can assign a name to an instruction sequence that appears repeatedly in a program. The programmer can thus avoid writing an instruction sequence that is required many times in a program by using macros. The macroassembler replaces a macroname with the appropriate instruction sequence each time it encounters a macroname.
- What is the difference between macroprogram and subroutine?

Assembler Delimiters

- Each line of an assembly language program consists of four fields. Most assemblers allow the programmer to use a special symbol or delimiter to indicate the beginning or end of each field. Typical delimiters:
  - Spaces are used between fields.
  - Commas (,) are used between addresses in an operand field.
  - A semicolon (;) is used before a comment.
  - A colon (:) or no delimiter is used after a label.

Specifying Numbers by Typical Assemblers

- Microchip's MPLAB assembler uses several ways to represent a hex number. Three most common ways are: 0x before the number, or H after the number, or default
- As an example, 60 in hexadecimal is represented by the MPLAB assembler as either 0x60 or 60H, or simply 60. → base16
- The MPLAB uses D before a 'number' to specify a decimal number. A binary number is specified by the MPLAB using B before the 'Number'.
  ↳ base10
  ↳ base2

Assembler Directives or Pseudoinstructions

- Assemblers use pseudoinstructions or directives to make the formatting of the edited text easier. Pseudoinstructions are not translated directly into machine language instructions. They equate labels to addresses, assign the program to certain areas of memory, or insert titles, page numbers, and so on.

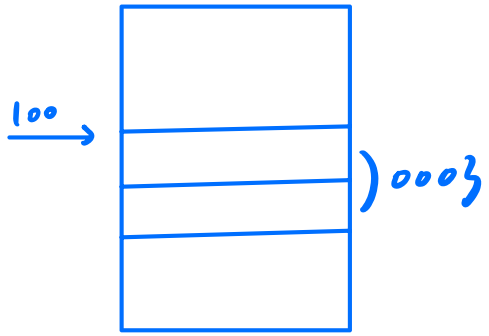
Assembler Directives or Pseudoinstructions

- **ORIGIN (ORG)** The directive ORG lets a programmer place programs anywhere in memory. Internally, the assembler maintains a program counter type of register called an *address counter*. This counter maintains the address of the next instruction or data to be processed.
- An ORG directive is similar JUMP. The JUMP instruction causes a processor to place a new address in the program counter. ORG causes the assembler to place a new value in the address counter.

Assembler Directives or Pseudoinstructions

Typical ORG statements are

```
ORG  0x100
SLEEP
```



The MPLAB assembler will generate the following code for these statements:

```
100 0003
```

SLEEP

Most assemblers assign a value of zero to the starting address of a program if the programmer does not define this by means of an ORG.

Assembler Directives or Pseudoinstructions

- **Equate (EQU)** The directive EQU assigns a value in its operand field to an address in its label field. This allows the user to assign a numerical value to a symbolic name. The user can then use the symbolic name in the program instead of its numeric value. This reduces errors.
- A typical example of EQU is START EQU 0x0200, which assigns the value 0200 in hexadecimal to the label START.

START = 0200

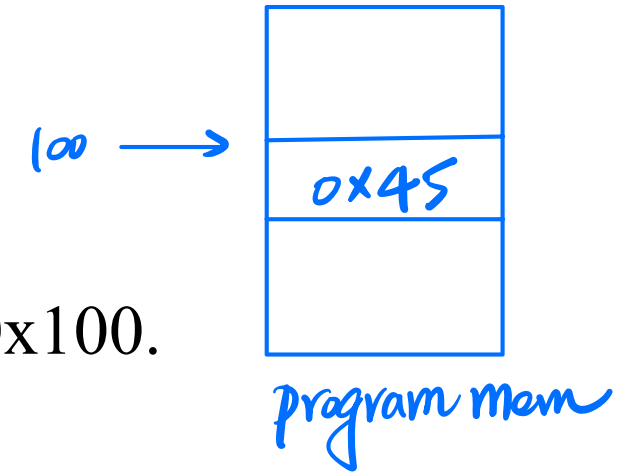
Assembler Directives or Pseudoinstructions

- **Define Byte (DB)** The directive DB (some assemblers use DC.B) is generally used to set a memory location in the program memory to a certain byte value.

↳ 設定 Program Mem 上的內容

```
ORG 0x100
DB 0x45
```

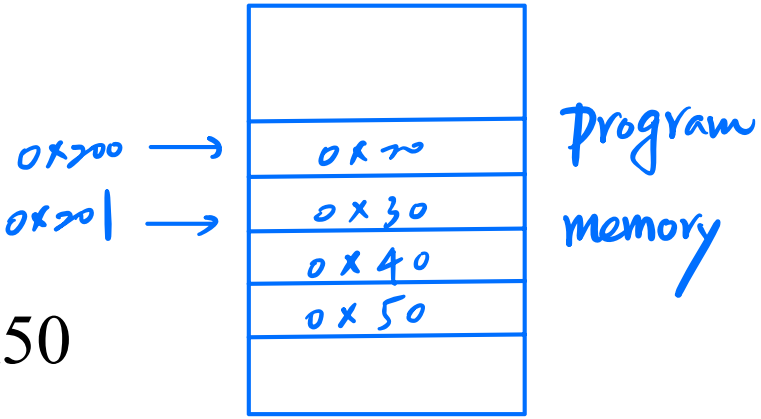
will store the data value 45 hex in address 0x100.



Assembler Directives or Pseudoinstructions

- With some assemblers, the DB pseudoinstruction can be used to generate a table of data as follows:

```
ORG 0x200
DB 0x20, 0x30, 0x40, 0x50
```



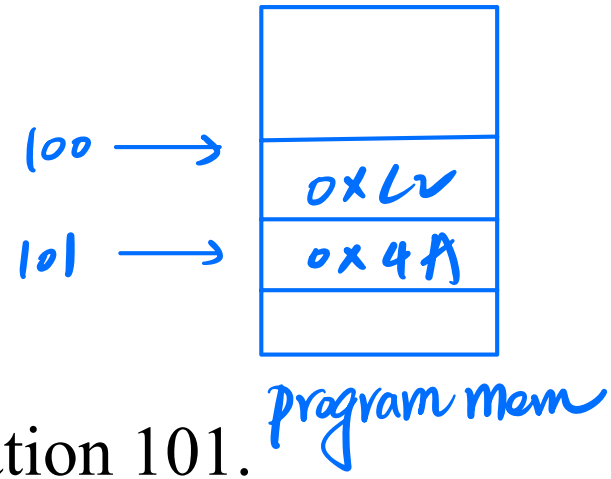
In this case, 20 hex is the first data of the memory location 0x200; 30 hex, 40 hex, and 50 hex occupy the next three memory locations. Therefore, the data in memory will look like this:

200	20
201	30
202	40
203	50

Assembler Directives or Pseudoinstructions

- **Define Word (DW)** The directive DW (some assemblers use DC.W) is typically used to assign a 16-bit value to two memory locations in the program memory.

```
ORG 0x100
DW 0x4AC2
```



will assign C2 to location 100 and 4A to location 101.

Assembler Directives or Pseudoinstructions

- The assembler will follow little endian; that is, it will assign the low byte first (C2) and then the high byte (4A). With some assemblers, the DW directive can be used to generate a table of 16-bit data as follows

```
ORG     0x80
DW      0x5000, 0x6000, 0x7000
```

81 80 83 82 85 84

the three 16-bit values 0x5000, 0x6000, and 0x7000 are assigned to memory locations starting at the address 0x80.

80	00
81	50
82	00
83	60
84	00
85	70

Assembly Language Instruction Formats

The general form of a *three-address instruction* is

<op-code> Addr1,Addr2,Addr3

Some typical *three-address instructions* are

```
MUL   A,B,C   ;   C ← A * B
ADD   A,B,C   ;   C ← A + B
SUB   R1,R2,R3  ;   R3 ← R1 - R2
```

destination

- The third address of this type of instruction is usually referred to as the destination address. The result of an operation is always assumed to be saved in the destination address.

Assembly Language Instruction Formats

Some typical *two-address instructions* are

```
MOV   A,R1      ;   R1 ← A
ADD   C,R2      ;   R2 ← R2 + C
SUB   R1,R2     ;   R2 ← R2 - R1
```

destination

In this format, the addresses Addr1 and Addr2 represent source and destination addresses, respectively.

Some typical *one-address instructions* are

```
LDA   B     ;   Acc ← B
ADD   C     ;   Acc ← Acc + C
MUL   D     ;   Acc ← Acc * D
STA   E     ;   E   ← Acc
```

Note that Acc means the accumulator.

All microcontrollers include some zero-address instructions in the instruction set. Typical examples of zero-address instructions are CLC (clear carry) and NOP (No operation).

Typical Instruction Set

- Each microcontroller has a unique instruction set designed by its manufacturer to do a specific task. We discuss some of the instructions that are common to all microcontrollers.
- Arithmetic Instructions Typical arithmetic instructions include ADD, SUBTRACT, COMPARE, MULTIPLY, and DIVIDE.

Typical Instruction Set

- The instruction set for a microprocessor typically includes the same ADD and SUBTRACT instructions for both unsigned and signed numbers. For example, consider adding two 8-bit numbers, A and B (A = FF_{16} and B = FF_{16}), using the ADD instruction by a microprocessor as follows:

$C_p=1 \oplus C_f=1 = 0$

$\Rightarrow$ no overflow

$$
\begin{array}{rl}
 & 1111111 \leftarrow \text{Intermediate carries} \\
 & FF_{16} = 11111111 \quad -1 \\
+ & FF_{16} = 11111111 \quad -1 \\
\hline
\text{final carry} & \rightarrow 111111110 = FE_{16} \quad -2
\end{array}
$$

Typical Instruction Set

- When the addition above is interpreted by the programmer as an unsigned operation, the result will be A+B=FF_{16} +FF_{16} =255_{10} +255_{10} =510_{10} which is FE_{16} with a carry, as shown above.
- However, if the addition is interpreted as a signed operation, then A+B = FF_{16} + FF_{16} = (-1_{10}) + (-1_{10}) = -2_{10} which is FE_{16} as shown above, and the final carry must be discarded by the programmer. Similarly, the unsigned and signed subtraction can be interpreted by the programmer.

Typical Instruction Set

- **Unsigned Multiplication**
- Several unsigned multiplication algorithms are available.

  Multiplication of two unsigned numbers can be accomplished via repeated addition. For example, to multiply 4_{10} by 3_{10}, the number 4_{10} can be added twice to itself to obtain the result, 12_{10}.

Typical Instruction Set

- **Signed Multiplication** ⇒ PDP-8 沒有
- A simple algorithm follows. Assume that M (multiplicand 被乘數) and Q (multiplier 乘數) are in two's-complement form. Assume that M_n and Q_n are the most significant bits (sign bits) of M and Q, respectively. The sign bit of the product is determined as $M_n \oplus Q_n$. (n=7 in pdp-8)
  ↳ 先判斷 sign bit 乘出來是正or負 $M_n \oplus Q_n$

Typical Instruction Set

1. If $M_n = 1$, compute the two's complement of M.
2. If $Q_n = 1$, compute the two's complement of Q.
3. Multiply the n - 1 bits of the multiplier and the multiplicand using unsigned multiplication.
4. The sign of the result, $S_n = M_n \oplus Q_n$.
5. If $S_n = 1$, compute the two's-complement of the result obtained in step 3.

先取出符號
再換成unsign相乘

ex: 1101 (-3) x 0100 (4)

$Q_3 = 1 \Rightarrow Q = 0011$ (3)

$\Rightarrow 0011 + 0011 + 0011 + 0011 = 00001100$

$\because S_n = 0 \oplus 1 = 1$

$\Rightarrow 11110100$ (-12)

Typical Instruction Set

- Assume that M and Q are two's-complement numbers. Suppose that $M = 1100_2$ and $Q = 0111_2$. Because $M_n = 1$, take the two's-complement of $M = 0100_2$; because $Q_n = 0$, do not change Q. Multiply 0111_2 and 0100_2 using the unsigned multiplication method discussed before. The product is 00011100_2. The sign of the product $S_n = M_n \oplus Q_n = 1 \oplus 0 = 1$. Hence, take the two's-complement of the product 00011100_2 to obtain 11100100_2, which is the final answer: -28_{10}.

Typical Instruction Set

- **Unsigned Division** Unsigned division can be accomplished via repeated subtraction. For example, consider dividing 7_{10} by 3_{10}:

Dividend	Divisor	Subtraction Result	Counter 商
7_{10}	3_{10}	$7 - 3 = 4$	1
		$4 - 3 = 1$	$1 + 1 = 2$

Quotient = counter value = 2

Remainder = subtraction result = 1 餘數

Here, 1 is added to a counter whenever the subtraction result is greater than the divisor. The result is obtained as soon as the subtraction result is smaller than the divisor.

Typical Instruction Set

- **Signed Division**
- Assume that DV (Dividend) and DR (Divisor) are in two's-complement form. For the first case, perform unsigned division using repeated subtraction of the magnitudes without the sign bits. The sign bit of the quotient is determined as $DV_n \oplus DR_n$, where DV_n and DR_n are the most significant bits (sign bits) of DV and DR, respectively.

先取出符號
換成正數除法

Typical Instruction Set

1. If $DV_n = 1$, compute the two's complement of DV, else keep DV unchanged.
2. If $DR_n = 1$, compute the two's complement of DR, else keep DR unchanged.
3. Divide the $n - 1$ bits of the dividend by the divisor using unsigned division algorithm (repeated subtraction).
4. The sign of the Quotient $Q_n = DV_n \oplus DR_n$. The sign of the remainder is the same as the sign of the dividend unless the remainder is zero. The following numerical examples illustrate this:
   The general equation for division can be used for signed division. Note that the general equation for division is *dividend* = *quotient* * *divisor* + *remainder*. For example, consider dividend = – 9, divisor = 2. Three possible solutions are shown below:
   (a) $-9 = -4 * 2 - 1$, Quotient = – 4, Remainder = – 1.
   (b) $-9 = -5 * 2 + 1$, Quotient = – 5, Remainder = +1.
   (c) $-9 = -6 * 2 + 3$, Quotient = – 6, Remainder = +3.
   However, the correct answer is shown in (a), in which, the Quotient = – 4 and the remainder = – 1. Hence, for signed division, the sign of the remainder is the same as the sign of the dividend, unless the remainder is zero.
5. If $Q_n = 1$, compute the two's-complement of the quotient obtained in step 3, else keep the quotient unchanged.

(I) 先 9÷2 = 4…1 再⇒ 商=-4, ∝=-1

(II) 9÷(-2)?
先 9÷2=4…1
再⇒ 商=-4, ∝=1

∝ 正負跟隨被除數！商看 DVn⊕DRn

Typical Instruction Set

Assume 4-bit numbers
Dividend = +6 = 0110_2 Divisor = -2 = Two's complement of 2 = 1110_2

Since the sign bit of Dividend is 0, do not change the dividend. Because the sign bit of the divisor is 1, take 2's complement of 1110 which is 0010. Now, divide 0110 by 0010 using repeated subtraction as follows:

DIVIDEND	DIVISOR	SUBTRACTION RESULT USING 2'S COMPLEMENT	COUNTER
0110	0010	0110-0010=0100	0001
		0100-0010=0010	0010
		0010-0010=0000	0011

Result of unsigned division: Quotient = Counter value = 0011_2,
Remainder = Subtraction result = 0000_2
Result of signed division 6 (0110) divided by -2 (1110):
Sign of the quotient = (Sign of dividend)$\oplus$(Sign of divisor) = $0 \oplus 1 = 1$
Hence, Quotient = 2's complement of $0011_2 = 1101_2 = -3_{10}$, Remainder = 0000_2

Typical Instruction Set

Dividend = -5 = Two's complement of $0101_2 = 1011_2$ Divisor = -2 = Two's complement of $2 = 1110_2$

Since the sign bit of Dividend is 1, take 2's complement of 1011 which is 0101. Because the sign bit of the divisor is 1, take 2's complement of 1110 which is 0010. Now, divide 0101 by 0010 using repeated subtraction as follows:

DIVIDEND	DIVISOR	SUBTRACTION RESULT USING 2'S COMPLEMENT	COUNTER
0101	0010	0101-0010=0011	0001
		0011-0010=0001	0010

↳不夠減了 ⇒ stop

Result of unsigned division: Quotient = Counter value = 0010_2,
Remainder = Subtraction result = 0001_2

Result of signed division -5 (1011) divided by -2 (1110):

Sign of the quotient = (Sign of dividend) ⊕ (Sign of divisor) = 1 ⊕ 1= 0. Hence, do not take two's complement of Quotient.

Quotient =0010_2 = $+2_{10}$, remainder has the same sign as the dividend which is negative (bit 3 = 1).

Hence, Remainder = 2's complement of $0001_2 = 1111_2 = -1_{10}$.

Note that the Sign of the quotient = (Sign of dividend) ⊕ (Sign of divisor). However, the sign of the remainder is the same as the sign of the dividend unless the remainder is 0.

Typical Instruction Set

Case 1: when the remainder is 0

i) Assume both the dividend and divisor are positive
Dividend = +6 Divisor = +2
Result: Quotient = +3 Remainder = 0

ii) Assume the dividend is negative and the divisor is positive.
Dividend = -6 Divisor = +2
Result: Quotient = -3 Remainder = 0

iii) Assume the dividend is positive and the divisor is negative.
Dividend = +6 Divisor = -2
Result: Quotient = -3 Remainder = 0

iv) Assume both the dividend and divisor are negative.
Dividend = -6 Divisor = -2
Result: Quotient = +3 Remainder = 0

Case 2: when the remainder is nonzero

Since, Dividend = Quotient x Divisor + Remainder
Hence, Remainder = Dividend - Quotient x Divisor.

i) Assume both the dividend and divisor are positive.
Dividend = +5 Divisor = +2
Result: Quotient = +2, Remainder can be obtained from the equation, Remainder = Dividend - Quotient x Divisor. Hence, Remainder = +5 - (+2 x +2) = +1.

ii) Assume the dividend is negative and the divisor is positive.
Dividend = -5 Divisor = +2
Result: Quotient = -2 Remainder can be obtained from the equation, Remainder = Dividend - Quotient x Divisor. Hence, Remainder = -5 - (-2 x +2) = - 1.

iii) Assume the dividend is positive and the divisor is negative.
Dividend = +5 Divisor = -2
Result: Quotient = -2, Remainder can be obtained from the equation, Remainder = Dividend - Quotient x Divisor. Hence, Remainder = +5 - (-2 x --2) = + 1.

iv) Assume both dividend and divisor are negative.
Dividend = -5 Divisor = -2
Result: Quotient = +2, Remainder can be obtained from the equation, Remainder = Dividend - Quotient x Divisor. Hence, Remainder = -5 - (+2 x -2) = - 1.
From the above, the sign of the remainder is the same as the sign of the dividend unless the remainder is zero.

Typical Instruction Set

- RISC microcontrollers such as the PIC18F include the unsigned multiplication instruction. The PIC18F instruction set does not contain instructions for signed multiplication, and unsigned and signed division. However, subroutines using PIC18F assembly language can be written to obtain them using the above algorithms.

Typical Instruction Set

- **Logic Instructions**. Typical logic instructions perform traditional Boolean operations such as AND, OR, and Exclusive-OR. The AND instruction can be used to perform a masking operation. If the bit value in a particular bit position is desired in a word, the word can be logically ANDed with appropriate data to accomplish this. For example, the bit value at bit 2 of an 8-bit number 0100 1Y10 can be obtained as follows:

```
        0 1 0 0  1 Y 1 0  -- 8-bit number
AND     0 0 0 0  0 1 0 0  -- masking data
        -------------------
        0 0 0 0  0 Y 0 0  -- result
```

以X取某bit

Typical Instruction Set

- If the bit value Y at bit 2 is 1, the result is nonzero (flag Z = 0). The Z flag can be tested using typical conditional JUMP instructions such as JZ (Jump if Z=1) or JNZ (Jump if Z = 0). This is called a *masking operation*. The AND instruction can also be used to determine whether a binary number is ODD or EVEN by checking the least significant bit (LSB) of the number.
- The OR instruction can be used to insert a 1 in a particular bit position of a binary number without changing other bits.

Typical Instruction Set

- For example, a 1 can be inserted using the OR instruction at bit 3 of the 8-bit binary number 0 1 1 1 0 0 1 1 without changing the values of the other bits:

```
                0 1 1 1 0 0 1 1 -- 8-bit  number
OR              0 0 0 0 1 0 0 0 -- data for inserting a 1 at  bit 3
                -------------------
                0 1 1 1 1 0 1 1 -- result
```

強制設定某bit為1

Typical Instruction Set

- The Exclusive-OR instruction can be used to find the one's-complement of a binary number by XORing the number with all 1's as follows:

```
        0 1 0 1 1 1 0 0 --  8-bit number
XOR     1 1 1 1 1 1 1 1 --  data
        -----------------------
        1 0 10 0 0 1 1 -- Result ( One's Complement  of the 8-bit
                                   number  0 1 0 1 1 1 0 0)
```

做1補數

將二進位數每個數字反轉，得到的數即為原二進位的一補數

Typical Instruction Set

- **Shift and Rotate Instructions**.
- In a *logical shift* operation, a bit that is shifted out will be lost, and the vacant position will be filled with a 0. For example, if we have the number $(11)_{10}$, after a logical right shift operation, the register contents shown in Figure 4.2 will occur.
- Typical examples of logic/arithmetic and shift/rotate operations are given in Table 4.2.

Typical Instruction Set

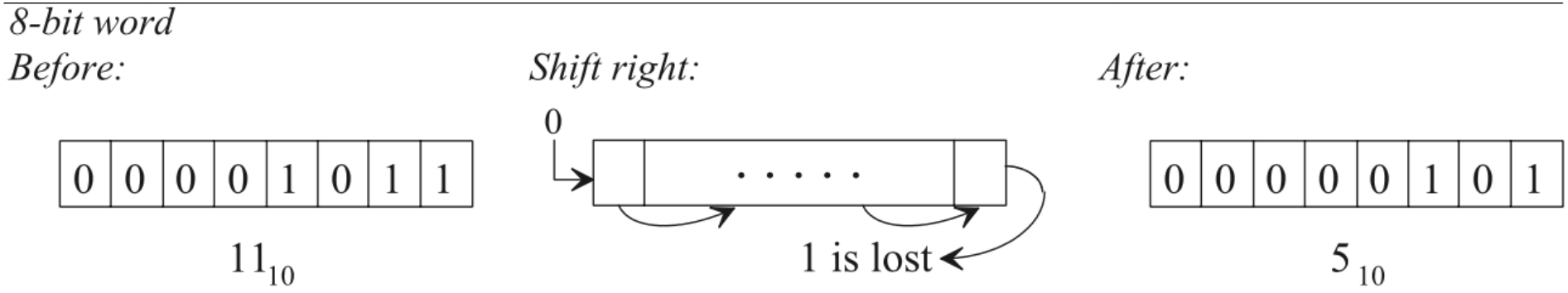


FIGURE 4.2 Logical right shift operation.

TABLE 4.2 Typical Logic/Arithmetic and Shift/ Rotate Operations

Shift type	Logic	Arithmetic	Rotate
Right	0 → Lost	保留 sign bit ! MSB → Lost	
Left	0 → Lost	0 → Lost	

Typical Instruction Set

- It must be emphasized that a logical left or right shift of an unsigned number by n positions implies multiplication or division of the number by 2^n, respectively, provided that a 1 is not shifted out during the operation.

Typical Instruction Set

- In the case of *true arithmetic left or right shift* operations, the sign bit of the number to be shifted must be retained. However, in computers, this is true for right shift and not for left shift operation. For example, if a register is shifted right arithmetically, the most significant bit (MSB) of the register is preserved, thus ensuring that the sign of the number will remain unchanged. This is illustrated in Figure 4.3.

Logical shift treats the number as a bunch of bits.
Arithmetic shift treats the number as a signed integer (in 2s complement), and "retains" the MSB.

Typical Instruction Set

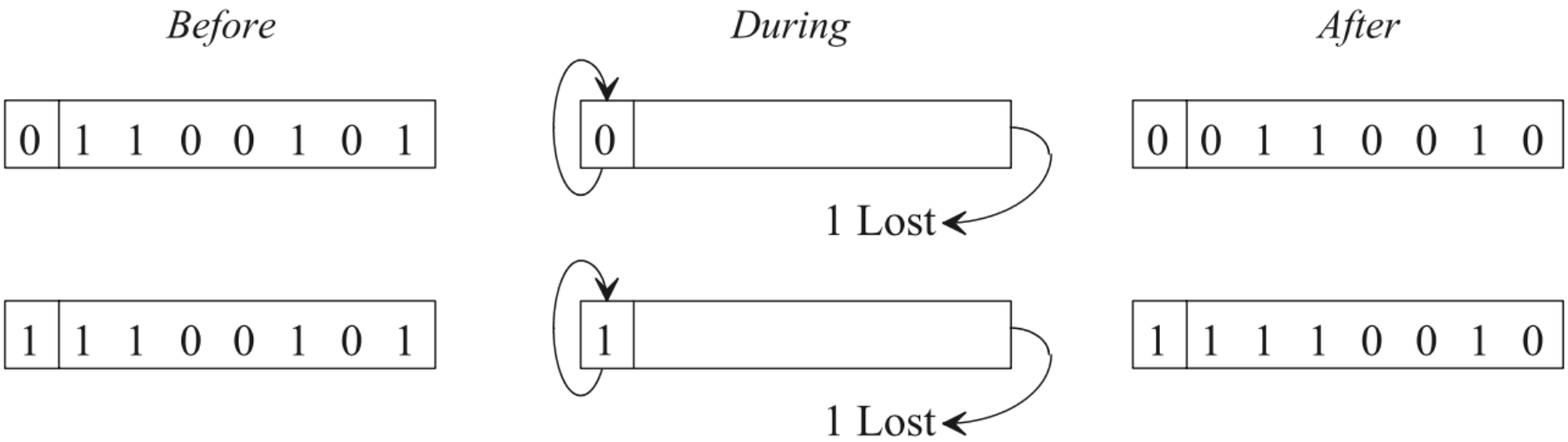


FIGURE 4.3 True arithmetic right shift operation.

先右移，再補回原本 sign bit

Typical Instruction Set

- If the most significant bit changes from 0 to 1 or vice versa in an arithmetic left shift, the result is incorrect and the CPU sets the overflow flag to 1. For example, if the original value of the register is ($+3_{10}$), the results of two successive arithmetic left shift operations are interpreted as follows:

Original	*After first shift*	*After second shift*
$0011_2 = (+3_{10})$	$0110_2 = (+6_{10})$	$1100_2 = (-4_{10})$; the result is incorrect
	$3 \times 2 = 6$; correct	$6 \times 2 = 12$ not -4; incorrect

The positive number ($+3_{10}$) is shifted arithmetically twice to the left generating a negative result (-4_{10}). Hence the overflow flag will be set to one.

Typical Instruction Set

- **Instructions for controlling microcontroller operations**. These instructions typically include those that set the reset specific flags and halt or stop the CPU.
- **Data movement instructions**. These instructions move data from a register to memory, and vice versa, between registers and between a register and an I/O device.

Typical Instruction Set

- **Instructions using memory addresses**. An instruction in this category typically contains a memory address, which is used to read a data word from memory into a microcontroller register or for writing data from a register into a memory location. Many instructions under data processing and movement fall in this category.

Typical Instruction Set

- **Conditional and unconditional JUMP**. These instructions typically include one of the following:
  - 1. An unconditional JUMP, which always transfers the memory address specified in the instruction into the program counter.
  - 2. A conditional JUMP, which transfers the address portion of the instruction into the program counter based on the conditions set by one of the status flags in the flag register.

ex: BZ, BNZ

Typical Addressing Modes

解讀 opcode + operand

- One of the tasks performed by a microcontroller during execution of an instruction is the determination of the operand and destination addresses. The manner in which a microcontroller accomplishes this task is called the "addressing mode".
- An instruction is said to have "implied or inherent addressing mode" if it does not have any operand. The SLEEP instruction is a no-operand instruction.

Typical Addressing Modes

- Whenever an instruction/operand contains data, it is called an "immediate mode" instruction. For example,

```
ADDLW  3        ;  [WREG] ← [WREG] + 3
```

This instruction adds 3 to the contents of the WREG and then stores the result in WREG.

Typical Addressing Modes

- An instruction is said to have an *absolute or direct addressing mode* if it contains a memory address in the operand field. For example, `MOVWF 0x20 ; [0x20] ← [WREG]`

  The MOVWF 0x20 instruction moves the contents of the WREG register into a memory location whose address is 0x20. The contents of WREG are unchanged. MOVWF 0x20 uses direct address mode since address 0x20 is directly specified.

Typical Addressing Modes

- When an instruction specifies a microcontroller register to hold the address, the resulting addressing mode is known as the *register indirect mode*. pointer

```
MOVWF       INDF0   ; Move contents of WREG into a data RAM
                    ; address pointed  to by FSR0 since INDF0
                    ; is associated with FSR0
```

This instruction moves the contents of WREG to a data memory location whose address is in PIC18F's FSR0 register.

Typical Addressing Modes

PC ← PC + 2 + 2 × offset

- Conditional branch instructions are used to change the order of execution of a program based on the conditions set by the status flags. The op-code verifies a condition set by a particular status flag. If the condition is satisfied, the program counter is changed to the value of the operand address (defined in the instruction). If the condition is not satisfied, the program counter is incremented, and the program is executed in its normal order.

Typical Addressing Modes

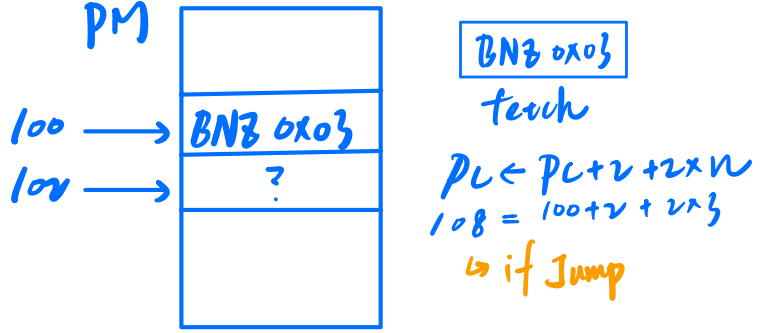
- PIC18F uses conditional branch instructions. Some conditional branch instructions are 16 bits wide. The first byte is the op-code for checking a particular flag. The second byte is an 8-bit offset, which is added to the contents of the program counter if the condition is satisfied to determine the effective address. This offset is considered as a signed binary number with the most significant bit as the sign bit. It means that the offset can vary from -128_{10} to $+127_{10}$. This is called the *relative mode*.

Typical Addressing Modes

- For forward branching, the range of the offset value is from 0x00 to 0x7F. For backward branching, this range varies from 0x80 to 0xFF. Since conditional branch instructions are 16-bit wide in the PIC18F, the PC (Program Counter) is incremented by 2 to point to the next instruction while executing the conditional branch instruction. The offset is multiplied by 2 and then added to PC+2 to find the branch address if the condition is true.

BNZ	**Branch if Not Zero**
Syntax:	BNZ n
Operands:	$-128 \le n \le 127$ ⇒ 8 bit
Operation:	if Zero bit is '0', (PC) + 2 + 2n → PC

Typical Addressing Modes



- Consider BNZ 0x03. Note that BNZ stands for "Branch if not zero". If the Z (zero flag) in the Status register is 0, then the PC is loaded with the address computed from (PC + 2 + 03H x 2). When the PIC18F executes the BNZ instruction, the PC points to the next instruction. This means that if BNZ is located at address 0050H in program memory, the PC will contain 0052H (PC + 2) when the PIC18F executes BNZ.

Example: `HERE BNZ Jump`

Before Instruction
PC = address (HERE)
After Instruction
If Zero = 0;
PC = address (Jump)
If Zero = 1;
PC = address (HERE + 2)

Typical Addressing Modes

- Hence, if Z = 0, then after execution of the BNZ 0x03 instruction, the PC will be loaded with address 0058H (0052H + 03H x 2). Hence, the program will branch to address 0058H. This is called "Relative Addressing Mode". Note that the Relative mode is useful for developing position independent code.

Subroutine Calls in Assembly Language

- A subroutine can be defined as a program carrying out a particular function that can be called by another program, known as the main program. The subroutine only needs to be placed once in memory starting at a particular memory location. Each time the main program requires this subroutine, it can branch to it, typically by using PIC18F's CALL to subroutine (CALL) instruction along with its starting address.