2

68HC12 Assembly Programming

2.1 Objectives

After completing this chapter you should be able to:

- Use assembler directives to allocate memory blocks, define constants, and create a message to be output
- Write assembly programs to perform simple arithmetic operations
- Write program loops to perform repetitive operations
- Use program loops to create time delays
- Use Boolean and bit manipulation instructions to perform bit field manipulations

2.2 Assembly Language Program Structure

An assembly language program consists of a sequence of statements that tells the computer to perform the desired operations. From a global point of view, a 68HC12 assembly program consists of three sections. In some cases these sections can be mixed to provide better algorithm design. The three sections are:

- *Assembler directives*. Assembler directives instruct the assembler how to process subsequent assembly language instructions. Directives also provide a way to define program constants and reserve space for dynamic variables. Some directives may also set a location counter.
- *Assembly language instructions*. These instructions are 68HC12 instructions. Some instructions are defined with labels.
- *Comments*. There are two types of comments in an assembly program. The first type is used to explain the function of a single instruction or directive. The second type explains the function of a group of instructions or directives or a whole routine. Adding comments makes a program more readable.

Each line of a 68HC12 assembly program, excluding certain special constructs, is comprised of four distinct fields. Some of the fields may be empty. The order of these fields is:

1. Label
2. Operation
3. Operand
4. Comment

2.2.1 The Label Field

Labels are symbols defined by the user to identify memory locations in the programs and data areas of the assembly module. For most instructions and assembler directives, the label is optional. The rules for forming a label are as follows:

- A label must start at column one and begin with a letter (A-Z, a-z), and the letter can be followed by letters, digits, or special symbols. Some assemblers permit special symbols to be used. For example, the assembler from IAR Inc. allows a symbol to start with a question mark (**?**), at character (**@**), and underscore (_), in addition to letters. Digits and the dollar (**$**) character can also be used after the first character in the IAR assembler.
- Most assemblers restrict the number of characters in a label name. The as12 assembler reference manual does not mention the limit. The IAR assembler allows a user-defined symbol to have up to 255 characters.
- The as12 assembler allows a label to be terminated by "**:**".

Example 2.1

Valid and Invalid labels.

The following instructions contain valid labels:

```
begin   ldaa        #10         ; label begins in column 1
```

```
print:  jsr     hexout    ; label is terminated by a colon
jmp     begin             ; instruction references the label begin
```

The following instructions contain invalid labels:

```
here is adda    #5        ; a space is included in the label
loop    deca              ; labels begins at column 2
```

2.2.2 The Operation Field

This field contains the mnemonic names for machine instructions and assembler directives. If a label is present, the opcode or directive must be separated from the label field by at least one space. If there is no label, the operation field must be at least one space from the left margin.

Example 2.2

Examples of operation fields

```
adda    #$02              ; adda is the instruction mnemonic
true    equ     1         ; equate directive equ occupies the operation field
```

2.2.3 The Operand Field

If an *operand field* is present, it follows the *operation field* and is separated from the operation field by at least one space. The operand field may contain operands for instructions or arguments for assembler directives. The following instructions include the operand field:

```
TCNT    equ     $0084     ; $0084 is the operand field
TC0     equ     $0090     ; $0090 is the operand field
```

2.2.4 The Comment Field

The comment field is optional and is added mainly for documentation purposes. The comment field is ignored by the assembler. Here are the rules for comments:

- Any line beginning with an * is a comment.
- Any line beginning with a ; (semi-colon) is a comment. In this book, we will use ; to start a comment.
- You must have a ; (semi-colon) prefixing any comment on a line with mnemonics.

Examples of comments appear in the following instructions:

```
; this program computes the square root of N 8-bit integers.
org     $1000             ; set the location counter to $1000
dec     lp_cnt            ; decrement the loop count
```

In this chapter, we will use the Motorola Freeware cross-assembler **as12** as the standard to explain every aspects of the assembly programming.

2.3 Assembler Directives

Assembler directives look just like instructions in an assembly language program, but they tell the assembler to do something other than creating the machine code for an instruction. The available assembler directives vary with the assembler. Interested readers should refer to the user's manual of the specific assembler for details.

We will discuss assembler directives supported by the as12 in detail. In the following discussion, statements enclosed in brackets **[]** are optional. All directives and assembly instructions can be in either upper- or lowercase.

End

The end directive is used to end a program to be processed by the assembler. In general, an assembly program looks like this:

```
(your program)
end
```

The **end** directive indicates the logical end of the source program. Any statement following the end directive is ignored. A warning message will be raised if the end directive is missing from the source code; however, the program will still be assembled correctly.

org (origin)

The assembler uses a *location counter* to keep track of the memory location where the next machine code byte should be placed. If the programmer wants to force the program or data array to start from a certain memory location, then this directive can be used.

For example, the statement:

```
org     $1000
```

forces the location counter to be set to $1000.

The **org** directive is mainly used to force a data table or a segment of instructions to start with a certain address. As a general rule, this directive should be used as infrequently as possible. Using too many **orgs** will make your program less reusable.

db (define byte)
dc.b (define constant byte)
fcb (form constant byte)

These three directives define the value of a byte or bytes that will be placed at a given memory location. The **db** (or fcb, or dc.b) directive assigns the value of the expression to the memory location pointed to by the location counter. Then the location counter is incremented. Multiple bytes can be defined at a time by using commas to separate the arguments.

For example, the statement:

```
array   db          $11,$22,$33,$44,$55
```

initializes five bytes in memory to:

```
$11
$22
$33
$44
$55
```

and the assembler will use **array** as the symbolic address of the first byte whose initial value is $11. The program can also force these five bytes to a particular address by adding the *org* directive. For example, the sequence:

```
org     $800
array   db      $11,$22,$33,$44,$55
```

initializes the contents of memory locations at $800, $801, $802, $803, and $804, to $11, $22, $33, $44, and $55, respectively.

dw (define word)
dc.w (define constant word)
fdb (form double bytes)

These three directives define the value of a word or words that will be placed at a given address. The value can be specified by an integer or an expression. For example, the statement:

```
vect_tab        dw      $1234, $5678
```

initializes the two words starting from the current location counter to $1234 and $5678, respectively. After this statement, the location will be incremented by four.

fcc (form constant character)

This directive allows us to define a string of characters (a message). The first character in the string is used as the delimiter. The last character must be the same as the first character because it will be used as the delimiter. The delimiter must not appear in the string. The space character cannot be used as the delimiter. Each character is encoded by its corresponding ASCII code.

For example, the statement:

```
alpha  fcc      "def"
```

will generate the following values in memory:

```
$64
$65
$66
```

and the assembler will use the label **alpha** to refer to the address of the first letter, which is stored as the byte $64. A character string to be output to the LCD display is often defined using this directive.

fill (fill memory)

This directive allows a user to fill a certain number of memory locations with a given value. The syntax of this directive is as follows:

```
fill    value, count
```

where the number of bytes to be filled is indicated by **count** and the value to be filled is indicated by **value.**

For example, the statement:

```
space_line      fill    $20, 40
```

will fill 40 bytes with the value of $20 starting from the memory location referred to by the label **space_line**.

ds (define storage)
rmb (reserve memory byte)
ds.b (define storage bytes)

Each of these three directives reserves a number of bytes given as the arguments to the directive. The location counter will be incremented by the number that follows the directive mnemonic.

For example, the statement:

```
buffer  ds      100
```

reserves 100 bytes in memory starting from the location represented by the label **buffer**. After this directive, the location counter will be incremented by 100. The content(s) of the reserved memory location(s) are not defined.

ds.w (define storage word)
rmw (reserve memory word)

Each of these directives increments the location counter by the value indicated in the number-of-words argument multiplied by two. In other words, if the ds.w expression evaluates to **k** then the location counter is advanced by 2k. These directives are often used with a label. For example, the statement:

```
dbuf    ds.w    20
```

reserves 40 bytes starting from the memory location represented by the label *dbuf*. None of these 40 bytes are initialized.

equ (equate)

This directive assigns a value to a label. Using **equ** to define constants will make our program more readable.

For example, the statement:

```
loop_cnt        equ     40
```

informs the assembler that whenever the symbol *loop_cnt* is encountered, it should be replaced with the value of 40.

loc

This directive increments and produces an internal counter used in conjunction with the backward tick mark (`). By using the **loc** directive and the ` mark you can write program segments like the following example, without thinking up new labels:

```
                loc
                ldaa    #2
loop`           deca
                bne     loop`
                loc
loop`           brclr   0,x $55 loop`
```

This code segment will work perfectly fine because the first loop label will be seen as loop001, whereas the second loop label will be seen as loop002. The assembler actually sees this:

```
                loc
                ldaa    #2
loop001         deca
                bne     loop001
                loc
loop002         brclr   0,x $55 loop002
```

You can also set the **loc** directive with a valid expression or number by putting that expression or number in the operand field. The resultant number will be used to increment the suffix to the label.

2.4 Software Development Issues

A complete discussion of issues involved in software development is out of the scope of this text. However, we do need to take a serious look at some software development issues because embedded system designers must spend a significant amount of time on software development.

As we all know, software development starts with *problem definition*. The problem presented by the application must be fully understood before any program can be written. At the problem definition stage, the most critical thing is to get you, the programmer, and your end user to agree upon what needs to be done. To achieve this, asking questions is very important. For complex and expensive applications, a formal, written definition of the problem is formulated and agreed upon by all parties.

Once the problem is known, the programmer can begin to lay out an overall plan of how to solve the problem. The plan is also called an *algorithm*. Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input, and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transforms the input into the output. We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

An algorithm is expressed in *pseudocode* that is very much like C or PASCAL. What separates pseudocode from "real" code is that in pseudocode, we employ whatever expressive method that is most clear and concise to specify a given algorithm. Sometimes the clearest method is English, so do not be surprised if you come across an English phrase or sentence embedded within a section of "real" code.

An algorithm provides not only the overall plan for solving the problem but also documentation to the software to be developed. In the rest of this book, all algorithms will be presented in the following format:

Step 1
….
Step 2
….

An earlier alternative for providing the overall plan for solving software problems was the use of flowcharts. A flowchart shows the way a program operates. It illustrates the logic flow of the program. Therefore, flowcharts can be a valuable aid in visualizing programs. Flowcharts are not only used in computer programming, they are also used in many other fields, such as business and construction planning.

The flowchart symbols used in this book are shown in Figure 2.1. The *terminal symbol* is used at the beginning and end of each program. When it is used at the beginning of a program, the word *Start* is written inside it. When it is used at the end of a program, it contains the word *Stop*.

The *process box* indicates what must be done at this point in the program execution. The operation specified by the process box could be shifting the contents of one general purpose register to a peripheral register, decrementing a loop count, and so on.

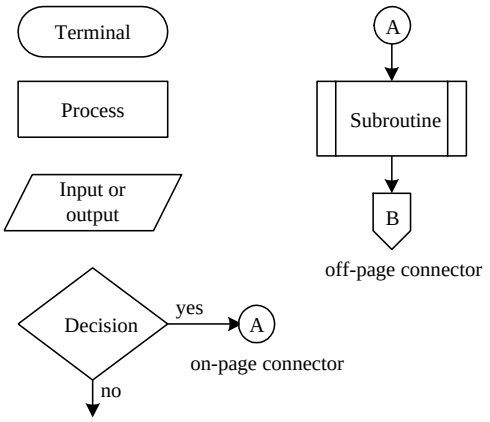


Figure 2.1 ■ Flowchart symbols used in this book

The *input/output box* is used to represent data that are either read or displayed by the computer.

The *decision box* contains a question that can be answered either yes or no. A decision box has two exits, also marked yes or no. The computer will take one action if the answer is yes and will take a different action if the answer is no.

The *on-page connector* indicates that the flowchart continues elsewhere on the same page. The place where it is continued will have the same label as the on-page connector. The *off-page connector* indicates that the flowchart continues on another page. To determine where the flowchart continues, you need to look at the following pages of the flowchart to find the matching off-page connector.

Normal flow on a flowchart is from top to bottom and from left to right. Any line that does not follow this normal flow should have an arrowhead on it.

When the program gets complicated, the flowchart that documents the logic flow of the program also becomes difficult to follow. This is the limitation of the flowchart. In this book, we will use both the flowchart and the algorithm procedure to describe the solution to a problem.

After you are satisfied with the algorithm or the flowchart, *convert it to source code in one of the assembly or high-level languages*. Each statement in your algorithm (or each block of your flowchart) will be converted into one or multiple assembly instructions or high-level language statements. If you find an algorithmic step (or a block in the flowchart) requires many assembly instructions or high-level language statements to implement, then it might be beneficial to either (1) convert this step (or block) into a subroutine and just call the subroutine, or (2) further divide the algorithmic step (or flowchart block) into smaller steps (or blocks) so that it can be coded with just a few assembly instructions or high-level language statements.

The next major step is to *test your program*. Testing a program means testing for anomalies. Here you will first test for normal inputs that you always expect. If the result is what you expected then you go on to test the borderline inputs. Test for the maximum and minimum values of the input. When your program passes this test, then test for illegal input values. If your algorithm includes several branches, then you must use enough values to exercise all the possible branches. This is to make sure that your program will operate correctly under all possible circumstances.

In the rest of this book, most of the examples are well defined. Therefore, our focus is on how to design the algorithm that solves the specified problem as well as how to convert the algorithm into source code.

2.5 Writing Programs to do Arithmetic

In this section, we will use small programs that perform simple computations to demonstrate how a program is written.

Example 2.3

Write a program to add the numbers stored at memory locations $800, $801, and $802, and store the sum at memory location $900.

Solution: This problem can be solved by the following steps:

Step 1
Load the contents of the memory location at $800 into accumulator A.

Step 2
Add the contents of the memory location at $801 into accumulator A.

Step 3
Add the contents of the memory location at $802 into accumulator A.

Step 4
Store the contents of accumulator A at memory location $900.
These steps can be translated into the as12 assembly program as follows:

```
org     $1000   ; starting address of the program
ldaa    $800    ; place the contents of the memory location $800 into A
adda    $801    ; add the contents of the memory location $801 into A
adda    $802    ; add the contents of the memory location $802 into A
staa    $900    ; store the sum at the memory location $900
end
```

Example 2.4

Write a program to subtract the contents of the memory location at $805 from the sum of the memory locations at $800 and $802, and store the result at the memory location $900.

Solution: The logic flow of this program is illustrated in Figure 2.2. The assembly program is as follows:

```
org     $1000   ; starting address of the program
ldaa    $800    ; copy the contents of the memory location at $800 to A
adda    $802    ; add the contents of memory location at $802 to A
suba    $805    ; subtract the contents of memory location at $805 from A
staa    $900    ; store the contents of accumulator A to $805
end
```

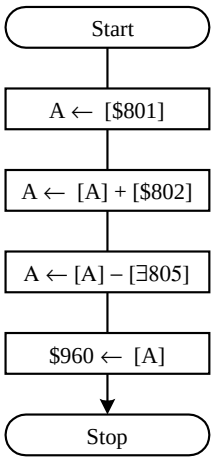


Figure 2.2 ■ Logic flow of program 2.4

Example 2.5

Write a program to subtract five from four memory locations at $800, $801, $802, and $803.

Solution: In the 68HC12, a memory location cannot be the destination of an ADD or SUB instruction. Therefore, three steps must be followed to add or subtract a number to or from a memory location:

Step 1
Load the memory contents into an accumulator.

Step 2
Add (or subtract) the number to (from) the accumulator.

Step 3
Store the result at the specified memory location.

The program is as follows:

```
org     $1000
ldaa    $800      ; copy the contents of memory location $800 to A
suba    #5        ; subtract 5 from A
staa    $800      ; store the result back to memory location $800
ldaa    $801
suba    #5
staa    $801
ldaa    $802
suba    #5
staa    $802
```

```
ldaa   $803
suba   #5
staa   $803
end
```

Example 2.6

Write a program to add two 16-bit numbers that are stored at $800~$801 and $802~$803, and store the sum at $900~$901.

Solution: This program is very straightforward:

```
org    $1000
ldd    $800      ; place the 16-bit number at $800~$801 in D
addd   $802      ; add the 16-bit number at $802~$803 to D
std    $900      ; save the sum at $900~$901
end
```

2.5.1 Carry/Borrow Flag

The 68HC12 can add and subtract either 8-bit or 16-bit numbers and place the result in either 8-bit accumulators, A or B, or the double accumulator D. The 8-bit number stored in accumulator B can also be added to index register X. However, programs can also be written to add numbers larger than 16 bits. Arithmetic performed in a 16-bit microprocessor/microcontroller on numbers that are larger than 16 bits is called *multiprecision arithmetic*. Multiprecision arithmetic makes use of the carry flag (C flag) of the condition code register (CCR).

Bit 0 of the CCR register is the C flag. It can be thought of as a temporary 9th bit that is appended to any 8-bit register or 17th bit that is appended to any 16-bit register. The C flag allows us to write programs to add and subtract hex numbers that are larger than 16-bit. For example, consider the following two instructions:

```
ldd    #$8645
addd   #$9978
```

These two instructions add the numbers $8645 and $9978.

```
 $8645
+$9978
$11FBD
```

The result is $11FBD, a 17-bit number, which is too large to fit into the 16-bit double accumulator D. When the 68HC12 executes these two instructions, the lower sixteen bits of the answer, $1FBD, are placed in double accumulator D. This part of the answer is called the *sum*. The leftmost bit is called a *carry.* A carry of 1 following an addition instruction sets the C flag of the CCR register to 1. A carry of 0 following an addition clears the C flag to 0. This applies to both 8-bit and 16-bit additions for the 68HC12. For example, execution of the following two instructions:

```
ldd    #$1245
addd   #$4581
```

will clear the C flag to 0 because the carry resulting from this addition is 0. In summary:

- If the addition produces a carry of 1, the carry flag is set to 1.
- If the addition produces a carry of 0, the carry flag is cleared to 0.

2.5.2 Multiprecision Addition

For a 16-bit microcontroller like the 68HC12, multiprecision addition is the addition of numbers that are larger than 16 bits. To add the hex number $1A598183 to $76548290, the 68HC12 has to perform multiprecision addition.

```
  1    1  1
 $1A598183
+$76548290
 $90AE0413
```

Multiprecision addition is performed one byte at a time, beginning with the least significant byte. The 68HC12 does allow us to add 16-bit numbers at a time because it has the ADDD instruction. The following two instructions can be used to add the least significant 16-bit numbers together:

```
ldd   #$8183
addd  #$8290
```

Since the sum of the most significant digit has a sum greater than 16, it generates a carry that must be added to the next more significant digit, causing the C flag to be set to 1. The contents of double accumulator D must be saved before the higher bytes are added. Let's save these two bytes at $802-$803:

```
std   $802
```

When the second-to-most significant bytes are added, the carry from the lower byte must be added in order to obtain the correct sum. In other word, we need an "add with carry" instruction. There are two versions of this instruction: the ADCA instruction for accumulator A and the ADCB instruction for accumulator B. The instructions for adding the second-to-most-significant bytes are:

```
ldaa  #$59
adca  #$54
```

We also need to save the *second-to-most-significant* byte of the result at $801 with the following instruction:

```
staa  $801
```

The most significant bytes can be added using similar instructions, and the complete program with comments appears as follows:

```
ldd   #$8183   ; place the lowest two bytes of the first number in D
addd  #$8290   ; add the lowest two bytes of the second number to D
std   $802     ; store the lowest two bytes of the sum at $802-$803
ldaa  #$59     ; place the second-to-most significant byte of the first number in A
adca  #$54     ; add the second-to-most-significant byte of the
               ; second number and carry to A
staa  $801     ; store the second-to-most-significant byte of the sum at $801
ldaa  #$1A     ; place the most-significant byte of the first number in A
adca  #$76     ; add the most-significant byte of the second number and carry to A
staa  $800     ; store the most significant byte of the sum
end
```

Note that the LOAD and STORE instructions do not affect the value of the C flag (otherwise, the program would not work). The 68HC12 does not have a 16-bit instruction with the carry flag as an operand. Whenever the carry needs to be added, we must use the 8-bit instruction ADCA or ADCB. This is shown in the previous program.

Example 2.7

Write a program to add two 4-byte numbers that are stored at $800~$803 and $804~$807, and store the sum at $810~$813.

Solution: The addition should start from the least significant byte and proceed to the most significant byte. The program is as follows:

```
org    $1000    ; starting address of the program
ldd    $802     ; place the lowest two bytes of the first operand in D
addd   $806     ; add the lowest two bytes
std    $812     ; save the sum of the lowest two bytes
ldaa   $801     ; place the second-to-most-significant byte of the
                ; first operand in A
adca   $805     ; add the second-to-most-significant byte of the
                ; second operand and carry to A
staa   $811     ; save the sum of the second-to-most significant bytes
ldaa   $800     ; place the most-significant byte of the first operand in A
adca   $804     ; add the most-significant byte of the second
                ; operand and carry to A
staa   $810     ; save the sum of the most-significant bytes
end
```

2.5.3 Subtraction & the C Flag

The C flag also enables the 68HC12 to borrow from the high byte to the low byte during a multiprecision subtraction. Consider the following subtraction problem:

```
  $39
−$74
```

We are attempting to subtract a larger number from a smaller one. Subtracting $4 from $9 is not a problem:

```
  $39
−$74
    5
```

Now we need to subtract $7 from $3. To do this, we need to borrow from somewhere. The 68HC12 borrows from the C flag, thus setting the C flag. When we borrow from the next higher digit of a hex number, the borrow has a value of decimal 16. After the borrow from the C flag, the problem can be completed:

```
  $39
−$74
  $C5
```

When the 68HC12 executes a subtract instruction, it always borrows from the C flag. The borrow is either 1 or 0. The C flag operates as follows during a subtraction:

- If the 68HC12 borrows a 1 from the C flag during a subtraction, the C flag is set to 1.
- If the 68HC12 borrows a 0 from the C flag during a subtraction, the C flag is set to 0.

2.5.4 Multiprecision Subtraction

For a 16-bit microcontroller, multiprecision subtraction is the subtraction of numbers that are larger than 16 bits. To subtract the hex number $16753284 from $98765432, the 68HC12 has to perform multiprecision subtraction:

```
  $98765432
–$16757284
```

Like multiprecision addition, multiprecision subtraction is performed one byte at a time, beginning with the least significant byte. The 68HC12 does allow us to subtract two bytes at a time because it has the SUBD instruction. The following two instructions can be used to subtract the least significant two bytes of the subtrahend from the minuend:

```
ldd   #$5432
subd  #$7284
```

Since a larger number is subtracted from a smaller one, there is a need to borrow from the higher byte, causing the C flag to be set to 1. The contents of double accumulator D should be saved before the higher bytes are subtracted. Let's save these two bytes at $802~$803:

```
std   $802
```

When the second-to-most-significant bytes are subtracted, the borrow 1 has to be subtracted from the second-to-most-significant byte of the result. In other words, we need a "subtract with borrow" instruction. There is such an instruction, but it is called *subtract with carry.* There are two versions: the SBCA instruction for accumulator A, and the SBCB instruction for accumulator B. The instructions to subtract the second-to-most-significant bytes are:

```
ldaa  #$76
sbca  #$75
```

We also need to save the second-to-most-significant byte of the result at $801 with the following instruction:

```
staa  $801
```

The most significant bytes can be subtracted using similar instructions, and the complete program with comments is as follows:

```
org   $1000     ; starting address of the program
ldd   #$5432    ; place the lower two bytes of the minuend in D
subd  #$7284    ; subtract the lower bytes of the subtrahend from D
std   $802      ; save the lower two bytes of the difference
ldaa  #$76      ; place the second-to-most-significant byte of the minuend in A
sbca  #$75      ; subtract the second-to-most-significant byte of the
                ; subtrahend and the borrow from A
staa  $801      ; save the second-to-most-significant byte of the difference
ldaa  #$98      ; put the most-significant-byte of the minuend in A
sbca  #$16      ; subtract the most-significant-byte of the
                ; subtrahend and the borrow from A
staa  $800      ; save the most-significant-byte of the difference
end
```

Example 2.8

Write a program to subtract the hex numbers stored at $804~$807 from the hex number stored at $800~$803, and save the difference at $900~$903.

Solution: We will perform the subtraction from the least significant byte towards the most significant byte as follows:

```
org     $1000     ; starting address of the program
ldd     $802      ; place the lowest two bytes of the minuend in D
subd    $806      ; subtract the lowest two bytes of the subtrahend from D
std     $902      ; save the lowest two bytes of the difference
ldaa    $801      ; put the second-to-most-significant byte of the minuend in A
sbca    $805      ; subtract the second-to-most-significant byte of the
                  ; subtrahend and the borrow from A
staa    $901      ; save the second-to-most-significant byte of the difference
ldaa    $800      ; put the most significant byte of the minuend in A
sbca    $804      ; subtract the most significant byte of the subtrahend
                  ; and the borrow from A
staa    $900      ; save the most significant byte of the difference
end
```

2.5.5 Binary-Coded-Decimal (BCD) Addition

Although virtually all computers work internally with binary numbers, the input and output equipment generally uses decimal numbers. Since most logic circuits only accept two-valued signals, the decimal numbers must be coded in terms of binary signals. In the simplest form of binary code, each decimal digit is represented by its binary equivalent. For example, 2,538 is represented by:

0010 0101 0011 1000

This representation is called *binary-coded-decimal*. If the BCD format is used, it must be preserved during arithmetic processing.

The principal advantage of the BCD encoding method is the simplicity of input/output conversion; its major disadvantage is the complexity of arithmetic processing. The choice between binary and BCD depends on the type of problems the system will be handling.

The 68HC12 microcontroller can add only binary numbers, not decimal numbers. The following instruction sequence appears to cause the 68HC12 to add the decimal numbers 25 and 31 and store the sum at the memory location $800:

```
ldaa   #$25
adda   #$31
staa   $800
```

This instruction sequence performs the following addition:

```
  $25
+$31
  $56
```

When the 68HC12 executes this instruction sequence, it adds the numbers according to the rules of binary addition and produces the sum $56. This is the correct BCD answer, because the

result represents the decimal sum of 25 + 31 = 56. In this example, the 68HC12 gives the appearance of performing decimal addition. However, a problem occurs when the 68HC12 adds two BCD digits and generates a sum greater than nine. Then the sum is incorrect in the decimal number system, as the following three examples illustrate:

```
 $18        $35        $19
+$47       +$47       +$47
 $5F        $7C        $60
```

The answers to the first two problems are obviously wrong in the decimal number system because the hex digits F and C are not between zero and nine. The answer to the third example appears to contain valid BCD digits, but in the decimal system 19 plus 47 equals 66, not 60; this example involves a carry from the lower nibble to the higher nibble.

In summary, a sum in the BCD format is incorrect if it is greater than $9 or if there is a carry to the next higher nibble. Incorrect BCD sums can be adjusted by adding $6 to them. To correct the examples, do the following:

1. Add $6 to every sum digit greater than nine.
2. Add $6 to every sum digit that had a carry of one to the next higher digit.

Here are the problems with their sums adjusted:

```
 $18        $35        $19
+$47       +$47       +$47
 $5F        $7C        $60
+$ 6       +$ 6       +$ 6
 $65        $82        $66
```

The fifth bit of the condition code register is the *half-carry,* or H flag. A carry from the lower nibble to the higher nibble during the addition operation is a half-carry. A half-carry of one during addition sets the H flag to one, and a half-carry of zero during addition clears it to zero. If there is a carry from the high nibble during addition, the C flag is set to one, which indicates that the high nibble is incorrect. A $6 must be added to the high nibble to adjust it to the correct BCD sum.

Fortunately, we don't need to write instructions to detect the illegal BCD sum following a BCD addition. The 68HC12 provides a *decimal adjust accumulator A* instruction, DAA, which takes care of all these detailed detection and correction operations. The DAA instruction monitors the sums of BCD additions and the C and H flags and automatically adds $6 to any nibble that requires it. The rules for using the DAA instruction are as follows:

1. The DAA instruction can only be used for BCD addition. It does not work for subtraction or hex arithmetic.
2. The DAA instruction must be used immediately after one of the three instructions that leave their sum in accumulator A. (These three instructions are ADDA, ADCA, ABA.)
3. The numbers added must be legal BCD numbers to begin with.

Example 2.9

Write an instruction sequence to add the BCD numbers stored at memory locations $800 and $801, and store the sum at $810.

Solution:

```
ldaa   $800     ; load the first BCD number in A
adda   $801     ; perform addition
daa             ; decimal adjust the sum in A
staa   $810     ; save the sum
```

2.5.6 Multiplication & Division

The 68HC12 provides three multiply and five divide instructions. A brief description of these instructions appears in Table 2.1.

Mnemonic	Function	Operation
EMUL	unsigned 16 by 16 multiply	(D) x (Y) → Y:D
EMULS	signed 16 by 16 multiply	(D) x (Y) → Y:D
MUL	unsigned 8 by 8 multiply	(A) x (B) → A:B
EDIV	unsigned 32 by 16 divide	(Y:D) ÷ (X) quotient → Y remainder → D
EDIVS	signed 32 by 16 divide	(Y:D) ÷ (X) quotient → Y remainder → D
FDIV	16 by 16 fractional divide	(D) ÷ (X) → X remainder → D
IDIV	unsigned 16 by 16 integer divide	(D) ÷ (X) → X remainder → D
IDIVS	signed 16 by 16 integer divide	(D) ÷ (X) → X remainder → D

Table 2.1 ■ Summary of 68HC12 multiply and divide instructions

The **EMUL** instruction multiplies the 16-bit unsigned integers stored in accumulator D and index register Y and leaves the product in these two registers. The upper 16 bits of the product are in Y whereas the lower 16 bits are in D.

The **EMULS** instruction multiplies the 16-bit signed integers stored in accumulator D and index register Y and leaves the product in these two registers. The upper 16 bits of the product are in Y whereas the lower 16 bits are in D.

The **MUL** instruction multiplies the 8-bit unsigned integer in accumulator A by the 8-bit unsigned integer in accumulator B to obtain a 16-bit unsigned result in double accumulator D. The upper byte of the product is in accumulator A whereas the lower byte of the product is in B.

The **EDIV** instruction performs an unsigned 32-bit by 16-bit division. The dividend is the register pair Y and D with Y as the upper 16-bit of the dividend. Index register X is the divisor. After division, the quotient and the remainder are placed in Y and D, respectively.

The **EDIVS** instruction performs a signed 32-bit by 16-bit division using the same operands as the EDIV instruction does. After division, the quotient and the remainder are placed in Y and D, respectively.

The **FDIV** instruction divides an unsigned 16-bit dividend in double accumulator D by an unsigned 16-bit divisor in index register X, producing an unsigned 16-bit quotient in X and an unsigned 16-bit remainder in D. The dividend must be less than the divisor. The radix point of the quotient is to the left of the bit 15. In the case of overflow (the denominator is less than or equal to the nominator) or division-by-zero, the quotient is set to $FFFF, and the remainder is indeterminate.

The **IDIV** instruction divides an unsigned 16-bit dividend in double accumulator D by the unsigned 16-bit divisor in index register X, producing an unsigned 16-bit quotient in X, and an unsigned 16-bit remainder in D. If both the divisor and the dividend are assumed to have radix points in the same positions (to the right of bit 0), the radix point of the quotient is to the right of bit 0. In the case of division by zero, the quotient is set to $FFFF, and the remainder is indeterminate.

The **IDIVS** instruction divides the signed 16-bit dividend in double accumulator D by the signed 16-bit divisor in index register X, producing a signed 16-bit quotient in X, and a signed 16-bit remainder in D. If division-by-zero is attempted, the values in D and X are not changed, but the values of the N, Z, and V status bits are undefined.

Example 2.10

Write an instruction sequence to multiply the contents of index register X and double accumulator D, and store the product at memory locations $800~$803.

Solution: There is no instruction to multiply the contents of double accumulator D and index register X. However, we can transfer the contents of index register X to index register Y and execute the EMUL instruction. If index register Y holds useful information, then we need to save it before the data transfer.

```
sty     $810     ; save Y in a temporary location
tfr     x,y      ; transfer the contents of X to Y
emul             ; perform the multiplication
sty     $800     ; save the upper 16 bits of the product
std     $802     ; save the lower 16 bits of the product
ldy     $810     ; restore the value of Y
```

Example 2.11

Write an instruction sequence to divide the signed 16-bit number stored at memory locations $805~$806 by the 16-bit unsigned number stored at memory locations $820~$821, and store the quotient and remainder at $900~$901 and $902~$903, respectively.

Solution: Before we can perform the division, we need to place the dividend and divisor in D and X, respectively.

```
ldd     $805     ; place the dividend in D
ldx     $820     ; place the divisor in X
idivs            ; perform the signed division
stx     $900     ; save the quotient
std     $902     ; save the remainder
```

Because most arithmetic operations can be performed only on accumulators, we need to transfer the contents of index register X to D so that further division on the quotient can be performed. The 68HC12 provides two exchange instructions in addition to the TFR instruction for this purpose:

- The XGDX instruction exchanges the contents of accumulator D and index register X.
- The XGDY instruction exchanges the contents of accumulator D and index register Y.

The 68HC12 provides instructions for performing unsigned 8-bit by 8-bit and both signed and unsigned 16-bit by 16-bit multiplications. Since the 68HC12 is a 16-bit microcontroller, we expect that it will be used to perform complicated operations in many sophisticated applications. Performing 32-bit by 32-bit multiplication will be one of them.

Since there is no 32-bit by 32-bit multiplication instructions, we will have to break a 32-bit number into two 16-bit halves and use the 16-bit by 16-bit multiply instruction to synthesize the operation. Assume M and N are the multiplicand and the multiplier, respectively. These two numbers can be broken down as follows:

$M = M_H M_L$
$N = N_H N_L$

where, M_H and N_H are the upper 16 bits and M_L and N_L are the lower 16 bits of M and N, respectively. Four 16-bit by 16-bit multiplications are performed, and then their partial products are added together as shown in Figure 2.3.

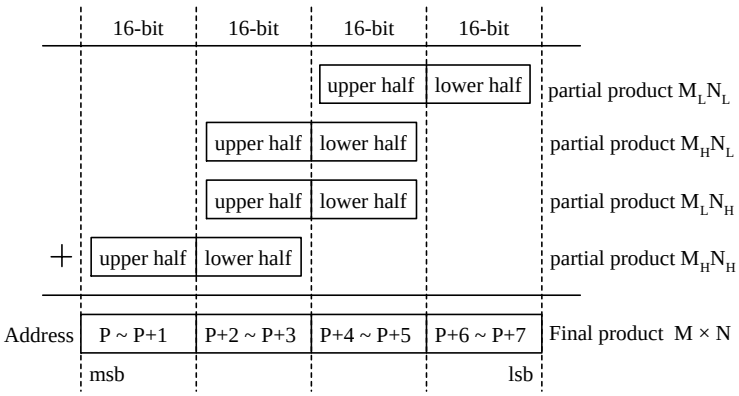


Figure 2.3 ■ Unsigned 32-bit by 32-bit multiplication

The procedure is as follows:

Step 1
Allocate eight bytes to hold the product. Assume these eight bytes are located at P, P+1, …, and P+7.

Step 2
Generate the partial product $M_L N_L$ (in Y:D) and save it at locations P+4 ~ P+7.

Step 3
Generate the partial product $M_H N_H$ (in Y:D) and save it at locations P ~ P+3.

Step 4
Generate the partial product M_HN_L (in Y:D) and add it to memory locations P+2 ~ P+5. The C flag may be set to 1 after this addition.

Step 5
Add the C flag to memory location P+1 using the ADCA (or ADCB) instruction. This addition may also set the C flag to 1. So, again, add the C flag to memory location P.

Step 6
Generate the partial product M_LN_H (in Y:D) and add it to memory locations P+2 ~ P+5. The carry flag may be set to 1. So add the C flag to memory location P+1 and then add it to memory location P.

Example 2.12

Write a program to multiply the 32-bit unsigned integers stored at M~M+3 and N~N+3, respectively and store the product at memory locations P~P+7.

Solution: The following program is a direct translation of the previous multiplication algorithm.

```
        org     $800
M       rmb     4       ; reserved to hold the multiplicand
N       rmb     4       ; reserved to hold the multiplier
P       rmb     8       ; reserved to hold the product
        org     $1000
        ldd     M+2     ; place ML in D
        ldy     N+2     ; place NL in Y
        emul            ; compute MLNL
        sty     P+4     ; save the upper 16 bits of the partial product MLNL
        std     P+6     ; save the lower 16 bits of the partial product MLNL
        ldd     M       ; place MH in D
        ldy     N       ; place NH in Y
        emul            ; compute MHNH
        sty     P       ; save the upper 16 bits of the partial product MHNH
        std     P+2     ; save the lower 16 bits of the partial product MHNH
        ldd     M       ; place MH in D
        ldy     N+2     ; place NL in Y
        emul            ; compute MHNL
; the following seven instructions add MHNL to memory locations P+2~P+5
        addd    P+4     ; add the lower half of MHNL to P+4~P+5
        std     P+4     ;           "
        tfr     y,d     ; transfer Y to D
        adcb    P+3
        stab    P+3
        adca    P+2
        staa    P+2
; the following six instructions propagate carry to the most significant byte
        ldaa    P+1
        adca    #0      ; add C flag to location P+1
        staa    P+1     ;           "
        ldaa    P
        adca    #0      ; add C flag to location P
        staa    P
```

```
; the following three instructions compute MLNH
        ldd     M+2     ; place ML in D
        ldy     N       ; place NH in Y
        emul            ; compute MLNH
; the following seven instructions add MLNH to memory locations P+2~P+5
        addd    P+4     ; add the lower half of MLNH to P+4~P+5
        std     P+4     ;           "
        tfr     y,d     ; transfer Y to D
        adcb    P+3
        stab    P+3
        adca    P+2
        staa    P+2
; the following six instructions propagate carry to the most significant byte
        ldaa    P+1
        adca    #0      ; add C flag to location P+1
        staa    P+1
        ldaa    P
        adca    #0      ; add C flag to location P
        staa    P
        end
```

Example 2.13

Write a program to convert the 16-bit binary number stored at $800~$801 to BCD format and store the result at $900~$904. Convert each BCD digit into its ASCII code and store it in one byte.

Solution: A binary number can be converted to BCD format using repeated division-by-10. The largest 16-bit binary number corresponds to the 5-digit decimal number 65535. The first division by 10 computes the least significant digit and should be stored in the memory location $904, the second division-by-10 operation computes the ten's digit, and so on. The ASCII code of a BCD digit can be obtained by adding $30 to each BCD digit. The program is as follows:

```
        org     $800
data    fdb     12345   ; place a number for testing
        org     $900
result  rmb     5       ; reserve five bytes to store the result
        org     $1000
        ldd     data    ; make a copy of the number to be converted
        ldy     #result
        ldx     #10     ; divide the number by 10
        idiv            ;           "
        addb    #$30    ; convert to ASCII code
        stab    4,Y     ; save the least significant digit
        xgdx            ; swap the quotient to D
        ldx     #10
        idiv
        addb    #$30    ; convert to ASCII code
        stab    3,Y     ; save the second-to-least-significant digit
        xgdx
        ldx     #10
```

```
idiv
addb    #$30
stab    2,Y       ; save the middle digit
xgdx
ldx     #10
idiv              ; separate the most-significant and second-to-most-
                  ; significant digits
addb    #$30
stab    1,Y       ; save the second-to-most-significant digit
xgdx              ; swap the most significant digit to B
addb    #$30      ; convert to ASCII code
stab    0,Y       ; save the most significant digit
end
```

2.6 Program Loops

Many applications require repetitive operations. We can write programs to tell computers to perform the same operation over and over. A *finite loop* is a sequence of instructions that will be executed by the computer for a finite number of times, while an *endless loop* is a sequence of instructions that the computer will execute forever.

There are four major loop constructs:

Do statement S forever

This is an endless loop in which statement S is repeated forever. In some applications, we might add the statement "If C then exit" to leave the infinite loop. An infinite loop is shown in Figure 2.4.

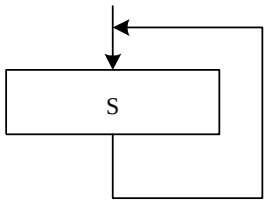


Figure 2.4 ■ An infinite loop

For *i = n1* ***to*** *n2* ***do*** *S or* ***For*** *i = n2* ***to*** *n1* ***do*** *S*

Here, the variable *i* is the *loop counter,* which keeps track of the number of remaining times statement S is to be executed. The loop counter can be incremented (the first case) or decremented (the second case). Statement S is repeated ***n2 – n1 + 1*** times. The value of n2 is assumed to be no smaller than n1. If there is concern that the relationship $n1 \leq n2$ may not hold, then it must be checked at the beginning of the loop. Four steps are required to implement a FOR loop:

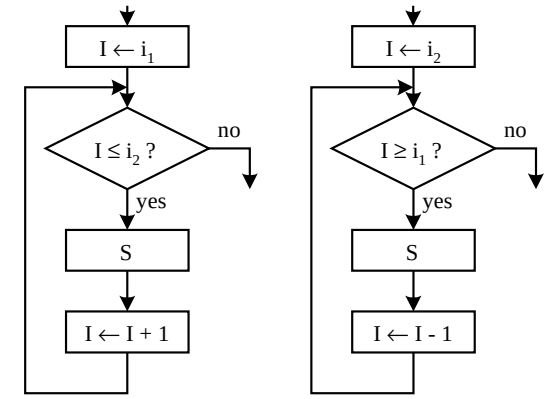
Step 1
Initialize the loop counter and other variables.

Step 2
Compare the loop counter with the limit to see if it is within bounds. If it is, then perform the specified operations. Otherwise, exit the loop.

Step 3
Increment (or decrement) the loop counter.

Step 4
Go to step 2.

A ***For-loop*** is illustrated in Figure 2.5.



(a) For I = i_1 to i_2 DO S (b) For I = i_2 downto i_1 DO S

Figure 2.5 ■ For looping construct

While C Do S

Whenever a **While** construct is executed, the logical expression C is evaluated first. If it yields a false value, statement S will not be executed. The action of a While construct is illustrated in Figure 2.6. Four steps are required to implement a While loop:

Step 1
Initialize the logical expression C.

Step 2
Evaluate the logical expression C.

Step 3
Perform the specified operations if the logical expression C evaluates to true. Update the logical expression C and go to step 2. (Note: The logical expression C may be updated by external conditions or by an interrupt service routine.)

Step 4
Exit the loop.

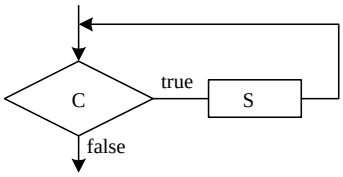


Figure 2.6 ■ The While ... Do looping construct

Repeat S Until C

Statement ***S*** is first executed then the logical expression ***C*** is evaluated. If C is false, the next statement will be executed. Otherwise, statement S will be executed again. The action of this construct is illustrated in Figure 2.7. Statement S will be executed at least once. Three steps are required to implement this construct:

Step 1
Initialize the logical expression C.

Step 2
Execute statement S.

Step 3
Go to Step 2 if the logical expression C evaluates to true. Otherwise, exit.

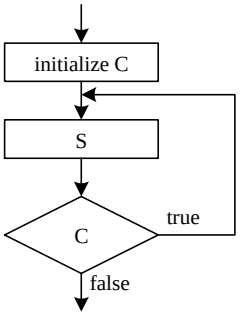


Figure 2.7 ■ The Repeat ... Until looping construct

To implement one of the looping constructs, we need to use unconditional branch or one of the conditional instructions. When executing conditional branch instructions, the 68HC12 checks the condition flags in the CCR register.

2.6.1 Condition Code Register

The contents of the condition code register are shown in Figure 2.8. The shaded characters are condition flags that reflect the status of an operation. The meanings of these condition flags are as follows:

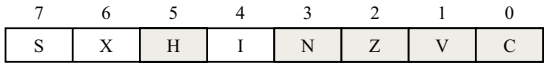


Figure 2.8 ■ Condition code register

- ***C: the carry flag***
  Whenever a carry is generated as the result of an operation, this flag will be set to 1. Otherwise, it will be cleared to 0.
- ***V: the overflow flag***
  Whenever the result of a two's complement arithmetic operation is out of range, this flag will be set to 1. Otherwise, it will be set to 0. The V flag is set to 1 when the carry from the most significant bit and the second most significant bit differ as the result of an arithmetic operation.
- ***Z: the zero flag***
  Whenever the result of an operation is zero, this flag will be set to 1. Otherwise, it will be set to 0.
- ***N: the negative flag***
  Whenever the most significant bit of the result of an operation is 1, this flag will be set to 1. Otherwise, it will be set to 0. This flag indicates that the result of an operation is negative.
- ***H: the half-carry flag***
  Whenever there is a carry from the lower four bits to the upper four bits as the result of an operation, this flag will be set to 1. Otherwise, it will be set to 0.

2.6.2 Branch Instructions

Branch instructions cause program flow to change when specific conditions exist. The 68HC12 has three kinds of branch instructions including *short branches*, *long branches*, and *bit-conditional branches*.

Branch instructions can also be classified by the type of condition that must be satisfied in order for a branch to be taken. Some instructions belong to more than one category.

- ***Unary (unconditional) branch*** instructions always execute.
- ***Simple branches*** are taken when a specific bit in the CCR register is in a specific state as a result of a previous operation.
- ***Unsigned branches*** are taken when a comparison or test of unsigned quantities results in a specific combination of condition code register bits.
- ***Signed branches*** are taken when a comparison or test of signed quantities results in a specific combination of condition code register bits.

When a short-branch instruction is executed, a signed 8-bit offset is added to the value in the program counter when a specified condition is met. Program execution continues at the new address. The numeric range of the short branch offset value is $80 (-128) to $7F (127) from the address of the instruction immediately following the branch instruction. A summary of the short branch instructions is shown in Table 2.2.

When a long-branch instruction is executed, a signed 16-bit offset is added to the value in the program counter when a specified condition is met. Program execution continues at the new address. Long branch instructions are used when large displacements between decision-making steps are necessary.

Unary Branches		
Mnemonic	**Function**	**Equation or Operation**
BRA	Branch always	1 = 1
BRN	Branch never	1 = 0
Simple Branches		
Mnemonic	**Function**	**Equation or Operation**
BCC	Branch if carry clear	C = 0
BCS	Branch if carry set	C = 1
BEQ	Branch if equal	Z = 1
BMI	Branch if minus	N = 1
BNE	Branch if not equal	Z = 0
BPL	Branch if plus	N = 0
BVC	Branch if overflow clear	V = 0
BVS	Branch if overflow set	V = 1
Unsigned Branches		
Mnemonic	**Function**	**Equation or Operation**
BHI	Branch if higher	C + Z = 0
BHS	Branch if higher or same	C = 0
BLO	Branch if lower	C = 1
BLS	Branch if lower or same	C + Z = 1
Signed Branches		
Mnemonic	**Function**	**Equation or Operation**
BGE	Branch if greater than or equal	N ⊕ V = 0
BGT	Branch if greater than	Z + (N ⊕ V) = 0
BLE	Branch if less than or equal	Z + (N ⊕ V) = 1
BLT	Branch if less than	N ⊕ V = 1

Table 2.2 ■ Summary of short branch instructions

The numeric range of long-branch offset values is $8000 (-32768) to $7FFF (32767) from the instruction immediately after the branch instruction. This permits branching from any location in the standard 64-KB address map to any other location in the map. A summary of the long-branch instructions appears in Table 2.3.

Unary Branches		
Mnemonic	**Function**	**Equation or Operation**
LBRA	Long branch always	1 = 1
LBRN	Long branch never	1 = 0
Simple Branches		
Mnemonic	**Function**	**Equation or Operation**
LBCC	Long branch if carry clear	C = 0
LBCS	Long branch if carry set	C = 1
LBEQ	Long branch if equal	Z = 1
LBMI	Long branch if minus	N = 1
LBNE	Long branch if not equal	Z = 0
LBPL	Long branch if plus	N = 0
LBVC	Long branch if overflow is clear	V = 0
LBVS	Long branch if overflow set	V = 1
Unsigned Branches		
Mnemonic	**Function**	**Equation or Operation**
LBHI	Long branch if higher	C + Z = 0
LBHS	Long branch if higher or same	C = 0
LBLO	Long branch if lower	C = 1
LBLS	Long branch if lower or same	C + Z = 1
Signed Branches		
Mnemonic	**Function**	**Equation or Operation**
LBGE	Long branch if greater than or equal	N ⊕ V = 0
LBGT	Long branch if greater than	Z + (N ⊕ V) = 0
LBLE	Long branch if less than or equal	Z + (N ⊕ V) = 1
LBLT	Long branch if less than	N ⊕ V = 1

Table 2.3 ■ Summary of long branch instructions

Although there are many possibilities in writing a program loop, the following one is a common format:

```
loop:  .
       .
       .
       Bcc (or LBcc)  loop
```

Where **cc** is one of the condition codes (CC, CS, EQ, MI, NE, PL, VC, VS, HI, HS, LO, LS, GE, GT, LS, and LT).

Usually there will be a comparison or arithmetic instruction to set up the condition code for use by the conditional branch instruction.

2.6.3 Compare & Test Instructions

The 68HC12 has a set of compare instructions that are dedicated to the setting of condition flags. The compare and test instructions perform subtraction between a pair of registers or between a register and a memory location. The result is not stored, but condition codes are set by the operation. In the 68HC12, most instructions update condition code flags automatically, so it is often unnecessary to include a separate test or compare instruction. Table 2.4 is a summary of compare and test instructions.

Compare instructions		
Mnemonic	**Function**	**Operation**
CBA	Compare A to B	(A) - (B)
CMPA	Compare A to memory	(A) - (M)
CMPB	Compare B to memory	(B) - (M)
CPD	Compare D to memory	(D) - (M:M+1)
CPS	Compare SP to memory	(SP) - (M:M+1)
CPX	Compare X to memory	(X) - (M:M+1)
CPY	Compare Y to memory	(Y) - (M:M+1)
Test instructions		
Mnemonic	**Function**	**Operation**
TST	Test memory for zero or minus	(M) - $00
TSTA	Test A for zero or minus	(A) - $00
TSTB	Test B for zero or minus	(B) - $00

Table 2.4 ■ Summary of compare and test instructions

2.6.4 Loop Primitive Instructions

A lot of the program loops are implemented by incrementing or decrementing a loop count. The branch is taken when either the loop count is equal to zero or not equal to zero depending on the applications. The 68HC12 provides a set of loop primitive instructions for implementing this type of looping mechanism. These instructions test a counter value in a register or accumulator (A, B, D, X, Y, or SP) for zero or nonzero values as a branch condition. There are predecrement, preincrement, and test-only versions of these instructions.

The range of the branch is from $80 (-128) to $7F (127) from the instruction immediately following the loop primitive instruction. Table 2.5 shows a summary of the loop primitive instructions.

Mnemonic	Function	Equation or Operation
DBEQ cntr, rel	Decrement counter and branch if = 0 (counter = A, B, D, X, Y, or SP)	counter ← (counter) - 1 If (counter) = 0, then branch else continue to next instruction
DBNE cntr, rel	Decrement counter and branch if ≠ 0 (counter = A, B, D, X, Y, or SP)	counter ← (counter) - 1 If (counter) ≠ 0, then branch else continue to next instruction
IBEQ cntr, rel	Increment counter and branch if = 0 (counter = A, B, D, X, Y, or SP)	counter ← (counter) + 1 If (counter) = 0, then branch else continue to next instruction
IBNE cntr, rel	Increment counter and branch if ≠ 0 (counter = A, B, D, X, Y, or SP)	counter ← (counter) + 1 If (counter) ≠ 0, then branch else continue to next instruction
TBEQ cntr, rel	Test counter and branch if = 0 (counter = A, B, D, X, Y, or SP)	If (counter) = 0, then branch else continue to next instruction
TBNE cntr, rel	Test counter and branch if ≠ 0 (counter = A, B, D, X, Y, or SP)	If (counter) ≠ 0, then branch else continue to next instruction

Note. 1. **cntr** is the loop counter and can be accumulator A, B, or D and register X, Y, or SP.
2. **rel** is the relative branch offset and is usually a label

Table 2.5 ■ Summary of loop primitive instructions

Example 2.14

Write a program to add an array of N 8-bit numbers and store the sum at memory location $800~$801. Use the ***For*** *i* = *n1* ***to*** *n2* ***do*** looping construct.

Solution: We will use variable **i** as the array index. This variable can also be used to keep track of the number of iterations remained to be performed. We will use a two-byte variable **sum** to hold the sum of array elements. The logic flow of the program is illustrated in Figure 2.9.

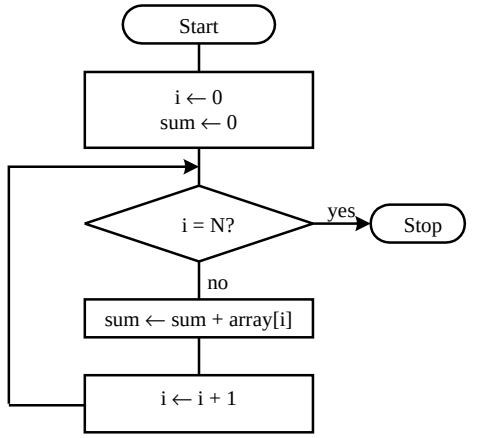


Figure 2.9 ■ Logic flow of example 2.14

The program is a direct translation of the flowchart shown in Figure 2.9.

```
N       equ     20                      ; array count
        org     $800                    ; starting address of on-chip SRAM
sum     rmb     2                       ; array sum
i       rmb     1                       ; array index
        org     $1000                   ; starting address of the program
        ldaa    #0
        staa    i                       ; initialize loop (array) index to 0
        staa    sum                     ; initialize sum to 0
        staa    sum+1                   ;     "

loop    ldab    i
        cmpb                    #N      ; is i = N?
        beq     done                    ; if done, then branch
        ldx     #array                  ; use index register X as a pointer to the array
        abx                             ; compute the address of array[i]
        ldab    0,x                     ; place array[i] in B
        dy      sum                     ; place sum in Y
        aby                             ; compute sum <- sum + array[i]
        sty     sum                     ; update sum
        inc     i                       ; increment the loop count by 1
        bra     loop
done    swi                             ; return to D-Bug12 monitor
; the array is defined in the following statement
array   db      1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20
        end
```

It is a common mistake for an assembly language programmer to forget to update the variable in memory. For example, we will not get the correct value for **sum** if we did not add the instruction **sty sum** in the program shown in Example 2.14.

Loop primitive instructions are especially suitable for implementing the **repeat** S **until** C looping construct as demonstrated in the following example.

Example 2.15

Write a program to find the maximum element from an array of N 8-bit elements using the ***repeat S until C*** looping construct.

Solution: We will use the variable **i** as the array index and also as the loop count. The variable **max_val** will be used to hold the array maximum. The logic flow of the program is shown in Figure 2.10.

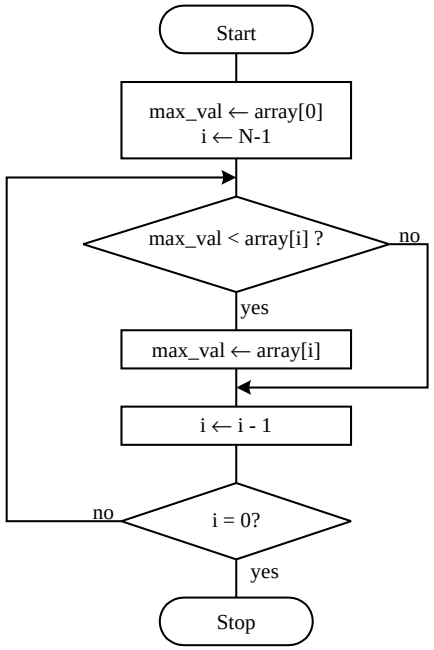


Figure 2.10 ■ Logic flow of example 2.15

The program is as follows:

```
N         equ     20              ; array count
          org     $800            ; starting address of on-chip SRAM
max_val   rmb     1               ; memory location to hold array max
          org     $1000           ; starting address of program
          ldaa    array           ; set array[0] as the temporary array max
          staa    max_val         ;      "
          ldx     #array+N-1      ; start from the end of the array
          ldab    #N-1            ; use B to hold variable i and initialize it to N-1
loop      ldaa    max_val
          cmpa    0,x             ; compare max_val with array[i]
          bge     chk_end         ; no update if max_val is larger
          ldaa    0,x             ; update max_val
          staa    max_val         ;      "
chk_end   dex                     ; move the array pointer
          dbne    b,loop          ; decrement the loop count, branch if not zero yet.
forever   bra     forever
array     db      1,3,5,6,19,41,53,28,13,42,76,14,20,54,64,74,29,33,41,45
          end
```

2.6.5 Decrementing & Incrementing Instructions

We often need to add or subtract one from a variable in our program. Although we can use one of the ADD or SUB instructions to achieve this, it would be more efficient to use a single instruction. The 68HC12 has a few instructions for us to increment or decrement a variable by one. A summary of decrement and increment instructions is listed in Table 2.6.

Decrement instructions		
Mnemonic	**Function**	**Operation**
DEC	Decrement memory by 1	M ← (M) - $01
DECA	Decrement A by 1	A ← (A) - $01
DECB	Decrement B by 1	B ← (B) - $01
DES	Decrement SP by 1	SP ← (SP) - $01
DEX	Decrement X by 1	X ← (X) - $01
DEY	Decrement Y by 1	Y ← (Y) - $01
Increment instructions		
Mnemonic	**Function**	**Operation**
INC	Increment memory by 1	M ← (M) + $01
INCA	Increment A by 1	A ← (A) + $01
INCB	Increment B by 1	B ← (B) + $01
INS	Increment SP by 1	SP ← (SP) + $01
INX	Increment X by 1	X ← (X) + $01
INY	Increment Y by 1	Y ← (Y) + $01

Table 2.6 ■ Summary of compare and test instructions

Example 2.16

Use an appropriate increment or decrement instruction to replace the following instruction sequence:

```
ldaa    i
adda    #1
staa    i
```

Solution: The above three instructions can be replaced by the following instruction:

```
inc     I
```

2.6.6 Bit Condition Branch Instructions

In certain applications, we need to make branch decisions based on the value of a few bits. The 68HC12 provides two special conditional branch instructions for this purpose. The syntax of the first special conditional branch instruction is:

```
[<label>]    BRCLR opr, msk, rel
```

where:

opr specifies the memory location to be checked and can be specified using direct, extended, and all indexed addressing modes.

msk is an 8-bit mask that specifies the bits of the memory location to be checked. The bits to be checked correspond to those bit positions that are ones in the mask.

rel is the branch offset and is specified in 8-bit relative mode.

This instruction tells the 68HC12 to perform bitwise logical AND on the contents of the specified memory location and the mask supplied with the instruction, then branch if the result is zero.

For example, for the instruction sequence:

```
here    brclr    $66,$80,here
        ldd      $70
```

the 68HC12 will continue to execute the first instruction if the most significant bit of the memory location at $66 is 0. Otherwise, the next instruction will be executed.

The syntax of the second special conditional branch instruction is:

```
[<label>]        BRSET opr, msk, rel
```

where:

opr specifies the memory location to be checked and can be specified using direct, extended, and all indexed addressing modes.

msk is an 8-bit mask that specifies the bits of the memory location to be checked. The bits to be checked correspond to those bit positions that are ones in the mask

rel is the branch offset and is specified in 8-bit relative mode

This instruction tells the 68HC12 to perform the logical AND of the contents of the specified memory location inverted and the mask supplied with the instruction, then branch if the result is zero (this occurs only when all bits corresponding to ones in the mask byte are ones in the tested byte).

For example, for the following instruction sequence:

```
loop    inc      count
        ...
        brset    $66,$e0,loop
        ...
```

the branch will be taken if the most significant three bits of the memory location at $66 are all ones.

Example 2.17

Write a program to count the number of elements that are divisible by 4 in an array of N 8-bit numbers. Use the **repeat S until C** looping construct.

Solution: The lowest two bits of a number divisible by four are 00. By checking the lowest two bits of a number, we can determine if a number is divisible by four. The program is as follows:

```
N       equ     20
        org     $800
total   rmb     1
        org     $1000          ; starting address of the program
        ldaa    #0
        staa    total          ; initialize total to 0
```

```
        ldx     #array          ; use index register X as the array pointer
        ldab    #N              ; use accumulator B as the loop count
loop    brclr   0,x,$03,yes
        bra     chkend
yes     inc     total           ; add 1 to the total
chkend  inx                     ; move the array pointer
        dbne    b,loop
forever bra     forever
array   db      2,3,4,8,12,13,19,24,33,32,20,18,53,52,80,82,90,94,100,102
        end
```

2.6.7 Instructions for Variable Initialization

We often need to initialize a variable to zero when writing a program. The 68HC12 has three instructions for this purpose. They are:

```
[<label>]       clr     opr
```

where **opr** is a memory location specified using the extended mode and all indexed addressing (direct and indirect) modes. The memory location is initialized to zero by this instruction.

```
[<label>]       clra
```

Accumulator A is cleared to 0 by this instruction.

```
[<label>]       clrb
```

Accumulator B is cleared to 0 by this instruction.

2.7 Shift & Rotate Instructions

Shift and *rotate instructions* are useful for bit field manipulation. They can be used to speed up the integer multiply and divide operations if one of the operands is a power of two. A shift/rotate instruction shifts/rotates the operand by one bit. The 68HC12 has shift instructions that can operate on accumulators A, B, and D, or on a memory location. A memory operand must be specified using the extended or indexed (direct or indirect) addressing modes. A summary of shift and rotate instructions is shown in Table 2.7.

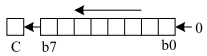
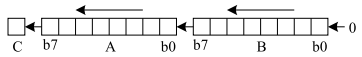
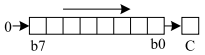
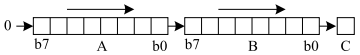
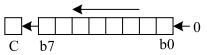
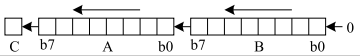
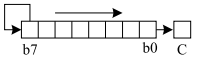
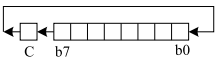
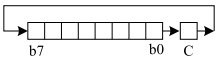
Logical shift instructions		
Mnemonic	**Function**	**Operation**
LSL <opr>	Logical shift left memory	
LSLA	Logical shift left A	
LSLB	Logical shift left B	
LSLD	Logical shift left D	
LSR <opr>	Logical shift right memory	
LSRA	Logical shift right A	
LSRB	Logical shift right B	
LSRD	Logical shift right D	
Arithmetic shift instructions		
Mnemonic	**Function**	**Operation**
ASL <opr>	Arithmetic shift left memory	
ASLA	Arithmetic shift left A	
ASLB	Arithmetic shift left B	
ASLD	Arithmetic shift left D	
ASR <opr>	Arithmetic shift right memory	
ASRA	Arithmetic shift right A	
ASRB	Arithmetic shift right B	
Rotate instructions		
Mnemonic	**Function**	**Operation**
ROL <opr>	Rotate left memory thru carry	
ROLA	Rotate left A through carry	
ROLB	Rotate left B through carry	
ROR <opr>	Rotate right memory thru carry	
RORA	Rotate right A through carry	
RORB	Rotate right B through carry	

Table 2.7 ■ Summary of shift and rotate instructions

Example 2.18

What are the values of accumulator A and the C flag after executing the ASLA instruction? Assume that originally A contains $95 and the C flag is 1.

Solution: The operation of this instruction is shown in Figure 2.11a.

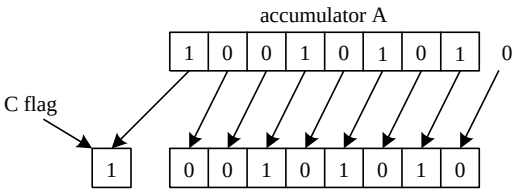


Figure 2.11a ■ Operation of the ASLA instruction

The result is shown in Figure 2.11b.

Original value	New value
[A] = 10010101 C = 1	[A] = 00101010 C = 1

Figure 2.11b ■ Execution result of the ASLA instruction

Example 2.19

What are the new values of the memory location at $800 and the C flag after executing the instruction ASR $800? Assume that the memory location $800 originally contains the value of $ED and the C flag is 0.

Solution: The operation of this instruction is shown in Figure 2.12a.

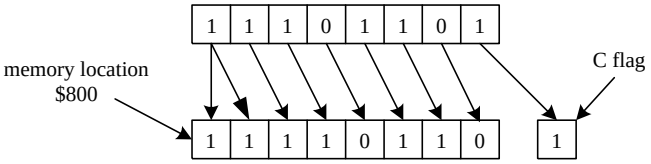


Figure 2.12a ■ Operation of the ASR $800 instruction

The result is shown in Figure 2.12b.

Original value	New value
[$800] = 11101101 C = 0	[$800] = 11110110 C = 1

Figure 2.12b ■ Result of the ASR $800 instruction

Example 2.20

What are the new values of the memory location at $800 and the C flag after executing the instruction LSR $800? Assume the memory location $800 originally contains $E7 and the C flag is 1.

Solution: The operation of this instruction is illustrated in Figure 2.13a.

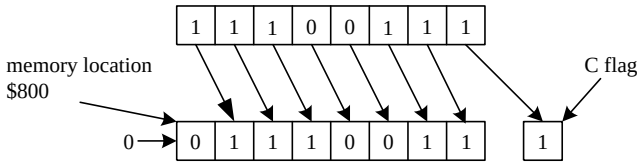


Figure 2.13a ■ Operation of the LSR $800 instruction

The result is shown in Figure 2.13b.

Original value	New value
[$800] = 11100111 C = 1	[$800] = 01110011 C = 1

Figure 2.13b ■ Execution result of LSR $800

Example 2.21

Compute the new values of accumulator B and the C flag after executing the instruction ROLB. Assume the original value of B is $BD and C flag is 1.

Solution: The operation of this instruction is illustrated in Figure 2.14a.

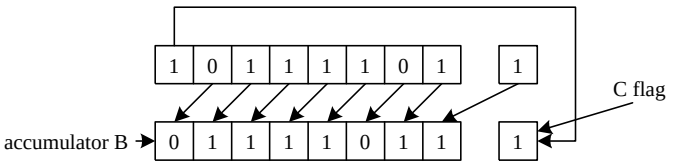


Figure 2.14a ■ Operation of the instruction ROLB

The result is shown in Figure 2.14b.

Original value	New value
[B] = 10111101 C = 1	[B] = 01111011 C = 1

Figure 2.14b ■ Execution result of ROLB

Example 2.22

What are the values of accumulator A and the C flag after executing the instruction RORA? Assume the original value of A is $BE and C = 1.

Solution: The operation of this instruction is illustrated in Figure 2.15a.

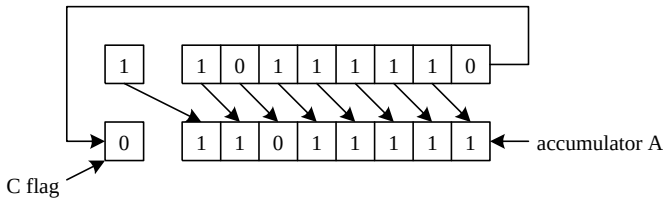


Figure 2.15a ■ Operation of the instruction RORA

The result is shown in Figure 2.15b.:

Original value	New value
[A] = 10111110 C = 1	[A] = 11011111 C = 0

Figure 2.15b ■ Execution result of RORA

Example 2.23

Write a program to count the number of zeros contained in memory locations $800~$801 and save the result at memory location $805.

Solution: The logical shift right instruction is available for double accumulator D. We can load this 16-bit value into D and shift it to the right sixteen times or until it becomes zero. The algorithm of this program is as follows:

Step 1
Initialize the loop count to 16 and the zero count to 0.

Step 2
Place the 16-bit value in D.

Step 3
Shift D to the right one place.

Step 4
If the C flag is 0, increment zero count by 1.
Step 5
Decrement loop count by 1.

Step 6
If loop count is zero, then stop. Otherwise, go to step 3.

The program is as follows:

```
            org         $800
            db          $23,$55
            org         $805
zero_cnt    rmb         1
lp_cnt      rmb         1
            org         $1000
            clr         zero_cnt        ; initialize the zero count to 0
            ldaa        #16
            staa        lp_cnt          ; initialize loop count to 16
            ldd         $800            ; place the 16-bit number in D
again       lsrd
            bcs         chk_end         ; branch if the lsb is a 1
            inc         zero_cnt
chk_end     dec         lp_cnt
            bne         again           ; have we tested all 16 bits yet?
forever     bra         forever
            end
```

Sometimes we need to shift a number larger than 16 bits. However, the 68HC12 does not have an instruction that does this. Suppose the number has k bytes and the most significant byte is located at ***loc***. The remaining k –1 bytes are located at loc+1, loc+2, …, loc+k-1, as shown in Figure 2.16.

Figure 2.16 ■ k bytes to be shifted

The logical shift-one-bit-to-the-right operation is shown in Figure 2.17.

Figure 2.17 ■ Shift-one-to-the-right operation

As shown in Figure 2.17:

- The bit seven of each byte will receive the bit zero of the byte on its immediate left with the exception of the most significant byte, which will receive a zero.
- Each byte will be shifted to the right by one bit. The bit zero of the least significant byte will be shifted out and lost.

The operation can therefore be implemented as follows:

Step 1
Shift the byte at *loc* to the right one place (using the LSR <opr> instruction).

Step 2
Rotate the byte at *loc+1* to the right one place (using the ROR <opr> instruction).

Step 3
Repeat step 2 for the remaining bytes.

By repeating this procedure, the given k-byte number can be shifted to the right as many bits as desired. The operation to shift a multi-byte number to the left should start from the least significant byte and rotate the remaining bytes toward the most significant byte.

Example 2.24

Write a program to shift the 32-bit number stored at $800~$803 to the right four places.

Solution: The most significant to the least significant bytes are stored at $800~$803. The following instruction sequence implements the algorithm that we just described:

```
        ldab    #4          ; set up the loop count
        ldx     #$800
again   lsr     0,x
        ror     1,x
        ror     2,x
        ror     3,x
        dbne    b,again
```

2.8 Boolean Logic Instructions

When dealing with input and output port pins, we often need to change the values of a few bits. For these types of applications, Boolean logic instructions come in handy. A summary of the 68HC12 Boolean logic instructions is described in Table 2.8.

Mnemonic	Function	Operation
ANDA <opr>	AND A with memory	A ← (A) • (M)
ANDB <opr>	AND B with memory	B ← (B) • (M)
ANDCC <opr>	AND CCR with memory (clear CCR bits)	CCR ← (CCR) • (M)
EORA <opr>	Exclusive OR A with memroy	A ← (A) ⊕ (M)
EORB <opr>	Exclusive OR B with memory	B ← (B) ⊕ (M)
ORAA <opr>	OR A with memory	A ← (A) + (M)
ORAB <opr>	OR B with memory	B ← (B) + (M)
ORCC <opr>	OR CCR with memory	CCR ← (CCR) + (M)
CLC	Clear C bit in CCR	C ← 0
CLI	Clear I bit in CCR	I ← 0
CLV	Clear V bit in CCR	V ← 0
COM <opr>	One's complement memory	M ← $FF - (M)
COMA	One's complement A	A ← $FF - (A)
COMB	One's complement B	B ← $FF - (B)
NEG <opr>	Two's complement memory	M ← $00 - (M)
NEGA	Two's complement A	A ← $00 - (A)
NEGB	Two's complement B	B ← $00 - (B)

Table 2.8 ■ Summary of Boolean logic instructions

The operand **opr** can be specified using all except the relative addressing modes. Usually, one would use the AND instruction to clear one or a few bits and use the OR instruction to set one or a few bits. The exclusive OR instruction can be used to toggle (change from 0 to 1 and from 1 to 0) one or a few bits.

For example, the instruction sequence:

```
ldaa    $56
anda    #$0F
staa    $56
```

clears the upper four pins of the I/O port located at $56.

The instruction sequence:

```
ldaa    $56
oraa    #$01
staa    $56
```

sets the bit 0 of the I/O port at $56.

The instructions sequence:

```
ldaa    $56
eora    #$0F
staa    $56
```

toggles the lower four bits of the I/O port at $56. The instructions (COMA and COMB) that perform one's complementing can be used if all of the port pins need to be toggled.

2.9 Bit Test & Manipulate Instruction

These instructions use a mask value to test or change the value of individual bits in an accumulator or in a memory location. BITA and BITB provide a convenient means of testing bits without altering the value of either operand. Table 2.9 shows a summary of bit test and manipulation instructions.

Mnemonic	Function	Operation
BCLR <opr>[2], msk8	Clear bits in memory	M ← (M) • ($\overline{mm}$)
BITA <opr>[1]	Bit test A	(A) • (M)
BITB <opr>[1]	Bit test B	(B) • (M)
BSET <opr>[2], msk8	Set bits in memory	M ← (M) + (mm)

Note. 1. <opr> can be specified using all except relative addressing modes for BITA and BITB.
2. <opr> can be specified using direct, extended, and indexed (exclude indiriect) addressing modes.
3. msk8 is an 8-bit value.

Table 2.9 ■ Summary of Boolean logic instructions

For example, the instruction:

```
bclr    0,x,$81
```

clears the most significant and least significant bits of the memory location pointed to by index register X.

The instruction:

```
bita    #$44
```

tests the bit six and bit two of accumulator A and updates Z and N flags of CCR register accordingly. The V flag in CCR register is cleared.

The instruction:

```
bitb    #$22
```

tests the bit five and bit one of accumulator B and updates the Z and N flags of CCR register accordingly. The V flag in CCR register is cleared.

Finally, the instruction:

```
bset    0,y,$33
```

sets the bits five, four, one, and zero of the memory location pointed to by index register Y.

2.10 Program Execution Time

The 68HC12 uses the ECLK (we will call it *E clock* from now on) signal as a timing reference. The frequency of the E clock is equal to one-half of the frequency of the crystal oscillator out of reset. The execution times of instructions are also measured in E cycles.

There are many applications that require the generation of time delays. Program loops are often used to create some amount of delay unless the time delay needs to be very accurate.

The creation of a time delay involves two steps:

1. Select a sequence of instructions that takes a certain amount of time to execute.
2. Repeat the instruction sequence for the appropriate number of times.

For example, the following instruction sequence takes 40 E clock cycles to execute:

```
loop    psha                    ; 2 E cycles
        pula                    ; 3 E cycles
        psha
        pula
        psha
        pula
        psha
        pula
        psha
        pula
        psha
        pula
        psha
        pula
        nop                     ; 1 E cycle
        nop                     ; 1 E cycle
        dbne    x,loop          ; 3 E cycles
```

If the 68HC12 runs under the control of a 16-MHz crystal oscillator, then the frequency and the period of the E clock signal are 8 MHz and 125 ns, respectively. The above instruction sequence will take 5 μs to execute.

Example 2.25

Write an instruction sequence to create a 100-ms time delay.

Solution: In order to create a 100-ms time delay, we need to repeat the above instruction sequence 20,000 times (100 ms ÷ 5 μs = 20,000). The following instruction sequence will create the desired delay:

```
            ldx     #20000      ; 2 E cycles
loop        psha                ; 2 E cycles
            pula                ; 3 E cycles
            psha                ; 2 E cycles
            pula                ; 3 E cycles
            psha                ; 2 E cycles
            pula                ; 3 E cycles
            psha                ; 2 E cycles
            pula                ; 3 E cycles
            psha                ; 2 E cycles
            pula                ; 3 E cycles
            psha                ; 2 E cycles
            pula                ; 3 E cycles
            psha                ; 2 E cycles
            pula                ; 3 E cycles
            psha                ; 2 E cycles
            pula                ; 3 E cycles
            nop                 ; 1 E cycle
            nop                 ; 1 E cycle
            dbne    x,loop      ; 3 E cycles
```

Example 2.26

Write an instruction sequence to create a delay of 10 seconds.

Solution: The instruction sequence in Example 2.25 can create no more than 327 ms delay. In order to create a longer time delay, we need to use a two-layer loop. For example, the following instruction sequence will create a 10-second delay:

```
            ldab    #100        ; 1 E cycle
out_loop    ldx     #20000      ; 2 E cycles
inner_loop  psha
            pula
            psha
            pula
            psha
            pula
            psha
            pula
            psha
            pula
            psha
            pula
            psha
            pula
            psha
            pula
```

```
        psha
        pula
        psha
        pula
        nop
        nop
        dbne    x,inner_loop
        dbne    b,out_loop      ; 3 E cycles
```

The time delay created by using program loops is not accurate. Some overhead is required to set up the loop count. For example, the one-layer loop has a 2-E-cycle overhead while the two-layer loop has much more overhead:

overhead = 1 E cycle (caused by the **ldab #100** instruction)
+ 100 x 2 E cycles (caused by the **out_loop ldx #20000** instruction)
+ 100 x 3 E cycles (caused by the **dbne b,out_loop** instruction)
= 501 E cycles = 62.625 µs (at 8 MHz E clock)

This overhead can be reduced by placing a larger value in index register X and a smaller value in accumulator B. For example, by placing 50,000 in X and 40 in B, the overhead can be reduced to 25.125 µs. If higher accuracy is required then you should use one of the timer functions to create the desired time delay.

2.11 Summary

An assembly language program consists of three major parts: *assembler directives*, *assembly language instructions*, and *comments*. A statement of an assembly language program consists of four fields: *label*, *operation code*, *operand*, and *comment*. Assembly directives supported by the freeware **as12** are all discussed in this chapter.

The 68HC12 instructions are explained category by category. Simple program examples are used to demonstrate the applications of different instructions. The 68HC12 is a 16-bit microcontroller. Therefore, it can perform 16-bit arithmetic. Numbers greater than 16 bits must be manipulated using multiprecision arithmetic.

Microcontrollers are designed to perform repetitive operations. Repetitive operations are implemented by program loops. There are two types of program loops: *infinite loop* and *finite loop.* There are four major variants of the looping constructs:

- **Do** statement *S* **forever**
- **For** i = n_1 **to** n_2 **do** *S* or **For** i = n_2 **downto** n_1 **do** *S*
- **While** C **do** S
- **Repeat** *S* **until** *C*

In general, the implementation of program loops requires:

- the initialization of a loop counter (or condition)
- performing the specified operation
- comparing the loop count with the loop limit (or evaluating the condition)
- making a decision regarding whether the program loop should be continued

The 68HC12 provides instructions to support the initialization of a loop counter, decrementing (or incrementing) the loop counter, and deciding whether looping should be continued.

The shifting and rotating instructions are useful for bit field operations. Integer multiplication by a power of two and division by a power of two can be sped up by using the shifting instructions.

The 68HC12 also provides many Boolean logical instructions that can be very useful for setting, clearing, and toggling the I/O port pins.

2.12 Exercises

E2.1 Find the valid and invalid labels in the following statements, and explain why the invalid labels are invalid.


```
a. ABC          decb
b. lp+:         adda        #1
c. too:         mul
d. not_true     nega
e. star+=       ldaa        #10
f. too_big      dec         count
```

E2.2 Identify the four fields of the following instructions:

```
a.              bne         not_done
b. loop         brclr       0,x,$01,loop      ; wait until the least significant bit is set
c. here:        dec         lp_cnt            ; decrement the variable lp_cnt
```

E2.3 Write a sequence of assembler directives to reserve 10 bytes starting from $800.

E2.4 Write a sequence of assembler directives to build a table of ASCII codes of lower-case letters a-z. The table should start from memory location $2000.

E2.5 Write a sequence of assembler directives to store the message "welcome to the robot demonstration!" starting from the memory location at $1050.

E2.6 Write an instruction sequence to add the two 24-bit numbers stored at $810~$812 and $813~$815, and save the sum at $900~902.

E2.7 Write an instruction sequence to subtract the 6-byte number stored at $800~$805 from the 6-byte number stored at $810~$805, and save the result at $900~$905.

E2.8 Write a sequence of instructions to add the BCD numbers stored at $800 and $801 and store the sum at $803.

E2.9 Write an instruction sequence to add the 4-digit BCD numbers stored at $800~$801 and $802~$803, and store the sum at $900~901.

E2.10 Write a program to compute the average of an array of N 8-bit numbers and store the result at $900. The array is stored at memory locations starting from $800. N is no larger than 255.

E2.11 Write a program to multiply two 3-byte numbers that are stored at $800~$802 and $803~$805, and save the product at $900~$905.

E2.12 Write a program to compute the average of the square of all elements of an array with thirty-two 8-bit unsigned numbers. The array is stored at $800~$81F. Store the result at $900~$901.

E2.13 Write a program to count the number of even elements of an array of N 16-bit elements. The array is stored at memory locations starting from $900.

E2.14 Write an instruction sequence to shift the 32-bit number to the left four places. The 32-bit number is located at $800~$803.
E2.15 Write a program to count the number of elements in an array that are smaller than 16. The array is stored at memory locations starting from $800. The array has N 8-bit unsigned elements.
E2.16 Write an instruction sequence to swap the upper four bits and the lower four bits of accumulator A (swap bit seven with bit three, bit six with bit two, and so on).
E2.17 Write a program to count the number of elements in an array whose bits three, four, and seven are zeroes. The array has N 8-bit elements and is stored in memory locations starting from $800.
E2.18 Write an instruction sequence to set bits three, two, one, and zero to one and clear the upper four bits.
E2.19 Find the values of condition flags N, Z, V, and C in the CCR register after the execution of each of the following instructions, given that [A] = $50 and the condition flags are N = 0, Z = 1, V = 0, and C = 1.

(a) SUBA #40	(b) TESTA	(c) ADDA #$50
(d) LSRA	(e) ROLA	(f) LSLA

E2.20 Find the values of condition flags N, Z, V, and C in the CCR register after executing each of the following instructions independently, given that [A] = $00 and the initial condition codes are N = 0, C = 0, Z = 1, and V = 0.

(a) TSTA	(b) ADDA #$40	(c) SUBA #$78
(d) LSLA	(e) ROLA	(f) ADDA #$CF

E2.21 Write an instruction sequence to toggle the odd number bits and clear the even number bits of the memory location at $66.
E2.22 Write a program to shift the 8-byte number located at $800-$807 to the left four places.
E2.23 Write a program to shift the 6-byte number located at $900-$905 to the right three places.
E2.24 Write a program to create a time delay of 100 seconds by using program loops.
E2.25 Write a program to create a time delay of five seconds using program loops.