Chapter 5

HCS12 Machine and Assembly Programming

Introduction

Chapter 4 covered the load and store instructions using different addressing modes. In this chapter (5), you will study many more instructions using the addressing modes you learned in Chapter 4. Moreover, you will take the first step of learning assembly-level programming. Writing a program is a design paradigm. Reading and understanding a program is an analysis paradigm. Generally speaking, design is more challenging than analysis. We also see this in Hardware Design and Analysis, do we not?

Definition: The *static* number of instructions in a program is the number of lines of the code. Let us assume that there is one instruction per line.

Note that some instructions may execute only once, while others multiple times due to the loops in the program.

Definition: The *dynamic* number of instructions in a program is the number of instructions that execute. Some instructions may count once, others more due to the loops in the program.

Before we study different instructions, it should be highlighted that two primary evaluation criteria of object codes are as follows: (Note that we use object code and machine code interchangeably.)

- Size (in bytes)
  - The size tells us how much memory we need to store the program.
  - The program's size is determined by the *static* number of instructions in the program and the type (mnemonic + addressing mode) of the instructions.

The smaller the program is, the better it is.

- Latency (in clock cycles)
  - By latency, we mean how slow the program runs.
  - The latency of a program is determined by the *dynamic* number of instructions and the number of clock cycles that each instruction needs to execute.
  - The reciprocal of the latency is called *performance*.

  The lower the latency is, the faster the program runs and the higher the performance is.

The Machine Coding column in the CPU reference manual shows the generic form of machine instructions and, therefore, the size. For example, the Machine Coding column entry for instruction ldaa in the immediate mode is {86 ii}, where 86 is the opcode and ii is a placeholder for the 8-bit immediate data value. Each character in this column represents 4 bits. Therefore, instruction {86 ii} is 2 bytes long. See the reference manual for other placeholders.

The Access Detail column in the CPU reference manual shows the number of clock cycles for each instruction. Each letter in this column represents one clock cycle. The Access Detail column has a letter P for instruction {86 ii}; therefore, this instruction needs only one cycle to execute. The specific meanings of the letters in this column are beyond the scope of this textbook.

You may also use the CodeWarrior software to obtain the size of instructions and the number of clock cycles each instruction needs. Ask your lab instructor for help.

The rightmost column in the CPU reference manual shows how the NZVC flags are affected by each instruction. The most common scenarios are as follows:

Δ means that the flag changes. Different instructions may have different rules on how to change a flag.

– means that the flag remains unchanged.

0 means that the flag is reset to 0.

1 means that the flag is set to 1.

Assembly is a low-level language compared to, say, Java. By lower level, we mean more difficult for us to understand. The highest-level language is the natural language. The closer a computer language is to the natural language, the higher its level is.

Programming a microcontroller at a low level is just a short distance from the hardware design you learn in a Digital Circuits course. You can "see" the underlying hardware when you use assembly language, while a high-level language such as C does not provide such insight. Programming is a skill that may be *developed over time* through *practice,* but first, you should know about the microcontroller you are using.

Add instructions

Add and similar instructions are classified as 2-address or 3-address instructions: The generic format of a 2-address add instruction is as follows:

Loc ← Loc + second number

Where Loc can be a register or a memory location. As you see, Loc is overwritten in a 2-address add. So, if you need Loc, save it before this instruction executes. We will see more about the "second number" shortly.

The generic format of a 3-address add instruction is as follows:

Loc2 ← Loc1 + second number

Loc1 is not overwritten anymore, but in return, 3-address instructions are longer, hence taking more memory space.

The HSC12's add instructions have the 2-address format with a **register-type destination,** as shown in Figure 1. Review this self-explanatory table rigorously. Operand [8] and Operand [16] mean an 8-bit and a 16-bit operand, respectively. Either Operand can be specified in any addressing mode you learned in Chapter 4 for the load instructions. The add instructions in the first five rows of this table and the load instructions have the same assembly and machine syntax, hence the same size, when using the same addressing mode. Remember that the Operand can be a constant or memory location, and it is always the same size as the destination register:

	Instruction Syntax	Instruction Semantics	Comments
1	adca Operand [8]	A ← A + Operand [8] + C	C is the carry flag NZVC affected meaningfully
2	adcb Operand [8]	B ← B + Operand [8] + C	C is the carry flag NZVC affected meaningfully
3	adda Operand [8]	A ← A + Operand [8]	NZVC affected meaningfully
4	addb Operand [8]	B ← B + Operand [8]	NZVC affected meaningfully
5	addd Operand [16]	D ← D + Operand [16]	NZVC affected meaningfully
6	aba	A ← A + B	NZVC affected meaningfully
7	abx	X ← X + B	B is zero-extended, NZVC not affected
8	aby	Y ← Y + B	B is zero-extended, NZVC not affected
The object codes are determined like the load instructions.			

Figure 1. Different types of add instructions.

Example 1. Figure 2 shows three pairs of add/load instructions as an example. *The same addressing mode is used in each pair*. The first byte in the Object row is the opcode:

	Pair 1		Pair 2		Pair 3	
	load	add	load	add	load	add
Assembly	ldaa 5, X	adca 5, X	ldab 8, -Y	adca 8, -Y	ldd D, SP	addd D, SP
Object	A605	A905	E668	A968	ECF6	E3F6

Figure 2. Three pairs of instructions in Example 1.

The instruction in row 6 of Figure 1 is aba. The addressing mode of aba is called Inherent or INH for short. Its object code is {18[1] 06} as shown in the Machine Coding column of the reference manual. There is no inherent mode defined for load/store instructions. Register-to-register-transfer-type instructions, for example, are not called load; they have different mnemonists, as you will see. This chapter covers more inherent-mode instructions, which do not need a separate byte for the operands.

[1] $18 is called the *prebyte*, which is followed by the opcode. The prebyte is not covered in this edition of the textbook.

Definition: A *true* instruction is an assembly-level instruction with an exact machine language translation. All the instructions you have seen so far are true.

Definition: A *pseudo* instruction (as opposed to a *true* instruction) is an assembly-level instruction that does *not* have an exact translation in machine language. The assembler lets us use pseudo instructions to make programming *easier*. A pseudo instruction is translated into one or more true instructions and then into machine language. You will see some pseudo instructions in this chapter.

abx and aby are *pseudo* instructions. See rows 7 and 8 in Figure 1. The assembler translates them into true instructions {LEAX B, X} and {LEAX B, Y}, respectively, and then into machine language. These instructions and more will be explained later in this chapter.

The abx and aby instructions do not affect the NZVC flags. After an add instruction, other than abx and aby, executes, the N flag shows the MSb of the sum. Note, however, that MSb is not necessarily the sign bit; it is the sign bit only if there is no overflow; in other words, in case of overflow, the correct sign bit is the *opposite* of the MSb or N flag. The Z flag is set to 1 only if the sum is zero. The V flag is set to 1 only if overflow occurs. The C flag is set to 1 only if carry is generated.

Example 2. As a result of the following addition, the N flag will be 0, although the correct sign bit is 1 because overflow occurs in this addition.

1101 + 1001 = 10110 (The numbers are 4-bits wide to keep the example simple.)

Example 3. As a result of the following addition, the N flag will be 1, which is the sign bit of the correct sum because overflow does not happen in this addition.

1011 + 1111 = 11010 (The numbers are 4-bits wide to keep the example simple.)

Example 4. Write a program to add the three 8-bit numbers in the memory at $1000, $1001, and $1002, and store the sum at $1010, regardless of whether the sum is incorrect due to overflow.

```
here:   ldaa    $1000    ; put comments after a semicolon
        adda    $1001
        adda    $1002
        staa    $1010
```

With some exceptions, there are up to 4 fields in each row of an assembly program, namely, Label, Mnemonic (or Operation), Operand, and Comments, as shown in this example, where "here" is a Label, "ldaa" is a Mnemonic, "$1000" is an Operand and "put comments after s semicolon" a Comment. A *colon* must follow labels. A *semicolon* must precede comments.

Note: Do not start Mnemonics at column 1 when using CodeWarrior.

Example 5. *Read, Modify, Write*: The following program adds the two 8-bit numbers located in the memory at $2000 and $3000 and places the sum at $2000:

```
ldaa    $2000    ; Read
adda    $3000    ; Modify
staa    $2000    ; Write
```

What happens here is called the {*read*, *modify*, *write*} algorithm, which means you read a memory location, make some changes, and then send it back to the same location. This process will frequently be used in this and the coming chapters when you need to change the content of a memory location.

Example 6. The following program uses the {*read*, *modify*, *write*} algorithm to replace the 16-bit number @ $3000 : $3001 with the sum of two 16-bit numbers at $3000 : $3001 and $3002 : $3003. Note that the most significant byte is at the smaller address:

```
ldd     $3000       ; read
addd    $3002       ; modify
std     $3000       ; write
```

Example 7. Assemble the following instruction (which is in the extended mode), place it in the memory starting at address $4000, and then execute it:

```
adda    $3001   ; A ← A + ($3001)
```

The object code is determined like the load instructions you learned in Chapter 4. Remember that we use Rule 2 to obtain the opcode. In Figure 3a, the assembled instruction is placed in the memory starting at address $4000. Let us assume that Figure 3a also shows the relevant values in the memory and registers *before* the instruction executes. The results of instruction execution are shown in Figure 3b.

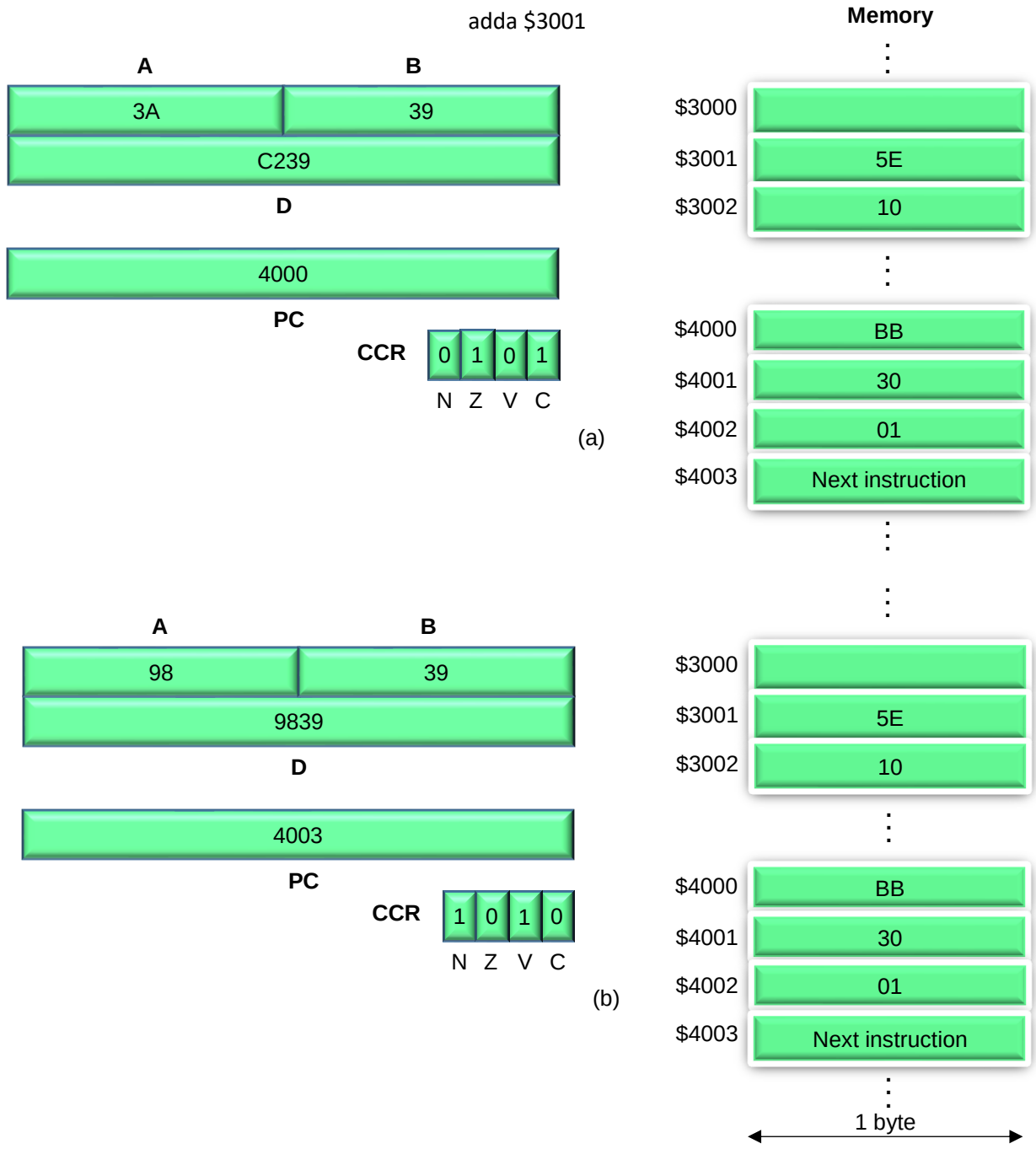


Figure 3. Instruction execution in Example 7.

Example 8. Write a program to add "the byte pointed to by X" to "the next byte" and place the sum at the memory location pointed to by Y:

```
ldaa    0, X
adda    1, X
staa    0, Y
```

Subtract instructions

Like the add instructions, the HSC12's subtract instructions have the 2-address format with a **register-type destination,** as shown in Figure 4. Review this self-explanatory table rigorously. Operand [8] and Operand [16] mean an 8-bit and a 16-bit operand, respectively. Either Operand can be specified in any addressing mode you learned in Chapter 4 for the load instructions. When using the same addressing mode, the subtract instructions in the first five rows of this table and the load instructions have the same assembly and machine syntax, hence the same size. Remember that the Operand can be a constant or memory location, and it is always the same size as the destination register:

	Instruction Syntax	Instruction Semantics	Comments
1	sbca Operand [8]	A ← A – Operand [8] – C	• C is the carry flag and means **active-high** borrow flag. • NZVC affected meaningfully. • The object codes are determined like the add instructions.
2	sbcb Operand [8]	B ← B – Operand [8] – C	
3	suba Operand [8]	A ← A – Operand [8]	
4	subb Operand [8]	B ← B – Operand [8]	
5	subd Operand [16]	D ← D – Operand [16]	
6	sba	A ← A – B	
7		~~X ← X – B~~	Not supported.
8		~~Y ← Y – B~~	

Figure 4. Different types of subtract instructions.

The instruction in row 6 of Figure 4 is sba. The object code of sba is {18 16} as shown in the reference manual. Similar to instruction aba, the addressing mode of sba is Inherent.

For the following two add instructions, there are no similar subtract instructions, as shown in Figure 4:

```
abx     ; X ← X + B
aby     ; Y ← Y + B
```

Example 9. Figure 5 shows three pairs of subtract/load instructions as an example. *The same addressing mode is used in each pair*. The first byte in the Object row is the opcode:

	Pair 1		Pair 2		Pair 3	
	subtract	**load**	**subtract**	**load**	**subtract**	**load**
Assembly	sbca 5, X	ldaa 5, X	sbca 8, -Y	ldab 8, -Y	subd D, SP	lddd D, SP
Object	A205	A605	A268	E668	A3F6	ECF6

Figure 5. Three pairs of instructions to show the similarity between the subtract and load.

The carry flag shows the active-high (not the active-low) borrow for the subtract instructions. So, if a borrow is generated in a subtract instruction, the C flag will be set to logic 1 and vice versa. The object codes of the subtract instructions are determined like the add instructions.

After a subtract instruction executes, the N flag shows that MSb of the difference, no matter whether the difference is incorrect (in case of overflow) or correct. Therefore, if overflow does not happen, the N flag is the sign bit of the correct difference; otherwise, the MSb of the incorrect difference must be inverted to get the correct sign bit. These rules also apply to the *compare* instructions coming up in this chapter. The compare instructions perform a subtraction and affect the flags similar to the subtract instructions; however, they do not save the difference anywhere.

Example 10. As a result of the following subtraction, the N flag will be 0, although the correct sign bit is 1 because overflow occurs in this subtraction:

1001 – 0101 = 1001 + 1011 = 10100 (The operands are 4-bits wide to keep the example simple.)

Example 11. As a result of the following subtraction, the N flag will be 1, which is the sign bit of the correct difference because overflow does not happen in this subtraction:

1101 – 0101 = 1101 + 1011 = 11000 (The operands are 4-bits wide to keep the example simple.)

Example 12. The following program subtracts constant 5 from the 8-bit number sitting in the memory at $3000 and returns the difference to address $3000:

```
ldaa    $3000
suba    #5
staa    $3000
```

Example 13. Assemble the following instruction, place it in the memory starting at address $4000, and then execute it.

```
suba $3001    ; A ← A – ($3001)
```

Following Rule 2, the opcode is looked up: $B0.

In Figure 6a, the assembled instruction is sitting in the memory, starting at $4000. Let us assume that Figure 6a also shows the relevant values in the memory and registers *before* the instruction executes. The results of instruction execution are shown in Figure 6b:

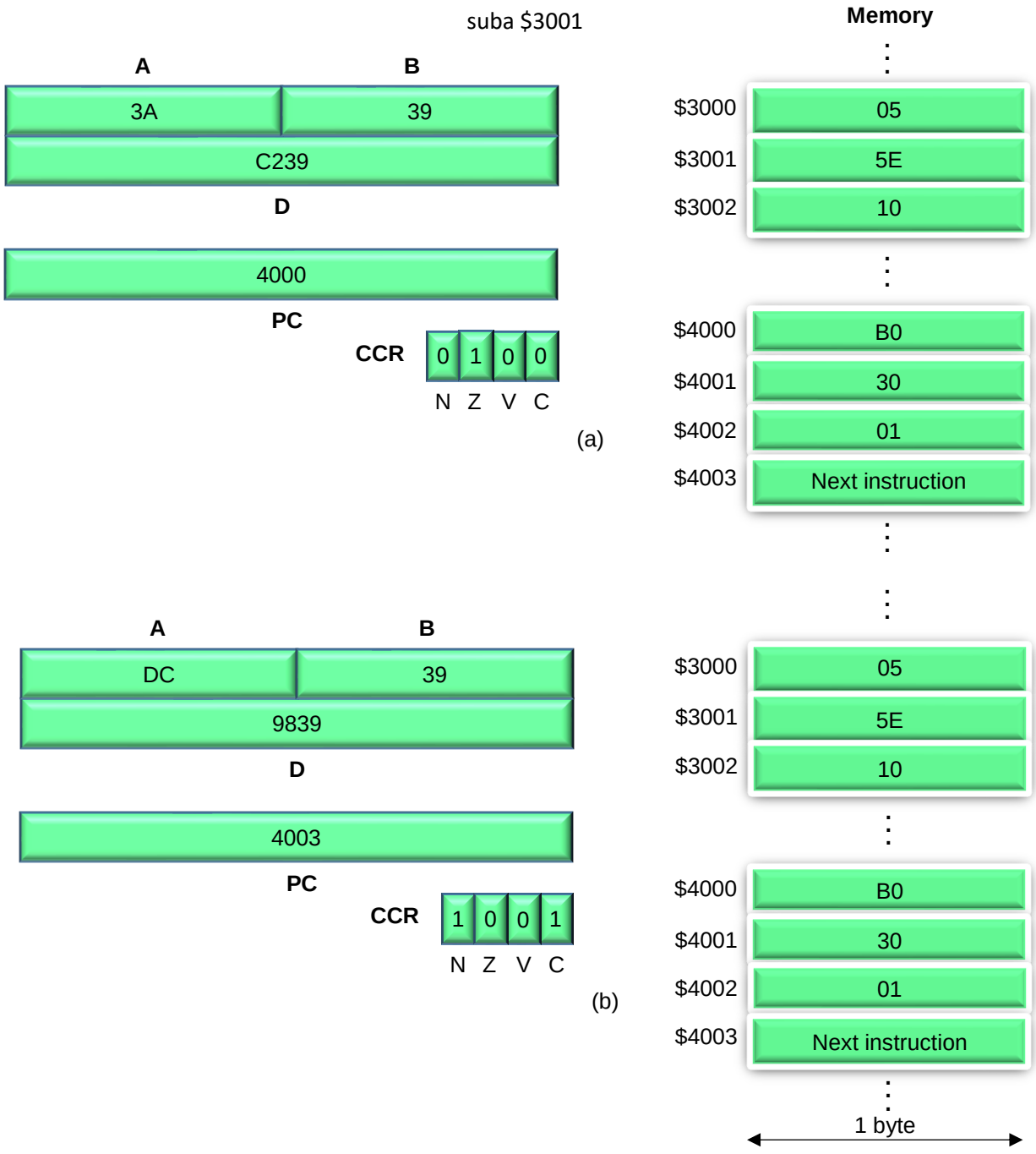


Figure 6. Instruction execution in Example 13.

Clear instructions

There are 3 types of clear instructions:

```
clra              ; Reset accumulator A to 0. INH mode. One byte long. Object ode: $87
clrb              ; Reset accumulator B to 0. INH mode. One byte long. Object ode: $C7
clr  Memory [8]   ; Reset memory byte represented with "Memory [8]" to 0.
```

Except for the direct mode, all the addressing modes used in the store instructions can be used to specify Memory [8]. The clear and store instructions have the same assembly and machine syntax when they use a memory operand with the same addressing mode.

The NZVC flags will be 0100 unconditionally after a clear instruction executes.

Example 14. Figure 7 shows 3 pairs of clear/store instructions as an example. The *same addressing mode* is used in each pair. The first byte in the Object row is the opcode:

	Pair 1		Pair 2		Pair 3	
	clear	**store**	**clear**	**store**	**clear**	**store**
Assembly	clr 5, X	staa 5, X	clr 8, -Y	stab 8, -Y	clr D, SP	staa D, SP
Object	6905	6A05	6968	6B68	69F6	6AF6

Figure 7. Three pairs of instructions in Example 14.

Example 15. Let us say X = $3000. Then the following instruction will reset the memory location at address $3005:

```
clr 5, X    ; Object code = $6905
```

Increment instructions

There are 6 types of increment instructions:

```
inca            ; A ← A + 1. INH mode. One byte long. Object code: $42
incb            ; B ← B + 1. INH mode. One byte long. Object code: $52
inc Memroy[8]   ; memory byte represented with "Memroy[8]" in incremented by 1.
```

Except for the direct mode, all the addressing modes used in the store instructions can be used to specify Memory [8]. The increment and store instructions have the same assembly and machine syntax when they use a memory operand with the same addressing mode.

The NZV flags are affected meaningfully, and the C flag remains unchanged.

```
inx     ; X ← X + 1, only the Z flag is affected (meaningfully). INH mode. One byte long. Object code: $08
iny     ; Y ← Y + 1, only the Z flag is affected (meaningfully). INH mode. One byte long. Object code: $02
ins     ; SP ← SP + 1, pseudo instruction. Translated to LEAS 1, X. Flags are not affected. Two bytes long.
        ; See LEAS instruction, coming soon.
```

No ind (increment D) instruction.

Example 16. Figure 8 shows 3 pairs of increment/store instructions as an example. The *same addressing mode* is used in each pair. The first byte in the Object row is the opcode:

	Pair 1		Pair 2		Pair 3	
	increment	**store**	**increment**	**store**	**increment**	**store**
Assembly	inc 5, X	staa 5, X	inc 8, -Y	stab 8, -Y	inc D, SP	staa D, SP
Object	6205	6A05	6268	6B68	62F6	6AF6

Figure 8. Three pairs of instructions in Example 16.

Example 17. Let us say Y = $3000, and Mem ($3005) = $0D. Then the following instruction will change the content of Mem (3005) to $0E:

```
inc 5, Y    ; Object code = $6245
```

Decrement instructions (similar to increment instructions)
There are 6 types of decrement instructions:

deca ; A ← A – 1. INH mode. One byte long. Object code: $43

decb ; B ← B – 1. INH mode. One byte long. Object code: $53

dec memory [8] ; memory byte represented with "memory [8]" in decremented by 1.

Except for the direct mode, all the addressing modes used in the store instructions can be used to specify Memory [8]. The increment and store instructions have the same assembly and machine syntax when they use a memory operand with the same addressing mode.

The NZV flags are affected meaningfully, and the C flag remains unchanged.

dex ; X ← X – 1, only the Z flag is affected (meaningfully). INH mode. One byte long. Object code: $09

dey ; Y ← Y – 1, only the Z flag is affected (meaningfully). INH mode. One byte long. Object code: $03

des ; SP ← SP – 1. Pseudo instruction. Translated to LEAS -1, X. Flags are not affected. Two bytes ; long. See LEAS instruction, coming soon.

No ded (decrement D) instruction.

Example 18. Figure 9 shows 3 pairs of decrement/store instructions as an example. The *same addressing mode* is used in each pair. The first byte in the Object row is the opcode:

	Pair 1		Pair 2		Pair 3	
	decrement	**store**	**decrement**	**store**	**decrement**	**store**
Assembly	dec 5, X	staa 5, X	dec 8, -Y	stab 8, -Y	dec D, SP	staa D, SP
Object	6305	6A05	6368	6B68	63F6	6AF6

Figure 9. Three pairs of instructions in Example 18.

Branch instructions

Introduction

In this introduction, the following topics will be discussed. The branch instructions are covered after the introduction:

- Loops and Arrays
- Flowcharts
- How to approach a problem and program evaluation
- If Then Else statements

Instructions in a program execute one after another, and in the same order they are sitting in the program, hence memory, unless a branch (AKA jump) instruction is reached and executed. Then the next instruction will be the one specified by a label in the assembly branch instruction. Soon, you will see how labels are translated into machine language.

In the following 2-line code, after the bra, branch instruction, the instruction labeled again (sitting somewhere in the program) will be fetched and executed, not instruction ada, which is seated right after the bra:

```
bra again
ada
```

This branch instruction is called a *unary or unconditional* branch, as it will *always* jump. A *conditional* branch instruction is a flexible version of the unconditional one. A conditional branch checks a *condition*; if it is *not satisfied*, program execution continues as if there were no branch instruction, i.e., the next instruction (after the branch instruction) will be the one physically located right after the branch instruction. This is called an *unsuccessful* or *untaken* branch. The branch instruction is called *successful* or *taken* if the condition is satisfied. A **successful branch** instruction looks like an **unconditional branch** instruction; therefore, the next instruction will be the one specified by the label in the branch instruction. In this chapter, you will see the *conditions* used in the conditional branch instructions.

Branch instructions are a must to create program loops, which are usually necessary to process arrays. So, before different types of conditions are covered, let us see what *arrays* and *loops* are:

Loops: A *loop* is a group of instructions executed from top to bottom (called one *iteration* or *cycle* of the loop), and then this iteration is repeated a number of times, or maybe forever! So, loops are created in the *code* segment.

We need unconditional branches to create infinite loops (AKA endless loops). To create *finite* loops, we need *conditional* branches, as elaborated in this section. Can you think of a program without a loop in it? In addition to loops, we need conditional branches to create "IF THEN ELSE"-like statements, as you will see later.

Arrays: A group of consecutive memory locations is called an *array*. You place data in an array to access and process them easily. There is usually a good reason for this group *formation*, e.g., you may form an array to store your grades; a text can be stored in an array; you may sample and digitize a voice clip and put the digital samples in an array, you may keep the 7-segment codewords in an array, etc. Note that arrays are created in the *data* segment.

Finite loops are (usually) a must for array processing. Infinite loops are typically used to check something forever and make a decision accordingly, e.g., you may check the temperature forever and turn on the air conditioner if the temperature is too high; otherwise, turn it off.

Examples of array processing: You may wish to

- Calculate the average of your grades sitting in an array.
- Send the message stored in an array to an LCD in an embedded system.

- Read the voice samples from an array, and perform signal processing.
- Use an array loaded with the 7-segment codewords as a lookup table to convert BCD to 7-segment code.

To process an array, you need to know.

- Where the array starts: Use a pointer that initially points to the array's last or first element (base), and update the pointer to move on to the next element when you finish the current one.
- Where the array ends: select one of the following methods:
  - If the array size is known,
    - use a *loop counter*, which is usually initialized to the array size and then decremented properly (updated) after each array element is processed. When the loop counter hits a zero, you are done.
    - Using the array size, calculate the address of the end of the array. Once an array element is processed, check if the array pointer has reached the end of the array; if it has, you are done; otherwise, update the pointer to reach the next element and continue.

    This method has two scenarios: the array size may be known at compile or run time. The compile time means when you write the program, so if you know the size at compile time, the size is constant. On the other hand, you may read the size when the program is running, then the size is not constant anymore:

    Example: Convert an ASCII array of 100 lowercase characters to uppercase text. The size (100) is constant, i.e., it is known at compile time (when you write the program).

    When the loop size is constant, you may *unroll* or *unwind* the loop if the loop size is constant[2]. So, for a 5-iteration loop (as an example), you may repeat the body of the loop 5 times, which will, of course, make the code longer. For long arrays, this technique is not an option. The *body* of a loop is defined in Figure 10 and the associated text. Soon, we will see more about loop unrolling.

    Example: Convert an ASCII array of N lowercase characters to uppercase text. The size (N) is not constant anymore, i.e., it will be known at run time. In this example, you may not unroll the loop, as the array size is not constant.

  - Place a special value in the last location of the array, then check the array elements (as you read them) to see if a match occurs; if it does, you are done; otherwise, continue. For text arrays, a NULL character (an all-zero byte) is what we place at the end of the array. Loop unrolling is impossible to process this type of array, as we do not know (even at rum time) how many iterations the loop will perform.

    Example: Convert a NULL-terminated lower-case ASCII array to uppercase.

A flowchart is a graphical description of the *algorithm* to solve the problem, or it shows the *control flow* (as opposed to *data flow*) to solve the problem, i.e., the steps that should be taken *one after another* to get the job done. Transition graphs, which you learn in your hardware design courses, are a more rigorous form of flowcharts. *Data flow*, on the other hand, shows how data moves around to get the job done. In the next chapter, you should gain a good understanding of the concept of data flow in *programming*. *Block diagrams* (also called datapaths), which you learn in your hardware design courses, show data flow in *hardware*. An example of hardware data flow is given in Chapter 3, when a simplified view of a microcomputer is presented.

[2] Provided that there is enough hardware support.

How to approach a problem

Let us say we are provided with the word description of a problem in the natural language. We usually design an assembly code to solve the problem in two stages, as described here:

Stage I: Draw a flowchart. Call it an *intermediate* solution. This way, you get one big step closer to the final program. At the same time, you may look at the flowchart as the *problem formulation*, a new level of the problem description. In stage II, described soon, you translate the flowchart into an assembly program.

A flowchart can be drawn at *different levels*: it can be very close to the word description of the problem (in the natural language); it can also be very close to the final solution, i.e., the assembly program. The best choice, however, is neither too close to nor too far from the final solution.

In general, we may have two types of *variables* in a program: *memory type* and *register type*. For example, we may use a register or a memory location as a loop counter in a program. Generally speaking, register-type variables are preferred as the associated instructions are shorter in terms of size and execution time. For example, instruction `clra` (clear register A) is only one byte long and takes only one clock cycle to execute, while a `clr` (clear) instruction using the extended addressing mode is 3 bytes long and takes 3 cycles to complete.

In the HCS12 and similar microcontrollers, there are few general-purpose registers to use as variables. Therefore, you must use memory-type variables when you run out of registers. This is why in the first draft of your flowchart, it may not be easy to specify the types (register or memory location) of the variables, so give the variables meaningful names, e.g., *sum, counter, char, temp, etc*., and draw a flowchart using these names. Then decide on the types of variables while giving priority to registers. Use memory locations when you have to.

Example 19. The flowchart in Figure 10 is the problem *formulation* for the following word description:
Write a program (in HCS12 assembly language) that converts an N-character uppercase-only text into lowercase. N can be a constant or a variable.

Go over the flowchart rigorously. As you see, the loop should perform N iterations, one iteration per character. We need a conditional branch, represented with a diamond, to implement this loop. **We cannot decide whether to end or continue the loop with no conditional branch.**

To implement this loop and in addition to a conditional jump, we also need an **indexed addressing mode** as the address from which a character is read **has to change** every loop iteration; in other words, now we need a *variable address* (and not a constant one), which is what the indexed addressing provides. (Remember, the extended or immediate modes cannot change the address while the program is running.) The indexed addressing can access all the array elements in consecutive loop iterations once the pointer has been updated properly. **And this shows how vital the indexed addressing is.**

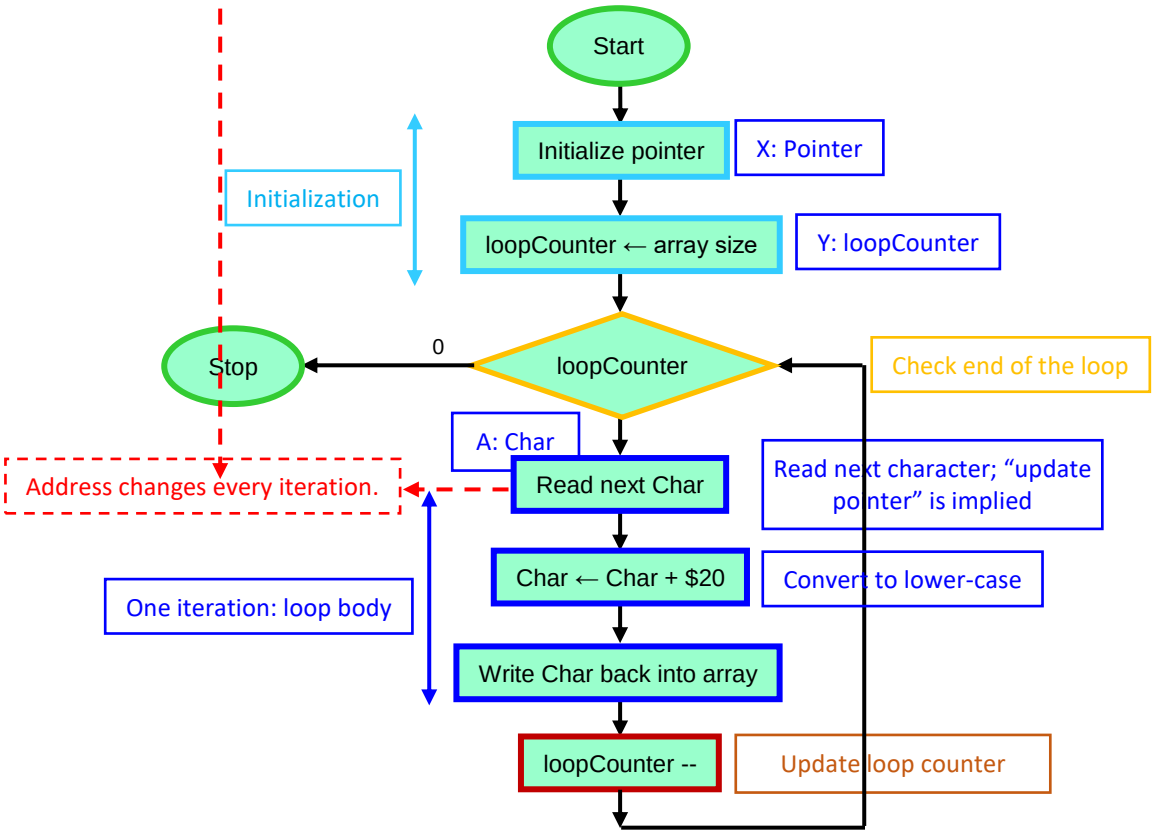


Figure 10. Flowchart to convert uppercase-only text to lowercase; text size can be constant or variable.

Before the loop starts, we need to initialize the pointer and the loop counter. See Figure 10. The loop consists of three sections, as shown in this figure:

- "Body of the loop", where the task is carried out,
- "Check the end of the loop", where we decide whether to end or continue the loop,
- "Update pointer and loop-counter" to prepare for the next iteration. **Note**: in Figure 10, the "update pointer" step is moved to the body of the loop and implied in the "Read next Char" block for more readability.

The last two sections are part of the cost (overhead) to create a loop. To reduce or avoid the price, we have to unroll the loop if the array size N is constant, i.e., repeat the body of the loop N times after it has been modified properly. Figure 11 shows the unrolled loop where N = 4. The unrolled version is faster than the loop-based version, as there is less or maybe no overhead of "check the end of the loop" and "update pointer/loop-counter"; in return, the program becomes longer. Therefore, for long arrays, loop unrolling is

not an option. Remember, we cannot unroll the loop when the array size is unknown at compile time (i.e., when it is not constant).

Note: In the loop body of Figure 10, we have used the *Read*, *Modify*, *Write* algorithm.

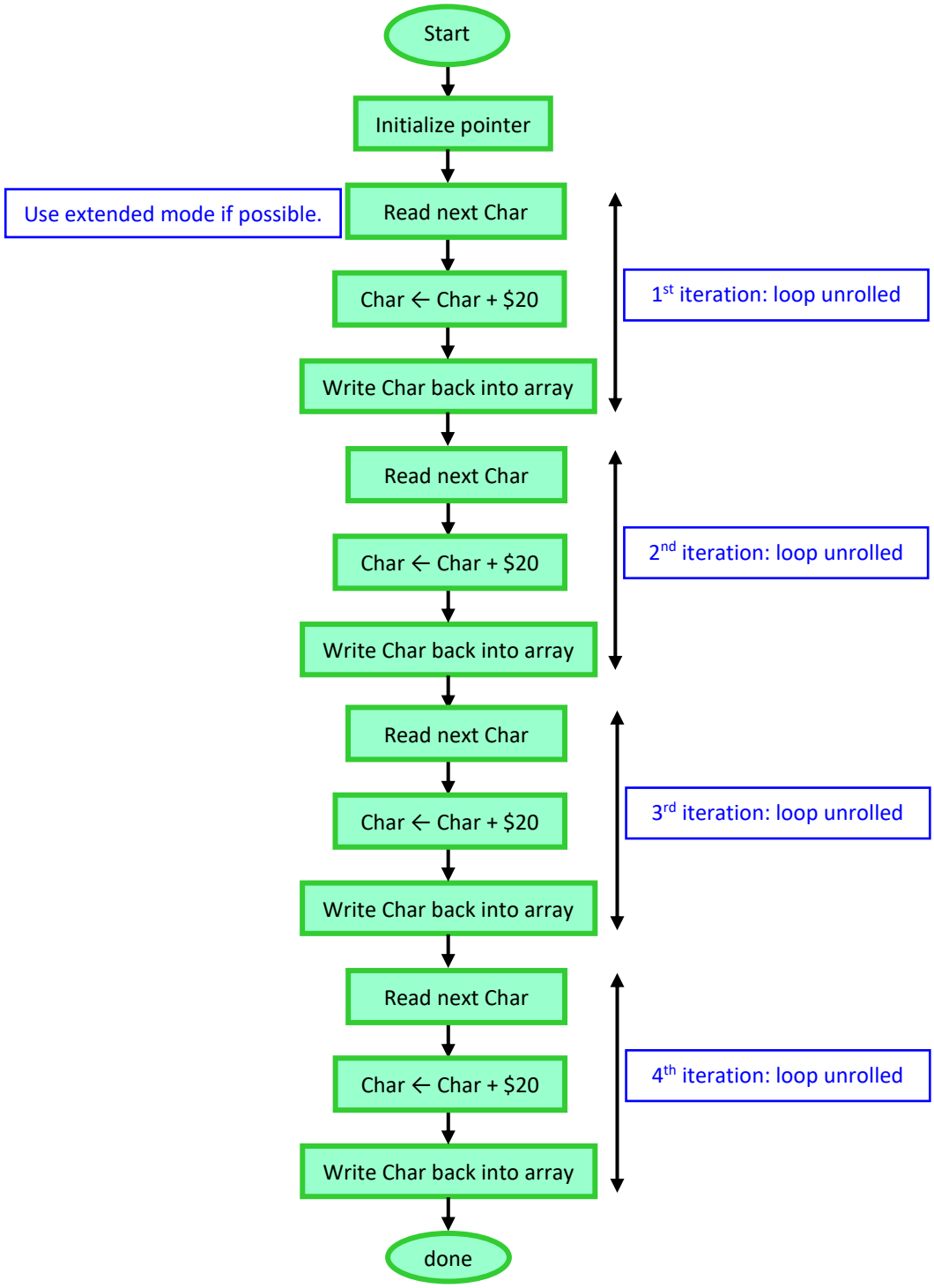


Figure 11. Flowchart to convert uppercase text to lowercase; text size = 4; loop unrolling.

Program evaluation

Two main criteria for program evaluation are the *size* and *performance* of the object code. The code size in terms of bytes is determined by the number of lines of instructions (the *static* number of instructions) and their types. The program performance is the reciprocal of the total number of clock cycles the program needs to run. This number depends on the *dynamic* number of instructions and the number of clock cycles each instruction needs to execute.

Example 20. Figure 12 shows a program with a *static* number of instructions equal to 7, as there are 7 lines of code. The size of each (machine) instruction (in bytes) and the number of clock cycles to execute the instruction are also shown in this table. The Machine Coding and Access Detail columns, respectively, in the CPU reference manual show these two pieces of information for every instruction. You may also obtain these numbers using CodeWarrior. When you add up the sizes of all the instructions, you get the size of the entire program:

Size of program = 3 + 3 + 4 + 3 + 1 + 2 + 2 = 18 bytes.

Line No	Assembly program	Size (bytes)	Number of clock cycles
1	ldy #0	3	2
2	ldx #$3000	3	2
3	brclr X, $FF, done	4	4
4	again: bset 1, X+, #$20	3	4
5	iny	1	1
6	ldaa 0, X	2	3
7	bne again	2	3/1 (taken/untaken)
	done: …		

Figure 12. The program in Example 20.

To obtain the latency of the program, in other words, the total number of clock cycles to run the program, we need to consider the *dynamic* number of instructions, i.e., the number of instructions that the CPU executes. Therefore, when there is a loop in a program, each instruction inside the loop counts once in every iteration; if the number of iterations is N, then each instruction counts as N.

Back to the program in Figure 12, the loop consists of instructions 4 through 7. Let us assume that the loop performs 1000 iterations. So, in the first 999 iterations, the bne instruction will be successful (taking 3 clock cycles in each iteration), while in the last iteration, the bne instruction will be unsuccessful, taking only one clock cycle as shown in the reference manual:

Number of clock cycles used per iteration in the first 999 iterations = 4 + 1 + 3 + 3 = 11.

The total number of clock cycles used by the first 999 iterations = 999 x 11.

The number of clock cycles for the last iteration = 4 + 1 + 3 + 1 = 9.

The number of clock cycles used by the first 3 instructions (outside the loop) = 2 + 2 + 4 = 8.

This program's total number of clock cycles = 999 x 11 + 9 + 8 = 11006.

Note that, generally speaking, instructions outside the loops do not significantly affect the performance as they execute only once. To improve the performance, you should focus on optimizing inside the loops. However, note that there is no difference between instructions inside or outside loops if memory usage is concerned.

Exercise: Figure 13 shows another version of the flowchart of Figure 10. Show that the new version needs fewer instructions inside the loop. Compare the two flowcharts and find out if performance or size is improved:

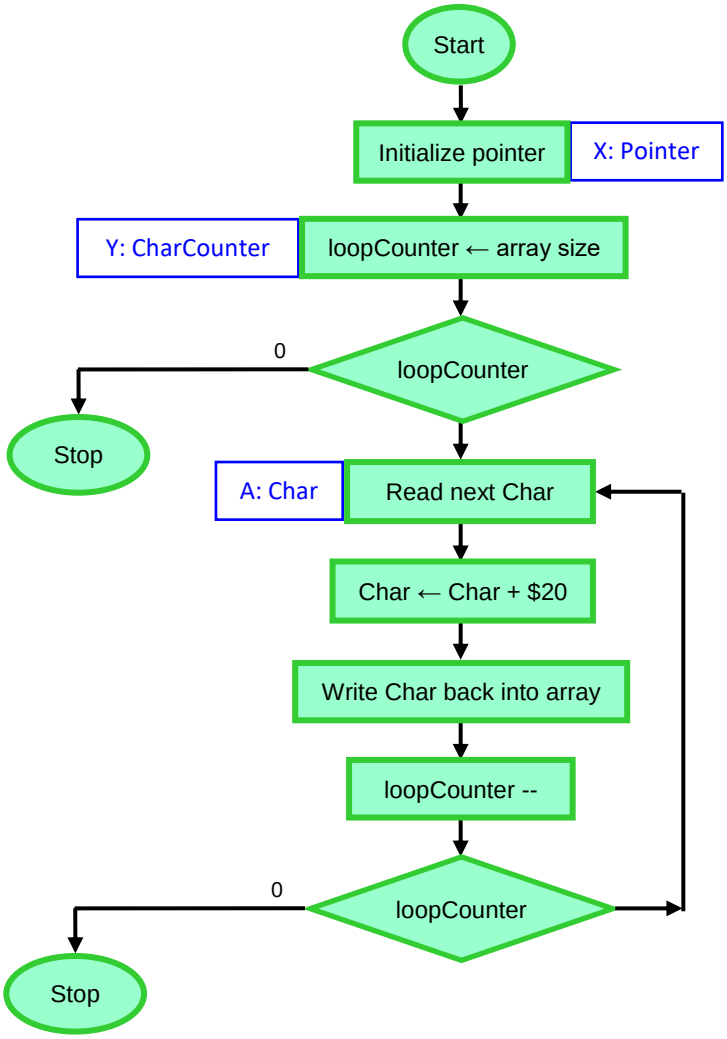


Figure 13. Another version of the flowchart shown in Figure 10.

Example 21. The flowchart in Figure 14 is the problem *formulation* for the following word description: Write a program (in HCS12 assembly language) that converts a NULL-terminated uppercase-only text into lowercase and, in the meantime, counts the characters as well.

In this problem, the array size is unknown not only at compile time but at run time as well; now, to check the end of the array, we have to check the array elements as they are read to see if a NULL character is reached:

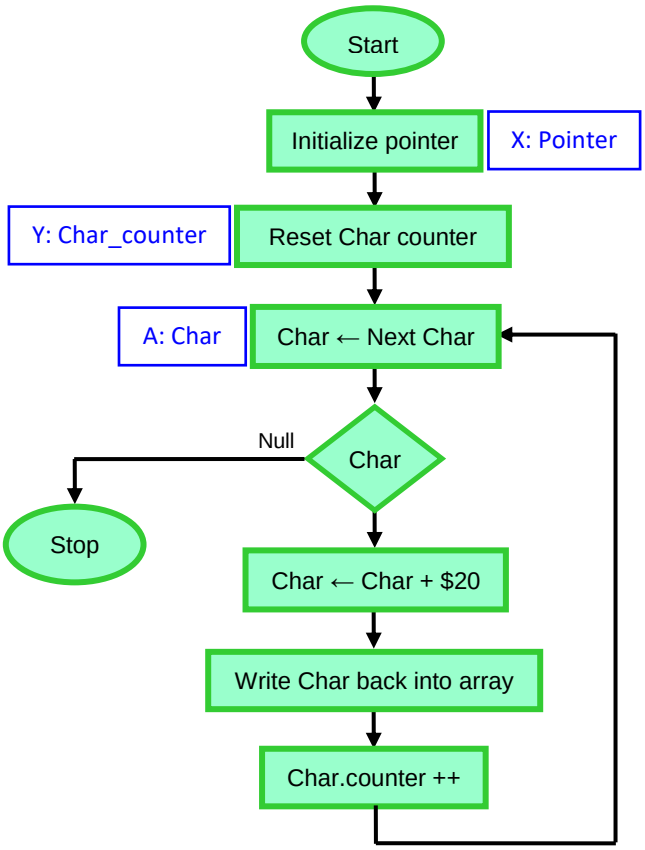


Figure 14. Flowchart to convert uppercase text to lowercase; unknown text size.

Stage II: Once you are done with the flowchart, and have decided on the variables, translate the flowchart into assembly language. Look at each block of your flowchart, either a **rectangle** (to update/initialize a variable) or a **diamond** (for comparisons/making decisions) and translate it. A single block (in the flowchart) may be translated into more than one assembly instruction; on the other hand, a single assembly instruction may perform a portion of one block and a portion of the following block in the flowchart. Remember, considering the complex instruction set of the HCS12 microcontroller, a flowchart may be translated into different assembly programs. Optimal translation, which improves the quality of your assembly code, is a programming skill you should develop in this textbook. It is **common to backtrack from the assembly program to the flowchart to fix logical errors (if any) or** improve the algorithm (if possible).

Example 21. (Cont'd): Figure 15 shows an assembly code based on the flowchart of Figure 14. Go over the code rigorously.

Exercise: An improved flowchart for Example 21 is illustrated in Figure 16, and an assembly code based on this flowchart is shown in Figure 17. Analyze them carefully.

Exercise: Compare the two programs shown in Figure 15 and Figure 17.

```
        ldy     #0      ; Init Char counter
        ldx     #$3000  ; Init pointer
again:  ldaa    0, X    ; read next Char
        beq     done    ; if NULL, we are done
        adda    #$20    ; otherwise, convert it to lowercase
        staa    1, X+   ; send it back to array, and then update pointer
        iny             ; update Char counter
        bra     again   ; do it again
done:   bra     done    ; stay here forever!
```

Figure 15. Assembly code based on the flowchart of Figure 14.

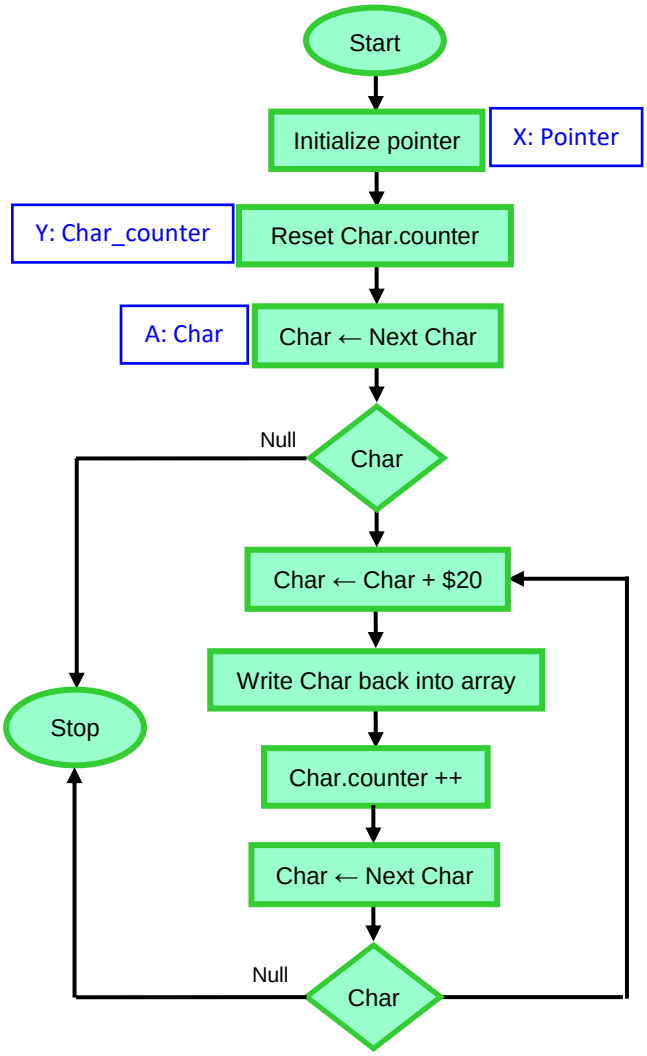


Figure 16. Improved flowchart to convert uppercase text to lowercase; unknown text size.

```
        ldy     #0          ; Init Char counter
        ldx     #$3000      ; Init pointer
        ldaa    0, X        ; read next Char
        beq     done        ; if NULL, we are done
again:  adda    #$20        ; otherwise convert it to lower case
        staa    1, X+       ; send it back to array and update pointer
        iny                 ; update Char counter
        ldaa    0, X        ; read next Char
        bne     again       ; continue if it is not NULL, otherwise we are done ☺
done:   bra     done        ; stay here forever!
```

Figure 17. Assembly code based on the flowchart of Figure 16.

Exercise: Figure 18 shows a different version of the code shown in Figure 15, and Figure 19 shows a different version of the code shown in Figure 17. Go over the new versions and see if they are improved in size or performance:

```
        ldy     #0          ; Init Char counter
        ldx     #$3000      ; Init pointer
        ldaa    0, X        ; read next Char
        beq     done        ; if NULL, we are done
again:  bset    1, X+, #$20 ; otherwise convert it to lower case
        iny                 ; update Char counter
        ldaa    0, X        ; read next Char
        bne     again       ; continue if it is not NULL, otherwise we are done ☺
done:   bra     done        ; stay here forever!
```

Figure 18. Another version of the code shown in Figure 15.

```
        ldy     #0          ; Init Char counter
        ldx     #$3000      ; Init pointer
        brclr   X, $FF, done ; if NULL, we are done
again:  bset    1, X+, #$20 ; otherwise convert it to lower case
        iny                 ; update Char counter
        ldaa  0, X          ; read next Char
        bne     again       ; continue if it is not NULL, otherwise, we are done ☺
done:   bra     done        ; stay here forever!
```

Figure 19. Another version of the code shown in Figure 17.

IF-Then-Else stalemates

As shown in Figure 20, we need a conditional branch instruction to implement the IF-THEN-ELSE statements as well:

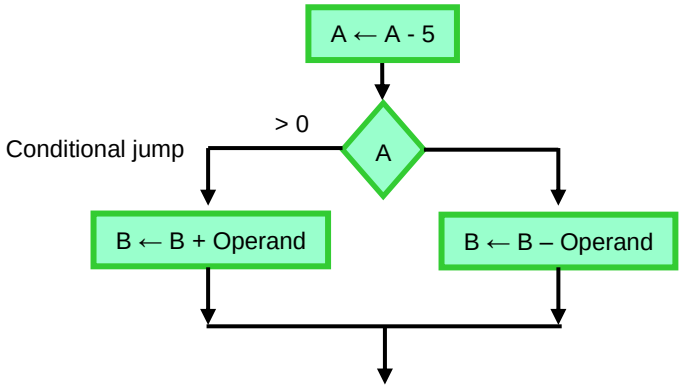


Figure 20. IF-THEN-ELST statements need conditional branch instructions as well.

End of Introduction

Branch instructions classification

The HCS12 supports five types of branch instructions: unconditional (or unary), simple, unsigned relational, signed relational, Boolean (or bit condition), and loop primitive.

Unconditional or unary branch instruction

Here is the assembly syntax of this branch instruction:

```
bra label
```

After this bra, the instruction labeled label will be fetched and executed unconditionally.

Question: Do you see any issue with a program that contains the following two code lines?

```
bra label
suba
```

Simple conditional branch instructions

There are 2 simple conditional branch instructions for each N, Z, V, or C flag. One branches only if the condition flag is 1; the other branches only if the condition flag is 0. Therefore, there are 8 simple branch instructions in total. They are listed in Figure 21. None of these instructions affect the condition flags:

Syntax of simple conditional branch instruction	Semantic (CPU's logic)
beq label	branch if equal (branch if Z = 1)
bne label	branch if not equal (branch if Z = 0)
bcs label	branch if carry set (branch if C = 1)
bcc label	branch if carry clear (branch if C = 0)
bmi label	branch if minus (branch if N = 1)
bpl label	branch if plus (branch if N = 0)
bvs label	branch if overflow set (branch if V = 1)
bvc label	branch if overflow clear (branch if V = 0)

Figure 21. HSCS12's simple branch instructions.

Example 22. Implement the partially drawn flowchart shown in Figure 22:

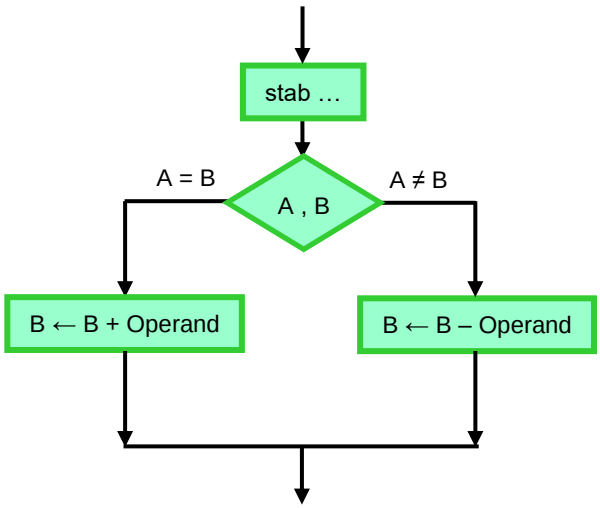


Figure 22. Flowchart for Example 22.

To implement this flowchart, either the beq or bne instruction may be used; however, they work correctly only if the Z flag has already been set to the correct value, such as 1 if A = B and 0 if A ≠ B. (Remember that the condition for these two instructions to branch is Z = 1 or 0, respectively, as shown in Figure 21.) In this example, let us use bne. How can we place the correct value into the Z flag to execute this instruction correctly? One way to do so is to execute an sba instruction before the bne executes. The sba instruction subtracts B from A and puts the difference in A while the Z flag (along with the NVC flags) is affected meaningfully. This will enable the bne to make the right decision. The problem with using a subtract

instruction is that it overwrites the destination register (register A in this example), while we may not want to lose this value. To avoid the problem, we use a *compare* instruction, which performs subtraction affecting the NZVC flags meaningfully (and similar to the subtract instruction); however, the difference is not saved anywhere, and therefore, no data is lost. The HCS12's compare instructions are shown in Figure 23, where the Operand is specified by the addressing modes used in the subtract instructions. The compare instructions affect the flags meaningfully and based on the subtraction that is carried out.

Compare Instruction Syntax	Semantic
cba	Compare A to B, (A – B)
cmpa Operand[8]	Compare A to Operand, (A – Operand)
cmpb Operand[8]	Compare B to Operand, (B – Operand)
cpd Operand[8]	Compare D to Operand, (D – Operand)
cps Operand[8]	Compare SP to Operand, (SP – Operand)
cpx Operand[8]	Compare X to Operand, (X – Operand)
cpy Operand[16]	Compare Y to Operand, (Y – Operand)
The "Operand" is specified by the addressing modes used in the load instructions.	

Figure 23. HSCS12's compare instructions.

Example 22 (cont'd): Using a compare instruction, we now implement the flowchart of Figure 22:

```
        cba             ; compare A and B: Z flag is set if A = B; otherwise, it is reset
        bne     subt    ; go to subt if Z = 0 (if A ≠ B)
        addb    $2000   ; otherwise (A = B), do addition
        bra     done    ; we are done
subt:   subb    $2000   ; do subtraction
done: …
```

The bne interpretation: *bne says "branch if Z = 0" according to the CPU's logic shown in Figure 21, but the Z flag is reset to 0 (by the cba instruction) only if A – B ≠ 0 or A ≠ B (this is the logic that controls the Z flag); therefore, bne says "branch if A ≠ B".*

Similarly, we may interpret the beq instruction that immediately follows a cba instruction:

The beq interpretation: *beq says "branch if Z = 1" according to the CPU's logic shown in Figure 21, but the Z flag is set to 1 (by the cba instruction) only if A – B = 0 or A = B (this is the logic that controls the Z flag); therefore, beq says "branch if A = B".*

The rest of the simple branch instructions (shown in Figure 21) have a more straightforward interpretation. For example, *a bcs that immediately follows a cba is successful only if the subtraction {A – B} results in borrow (C flag = 1).*

Branch instruction in machine language

Branch instructions use the familiar *indexed* addressing mode, now called *relative* addressing mode (REL for short), **where the index register is always the PC**. The branch instructions that you learned above use an 8-bit *signed constant* offset. (Coming up … complex branch instructions, some use a variable offset, some use a longer offset.) Considering the offset width (8 bits), the offset range will be -128 through +127. **Note** that the offset is relative to the *updated* PC, which is the **address of the instruction physically located right after the branch instruction**. The following two examples should clarify this addressing mode:

Example 23. A partially written assembly code (with one beq instruction) and its machine code sitting in the memory starting at $4000 are shown in Figure 24. The Machine Coding column (of the reference manual) shows the syntax of the branch object code: {27 rr}, where $27 is the opcode and rr is a placeholder for the 8-bit signed offset. Instruction {stx …} is labeled done. This label is used in the {beq done} instruction to tell the assembler which instruction must be executed after the branch instruction if the branch happens to be successful:

```
        beq  done        ; 2 bytes
        ldd  $12FF, X    ; 4 bytes
        ldy  #$F0        ; 3 bytes
done:   stx  …

Offset = 4009 – 4002 = +7
```

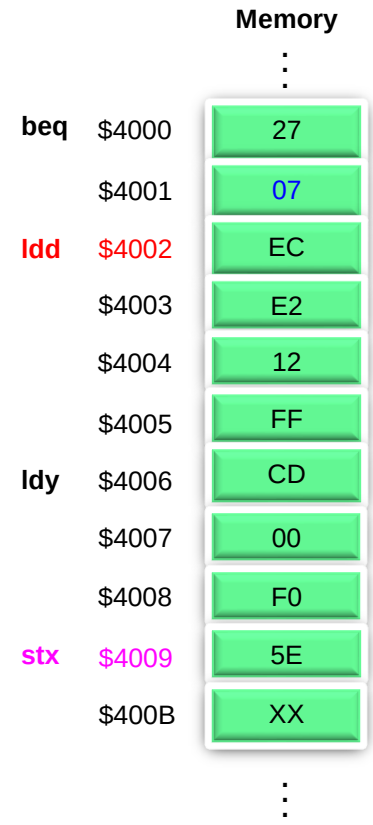


Figure 24. Offset calculation for Example 23.

We need to see how far stx, the destination, is from ldd, the instruction located right *after* the branch instruction. In other words, we need to calculate the distance between the ldd and stx instructions. As you see in Figure 24, the instruction stx is placed at address 4009. Therefore,

Offset = 4009 – 4002 = +7

To calculate the offset, note that we do not need the *exact* address of the two ldd and stx instructions; we only need to know how many bytes they are apart from each other: ldd is 4 bytes long, and ldy is 3 bytes long; therefore, the offset should be 4 + 3 = 7. Is this '7' positive or negative? Since the branch is *forward*, the PC will *increase* when this jump happens; therefore, the offset is +7. The offset would be negative if the jump was backward.

Example 24. A partially written code (with two branch instructions) along with its machine code sitting in the memory starting at $4000 is shown in Figure 24. Offset calculation is also presented in this figure:

```
        clra            ;
        ldab $2000      ;
        beq  done       ; 2 bytes
loop:   aba             ; 2 bytes
        decb            ; 1 byte
        bne  loop       ; 2 bytes
done:   staa $2100      ; 3 bytes
```

Offset1 (for beq) = 4007 – 4002 = **+5**

Offset2 (for bne) = 4002 – 4007 = **-5 = $FB**

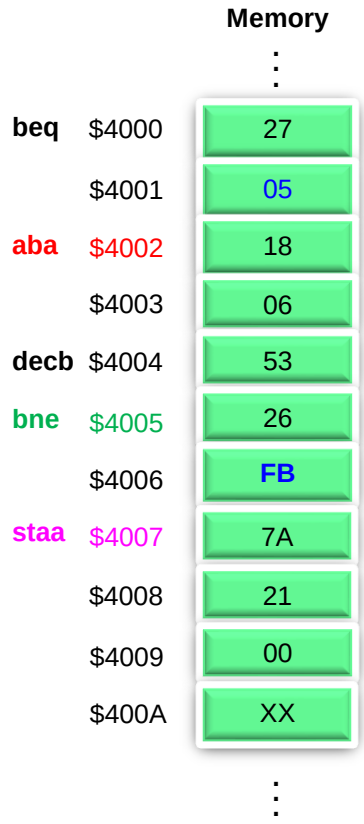


Figure 25. Offset calculation for Example 24.

Again, we do not need the exact addresses. What we need is how many bytes are there between staa and aba:

To calculate the offset for the beq we need to know:

aba is 2 bytes long.

decb is 1 byte long.

bne is 2 bytes long.

The offset for beq = **2** + 1 + **2** = 5. This is a *positive* 5 as beq is a *forward* jump.

Similarly, the offset for the bne is calculated as follows:

The offset for the bne = **2** + 1 + **2** = 5. This is a *negative* 5 as bne is a *backward* jump.

Note that the offset for **bne** is relative to instruction **staa**, the instruction located right after **bne**.

Unsinged relational branch instructions

It should be a better idea if we called the two beq and bne instructions bzs (branch if zero flag set) and bzc (branch if zero flag clear), respectively, similar to other simple conditional branches such as bvs (branch if overflow flag set) and bvc (branch if overflow flag clear). When we call those two instructions beq and bne, they fall into another category called *relational* branches, which jump or do not jump based on the result of a *comparison* between two operands (numbers). The group of relational branch instructions has more members, as explained in this section:

Example 25. Implement the partially drawn flowchart shown in Figure 26:

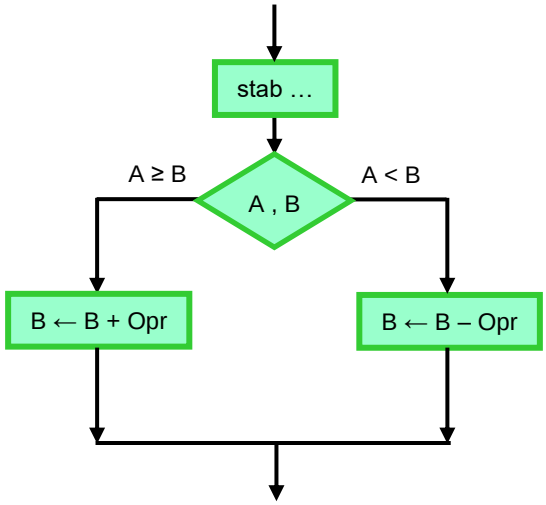


Figure 26. Flowchart for Example 25.

Now the branch condition is not '=' or '≠'; it is < or ≥, meaning we need a different conditional branch instruction. The list of the HCS12 relational conditional branch instructions (which cover all the possibilities) and the necessary and sufficient conditions to branch are shown in Figure 27. Note that these mnemonics are used for *unsigned* comparisons. The *signed* versions will be covered later:

Syntax of unsigned relational conditional branch instruction	Mnemonics spelled out	Semantic Necessary and sufficient condition to branch
beq label	Branch if equal =	Branch if Z = 1
bne label	Branch if not equal ≠	Branch if Z = 0
bhi label	Branch if higher >	Branch if C + Z = 0
bhs label	Branch if higher or same ≥	Branch if C = 0
blo label	Branch if lower <	Branch if C = 1
bls label	Branch if lower or same ≤	Branch if C + Z = 1

Figure 27. Logic implemented in the CPU: necessary and sufficient conditions for HSCS12's unsigned relational branch instructions to branch.

As shown in this figure, for each branch instruction, the necessary and sufficient condition to branch (implemented in the hardware of the CPU) is determined by the Z C flags.

Question: *To branch, why did the CPU designers choose the necessary and sufficient conditions shown in Figure 27?*

Answer: As shown in Homework 01, when two unsigned arbitrary numbers are compared (or subtracted), the values of the Z C flags will be what is shown in Figure 28 for different scenarios, namely, A > B, A ≥ B, A < B, A ≤ B. For example, when A – B is carried out, A > B if C + Z = 0, and vice versa, if C + Z = 0, then A > B. As you see, the necessary and sufficient conditions that the CPU designers chose in Figure 27 are based on the Z C flag values generated by a cba (compare A, B) instruction, as shown in Figure 28:

Z C flag values after A – B is carried out. Necessary and sufficient conditions for the conclusion.	**Conclusion (A and B are unsigned.)**
C + Z = 0 (after cba)	A > B
C = 0, Z is a don't care (after cba)	A ≥ B
C = 1 (after cba), then Z has to be a 0	A < B
C + Z = 1 (after cba)	A ≤ B
Note: C flag shows active-high borrow.	

Figure 28. ZC flag values generated by a cba instruction, and the conclusion based on the values.

Figure 28 shows the traditional truth table to generate the five output signals, A=B, A<B, A≤ B, A>B, and A≥B:

After A – B is performed. (A and B are unsigned.)		**Active-high Outputs**				
Z	**C**	A=B	A<B	A≤ B	A>B	A≥B
0	0	0	0	0	1	1
0	1	0	1	1	0	0
1	0	1	0	1	0	1
1	1	X	X	X	X	X

Figure 29. Traditional truth table to generate unsigned comparison results.

If the two tables in Figure 27 and Figure 28 were combined, the table shown in Figure 30 would be obtained, which is what we were looking for:

cba followed by an unsigned branch instruction.	Net effect (A and B are unsigned.)
cba bhi label	branch if A > B
cba bhs label	branch if A ≥ B
cba blo label	branch if A < B
cba bls label	branch if A ≤ B

Figure 30. HSCS12's unsigned relational branch instructions following a cba and their net effects.

Remember: The decisions made by bhi, bhs, blo, and bls are based on an *unsigned* comparison, which tells us that FFFF > 0FFF, as an example. Soon, you will see how to look at the two numbers (A and B) as *signed* numbers.

Example 25 (Cont'd): Here a blo instruction works:

```
        cba             ; compare A, B. It sets flags accordingly
        blo     subt    ; jump to subt if A < B
        addb    $2000   ; otherwise do addition
        bra     done    ; we are done
subt:   subb    $2000   ; do subtraction
done:   …
```

Summary of blo: *blo says "branch if C = 1" according to the CPU's logic shown in Figure 27; but the C flag is set to 1 by the cba instruction only if A < B according to the facts in Figure 28; therefore, blo says "branch if A < B".*

Similarly, we can summarize the other 3 relational branch instructions as follows:

Summary of bls: *bls says "branch if C + Z = 1" according to the CPU's logic shown in Figure 27; but C + Z becomes 1 by the cba instruction only if A ≤ B according to the facts in Figure 28; therefore, bls says "branch if A ≤ B".*

Summary of bhi: *bhi says "branch if C + Z = 0" according to the CPU's logic shown in Figure 27; but C + Z becomes 0 by the cba instruction only if A > B according to the facts in Figure 28; therefore, bls says "branch if A > B".*

Summary of bhs: *bhs says "branch if C = 0" according to the CPU's logic shown in Figure 27; but C becomes 0 (while Z is a don't care) by the cba instruction only if A ≥ B according to the facts in Figure 28; therefore, bls says "branch if A ≥ B".*

Example 26. Let A = 7 and B = 4 when the following two instructions execute:

```
cba      ; compare A to B, or do A – B to set the ZC flags accordingly
bls again ; branch if A ≤ B
```

Question: How does the CPU know whether or not the bls instruction is successful?

Answer: When the CPU executes the cba instruction, it calculates A – B = 7 – 4 = 3, and sets the ZC flags accordingly, i.e., ZC ← 00. The CPU knows that (according to the table of Figure 28) the necessary and sufficient condition for a successful bls is C + Z = 1. So, the CPU calculates C + Z (while executing the bls) and notices C + Z = 0 (remember, ZC = 00 after the cba executes), concluding that the branch condition is not met; i.e., the branch is unsuccessful. **Note that this answer has already been summarized in the table of Figure 30.**

In the above discussion, we assumed that A and B are the contents of registers A and B, respectively. However, we may use any operand pairs shown in Figure 23.

Signed relational branch instructions

In the previous section, the *unsigned* relational branch instructions were introduced. You will learn the *signed* version of these instructions in this section. The HCS12's signed relational branch instructions (which cover all the possibilities) and the necessary and sufficient conditions to branch are shown in Figure 31. None of these branch instructions affect the CCR.

Syntax of signed relational conditional branch instruction	Mnemonics spelled out	Semantic Necessary and sufficient condition to branch
bgt label	Branch if greater	Branch if N XOR V + Z = 0
bge label	Branch if greater than or equal	Branch if N XOR V = 0
blt label	Branch if less than	Branch if N XOR V = 1
ble label	Branch if less than or equal	Branch if N XOR V + Z = 1

Figure 31. Logic implemented in the CPU: necessary and sufficient conditions for HSCS12's bgt, bge, blt, and ble instructions to branch.

As shown in this figure, for each branch instruction, the necessary and sufficient condition to branch (implemented in the CPU hardware) is determined by the NZV flags.

Question: *To branch, why did the CPU designers choose the necessary and sufficient conditions shown in Figure 31?*

Answer: As shown in Homework 01, when two arbitrary *signed numbers* are compared (or subtracted), the values of the NZV flags will be what is shown in Figure 32 for different scenarios, namely, A > B, A ≥ B, A < B, A ≤ B. For example, when A – B is carried out, A > B if N XOR V + Z = 0, and vice versa if N XOR V + Z = 0, then A > B. As you see, the necessary and sufficient conditions that the CPU designers chose in Figure 31 are based on the NZV flag values generated by a cba instruction, as shown in Figure 32:

ZC flag values after A – B is carried out. Necessary and sufficient conditions for the conclusion.	→ Conclusion (A and B are signed.)
N XOR V + Z = 0 (after cba)	A > B
N XOR V = 0 (after cba)	A ≥ B
N XOR V = 1 (after cba)	A < B
N XOR V + Z = 1 (after cba)	A ≤ B
Note: C flag shows active-high borrow.	

Figure 32. NZV flag values generated by a cba instruction.

Figure 33 shows the regular truth table to generate the five output signals, A=B, A<B, A≤ B, A>B, and A≥B:

After A – B is performed:			Active-high outputs				
N	Z	V	A=B	A<B	A≤B	A>B	A≥B
0	0	0	0	0	0	1	1
0	0	1	0	1	1	0	0
0	1	0	1	0	1	0	1
0	1	1	X	X	X	X	X
1	0	0	0	1	1	0	0
1	0	1	0	0	0	1	1
1	1	X	X	X	X	X	X

Figure 33. Truth table to generate signed comparison results.

If the two tables in Figure 31 and Figure 32 were combined, the table shown in Figure 34 would be obtained, which is what we were looking for:

cba followed by a signed branch instruction	Net effect (A and B are signed.)
cba bgt label	branch if A > B
cba bge label	branch if A ≥ B
cba blt label	branch if A < B
cba ble label	branch if A ≤ B

Figure 34. HSCS12's unsigned relational branch instructions following a cba (on the left) and their net effects (on the right).

Remember: The decisions made by bgt, bge, blt, and ble are based on a *signed* comparison, which tells us FFFF < 0FFF, as an example.

Example 27. Let A = $A002 and B = 1 when the following two instructions execute:

```
cba        ; compare A to B, or do A – B. This will set the NZV flags meaningfully: NZV = 100
ble again  ; branch if A ≤ B.
```

Question: How does the CPU know if the ble instruction is successful?

Answer: When the CPU executes the cba instruction, it calculates A – B = $A002 – 1 = $A001, and sets the flags accordingly, i.e., NZV ← 1 0 0. The CPU knows that (according to the table of Figure 31) the necessary and sufficient condition for a successful ble is N XOR V + Z = 1. So, the CPU calculates N XOR V + Z (while executing the ble) and notices N XOR V + Z = 1 (remember, NZV = 1 0 0 after the cba executes), concluding that the branch condition is met; i.e., the branch is successful. **Note that this answer has already been summarized in the table of Figure 34.**

Note: In the above discussion, we assumed that A and B are the contents of registers A and B, respectively. However, we may use any operand pairs shown in Figure 23.

Example 28. Implement the partially drawn flowchart shown in Figure 26, but now interpret the values of A and B as *signed* numbers:

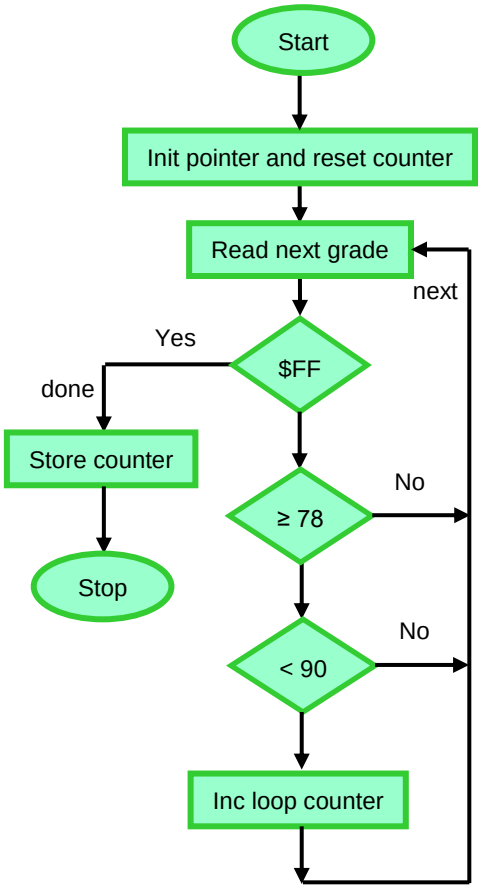
```
        cba             ; compare A, B. This sets the flags accordingly
        blt     subt    ; go to subt if A < B
        addb    $2000   ; otherwise, do addition
        bra     done    ; we are done
subt:   subb    $2000   ; do subtraction
done:   …
```

Summary of blt: *blt says "branch if N XOR V = 1" according to CPU's logic shown in Figure 31, but N XOR V is pulled up by the cba instruction only if A < B according to the facts in Figure 32; therefore, blt says "branch if A < B".*

Similarly, we can summarize the other three relational branch instructions shown in Figure 31.

Example 29. Students' grades between 0 to 100 are placed in the memory starting at $3001. The array ends with an $FF. Determine the number of students with a grade B (78 ≤ B < 90), and put it in the memory at $3000.

Figure 35 shows a flowchart and the associated assembly program for this problem:




```
      ldx  #$3001     ; initialize pointer
      clrb            ; reset counter (B)
next:
      ldaa  1, X+     ; read next grade
      cmpa   #$FF     ; check if end of array
      beq   done      ; reached FF? Then we are done
      cmpa  #78       ; otherwise, check if < min
      blo    next     ; if it is, skip
      cmpa   #90      ; otherwise, check if > max
      bhs    next     ; if it is, skip
      incb            ; otherwise, it is grade B; Inc counter
      bra    next     ; read next grade
done:
      stab $3000      ; save number of Bs
```

Figure 35. Flowchart and assembly program for Example 29.

Example 30. A different version of the flowchart and the associated assembly program for Example 29 are shown in Figure 36. Go over them rigorously.

Exercise: Compare the two versions.

Object code for relational branch instructions

Here is the list of all the branch instructions that we have studied so far:

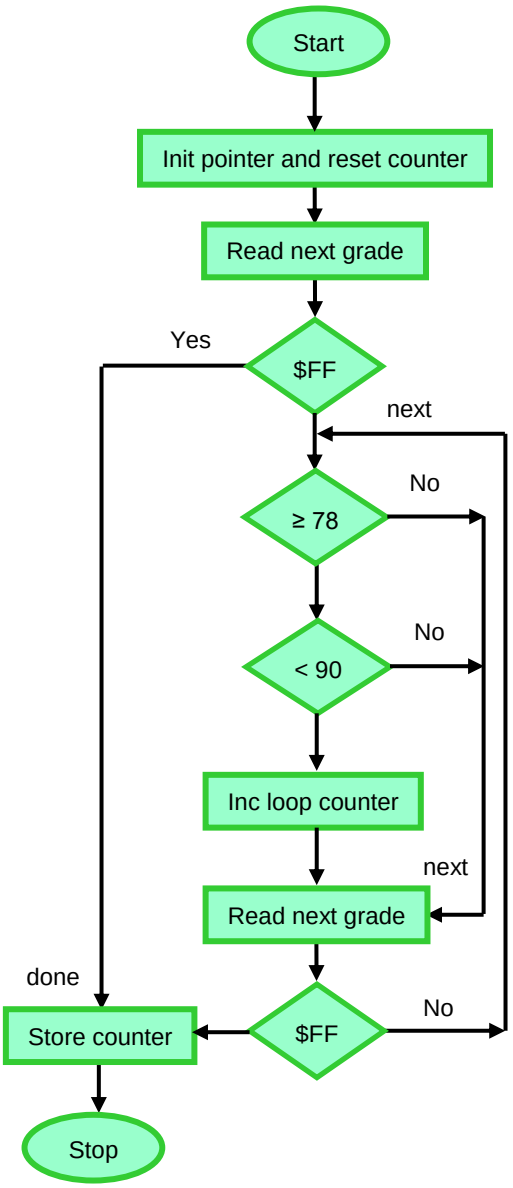
bmi, bpl, beq, bne, bcs, bcc, bvs, bvc, blo, bls, bhi, bhs blt, ble, bgt, bge

All these branch instructions share the following format in machine language:

Opcode followed by an 8-bit signed offset.

Rule 2 tells us how to determine the opcode.

The offset is calculated similar to the simple branch instructions. So, the offset is still relative to the *updated* PC, which is the address of the instruction physically located right after the branch instruction. See Example 23 and Example 24.




```
        ldx    #$3001   ; initialize pointer
        clrb            ; reset counter (B)
        ldaa   1, X+    ; read next grade
        cmpa   #$FF     ; check if end of array
        beq    done     ; reached FF? Then we are done
next:
        cmpa   #78      ; otherwise, check if < min
        blo    next     ; if it is, skip
        cmpa   #90      ; otherwise, check if > max
        bhs    next     ; if it is, skip
        incb            ; otherwise, it is a grade B; Inc counter
        ldaa   1, X+    ; read next grade
        cmpa   #$FF     ; check if end of array
        bne    next
done:
        stab   $3000    ; save number of Bs

        next:
```

Figure 36. Faster flowchart and assembly program for Example 29.

Location Counter

How does the assembler calculate the offset of branch instructions? Consider the code shown in Figure 37:

Address	Instruction	Instruction size (in terms of bytes)
$4000	beq **done**	2
$4002	ldd $12FF, X	4
$4006	ldy #$F0	3
$4009	**done**: stx ...	

Figure 37. Arbitrary code to calculate the branch offset.

The assembler uses a *software* counter called the *Location Counter* (LC) and initializes this counter (let us say to an arbitrary address for now) and assigns this address (say $4000) to the first instruction as shown in Figure 37. Once the assembler translates the first instruction, increments the Location Counter by the number of the bytes of this instruction. This way the assembler determines the address of the next instruction. The assembler continues this process to determine the address of each instruction in the program. Once each instruction has received an address, the assembler can calculate the distance between any two instructions, including the instruction located right after every branch instruction and the instruction that the branch may jump to. As an example, the assembler calculates the offset of the branch instruction used in Figure 37 by subtracting $4002 (the address of the instruction located right after the branch) from $4009 (the address of the instruction that the branch instruction may jump to):

Offset = $4009 – $4002 = +7

Following a similar mechanism, the Location Counter also helps the assembler assign an address to each data. Soon, we will see some examples to better understand the role of the Location Counter.

Assembler Directives

Unlike machine instructions, assembler directives are commands to the assembler, NOT to the CPU. In this section, you will learn different types of assembler directives supported by CodeWarrior:

end marks the *physical* END of the program. Nothing beyond it will be assembled.

Note: The physical END (as opposed to the logical END) of a program is the last line of the program. So, if a program has 20 instructions, the 20th instruction is the physical END of the program. By the logical END of a program, we mean the last instruction that executes before we leave that program. The logical END and the physical END can be the same or different in a program. We will see more about it in an upcoming chapter in which "subroutines" are covered.

org (origin)

The org directive initializes the Location Counter, e.g., org $1000 sets the Location Counter to $1000. So, the next instruction or data will be placed at $1000.

dc.b (define constant byte), **fcb** (form constant byte)

These directives assign one or more 8-bit values to memory bytes:

Example 31.

```
org   $3000     ; Location Counter is set to $3000
fcb   $6D       ; $6D will sit at $3000 (when the program is loaded into the memory), and the LC is updated
                ; to $3301.
```

Note that we could also use machine instructions to initialize location $3000:

```
ldaa   #$6D     ; 2 bytes long, 1 clock cycle long
staa   $3000    ; 3 bytes long, 3 clock cycles long
```

Or

```
movb #$6D, $3000      ; Place value $6D in memory at address $3000; 5 bytes and 4 clock cycles long.
                      ; You will see more move instructions in this chapter.
```

But either one wastes 5 bytes of memory and 4 clock cycles of CPU time!

Example 32.

```
org   $3000
dc.b $6D        ; $6D will sit @3000 (when the program is loaded), LC ← $3001
dc.b $29, $84   ; $29 will sit @ 3001 (when the program is loaded), LC ← $3002
                ; $84 will sit @ 3002 (when the program is loaded), LC ← $3003
```

In your program, you may read from the locations initialized above:

```
ldx   $3000     ; X ← $6D29
ldab $3002      ; B ← $84
```

Note: A program called the *Loader* seats data and instructions in the memory. The Loader is implied when CodeWarrior is sued to start the debugger.

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

These directives initialize one or more words.

Example 33.

```
org   $1000           ;LC ← $1000
dc.w $1234            ;$1000 ← $12, $1001 ← $34, LC ← $1002
fdb   $ABCD           ;$1002 ← $AB, $1003 ← $CD, LC ← $1004
```

You may use the initialized words in your program:

```
ldx   $1001           ; X ← $34AB
ldab $1003            ; B ← $CD
```

fcc (form constant characters)

fcc assigns a string of ASCII characters to a byte array. The Location Counter is updated accordingly.

Example 34.

```
org   $1000
fcc   "microcomputersOne"
```

ASCII codes of m, i, c, r, o, c, o, m, p, u, t, e, r, s, O, n, and e are placed at addresses $1000 to $1010; at the end, the Location Counter will be $1011.

After this initialization, the following instruction loads the ASCII code of the letter 'p' into register A:

```
ldaa $1008      ; loads ASCII code of letter p into Register A.
```

ds (define storage), **rmb** (reserve memory byte), **ds.b** (define storage byte)

These directives only reserve (not initialize) a specific number of bytes given as the argument, i.e., they simply increment the Location Counter by a specified number (the argument). The contents of reserved locations are unknown.

Example 35. The following ds changes the Location Counter to $1010.

```
org  $1000
ds   16
```

ds.w (define storage word)

This is the word version of directive ds, i.e., it simply increments the Location Counter by the specified number x 2. The contents of the reserved locations are unknown.

Example 36.
The following ds.w changes the Location Counter to $1040.

```
        org  $1000
buffer: ds.w $20      ; note that one word is 2 bytes long.
```

"buffer" is the symbolic address of the first byte of this array.

equ (equate)

Directive equ assigns a constant to a name:

Example 37.

```
thisYear: equ 2022
```

Every occurrence of thisYear in the program will be replaced with 2022 by the assembler.

equ can make the program more readable. Additionally, constants may also be changed easily by directive equ.

Example 38. In 2025, you do not have to change all the occurrences of 2022 to 2025: just replace the above directive with:

```
thisYear: equ  2025
```

Boolean (or bit condition) branch instructions

There are two mnemonics for this group of conditional branch instructions as follows:

- brclr: Branch if a specific subset of the bits of a memory location are all zeros.
- brset: Branch if a specific subset of the bits of a memory location are all ones.

The above subset is specified by an 8-bit constant called the *mask8*.

The top two rows in Figure 38 show the **semantics** in the algebraic format and the **syntax** of these two conditional branch instructions.

Here is the assembly syntax of the bit condition branch instructions:

Mnemonic (brclr or brset), Mem8, mask8, label

Memory[8] or Memroy8, abbreviated as Mem8, is one byte of memory specified in the following modes:

DIR, EXT, IDX, IDX1, IDX2.

Mask8 is an 8-bit constant showing the condition for the successful branch. More details are coming up …

The label is the symbolic address of the next instruction in case of a successful branch.

CCR is not affected by these branch instructions.

Pay close attention to the semantics: for each mnemonic a bitwise AND operation selects the correct subsets of the memory bytes used as the conditions for successful branches.

The branch instructions specified in the last two rows in Figure 38 are note supported.

Assembly Syntax	Machine Syntax	Semantics
brclr, Mem8, mask8, label	Opcode, Mem8, mask8, offset8	Brach if (Mem8 ● mask8) = 0
brset, Mem8, mask8, label	Opcode, Mem8, mask8, offset8	Brach if (NotOfMem8 ● mask8) = 0
Unsupported branches	---	Functions
Branch if not all specified bits = 0	---	Brach if (Mem8 ● mask8) ≠ 0
Branch if not all specified bits = 1	---	Brach if (NotOfMem8 ● mask8) ≠ 0
● is the Bit-Wise AND operator.		

Figure 38. Bit condition branch instructions.

Note: The bit condition branch instructions may need two offsets in machine language: one for the indexed mode addressing to access data, the other one for the relative addressing mode to access the next instruction when the branch is successful. What we mean by "offset" should be clear from the context.

Here is the **machine format** of the bit condition branch instructions:

Opcode, Mem8, mask8, offset8.

Rule 2 tells us how to obtain the mnemonic.

Mem8 is one byte of memory specified in any of the following modes:

DIR, EXT, IDX, IDX1, IDX2.

Mask8 is an 8-bit constant showing the condition for the successful branch. Coming up more details …

Similar to the previous branch instructions, offset8 is an 8-bit constant showing how far the next instruction is located from the instruction right after the branch instruction if the branch is successful.

Example 39. Figure 39 shows 3 x bit condition branch instructions in assembly and machine formats as an example:

	Auto pre-decrement	Indexed with offset in D	Indexed with a 5-bit constant offset
Assembly	brclr 4, -X, $0A, label	brset D, Y, $F7, label	brclr 2, SP, $49, label
Object	0F2C 0A Offset[8]	0EEE F7 Offset[8]	0F82 49 Offset[8]

Figure 39. Assembly and machine bit condition branches in Example 39.

Example 40. Let us say the content of memory at $3000 is $4A. Is the following branch successful?

```
brclr $3000, $2C, again  ; $2C is the mask.
                         ; "again" is the label of the next instruction in case of a successful branch.
```

Let us see if (Mem8 ● mask8) equals 0: (See Figure 38.)

Mem8● mask8 = $4A BitWiseAND $2C = %0100 1010 BitWiseAND %0010 1100 = %0000 1000 ≠ 0

So, according to the definitions in Figure 38, the branch is unsuccessful.

We may reword the above BitwiseAND condition as follows:

Rule 19. The condition of the brclr instruction is satisfied; in other words, the branch is successful only if all the Mem8 bits associated with logic 1 bits of the mask are zero.

Back to the example, only bits 2, 3, and 5 of the mask are 1. Mem8 = %0100 1010 in which bits 2, 3, and 5 are 0, 1, and 0, respectively. Therefore, the condition is not met; in other words, the branch is unsuccessful according to Rule 19. The branch condition is graphically shown in Figure 40:

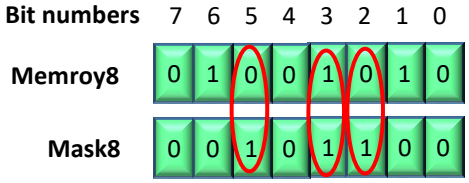


Figure 40. Branch condition in Example 40.

Example 41. Let us say the content of memory at $3000 is $4A. Is the following branch successful or unsuccessful?

```
brclr $3000, $31, again  ; $31 is the mask.
                         ; "again" is the label of the next instruction in case of a successful branch.
```

Let us see if (Mem8 ● mask8) equals 0: (See Figure 38.)

Mem8 ● mask8 = $4A BitWiseAND $31 = %0100 1010 BitWiseAND %0011 0001 = %0000 0000 = 0

So, according to the definitions in Figure 38, the branch is successful.

Use Rule 19: only bits 0, 4, and 5 of the mask are 1. Mem8 = %0100 1010, in which bits 0, 4, and 5 are all 0s. Therefore, the condition is met; in other words, the branch is successful according to Rule 19. The branch condition is graphically shown in Figure 41:

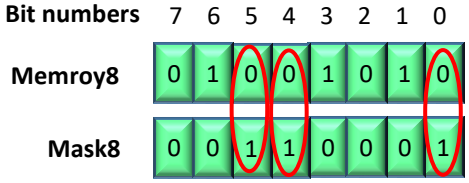


Figure 41. Branch condition in Example 41.

Example 42. Let us say the content of memory at $3000 is $4A. Is the following branch successful or unsuccessful?

```
brset $3000, $32, again  ; $32 is the mask.
                         ; "again" is the label of the next instruction in case of a successful branch.
```

Let us see whether (NotOf Mem8 ● mask8) equals 0 or not:

NotOfMem8 = NOT($4A) = $B5

NotOfMem8 ● mask8 = $B5 BitWiseAND $32 = %1011 0101 BitWiseAND %0011 0010 = %0011 0000 ≠ 0

So, according to the definitions in Figure 38, the branch is unsuccessful.

We may reword the above BitwiseAND condition as follows:

Rule 20. The condition of the brset instruction is satisfied; in other words, the branch is successful only if all the Mem8 bits associated with logic 1 bits of the mask are 1.

Back to the example, only bits 1, 4, and 5 of the mask are 1. Mem8 = %0100 1010, in which bits 1, 4, and 5 are 1, 0, 0, respectively. Therefore, the condition is not met; in other words, the branch is unsuccessful according to Rule 20. The branch condition is graphically shown in Figure 42:

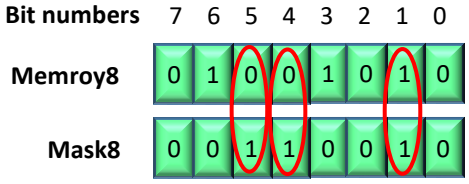


Figure 42. Branch condition in Example 42.

Example 43. Let us say the content of memory at $3000 is $4A. Is the following branch successful?

```
brset $3000, $48, again  ; $48 is the mask.
                         ; "again" is the label of the next instruction in case of a successful branch.
```

Let us see whether (NotOfMem8 ● mask8) equals 0 or not:

NotOfMem8 = NOT($4A) = $B5

NotOfMem8 ● mask8 = $B5 BitWiseAND $A0 = %1011 0101 BitWiseAND %0100 1000 = %0000 0000 = 0

So, according to the definitions in Figure 38, the branch is successful.

Use Rule 20: only bits 3 and 6 of the mask are 1. Mem8 = %0100 1010, in which bits 3 and 6 are 1. Therefore, the condition is met; in other words, the branch is successful according to Rule 20. The branch condition is graphically shown in Figure 43:

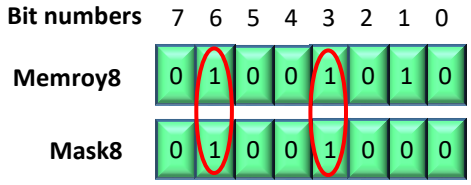


Figure 43. Branch condition in Example 43.

Example 44. Consider the following instruction, which is in the indexed addressing mode with a 5-bit constant offset:

brclr 4, X, 3, loop ; mask = %0000 0011

Let us first translate it into the machine code:

Following Rule 2, the opcode is determined: $0F.

The offset is +4, which fits in 5 bits, so use the following postbyte format:

rr0nnnnn

The base register is X, so the rr field should be 00. And the 5-bit offset is 00100; therefore, the postbyte will be 0000 0100 = $04.

Let us assume the offset is the arbitrary number $06. Therefore, the machine instruction will be $0F040306.

In Figure 48a, the assembled instruction is sitting in the memory, starting at $4000. Let us assume that Figure 44a also shows the values in the relevant memory locations and registers right *before* the instruction executes. The results of instruction execution are shown in Figure 44b.

In this example, the effective address of the data byte to be tested is calculated as follows:

Effective address = base register + offset = $1C3C + 4 = $1C40.

The content of $1C40 = $58, as shown in Figure 44.

The mask is $03, as shown in the instruction.

To execute the instruction, we need to obtain {$58 BitwiseAND $03}, or {%0101 1000 BitwiseAND %0000 0011}:

{%0101 1000 Bitwise-AND %0000 0011} = %0000 0000.

As you see, the result of the BitwiseAND is all zeros, meaning that the condition is met; therefore, the branch is successful.

Use Rule 19: mask8 = %0000 0011, in which only the two LSbs are 1. Mem8 = %0101 1000, in which the two LSb are 00. Therefore, the condition is met; in other words, the branch is successful according to Rule 19.

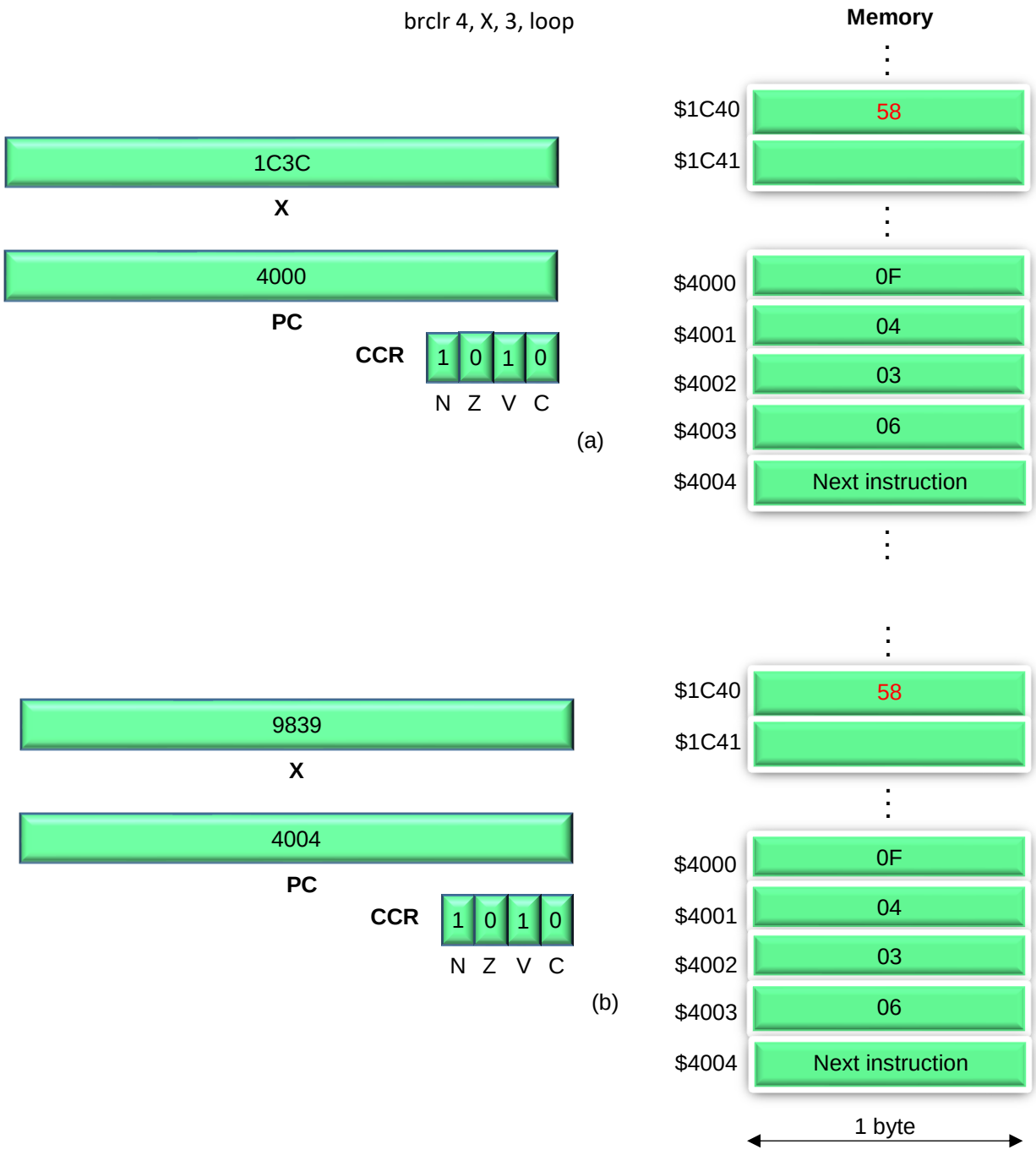


Figure 44. Instruction execution in Example 44.

Example 45. Consider the following instruction, which is in the indexed mode with a 5-bit constant offset:

```
brset 4, X, $28, loop  ; mask = $28
```

Let us first translate it into the machine language:

Following Rule 2, the opcode is determined: $0E.

The offset is +4; therefore, the postbyte will be $04, as obtained in Example 44.

Let us assume the offset is the arbitrary number $06. Therefore, the object code will be $0E0406.

In Figure 45a, the assembled instruction is sitting in the memory, starting at $4000. Let us assume that Figure 45a also shows the values in the relevant memory locations and registers right *before* the instruction executes. The results of instruction execution are shown in Figure 48b.

In this example, the effective address of the data byte to be tested is calculated as follows:

Effective address = base register + offset = $1C3C + 4 = $1C40.

The content of $1C40 = $48, as shown in Figure 45.

The mask is $28, as shown in the instruction.

To execute the instruction, we need to obtain (NOT of $48) BitwiseAND $03, or {%1011 0111 BitwiseAND %0010 1000}:

{%1011 0111 bitwise AND %0010 1000} = %0010 0000 ≠ 0.

As you see, the condition is not met, so the branch is unsuccessful.

Use Rule 20: mask8 = %0010 1000, in which only bits 3 and 5 are 1. Mem8 = %0100 1000, in which bits 3 and 5 all are 1 and 0, respectively. Therefore, the condition is not met; in other words, the branch is unsuccessful according to Rule 20.

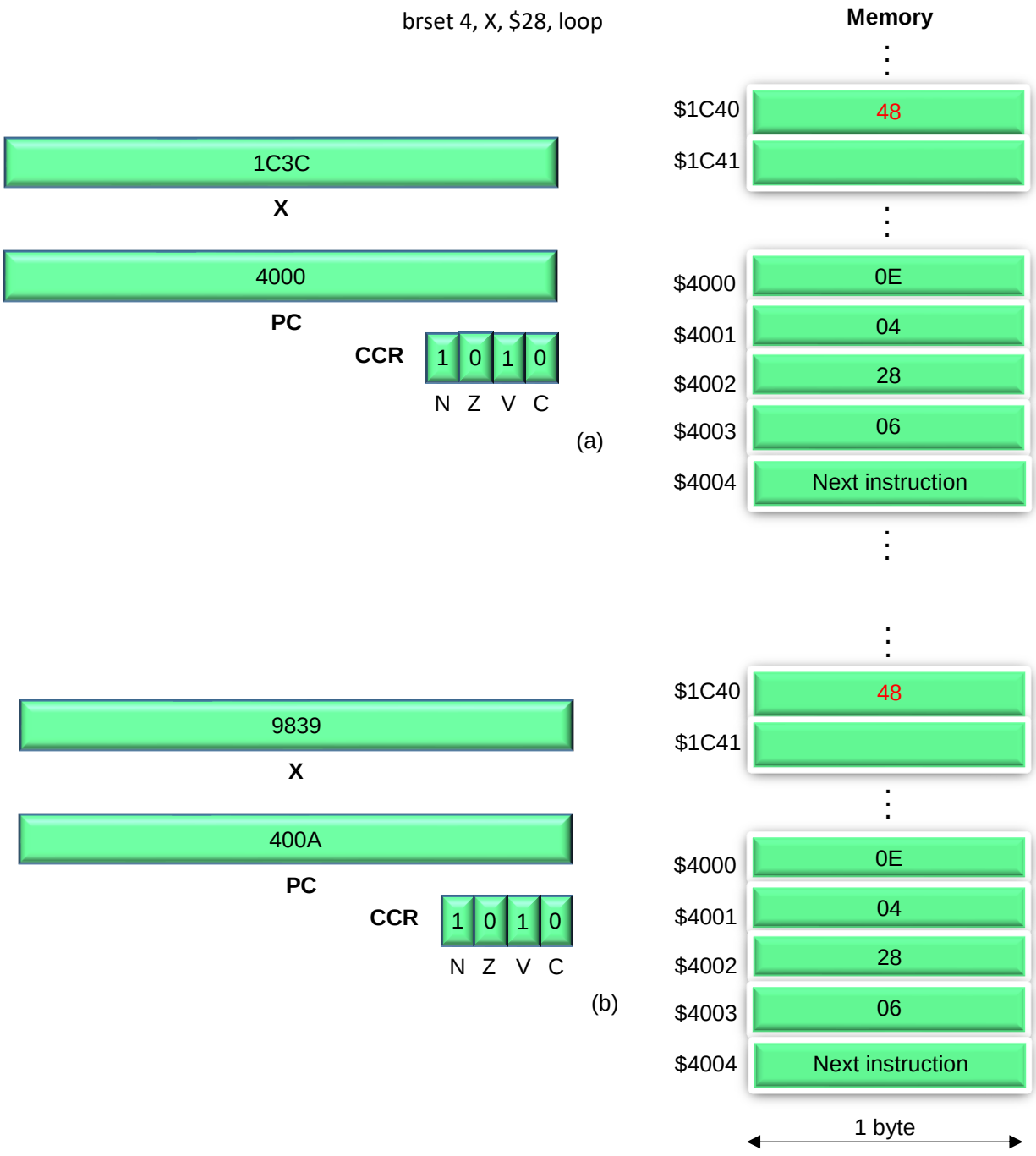


Figure 45. Instruction execution in Example 45.

Loop primitive branch instructions

The loop primitive branch instructions are a combination of three operations: increment or decrement a register-type counter, check the content of the counter, and then branch if the counter is 0 or not 0. So, altogether there are 4 different versions of these instructions, as summarized in the top four rows of Figure 46. The register counter is one of the following:

A, B, D, X, Y, or SP.

Syntax	Semantics
dbeq counter, label	counter ← counter – 1, then branch if counter = 0
dbne counter, label	counter ← counter – 1, then branch if counter ≠ 0
ibeq counter, label	counter ← counter + 1, then branch if counter = 0
ibne counter, label	counter ← counter + 1, then branch if counter ≠ 0
tbeq counter, label	branch if counter = 0
tbne counter, label	branch if counter ≠ 0

Figure 46. Loop primitive conditional branch instructions.

The last two rows in Figure 46 show two more instructions, Test-and-Branch-if-Equal (tbeq) and Test-and-Branch-if-Not-Equal (tbne). They do not update the counter unlike the loop primitive instructions, but still check the counter and perform a conditional branch as shown in this figure.

The above six instructions use relative addressing mode with a 9-bit signed offset. They are 3 bytes wide, where the first byte is the opcode, followed by the loop primitive postbyte, and the 8 LSbs of the offset. **The formant of the postbyte is not covered in this edition of the textbook.** None of these instructions affect the CCR.

Example 46. Write a program to add the elements of an N-byte array. Let us assume that N, the array size, is in location $3300, the array base is in location $3302, and the sum should go to location $3301.

The program is shown here. Go over the it rigorously:

```
        clra            ; reset sum
        ldab    $3300   ; B is loop counter, initialize B to array size
        beq     done    ; check if there is no number to add!
        ldx     #$3302  ; initialize array pointer (X)
loop:   adda    X       ; add next number to sum
        inx             ; update pointer (increment)
        decb            ; update loop counter (decrement)
        bne     loop    ; do it again if loop counter is not 0 yet
done:   staa    $3301   ; otherwise we are done, save sum
```

In the following version of the code, the two highlighted instructions (“adda” and “inx”) are replaced with an auto-post-increment add instruction:

```
        clra            ; reset sum
```

```
        ldab    $3300   ; B is loop counter, initialize B to array size
        beq     done    ; check if there is no number to add!
        ldx     #$3302  ; initialize array pointer (X)
loop:   adda    1, X+   ; add next number to sum
                        ; update pointer (increment)
        decb            ; update loop counter (decrement)
        bne     loop    ; do it again if loop counter is not 0 yet
done:   staa    $3301   ; otherwise we are done, save sum
```

***

In the third version of the code, the two instructions in bold (decb and bne) are replaced with dbne, a loop primitive instruction:

```
        clra            ; reset sum
        ldab    $3300   ; B is loop counter, initialize B to array size
        beq     done    ; check if there is no number to add!
        ldx     #$3302  ; initialize array pointer (X)
loop:   adda    1, X+   ; add next number to sum
                        ; update pointer (increment)
; update loop counter (decrement); if it is not 0 yet, do another iteration
        dbne    B, loop
done:   staa $3301      ; otherwise we are done, save sum
```

***

Comparison: Look at the similarity:

We may combine "Using the Pointer" & "Updating the Pointer":

```
loop: adda      X       ; Use Pointer
      inx               ; Update Pointer
```

The above two instructions can be replaced with this one:

```
loop: adda      1, X+   ; Use & Update combined
```

We may also Combine "Using the Loop Counter" and "Updating the Loop Counter":

```
      decb              ; Update loop counter
      bne       loop    ; Use loop counter
```

The above two instructions can be replaced with this one:

```
      dbne      B, loop ; Update & Use combined
```

Multiplication

The HCS12 has three different multiplication instructions: emuls, a 16 x 16-bit signed multiplication; emul, a 16 x 16-bit unsigned multiplication; and mul, an 8 x 8-bit unsigned multiplication. Only register-type operands and register-type results may participate in multiplication. Additionally, the registers are specific, as you will see shortly. Figure 47 shows the type, size, operand registers and the result registers of the three instructions and how the NZVC flags are affected by them. None of these instructions affect the V flag. The N and Z flags are affected meaningfully, i.e., the Z flag is set to 1 if the product is 0; otherwise, it is reset to 0. Also, the MSb of the product is copied to the N flag, whether the multiplication is signed or unsinged. The MSb of the lower half of the product is copied to the C flag, which can be used to round the lower half of the product. In the two 16-bit multiplications, the upper half of the product goes to register Y and the lower half goes to register D.

Mnemonic (entire instruction)	Sign and size	Operands and results	NZVC
emuls	Signed 16 x 16	Y & D ← Y x D	N ← MSb of product Z ← 1 if product = 0, 0 otherwise V : No change C ← MSb of lower half of product
emul	Unsigned 16 x 16	Y & D ← Y x D	
mul	Unsigned 8 x 8	D ← A x B	NZV : No change C ← MSb of lower half of product

"&" is used as the concatenation operator.

Figure 47. HCS12's multiplication instructions.

Example 47. Signed multiplication: Let us say Y = $FFFF = -1 and D = $FFFF = -1. Determine the contents of Y and D after instruction emuls executes:

-1 x -1 = +1, so: the product would be 0000 0001, i.e., Y = $0000, D = $0001.

N = 0 MSb of the product

Z = 0

V = NC (No change)

C = 0 MSb of the lower half of the product

Example 48. Unsigned multiplication: Let us say Y = $FFFF = 65535 and D = $FFFF = 65535. Determine the contents of Y and D after instruction emul executes:

$FFFF x $FFFF = $FFFF x ($10000 - 1) = $FFFF 0000 - $0000 FFFF = $FFFF 0000 + ($FFFF 0000 + 1) = 2 x $FFFF 0000 + 1 = $FFFE 0000 + 1 = $FFFF 0001, so:

Y = $FFFE, D = $0001

N = 1 MSb of the product

Z = 0

V = NC (No change)

C = 0 MSb of the lower half of the product

Division

Only register-type operands and register-type results may participate in division. Additionally, the source/destination registers are specific, as you will see next. Figure 48 shows the type, size, operand registers and the result registers of the 5 different division instructions supported by the HCS12:

Mnemonic (entire instruction)	Sign and size	Operands and results
ediv	Unsigned 32 ÷ 16	Y & D ÷ X Y ← Quotient D ← Remainder
edivs	Signed 32 ÷ 16	Y & D ÷ X Y ← Quotient D ← Remainder
fdiv dividend < divisor	Unsigned Fractional 16 ÷ 16	D ÷ X X ← Quotient D ← Remainder
idiv	Unsigned 16 ÷ 16	D ÷ X X ← Quotient D ← Remainder
idivs	Signed 16 ÷ 16	D ÷ X X ← Quotient D ← Remainder
"&" is used as the concatenation operator.		

Figure 48. HCS12 division instructions.

Figure 49 shows how the NZVC flags are affected by the division instructions[3]:

[3] The CPU reference manual uses the term *undefined* for three flags in some special cases; however, the author of this manuscript believes that what the manual means is *unknown* (not undefined).

Mnemonic	N flag	Z flag	V flag	C flag
ediv **Unsigned 32 ÷ 16**	Becomes the MSb of the quotient.	Set to 1 only if the quotient is $0000.	Set to 1 only if the quotient is wider than 16 bits.	Set to 1 only if the divisor is $0000.
Special cases	Undefined in case of overflow or divide by 0.	Undefined in case of overflow or divide by 0.	Undefined in case of divide by 0.	None
edivs **Signed 32 ÷ 16**	Becomes the MSb of the quotient.	Set to 1 only if the quotient is $0000.	Set to 1 only if the quotient is out of range. Note: quotient is signed.	Set to 1 only if divisor is $0000.
Special cases	Undefined in case of overflow or divide by 0.	Undefined in case of overflow or divide by 0.	Undefined in case of division by 0.	None
fdiv **Unsigned** **Fractional** **16 ÷ 16**	Remains unchanged	Set to 1 if quotient is $0000; otherwise, it is reset to 0.	Set to 1 only if Denominator X ≤ Numerator D.	Set to 1 only if denominator is $0000.
Special cases	None	None	None	None
idiv **Unsigned 16 ÷ 16**	Remains unchanged	Set to 1 only if the quotient is $0000.	Reset to 0.	Set to 1 only if the denominator is $0000.
Special cases	None	None	None	None
Idivs **Signed 16 ÷ 16**	Becomes the MSb of the quotient.	Set to 1 only if the quotient is $0000.	Set to 1 only if the quotient is out of range. Note: quotient is signed.	Set to 1 only if the denominator is $0000.
Special cases	Undefined in case of overflow or divide by 0.	Undefined in case of overflow or divide by 0.	undefined in case of divide by 0.	None

Figure 49. How the flags are affected by division instructions.

Figure 50 shows five examples of the five divide instructions:

*Instruction fdiv deserves special attention: Let us say D = 3 and X = 9. Then execute fdiv:

Remainder D = 3, which means D = $0.0003 = 0.0000 0000 0000 0011 = 3 x 16 **-4

Quotient X = $5555, which means X = $0.5555 = 0.3333282470703125 decimal

Instruction	Inputs			Outputs							Comments
	Y	D	X	Y	D	X	N	Z	V	C	
ediv	FFFF	FFFF	FFFF	FFFF	FFFF	--	UD	UD	1	0	Y & D ÷ X Y ← Quotient (Correct quotient = $1001, out of range) D ← Remainder
edivs	FFFF	FFFF	FFFF	+1	0	--	0	0	0	0	Y & D ÷ X (dividend = -1, divisor = -1) Y ← Quotient D ← Remainder
fdiv*	X	3	9	X	3 **Means: $0.0003**	5555 **Means: $0.5555**	NC	0	0	0	D ÷ X X ← Quotient D ← Remainder
idiv	X	FFFF	A	X	5	1999	NC	0	R	0	D ÷ X X ← Quotient D ← Remainder
idivs	X	FFFF	A		FFFF	0	0	1	0	0	D ÷ X X ← Quotient D ← Remainder
Legend **&**: concatenation operator; **UD**: unedified; Legend: **NC**: No change; **R**: reset unconditionally.											

Figure 50. Five examples of five divide instructions.

Verify:

3 = 0.3333282470703125 x 9 + 3 x 16 **-4 = 3

N remains unchanged

Z = 0, it is set to 1 only if the quotient is $0000.

V = 0, it set to 1 only if Denominator X ≤ Numerator D.

C = 0, it is set to 1 only if denominator is $0000.

Shift instructions

Figure 51 shows all the shift instructions supported by the HCS12 microcontroller. Multiple shifts are not supported, i.e., you may only shift one bit at a time.

<table>
<tr><th colspan="5">Shift Instructions</th></tr>
<tr><td colspan="3">Shift 8-bit register A or B, or memory location</td><td colspan="2">Shift 16-bit register D (only logical)</td></tr>
<tr><td rowspan="2">Shift left
Logical and Arithmetic are similar</td><td colspan="2">Shift right</td><td rowspan="2">Shift left
Logical and Arithmetic are similar</td><td rowspan="2">Shift right
(only logical)</td></tr>
<tr><td>Arithmetic</td><td>Logical</td></tr>
<tr><th colspan="5">Shift Instructions' Mnemonics and Syntax</th></tr>
<tr><td>lsla
lslb
lsl Memory[8]

asla
aslb
asl Memory[8]</td><td>asra
asrb
asr Memory[8]</td><td>lsra
lsrb
lsr Memory[8]</td><td>asld (Arithmetic)
lsld (Logical)</td><td>lsrd</td></tr>
<tr><td colspan="5">All the addressing modes, except for the direct mode, can be used to specify a memory location.</td></tr>
</table>

Figure 51. HCS12's shift instructions Classification.

Eight-bit shift instructions shift register A, register B or an 8-bit memory location (Memory[8]). **Except for the direct mode, all the addressing modes used in the store instructions can be used to specify Memory [8].** The shift and store instructions have the same assembly and machine syntax when they use a memory operand with the same addressing mode.

Example 49. Figure 52 shows 3 pairs of shift/store instructions as an example. The *same addressing mode* is used in each pair. The first byte in the Object row is the opcode:

	Pair 1		Pair 2		Pair 3	
	Shift	**store**	**Shift**	**store**	**Shift**	**store**
Assembly	asl 5, X	staa 5, X	asr 8, -Y	stab 8, -Y	lsr D, SP	staa D, SP
Object	6805	6A05	6768	6B68	64F6	6AF6

Figure 52. Three pairs of instructions in Example 49.

The shift to left instructions shift all the bits to the left by one bit; a 0 goes into the LSb, the MSb goes to the carry flag, and its previous value is lost. There is no difference between the arithmetic shift left and the logical shift left. Figure 53 shows the shift to left instructions graphically:

Figure 53. Figure 1. HCS12's shift to left instructions.

The number is multiplied by 2 when it is shifted to the left. Here is how the shift-to-left instructions affect the NZVC flags:

The **N** flag becomes a copy of the result's MSb. (So, the N flag changes meaningfully)

The **Z** flag is set to 1 only if the result is 0. (So, the Z flag changes meaningfully)

The **V** flag is set to 1 when the sign bit (MSb) toggles. (So, the V flag changes meaningfully)

The **C** flag receives the bit that leaves the operand (register or memory), e.g., if %0100 0000 is shifted to left by two bits, the C flag is set to 1.

The shift to right instructions have two versions: *arithmetic* and *logical*.

The logical shift to right instructions shift all the bits to right by one bit, a 0 goes into the MSb, the LSb goes to the carry flag, and the previous value of the carry flag is lost. Figure 54 shows the logical shift to right instructions graphically:

Figure 54. HCS12's logical shift to right instructions.

When you look at a number as an *unsigned* number, the number is divided by 2 when it is *logically* shifted to *right*, and the remainder goes to the C flag. Here is how the logical shift to right instructions affect the NZVC flags:

The **N** flag becomes a copy of the result's MSb. So, N = 0 after every logical shirt to right. (The N flag changes meaningfully)

The **Z** flag is set to 1 only if the result is 0. (The Z flag alters meaningfully.)

The V flag becomes N XOR C of the result. Since N = 0 after every logical shift to right (see Figure 54), the V flag is always equal to the C flag after every logical shift to right.

The **C** flag receives the bit that leaves the operand (register or memory), e.g., if %0000 0010 is shifted to right by two bits, the C flag is set to 1. In other words, the C flag becomes the remainder of the divide by 2.

The arithmetic shift to right instructions shift all the bits to right by one bit, the sign bit of the original number goes into the MSb of the result (in other words a 1-bit sign extension is performed), the LSb of the original number goes to the carry flag, and the previous value of the carry flag is lost. Figure 55 shows the arithmetic shift to right instructions graphically. Note that the 16-bit shift to right instruction has only the logical version (i.e., there is no 16-bit arithmetic shift to right):

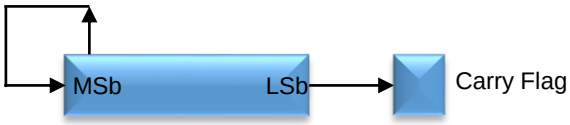


Figure 55. HCS12 arithmetic shift to right instructions.

When you look at a number as a *signed* number, the number is divided by 2 when it is *arithmetically* shifted to *right*, and the remainder goes to the C flag. Here is how the logical shift to right instructions affect the NZVC flags:

The **N** flag becomes a copy of the result's MSb. (The N flag changes meaningfully.)

The **Z** flag is set to 1 only if the result is 0. (The Z flag changes meaningfully.)

The **V** flag becomes N XOR C of the result; in other words, it becomes the XOR of the LSb and MSb of the original number. So, for positive numbers, V = C, and for negative numbers V = C′ after every shift.

The **C** flag receives the bit that leaves the operand (register or memory), e.g., if %0000 0010 is shifted to right by two bits, the C flag is set to 1. In other words, the C flag becomes the remainder of the divide by 2.

Rotate instructions

The rotate instructions rotate the concatenation of the carry flag and an **8-bit number** in either direction. The number can be in an 8-bit register or **an 8-bit memory location specified by the addressing modes used for the store instruction except for the direct mode.** The assembly syntax of the rotate instructions is as follows:

```
rola            ; rotate left register A
rolb            ; rotate left register B
rol Memory[8]   ; rotate left an 8-bit memory location, use any addressing mode used for the store
                ; instruction except for the direct mode.
rora            ; rotate right register A
rorb            ; rotate right register B
ror Memory[8]   ; rotate right an 8-bit memory location, use any addressing mode used for the store
                ; instruction except for the direct mode.
```

The rotate and store instructions have the same assembly and machine syntax when they use a memory operand with the same addressing mode.

Example 50. Figure 56 shows 3 pairs of rotate/store instructions as an example. The *same addressing mode* is used in each pair. The first byte in the Object row is the opcode:

	Pair 1		Pair 2		Pair 3	
	rotate	**store**	**rotate**	**store**	**rotate**	**store**
Assembly	rol 5, X	staa 5, X	ror 8, -Y	stab 8, -Y	rol D, SP	staa D, SP
Object	6505	6A05	6668	6B68	65F6	6AF6

Figure 56. Three pairs of instructions in Example 50.

Figure 57 shows the rotate to left instructions graphically:

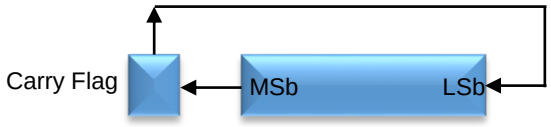


Figure 57. HCS12's rotate to left instruction.

Figure 58 shows the rotate to right instructions graphically:

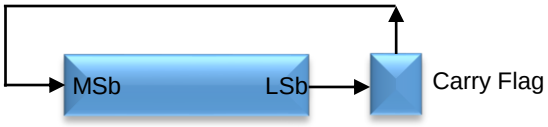


Figure 58. HCS12's rotate to right instruction.

Here are how the rotate instructions affect the NZVC flags**:**

The **N** flag becomes a copy of the result's MSb. (The N flag changes meaningfully.)

The **Z** flag is set to 1 only if the result is 0. (The Z flag changes meaningfully.).

The **V** flag becomes N XOR C of the result.

The **C** flag is part of the rotation as explained above.

Example 51. Write a program to count the number of 0s in the 16-bit number stored at $3000 and save the result in $3005:

Here is how we approach the problem:

Shift the number (to right or left) 16 times. After each shift, check the carry flag:

(**Note:** Since 16-bit rotations are not supported, it is easier to use a shift instruction.)

And here is the code:

```
            ; Data Segment
            org   $3000
            dc.b $23, $55  ; the test data
            org   $3005
zero_cnt:
            dc.b 0          ; zero counter
loop_cnt:
            dc.b 16         ; loop counter

            ; Code Segment
            ; N, the input number, is placed @ $3000
            ldd   $3000          ; place the test day in register D
again:      lsrd                 ; shift R. LSb of D goes to C flag (you may also shift it to left)
            bcs   skip           ; If C flag = 1, skip
            inc   zero_cnt       ; otherwise, update (increment) zero counter
skip:       dec   loop_cnt       ; update (decrement) loop counter
            bne   again          ; if loop counter ≠ 0, do it again
wait:       bra   wait           ; otherwise we're done, stay here!
```

Boolean logic and rese/set instructions

Figure 59 shows a self-explanatory list of the HCS12's Boolean logic instructions, followed by six single-bit reset and set instructions. Go over this table rigorously. The I flag is introduced in Chapter 8.

Mnemonic	Function	Comments
Boolean Logic Instructions		
anda Operand[8]*	A ← A ● Operand[8]	Bit-wise AND A with Operand
andb Operand[8]	B ← B ● Operand[8]	Bit-wise AND B with Operand
andcc Immediate	CCR ← CCR ● Immediate	Bit-wise AND CCR with immediate
oraa Operand[8]	A ← A + Operand[8]	Bit-wise OR A with Operand
orab Operand[8]	B ← B + operand[8]	Bit-wise OR B with Operand
orcc Immediate	CCR ← CCR + Immediate	Bit-wise OR CCR with immediate
eora Operand[8]	A ← A ⊕ Operand[8]	Bit-wise XOR A with Operand
eorb Operand[8]	B ← B ⊕ Operand[8]	Bit-wise XOR B with Operand
com Memory[8]**	Memory[8] ← Not Memory[8]	Bit-wise complement memory
coma	A ← Not A	Bit-wise complement A
comb	B ← Not B	Bit-wise complement B
Reset and set Instructions		
clc	C flag ← 0	Clear C flag, pseudo instruction, replaced with andcc #$FE.
cli	I flag ← 0 ***	Clear I flag, pseudo instruction, replaced with andcc #$EF.
clv	V flag ← 0	Clear V flag, pseudo instruction, replaced with andcc #$FD.
sec	C flag ← 1	Set C flag, pseudo instruction, replaced with orcc #$01.
sei	I flag ← 1 ***	Set I flag, pseudo instruction, replaced with orcc #$10.
sev	V flag ← 1	Set V flag, pseudo instruction, replaced with to ORCC #$02.

* ** *** See the next page for more info.

Figure 59. HCS12's Boolean logic and reset instructions.

*__Operand[8]__ can be specified in any addressing mode used for the load instructions.
** __Memory[8]__ can be specified in any addressing mode used for the store instruction (except for the DIR mode).
*** The __I flag__ is introduced in Chapter 8.

Example 52. Using the {Read, Modify, Write} algorithm:

- Reset the upper 4 bits of the byte at address $2056

```
ldaa    $2056    ; Read
anda    #$0F     ; Modify
staa    $2056    ; Write
```

- Set LSb of the byte at address $2056

```
ldaa    $2056    ; Read
oraa    #$01     ; Modify
staa    $2056    ; Write
```

- Toggle the lower 4 bits of the byte at address $2056

```
ldaa    $2056    ; Read
eora    #$0F     ; Modify
staa    $2056    ; Write
```

***

Here are how the Boolean logic instructions affect the NZVC flags**:**

The N flag becomes identical to the MSb of the result.

The Z flag is set to 1 only if the result is 0.

The V flag is always reset to 0 unconditionally.

The C flag remains unchanged.

The rest and set instructions only affect the flag that they reset or set, respectively.

Operand[8] in Figure 59 can be specified in any addressing mode used for the load instructions. The instructions using Operand[8] in Figure 59 and the load instructions have the same assembly and machine syntax when they use the same addressing mode.

Example 53. Figure 56 shows 3 pairs of logic/load instructions as an example. The *same addressing mode* is used in each pair. The first byte in the Object row is the opcode:

	Pair 1		**Pair 2**		**Pair 3**	
	logic	**load**	**logic**	**load**	**logic**	**load**
Assembly	andb 5, X	ldaa 5, X	eora 8, -Y	ldab 8, -Y	oraa D, SP	ldd D, SP
Object	E405	A605	A868	E668	AAF6	ECF6

Figure 60. Three pairs of instructions in Example 53.

Memory[8] in Figure 59 can be specified in any addressing mode used for the store instructions (except for DIR). The com Memory[8] instruction (shown in Figure 59) and the store instructions have the same assembly and machine syntax when they use the same addressing mode.

Example 54. Figure 61 shows 3 pairs of com/store instructions as an example. The *same addressing mode* is used in each pair. The first byte in the Object row is the opcode:

	Pair 1		Pair 2		Pair 3	
	complement	**store**	**complement**	**store**	**complement**	**store**
Assembly	com 5, X	staa 5, X	com 8, -Y	stab 8, -Y	com D, SP	staa D, SP
Object	6105	6A05	6168	6B68	61F6	6AF6

Figure 61. Three pairs of instructions in Example 54.

Bit test instructions

The HSC12's two bit-test instructions are shown in Figure 62. As explained in this section, these instructions check a subset of the bits of an 8-bit register or a memory location to see if they are all 0s (and more).

Assembly syntax	Operation	Comments
bita Operand[8] (same modes used in load)	A ● Operand[8]	Bit test A
bitb Operand[8] (same modes used in load)	B ● Operand[8]	Bit test B

The N and Z flags are affected meaningfully. The V flag is reset, the C flag is not affected.

Figure 62. HCS12's bit-test instructions.

Instruction bita (bitb) bit-wise ANDs register A (B) with Operand[8] specified in any addressing mode used in the load instructions. **The 8-bit result is not stored anywhere;** however, the flags are affected as follows:

- The MSb of the result is copied into the N flag.
- The Z flag is set to 1 if the result is 0; otherwise, it is reset to 0.
- The V flag is reset to 0 unconditionally.
- The C flag is left unchanged.

The bit-test and the load instructions have the same assembly and machine syntax when they use the same addressing mode.

Example 55. Go over the following two instructions and the comments:

```
bita #$80          ; the N flag is set to 1, and the Z flag is reset to 0 if A < 0; otherwise,
                   ; the N flag is reset to 0, and the Z flag is set to 1.
bitb #%1010 1010   ; the N flag is set to 1 if B < 0; otherwise, it is reset to 0.
                   ; the Z flag is set to 1 if the odd bits of B are all 0s; otherwise, it is reset to 0 or,
                   ; the Z flag is reset to 0 if at least one odd bit of B is 1, otherwise it is set to 1.
```

Note: There is no such a bit-test instruction that checks a subset of the bits of a register or a memory location to see if they are all 1s.

Example 56. Let us say A = $6B = %0110 1011. Then, instruction bita #$80 will calculate {%0110 1011 Bit-wise AND %1000 000}. The result is $00. Therefore, the N flag will be reset to 0, and the Z flag will be set to 1.

Example 57. Let us say B = $B5 = %1011 0101. Then, instruction bitb #%1010 1010 will calculate {%1011 0101 bit-wise AND #%1010 1010}. The result is $A0. Therefore, the N flag will be set to 1, and the Z flag will be reset to 0.

Example 58. Figure 63 shows 3 pairs of bit-test/load instructions as an example. The *same addressing mode* is used in each pair. The first byte in the Object row is the opcode:

	Pair 1		Pair 2		Pair 3	
	bit-test	**load**	**bit-test**	**load**	**bit-test**	**load**
Assembly	bita #$6C	ldaa #$6C	bitb 8, -Y	ldab 8, -Y	bita D, SP	ldd D, SP
Object	856C	**866C**	E568	E668	A5F6	ECF6
The machine and assembly syntax of the bit-test instructions are similar to the load instructions.						

Figure 63. Three pairs of instructions in Example 58.

Bit manipulation instructions

The HCS12 has two bit-manipulation instructions, namely bclr and bset, as shown in Figure 64. The bclr instruction **resets** the subset of the bits of a memory byte (Memory8) that correspond to logic 1s in an 8-bit constant called the *mask8*. The bset instruction **sets** the subset of the bits of a memory byte that correspond to logic 1s in an 8-bit constant called the *mask*. **The memory can be specified in any addressing mode except for the indirect modes.**

Assembly Syntax	Semantics	Comment
bclr Memory8, mask8 (no indirect mode)	Memory8 ← Memory8 ● not mask8	Clear bits in memory8
bset Memory8, mask8 (no indirect mode)	Memory8 ← Memory8 + mask8	Set bits in memory8
The N and Z flags are affected meaningfully. The V flag is reset; the C flag is not affected. The memory can be specified in any addressing mode except for the indirect modes.		

Figure 64. HCS12 bit manipulation instructions.

You may also use Rule 21 and Rule 22 to obtain the results of bclr and bset instructions, respectively:

Rule 21. When a bclr executes, the following bits of the memory location will be reset to 0, and the rest of the bits will remain unchanged:

The bits that correspond to logic 1s in the mask.

Rule 22. When a bset executes, the following bits of the memory location will be set to 1, and the rest of the bits will remain unchanged:

The bits that correspond to logic 1s in the mask.

Example 59. Let us say Mem($3000) = % 1101 1011. Instruction {bclr $3000, %10101010} resets the odd bits of the memory location at $3000 according to Rule 21:

Mask = % 10101010

Initial Mem($3000) = % 1101 1011

Final Mem($3000) = % 0101 0001

You may also use the algebraic description of the operation shown in Figure 64:

Not mask = % 01010101

Mem($3000) ← % 1101 1011 Bitwise-AND % 01010101 = 0101 0001

Example 60. Let us say the memory pointed to by register X = % 1000 1011. Instruction {bset X, %0100 0101} sets the bits 0, 2, and 6 of this memory location according to Rule 21:

Mask = %0100 0101

Initial Mem(X) = % 1000 1011

Final Mem(X) = % 1100 1111 = $CF

You may also use the algebraic description of the operation shown in Figure 64:

Mask = % 0100 0101

Mem(X) ← % 1000 1011 Bitwise-OR % 0100 0101= % 1100 1111 = $CF ◊

The NZVC flags are affected as follows:

- The MSb of the result is copied into the N flag.
- The Z flag is set to 1 if the result is 0, otherwise, it is reset to 0.
- The V flag is reset to 0 unconditionally.
- The C flag is left unchanged.

Here is the format of the bit manipulation instructions in machine language:
Opcode + 8-bit memory (in any mode except for the indirect modes) + 8-bit constant (mask).

Example 61. Figure 65 shows 3 pairs of bit-manipulation/load instructions as an example. The *same addressing mode* is used in each pair. The first byte in the Object row is the opcode:

	Pair 1		Pair 2		Pair 3	
	bit-manipulation	**load**	**bit-manipulation**	**load**	**bit-manipulation**	**load**
Assembly	bclr $3000, $6C	ldaa $3000	bset 8, -Y, $6C	ldab 8, -Y	bclr D, SP, $6C	ldd D, SP
Object	1D30006C	B63000	0C686C	E668	0DF66C	ECF6
For the bit manipulation instructions, the memory can be specified in any addressing mode except for the indirect modes.						

Figure 65. Three pairs of instructions in Example 61.

Example 62. Assemble instruction {bset 4, X, $96}, place it in the memory starting at address $4000, and then execute it.

Following Rule 2, the opcode is looked up: $0C.

The instruction is in the index mode with a 5-bit offset (IDX). So, the postbyte is rr0nnnnn = 00000100 = $04. Therefore, the instruction in machine language is $0C0496.

In Figure 66a, the assembled instruction is sitting in the memory, starting at $4000. Let us assume that Figure 66a also shows the relevant values in the memory and registers *before* the instruction executes. The results of instruction execution are shown in Figure 66b.

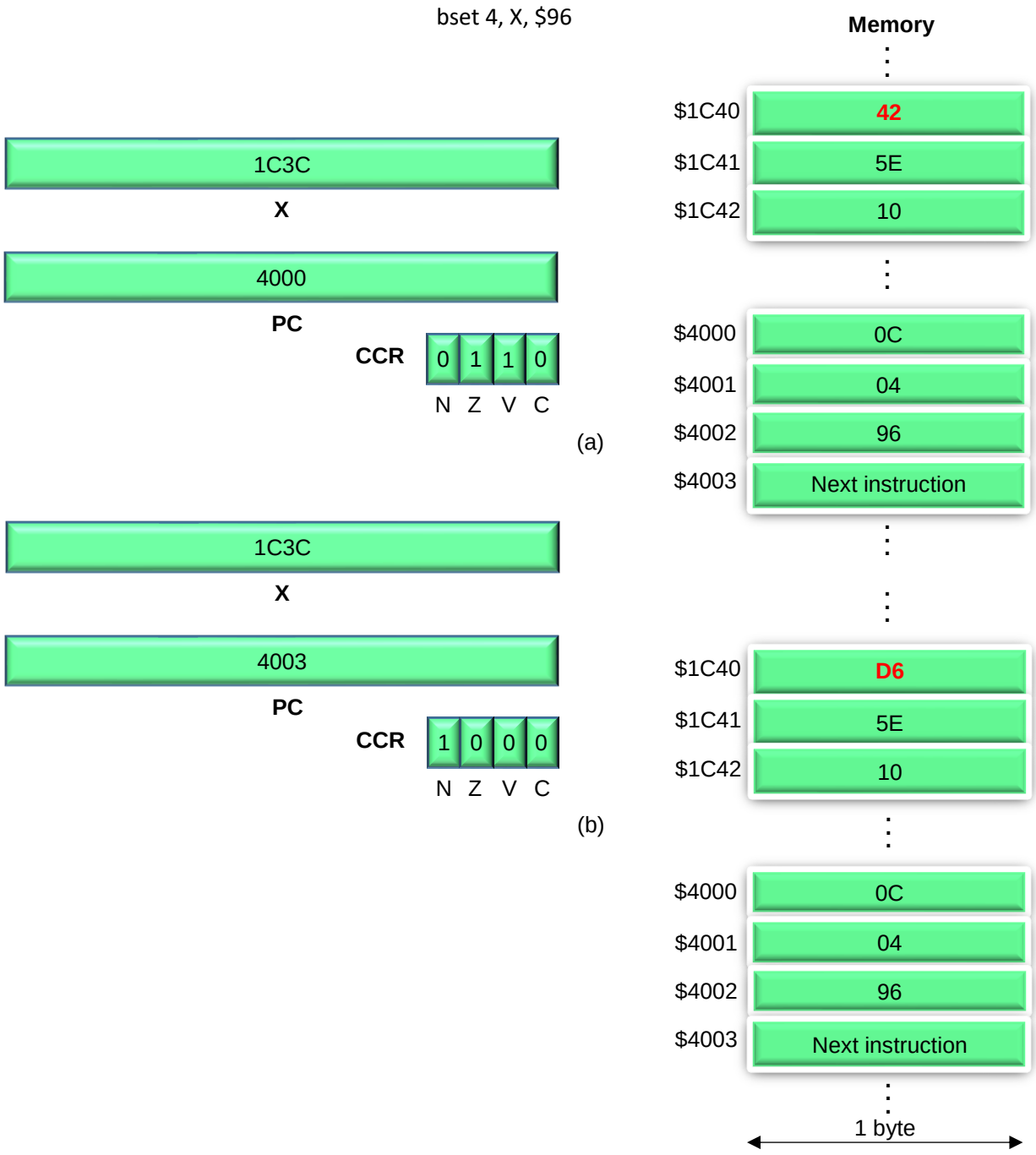


Figure 66. Instruction execution in Example 62.

The transfer instructions copy any register (ABCDXYS) into another one except for the PC. Here is their assembly syntax:

```
tfr R1, R2      ; R1 → R2 (from left to right.)
```

The transfer instructions are two bytes long: the opcode followed by a postbyte. **The posbyte format is not covered in this edition of the textbook.**

The NZVC flags are not affected unless the CCR is the destination register.

If R2 is shorter, the *lower half* of R1 is copied.

If R1 is shorter, then it is *sign-extended* and then copied.

Example 63. Let us say D = $8B1F. When instruction {tfr A, B} executes, register D changes to $8B8B.

Example 64. Let us say D = $8B15. When instruction {tfr A, D} executes, register D changes to $FF8B.

Example 65. Let us say D = $28D5. When instruction {tfr D, A} executes, register D changes to $D5D5.

Exchange instructions swap any two registers (ABCDXY) except for the PC. Here is their assembly syntax:

```
exg R1, R2      ; R1 ↔ R2
```

The exchange instructions are two bytes long: the opcode followed by a postbyte. **The posbyte format is not covered in this edition of the textbook.**

The NZVC flags are not affected unless the CCR is involved.

In case of different size registers

- The shorter register receives the *lower half* of the longer register.
- The longer register receives the *zero-extended* shorter register.

Exercise: Determine the content of register D when the following two instructions execute:

```
ldd #$C274
exg D, A  ◊
```

The following are the pseudo instructions (abbreviations) for two exchange instructions:

```
xgdx; translated to {exg D, X}
xgdy; translated to {exg D, Y}
```

Example 66. Let us say D = $D472 and Y = 613C. When instruction {exg A, Y} executes, the following changes will be made:

```
D = $3C72
Y =  $00D4
```

The load effective address instructions (LEA) calculate the effective address for IDX, IDX1, and IDX2 addressing modes and put the result in one of the following registers:

X, Y, or S.

The mnemonics associated with these three registers are leax, leay, and leas, respectively. The NZVC flags are not affected by these instructions.

The LEA instructions have a syntax similar to the corresponding load instructions:

Assembly syntax of LEA: Mnemonic + Memroy8 in IDX, IDX1, or IDX2 format.

Machine syntax of LEA: Opcode + Memroy8 in IDX, IDX1, or IDX2 format.

Example 67. Figure 67 shows 3 pairs of load-effective-address/load instructions as an example. The *same addressing mode* is used in each pair. The first byte in the Object row is the opcode:

	Pair 1		**Pair 2**		**Pair 3**	
	load effective address	**load**	**load effective address**	**load**	**load effective address**	**load**
Assembly	leas 53, SP	lds 53, SP	leax 8, -Y	ldx 8, -Y	leay D, X	ldy D, X
Object	1Bf035	EFF035	1A68	EE68	19E6	EDE6
The machine and assembly syntax of the load effective address instructions are similar to the load instructions.						

Figure 67. Three pairs of instructions in Example 67.

Example 68. Let us say Y = $1000, Mem($1006) = $12 and Mem($1007) = $34.

When instructions {leay 6, Y} executes, the content of Y changes to $1000 + 6 = $1006.

Let us compare this instruction with {ldy 6, Y}. When {ldy 6, Y} executes, the content of Y changes to

Mem($1000 + 6) = $1234.

Example 69. Consider the following instruction, which is in the indexed addressing mode with a 5-bit constant offset:

```
leax 8, SP-          ; auto post decrement mode
```

Let us first translate it into the machine code:

Following Rule 2, the opcode is determined: $1A.

The offset is +8. Following the rule introduced in Chapter 4, the postbyte is determined: $B8; and therefore, the machine instruction will be $1AB8.

In Figure 68a, the assembled instruction is sitting in the memory, starting at $4000. Let us assume that Figure 68a also shows the values in the relevant memory locations and registers right *before* the instruction executes. The results of instruction execution are shown in Figure 68b.

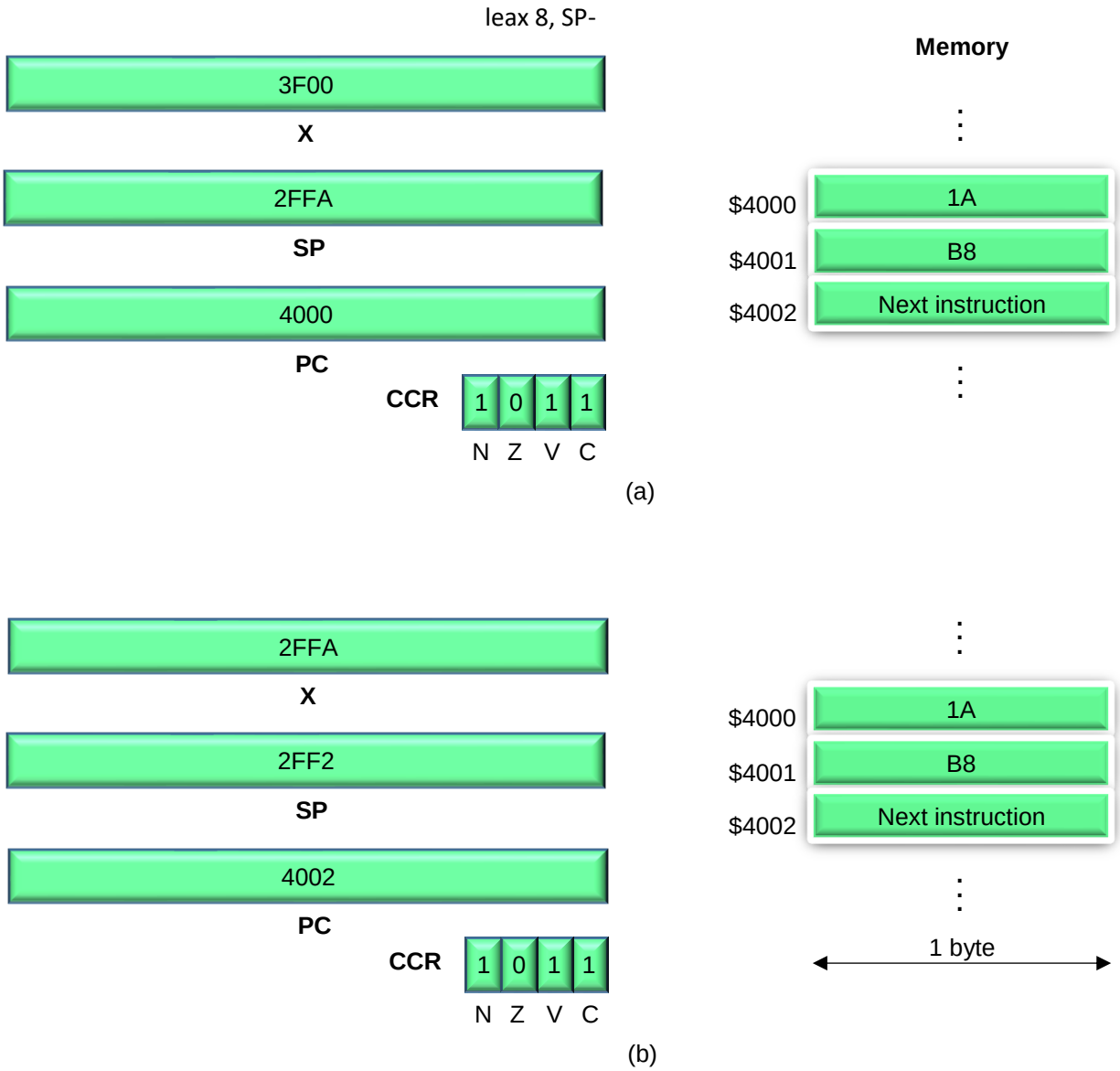


Figure 68. Instruction execution in Example 69.

The move instructions copy a constant or the content of a memory location to another memory location. The CCR is not affected. This instruction has two versions: movb copies one byte, and movw copies one word. Figure 69 shows the addressing modes and object codes of the move instructions. The generic form of the move instructions in the assembly format is as follows:

movb (movw) source, destination

where the valid {source, destination} pairs are shown in Figure 69:

Source → Destination	Mode	Object code format	
IMM → EXT	IMM-EXT	18* + 0B + imm operand[8] + 16-bit address	movb
		18 + 03 + imm operand[16] + 16-bit address	movw
IMM → IDX	IMM-IDX	18 + 08 + postbyte + imm operand[8]	movb
		18 + 00 + postbyte + imm operand[16]	movw
EXT → EXT	EXT-EXT	18 + 0C + 16-bit address + 16-bit address	movb
		18 + 04 + 16-bit address + 16-bit address	movw
EXT → IDX	EXT-IDX	18 + 09 + postbyte + 16-bit address	movb
		18 + 01 + postbyte + 16-bit address	movw
IDX → EXT	IDX-EXT	18 + 0D + postbyte + 16-bit address	movb
		18 + 05 + postbyte + 16-bit address	movw
IDX → IDX	IDX-IDX	18 + 0A + postbyte + postbyte	movb
		18 + 02 + postbyte + postbyte	movw
* $18 is called the *prebyte*, which is followed by the opcode. The prebyte is not covered in this edition of the textbook.			

Figure 69. Addressing modes and object codes of move instructions.

The following are some examples of the move instructions:

movb 4, -X, 3, Y+

movw 4, X, $2000

movb D, SP, 3, SP+

movw $2500, D, Y

Example 70.
Instruction {movb $2000, $3000} copies the content of the memory location at $2000 to the memory location at $3000.

The result of instruction {movw 0, X, 0, Y} can be summarized as follows:

Mem (Y) ← Mem (X) and Mem (Y+1) ← Mem (X+1)

Instruction {movb #$3A, $3000} copies $3A to the memory location at $3000.

Example 71. The following code swaps two bytes at $3100 and $3200:

```
ldaa $3100
movb $3200, $3100
staa $3200
```

The negate instructions negate a memory byte, register A, or register B**.** The three different forms of the negate instruction are as follows. All the addressing modes used for the store instructions (except for the DIR mode) can also be used here:

```
nega            ; negate A, INH mode, one byte long, opcode = $40.
negb            ; negate B, INH mode, one byte long, opcode = $50.
neg  Memory[8]  ; negate memory location
```

Memory[8] can be specified by the addressing modes used in the store instructions (except for the DIR mode.) The machine and assembly syntax of the negate instructions are similar to the store instructions.

Example 72. Figure 70 shows three pairs of negate/store as an example. The *same addressing mode* is used in each pair. The first byte in the Object row is the opcode:

	Pair 1		Pair 2		Pair 3	
	negate	**store**	**negate**	**store**	**negate**	**store**
Assembly	neg $3000	staa $3000	neg 8, -Y	stab 8, -Y	neg D, SP	std D, SP
Object	703000	7A3000	6068	6B68	60F6	6CF6

Figure 70. Three pairs of instructions in Example 72.

The CPU performs the following *subtraction* to negate the Operand:

- Operand = 0 – Operand or Not of Operand + 1.

Overflow occurs only if the Operand = $80.

The C flag is the active-high borrow; therefore, it is always set to 1 unless the Operand = 00.

The N flag is set to 1 only if the result's MSb = 1.

The Z flag is set to 1 only if the result = 0.