Chapter 4

Addressing Modes of HCS12

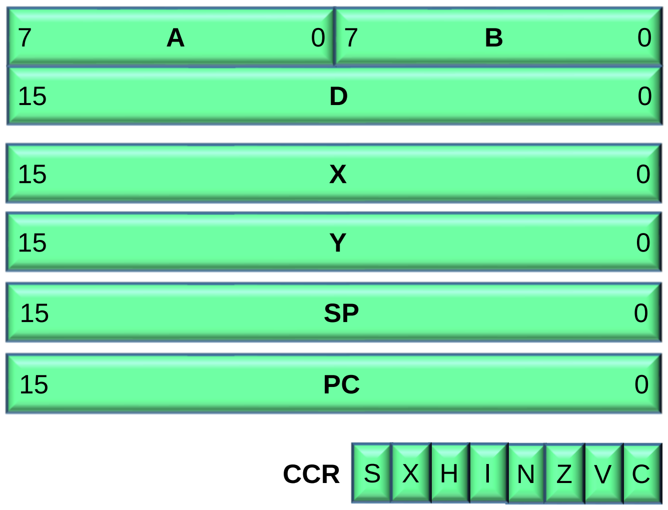
As a reminder, consider the following instruction for a hypothetical machine:

Add K, L, M

Which means K ← L + M

Here is the question: what/where is K, L, or M? And the answer is that operands L and M might be a constant (part of the instruction) or a variable (sitting in the memory or the CPU's registers). And K is always a variable (a register or a memory location). The *addressing mode* included in this and similar instructions tells the CPU where K, L, and M are. Furthermore, for memory-type variables, the addressing mode tells the CPU how to calculate the address of the variable.

HCS12's registers
The CPU's registers of the HCS12 are illustrated in Figure 1a, and their sizes and full names are shown in Figure 1b:



(a)

Abbreviation	Register Name	Size
A, B	Accumulators A, B	8
D	Accumulator D	16
X	Index X	16
Y	Index Y	16
SP or S	Stack Pointer	16
PC	Program Counter	16
CCR	Condition Code Register	8

(b)

Figure 1. CPU registers: (a) abbreviations, (b) register names/meanings.

The 16-bit Register D is the concatenation of two 8-bit registers, A and B. In other words, the upper half of register D is called register A, and the lower half of register D is called register B. As an example, the following two statements are equivalent:

- Write hexadecimal value 29E5 in register D.
- Write hexadecimal 29 in register A and hexadecimal E5 in register B.

As another example, Bit 1 of register A is Bit 9 of register D as well; Bit 6 of register B is also bit 6 of register D. See Figure 1a.

Each of the A, B, and D registers is called an *accumulator*. Each of the X and Y registers is called an *index* register. The SP stands for ***S**tack **P**ointer*. The CCR stands for ***C**ondition **C**ode **R**egister*. The registers in the HCS12 do not have identical functions; you will gradually learn more about them. The PC stands for **P**rogram **C**ounter; we already know how important it is.

CCR: **C**ondition **C**ode **R**egister

In this section, you will learn about the 4 LSbs of register CCR, namely, N, Z, V, and C, which are the abbreviations for the Negative flag, Zero flag, Overflow flag, and Carry flag. An instruction may affect 0 or more flags when it executes. For example, let us see how these flags are affected when an `add` instruction adds two numbers. The different types of `add` instructions and many more are covered in the next chapter.

When an `add` instruction executes:

- The N flag is set to 1 if the MSb of the sum is 1; otherwise, the N flag is reset to zero. In other words, the MSb of the sum is copied to the N flag (regardless of whether overflow occurs).
- The Z flag is set to 1 if the sum is 0; otherwise, the Z flag is reset to 0.
- The V flag is set to 1 if overflow occurs; otherwise, the V flag is reset to 0.
- The C flag is set to 1 if carry happens; otherwise, the C flag is reset to 0.

Example 1. The sum of 5C and A0, two hexadecimal numbers, is hexadecimal FC. Let us say the NZVC flags were 0001 right before this addition was performed. The flags will change to 1010 when the addition is done. ◊

Figure 2 shows how to represent different bases in assembly language:

Base	**Prefix**	**Example**
Binary	%	%10001101
Octal	@	@215
Decimal		141
Hexadecimal	$	$8D
Hexadecimal	Use h as a **suffix**	8Dh

Figure 2. Representing different bases in assembly language.

Memory space and memory size

Each memory byte has a 16-bit or 4-hexadecimal-digit address. Therefore, the memory space is 2^{16} or 64 Kbytes. The smallest address is $0000, and the largest is $FFFF or $2^{16} - 1 = 64K - 1$. Note that

- By 1 K, we mean $2^{10} = 1024$.
- The actual memory size (the number of memory bytes *physically* available to the CPU) in a microcomputer could be fewer than 64K; in other words, some spots in the memory space could be empty.

In this chapter, you will learn the HSC12 addressing modes in the context of two representative instructions: *load* and *store*. In the next chapter, the addressing modes are generalized to more instructions.

Load instructions

The load instructions write a constant value or a value from memory into one of the following registers:

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

Figure 3 shows the mnemonics of the load instructions, one mnemonic for each destination register:

Mnemonic	Meaning	Mnemonic	Meaning
ldaa	load accumulator A	ldx	load register X
ldab	load accumulator B	ldy	load register Y
ldd	load accumulator D	lds	load register SP

Figure 3. Mnemonics for load instructions.

How the load instructions affect the flags

The N and Z flags are affected *meaningfully*, i.e., the N flag is set to logic 1 if the value (that is loaded) is negative; otherwise, the flag is reset to 0; in other words, the MSb of the value is copied to the N flag.

The Z flag is set to logic 1 if the value is 0; otherwise, the Z flag is reset to 0.

The V flag is reset to 0 unconditionally.

The C flag remains unchanged.

As you will see, the store instructions affect the four flags, like the load instructions. The way that the load (as well as the store) instructions affect the NZVC flags is summarized in Figure 4:

N flag	Z flag	V flag	C flag

N ← sign bit (MSb) of value	Z ← 1 if value = 0
	Z ← 0 if value ≠ 0
		V ← 0	C remains unchanged

Figure 4. The way that NZVC flags are affected by load and store instructions.

In programming, we may talk about two different times: *compile time* and *run time*. By compile time, we mean the time the program is written. You can change the program at compile time. By run time, we mean the time the program runs by the CPU, so you cannot change the program anymore at run time.

Immediate addressing mode

The addressing mode is called the *immediate* mode, or **IMM** for short if the operand is constant. Constant operands cannot change at run time. If an operand is constant, it has to be part of the instruction.

Rule 1. Here is the generic assembly format of the load instructions in the immediate mode:
Mnemonic, followed by #, followed by an 8-bit (2-hex-digit) constant when the destination is an 8-bit register.

Mnemonic, followed by #, followed by a 16-bit (4-hex-digit) constant when the destination is a 16-bit register.

Example 2. Consider the following load immediate instruction:

```
ldaa  #$C6    ; Write $C6 in register A
              ; Use character #, the number sign (or the pound symbol) to indicate a constant
              ; "Value" $C6 is a constant, hence part of the instruction
              ; Use a semicolon to comment out a whole line or part of a line
```

Let us assume that the contents of the NZVC flags were 0011 right before this instruction was executed. The outcome of the execution of this instruction is shown in Figure 5: (The flags are affected in accordance with the rules specified in Figure 4.)

Instruction	Operation	NZVC (see Figure 4)	
		Before	After
ldaa #$C6	A ← $C6	0011	1001

Figure 5. The load instruction and its outcome in Example 2.

Note that the most significant digit of value $C6 is greater than 7, so $C6 is negative; this is why the N flag changes to 1. Since $C6 is not 0, the Z flag remains 0. The overflow flag is reset to 0 unconditionally, and the C flag remains 1, unchanged.

Example 3. Consider the following load immediate instruction:

```
ldx  #$2A75   ; Write $2A75 in register X.
```

Constant $2A75, the value, is part of the instruction. The destination register (X) is 16 bits wide, so we need a 4-hex-digit value ($2A75).

Let us assume that the contents of the NZVC flags were 1010 right before this instruction was executed. The outcome of the execution of this instruction is shown in Figure 6:

Instruction	Operation	NZVC (see Figure 4)	
		Before	After
ldx #$2A75	X ← $2A75	1010	0000

Figure 6. The load instruction and its outcome in Example 3.

The most significant digit of value $2A75 is 2, less than 8, so $2A75 is positive; this is why the N flag changes to 0. Since $2A75 is not 0, the Z flag remains 0. The overflow flag is reset to 0 unconditionally, and the C flag remains 0, unchanged.

How to manually translate assembly instructions

The CPU12/CPU12X Reference Manual[1] has a lookup table called **Instruction Set Summary**, which provides all the information about the HCS12 instructions that programmers may need. Remember that a lookup table has a Key column and one or more Value columns. For example, people's names are the Keys,

[1] https://www.nxp.com/files-static/microcontrollers/doc/ref_manual/S12XCPUV2.pdf

and the phone numbers are the Values in a telephone directory. If you know a person's name, the lookup table gives you their phone number.

The leftmost column of the **Instruction Set Summary** table, the Key column, is called the **Source Form**, in which the generic forms of the HCS12 assembly instructions are listed alphabetically. The rest of the columns are the Value columns.

Every generic form starts with a mnemonic.

To translate an assembly instruction into its machine language equivalent, first, determine the *opcode* of the machine instruction.

The opcode of a machine instruction is the first byte of the instruction[2]. Soon, you will see what the opcode is for.

When we say the *opcode of an assembly instruction,* we mean the opcode of the machine instruction that is the translation of the assembly instruction.

Rule 2. To find the *opcode of an assembly instruction* with the mnemonic NNN and the addressing mode MMM, locate the mnemonic NNN in the **Source Form** column. Multiple generic forms of assembly instructions with the same NNN mnemonic may exist in back-to-back rows of the table, but each row has its own addressing mode in the **Addr. Mode** column.

Locate the addressing mode MMM in the **Addr. Mode** column of those rows with the NNN mnemonic. **Stay in this row.**

The first 2-digit number (byte) in the **Machine Coding** column **in this row** is the opcode we are looking for.

Instruction opcodes are determined using Rule 2, but instructions usually have multiple bytes. The way that the rest of the bytes are obtained is instruction dependent and will be explained when an instruction is introduced.

Example 4. Manually assemble the following instruction:
ldaa #$C6 ; write constant $C6 is register A.

Let us follow Rule 2 to determine the opcode: Locate the mnemonic ldaa in the column Source Form. You will see that there are eight generic instructions (sitting in 8 rows) with the same mnemonic (ldaa). In these 8 rows, locate the row with the addressing mode IMM; the first byte in the **Addr. Mode** column of this row is 86, which is the opcode. Use Rule 3 to assemble the above instruction:

Rule 3. Here is the generic machine format of the load instructions in the immediate mode when the destination is an 8-bit register:
Opcode, followed by an 8-bit (2-hex-digit) constant data.

So, the assembly instruction {ldaa #$C6} is translated into $86 C6. ◊

The opcode, the first byte of machine instructions, provides the CPU with information about the instruction. In the above example, what opcode $86 tells the CPU is as follows:

- The instruction is a load instruction.
- The addressing mode is immediate, and the immediate data is in the memory byte right after the opcode.
- The destination is register A.

[2] By "instruction," we mean those instructions covered in this edition of the textbook.

As you see, the addressing mode of this instruction is included in the opcode. Different opcodes may carry different amounts of information, as seen in this textbook.

Example 5. Consider the following assembly instruction:

```
ldd   #$2A75   ; write $2A75  in register D
```

Following Rule 2, the opcode is determined as CC. Use Rule 4 to translate the load instructions when the destination is a 16-bit register:

Rule 4. Here is the generic machine format of the load immediate instructions when the destination is a 16-bit register:
Opcode, followed by a 16-bit (4-hex-digit) constant data.

Therefore, the assembly instruction {ldd #$2A75} is translated into $CC 2A 75.

Figure 7a shows how instruction {CC 2A 75} sits in the memory starting at address $4000 as an example: The opcode ($CC) goes to the memory location at $4000, followed by $2A, the most significant byte of the immediate data; the least significant byte of data will be sitting at $4002, as shown in Figure 7a.

What the opcode $CC tells the CPU is as follows:

- The instruction is a load instruction.
- The addressing mode is immediate, and the immediate data is located in the two back-to-back memory bytes right after the opcode; the most significant byte is sitting at the lower address, the one next to the opcode.
- The destination register is register X.

Note that the opcode and the mnemonic of the same instruction are not 100% equivalent. In general, an opcode provides more information than what the associated mnemonic does. For example, mnemonic ldaa provides the assembler with the following information:

- The current instruction is a load instruction.
- The destination is register A.

As you see, mnemonic ldaa does not tell the assembler anything about the addressing mode. We need to put a pound symbol (#) before the immediate value to inform the assembler of the addressing mode.

Example 6. Instruction {ldd #$2A75} was assembled in Example 5. In Figure 7a, the assembled instruction is placed in the memory. Let us assume that Figure 7a also shows the values in the relevant registers right *before* the instruction executes. The results of instruction execution are shown in Figure 7b. Here is the list of changes made to the state of the CPU by this instruction:

The immediate values $2A and $75 go to registers A and B, respectively; in other words, the immediate value #2A75 goes to register D.

The N flag changes to 0 because $2A75 is positive. (The most significant digit of $2A75 is 2, less than 8.)

The Z flag remains zero because the value $2A27 is not 0.

The V flag is reset to 0 unconditionally.

The C flag remains 1, unchanged.

The program counter is incremented by 3, pointing to the next instruction. The 3 is the length of the instruction $CC 2A 75.

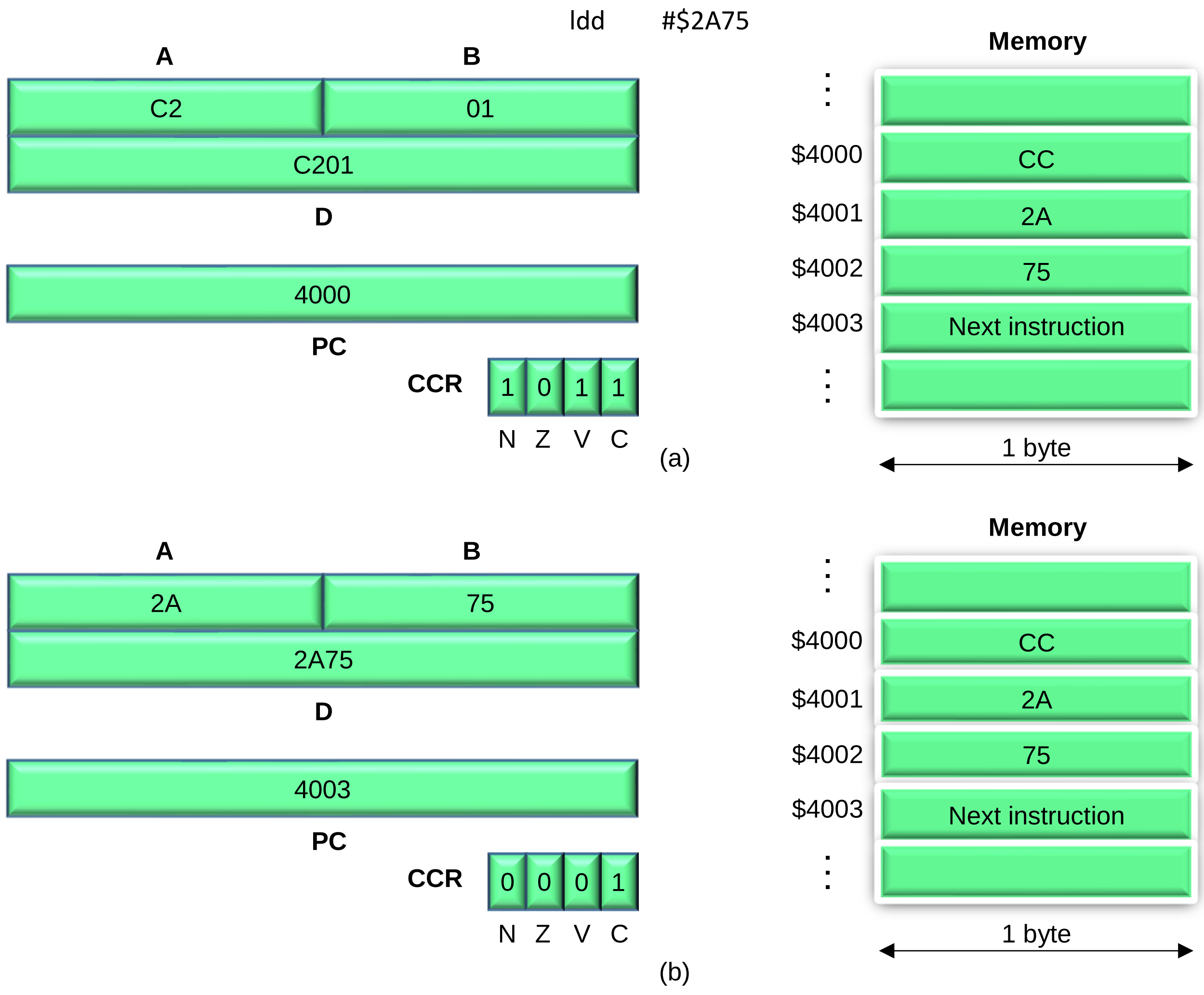


Figure 7. Instruction execution in Example 6

Example 7. Consider instruction {ldab #$1C40}, where B, the destination register, is only 1 byte long, while the constant ($1C40) is 2 bytes long. The assembler drops 1C and replaces this instruction with the following:

```
ldab #$40
```

Extended addressing mode
In the extended addressing mode, or EXT for short, the programmer provides the address of the value (data or operand) instead of the value itself; in other words, now the value is variable while the address is constant; so, the value can change at run time but the address cannot. This is the simplest form of a variable value. You will see more complex variables in this chapter.

Rule 5. Here is the generic assembly format of the load (and store) instructions in the extended mode: Mnemonic, followed by a 16-bit (4-hex-digit) constant address.

Let us say we want to write a program that reads the temperature every day and calculates the temperature change compared to the day before. We can no longer use the immediate addressing mode as the temperature is not constant. The extended addressing mode is a solution to this problem:

Example 8. If we drop the immediate sign from, say, the instructions used in Example 7, the addressing mode changes to the extended mode:

```
ldab  $1C40    ; B ← Mem($1C40) or simply B ← ($1C40)
```

This instruction loads (or reads) one byte from the memory location at address $1C40 and places the value in register B. Look at the comments: by Mem($1C40), we mean the memory content at address $1C40. We may drop Mem and simply say B ← ($1C40). The parentheses around the address indicate that we mean the contents of $1C40, not the value $1C40 itself.

You may wonder why only one byte is read from memory, not, say, 3 bytes. And the answer is that because B, the destination register, is only one byte long.

Rule 6. Here is the generic machine format of the load (and store) instructions in the extended mode: Opcode, followed by a 16-bit (4-hex-digit) constant address.

Example 9. Assemble the instruction in Example 8, place it in the memory starting at address $4000, and execute it. The instruction is shown here again for ease of reference:

```
ldab  $1C40    ; B ← ($1C40)
```

The first byte is the opcode. Following Rule 2, the opcode is determined: $F6.

Following Rule 6, the machine instruction is obtained: $F4 1C 40

Figure 8a shows how instruction {$F4 1C 40} sits in the memory starting at address $4000 as an example: The opcode ($F4) goes to the memory location at $4000, followed by $1C, the most significant byte of the address. The least significant byte of the address will be sitting at $4002, as shown in Figure 8a. Let us assume that Figure 8a also shows the values in the relevant registers and memory locations right *before* the instruction executes. The results of instruction execution are shown in Figure 8b. Here is the list of changes made to the state of the CPU by this instruction:

Value $05, the content of location $1C40, is read and placed in register B. In other words, the content of register D changes from $C27D to $C205.

The N flag changes to 0 because $05 is positive. (The most significant digit of $05 is 0, less than 8.)

The Z flag remains zero because the value $05 is not 0.

The V flag is reset to 0 unconditionally.

The C flag remains 1, unchanged.

The program counter is incremented by 3, pointing to the next instruction. The 3 is the length of instruction {$F4 1C 40}.

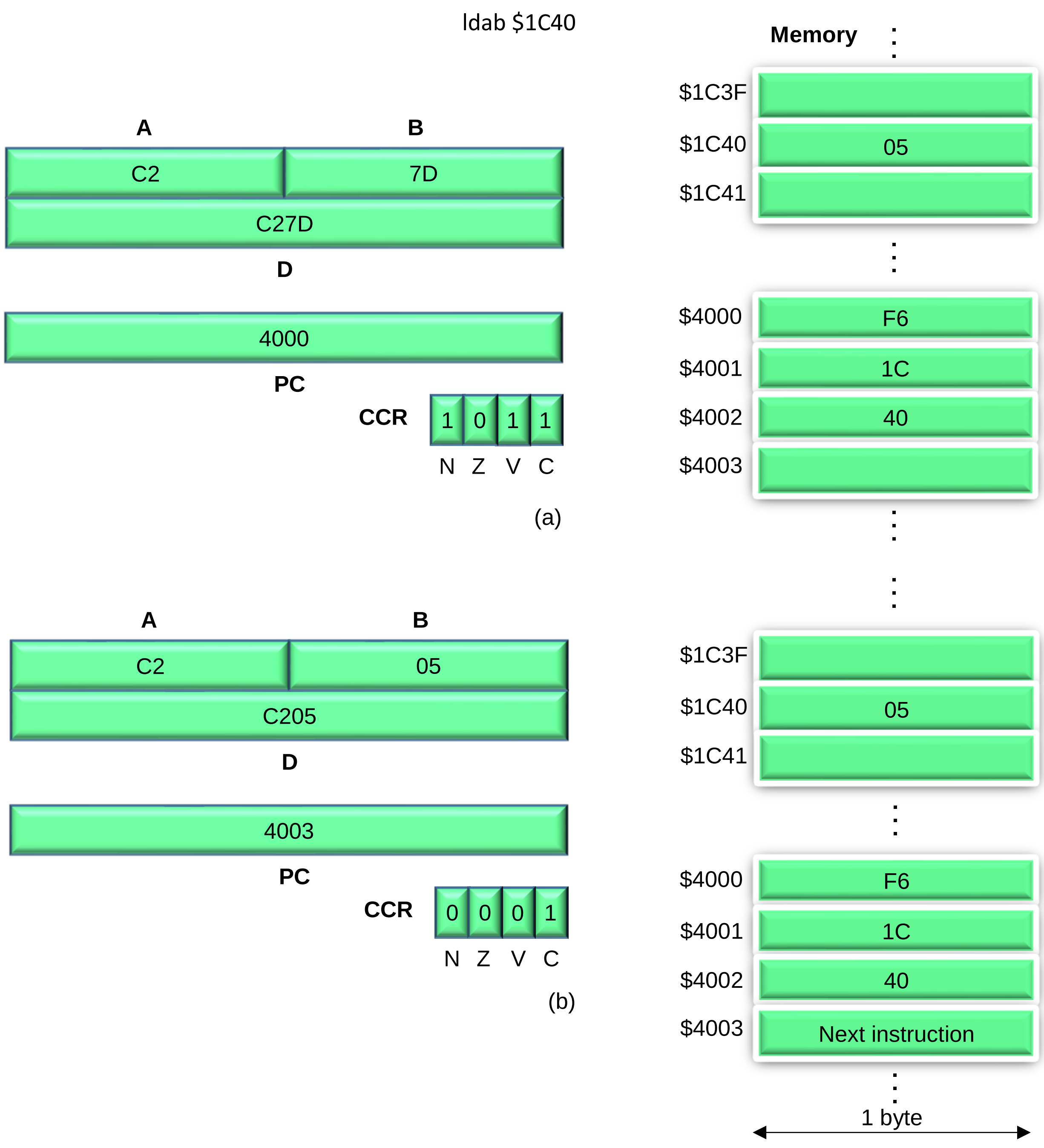


Figure 8. Instruction execution in Example 9

Example 10. The following assembly instruction loads (or reads) 2 bytes from the memory location at address $1C40 and the following address, $1C41, and places them in the upper half and the lower half of register SP, respectively:

```
lds   $1C40     ; upper half of SP ← ($1C40), lower half of SP ← ($1C41)
```

You may wonder why 2 bytes are read from memory, not, say, 3 bytes. And the answer is that because register SP, the destination register, is 2 bytes long.

Pay close attention to the logic implemented in the CPU: We only provide $1C40, the address of the first byte of data (the most significant byte), while $1C41, the address of the second byte (the least significant byte), is implied, i.e., the hardware knows that the second byte must be taken from the memory location located right after $1C40.

Following Rule 2, the opcode is determined: $FF.

Following Rule 6, the machine instruction is obtained: $FF 1C 40

Example 11. In Figure 9a, $FF 1C 40, the machine instruction obtained in Example 10, is placed in the memory starting at address $4000. Let us assume that Figure 9a 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 9b:

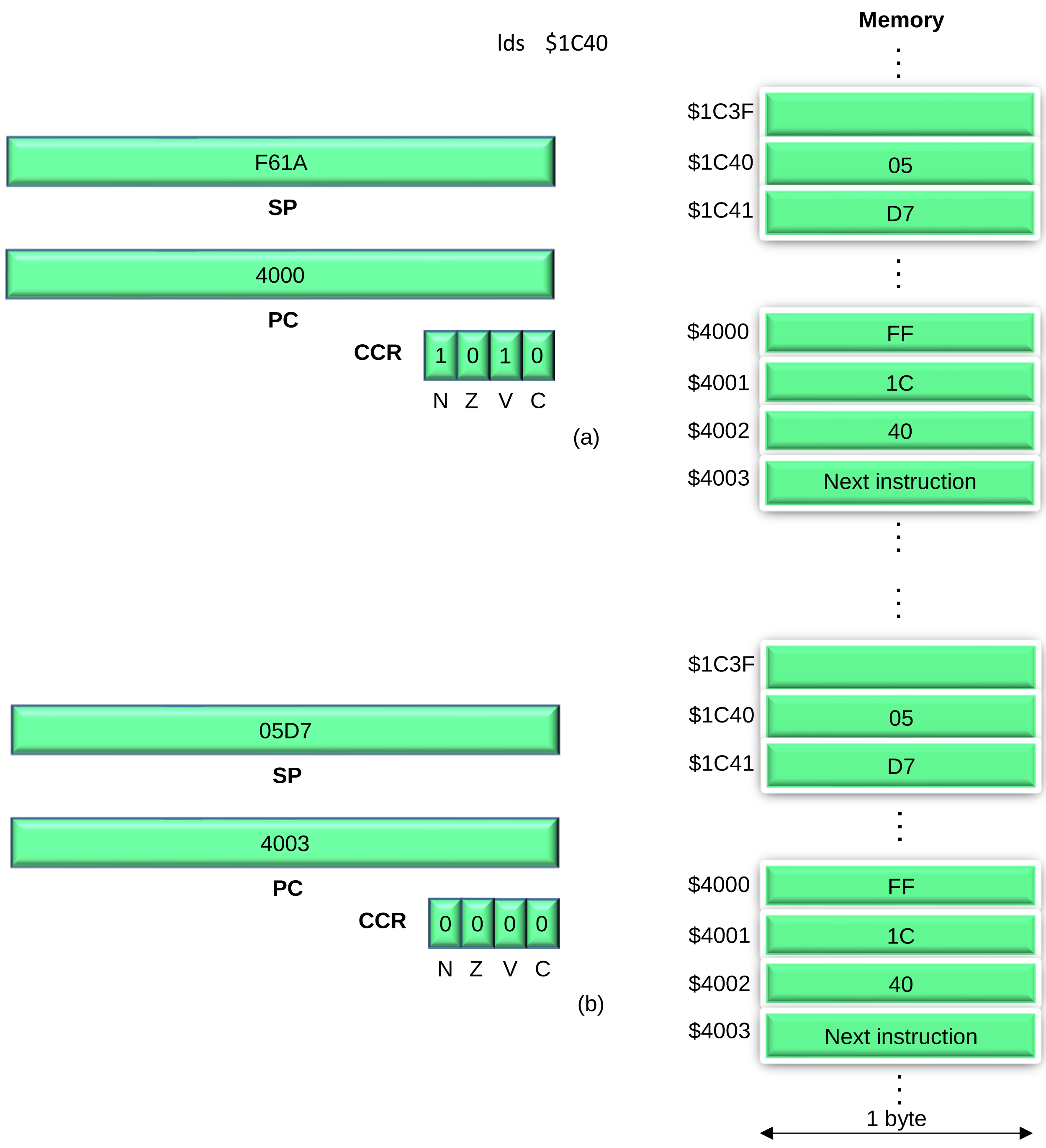


Figure 9. Instruction execution in Example 11.

Store instructions
The store instructions store or write the content or value of one of the following registers into the memory:

A, B, D, S, X, and Y

Figure 10 shows the mnemonics of the store instructions, one mnemonic for each source register:

Mnemonic	Meaning	Mnemonic	Meaning
staa	store accumulator A	stx	store register X
stab	store accumulator B	sty	store register Y
std	store accumulator D	sts	store register SP

Figure 10. Mnemonics for store instructions.

The store instructions have the same impact on the four N, Z, V, and C flags as the load instructions. The rules are shown here again for ease of reference:

The N and Z flags are affected meaningfully, i.e., the N flag is set to logic 1 if the value (to be stored) is negative; otherwise, the flag is reset to 0; in other words, the sign bit of the value is copied into the N flag.

The Z flag is set to logic 1 if the value is 0; otherwise, the flag is reset to 0.

The V flag is reset to 0 unconditionally.

The C flag remains unchanged.

Store instructions in the extended addressing mode
You can also utilize the extended addressing mode (used for the load instructions) to specify the target address for the store instructions. Rule 5 and Rule 6 are shown here again for ease of reference:

Rule 5. Here is the generic assembly format of the load (and store) instructions in the extended mode: Mnemonic, followed by a 16-bit (4-hex-digit) constant address.

Rule 6. Here is the generic machine format of the load (and store) instructions in the extended mode: Opcode, followed by a 16-bit (4-hex-digit) constant address.

Example 12. The following instruction copies the content of register B into memory location at address $1C40: (See Rule 5.)

```
stab  $1C40     ; Mem($1C40) ← B, or simply $1C40 ← B
```

Note that only one byte is copied into the memory because B, the source register, is only one byte wide.

The first byte is the opcode. Following Rule 2, the opcode is determined: $7B.

Following Rule 6, the machine instruction is obtained: $7B 1C 40

In Figure 11a, the assembled instruction is placed in the memory starting at address $4000. Let us assume that Figure 11a 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 11b:

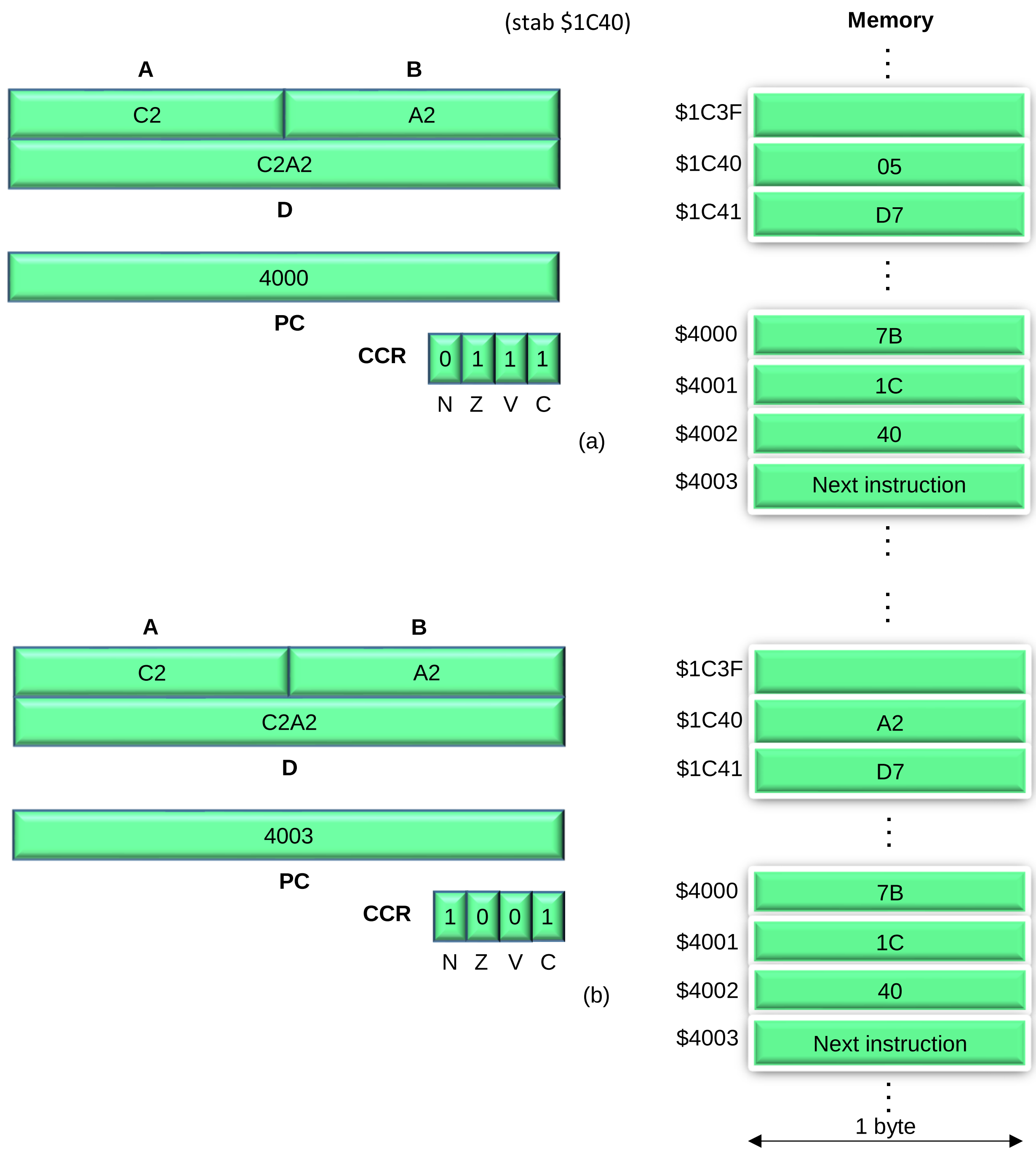


Figure 11. Instruction execution in Example 12.

Example 13. The following instruction copies the content of register SP into memory at addresses $1C40 and $1C41:

```
sts   $1C40     ; $1C40 ← upper half of SP, $1C41 ← lower half of SP
```

Pay close attention to the logic implemented in the CPU: the programmer only provides $1C40, the address of the most significant byte of the destination, while $1C41, the address of the least significant byte, is implied, i.e., the hardware knows that the lower half of register SP must be stored in the memory byte located right after $1C40.

Following Rule 2, the opcode is determined: $7F.

Following Rule 6, the machine instruction is obtained: $7F 1C 40.

Indexed addressing modes

In this section, you learn the syntax and semantics (but not the importance) of the indexed addressing modes; in the next chapter, you will see how powerful and useful these modes are 😊

There are different types of indexed addressing modes:

Indexed addressing mode with a constant offset

Rule 7. Here is the generic assembly format of the load and store instructions:
Mnemonic, followed by a signed constant called the *offset*, followed by a *base* register.

The base register may be one of the following:

X, Y, SP, or PC

The offset can be up to a 16-bit signed constant.

The lists of the load mnemonics and the store mnemonics are shown in Figure 3 and Figure 10, respectively.

The effective address in this mode is calculated (at run time) as follows:

Effective address = Base register + Offset

The effective address for the store instructions is the memory address at which data will be written.

The effective address for the load instructions is the memory address from which data will be read.

As you see, now, not only the data but also the address is variable, so either one can change at run time.

Based on the size of the offset, this addressing mode is further classified as IDX, IDX1, and IDX2, in which the offset is up to 5, 9, and 16 bits wide, respectively.

Example 14. Consider the following load instruction:

```
ldab  4, X ; B ← Mem (4 + X)
```

Where constant +4 is the offset and X is the base register. When this instruction executes, one byte from memory located at the address 4 + X is read and placed in register B. This instruction affects the N, Z, V, and C flags like the other load instructions. Note that +4 is small enough to fit in 5 bits, so this addressing mode is of IDX type.

Example 15. Consider the following load instruction:

```
ldd 4, Y    ; A ← Mem (Y + 4) and B ← Mem (Y + 4 + 1)
```

When this instruction executes, the two back-to-back memory bytes at addresses Y + 4 and Y + 5 are read and placed in registers A and B, respectively. (The byte at the lowest address goes to A). This instruction affects the N, Z, V, and C flags like the other load instructions.

How to assemble the indexed mode with a constant offset

The first byte is always the opcode. The second byte is called the *postbyte*, to be explained shortly. For the three different sizes of the offset (5 bits, 9 bits, and 16 bits), we need to add 0, 1, and 2 extra bytes, respectively, as summarized below:

5-bit offset needs no extra byte: so, the load/store instruction is 2 bytes long (the mode is called IDX)

9-bit offset needs 1 extra byte: so, the load/store instruction is 3 bytes long (the mode is called IDX1)

16-bit offset needs 2 extra bytes: so, the load/store instruction is 4 bytes long (the mode is called IDX2)

Indexed mode with a 5-bit constant offset, IDX

Rule 8. Here is the generic machine format of the load and store instructions:
Opcode, followed by a postbyte.

The format of the postbyte is as follows:

rr0nnnnn

Where nnnnn is the 5-bit signed offset (in the 2's complement system), and rr is the 2-bit binary code for the four different base registers, as shown in Figure 12. Note that there is **always** a 0 after the rr field.

rr	Base register
00	X
01	Y
10	SP
11	PC

Figure 12. Base registers' binary codes in indexed addressing modes with a constant offset.

Example 16. Assemble ldy -14, Y.

Following Rule 2, the opcode is determined: ED.

The base register is Y, so rr = 01 from Figure 12.

Offset = -14 = 10010, which fits in 5 bits.

Therefore,
postbyte = 01 0 10010 = $52

Machine instruction: $ED 52

Since this is an IDX-type mode (offset fits in 5 bits), there is no extra byte, i.e., the instruction is 2 bytes long.

Note:
- The assembler looks at the offset (-14 in this example) as a sign & magnitude number.
- Five-bit offsets are in the following range:

  10000 ≤ 5-bit offset ≤ 01111 or -16 ≤ 5-bit offset ≤ +15

Indexed mode with a 9-bit constant offset, IDX1

Rule 9. Here is the generic machine format of the load and store instructions:
Opcode, followed by a postbyte, followed by the 8 LSbs of the offset (3 bytes altogether).

The postbyte has the following format:

111 rr 00 s

The 3 MSbs have to be 111, followed by the same 2-bit binary code of the base register shown in Figure 12, followed by s, the sign bit of the offset.

This mode needs one extra byte, which holds the 8 LSbs of the offset.

Example 17. Assemble ldy -46, Y.

Following Rule 2, the opcode is determined: $ED.

The base register is Y, so rr = 01 from Figure 12.

Offset = -46 = 1010010, which does not fit in 5 bits, but it does in 9 bits:

Offset = -46 = 1010010

Offset = -46 (in 9 bits) = 1 1101 0010 (do sign extension to make it 9 bits wide.)

Postbyte = 1110 1001

One extra byte (3rd byte of the instruction) for the 8 LSbs of the offset = 1101 0010 = $D2

Machine instruction: $ED E9 D2

Note:

- The assembler looks at the offset (-46 in this example) as a sign & magnitude number.
- Nine-bit offsets are in the following range:

[100000000 ≤ offset ≤ 011111111 or -256 ≤ offset ≤ +255] while [offset < -16 or +15 < offset].

- Five-bit offsets can, of course, be represented in 9 bits as well, but this only makes the instruction longer, a waste of memory, and more.

Indexed mode with a 16-bit constant offset, IDX2

Rule 10. Here is the generic machine format of the load and store instructions:
Opcode, followed by a postbyte, followed by a 2-byte offset (4 bytes altogether).

The postbyte has the following format:

111 rr 010

The 3 MSbs have to be 111, followed by the same 2-bit binary code of the base register, as shown in Figure 12, followed by the constant 010.

This mode needs two extra bytes, which hold the 16-bit offset.

Example 18. Assemble ldab -$E52, SP.

Following Rule 2, the opcode is determined: E6.

Offset -$E52 = $11AE (in the 16's complement system)

The offset ($11AE) is 13 bits wide, so it does not fit in 9 bits, but it does in 16 bits.

The base register is the SP with the binary code of 10 from Figure 12.

Postbyte = 1111 0010 = $F2

Sign extended offset = Two extra bytes = -E52 = -0E52 = F1AE (do sign extension to make the offset 16 bits wide.)

Machine Instruction: $E6 F2 F1 AE (4-byte instruction including two extra bytes)

Note:

- The assembler looks at the offset (-$E52 in this example) as a sign & magnitude number.
- Sixteen-bit offsets are in the following range:

[1000 0000 0000 0000 ≤ offset ≤ 0111 11111111 1111 or -32K ≤ offset ≤ +32k – 1] while [offset < -256 or +255 < offset]

- Five-bit offsets and 9-bit offsets can, of course, be represented in 16 bits as well, but this only makes the instruction longer, which is a waste of memory and more.

Example 19. Consider the following instruction:

ldab 4, X

Let us first translate it into the machine code:

Following Rule 2, the opcode is determined: $E6.

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, and finally, the machine instruction will be $E604.

In Figure 13a, the assembled instruction is sitting in the memory starting at address $4000. Let us assume that Figure 13a 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 13b.

In this example, the effective address from which data is read is calculated (at run time) as follows:

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

So, the memory content at address $1C40 will be read and placed in accumulator B.

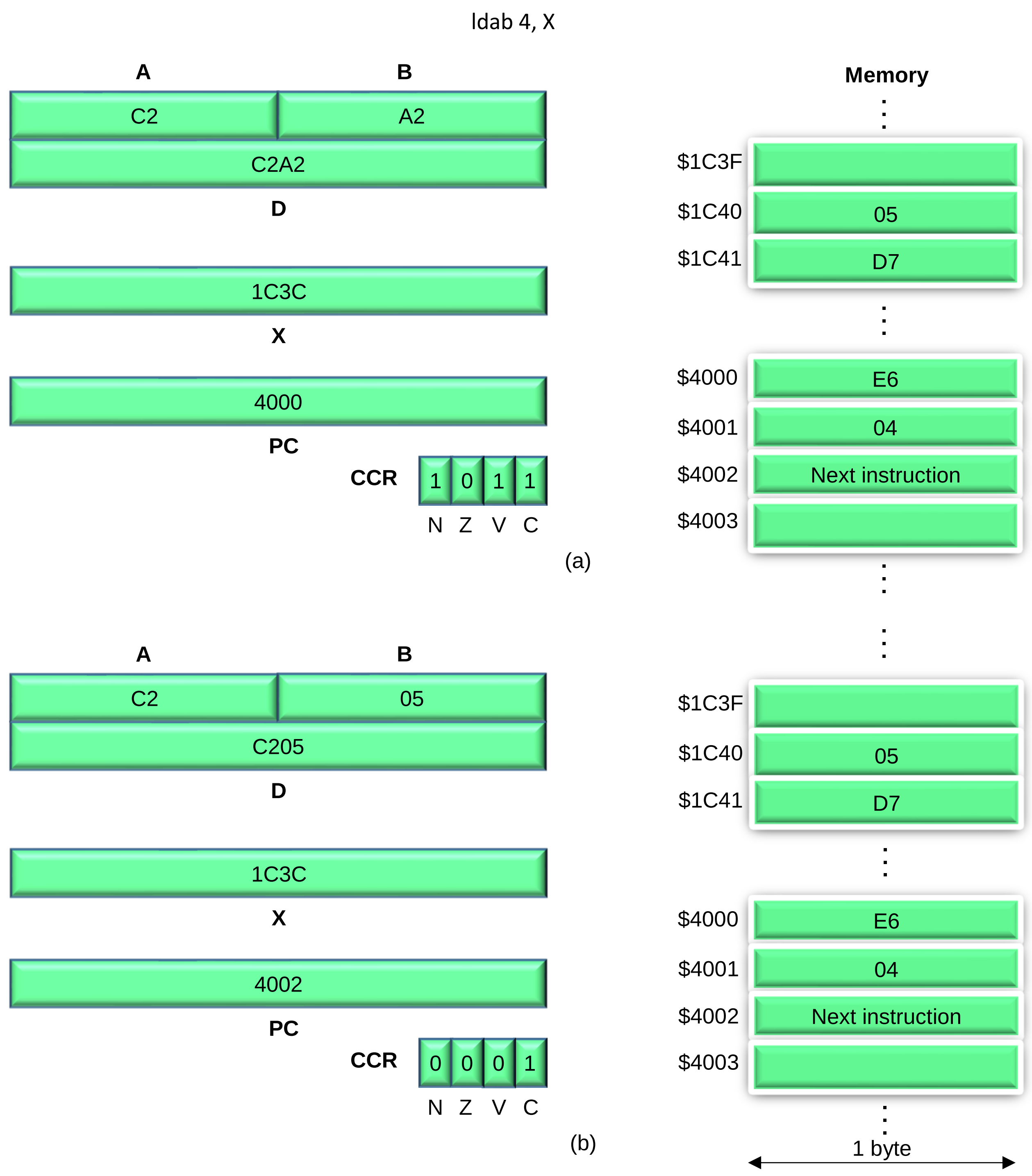


Figure 13. Instruction execution in Example 19.

Example 20. Consider the following instruction:

sty -5, Y ; Mem (Y – 5) ← Upper half of Y, Mem (Y – 5 + 1) ← Lower half of Y

Let us first translate it into the machine code:

Following Rule 2, the opcode is determined: 6D.

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

rr0nnnnn

The base register is Y, so the rr field should be 01. And the 5-bit offset is 11011; therefore, the postbyte will be 0101 1011 = $5B, and therefore the machine instruction will be $6D5B.

In Figure 14a, the assembled instruction is sitting in the memory starting at address $4000. Let us assume that Figure 14a 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 14b.

In this example, the effective address at which data is written is calculated as follows:

Effective address = base register + offset = $1C3D – 5 = $1C38

So, the upper and the lower half of register Y will be written at the memory addresses $1C38 and $1C39, respectively.

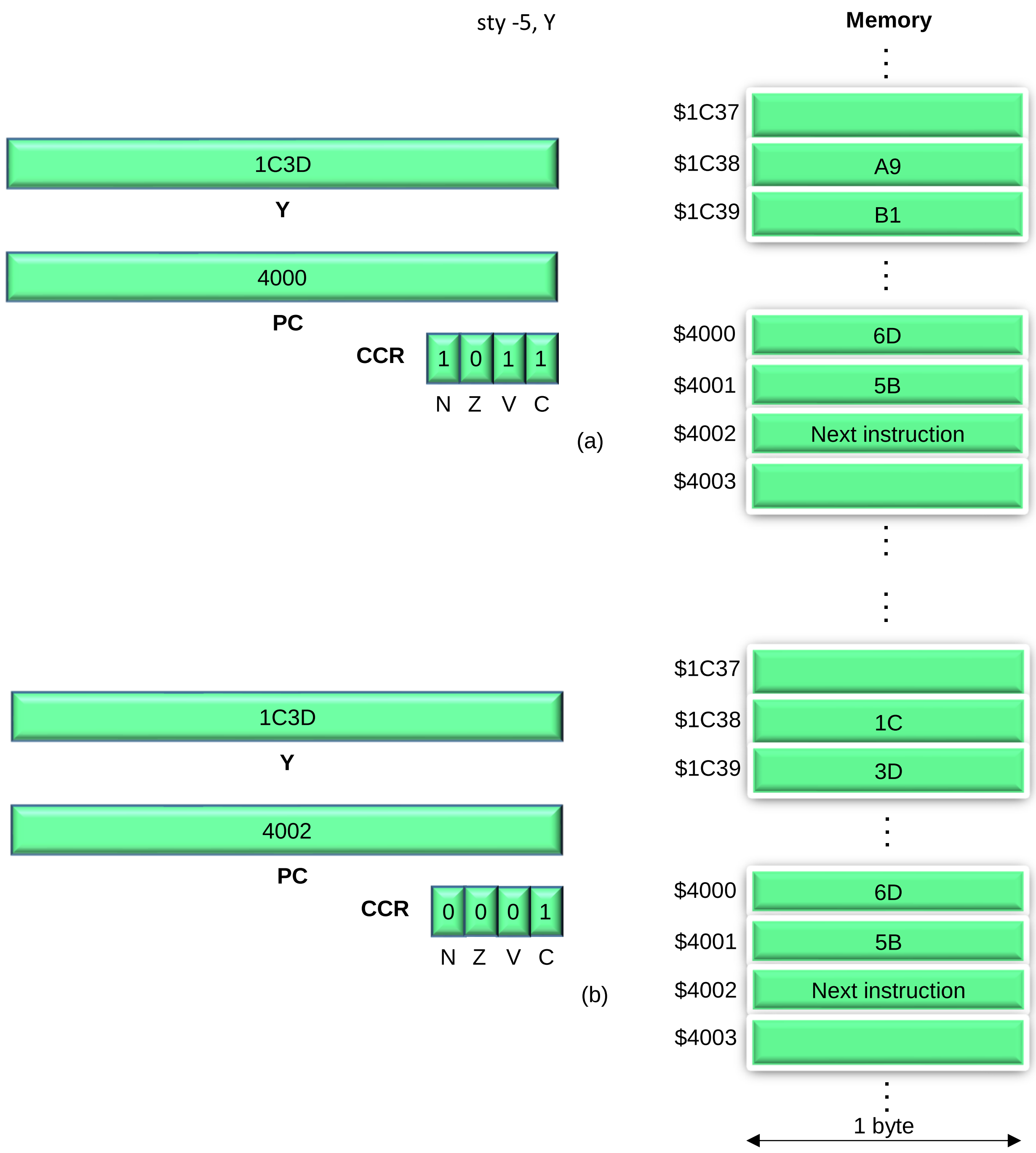


Figure 14. Instruction execution in Example 20.

Auto indexed mode, IDX

There are four categories in this addressing mode, namely predecrement, preincrement, postdecrement, and postincrement:

Rule 11. The generic assembly formats of the 4 auto indexed categories (using the ldx and stx mnemonics) are as follows:

Sixteen-bit data; the ldx and stx mnemonics are used as an example:

- Predecrement:

ldx n, -r ; r ← r – n, **then** Upper half of X ← Mem(r), Lower half of X ← Mem(r + 1).

- Preincrement:

ldx n, +r ; r ← r + n, **then** Upper half of X ← Mem(r), Lower half of X ← Mem(r + 1).

- Postdecrement:

ldx n, r- ; Upper half of X ← Mem(r), Lower half of X ← Mem(r + 1), **then** r ← r – n.

- Postincrement:

ldx n, r+ ; Upper half of X ← Mem(r), Lower half of X ← Mem(r + 1), **then** r ← r + n.

- Predecrement:

stx n, -r ; r ← r – n, **then** Mem(r) ← Upper half of X, Mem(r + 1) ← Lower half of X.

- Preincrement:

stx n, +r ; r ← r + n, **then** Mem(r) ← Upper half of X, Mem(r + 1) ← Lower half of X.

- Postdecrement:

stx n, r- ; Mem(r) ← Upper half of X, Mem(r + 1) ← Lower half of X, **then** r ← r – n.

- Postincrement:

stx n, r+ ; Mem(r) ← Upper half of X, Mem(r + 1) ← Lower half of X, **then** r ← r + n.

Eight-bit data; the ldaa and staa mnemonics are used as an example:

- Predecrement:

ldaa n, -r ; r ← r – n, **then** A ← Mem(r).

- Preincrement:

ldaa n, +r ; r ← r + n, **then** A ← Mem(r).

- Postdecrement:

ldaa n, r- ; A ← Mem(r), **then** r ← r – n.

- Postincrement:

ldaa n, r+ ; A ← Mem(r), **then** r ← r + n.

- Predecrement:

staa n, -r ; r ← r – n, **then** Mem(r) ← A.

- Preincrement:

staa n, +r ; r ← r + n, **then** Mem(r) ← A.

- Postdecrement:
  staa n, r- ; Mem(r) ← A, **then** r ← r – n.

- Postincrement:
  staa n, r+ ; Mem(r) ← A, **then** r ← r + n.

Where constant n is called the offset, +1 ≤ n ≤ +8, and r is the base register.

The base register may be one of the following:

X, Y, SP (PC is not an option)

If you need a negative offset, use the auto *decrement* mode. ◊

Review the above comments and ensure you understand how the effective address is generated and how the base register is updated in each category.

Format of machine instructions in the auto indexed mode addressing
Rule 12. Here is the generic machine format of the load and store instructions:
Opcode, followed by a postbyte.

The postbyte has the following format:

rr1 p nnnn

The 2 MSbs are the base register's binary code, followed by a constant 1, followed by a bit called P, followed by nnnn, a 4-bit constant offset.

The same binary codes shown in Figure 12 are used for this addressing mode with one exception: binary code 11 associated with the PC has a different meaning coming up soon, so do not use it for this addressing mode, *i.e., the postbyte in the format of 111x xxxx has different meanings coming soon …*

The bit called P has the following meaning:

P = 0 indicates the pre-mode.

P = 1 indicates the post-mode.

Figure 15 shows how the offset of this addressing mode is determined based on the nnnn, the 4 LSbs of the postbyte. In the right half of the table in Figure 15, the MSb of nnnn is 1; the CPU looks at them as regular negative numbers (in the 2's complement system); nothing is new in this half. However, in the left half of the table, there is no zero entry in the implied offset column because offset = 0 is useless in the auto mode. Remember, the point of the auto mode is to update (change) the base register *automatically*. Therefore, if nnnn = 0 was interpreted as a 0, then what would be the difference between the following two instructions? No difference!

ldab 0, +X ; auto index mode: the effective address is X, and X remains unchanged.

ldab 0, X ; regular index mode: the effective address is X, and X remains unchanged.

This means that nnnn = 0 would have been wasted in the auto mode if nnnn = 0 had been interpreted as a 0. The designers of this CPU decided not to leave nnnn = 0 unused, as shown in Figure 15: For each non-negative nnnn in the left half of the table, the CPU calculates a positive offset as follows:

Offset = nnnn + 1.

For example, nnnn = 0 means offset = +1, or nnnn = +7 means offset = +8.

nnnn	Implied offset	nnnn	Implied offset
0000	+1	1000	-8
0001	+2	1001	-7
0010	+3	1010	-6
0011	+4	1011	-5
0100	+5	1100	-4
0101	+6	1101	-3
0110	+7	1110	-2
0111	+8	1111	-1

Figure 15. Implied values of offset in auto-indexed mode.

Example 21. Assemble the following instruction, place it in the memory starting at address $4000, and execute it:

ldy 4, SP+ ; upper half of register Y ← Mem(SP), lower half of register Y ← Mem(Sp + 1), **then** SP ← SP+4.

The first byte of the machine instruction is still the opcode. Following Rule 2, the opcode is obtained: $ED

The second and last byte is the postbyte: **rr1p nnnn**.

Since the base register is the SP, the two MSbs of the postbyte should be 10, according to Figure 12.

Bit P must be 1 as the addressing is post-mode.

The offset is 4 in the assembly instruction and the mode is auto *increment*, resulting in the signed offset of +4, which translates into nnnn = +3 = 0011, as shown in Figure 15. Therefore,

Postbyte: 1011 0011= $B3

Machine instruction: $ED B3 (a 2-byte instruction, no extra byte.)

There is no extra byte when using this addressing mode; this is why it is called IDX as well.

In Figure 16a, the assembled instruction is sitting in the memory starting at address $4000. Let us assume that Figure 16a 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 16b:

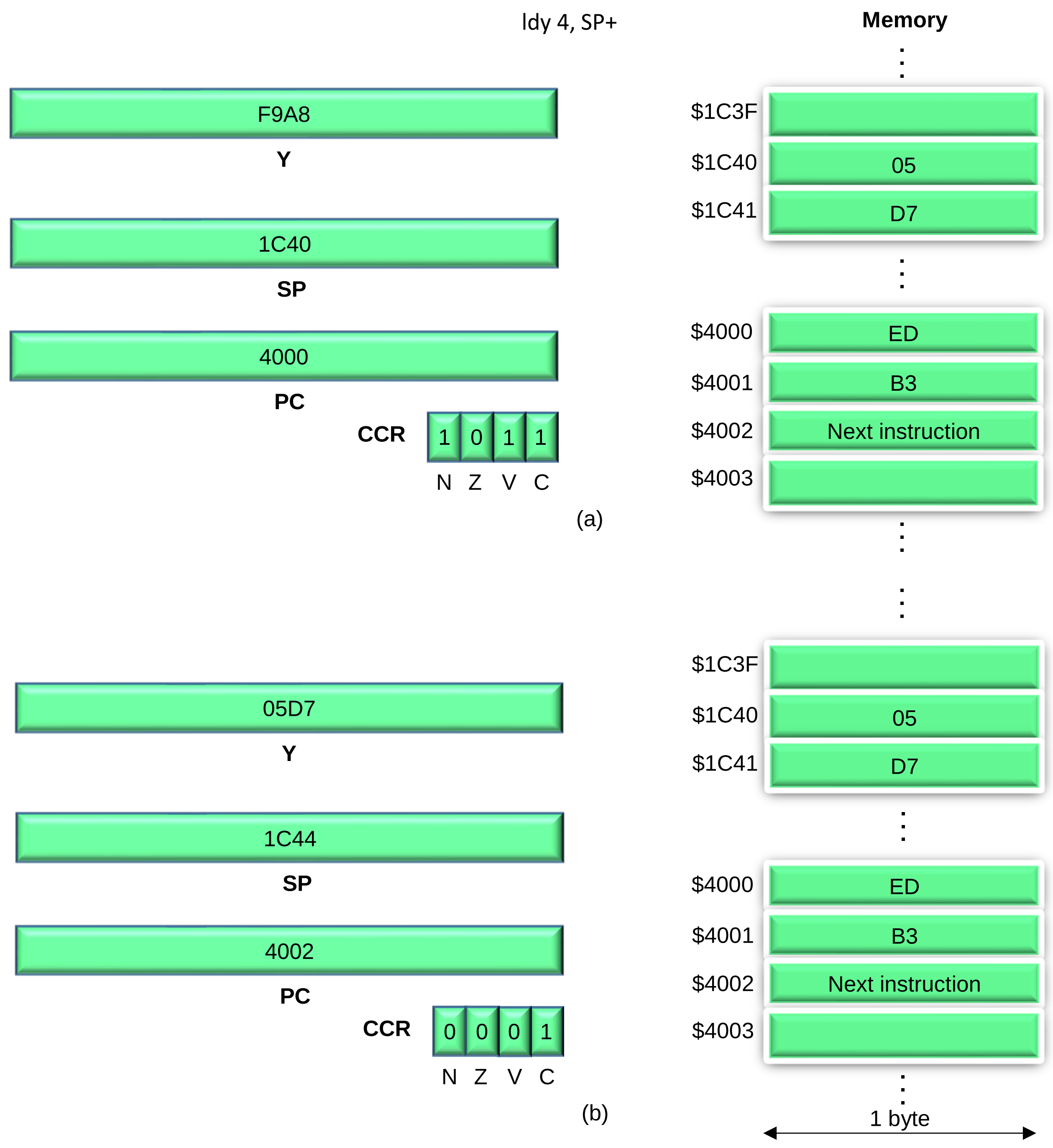


Figure 16. Instruction execution in Example 21.

The addressing mode in this example is the post-increment, which means that the base register SP is first used to calculate the effective address and **then** incremented, meaning that the effective address is the content of SP *before* it is incremented, which is $1C40. So, two bytes are first read from this address, $1C40, and the following address, $1C41, placed in the upper half and the lower half, respectively, of register Y, the destination register, and **then** the SP, the base register, is incremented by 4.

Example 22. Assemble the following instruction, place it in the memory starting at address $4000, and execute it:

sts 5, -X ; X ← X – 5, **then** Mem(X) ← upper half of register SP, Mem(X + 1) ← lower half of register SP.

The first byte of the machine instruction is still the opcode. Following Rule 2, the opcode is obtained: $6F

The second and last byte is the postbyte: rr1p nnnn.

Since the base register is X, the 2 MSbs of the postbyte should be 00, according to Figure 12.

Bit p must be 0 as the addressing is pre-mode.

Since the mode is auto-decrement, the offset will be -5: nnnn = 1011; therefore,

Postbyte = 0010 1011= $2B.

Machine instruction: $6F2B (a 2-byte instruction, no extra byte.)

There is no extra byte when using this addressing mode; this is why it is also called IDX.

In Figure 17a, the assembled instruction is sitting in the memory starting at address $4000. Let us assume that Figure 17a 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 17b. In this example, the addressing mode is pre-decrement, meaning that X, the base register, is first decremented and **then** used. Specifically, first X becomes $1C4D – 5 = $1C48. Then the upper half of register SP is written in the memory at address $1C48, and the lower half of register SP is written in the memory location at $1C49.

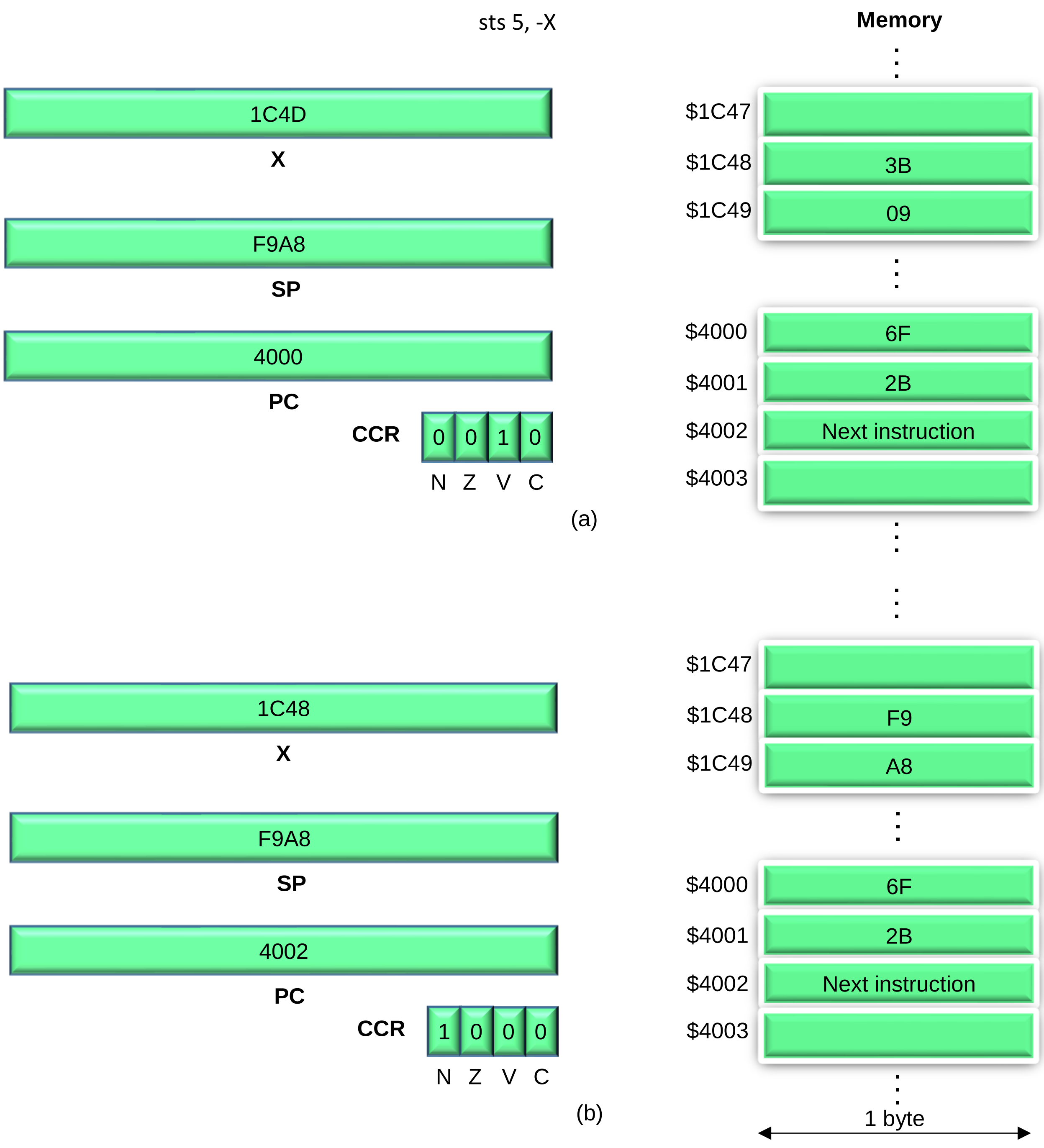


Figure 17. Instruction execution in Example 22.

Indexed mode with offset in an accumulator, IDX

The indexed addressing modes you learned above have a *constant* offset, while the base is *variable* because it is a register. If you need a variable offset as well, you should use the indexed mode with offset in an accumulator explained in this section:

Rule 13. The generic assembly formats of this mode are as follows. Mnemonics ldx, stx, ldaa, and staa are used as an example:

ldx Acc, r ; Upper half of X ← Mem (Acc + r), Lower half of X ← Mem (Acc + r + 1).

stx Acc, r ; Mem (Acc + r) ← Upper half of X, Mem (Acc + r + 1) ← Lower half of X.

ldaa Acc, r ; A ← Mem (Acc + r).

staa Acc, r ; Mem (Acc + r) ← A.

where Acc (the offset) is one of the accumulators, namely, A, B, or D, and r (the base register), is one of the following:

X, Y, SP, or PC

Note: If the Acc is 8 bits wide, then to add Acc + r, the Acc is zero extended.

Example 23.

- Instruction {ldaa B, Y} reads one byte from memory at address {B + Y} and puts it in register A.
- Instruction {ldx D, Y} reads on byte from memory at address {D + Y} and puts it in the upper half of register X; also, reads another byte from the next memory location at address {D + Y + 1}, and puts it in the lower half of register X.
- Instruction {staa B, Y} stores register A in memory at address {B + Y}.
- Instruction {stx D, Y} stores the upper half of register X in the memory at address {D + Y}, and stores the lower half of register X in the next memory location at address {D + Y + 1}.

Format of machine instructions in the indexed mode with offset in an accumulator

Rule 14. The generic machine format of the load and store instructions is as follows:
Opcode, followed by a postbyte.

The postbyte has the following format:

111 rr 1aa (aa ≠11)

Starting with the MSb, the first 3 bits are constant 111, followed by rr, the binary code of the base register (see Figure 12), followed by constant 1, followed by aa, the binary code for the accumulator, which is shown in Figure 18.

Note that in this addressing mode, you should not use aa = 11 (postbyte = 111 xx 111); otherwise, it would be interpreted as the ***indexed indirect mode with an offset in accumulator D*** *coming up soon …*

aa	Accumulator
00	A
01	B
10	D
11	Coming up …

Figure 18. Accumulators' binary codes in indexed addressing mode with offset in an accumulator.

Example 24. Assemble the following instruction, place it in the memory starting at address $4000, and execute it:

ldaa B, Y ; A ← Mem(B + Y),

The first byte is the opcode. Following Rule 2, the opcode is determined: A6.

The second and last byte is the postbyte: **111 rr 1aa**.

Since the base register is Y, the rr field of the postbyte should be 01, according to Figure 12. Since the accumulator is B, the aa field of the postbyte should also be 01 according to Figure 18; therefore, the Postbyte = 11101101= $ED, and the machine instruction will be $A6ED. Note that this addressing mode is of the IDX type as well, i.e., there is no extra byte.

In Figure 19a, the assembled instruction is sitting in the memory starting at $4000. Let us assume that Figure 19a also shows the values in the relevant memory locations and registers *before* the instruction executes. The results of instruction execution are shown in Figure 19b. The address from which data is loaded is B + Y = 80 + 1C3E = $1CBE. Note that the content of this location is $5A; this is why A, the destination register, changes to $5A.

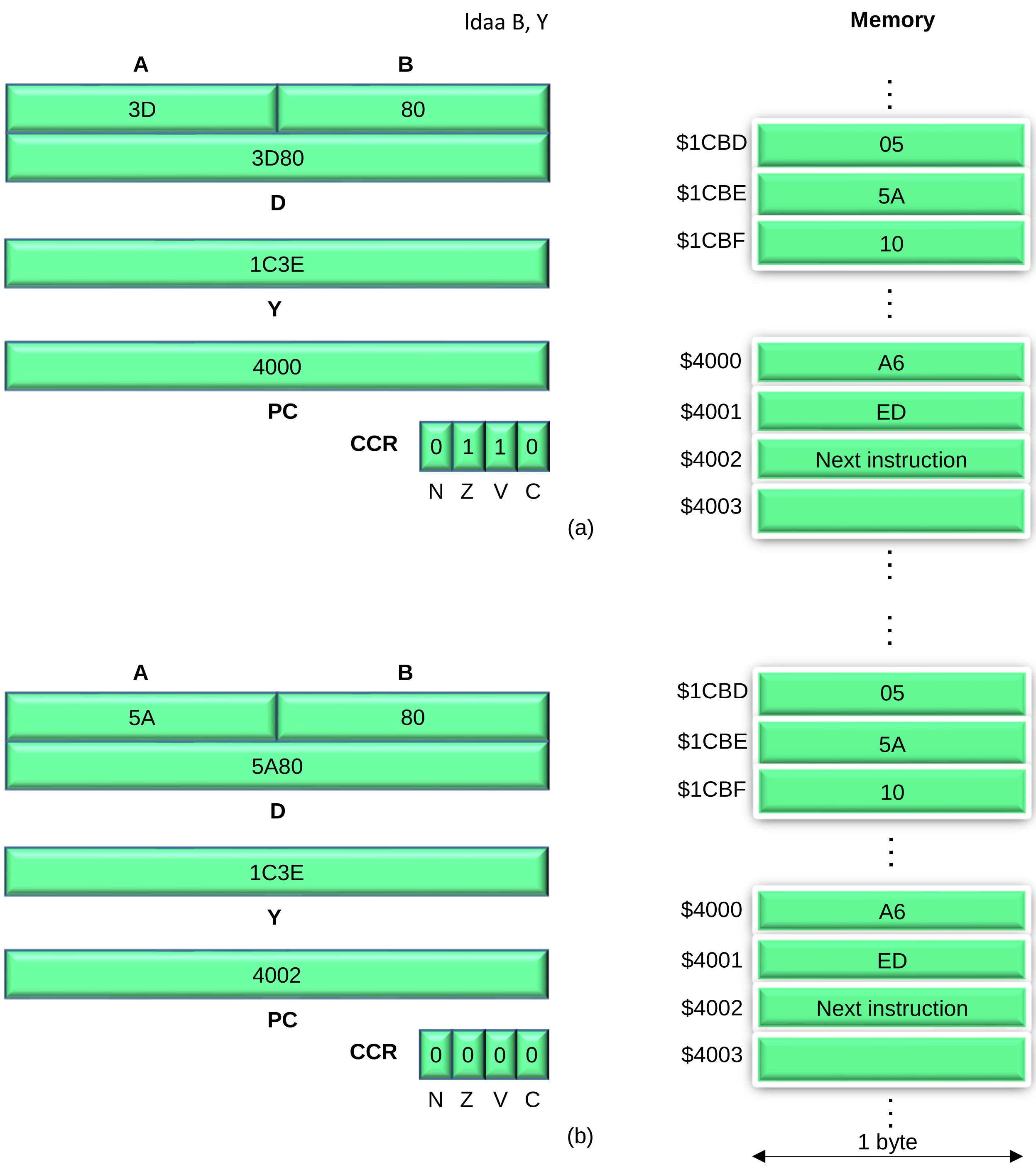


Figure 19. Instruction execution in Example 24.

Example 25. Assemble the following instruction, place it in the memory starting at address $4000, and execute it:

```
staa B, Y     ; Mem(B + Y) ← A
```

The opcode is the first byte. Following Rule 2, the opcode is determined: $6A

The second and last byte is the postbyte: **111 rr 1aa**.

Since the base register is Y, the rr field of the postbyte should be 01, according to Figure 12. Since the accumulator is B, the aa field of the postbyte should also be 01 according to Figure 18; therefore, the Postbyte = 11101101= $ED, and the machine instruction will be $6AED. Note that this addressing mode is of the IDX type as well, as there is no extra byte.

In Figure 20a, the assembled instruction is sitting in the memory starting at $4000. Let us assume that Figure 20a also shows the values in the relevant memory locations and registers *before* the instruction executes. The results of instruction execution are shown in Figure 20b. Note that data is stored in the memory location at B + Y = $80 + $1C3E = $1CBE.

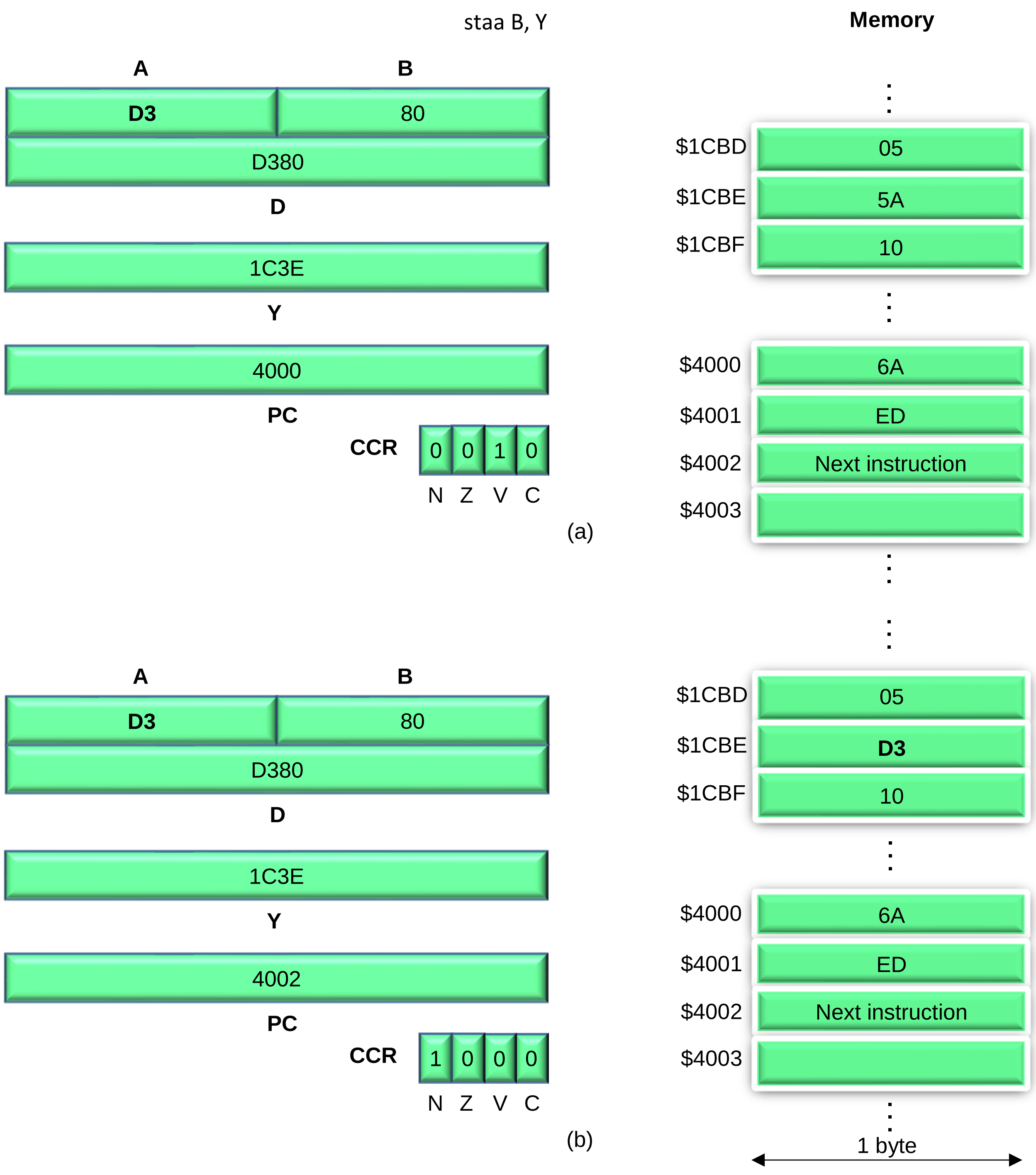


Figure 20. Instruction execution in Example 25.

Indexed indirect mode with a 16-bit constant offset, [IDX2]
Remember that in the regular indexed addressing mode with a constant offset, {base register + offset} is a *pointer to data* or the *address of data*. In the indexed *indirect* mode with a constant offset (covered in this section), however, {base register + offset} is *a pointer to a pointer to data* or the *address of the address of data*.

Rule 15. Here are the generic assembly formats of this mode using the ldy, sty, ldb, and stb mnemonics:

- ldy [offset, r] ; Upper half of Y ← **Mem**(Mem(offset + r) & Mem(offset + r + 1))

  ; Lower half of Y ← **Mem**(Mem(offset + r) & Mem(offset + r + 1) + 1)
- sty [offset, r] ; **Mem**(Mem(offset + r) & Mem(offset + r + 1)) ← Upper half of Y

  ; **Mem**(Mem(offset + r) & Mem(offset + r + 1) + 1) ← Lower half of Y
- ldb [offset, r] ; B ← **Mem**(Mem (offset + r) & Mem(offset + r + 1))
- stb [offset, r] ; **Mem**(Mem(offset + r) & Mem(offset + r + 1)) ← B

Where
we use "&", the and sign (ampersand) to concatenate two numbers,

the offset is a 16-bit signed constant, and

r (the base register) is one of the following:

X, Y, SP, or PC.

Format of machine instructions in the indexed mode with offset in an accumulator
Rule 16. Here is the machine format of the load and the store instructions using this addressing mode: Opcode, followed by a postbyte, followed by a 16-bit constant offset.

The postbyte has the following format:

111rr011

Starting with the MSb, the first 3 bits are constant 111, followed by rr, the binary code of the base register (see Figure 12), followed by constant 011.

The 16-bit offset goes to the third and fourth bytes of the instruction; therefore, this addressing mode needs two extra bytes, which is why this mode is categorized under [IDX2]. The brackets signify the additional level of indirection: *address of address*.

Example 26. Assemble the following instruction, place it in the memory starting at address $4000, and execute it:

ldab [$2F00, Y]

The first byte is the opcode. Following Rule 2, the opcode is determined: $E6.

The second byte is the postbyte: **111rr011**.

Since the base register is Y, the rr field of the postbyte should be 01 according to Figure 12; therefore, the Postbyte = 11101011= $EB.

The third and fourth bytes of the instruction are the 16-bit offset; therefore, the machine instruction will be $E6 EB 2F00.

In Figure 21a, the assembled instruction is sitting in the memory starting at $4000. Let us assume that Figure 21a 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 21b. Let us take a close look at the execution of this instruction:

ldab [$2F00, Y] ; B ← **Mem**(Mem (offset + r) & Mem (offset + r + 1))

r (which is Y) = $003E

Offset = $2F00

Offset + r = $2F3E

Offset + r + 1 = $2F3F

Mem (offset + r) = Mem ($2F3E) = $30

Mem (offset + r + 1) = Mem ($2F3F) = $00

Mem(Mem (offset + r) & Mem (offset + r + 1)) = Mem($3000) = $05

B ← $05.

The *address of the address* from which data is loaded is $2F00 + Y = $2F00 + 003E = $2F3E. Now we need to read the destination address (2 bytes) from $2F3E and the next location, $2F3F. The content of $2F3E is $30, and the content of $2F3F is $00. So, the destination address from which one byte will be read is $3000. The content of $3000 is $05. This is why B, the destination register, changes to $05.

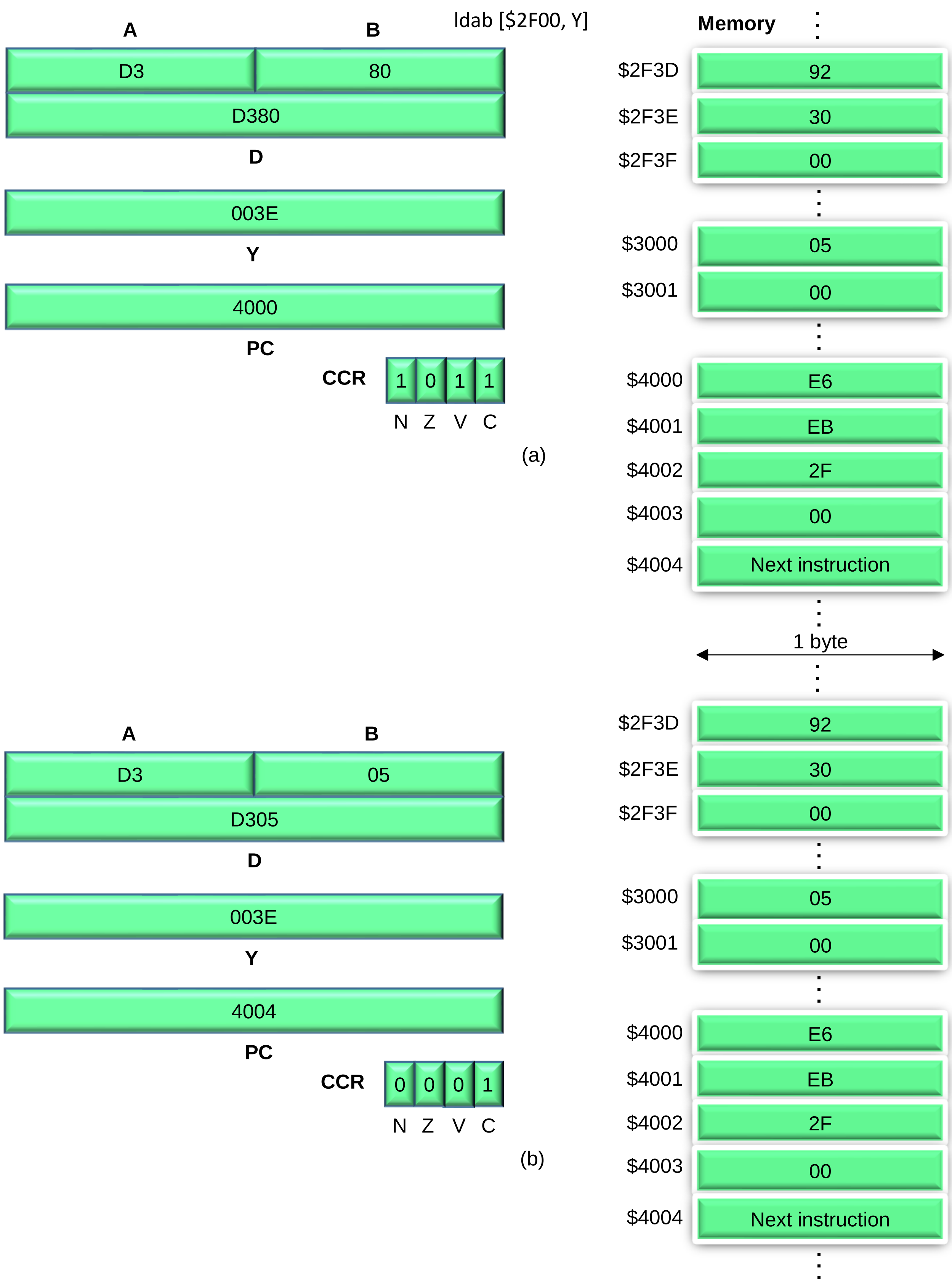


Figure 21. Instruction execution in Example 26.

Example 27. Assemble the following instruction, place it in the memory starting at address $4000, and execute it:

std [$1200, X]

The first byte is still the opcode. Following Rule 2, the opcode is determined: $6C.

The second byte is the postbyte: **111rr011**.

Since the base register is X, the rr field of the postbyte should be 00 according to Figure 12; therefore, the Postbyte = 11100011= $E3.

The third and fourth bytes are the 16-bit offset; therefore, the machine instruction will be $6C E3 1200.

In Figure 22a, the assembled instruction is sitting in the memory starting at $4000.Let us assume that Figure 22a 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 22b. Let us take a close look at the execution of this instruction:

std [$1200, X] ; **Mem** (Mem (offset + X) & Mem (offset + X + 1)) ← Upper half of D (which is A)

; **Mem** (Mem (offset + X) & Mem (offset + X+ 1) + 1) ← Lower half of D (which is B)

X = $103E

Offset = $1200

Offset + r = $223E

Offset + r + 1 = $223F

Mem (offset + r) = Mem ($223E) = $32

Mem (offset + r + 1) = Mem ($223F) = $A0

Mem (Mem (offset + r) & Mem (offset + r + 1)) = Mem($32A0) ← A = $00

Mem (Mem (offset + r) & Mem (offset + r + 1) + 1) = Mem($32A1) ← B = $3E

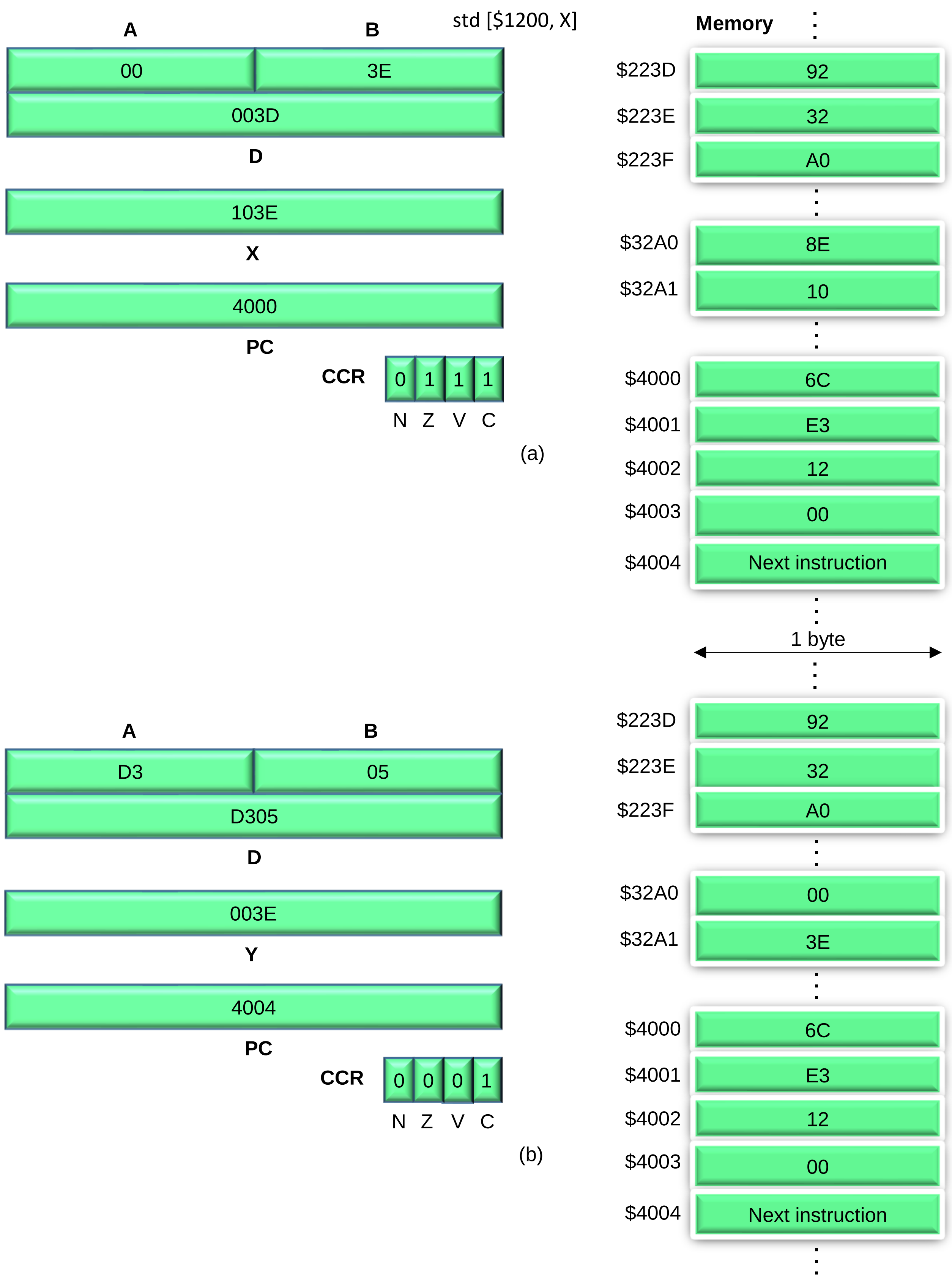


Figure 22. Instruction execution in Example 27.

Indexed indirect mode with an offset in register, D [D, IDX]
The offset was constant in the previous mode (the indexed indirect with a constant offset). In this section, the addressing mode is identical to the previous one with one difference: now the offset is not constant; it is register D, hence variable.

Rule 17. Here is the generic assembly format of this mode using ldy, sty, ldaa, and staa mnemonics:

- ldy [D, r] ; Upper half of Y ← **Mem**(Mem(D + r) & Mem(D + r + 1))

  ; Lower half of Y ← **Mem**(Mem(D + r) & Mem(D + r + 1) + 1)

- sty [D, r] ; **Mem**(Mem(D + r) & Mem(D + r + 1)) ← Upper half of Y

  ; Mem(Mem(D + r) & Mem(D + r + 1) + 1) ← Lower half of Y

- lda [D, r] ; A ← **Mem**(Mem(D + r) & Mem(D + r + 1))

- sta [D, r] ; **Mem**(Mem(D + r) & Mem(D + r + 1)) ← A

Where
we use "&", the and sign (ampersand) to concatenate two numbers,

the offset is a 16-bit signed constant, and

r (the base register) is one of the following:

X, Y, SP, or PC

Format of machine instructions in the indexed mode with an offset in register D
Rule 18. Here is the machine format of the load and store instructions using this addressing mode: Opcode, followed by a postbyte.

The postbyte has the following format:

111rr111

As you see, the 3 MSbs and the 3 LSbs of the postbyte are constant 111. The middle 2 bits are the rr field, the binary code of the base register (see Figure 12).

The instruction is only 2 bytes long; this is why the label IDX is used to categorize this mode as [D, IDX]; the brackets signify one additional level of indirection: *address of address*.

Example 28. Assemble the following instruction, place it in the memory starting at address $4000, and execute it:

stx [D, SP] ; **Mem**(Mem(D + SP) & Mem(D + SP + 1)) ← Upper half of X

; Mem(Mem(D + SP) & Mem(D + SP + 1) + 1) ← Lower half of X

The first byte is still the opcode. Following Rule 2, the opcode is determined: $6E.

The second byte is the postbyte: **111rr111**.

Since the base register is SP, the rr field of the postbyte should be 10 according to Figure 12; therefore, the Postbyte = 11110111= $F7; therefore, the machine instruction will be $6EF7.

In Figure 23a, the assembled instruction is sitting in the memory starting at $4000. Let us assume that Figure 23a 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 23b. Let us take a close look at the instruction execution:

SP = \$2004

D = \$002E

D + SP = \$2032

D + SP + 1 = \$2033

Mem (D + SP) = Mem (\$2032) = \$32

Mem (D + SP + 1) = Mem (\$2033) = \$A0

Mem (Mem (D + SP) & Mem (D + SP + 1)) = Mem (\$32A0)

Mem (Mem (D + SP) & Mem (D + SP + 1) + 1) = Mem (\$32A1)

Mem (\$32A0) $\leftarrow$ Upper half of X = \$8B

Mem (\$32A1) $\leftarrow$ Lower half of X = \$09

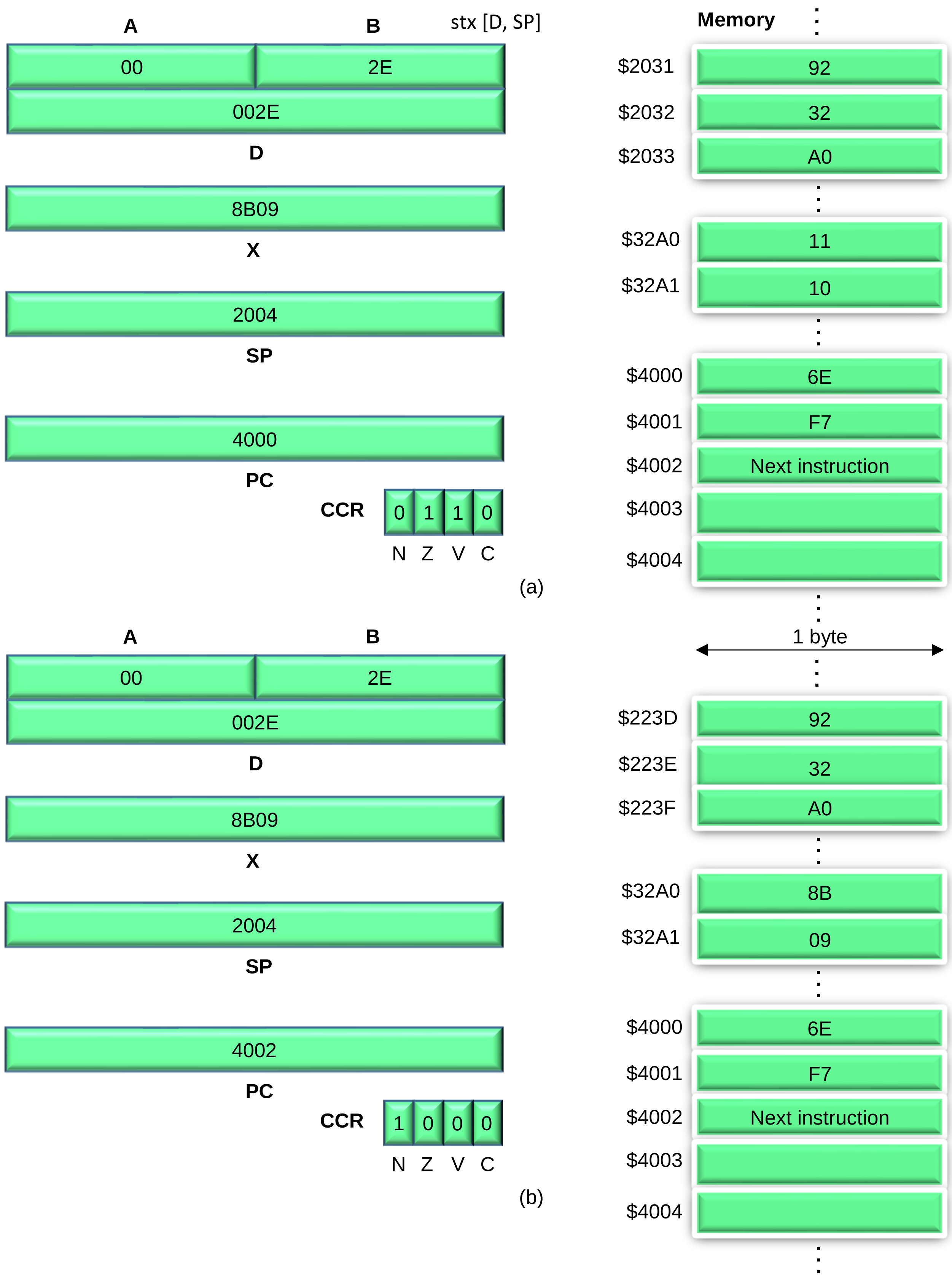


Figure 23. Instruction execution in Example 28.

Direct addressing mode, DIR

The *direct* addressing mode, or DIR for short, is a special case of the extended addressing mode when the two most significant hex digits of the address are zero: instead of a 16-bit address, the CPU is now provided with an 8-bit address, the two least significant hex digits of the address, making this and similar instructions, hence the program shorter. Then the CPU zero-extends the two digits to get a 16-bit address. For example, ldd $40 reads memory location $0040 into register A and location $0041 into register B.

Every 256 consecutive bytes are called one *page* provided that the address of the first byte ends in $00. The two most significant hex digits of an address show the page number. This is why the direct mode is also called zero-page mode because this mode can only access page 0.

As another example, consider the following zero-page mode instruction:

```
lds $40      ; upper half of SP ← Mem ($0040), lower half of SP ← Mem ($0041)
```

Example 29. Assemble the following instruction, place it in the memory starting at address $4000, and execute it:

```
lds $40
```

The first byte is still the opcode. Following Rule 2, the opcode is determined: $DF.

So, this instruction is translated into $DF 40.

In Figure 24a, the assembled instruction is sitting in the memory starting at $4000. Let us assume that Figure 24a 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 24b.

Example 30. Assemble the following instruction, place it in the memory starting at address $4000, and execute it:

```
sts $40
```

The first byte is still the opcode. Following Rule 2, the opcode is determined: $5F.

So, this instruction is translated into $5F 40.

In Figure 25a, the assembled instruction is sitting in the memory starting at $4000. Let us assume that Figure 25a 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 25b.

Figure 26 shows a summary of the indexed addressing modes.

Note that a single instruction may use more than one addressing mode, as you will see in the next chapter.

Example 31. Instruction {movb #$81, $3000} copies (moves) constant $81 to the memory at address $3000. This instruction uses the immediate mode for the source and the extended mode for the destination.

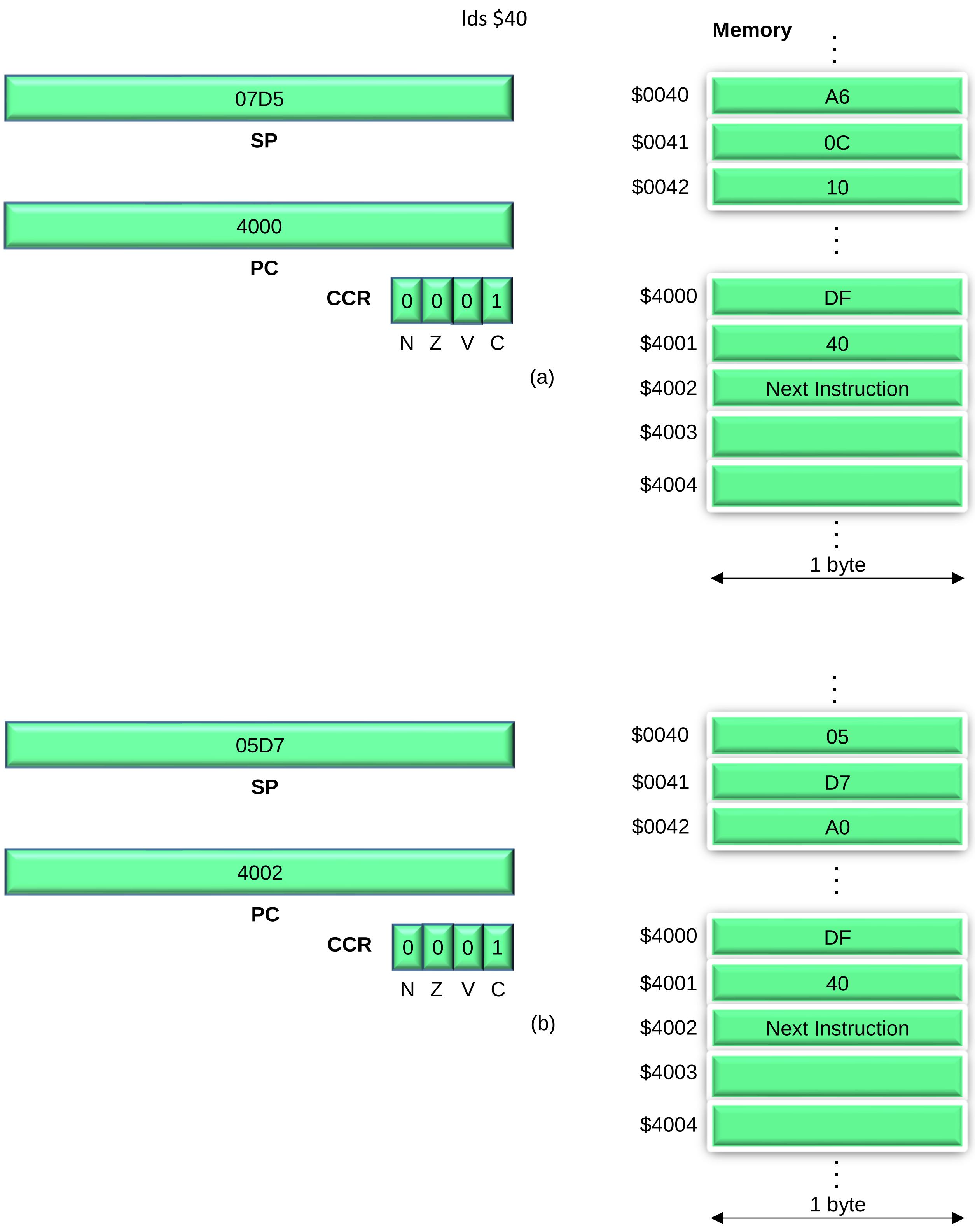
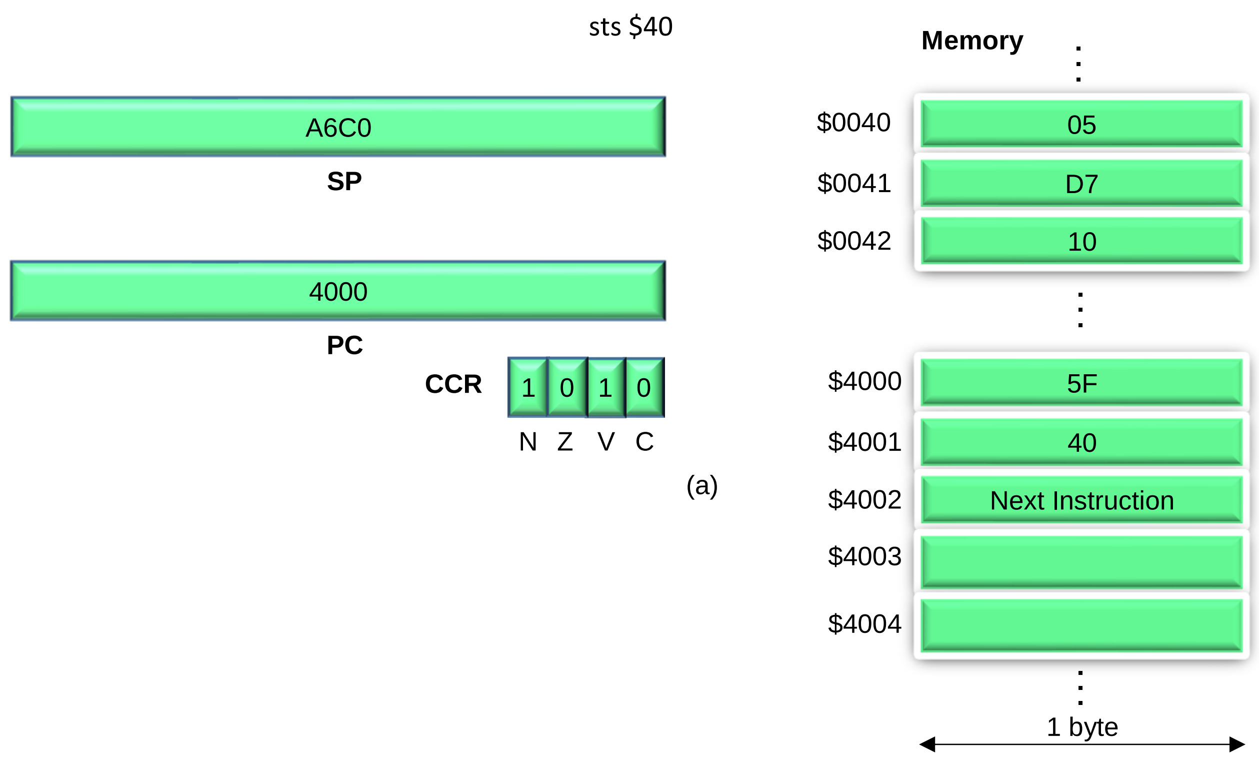


Figure 24. Instruction execution in Example 29.



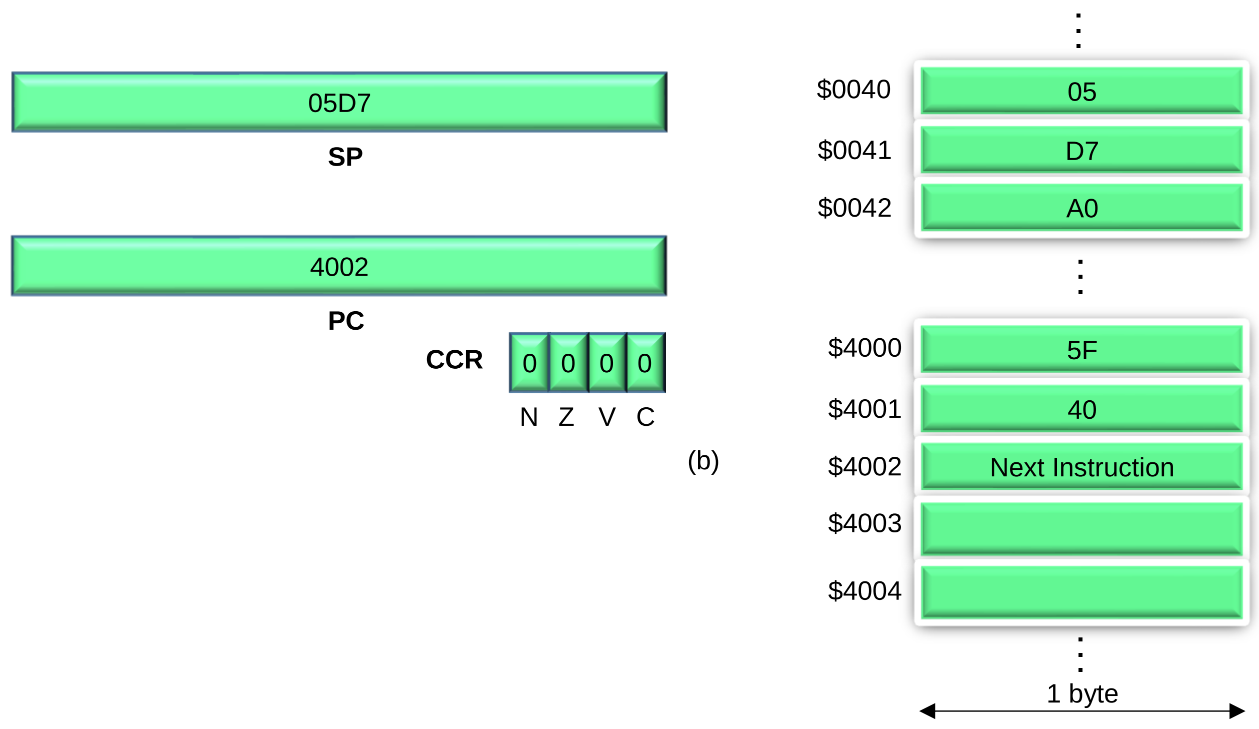


Figure 25. Instruction execution in Example 30.

	Addressing Modes	**Post-byte**	**Comments**
1	Indexed with a 5-bit Constant Offset: IDX	**rr** 0**nnnnn**	**nnnnn**: 5-bit signed offset
2	Indexed with a 9-bit Constant Offset: IDX1	111 **rr** 00**s**	**s**: MSb (sign bit) of offset
3	Indexed with a 16-bit Constant Offset: IDX2	111 **rr** 010	
4	Auto Pre-/Postdecrement/-increment Indexed with a Constant Offset: IDX	**rr** 1**pnnnn**	**rr** ≠ **11**; **P** = (**0:** Pre), (**1:** Post); If nnnn ≥ 0 then offset = nnnn +1
5	Indexed with Offset in an Accumulator: IDX	111 **rr** 1aa	aa = 00 (A), 01 (B), 10 (D)
6	Indexed **Indirect** with a 16-bit Constant Offset: [IDX2]	111 **rr** 011	
7	Indexed **Indirect** with an Offset in Accumulator D: [D, IDX]	111 **rr** 111	

Note: **rr**: 00 (X), 01 (Y), 10 (SP), 11 (PC)

Figure 26. Summary of indexed addressing modes