CHAPTER 4

LOGICAL OPERATING PRINCIPLES OF COMPUTERS

The most important parts in a computer are its processor and its main memory that can be read and written by the processor. For a computer to do something useful, there must be an executable program in its main memory. An executable program is a sequence of machine instructions which the processor of the computer can read and interpret.

In this chapter, we shall study machine instructions and some general operating principles of computers. First, we will examine the operation of the main memory. Then we will study a processor and programs which are executed by the processor. You will learn how some basic computing activities, like repetition and subroutine calling, are performed at the machine level. This will help you to understand high-level C# programming in later chapters.

The general operating principles of computers will be explained with the help of an imaginary computer which can be regarded as a logical model of real computers. All computers which are commonly used operate according to the same principles as the imaginary computer, although the imaginary computer is simpler than a real computer. The simplicity of the imaginary computer makes it an ideal instrument for learning the basics of computer operation. The computer is imaginary because it has not been built by using electronic or any other physical components. The Internet site of this book provides, however, simulation programs that imitate the imaginary computer on a real computer.

4.1 How does the main memory operate?

Computers are able to process information that is stored in their memory in binary form. There are basically two kinds of memory devices in a computer. The main memory is built of RAM memory devices. (RAM is an abbreviation of Random Access Memory.) All other memory devices can be considered auxiliary memory devices. The main memory of a computer is more important than the auxiliary memory devices because programs that are being executed must be kept in the main memory. A computer must have a main memory in order to operate. Although computers like PCs are equipped with auxiliary memory devices (e.g. hard disk), it is possible to build computers without the auxiliary devices.

The main memory of a computer is a device which can be read from and written to by the processor of the computer. Figure 4-1 illustrates a small main memory that is only 16 bytes (16 x 8 bits) in size. Computers generally have a much larger main memory, but we can study the memory operations with just this small main memory. Figure 4-1 shows that four address lines A0, A1, A2, and A3 are needed to select one of the 16 bytes in the memory. The memory addressing control takes care that the right memory location is selected when a certain bit combination is switched to the address lines.

While studying the memory device in Figure 4-1, we suppose that it can be used by switching either zero Volts (0V) or five Volts (5V) to the lines of the device. By switching various voltages to the address lines, it is possible to select certain memory locations, for example in the following way:

A3	A2	A1	A0	MEANING
0V	0V	0V	0V	memory address 0 is selected
0V	0V	0V	5V	memory address 1 is selected
0V	5V	0V	5V	memory address 5 is selected
5V	0V	0V	5V	memory address 9 is selected
5V	5V	5V	5V	memory address 15 is selected

The main memory of a computer has data lines through which information is either moved into the memory (writing of data), or information is moved out of the memory (reading of data). In Figure 4-1 there are 8 data lines D0, D1, D2, D3, D4, D5, D6, and D7 through which one byte of information can either be written to or read from the memory.

Address lines are used to select the desired location in the memory, and data lines are needed to carry information to/from the memory. In addition to address and data lines, there are usually control lines and power supply lines in memory devices. The control lines ensure that the reading and writing operations are carried out in an accurate manner. The memory device in Figure 4-1 has two control lines which have names READ MEMORY and WRITE MEMORY. With these lines (signals) the processor which is using the memory device can perform either a writing operation or a reading operation. Power supply lines are needed to supply electricity for physical memory components, but for simplicity these lines are left out from Figure 4-1.

The control signals of a memory device are activated by the processor that is using the memory. The control signals of the memory device in Figure 4-1 can have the following values and meanings

WRITE MEMORY	READ MEMORY	MEANING
0 (0V)	0 (0V)	No memory operations
0 (0V)	1 (5V)	Read selected memory address
1 (5V)	0 (0V)	Write selected memory address
1 (5V)	1 (5V)	Not allowed combination

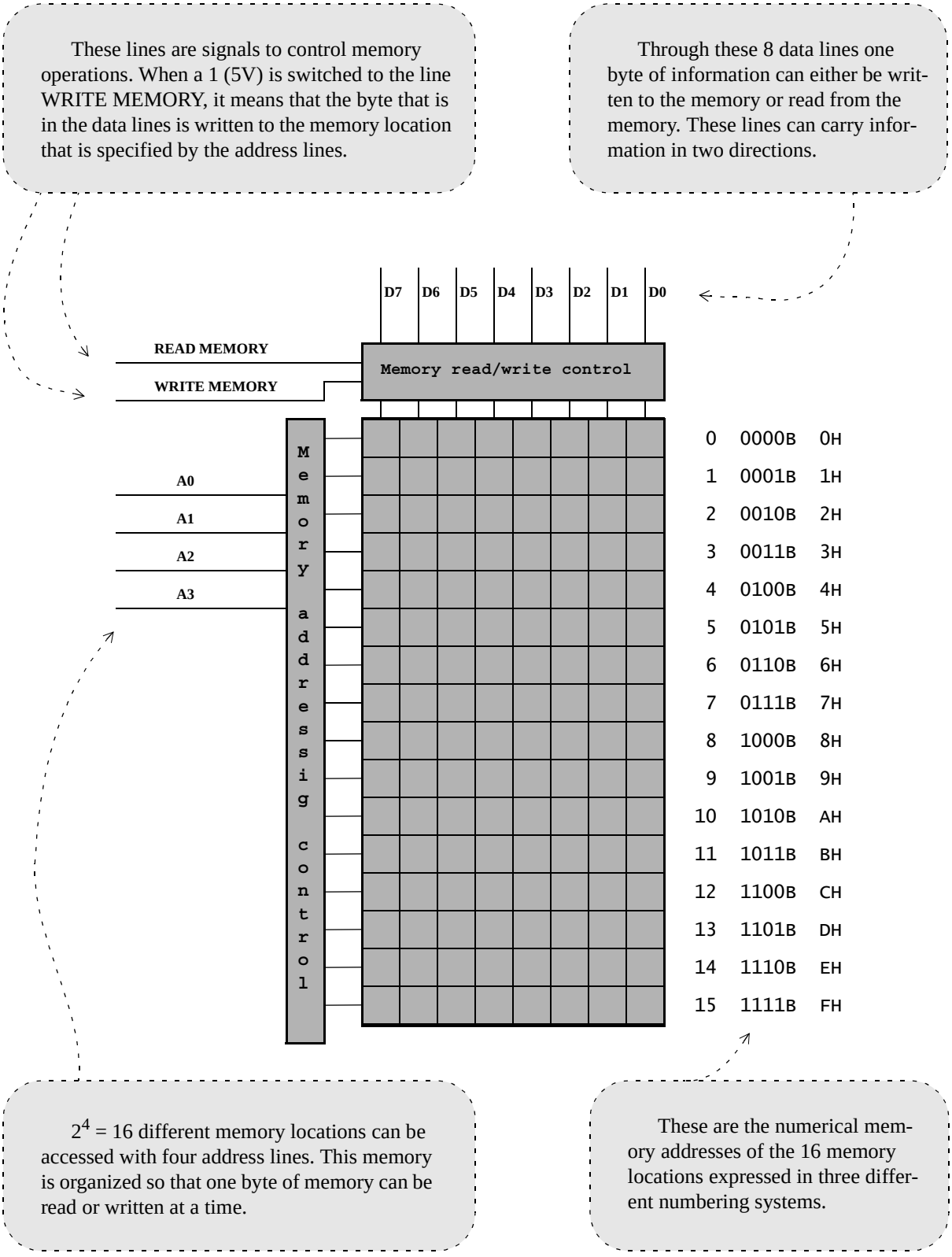


Figure 4-1. A theoretical 16-byte main memory device.

For example, if a processor wants to write the binary number 01101110B into memory address 5 of the memory device in Figure 4-1, it must switch the following voltages to the various lines:

Address lines	Data lines	Control lines	
		WRITE	READ
A3 A2 A1 A0	D7 D6 D5 D4 D3 D2 D1 D0	MEMORY	MEMORY
0V 5V 0V 5V	0V 5V 5V 0V 5V 5V 5V 0V	5V	0V

With the above voltages in the address lines, the address 0101B (5 decimal) is selected. When address 0101B is switched to the address lines and the WRITE MEMORY signal is active, the binary number that is switched to the data lines is stored in memory address 5 (0101B) and the information that was previously in that location is written over and lost. This is how writing to the main memory takes place. The WRITE MEMORY signal must be 1 (5V) and the READ MEMORY signal must be 0 (0V) at the moment when data is stored in the memory.

Different memory locations can be written by switching different addresses (different voltage combinations) to the address lines. Figure 4-2 shows a timing diagram that describes four writing operations on the memory device of Figure 4-1. Time goes from left to right in the diagram, and voltages are altered in all of the lines that enter the memory device. Note that the address and input data are always changed before the actual writing takes place. Data is written when the WRITE MEMORY signal goes up. For you to understand Figure 4-2 properly, it may be useful to mark zeroes or ones on those points where input signals change state.

A memory device like the one in Figure 4-1 keeps its contents as long as no memory locations are written over in writing operations. A reading operation cannot change the data stored in a memory device. When a reading operation is activated, the data lines of a memory device work in the opposite direction as for a writing operation. A memory location in the memory device of Figure 4-1 can be read by setting the address of the memory location on the address lines and activating the READ MEMORY signal. For example, if the lines of the memory device would be set to voltages

A3 A2 A1 A0	WRITE MEMORY	READ MEMORY
0V 5V 5V 0V	0V	5V

the data in the memory address 6 (0110B) would be copied to the data lines. The processor which is using the memory device could then read the data from the data lines.

Exercise related to main memory usage

Exercise 4-1. Below, write the voltages that need to be switched to address and data lines in Figure 4-1 in order to write the decimal number 97 into memory address 9 of the memory device. Which voltages must simultaneously be switched to the control lines?

Address lines	Data lines
A3 A2 A1 A0	D7 D6 D5 D4 D3 D2 D1 D0

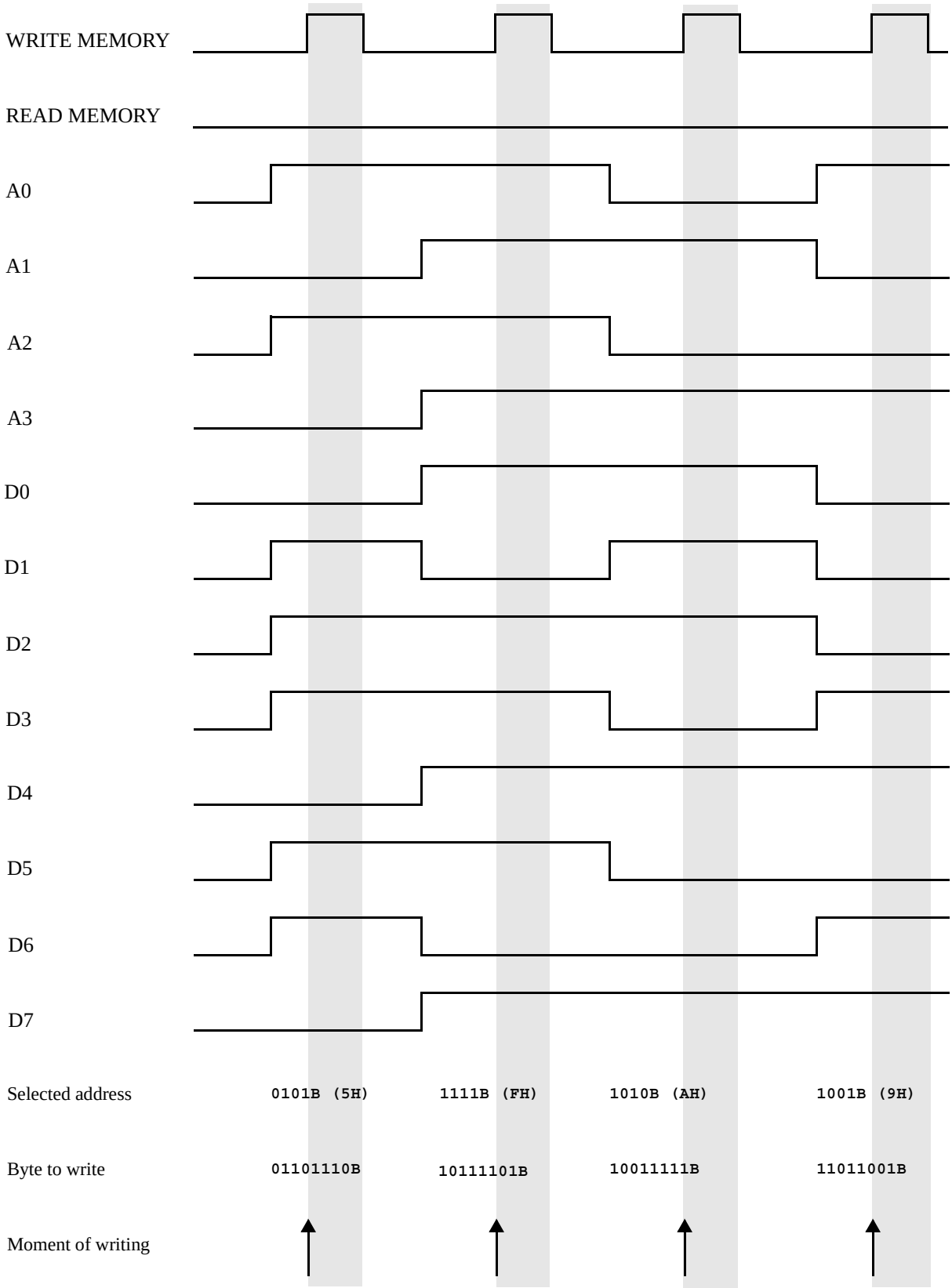


Figure 4-2. Writing four memory locations in the memory device of Figure 4-1.

4.2 The components of an imaginary computer

Now, after having studied how the main memory of a computer can be accessed, we can start investigating the imaginary computer presented in Figure 4-3. This computer is an imaginary one, because it exists only on these pages and in our minds. It would be possible to use electronic components to build this kind of a computer, but because the imaginary computer has been created solely for educational purposes, a "real" imaginary computer would not be as good as the existing commercially available computers. The imaginary computer in Figure 4-3 is a general model of a computer, and it teaches us how real computers work. The Internet site of this book provides simulation programs which show how the imaginary computer would behave if it were built.

As shown in Figure 4-3, the imaginary computer consists of a main memory, a processor, a keyboard, and a screen. The keyboard and the screen are connected by cables to the processor. The keyboard is an input device through which input data can be supplied to the processor. For example, if the user of the imaginary computer presses key A on the keyboard, the character code of letter A, 41H, is transferred to the processor. Similarly, the processor can output data by sending character codes to the screen. The screen is an output device which shows the character codes as visible characters. For example, if the processor sends code 42H to the screen, it appears as letter B on the screen.

The main memory in Figure 4-3 is similar to the smaller memory that we saw in Figure 4-1. The difference is that the main memory of our imaginary computer is 16 times larger than the memory in Figure 4-1. The imaginary computer has 8 address lines A0, A1, A2, A3, A4, A5, A6, and A7 which make it capable of addressing all 256 bytes of the main memory. Note that not all memory locations are shown in Figure 4-3. The main memory of our imaginary computer is very small when compared to modern commercially available computers, whose main memory consists of millions of bytes.

The imaginary processor uses two control signals, READ MEMORY and WRITE MEMORY, to either read or write data from/to the main memory. The processor can access one byte of memory at a time. For example, if the processor wants to access the memory location A2H (10100010B), it has to set the following voltages to the address lines

```
A7  A6  A5  A4  A3  A2  A1  A0      address lines

5V  0V  5V  0V  0V  0V  5V  0V      voltages

1   0   1   0   0   0   1   0       binary memory address
```

When a memory location has been selected by switching appropriate voltages to the address lines, the processor can read the contents of that memory location by setting the READ MEMORY signal to 1 (5V), or it can write the memory location by setting the WRITE MEMORY signal to 1 (5V). In a reading operation, the contents of the selected memory location are copied as 0V and 5V voltages to the data lines D0, D1, D2, D3, D4, D5, D6, and D7. In a writing operation, the voltages on the data lines are copied to the selected memory location, and the old contents of the memory location are lost.

The processor of the imaginary computer is a complex device which is capable of switching voltages of 0V or 5V on the address, data, and control lines. In some situations it can read voltages from the data lines. 0V means binary 0 and 5V means binary 1 for the processor. The processor is made active by supplying a clock signal to it. The clock signal is such that its voltage varies constantly between zero and one. If the clock signal goes from zero to one and back to zero 200 times in a second, we say that the processor uses a 200 Hz clock. As we shall learn later, the processor keeps repeating a certain sequence when it is working. The clock signal determines how fast the processor does its job. In our imaginary computer the speed of the processor is not essential, but we must imagine that there is a clock signal which makes the processor active.

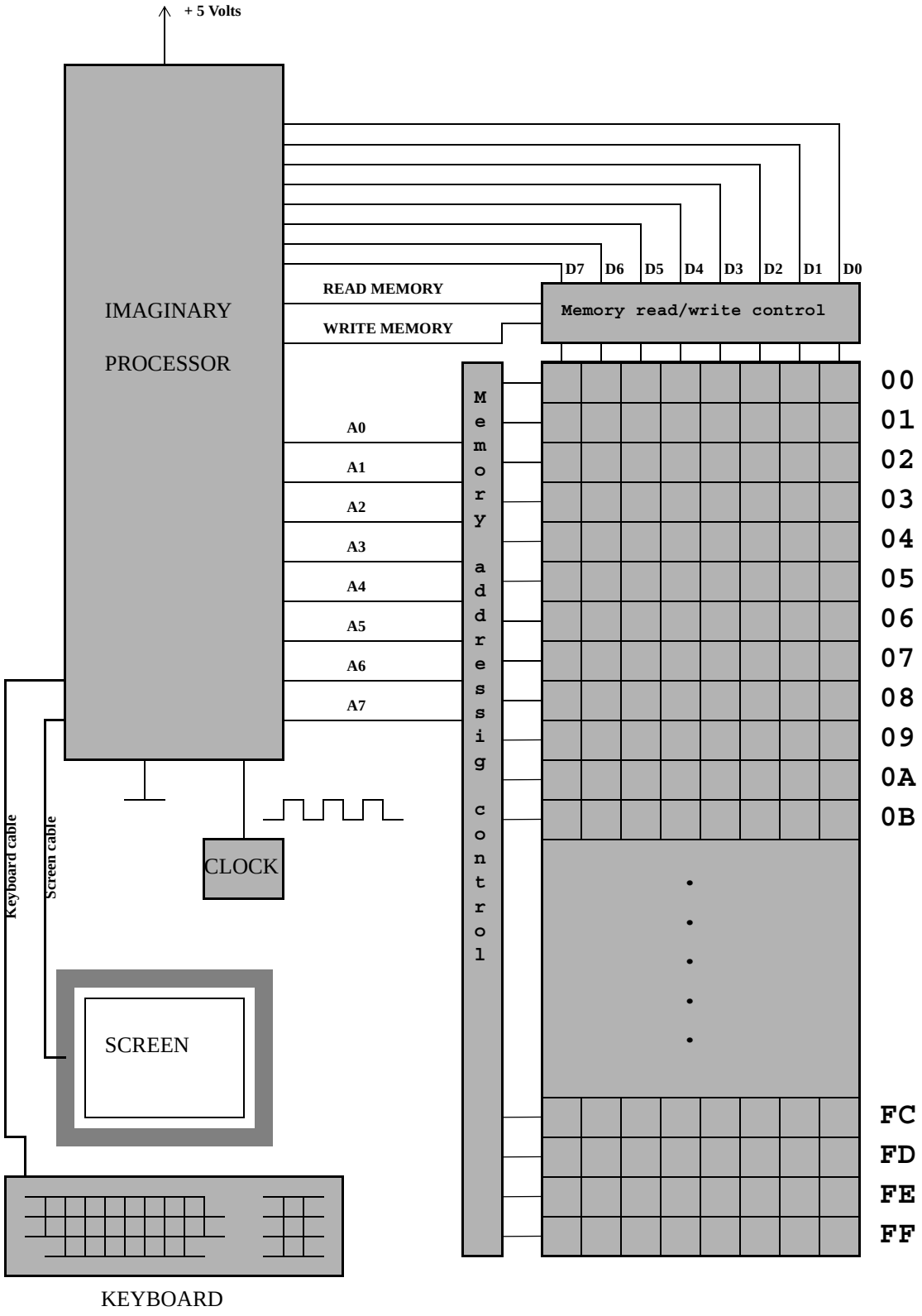


Figure 4-3. An imaginary computer with a 256-byte main memory.

4.3 Inside the imaginary processor

A processor is certainly the most complex electronic device inside a computer. Processors control nearly all activities in computers. A processor receives a clock signal according to which all activities are performed in a strict time schedule. Processors are called by that name because they process machine instructions which are placed in the main memory. A sequence of machine instructions in the main memory is a program which defines the behavior of the processor. Although it is a complex and fine device, a processor cannot do anything clever without the instructions that are given in a program.

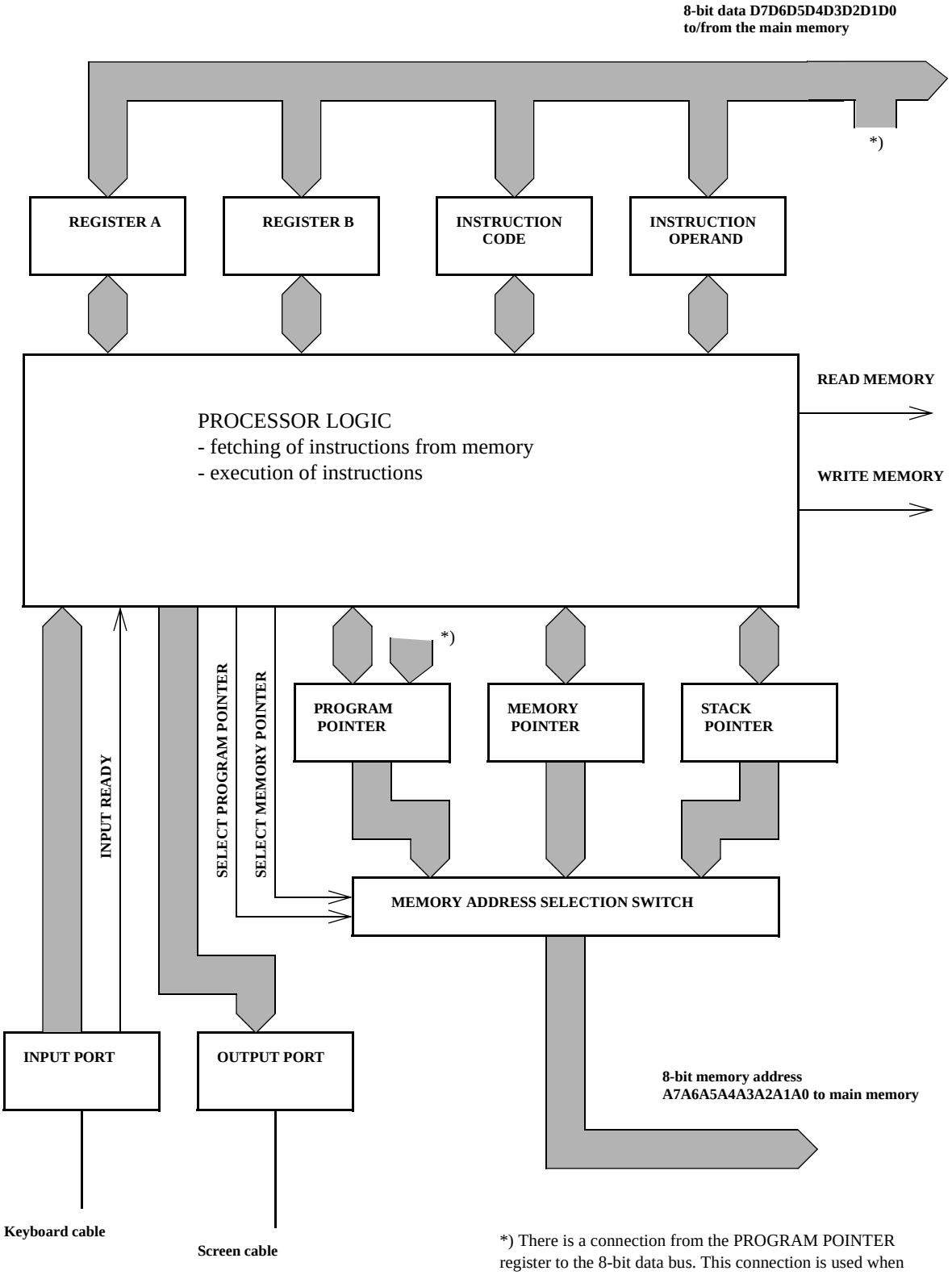
Figure 4-4 describes the main parts of the imaginary processor which is the brain of our imaginary computer. There are many lines and signals which can be found both in Figure 4-4 and Figure 4-3. The eight data lines and the eight address lines that connect the processor to the main memory are drawn as two thick lines in Figure 4-4. These thick lines are called buses in processor technology. Our imaginary processor has an 8-bit data bus and 8-bit address bus. The smaller rectangles in Figure 4-4 are registers that can store 8-bit numerical values. All registers are connected with an 8-bit bus to the PROCESSOR LOGIC that controls the processor. Because the internal data bus in the imaginary processor is an 8-bit bus, and all the internal registers inside the processor are 8-bit registers, we can say that the processor is an 8-bit processor, or the processor has an 8-bit architecture.

The internal registers of a processor, such as REGISTER A and REGISTER B in Figure 4-4, are small pieces of memory to store numerical values in binary form. As the registers of the imaginary processor are 8-bit registers, they can store numerical values 0, 1, 2, 3, 4, ..., 253, 254, and 255. The operation of the imaginary processor and the entire imaginary computer is based on elementary actions with numerical values stored in the internal registers. The PROCESSOR LOGIC can perform the following kinds of actions with the values stored in the internal registers:

- The content of one register can be copied to another register.
- The content of a register can be copied to a certain location in the main memory of the computer.
- A byte stored in a location in the main memory can be copied to an internal register of the processor.
- A numerical value stored in one register can be added to the numerical value in another register.
- The numerical value stored in a register can be incremented by one.
- It is possible to find out which one of two registers contains a larger numerical value.

Although all registers inside the imaginary processor are 8-bit registers, different registers have different roles in relation to the operation of the processor. The registers of the imaginary processor can be classified in the following way:

- REGISTER A and REGISTER B are general-purpose registers which can be used to transfer data to/from the main memory. Arithmetic operations can be carried out with the values in REGISTER A and REGISTER B.
- Registers INSTRUCTION CODE and INSTRUCTION OPERAND are used to store a machine instruction and its operand.
- Registers PROGRAM POINTER, MEMORY POINTER, and STACK POINTER are used to store memory addresses for different purposes. PROGRAM POINTER contains the address of the next instruction in a program. MEMORY POINTER is needed to move a byte between the general-purpose registers and the main memory. STACK POINTER contains the address of the first free position on the stack.



*) There is a connection from the PROGRAM POINTER register to the 8-bit data bus. This connection is used when the content of PROGRAM POINTER is stored to stack during subroutine calls.

Figure 4-4. The internal structure of the imaginary processor.

- INPUT PORT and OUTPUT PORT are different from the other registers. They are ports through which the processor can communicate with the outside world. The screen cable is connected to the OUTPUT PORT, and the keyboard cable is connected to the INPUT PORT. When a character code is written to the OUTPUT PORT, a character corresponding to that character code appears on the screen. When the user of the imaginary computer presses a key on the keyboard, the character code corresponding to the pressed key is transferred to the INPUT PORT.

The largest part of the processor in Figure 4-4 is the rectangle that is labeled PROCESSOR LOGIC. We can imagine that PROCESSOR LOGIC contains all necessary electronic circuitry to control the entire processor. The machine instructions, which we will study in the next section, are interpreted and processed by the PROCESSOR LOGIC. At this phase the role of PROCESSOR LOGIC may be somewhat obscure, but it will become clearer when we learn how the processor operates when it executes a program.

Various signals are drawn in Figure 4-4. The purpose of electronic signals in processors is to synchronize processor activities. The signals drawn in Figure 4-4 have the following meanings:

- With signals READ MEMORY and WRITE MEMORY the processor tells the main memory whether it wants to read from or write to it.
- Signal INPUT READY is an input signal that is sent from the INPUT PORT to the PROCESSOR LOGIC. This signal is needed to ensure that the processor reads an input character only once. By setting this signal to value 1, the INPUT PORT says to PROCESSOR LOGIC that it has received a character from the keyboard, and the character code is available in the INPUT PORT. The value of INPUT READY is set to 0 when the character is read away from the INPUT PORT.
- Signals SELECT PROGRAM POINTER and SELECT MEMORY POINTER are used to control the MEMORY ADDRESS SELECTION SWITCH that can connect one of the three pointer registers to the address lines which go to the main memory. These signals are needed because the pointer registers PROGRAM POINTER, MEMORY POINTER, and STACK POINTER can contain different memory addresses of which only one address can be used at a time. These signals work so that if SELECT PROGRAM POINTER is set to value 1 and SELECT MEMORY POINTER is set to value 0, register PROGRAM POINTER is selected. If SELECT PROGRAM POINTER has value 0 and SELECT MEMORY POINTER has value 1, register MEMORY POINTER is selected. If both signals have value 0 or both signals have value 1, register STACK POINTER is selected.

4.4 Machine instructions

Computers are machines that process information. For this reason, the most elementary instructions that are interpreted by a processor of a computer are called machine instructions. A series of machine instructions form a program that can be executed by a processor. The program is placed in the main memory, from where the processor can automatically read the program, instruction by instruction.

The machine instructions of a processor are unique numerical codes which cause the processor to act in a certain way. Machine instructions determine what the processor does. We can say that a processor can understand machine instructions which are fed to it via the main memory. Machine instructions thus form a machine language which is uniquely meaningful to the processor.

The machine instructions of our imaginary processor consist of either one or two bytes. The first byte is the numerical instruction code, according to which the processor determines what it should do next. Each machine instruction has a different numerical instruction code. The codes must be different, because otherwise the processor could not distinguish its machine instructions from one another. Some machine instructions, like the one in Figure 4-5, are two bytes long. In these instructions, the second byte is an operand for the actual machine instruction, the first byte. The operand byte can be a numerical value to be used in an arithmetic operation, or it can be a memory address.

The machine instructions of the imaginary computer are small pieces of information which the processor reads from the main memory. A complete computer program of the imaginary processor consists of a sequential list of these small pieces of information.When the processor reads the machine instructions from the main memory, it places them in the registers INSTRUCTION CODE and INSTRUCTION OPERAND. When the processor has done what is required by the current instruction contained in these registers, it copies a new instruction to these registers, and starts processing that. This kind of one-instruction-at-a-time processing goes on until the processor finds a "stop processing" instruction in the main memory.

All machine instructions of the imaginary processor are listed in Table 4-1. The table contains all the numerical instruction codes, and explains the meaning of each for the programmer who creates computer programs by using these instructions. The last column of the table describes what actions the processor performs when it processes each instruction. The table is called an "instruction decoding table" because we can imagine that there is such a table, in electronic form, inside the PROCESSOR LOGIC of the imaginary processor. We can think that when the imaginary processor reads a machine instruction from the main memory, it consults this kind of a table and, according to what is said in this "instruc-

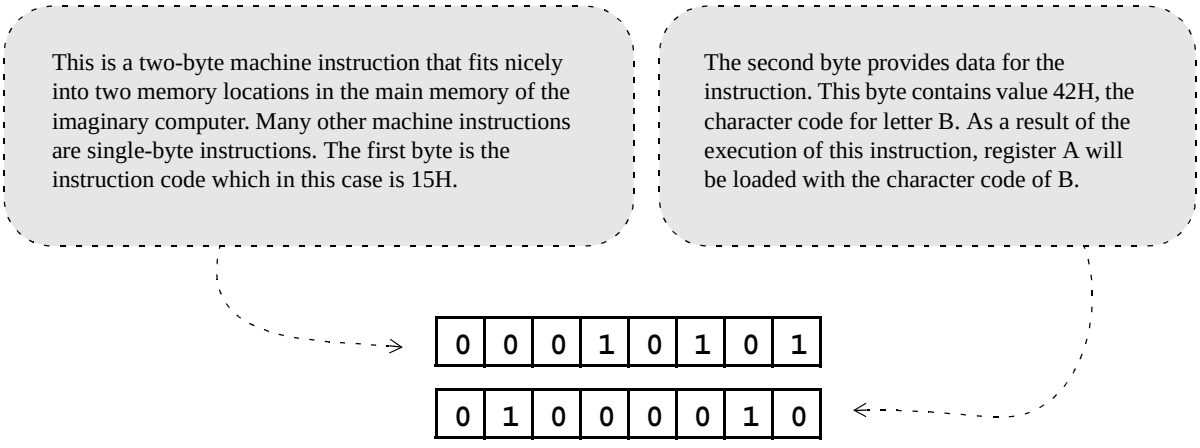


Figure 4-5. Instruction "load register A with value" of the imaginary computer.

tion decoding table", it then determines what it should do. For example, if the processor read the instruction 14H, it would use the fourth line of the instruction decoding table because the instruction with code 14H is explained there. The processor would find out that it must subtract the content of REGISTER B from the content of REGISTER A, and place the result in REGISTER A.

You will need to consult Table 4-1 when we start studying example computer programs in the next section. The nature of the machine instructions will become clearer when you learn how the imaginary computer actually processes a program. At this point, you can find out that there are seven categories of instructions in Table 4-1. The category an instruction belongs to can be identified from the first hexadecimal digit of the instruction code. For example, instructions for arithmetic and moving operations have codes that start with 1, and the codes of memory-related instructions start with 2. When an instruction is a two-byte instruction, its instruction code is odd. Single-byte instructions have even instruction codes. The operand byte of two-byte instructions is marked either VVH or MMH in Table 4-1. The values of operand bytes can be anything from 0 to FFH. The creator of a program determines which values are needed as operands for various machine instructions. VVH means a value to be loaded to some register or to be used in an arithmetic operation. MMH means a memory address which is needed by the instruction.

Table 4-1: Instruction decoding table of the imaginary processor .

Instruction code and optional operand.	Meaning for the programmer	Actions taken by the processor
	INSTRUCTIONS FOR ARITHMETIC AND MOVING OPERATIONS	
11H VVH	"add value to register A"	The value VVH, the operand of this instruction, will be added to the content of register A. The content of register A will thus be incremented by the given value. For example, if the content of register A is 34H and VVH is 05H, the content of register A is 39H after the execution of this instruction.
12H	"add register B to A"	The content of register B is added to the content of register A and the result (the sum of the two registers) is placed in register A. The old content of register A is lost while the content of register B remains untouched.
13H VVH	"subtract value from register A"	The value VVH, the operand of this instruction, will be subtracted from register A. The content of register A will thus be decremented by the given value. For example, if the content of register A is 34H and VVH is 05H, the content of register A is 2FH after this instruction is executed.
14H	"subtract register B from A"	The content of register B is subtracted from the content of register A. The result (A minus B) is placed in register A, while the content of register B remains untouched. For example, if register A has value 34H and register B is 12H, register A contains 22H and register B contains 12H after this instruction is executed.

Table 4-1: Instruction decoding table of the imaginary processor (Continued).

15H VVH	"load register A with value"	The value VVH, the operand byte, will be copied to register A. The old content of register A will be lost. For example, if the value VVH is 05H, the content of register A will be 05H regardless of its previous value.
16H	"increment register A"	The content of register A will be incremented by 1. For example, if the content of register A is 34H, its content will be 35H after the execution of this instruction.
17H VVH	"load register B with value"	The value VVH will be copied to register B. The old content of register B will be written over and lost. This instruction is similar to 15H but this one affects register B.
18H	"increment register B"	The content of register B will be incremented by 1. For example, if the content of register B is 12H, its content will be 13H when this instruction is executed.
1AH	"decrement register A"	The content of register A will be decremented by 1. For example, if the content of register A is 34H, it will change to 33H as a result of this instruction.
1CH	"decrement register B"	The content of register B will be decremented by 1. For example, if the content of register B is 12H, it will be 11H after the execution of this instruction.
1EH	"move content of register A to B"	The content of register A is copied to register B. The old content of register B is written over. After this instruction has been executed, both registers have the same content.
	MEMORY-RELATED INSTRUCTIONS	
21H MMH	"set memory pointer"	The memory address MMH that is given as the operand for this machine instruction is copied to register MEMORY POINTER. With this instruction, it is possible to start manipulating new areas of the computer's main memory. The content of MEMORY POINTER determines which memory address is affected by instructions "store register A to memory", "load register A from memory", "store register B to memory", and "load register B from memory".
22H	"increment memory pointer"	The content of MEMORY POINTER is incremented by 1. For example, if the old content of MEMORY POINTER is 9BH, its new content will be 9CH.
24H	"decrement memory pointer"	The content of MEMORY POINTER is decremented by 1. For example, if the content of MEMORY POINTER is B3H, its content will change to B2H.
26H	"store register A to memory"	The content of register A will be written to that memory location which is currently the content of MEMORY POINTER. For example, if the content of register A is 34H and the content of MEMORY POINTER is 9BH, the memory location in address 9BH will contain value 34H after this instruction has been executed.

Table 4-1: Instruction decoding table of the imaginary processor (Continued).

28H	"load register A from memory"	This is the opposite of instruction "store register A to memory". This instruction copies one byte from the main memory to register A. The byte that will be copied will be determined by the content of MEMORY POINTER. For example, if the content of MEMORY POINTER is B3H, this instruction loads register A with the byte that is in memory location B3H.
2AH	"store register B to memory"	This is similar to the instruction with code 26H, but this copies the contents of register B to a location in the main memory.
2CH	"load register B from memory"	This is similar to the instruction with code 28H, but this instruction modifies the content of register B.
	INSTRUCTIONS RELATED TO JUMPING IN PROGRAMS	These instructions modify the content of PROGRAM POINTER. "jump to address" modifies it always. Other instructions modify the content of PROGRAM POINTER only if certain conditions are valid. All these instructions are two-byte instructions. Instruction operand MMH is a possible new value for PROGRAM POINTER.
41H MMH	"jump to address"	This instruction performs an unconditional jump in the program by modifying the content of register PROGRAM POINTER. The next instruction that will be executed after this instruction is the one that resides in the address given in MMH, the operand of this instruction. For example, if the value of MMH is 08H, the value of PROGRAM POINTER is 08H after this instruction has been executed.
43H MMH	"jump if registers equal"	This instruction is a conditional jump within the program. If the contents of register A and register B are the same, MMH is copied to PROGRAM POINTER, and the next instruction to be executed is the one in the address that is given in the operand of this instruction.
45H MMH	"jump if register A zero"	This is another conditional jump. If the content of register A is zero, PROGRAM POINTER is loaded with the value MMH. If register A is not equal to zero, the program execution continues in the normal way from the instruction that follows this instruction.
47H MMH	"jump if register A smaller than B"	A jump to memory location MMH occurs if the content of register A is smaller than the content of register B. If register A is larger than or equal to register A, no jump takes place, and the program execution continues from the instruction that follows this instruction.
49H MMH	"jump if register A greater than B"	This is a kind of opposite instruction to the previous one. A jump occurs when the content of register A is greater than the content of register B. For example, if register A is 34H, register B is 12H, and the operand byte MMH is 08H, the value of PROGRAM POINTER is changed to 08H because 34H is greater than 12H.
4BH MMH	"jump if input not ready"	This instruction tests whether a byte is ready to be read from the input port. This instruction causes a jump if the input is not ready, i.e., INPUT READY signal has value 0, which means logically FALSE. This instruction is used when a program is waiting for a human to give it a byte of data. Because humans tend to be slower than computers, a computer program usually has to jump and wait until the user of the computer has entered its input.

Table 4-1: Instruction decoding table of the imaginary processor (Continued).

	INSTRUCTIONS TO HANDLE SUBROUTINE CALLS	
81H MMH	"call subroutine"	The execution of this instruction causes a jump to address MMH but the execution will eventually return to the instruction that follows this instruction. This kind of behavior is accomplished by storing the current value of PROGRAM POINTER to the stack. Three separate actions take places when this instruction is executed. First the content of register PROGRAM POINTER is stored to that memory address which is the content of register STACK POINTER. Then the content of register STACK POINTER is decremented by one. Finally, value MMH is copied to register PROGRAM POINTER. This way the program execution continues from address MMH. PROGRAM POINTER is stored on the stack to be used by the subroutine which starts in address MMH. Instruction "call subroutine" can be explained with the phrase "go and execute the machine instructions starting from address MMH, but come back when you encounter the instruction code 82H".
82H	"return to calling program"	This instruction is a kind of counterpart to the instruction "call subroutine". "return to calling program" must be the last instruction in a subroutine. It marks the end of a called subroutine and causes a return to the calling program. This instruction loads PROGRAM POINTER with the memory address that was put to the stack by instruction "call subroutine". This causes a return to the instruction that immediately follows the "call subroutine" instruction in the calling program. The following two actions take place when this instruction is executed: The value in register STACK POINTER is incremented by one. Register PROGRAM POINTER is loaded from the memory address which is the content of register STACK POINTER.
	INPUT/OUTPUT INSTRUCTIONS	
92H	"output byte from register A"	This instruction is used when a program wants to output a character to the screen. The instruction copies the content of register A to OUTPUT PORT. A character code must be present in register A before this instruction can be successfully executed. The result of the execution is that the character corresponding to the character code in register A is displayed on the screen. The OUTPUT PORT works so that always when a character code is written to it, the corresponding character appears on the screen.
94H	"output byte from register B"	This is similar to the previous instruction, but this one outputs the content of register B.
96H	"input byte to register A"	This instruction is used when a program wants to input a character code from the keyboard to register A. The instruction copies a byte from the INPUT PORT to register A. Signal INPUT READY indicates whether a character code of a character has been received from the keyboard to INPUT PORT. INPUT READY must have value 1 (i.e. TRUE) before this instruction can be executed. After the input character code has been copied to register A, INPUT READY is set back to zero.

Table 4-1: Instruction decoding table of the imaginary processor (Continued).

	STACK INSTRUCTIONS	
A1H MMH	"set stack pointer"	This instruction is rarely needed because register STACK POINTER is set to value FFH when the imaginary computer is switched on. The last bytes of the main memory are thus used as stack. With this instruction it is possible to select a new memory area to be used as stack memory. Value MMH is copied to register STACK POINTER when this instruction is executed.
A2H	"push register A to stack"	This instruction stores (pushes) register A to the top of the stack. Register STACK POINTER always contains a value that "points" to the first free memory location on the stack. This instruction causes two separate actions: first the content of register A is stored to that memory address which is the content of register STACK POINTER, and then the memory address in STACK POINTER is decremented by one. The stack thus grows towards the smaller memory addresses. After this instruction has been executed, the topmost element on the stack and register A contain the same information.
A4H	"pop register A from stack"	This instruction performs an opposite operation compared to the actions caused by the previous instruction. First, the memory address in register STACK POINTER is incremented by one, and then register A is loaded from the memory address that is in register STACK POINTER. This instruction takes away that byte that was the last one pushed to the stack. The stack gets smaller when this instruction is executed.
	INSTRUCTION TO HALT THE PROCESSOR AT THE END OF A PROGRAM	
B2H	"stop processing"	This is a special kind of instruction which stops the imaginary processor entirely. This instruction marks the end of the program. No more instructions will be executed after this one until imaginary electricity is switched off and on again.

A classification of computers

Both the technology and terminology related to computers are evolving rapidly, and there exist different terms to describe different types of computers. Much of this terminology may be somewhat confusing. The following list may reduce the confusion to some extent.

- Supercomputers are large computers and they are the fastest computers in the world. Supercomputers have been built by using special electronic components and processors. Supercomputers are used to run applications that perform massive calculations. Such applications include various scientific and military applications. Weather forecasting is one application domain where supercomputers have traditionally been used.
- Mainframe computers are those computers that, at least some decades ago, tended to occupy entire air-conditioned rooms. Mainframe computers are used by large organizations like government offices, banks, and insurance companies who need to process massive amounts of data. The most famous provider of mainframe computers is without doubt IBM (International Business Machines).
- Minicomputers are something smaller than mainframe computers. The term "minicomputer" is not widely used any more, probably because minicomputers have become equally small as microcomputers or microcomputers have become equally efficient as the traditional minicomputers.
- The term "microcomputer" was brought into use to describe computers that could be placed on a table, and were built by using the commercially available microprocessors. Obviously the world's most well-known software company was originally named as Microsoft because the company produced software for microcomputers.
- The term "personal computer" (PC) was brought into use when IBM introduced its IBM Personal Computer in 1981. The original IBM PC was running an operating system from Microsoft. Nowadays, a personal computer (PC) is any computer that runs the Windows operating system.
- The term "workstation computer", or simply "workstation", has traditionally meant a powerful desktop computer that runs the UNIX operating system. Nowadays the distinction between a workstation and a personal computer is harder to make.
- A server computer serves other computers connected to a local network of computers. Server computers are typically connected to other servers. The Internet is a worldwide network of server computers. The Linux operating system has become popular in servers.
- Laptop computers are portable computers whose size is small, but they have a normal-size keyboard and display.
- Palmtop computers are small enough to be carried in a pocket. The display of a palmtop computer is small. Because they do not have a traditional keyboard, some palmtop computers are operated by using a special pen. Top models of modern mobile phones are like palmtop computers.
- Nanocomputers ... these are something that may exist in the future.
- Theoretical computers do not exist in reality. The imaginary computer that is presented in this chapter is an example of a theoretical computer, but there are many other theoretical computers. For example, in 1937 Alan Turing published an article that presented a theoretical computer that was later named the Turing machine. The architecture of the Turing machine differs significantly from the architecture of the modern computers. Therefore, the Turing machine is not appropriate for educational purposes these days.

4.5 The steps and states of program execution

Because the imaginary computer does not exist in reality, we must make some assumptions about how it would operate in reality. Since no computer can operate without a program in its main memory, we must assume that a program can somehow be loaded to the main memory of the imaginary computer. A program is always loaded so that the first instruction is in the memory address 00H, and other instructions of the program are in the subsequent memory addresses.

We assume also that there is an ON/OFF switch in the imaginary computer. When there is a program loaded in the main memory, and the computer is switched on, it starts executing the program according to the following steps:

STEP 1: Load value 00H to register PROGRAM POINTER, and value FFH to register STACK POINTER. So at the beginning PROGRAM POINTER points to the first location in the main memory, and STACK POINTER points to the last memory location. Go to STEP 2.

STEP 2: Fetch the instruction code of a machine instruction from that memory address which is stored in register PROGRAM POINTER. Store the instruction code to register INSTRUCTION CODE. Increment the address in register PROGRAM POINTER by one. Go to STEP 3.

STEP 3: If the content of register INSTRUCTION CODE is odd (i.e. if the instruction is a two-byte instruction), fetch a byte from the memory address which is stored in PROGRAM POINTER, store the byte to register INSTRUCTION OPERAND, and increment the address in register PROGRAM POINTER by one. Otherwise, if the content of INSTRUCTION code is even, do nothing. Go to STEP 4.

STEP 4: If the content of register INSTRUCTION CODE is B2H (the code for instruction "stop processing"), go to STEP 6. Otherwise, go to STEP 5.

STEP 5: Interpret the instruction by using the instruction decoding table (Table 4-1) and execute the actions required by that instruction. After all necessary actions are performed, go to STEP 2.

STEP 6: Stop processing. Do nothing. The end of the program has been encountered.

The six steps above describe what the processor of the imaginary computer is doing when it is running. What really happens is determined by the program that is stored in the main memory. The processor repeats steps from 2 to 5 over and over, depending on how many instructions and which instructions there are in the program.

A computer is a machine that can have different states while it is operating. Figure 4-6 is a diagram which describes the operation of the imaginary computer as a machine which changes states. The rounded rectangles in the figure are states in which certain activities are carried out. After the activities of a state have been performed, it is possible to make a transition to another state. The arrows describe transitions from one state to another. The texts in brackets near some arrows mean conditions under which the transition can occur. For example, the transition from state FETCH INSTRUCTION CODE to state FETCH INSTRUCTION OPERAND occurs only when the instruction code is odd. The first state of the computer is RESET PROCESSOR, and that state is entered only once. The final state of the computer is PROCESSOR STOPPED. That state is reached when the instruction code is B2H.

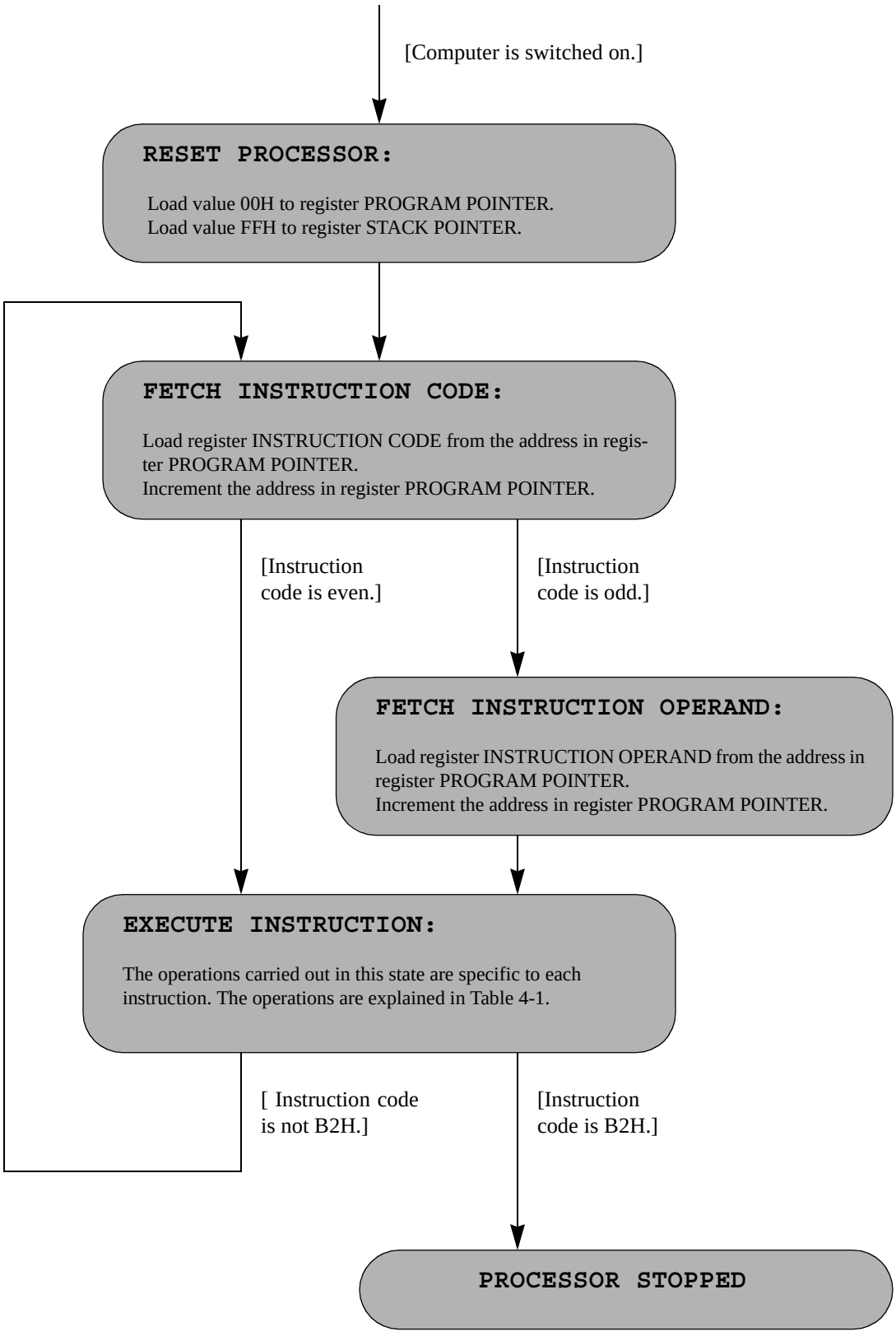


Figure 4-6. A state diagram that describes the operation of the imaginary computer.

4.6 Programs to print text "Hello!"

Perhaps the best way to understand the operation of computers is to study and run programs made for them. Therefore, we now start studying complete computer programs which are made for the imaginary computer. Our first example programs are machine-level programs because they are made of the machine instructions. Machine-level programs consist of numerical codes which the processor interprets.

Figure 4-7a shows a simple program loaded into the main memory of our computer. The program prints text "Hello!" to the screen of the imaginary computer. The program, which is 13 bytes long, starts in the memory address 00H, and the last byte of the program is in the address 12H. The program uses only three different machine instructions:

- Instruction 15H, "load register A with value", is used six times to load the character codes of different characters to register A. Instruction 15H is a two-byte instruction. The byte that follows the instruction in the program is an operand byte. (Remember that two-byte instructions have odd instruction codes!)
- Instruction 92H, "output byte from register A", is used six times to print each character to the screen.
- Instruction B2H, "stop processing", is used once at the end of the program to stop the imaginary processor.

Figure 4-7b describes the execution of the program in Figure 4-7a. The values of all registers of the imaginary processor are shown after all processor operations in Figure 4-7a. Most registers have value FFH when the computer starts operating. Registers are marked so that PP means PROGRAM POINTER, IC means INSTRUCTION CODE register, IO means INSTRUCTION OPERAND, RA means register A, etc. FE and FF mean the last two memory locations in the main memory. All numbers in both figures are hexadecimal numbers without the H at the end.

```
Address   Program    Explanation

00        15         "load register A with value"
01        48         character code of letter 'H'
02        92         "output byte from register A"
03        15         "load register A with value"
04        65         character code of letter 'e'
05        92         "output byte from register A"
06        15         "load register A with value"
07        6c         'l'
08        92         "output byte from register A"
09        15         "load register A with value"
0a        6c         'l'
0b        92         "output byte from register A"
0c        15         "load register A with value"
0d        6f         'o'
0e        92         "output byte from register A"
0f        15         "load register A with value"
10        21         '!'
11        92         "output byte from register A"
12        b2         "stop processing"
```

Figure 4-7a. A machine-level program that prints text "Hello!"

The first status of the processor is always RESET. Here the processor performs two fetch operations because the first machine instruction, 15H, "load register A with value", is a two-byte instruction. After the last fetch operation, register INSTRUCTION OPERAND (IO) contains value 48H which is the character code of uppercase letter H.

At the beginning, the screen of the computer is empty. Text "Hello!" emerges gradually on the screen, character by character, as the program execution proceeds.

Status	PP	IC	IO	RA	RB	MP	SP	FE	FF	Screen contents
RESET	00	ff	ff	ff	ff	ff	ff	ff	ff	
FETCH	01	15	ff	ff	ff	ff	ff	ff	ff	
FETCH	02	15	48	ff	ff	ff	ff	ff	ff	
EXECUTE	02	15	48	48	ff	ff	ff	ff	ff	
FETCH	03	92	48	48	ff	ff	ff	ff	ff	
EXECUTE	03	92	48	48	ff	ff	ff	ff	ff	H
FETCH	04	15	48	48	ff	ff	ff	ff	ff	H
FETCH	05	15	65	48	ff	ff	ff	ff	ff	H
EXECUTE	05	15	65	65	ff	ff	ff	ff	ff	H
FETCH	06	92	65	65	ff	ff	ff	ff	ff	H
EXECUTE	06	92	65	65	ff	ff	ff	ff	ff	He
FETCH	07	15	65	65	ff	ff	ff	ff	ff	He
FETCH	08	15	6c	65	ff	ff	ff	ff	ff	He
EXECUTE	08	15	6c	6c	ff	ff	ff	ff	ff	He
FETCH	09	92	6c	6c	ff	ff	ff	ff	ff	He
EXECUTE	09	92	6c	6c	ff	ff	ff	ff	ff	Hel
FETCH	0a	15	6c	6c	ff	ff	ff	ff	ff	Hel
FETCH	0b	15	6c	6c	ff	ff	ff	ff	ff	Hel
EXECUTE	0b	15	6c	6c	ff	ff	ff	ff	ff	Hel
FETCH	0c	92	6c	6c	ff	ff	ff	ff	ff	Hel
EXECUTE	0c	92	6c	6c	ff	ff	ff	ff	ff	Hell
FETCH	0d	15	6c	6c	ff	ff	ff	ff	ff	Hell
FETCH	0e	15	6f	6c	ff	ff	ff	ff	ff	Hell
EXECUTE	0e	15	6f	6f	ff	ff	ff	ff	ff	Hell
FETCH	0f	92	6f	6f	ff	ff	ff	ff	ff	Hell
EXECUTE	0f	92	6f	6f	ff	ff	ff	ff	ff	Hello
FETCH	10	15	6f	6f	ff	ff	ff	ff	ff	Hello
FETCH	11	15	21	6f	ff	ff	ff	ff	ff	Hello
EXECUTE	11	15	21	21	ff	ff	ff	ff	ff	Hello
FETCH	12	92	21	21	ff	ff	ff	ff	ff	Hello
EXECUTE	12	92	21	21	ff	ff	ff	ff	ff	Hello!
FETCH	13	b2	21	21	ff	ff	ff	ff	ff	Hello!
STOP	13	b2	21	21	ff	ff	ff	ff	ff	Hello!

This line shows the situation after the instruction 15H has been executed. The execution resulted in the value 21H, the character code of exclamation mark !, being copied from register INSTRUCTION OPERAND (IO) to register A (RA). Register PROGRAM POINTER (PP) already points to the address of the next instruction in the program.

Figure 4-7b. Step-by-step execution of the program in Figure 4-7a.

I advise you to study Figure 4-7b very carefully. You should read all the lines which show the values of the registers, and note changes in the values. It may be useful to re-read the processing steps which were discussed in the previous section. You should also read Table 4-1 to find out how the used machine instructions should work. If you can understand what is said in Figure 4-7b, you have learned the basics about the operation of the imaginary computer.

Usually there are several possibilities to construct a computer program that performs a certain activity. The program in Figure 4-7a is one way to print text "Hello!" to the screen. In Figure 4-8a, there is another program that prints the text "Hello!" to the screen of the imaginary computer. The program in Figure 4-7a is rather simple because the two instructions "load register A with value" and "output byte from register A" are repeated six times in it. The program in Figure 4-8a is constructed in such a way that necessary instructions are repeated six times although they exist only once in the program.

In the program in Figure 4-8a, the characters of the text "Hello!" are separated from the actual program. The character codes of the characters are in different memory area than the machine instructions. The last machine instruction is in address 09H, and the first character of the text is in address 0AH. The character codes are in the order in which they should be printed, and 00H, the NULL character, marks the end of the text to be printed.

The program in Figure 4-8a uses the following instructions not used in the program in Figure 4-7a:

- With instruction 21H, "set memory pointer", register MEMORY POINTER is set to point to address 0AH which is the beginning of text "Hello!" in the main memory.
- Instruction 28H, "load register A from memory" is used to read the characters of the text "Hello!" from the main memory. The value of MEMORY POINTER specifies which memory location is read by instruction "load register A from memory". Since MEMORY POINTER points to address 0AH at the beginning, the character code of letter H is the first value that is loaded to register A.
- Instruction 22H, "increment memory pointer", adds 1 to the value of MEMORY POINTER. This ensures that instruction "load register A from memory" loads different character codes from the main memory when the program is executed.

```
Address   Program    Explanation

00        21         "set memory pointer"
01        0a         value to register MEMORY POINTER
02        28         "load register A from memory"
03        45         "jump if register A zero"
04        09         address for possible jump
05        92         "output byte from register A"
06        22         "increment memory pointer "
07        41         "jump to address"
08        02         address for unconditional jump
09        b2         "stop processing"
0a        48         'H'
0b        65         'e'
0c        6c         'l'
0d        6c         'l'
0e        6f         'o'
0f        21         '!'
10        00         NULL  (zero)
```

Figure 4-8a. Another program that prints text "Hello!"

Here instruction 41H, "jump to address", is executed. The execution modifies the content of register PROGRAM POINTER so that the next instruction to be executed is in address 02H.

Execution of instruction 45H, "jump if register A zero", results in a jump only when register A contains value 00H.

Status	PP	IC	IO	RA	RB	MP	SP	FE	FF	Screen contents
RESET	00	ff	ff	ff	ff	ff	ff	ff	ff	
FETCH	01	21	ff	ff	ff	ff	ff	ff	ff	
FETCH	02	21	0a	ff	ff	ff	ff	ff	ff	
EXECUTE	02	21	0a	ff	ff	0a	ff	ff	ff	
FETCH	03	28	0a	ff	ff	0a	ff	ff	ff	
EXECUTE	03	28	0a	48	ff	0a	ff	ff	ff	
FETCH	04	45	0a	48	ff	0a	ff	ff	ff	
FETCH	05	45	09	48	ff	0a	ff	ff	ff	
EXECUTE	05	45	09	48	ff	0a	ff	ff	ff	
FETCH	06	92	09	48	ff	0a	ff	ff	ff	
EXECUTE	06	92	09	48	ff	0a	ff	ff	ff	H
FETCH	07	22	09	48	ff	0a	ff	ff	ff	H
EXECUTE	07	22	09	48	ff	0b	ff	ff	ff	H
FETCH	08	41	09	48	ff	0b	ff	ff	ff	H
FETCH	09	41	02	48	ff	0b	ff	ff	ff	H
EXECUTE	02	41	02	48	ff	0b	ff	ff	ff	H
FETCH	03	28	02	48	ff	0b	ff	ff	ff	H
EXECUTE	03	28	02	65	ff	0b	ff	ff	ff	H
FETCH	04	45	02	65	ff	0b	ff	ff	ff	H
FETCH	05	45	09	65	ff	0b	ff	ff	ff	H
EXECUTE	05	45	09	65	ff	0b	ff	ff	ff	H
FETCH	06	92	09	65	ff	0b	ff	ff	ff	H
EXECUTE	06	92	09	65	ff	0b	ff	ff	ff	He
FETCH	07	22	09	65	ff	0b	ff	ff	ff	He
EXECUTE	07	22	09	65	ff	0c	ff	ff	ff	He
FETCH	08	41	09	65	ff	0c	ff	ff	ff	He
FETCH	09	41	02	65	ff	0c	ff	ff	ff	He
EXECUTE	02	41	02	65	ff	0c	ff	ff	ff	He
FETCH	03	28	02	65	ff	0c	ff	ff	ff	He
EXECUTE	03	28	02	6c	ff	0c	ff	ff	ff	He
FETCH	04	45	02	6c	ff	0c	ff	ff	ff	He
FETCH	05	45	09	6c	ff	0c	ff	ff	ff	He
EXECUTE	05	45	09	6c	ff	0c	ff	ff	ff	He
FETCH	06	92	09	6c	ff	0c	ff	ff	ff	He
EXECUTE	06	92	09	6c	ff	0c	ff	ff	ff	Hel

Not all lines are printed here because of space limitations.

FETCH	06	92	09	21	ff	0f	ff	ff	ff	Hello
EXECUTE	06	92	09	21	ff	0f	ff	ff	ff	Hello!
FETCH	07	22	09	21	ff	0f	ff	ff	ff	Hello!
EXECUTE	07	22	09	21	ff	10	ff	ff	ff	Hello!
FETCH	08	41	09	21	ff	10	ff	ff	ff	Hello!
FETCH	09	41	02	21	ff	10	ff	ff	ff	Hello!
EXECUTE	02	41	02	21	ff	10	ff	ff	ff	Hello!
FETCH	03	28	02	21	ff	10	ff	ff	ff	Hello!
EXECUTE	03	28	02	00	ff	10	ff	ff	ff	Hello!
FETCH	04	45	02	00	ff	10	ff	ff	ff	Hello!
FETCH	05	45	09	00	ff	10	ff	ff	ff	Hello!
EXECUTE	09	45	09	00	ff	10	ff	ff	ff	Hello!
FETCH	0a	b2	09	00	ff	10	ff	ff	ff	Hello!
STOP	0a	b2	09	00	ff	10	ff	ff	ff	Hello!

Figure 4-8b. Step-by-step execution of the program in Figure 4-8a.

- Instruction 45H, "jump if register A zero", is a so-called conditional jump. The execution of the program jumps to address 09H if the content of register A is zero. If the content of register A is not zero, the program execution continues from the instruction that follows "jump if register A zero". Instruction "jump if register A zero" causes a jump in the program when all characters of text "Hello!" have been displayed on the screen. Because there is a zero at the end of the text, register A becomes zero and a jump to address 09H takes place. Because address 09H contains the instruction "stop processing", the program terminates.
- Instruction 41H, "jump to address", is an unconditional jump which causes a jump to address 02H in the program. With this instruction, the program prepares to read and display the next character of text "Hello!".

The execution of the program in Figure 4-8a is described in Figure 4-8b. Again, it is very important that you study carefully how the values in the registers of the imaginary computer change while the program is being executed. While you study Figure 4-8b, you should note the following points:

- During the FETCH operations, when the processor is reading instructions and their operands from the main memory, the values of registers A (RA), B (RB), MEMORY POINTER (MP), and STACK POINTER (SP) do not change, but the value of register PROGRAM POINTER (PP) grows after every FETCH operation.
- Values of registers INSTRUCTION CODE (IC) and INSTRUCTION OPERAND (IO) do not change during EXECUTE operations.
- When the processor reads an instruction which has an odd instruction code, it performs another FETCH operation that reads an operand for the instruction. After the second FETCH operation, register INSTRUCTION OPERAND (IO) has a new value. The imaginary computer is designed in such a way that two-byte instructions have odd instruction codes and single-byte instructions have even instruction codes.

Exercises related to machine-level programming

Exercise 4-2. Study the program in Figure 4-7a. Which instruction can be taken away from the program without affecting the output of the program? The program should still print "Hello!" if the instruction were removed from the program.

Exercise 4-3. Modify one (and only one) byte in the program of Figure 4-8a so that the program prints "Hello" instead of "Hello!".

Exercise 4-4. Modify one (and only one) byte in the program of Figure 4-8a so that the program prints "ello!" instead of "Hello!".

4.7 Programming language IML and compilation

Now we have seen that executable computer programs are sequences of numerical machine instructions. By putting the right machine instructions into a program, it is possible to make a computer behave in a certain manner. But it is quite difficult to construct a program if you only have the numerical machine instructions in your mind. It is also hard to read the computer programs which are made of numerical machine instructions. For these reasons, different kinds of textual programming languages have been invented. Textual programming languages allow us to write computer programs in text form. Textual programming languages are defined in such a way that the numerical machine codes can be generated automatically in a process that is called compilation.

Here we will define and study a simple textual programming language for the imaginary computer. The name of the simple language is IML, an abbreviation of "Imaginary Computer's Machine-Level Language". IML is a machine-level language because you still have to think about the registers of the imaginary processor when you are writing programs with it. But when you use IML, you do not have to remember the numerical machine instructions, and the programs become readable. IML is presented here just to show you some principles of programming languages. IML helps you also to understand what happens in the compilation of computer programs. The purpose is not that you will become an expert in IML programming. You may forget most of IML after you have read this chapter and understood how the programs work. Languages like IML should not be used if there are languages like C# available. When you write programs with a high-level programming language like C#, programming is easier, because you do not have to think about the registers of the processor that is being used.

The following is the program of Figure 4-7a written with the IML programming language:

```
//  hello.iml  (c) 1999-2001 Kari Laitinen

//  A program that prints the text "Hello!" on the screen.

      load_register_a_with_value    'H'
      output_byte_from_register_a
      load_register_a_with_value    'e'
      output_byte_from_register_a
      load_register_a_with_value    'l'
      output_byte_from_register_a
      load_register_a_with_value    'l'
      output_byte_from_register_a
      load_register_a_with_value    'o'
      output_byte_from_register_a
      load_register_a_with_value    '!'
      output_byte_from_register_a
      stop_processing
```

This program is largely made of the explanations which are present in Figure 4-7a. The numerical instruction codes of the imaginary processor are represented by textual instructions in an IML program. As IML is a programming language, it is possible to make a compiler which can translate an IML program from the textual form to numerical form. An IML compiler would process the above program according to the following rules:

- A program is translated line by line, from the beginning to the end.
- All empty lines, lines that contain no text, are omitted.
- All lines starting with character pair // are omitted. Lines starting with // are comment lines which do not belong to the actual program.

- All other lines are translated to machine instructions:

```
load_register_a_with_value    'H'   translates to  15H  48H
load_register_a_with_value    'e'   translates to  15H  65H
load_register_a_with_value    'l'   translates to  15H  6CH
      etc.
output_byte_from_register_a         translates to  92H
stop_processing                     translates to  B2H
```

Textual IML instructions like

```
load_register_a_with_value
add_register_b_to_a
```

correspond to the numerical instruction codes of the imaginary processor. You can think that the textual IML instructions are names invented for the numerical instructions. The textual instructions are formed of the phrases which can be found in the second column of Table 4-1. The words of the phrases must be concatenated with underscores so that an IML compiler can interpret them as whole textual entities. An IML compiler translates each textual instruction to a numerical instruction code, and, if the instruction uses an operand, adds an operand byte after the instruction code. Table 4-2 lists all textual instructions together with the corresponding numerical instruction codes. If an instruction needs a value or a memory address as an operand, there is VVH or MMH on the instruction's line in Table 4-2. The table also shows which registers of the imaginary processor are modified when the instructions are executed. To compile an IML source program, an IML compiler must have an internal instruction translation table that resembles Table 4-2 and describes which textual instruction corresponds to which numerical instruction.

Because many of the instructions of the imaginary processor need an operand byte, there have to be special notations in the IML programming language to present the operand bytes. When numerical constants are needed as operands in IML programs, they can be expressed in the following ways:

'H'	means 48H, the character code of uppercase letter H
'a'	means 61H, the character code of lowercase letter a
' '	means 20H, the character code of the space character
'\n'	means 0AH, the character code of the newline character
'n'	means 6EH, the character code of lowercase letter n
123	means 7BH, the decimal number 123
0x22	means 22H, the decimal number 34
0x1F	means 1FH, the decimal number 31

So if you need a character code of a letter in an IML program, you do not have to remember the character code. You can write the letter inside single quote characters, and let the IML compiler translate it to the correct character code. If you need a number in your program, you can write it as a normal decimal number. In the case that you want to write a number in hexadecimal form you must put the prefix 0x before the hexadecimal digits. Because there are several ways to express operand values in an IML program, there are many possibilities to write IML program lines. For example, as the character code of letter A is 65 as a decimal number and 41H as a hexadecimal number, the following three IML program lines mean the same

```
load_register_a_with_value      'A'
load_register_a_with_value      65
load_register_a_with_value      0x41
```

Table 4-2: IML instruction translation table.

TEXTUAL INSTRUCTION	CODE	OPERAND	REGISTERS WRITTEN					
			RA	RB	PP	MP	SP	Memory
add_value_to_register_a	11H	VVH	x					
add_register_b_to_a	12H		x					
subtract_value_from_register_a	13H	VVH	x					
subtract_register_b_from_a	14H		x					
load_register_a_with_value	15H	VVH	x					
increment_register_a	16H		x					
load_register_b_with_value	17H	VVH		x				
increment_register_b	18H			x				
decrement_register_a	1AH		x					
decrement_register_b	1CH			x				
move_content_of_register_a_to_b	1EH			x				
set_memory_pointer	21H	MMH				x		
increment_memory_pointer	22H					x		
decrement_memory_pointer	24H					x		
store_register_a_to_memory	26H							x
load_register_a_from_memory	28H		x					
store_register_b_to_memory	2AH							x
load_register_b_from_memory	2CH			x				
jump_to_address	41H	MMH			x			
jump_if_registers_equal	43H	MMH			?[a]			
jump_if_register_a_zero	45H	MMH			?			
jump_if_register_a_smaller_than_b	47H	MMH			?			
jump_if_register_a_greater_than_b	49H	MMH			?			
jump_if_input_not_ready	4BH	MMH			?			
call_subroutine	81H	MMH			x		x	x
return_to_calling_program	82H				x		x	
output_byte_from_register_a	92H							
output_byte_from_register_b	94H							
input_byte_to_register_a	96H		x					
set_stack_pointer	A1H	MMH					x	
push_register_a_to_stack	A2H						x	x
pop_register_a_from_stack	A4H		x				x	
stop_processing	B2H							

a. The question mark ? means that the register is possibly written. The conditional jump instructions write register PROGRAM POINTER only when certain conditions exist.

Most instructions for the imaginary processor are plain instructions which do not need any operands, some instructions take numerical values as operands, and some instructions must be given memory addresses as operands. The memory addresses are numerical values, but they are different kinds of numerical values than, for example, character codes. In the program of Figure 4-8a, memory addresses are needed by instructions which cause jumps in the program, and the memory address of the text "Hello!" needs to be stored to MEMORY POINTER at the beginning of the program.

In the IML programming language, memory addresses are described with address names which have to be invented by the program author. When memory addresses are correctly described with names, an IML compiler can find correct numerical values for the memory addresses. This helps a lot when programs must be created for the imaginary computer.

Figure 4-8c shows what the program of Figure 4-8a looks like when it is written with the IML programming language, and translated to machine instructions with an IML compiler. There are four address names in the program. The compiler has found numerical values for the address names in the following way:

```
ADDRESS NAME                NUMERICAL ADDRESS

beginning_of_program        00H
display_characters          02H
end_of_program              09H
address_of_text             0AH
```

The address names refer to memory locations in the program. For example, the name **address_of_text** refers to the memory address of the first letter of the text "Hello!", the name **display_characters** refers to the address where instruction **load_register_a_from_memory** is, and the name **end_of_program** refers to the address where the instruction **stop_processing** is located. An address name must be written before the instruction whose address is referred to by the name. The colon character : must be written after an address name in that place where it specifies a memory location.

When an IML compiler processes an IML source program, it first finds all address names which are followed by the colon character (;). Then the compiler counts how many instructions and what kinds of instructions precede each address name in the program. That way the compiler is able to find numerical values for the address names. In the program in Figure 4-8c, address name **display_characters** is given the value 02H because it is preceded by a single two-byte instruction in the program. Address name **end_of_program** is given the value 09H because its position in the program is such that it is preceded by three single-byte instructions and three two-byte instructions which occupy addresses 00H, 01H, ..., and 08H.

After the IML compiler has found numerical values for the address names, it is able to translate those instructions which use an address name as an operand. For example, when the compiler has found out that the name **end_of_program** refers to address 09H in the program of Figure 4-8c, it is able to make the following translation

```
jump_if_register_a_zero   end_of_program     -->   45H  09H
```

The compiler has an internal instruction translation table from which it can find out that the textual instruction **jump_if_register_a_zero** must be translated with the numerical code 45H, and the address value 09H it has calculated by itself.

As IML is a programming language for which it is possible to build a compiler, it is possible to write a program which does not compile because some rules of the IML language are violated in the program. For example, if the above instruction were written like

```
jump_if_register_aaa_zero   end_of_program
```

an IML compiler could not compile the instruction because it could not find the textual instruction **jump_if_register_aaa_zero** in its internal instruction translation table. Similarly, the instruction

```
jump_if_register_a_zero    end_of_programmm
```

would result in a compilation error because the compiler could not find the address name **end_of_programmm** in its internal table of address names. It is important that a name in a computer program is always written in exactly the same way. This is equally important both in IML programs and in C# programs.

The text "Hello!" is defined

```
address_of_text:        STRING       "Hello!"
```

in the program in Figure 4-8c. When an IML compiler recognizes this definition, it puts the character codes of characters H, e, l, l, o, and ! to those memory locations where the declaration is. In addition to the character codes of the characters, the compiler inserts 00H, a zero, after the last character. Thus the above line is translated to codes 48H, 65H, 6CH, 6CH, 6FH, 21H, and 00H in memory addresses from 0AH to 10H in Figure 4-8c.

```
//  hello_loop.iml  (c) 1997 - 2000 Kari Laitinen

//  This program prints the characters of text "Hello!"
//  in a loop. The text is defined with the keyword STRING
//  at the end of the program. As the IML compiler puts a
//  zero at the end of the string, the program knows when
//  to stop printing characters.

beginning_of_program:
      set_memory_pointer             address_of_text

display_characters:
      load_register_a_from_memory
      jump_if_register_a_zero        end_of_program
      output_byte_from_register_a
      increment_memory_pointer
      jump_to_address                display_characters

end_of_program:
      stop_processing

address_of_text:        STRING       "Hello!"
```

```
00    21  0A

02    28
03    45  09
05    92
06    22
07    41  02

09    B2

0A    48
0B    65
0C    6C
0D    6C
0E    6F
0F    21
10    00
```

This is the compiled version of the program. All numbers are hexadecimal without the letter H at the end. Memory addresses are shown on the left and generated numerical codes are on the right.

Figure 4-8c. The program of Figure 4-8a written with IML and translated to machine instructions.

The word STRING is a reserved keyword in the IML programming language. This means that the word STRING cannot be used as an address name in an IML program. STRING is reserved for situations when textual strings need to be defined (or declared). The characters of the textual string must be given inside double quote characters "". An IML compiler automatically converts the characters of the textual string to character codes and adds a NULL character, a zero, to the end of the visible characters. When there is a NULL character, 00H, at the end of the visible characters, a program can easily recognize where the text ends, and, for example, stop printing the text. Because a textual string can be defined this way in an IML program, it is very easy to make the program of Figure 4-8c print other texts. For example, if the last line of the textual program were changed to

```
address_of_text:        STRING      "How are you doing?"
```

the program would print

```
How are you doing?
```

if it were compiled with an IML compiler and executed in the imaginary computer.

In addition to word STRING there are two other similar reserved keywords in the IML language. These words are DATA and CONSTANT. If you write in an IML program

```
result_of_calculation:   DATA       2
```

you reserve two bytes from the main memory and the address name **result_of_calculation** refers to the address of the first reserved byte. By using a larger number in place of 2, it is possible to reserve larger areas from the main memory.

With the reserved keyword CONSTANT it is possible to reserve a single byte from the main memory, and initialize the reserved memory location with a certain numerical value. For example, the definition

```
number_of_states_in_usa:   CONSTANT   50
```

would reserve one byte of memory and store the value 50 in that memory location. By using the address name **number_of_states_in_usa**, it would be possible to refer to the reserved memory location. The definition

```
year_of_french_revolution:  CONSTANT  1789
```

would be incorrect in the case of IML and the imaginary computer because numbers larger than 255 cannot be stored in a single byte. It is true, though, that a historical revolution started in France in 1789.

In the example definitions above, all reserved memory locations are given an address name with which they can be referred to. It is common in computer programming that the program writer must invent and write various names in programs. The program writer must follow the rules of the programming language when he or she invents the names. In IML, the names must consist of letters and underscore characters. There may not be any space characters in a name, and a name may not be STRING, DATA, CONSTANT, or any of the textual instructions in the first column of Table 4-2.

How do real computers differ from the imaginary computer?

The imaginary computer is an 8-bit computer with only 256 bytes of main memory. Although this kind of a computer operates, and can be programmed, like a real computer, computers like the imaginary computer are not used in the real world. There can be 8-bit processors in use in some special applications, but no modern computer works with a small 256-byte main memory. If we, for example, compare the imaginary computer to a personal computer, we can find the following essential differences:

- The processors which are used in personal computers are usually 32-bit processors in which all internal registers can hold 32-bit values. The registers of the imaginary processor are all 8-bit registers.
- The processors of personal computers have more machine instructions than our imaginary processor. The machine instructions are more complicated, they can perform a wide range of operations, and they can operate with many general-purpose registers. The imaginary processor has only a simple set of machine instructions with which some basic computing activities can be performed.
- The main memory of a personal computer can be a million times larger than the main memory of the imaginary computer. Personal computers use auxiliary memory devices in addition to the main memory. The imaginary computer works only with its small main memory.
- Various peripheral equipment such as modems, printers, scanners, cameras, etc. can be connected to personal computers. The imaginary computer has only two ports to which a screen and a keyboard are connected.
- In a personal computer there is always an operating system, a program which can control the execution of other programs. In the imaginary computer programs are loaded to the main memory with some imagination, and they are executed by switching the computer on.

Compiler for IML?

You may be wondering why I do not provide a compiler for the IML language. The truth is that I do have a rudimentary compiler for the language, but I'm afraid that it does not work well enough to be distributed for wider use. Compilers are extremely complicated computer programs, and it is not easy to make a reliable compiler even for such a simple language like IML. The existing compiler inputs an IML source program file and produces another file which contains the machine instructions of the program. For example, when the file **hello_loop.iml** is compiled, a file named **hello_loop.ice** is produced. Files ending with **.ice** can be loaded to the ICOM simulator. (Simulators for the imaginary computer will be discussed in the following section.) The file name extension **.ice** means "imaginary computer executable". There are some **.ice** files available among the electronic material of this book, but the same programs can be found built in the simulators.

Another reason for not providing an IML compiler is that IML is just a language for educational purposes. It is not necessary to write more IML programs than those that are presented in this book and built in the IC8 and ICOM simulators. You should start learning C# and compiling C# programs as soon as you have learned the basics of computer technology in this chapter.

4.8 IC8 and ICOM – simulator programs for the imaginary computer

Although the imaginary computer described in this chapter does not exist as a computer built of hardware components, there are two programs which simulate the imaginary computer on a personal computer. The older one of the simulator programs is called ICOM, and you can find information about it if you read the sample pages of Chapter 4 of my C++ book. You can find those pages via the Internet address www.naturalprogramming.com. The ICOM simulator runs in a command prompt window, and you can command it only with the keyboard of your computer. It does not recognize mouse commands. If you decide to use the ICOM simulator, please read the help pages that are built in it.

A better simulation program[1] for the imaginary computer is called IC8. This program is a so-called Java applet that can be executed on an Internet page. To use the IC8 simulator, you must go with your Internet browser (e.g. Internet Explorer) to a certain page on the Internet. That page can be reached through the address www.naturalprogramming.com. In order to run the simulator, the settings of your Internet browser must be such that Java applets can be executed. (I'll put some information about the browser settings on the Internet pages.)

When you have found the page on which the IC8 simulator is located, and the execution of Java applets is enabled in your browser, the browser should display a view like the one that is shown in Figure 4-9. The IC8 simulator is a program which you can control by using the mouse to press its buttons. The most important rule in the use of the simulator is that you must first load a program into its main memory with the **Load Program** button, and only after that can you actually start using the simulator. Some of the buttons of the simulator are explained in Figure 4-9. The rest of the buttons have the following effects

- The **Reset** button loads value 00H to register PROGRAM POINTER and value FFH to all other important processor registers. The INPUT READY signal is set to 0. The screen is cleared, but the contents of the main memory are not modified.
- By pressing the **Translate** button, you can see a kind of textual form of the program in the main memory. Repressing the button brings back the normal view.
- With the **Modify** button you can put the simulator to a mode in which you can modify the contents of the main memory. After the **Modify** button has been pressed, you can click on memory locations with the mouse and enter new values to the selected locations through the keyboard. You must repress the **Modify** button to get back to the normal simulation mode.
- By selecting one of the buttons **HEX**, **BIN**, or **DEC**, you can choose the numbering system in which numbers are shown.

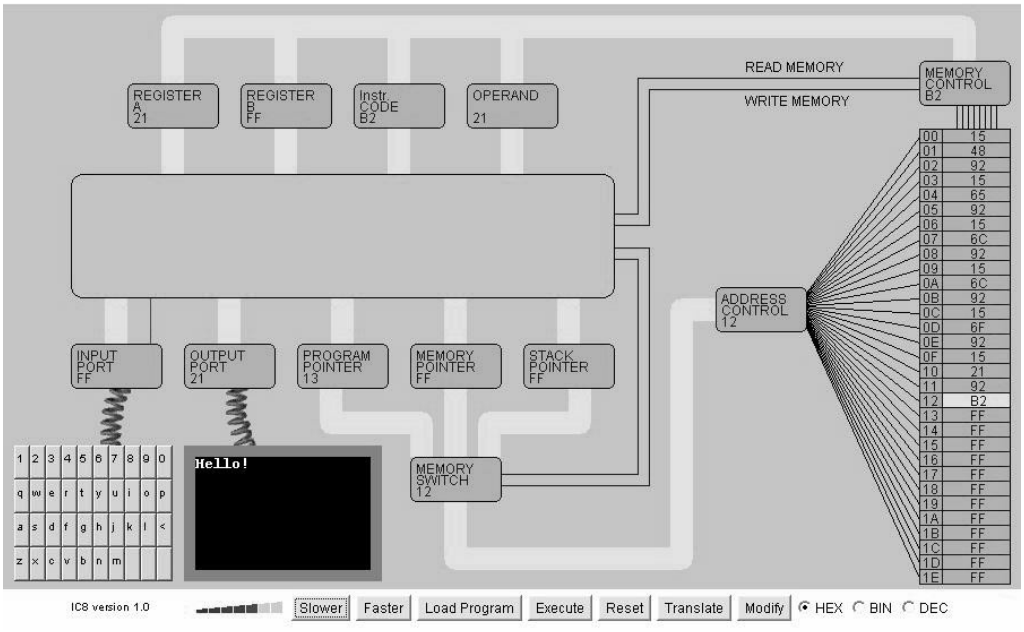
While you are using the IC8 simulator program, you should remember that it is a program that has not been widely tested, and there may be errors in it. Although I do not know about any serious errors in IC8, I cannot guarantee that it operates correctly. And most importantly, I shall not assume any kind of responsibility for any kind of damages the IC8 simulator (or the ICOM simulator) causes while you are using it. Because IC8 is an applet, not a real application, it should not cause any serious troubles on your computer. If it seems that the program is not operating properly, a wise thing to do is to first close your browser, restart the browser, and then go back to the page of the simulator.

Using the IC8 simulator, or the ICOM simulator, helps you to understand how computers operate. Of course, if you feel that you understand the example IML programs of this chapter by just studying them on paper, it is not necessary to use any simulator programs. The programs are in any case an easy way to verify the answers which you got for the exercises in this chapter.

---

1. The word "simulation" is common in the world of computers. It means that a computer program imitates a real-life phenomenon or something that could exist in reality.

Here, the simulator shows the first 31 locations of the main memory. The machine instructions of program **hello.iml** are loaded into the memory. An important feature of this simulator is that if you click with the mouse on an instruction in a memory location, when the simulator is executing a program, the execution goes with maximum speed until the clicked instruction, and continues then with the normal, selected speed.



By pressing the **Load Program** button, you can load an executable program into the main memory of the imaginary computer. There are many compiled ready-to-run executable programs built in the simulation program. All the example programs of this chapter are there for you to try. After this button is pressed, you'll see a menu from which you can select a program when you know the file names of the IML programs. For example, if you want to run the program of Figure 4-8c in the simulator, you can easily load the correct program because the file name **hello_loop.iml** is mentioned in Figure 4-8c.

After a program has been loaded with the **Load Program** button, it can be executed by pressing the **Execute** button. The **Execute** button transforms to **Pause** button with which you can stop the execution temporarily. When the program is stopped, this button is the **Continue** button with which you can resume program execution. The execution speed of the simulator can be altered with the **Slower** and **Faster** buttons.

Figure 4-9. The screen of the IC8 simulator after the program hello.iml has been executed.

Exercise with the IC8 simulator program

Exercise 4-5. Verify your answers to exercises 4-3 and 4-4 by using the IC8 simulator program. Use the Load Program button to load program **hello_loop.iml** to the simulator, and then modify the program by pressing the Modify button. Finally you must execute it to see how it works.

4.9 A program that contains a loop

In principle, programs are executed so that those instructions which are at the beginning of a program are executed first, and the program execution proceeds instruction by instruction towards the end of the program. By using jump instructions it is possible to violate this principle. A jump instruction can cause a jump from the end of a program to the beginning of the program. We say that jump instructions can create a loop in a program. Loops are sequences of instructions which are repeated many times during the execution of a program. In program **hello_loop.iml** in Figure 4-8c we have already seen a loop which prints the characters of the text "Hello!". Another program in which a loop is used to print characters is **abcde.iml** in Figure 4-10a.

Only the textual IML instructions are shown for the program in Figure 4-10a. For you to learn how the program operates, it might be useful to write the numerical instructions by hand besides the textual instructions. You could thus compile the program manually. Figure 4-10b shows what happens on the screen and in the registers of the imaginary processor when the program is being executed. Figure 4-10b can help you to hand-compile the program. You can also exploit tables 4-1 and 4-2.

A key idea in program **abcde.iml** is that it prints letters A, B, C, D, and E because the character codes of these letters are sequential numbers 41H, 42H, 43H, 44H, and 45H. First the program loads the character code of letter A, 41H, to register B. When register B is incremented inside the loop of the program, it first increments to 42H, then to 43H, then to 44H, then to 45H, and finally to 46H. Value 46H is never printed. The other general purpose register of the processor, register A, is used to count how many letters are left to be printed to the screen. Register A is 5 at the beginning. As it is decremented inside the loop, it first decrements to 4, then to 3, then to 2, then to 1, and finally to 0. When register A reaches value 0, a jump to address name **`end_of_program`** takes place, and the program terminates.

```
//  abcde.iml   (c) 1998-2000 Kari Laitinen

//  This program prints the letters ABCDE to the screen.
//  Register A is used to count how many characters have
//  been printed. The character code being output to the
//  screen is held in register B. The first code is 'A'
//  which means 41H, the character code of letter A.
//  As the character code in register B is incremented
//  after each printing, a different letter will be
//  printed each time.

beginning_of_program:
      load_register_a_with_value       5
      load_register_b_with_value       'A'

print_next_letter:
      output_byte_from_register_b
      increment_register_b
      decrement_register_a
      jump_if_register_a_zero          end_of_program
      jump_to_address                  print_next_letter

end_of_program:
      stop_processing
```

Two jump instructions are usually needed to construct a loop. Here the first jump instruction terminates the loop at the right moment. The second jump instruction continues the loop.

Figure 4-10a. A program that prints letters ABCDE in a loop.

Here the first jump in the program takes place. Instruction 41H, "jump to address", is executed. The execution modifies PROGRAM POINTER (PP) so that the address of the next instruction is 04H.

Register A (RA) reaches value 0 at the end. At that moment register B has already value 46H, the character code of letter F. That letter is, though, never printed because the program terminates when register A has value 0.

Status	PP	IC	IO	RA	RB	MP	SP	FE	FF	Screen contents
RESET	00	ff	ff	ff	ff	ff	ff	ff	ff	
FETCH	01	15	ff	ff	ff	ff	ff	ff	ff	
FETCH	02	15	05	ff	ff	ff	ff	ff	ff	
EXECUTE	02	15	05	05	ff	ff	ff	ff	ff	
FETCH	03	17	05	05	ff	ff	ff	ff	ff	
FETCH	04	17	41	05	ff	ff	ff	ff	ff	
EXECUTE	04	17	41	05	41	ff	ff	ff	ff	
FETCH	05	94	41	05	41	ff	ff	ff	ff	
EXECUTE	05	94	41	05	41	ff	ff	ff	ff	A
FETCH	06	18	41	05	41	ff	ff	ff	ff	A
EXECUTE	06	18	41	05	42	ff	ff	ff	ff	A
FETCH	07	1a	41	05	42	ff	ff	ff	ff	A
EXECUTE	07	1a	41	04	42	ff	ff	ff	ff	A
FETCH	08	45	41	04	42	ff	ff	ff	ff	A
FETCH	09	45	0b	04	42	ff	ff	ff	ff	A
EXECUTE	09	45	0b	04	42	ff	ff	ff	ff	A
FETCH	0a	41	0b	04	42	ff	ff	ff	ff	A
FETCH	0b	41	04	04	42	ff	ff	ff	ff	A
EXECUTE	04	41	04	04	42	ff	ff	ff	ff	A
FETCH	05	94	04	04	42	ff	ff	ff	ff	A
EXECUTE	05	94	04	04	42	ff	ff	ff	ff	AB
FETCH	06	18	04	04	42	ff	ff	ff	ff	AB
EXECUTE	06	18	04	04	43	ff	ff	ff	ff	AB
FETCH	07	1a	04	04	43	ff	ff	ff	ff	AB
EXECUTE	07	1a	04	03	43	ff	ff	ff	ff	AB
FETCH	08	45	04	03	43	ff	ff	ff	ff	AB
FETCH	09	45	0b	03	43	ff	ff	ff	ff	AB
EXECUTE	09	45	0b	03	43	ff	ff	ff	ff	AB
FETCH	0a	41	0b	03	43	ff	ff	ff	ff	AB
FETCH	0b	41	04	03	43	ff	ff	ff	ff	AB
EXECUTE	04	41	04	03	43	ff	ff	ff	ff	AB
FETCH	05	94	04	03	43	ff	ff	ff	ff	AB
EXECUTE	05	94	04	03	43	ff	ff	ff	ff	ABC
FETCH	06	18	04	03	43	ff	ff	ff	ff	ABC
EXECUTE	06	18	04	03	44	ff	ff	ff	ff	ABC
FETCH	07	1a	04	03	44	ff	ff	ff	ff	ABC

Because of space limitation, 16 lines have been left out here.

FETCH	0a	41	0b	01	45	ff	ff	ff	ff	ABCD
FETCH	0b	41	04	01	45	ff	ff	ff	ff	ABCD
EXECUTE	04	41	04	01	45	ff	ff	ff	ff	ABCD
FETCH	05	94	04	01	45	ff	ff	ff	ff	ABCD
EXECUTE	05	94	04	01	45	ff	ff	ff	ff	ABCDE
FETCH	06	18	04	01	45	ff	ff	ff	ff	ABCDE
EXECUTE	06	18	04	01	46	ff	ff	ff	ff	ABCDE
FETCH	07	1a	04	01	46	ff	ff	ff	ff	ABCDE
EXECUTE	07	1a	04	00	46	ff	ff	ff	ff	ABCDE
FETCH	08	45	04	00	46	ff	ff	ff	ff	ABCDE
FETCH	09	45	0b	00	46	ff	ff	ff	ff	ABCDE
EXECUTE	0b	45	0b	00	46	ff	ff	ff	ff	ABCDE
FETCH	0c	b2	0b	00	46	ff	ff	ff	ff	ABCDE
STOP	0c	b2	0b	00	46	ff	ff	ff	ff	ABCDE

Figure 4-10b. Step-by-step execution of the program in Figure 4-10a.

4.10 Subroutine calls and stack operations

It is common that larger computer programs are organized so that they consist of smaller pieces of programs. The smaller pieces are usually subroutines. A subroutine typically performs a certain well-defined operation in a program. Such an operation can be, for example, printing a line of text, or reading a character from the keyboard. In every computer program there must be a main program. In addition to the main program, there can be subroutines which are called by the main program. The term "calling" is used to describe the relation between a main program and a subroutine. A main program calls a subroutine to perform a certain activity. When the subroutine has performed its activity, there is a return to the main program, the calling program.

The programs which we have studied so far are just main programs which do not call any subroutines. Program **aaaabbbbcccc.iml** in Figure 4-11a is an example where a subroutine is called. Figure 4-11b shows what happens in the execution of the program. The first part of **aaaabbbbcccc.iml** is the main program. The last part of the program is subroutine named **`output_register_a_four_times`**. As its address name suggests, the subroutine outputs the content of register A four times. As the subroutine is called three times with register A contents 'a', 'b', and 'c', the text aaaabbbbcccc emerges on the screen when the program is executed. (It may again be a good idea to hand-compile the program by writing numerical instruction codes and their addresses besides the textual instruction codes.)

STACK POINTER is an important processor register in subroutine calls. It is one of the three registers with which the main memory can be accessed. The value of STACK POINTER is FFH when the imaginary computer starts operating. At the beginning, the STACK POINTER thus points to the last memory location in the main memory.

```
//  aaaabbbbcccc.iml

//  This program prints the text "aaaabbbbcccc" to the
//  screen. It calls a subroutine which prints the contents
//  of register A four times to the screen.

//  First register A is loaded with 'a', the character code
//  of letter a (61H). When register A is incremented, its
//  content becomes first 61H and later 62H. These are the
//  character codes of letters b and c.

beginning_of_program:
      load_register_a_with_value    'a'
      call_subroutine               output_register_a_four_times
      increment_register_a
      call_subroutine               output_register_a_four_times
      increment_register_a
      call_subroutine               output_register_a_four_times
      stop_processing

output_register_a_four_times:
      output_byte_from_register_a
      output_byte_from_register_a
      output_byte_from_register_a
      output_byte_from_register_a
      return_to_calling_program
```

The start address of the subroutine is the operand of instruction **`call_subroutine`**. The address name that is written here is the name of the subroutine.

Figure 4-11a. A program in which subroutine output_register_a_four_times is called.

Here the first subroutine call, instruction 81H, is executed. PROGRAM POINTER gets the value 0BH which is the beginning of the subroutine. The return address 04H is stored onto the stack in memory location FFH. STACK POINTER is decremented to value FEH.

When the subroutine is called for the second time, the return address stored on the stack is 07H. The second instruction to increment register A is in that address.

Here instruction 82H is executed, and a return to calling program takes place. 0AH is the return address in the calling program which is copied from the stack to PROGRAM POINTER (PP). The value of STACK POINTER is incremented to FFH.

Status	PP	IC	IO	RA	RB	MP	SP	FE	FF	Screen contents
RESET	00	ff	ff	ff	ff	ff	ff	ff	ff	
FETCH	01	15	ff	ff	ff	ff	ff	ff	ff	
FETCH	02	15	61	ff	ff	ff	ff	ff	ff	
EXECUTE	02	15	61	61	ff	ff	ff	ff	ff	
FETCH	03	81	61	61	ff	ff	ff	ff	ff	
FETCH	04	81	0b	61	ff	ff	ff	ff	ff	
EXECUTE	0b	81	0b	61	ff	ff	fe	ff	04	
FETCH	0c	92	0b	61	ff	ff	fe	ff	04	
EXECUTE	0c	92	0b	61	ff	ff	fe	ff	04	a
FETCH	0d	92	0b	61	ff	ff	fe	ff	04	a
EXECUTE	0d	92	0b	61	ff	ff	fe	ff	04	aa
FETCH	0e	92	0b	61	ff	ff	fe	ff	04	aa
EXECUTE	0e	92	0b	61	ff	ff	fe	ff	04	aaa
FETCH	0f	92	0b	61	ff	ff	fe	ff	04	aaa
EXECUTE	0f	92	0b	61	ff	ff	fe	ff	04	aaaa
FETCH	10	82	0b	61	ff	ff	fe	ff	04	aaaa
EXECUTE	04	82	0b	61	ff	ff	ff	ff	04	aaaa
FETCH	05	16	0b	61	ff	ff	ff	ff	04	aaaa
EXECUTE	05	16	0b	62	ff	ff	ff	ff	04	aaaa
FETCH	06	81	0b	62	ff	ff	ff	ff	04	aaaa
FETCH	07	81	0b	62	ff	ff	ff	ff	04	aaaa
EXECUTE	0b	81	0b	62	ff	ff	fe	ff	07	aaaa
FETCH	0c	92	0b	62	ff	ff	fe	ff	07	aaaa
EXECUTE	0c	92	0b	62	ff	ff	fe	ff	07	aaaab
FETCH	0d	92	0b	62	ff	ff	fe	ff	07	aaaab
EXECUTE	0d	92	0b	62	ff	ff	fe	ff	07	aaaabb
FETCH	0e	92	0b	62	ff	ff	fe	ff	07	aaaabb
EXECUTE	0e	92	0b	62	ff	ff	fe	ff	07	aaaabbb
FETCH	0f	92	0b	62	ff	ff	fe	ff	07	aaaabbb
EXECUTE	0f	92	0b	62	ff	ff	fe	ff	07	aaaabbbb
FETCH	10	82	0b	62	ff	ff	fe	ff	07	aaaabbbb
EXECUTE	07	82	0b	62	ff	ff	ff	ff	07	aaaabbbb
FETCH	08	16	0b	62	ff	ff	ff	ff	07	aaaabbbb
EXECUTE	08	16	0b	63	ff	ff	ff	ff	07	aaaabbbb
FETCH	09	81	0b	63	ff	ff	ff	ff	07	aaaabbbb
FETCH	0a	81	0b	63	ff	ff	ff	ff	07	aaaabbbb
EXECUTE	0b	81	0b	63	ff	ff	fe	ff	0a	aaaabbbb
FETCH	0c	92	0b	63	ff	ff	fe	ff	0a	aaaabbbb
EXECUTE	0c	92	0b	63	ff	ff	fe	ff	0a	aaaabbbbc
FETCH	0d	92	0b	63	ff	ff	fe	ff	0a	aaaabbbbc
EXECUTE	0d	92	0b	63	ff	ff	fe	ff	0a	aaaabbbbcc
FETCH	0e	92	0b	63	ff	ff	fe	ff	0a	aaaabbbbcc
EXECUTE	0e	92	0b	63	ff	ff	fe	ff	0a	aaaabbbbccc
FETCH	0f	92	0b	63	ff	ff	fe	ff	0a	aaaabbbbccc
EXECUTE	0f	92	0b	63	ff	ff	fe	ff	0a	aaaabbbbcccc
FETCH	10	82	0b	63	ff	ff	fe	ff	0a	aaaabbbbcccc
EXECUTE	0a	82	0b	63	ff	ff	ff	ff	0a	aaaabbbbcccc
FETCH	0b	b2	0b	63	ff	ff	ff	ff	0a	aaaabbbbcccc
STOP	0b	b2	0b	63	ff	ff	ff	ff	0a	aaaabbbbcccc

Figure 4-11b. Step-by-step execution of the program in Figure 4-11a.

In general, a stack is a memory area that is used so that what is last put to the stack, comes out first from the stack. When a value is put to the stack, we say that a push operation is made. The opposite operation is a pop operation that takes a value away from the stack. The imaginary processor uses the last part of the main memory as its stack. The STACK POINTER register always points to the first free memory address of the stack. When values are pushed to the stack, the value of STACK POINTER is decremented. In pop operations the value of STACK POINTER is incremented. We say that a pop operation takes a value away from the stack although in reality the value remains in a memory location in the main memory. When the pop operation increments the STACK POINTER, it frees a memory location for a push operation.

The stack is always used when subroutine calls are made. After a subroutine has been called and executed, the execution of the calling program must continue from the next statement after the subroutine call. This can be accomplished by storing a return address into the stack. Instruction **`call_subroutine`** automatically pushes the return address, the address of the next instruction, to the stack, and its counterpart instruction **`return_to_calling_program`** pops the return address from the stack when it terminates the execution of the subroutine.

Figure 4-11b shows how the contents of the registers change when the program in Figure 4-11a is being executed. Note that all registers except PROGRAM POINTER contain FFH when the computer starts operating. Note also, that in addition to the contents of all registers, the contents of memory locations FEH and FFH are shown in Figure 4-11b. The value in memory location FFH, the first free position of the stack, changes when a subroutine call is made. Figure 4-11b shows that after the **`call_subroutine`** instruction, 81H, is executed, both PROGRAM POINTER (PP) and STACK POINTER (SP) have new values, and stack memory location FFH contains a return address to the calling program. An appropriate return address is on the stack during the time when the subroutine is being executed. At the end of the subroutine, when the instruction with code 82H is executed, the return address is read from the stack and STACK POINTER is incremented by one.

The stack is simply one particular area of the main memory. This special memory area is automatically used in subroutine calls to store return addresses in calling programs. The stack is particularly useful in subroutine calls because it can also handle the difficult situation when a called subroutine calls another subroutine. In such situations, many return addresses are pushed onto the stack, and when they are popped away from the stack, they automatically come out in the correct order.

Exercises with programs abcde.iml and aaaabbbbcccc.iml

Exercise 4-6. Which small modification should be made to program **abcde.iml** in order to make it print letters ABCDEFGHI (nine letters from the beginning of the alphabet instead of just five letters) ?

Exercise 4-7. With very small modification to the textual form of **abcde.iml** the program would print HIJKL. What should be done to make this happen?

Exercise 4-8. Program **abcde.iml** can be made to print EDCBA by making two modifications to the program. The first modification is that 'A' should be changed to 'E' in Figure 4-10a. The other modification involves replacing an instruction with another similar instruction. Which are these instructions?

Exercise 4-9. How can program **abcde.iml** be made to print AABBCCDDEE ? This can be achieved by inserting one new instruction inside the loop of the program.

Exercise 4-10 With which small modification can **aaaabbbbcccc.iml** be made to print xxxxyyyyzzzz ? Which instruction should be removed to make it print aaaabbbbbbbb ?

Remember that you can verify your answers with the IC8 simulator!

4.11 Programs that use the keyboard, memory area, and stack

In this section we will study two IML programs which both read a text from the keyboard, and display the characters of the text in reverse order. This means that if, for example, the text "Hello " is typed in from the keyboard, the screen of the imaginary computer will look like

```
Hello  olleH
```

after program execution. These programs again show us that there are usually several possibilities to write a computer program to perform a certain activity.

Program **reverse_in_memory.iml**, which is shown in Figure 4-12a, is the first program to print the characters of a text in reverse order. The program reads characters from the keyboard until a space character ' ' has been entered. Each character is stored in a reserved memory area. When the program has read the space character from the keyboard, it starts printing the whole text in reverse order. The text is printed in reverse order because the memory area is processed backwards.

You should consult tables 4-1 and 4-2 to find out what the instructions in Figure 4-12a do. In addition, Figure 4-12b shows what happens inside the imaginary processor and on the screen when program **reverse_in_memory.iml** is being executed. The execution of the program takes so long that only the beginning of it can be seen in Figure 4-12b. If you study Figure 4-12b carefully, you will notice that the start address of the memory area is 19H. The name **memory_for_characters** refers to this address. For example, if the text "Hello " is given to the program, it is stored in the memory in the following way:

```
ADDRESS    CHARACTER CODE   CHARACTER

19H          00H
1AH          48H            'H'
1BH          65H            'e'
1CH          6CH            'l'
1DH          6CH            'l'
1EH          6FH            'o'
1FH          20H            ' '
```

The program uses a kind of trick when it stores value 00H to the first location of the reserved memory area. When the program reads the characters from the memory, while it is printing them in reverse order, the value 00H tells it when to stop reading and printing. The program does not need to know how many characters were entered from the keyboard. The value 00H marks the end of the characters in their reverse order. 00H is a convenient value for this kind of purpose because it is not a character code of any visible character.

Program **reverse_in_memory.iml** contains three loops. The shortest loop is

```
waiting_a_character:
      jump_if_input_not_ready        waiting_a_character
```

When the above instruction is executed, it causes a jump to itself, the same instruction, if the input is not ready. The program waits in this loop until the user of the program types in a character from the keyboard. The program keeps waiting forever if the user decides not to touch the keyboard. The loop terminates when signal INPUT READY inside the imaginary processor is set to value 1 (true). That happens when the INPUT PORT receives a character code from the keyboard.

The short loop is inside another loop which has the following structure

```
read_character:

      ...

      jump_if_registers_equal      print_characters
      jump_to_address              read_character

print_characters:
```

This loop can be called the input loop. As usual, the loop is constructed by using first a conditional jump instruction, and then an unconditional jump instruction. The conditional jump instruction makes the loop terminate when the code of the space character ' ' is in both registers A and B.

The last loop, the output loop, is also made with two jump instructions. The instruction

```
jump_if_register_a_zero      all_characters_printed
```

makes the program jump to its end when register A contains value 00H. That value is the trick value which marks the beginning of the reserved memory area.

You may be wondering why there is an output instruction inside the input loop. The output instruction is written right after the input instruction:

```
input_byte_to_register_a
output_byte_from_register_a
```

The reason for having the output instruction in the input loop is that it is convenient for the user of the program to see what he or she is typing into the computer. When a low-level programming language like IML is used, programs must be written so that they take care that input data is displayed on the screen. This activity is called echoing. Program **reverse_in_memory.iml** thus echoes the input to the screen.

Program **reverse_in_stack.iml**, which is shown in Figure 4-13a and "executed" in Figure 4-13b, is another program which prints the characters of its input text in reverse order. The main difference between the two programs is that, instead of a memory area, **reverse_in_stack.iml** uses the stack to store the input text.

The last bytes of the main memory of the imaginary computer are used as stack memory. Register STACK POINTER controls the use of the stack. For example, when instruction

```
push_register_a_to_stack
```

is executed for the first time in program **reverse_in_stack.iml**, value 00H is written to memory location FFH which is the current value of register STACK POINTER. After that the value of STACK POINTER is decremented to value FEH. This way the next push instruction writes memory location FEH.

When program **reverse_in_stack.iml** is executed by giving it text "Hello ", the last part of the main memory looks like the following

ADDRESS	CHARACTER CODE	CHARACTER
F9H	20H	' '
FAH	6FH	'o'
FBH	6CH	'l'
FCH	6CH	'l'
FDH	65H	'e'
FEH	48H	'H'
FFH	00H	

after the execution of the program. The stack is a particularly useful means to reverse the characters of a text because the stack always gives away the last byte that was put onto it.

Therefore, the last character of the input text comes out first from the stack. Characters are popped away from the stack with the instruction

```
pop_register_a_from_stack
```

which first increments the value of register STACK POINTER, and then copies the content of the memory location whose address is in STACK POINTER to register A.

Another difference between programs **reverse_in_memory.iml** and **reverse_in_stack.iml** is that the latter calls a subroutine to read characters from the keyboard. You should note, while studying the program execution in Figure 4-13b, that values on the stack change also because the subroutine calling mechanism uses the stack. A subroutine which reads a character from the keyboard is particularly useful. The subroutine `read_and_echo_a_character` could be copied from program **reverse_in_stack.iml** and used in many other programs.

Exercises with programs reverse_in_memory.iml and reverse_in_stack.iml

Exercise 4-11. Now the programs stop reading characters from the keyboard when they encounter a space character ' ', 20H. It is thus not possible to type in complete sentences which contain spaces between words. Modify program **reverse_in_memory.iml** so that, instead of a space character, it stops reading characters when it encounters character '.', the full stop.

Exercise 4-12. Modify program **reverse_in_stack.iml** so that you take away the call to subroutine `read_and_echo_a_character`, and input the characters inside the first loop of the main program. This can be achieved by moving the instructions which are inside the subroutine to the input loop of the calling program. (I'm asking you to do this just as an exercise. In general, it is a good habit to use subroutines in computer programming.)

Exercise 4-13. Modify program **reverse_in_memory.iml** so that you put there the subroutine `read_and_echo_a_character` which is in **reverse_in_stack.iml**. You should call the subroutine in the input loop in the same way as it is called in **reverse_in_stack.iml**. The behavior of the program may not change.

The last two exercises require quite a lot of modifications to the programs. You must hand-compile the programs and carefully calculate correct memory addresses if you test your answers with the IC8 simulator.

Other IML programs

The electronic material that is available for the readers of this book contains many IML programs which are not discussed in this chapter. These programs are also built in the simulators. In this chapter only the most essential programs are shown and explained.

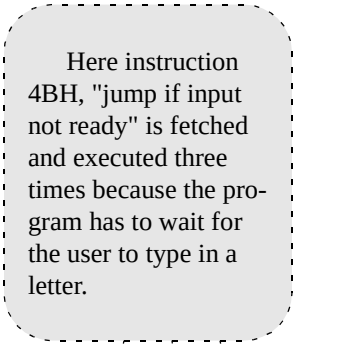
If you are interested, you can study the other programs by printing them on paper, and by running them with a simulator. Among the extra programs there is one which shows what happens when a subroutine calls another subroutine. Another interesting program shows how numbers can be multiplied with the imaginary computer. Although the imaginary processor does not have machine instructions to perform multiplication operations, it is possible to multiply by executing many addition operations in a loop.

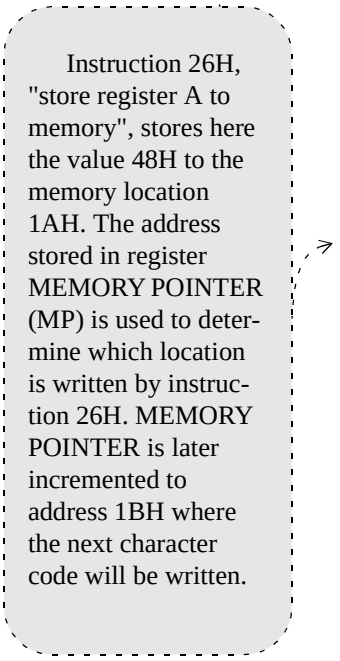
```
//  reverse_in_memory.iml  (c) 1997 - 2000 Kari Laitinen

//  The following program reads text from the keyboard. After the
//  space key has been pressed, the program displays the characters
//  of the entered text in reverse order. Thus if the user typed in
//
//      Hello
//
//  the computer would respond
//
//       olleH

beginning_of_program:
      load_register_a_with_value       0
      set_memory_pointer               memory_for_characters
      store_register_a_to_memory

read_character:
      increment_memory_pointer
waiting_a_character:
      jump_if_input_not_ready          waiting_a_character
      input_byte_to_register_a
      output_byte_from_register_a
      store_register_a_to_memory
      load_register_b_with_value       ' '   // code for space
      jump_if_registers_equal          print_characters
      jump_to_address                  read_character

print_characters:
      output_byte_from_register_a
      decrement_memory_pointer
      load_register_a_from_memory
      jump_if_register_a_zero          all_characters_printed
      jump_to_address                  print_characters

all_characters_printed:
      stop_processing

memory_for_characters:     DATA        20
```

This definition reserves a memory area of 20 bytes where the character codes, which are read from the keyboard, are stored. An IML compiler puts this memory area to those memory locations which follow the executable machine instructions. Because the machine instructions need the first 25 (19H) bytes from the main memory, this memory area starts from address 19H. The executable program starts, as always, from address 00H.

The text that follows the character pair //, double slash, on this line is a comment, which is ignored by the compiler. The double slash is part of the comment. The text before the double slash on this line is a valid IML instruction.

Figure 4-12a. A program that prints the characters of an input text in reverse order.





Status	PP	IC	IO	RA	RB	MP	SP	FE	FF	Screen contents
RESET	00	ff	ff	ff	ff	ff	ff	ff	ff	
FETCH	01	15	ff	ff	ff	ff	ff	ff	ff	
FETCH	02	15	00	ff	ff	ff	ff	ff	ff	
EXECUTE	02	15	00	00	ff	ff	ff	ff	ff	
FETCH	03	21	00	00	ff	ff	ff	ff	ff	
FETCH	04	21	19	00	ff	ff	ff	ff	ff	
EXECUTE	04	21	19	00	ff	19	ff	ff	ff	
FETCH	05	26	19	00	ff	19	ff	ff	ff	
EXECUTE	05	26	19	00	ff	19	ff	ff	ff	
FETCH	06	22	19	00	ff	19	ff	ff	ff	
EXECUTE	06	22	19	00	ff	1a	ff	ff	ff	
FETCH	07	4b	19	00	ff	1a	ff	ff	ff	
FETCH	08	4b	06	00	ff	1a	ff	ff	ff	
EXECUTE	06	4b	06	00	ff	1a	ff	ff	ff	
FETCH	07	4b	06	00	ff	1a	ff	ff	ff	
FETCH	08	4b	06	00	ff	1a	ff	ff	ff	
EXECUTE	06	4b	06	00	ff	1a	ff	ff	ff	
FETCH	07	4b	06	00	ff	1a	ff	ff	ff	
FETCH	08	4b	06	00	ff	1a	ff	ff	ff	
EXECUTE	08	4b	06	00	ff	1a	ff	ff	ff	
FETCH	09	96	06	00	ff	1a	ff	ff	ff	
EXECUTE	09	96	06	48	ff	1a	ff	ff	ff	
FETCH	0a	92	06	48	ff	1a	ff	ff	ff	
EXECUTE	0a	92	06	48	ff	1a	ff	ff	ff	H
FETCH	0b	26	06	48	ff	1a	ff	ff	ff	H
EXECUTE	0b	26	06	48	ff	1a	ff	ff	ff	H
FETCH	0c	17	06	48	ff	1a	ff	ff	ff	H
FETCH	0d	17	20	48	ff	1a	ff	ff	ff	H
EXECUTE	0d	17	20	48	20	1a	ff	ff	ff	H
FETCH	0e	43	20	48	20	1a	ff	ff	ff	H
FETCH	0f	43	11	48	20	1a	ff	ff	ff	H
EXECUTE	0f	43	11	48	20	1a	ff	ff	ff	H
FETCH	10	41	11	48	20	1a	ff	ff	ff	H
FETCH	11	41	05	48	20	1a	ff	ff	ff	H
EXECUTE	05	41	05	48	20	1a	ff	ff	ff	H
FETCH	06	22	05	48	20	1a	ff	ff	ff	H
EXECUTE	06	22	05	48	20	1b	ff	ff	ff	H
FETCH	07	4b	05	48	20	1b	ff	ff	ff	H
FETCH	08	4b	06	48	20	1b	ff	ff	ff	H
EXECUTE	06	4b	06	48	20	1b	ff	ff	ff	H
FETCH	07	4b	06	48	20	1b	ff	ff	ff	H
FETCH	08	4b	06	48	20	1b	ff	ff	ff	H
EXECUTE	08	4b	06	48	20	1b	ff	ff	ff	H
FETCH	09	96	06	48	20	1b	ff	ff	ff	H
EXECUTE	09	96	06	65	20	1b	ff	ff	ff	H
FETCH	0a	92	06	65	20	1b	ff	ff	ff	H
EXECUTE	0a	92	06	65	20	1b	ff	ff	ff	He
FETCH	0b	26	06	65	20	1b	ff	ff	ff	He
EXECUTE	0b	26	06	65	20	1b	ff	ff	ff	He
FETCH	0c	17	06	65	20	1b	ff	ff	ff	He
... etc.										

Figure 4-12b. Step-by-step execution of the program in Figure 4-12a.

```
//  reverse_in_stack.iml  (c) 1997 - 2000 Kari Laitinen

//  The following program is similar to "reverse_in_memory.iml".
//  It reads a text from keyboard as a string of characters.
//  After receiving a space, it displays the characters
//  in reverse order. This program puts the characters into the
//  stack. At the end it reads the characters away from the stack.
//  Due to the nature of stack as a data storage, the characters
//  will automatically come out in reverse order.

beginning_of_program:
      load_register_a_with_value       0
      push_register_a_to_stack

read_character:
      call_subroutine                  read_and_echo_a_character
      push_register_a_to_stack
      load_register_b_with_value       ' '  //  code for space
      jump_if_registers_equal          print_characters_from_stack
      jump_to_address                  read_character

print_characters_from_stack:
      pop_register_a_from_stack
      jump_if_register_a_zero          all_characters_printed
      output_byte_from_register_a
      jump_to_address                  print_characters_from_stack

all_characters_printed:
      stop_processing


read_and_echo_a_character:
      jump_if_input_not_ready          read_and_echo_a_character
      input_byte_to_register_a
      output_byte_from_register_a
      return_to_calling_program
```

This is a subroutine which is called to read a character from the keyboard. The address name **read_and_echo_a_character** refers to the beginning of the subroutine, and is also the name of the subroutine. This subroutine echoes the read character to the screen. The calling program receives the character in register A.

This jump instruction is executed as long as nothing has been entered from the keyboard. The program waits here until the user types in a character.

Figure 4-13a. Another version of a program to reverse the characters of a text.

Here instruction 81H, "call subroutine", is executed. The subroutine is in address 13H. At this moment the value 00H and the return address 05H are on the stack.

When instruction A2H is executed here, the content of register A (48H, the character code of letter H) is copied to the top of the stack which is in address FEH. After that the value of register STACK POINTER is decremented to value FDH which is the new top location of the stack. Note that the stack locations FDH, FCH, FBH, etc. are not shown here because of space limitations.

Status	PP	IC	IO	RA	RB	MP	SP	FE	FF	Screen contents
RESET	00	ff	ff	ff	ff	ff	ff	ff	ff	
FETCH	01	15	ff	ff	ff	ff	ff	ff	ff	
FETCH	02	15	00	ff	ff	ff	ff	ff	ff	
EXECUTE	02	15	00	00	ff	ff	ff	ff	ff	
FETCH	03	a2	00	00	ff	ff	ff	ff	ff	
EXECUTE	03	a2	00	00	ff	ff	fe	ff	00	
FETCH	04	81	00	00	ff	ff	fe	ff	00	
FETCH	05	81	13	00	ff	ff	fe	ff	00	
EXECUTE	13	81	13	00	ff	ff	fd	05	00	
FETCH	14	4b	13	00	ff	ff	fd	05	00	
FETCH	15	4b	13	00	ff	ff	fd	05	00	
EXECUTE	13	4b	13	00	ff	ff	fd	05	00	
FETCH	14	4b	13	00	ff	ff	fd	05	00	
FETCH	15	4b	13	00	ff	ff	fd	05	00	
EXECUTE	15	4b	13	00	ff	ff	fd	05	00	
FETCH	16	96	13	00	ff	ff	fd	05	00	
EXECUTE	16	96	13	48	ff	ff	fd	05	00	
FETCH	17	92	13	48	ff	ff	fd	05	00	
EXECUTE	17	92	13	48	ff	ff	fd	05	00	H
FETCH	18	82	13	48	ff	ff	fd	05	00	H
EXECUTE	05	82	13	48	ff	ff	fe	05	00	H
FETCH	06	a2	13	48	ff	ff	fe	05	00	H
EXECUTE	06	a2	13	48	ff	ff	fd	48	00	H
FETCH	07	17	13	48	ff	ff	fd	48	00	H
FETCH	08	17	20	48	ff	ff	fd	48	00	H
EXECUTE	08	17	20	48	20	ff	fd	48	00	H
FETCH	09	43	20	48	20	ff	fd	48	00	H
FETCH	0a	43	0c	48	20	ff	fd	48	00	H
EXECUTE	0a	43	0c	48	20	ff	fd	48	00	H
FETCH	0b	41	0c	48	20	ff	fd	48	00	H
FETCH	0c	41	03	48	20	ff	fd	48	00	H
EXECUTE	03	41	03	48	20	ff	fd	48	00	H
FETCH	04	81	03	48	20	ff	fd	48	00	H
FETCH	05	81	13	48	20	ff	fd	48	00	H
EXECUTE	13	81	13	48	20	ff	fc	48	00	H
FETCH	14	4b	13	48	20	ff	fc	48	00	H
FETCH	15	4b	13	48	20	ff	fc	48	00	H
EXECUTE	13	4b	13	48	20	ff	fc	48	00	H
FETCH	14	4b	13	48	20	ff	fc	48	00	H
FETCH	15	4b	13	48	20	ff	fc	48	00	H
EXECUTE	15	4b	13	48	20	ff	fc	48	00	H
FETCH	16	96	13	48	20	ff	fc	48	00	H
EXECUTE	16	96	13	65	20	ff	fc	48	00	H
FETCH	17	92	13	65	20	ff	fc	48	00	H
EXECUTE	17	92	13	65	20	ff	fc	48	00	He
FETCH	18	82	13	65	20	ff	fc	48	00	He
EXECUTE	05	82	13	65	20	ff	fd	48	00	He
FETCH	06	a2	13	65	20	ff	fd	48	00	He
EXECUTE	06	a2	13	65	20	ff	fc	48	00	He
FETCH	07	17	13	65	20	ff	fc	48	00	He
... etc.										

Figure 4-13b. Step-by-step execution of the program in Figure 4-13a.

4.12 Chapter summary – towards high-level programming

In this chapter, we have studied an imaginary computer and the simple IML language which we used to write programs for the computer. Our aim was to learn how computers work in general: how they execute sequences of machine instructions which are called programs. Machine instructions are numerical codes which the processor executes in a certain way.

A computer programmer needs to know something about the logical operation of computers. That is the reason why the imaginary computer and its machine-level programming language were introduced here. However, I would like to emphasize that a serious computer programmer should not write programs by using languages like IML. It is better to use high-level languages such as C#, which allow one to think in terms of application domain concepts.

During the early days of computing, machine-level programming languages similar to IML were widely used in software development. They have been traditionally called assembly languages. However, because software development was found to be rather difficult with assembly languages, high-level programming languages were invented. Writing programs with assembly languages is difficult because of the following reasons:

- The program writer must know the registers and the behavior of the processor for which he or she is writing a program.
- An assembly language programmer must think in terms of memory locations and register contents.
- Even for simple operations, many lines of source program code must be written with an assembly language.

High-level languages allow programmers to pay more attention to what the programs should do, and they free programmers from knowing the internal structure of the used processor. High-level programming languages are usually machine-independent, which means that a program written with a high-level language can be run by different computers. Every type of computer must, though, have its own compiler to transform a high-level program into machine instructions of that particular computer. C# is a high-level programming language. Although a C# compiler is available only for personal computers and the Windows operating system, it would be possible to build a C# compiler for a different operating system, and all C# programs could be compiled to be run in that operating system.

Programs written with a high-level language are usually shorter than programs written with a language like IML. To demonstrate this, let's consider the following piece of IML source program:

```
// Some values could be stored beforehand to memory
// locations first_number and second_number.

      set_memory_pointer              first_number
      load_register_a_from_memory
      set_memory_pointer              second_number
      load_register_b_from_memory

      add_register_b_to_a
      set_memory_pointer              sum_of_numbers
      store_register_a_to_memory

first_number:     DATA      1
second_number:    DATA      1
sum_of_numbers:   DATA      1
```

The instructions above first calculate the sum of the numbers that are in the memory locations which have names **first_number** and **second_number**. Then the sum is stored in the memory location which has name **sum_of_numbers**. If a high-level language like the C# programming language existed for the imaginary computer, the above IML instructions could be written with the high-level language in the following way

```
int  first_number ;
int  second_number ;
int  sum_of_numbers ;

// Some values should be stored to variables
// first_number and second_number.

sum_of_numbers  =  first_number  +  second_number ;
```

By comparing the two pieces of program it is easy to see that the program written with the high-level language is shorter. Later on, after you have learned C# programming, you will most likely say that the high-level program is also easier to understand.

You may wonder why we studied the IML language if high-level programming languages are more convenient. The answer is, that in order to understand high-level languages properly, it is important to know something about processors and machine-level programming. These matters are easy to explain with a simple processor and with a simple language. Without knowing anything about machine-level programming, it is hard to understand why high-level programs need to be compiled, and what is happening in compilation. Just like an IML compiler, the compiler of a high-level programming language translates textual source programs to list of numerical machine instructions. If there existed a compiler to translate the above high-level program for the imaginary computer, the compiler could produce the same machine instructions which would result in the translation of the corresponding piece of an IML source program. (In reality, however, it is very difficult to build such a compiler.)

"Bugs" in computer programs

While reading this chapter you may have realized, or you may have experienced personally with the IC8 simulator, that it is possible to make computer programs which do not behave as intended, or which never terminate. The following is an example of a program that never terminates:

```
beginning_of_program:
     load_register_a_with_value         'X'
     output_byte_from_register_a
     jump_to_address                    beginning_of_program
```

The above program is an infinite (endless) loop which keeps displaying the letter X forever. These kinds of loops can be made accidentally in computer programs. When a computer program does not work as originally planned, we say that there is an error in the program. A never-terminating program is one example of an erroneous program. There can be many kinds of errors in programs, and it is common in the world of computers to call errors by the word "bug". The activity of searching and removing errors in computer programs is called "debugging". Computer programs which can help in the search of errors in other programs are called debugging tools, or simply debuggers.

The English word "bug" can mean any small insect. That word started to mean a problem in a computer because small insects really did cause malfunctions in early computers. A small insect could cause an erroneous electrical connection among the relays of an early computer, and make the computer behave in an unexpected manner. Small insects are no threat to modern computers because their electronic circuits are so small that no bug can walk inside them, but humans can still make all kinds of errors when they write computer programs.