CHAPTER 4

A Simple Computer Architecture

Modern computers are *digital systems* that process digital information. The digital information is represented by two discrete values 0 and 1. A digital system is typically divided into two parts: a datapath and a control unit. As a matter of fact, most CPUs consist of these two parts. The datapath is a collection of functional units, registers, and interconnections between them that together perform data-processing operations.

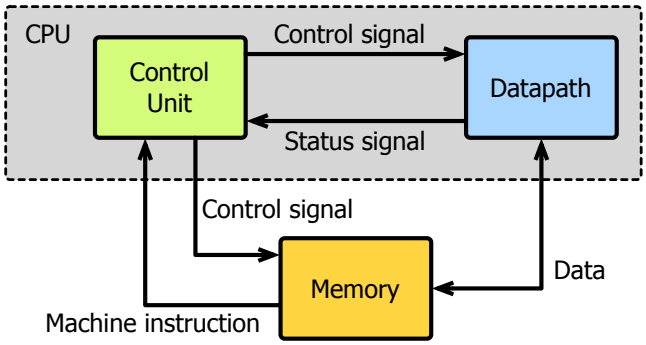


Figure 4.1: The relationship between the control unit, datapath, and main memory.

The *control unit* (CU) controls operations of the datapath and determines the sequence of the operations. It also coordinates interactions between the datapath and main memory (Figure 4.1). Machine instructions in a program stored in main memory are fetched one by one. Each machine instruction is decoded by the CU, and the CU generates control signals required to execute the instruction. The datapath sends status signals to the CU after executing an instruction. The CU takes an appropriate action according to the status signal.

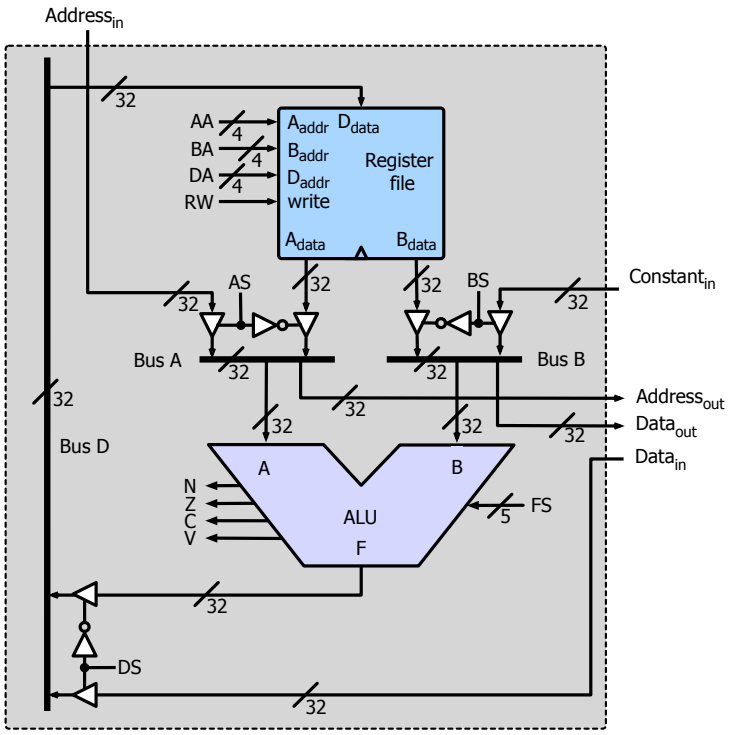


Figure 4.2: A 32-bit datapath.

4.1 Datapath

Figure 4.2 shows a simple 32-bit datapath that consists of a register file and an ALU. The organizations of the register file and ALU are explained in the previous chapter. The input AA (BA) to the register file selects a register for the output A_{data} (B_{data}) that is connected to the bus A (B).

The selection input AS connects either A_{data} from the register file or the address $Address_{in}$ from outside the datapath to the bus A. Similarly, the selection input BS connects either B_{data} or the data $Constant_{in}$ from outside the datapath to the bus B. The operands A and B of the ALU come from the bus A and bus B, respectively. The bus A also connects to $Address_{out}$ to send an address outside the datapath to other components of the system (e.g., memory). Similarly, the bus B connects to $Data_{out}$ to send data outside the datapath (e.g., to memory). An ALU operation is selected by the 5-bit input FS to the ALU. The selection input DS for the bus D selects one between the output F of the ALU and the data $Data_{in}$ from outside of the datapath. Specifying a register using the input DA and activating the input RW make the value appearing on the bus D to be transferred to the specified register.

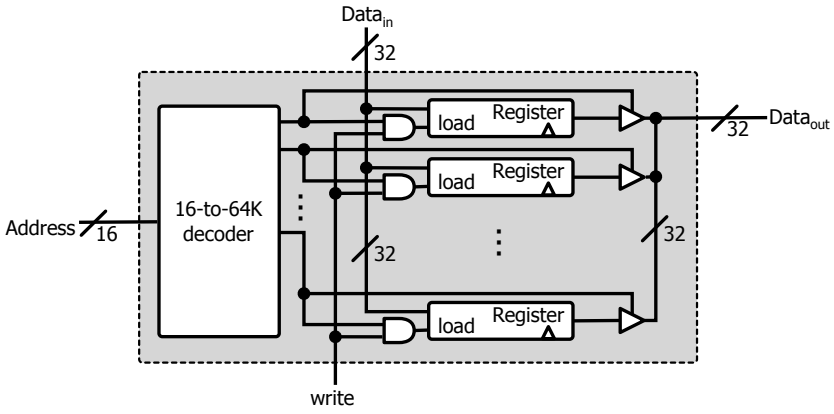


Figure 4.3: The memory treated as an array of registers.

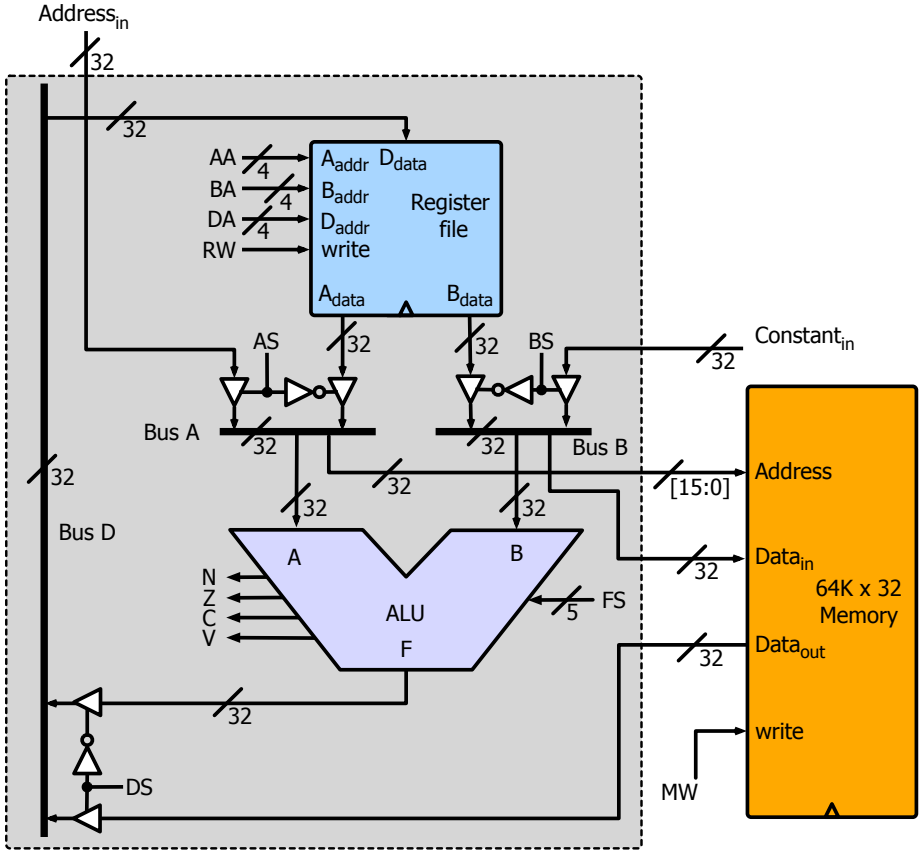
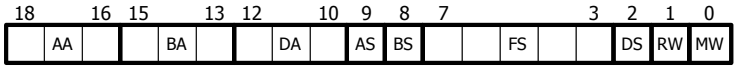


Figure 4.4: Connecting a memory unit to the datapath.



(a)

Control bits	Description
AA	specifies a register for the value on the bus A
BA	specifies a register for the value on the bus B
DA	specifies a register to which the value on the bus D is loaded
AS	selects one between the value of a register and $Address_{in}$ for the value on the bus A
BS	selects one between the value of a register and $Constant_{in}$ for the value on the bus B
FS	selects an ALU operation
DS	selects one between the result of the ALU and the data from the memory for the value on the bus D
RW	stores the value on the bus D to the register specified by DA
MW	stores the value that appears on the bus B into the memory location specified by the address on the bus A

(b)

Figure 4.5: The control word: (a) its format; (b) the description of its content.

4.2 Attaching a RAM to the Datapath

The steps required to read or write a word in the memory can be precisely controlled by the CU to satisfy the timing requirements. However, the easiest way of ignoring the details of the timing requirements is treating the memory as an array of 2^m w-bit registers as shown in Figure 4.3, where m is the number of address bits and w is the word size. In this case, storing a value into the memory occurs on every positive edge of the clock signal when MW is 1. Such a memory can be attached to the datapath in the way as shown in Figure 4.4. The address output $Address_{out}$ from the bus A in the datapath is connected to the address input lines of the memory. The data output $Data_{out}$ from and the data input $Data_{in}$ to the datapath are connected to the data input lines and data output lines of the memory, respectively. The control input MW controls write operations to the memory.

To write a new value into the memory, the address of the desired word is applied to the bus A. In turn, the control input MW is activated by the CU to enable the write operation. Then, the new value is applied to the bus B. Finally, the value is written to the desired location.

Among those inputs to the datapath and memory, AA, BA, DA, AS, BS, FS,

DS, RW, and MW are control inputs that specify *microoperations* executed by the datapath and memory. A collection of these control bits are called a *control word*, and it is provided by the CU in a prescribed manner. The content of a control word determines the operation performed in the datapath and memory. The format of a control word for the datapath and memory is shown in Figure 4.5 (a). The role of each control input to the datapath and memory is described in the table shown in Figure 4.5 (b).

The elementary operations on the data stored in the registers are typically called microoperations. Microoperations on the datapath include loading a constant value to a register, loading the content of one register to another, adding the contents of two registers and storing the result to another, storing the contents of one register in the memory, etc. In our case, each microopration completes in a single clock cycle.

4.3 Register Transfer Language

Register transfer language (*RTL*) is a convenient notation for representing microoperations. There are total 8 registers in the register file of our datapath. We name each register in the register file as Rx, where x is the address of the register in the register file ranging from 0 to 7. That is, we have registers R0, R1, R2, R3, R4, R5, R6, and R7.

Register transfer microoperations

RTL statements for data transfer microoperations between registers are in the following form:

$$U \leftarrow V$$

where U and V are registers (possibly the same), but V may be a constant. It denotes that the content of a register or a constant V is transferred to a register U. The content of U is overwritten on the positive edge of the next clock cycle, but the content of V remains unchanged. The following is some examples of RTL statements that perform register transfer microoperations and the corresponding control word produced by the CU (a hexadecimal number is prefixed with 0x):

- $R2 \leftarrow R1$ (a register to another register)

AA	BA	DA	AS	BS	FS	DS	RW	MW	$Constant_{in}$	$Address_{in}$
0 0 1	X X X	0 1 0	0	X	0 0 0 0 0	0	1	0	0xXXXXXXXX	0xXXXXXXXX

- $R3 \leftarrow 5$ (transferring a constant to a register)

AA	BA	DA	AS	BS	FS	DS	RW	MW	$Constant_{in}$	$Address_{in}$
X X X	X X X	0 1 1	X	1	1 0 0 X X	0	1	0	0x00000005	0xXXXXXXXX

- $R5 \leftarrow 0$ $(R0 \oplus R0 = 0)$

AA	BA	DA	AS	BS	FS	DS	RW	MW	Constant_{in}	Address_{in}
0 0 0	0 0 0	1 0 1	0	0	0 1 1 0 X	0	1	0	0xXXXXXXXX	0xXXXXXXXX

Arithmetic microoperations

An arithmetic microoperation performs an arithmetic operation provided by the arithmetic unit in the ALU. Its RTL statement has the following form:

$$U \leftarrow V \ \mathbf{op}_a \ W$$

where U and V are registers, and W may be either of a register and a constant. Two or all three of U, V, and W are possibly the same. The binary operation $\mathbf{op}_a$ is one of $+$ and $-$. The meaning is that a binary arithmetic operation $\mathbf{op}_a$ is performed on the contents of V and W (W can be a constant), and the result is transferred to U. The content of U is overwritten, but the contents of V and W remain unchanged. The following is some examples of RTL statements that perform arithmetic microoperations:

- $R0 \leftarrow R1 + R2$ (addition)

AA	BA	DA	AS	BS	FS	DS	RW	MW	Constant_{in}	Address_{in}
0 0	1 0	1 0 0 0	0	0	0 0 0 1 0	0	1	0	0xXXXXXXXX	0xXXXXXXXX

- $R3 \leftarrow R7 - 4$ (subtraction, $R7 + 4' + 1$, 4' is the one's complement of 4)

AA	BA	DA	AS	BS	FS	DS	RW	MW	Constant_{in}	Address_{in}
1 1 1	X X X	0 1 1	0	1	0 0 1 0 1	0	1	0	0x00000004	0xXXXXXXXX

- $R3 \leftarrow R1 + 1$ (increment R1)

AA	BA	DA	AS	BS	FS	DS	RW	MW	Constant_{in}	Address_{in}
0 0 1	X X X	0 1 1	0	X	0 0 0 0 1	0	1	0	0xXXXXXXXX	0xXXXXXXXX

- $R5 \leftarrow R1 - 1$ (decrement R1)

AA	BA	DA	AS	BS	FS	DS	RW	MW	Constant_{in}	Address_{in}
0 0 1	X X X	1 0 1	0	X	0 0 1 1 0	0	1	0	0xXXXXXXXX	0xXXXXXXXX

Logic microoperations

A logic microoperation performs a logic operation provided by the logic unit in the ALU. An RTL statement that performs a binary logic operation has the following form:

$$U \leftarrow V \ \mathbf{op}_l \ W$$

where U and V are registers, and W may be either of a register and a constant. Two or all three of U, V, and W are possibly the same. The binary operation $\mathbf{op}_l$ is one of $\wedge$, $\vee$, and $\oplus$. The meaning of the RTL statement is that a bitwise binary logic operation $\mathbf{op}_l$ is performed on the contents of V and W (W can be a constant), and the result is transferred to U. The content of U is overwritten, but the contents of V and W remain unchanged.

A unary logic microoperations is bitwise negation ('). An RTL statement that performs the bitwise negation operation has the following form:

$$U \leftarrow V'$$

where U is a register, and V may be either of a register and a constant. U and V are possibly the same. The bitwise negation operation is performed on the content of V or a constant V, and the result is transferred to U. The content of U is overwritten, but the content of V remains unchanged. The following is some examples of RTL statements that perform logic microoperations:

- $R0 \leftarrow R1'$ (bitwise NOT)

AA	BA	DA	AS	BS	FS	DS	RW	MW	Constant_{in}	Address_{in}
0 0 1	X X X	0 0 0	0	X	0 1 1 1 X	0	1	0	0xXXXXXXXX	0xXXXXXXXX

- $R3 \leftarrow R1 \wedge R2$ (bitwise AND)

AA	BA	DA	AS	BS	FS	DS	RW	MW	Constant_{in}	Address_{in}
0 0 1	0 1 0	0 1 1	0	0	0 1 0 1 X	0	1	0	0xXXXXXXXX	0xXXXXXXXX

- $R1 \leftarrow R1 \vee R2$ (bitwise OR)

AA	BA	DA	AS	BS	FS	DS	RW	MW	Constant_{in}	Address_{in}
0 0 1	0 1 0	0 0 1	0	0	0 1 0 0 X	0	1	0	0xXXXXXXXX	0xXXXXXXXX

- $R3 \leftarrow R1 \oplus R2$ (bitwise XOR)

AA	BA	DA	AS	BS	FS	DS	RW	MW	Constant_{in}	Address_{in}
0 0 1	0 1 0	0 1 1	0	0	0 1 1 0 X	0	1	0	0xXXXXXXXX	0xXXXXXXXX

- $R3 \leftarrow R1 \vee$ 0x0F0F0F0F

AA	BA	DA	AS	BS	FS	DS	RW	MW	Constant_{in}	Address_{in}
0 0 1	X X X	0 1 1	0	1	0 1 0 0 X	0	1	0	0x0F0F0F0F	0xXXXXXXXX

Shift microoperations

A shift microoperation performs a 1-bit shift left or right operation provided by the shifter in the ALU. An RTL statement that performs a shift microoperation has the following form: It has the following form:

$$U \leftarrow \mathbf{op}_s\ V$$

where U is a register, and V may be either of a register and a constant. The unary operation $\mathbf{op}_s$ is one of lsl (logical shift left), lsr (logical shift right), and asr (arithmetic shift right). The meaning of the statement is that a 1-bit shift operation $\mathbf{op}_s$ is performed on the content of a register V or a constant V. The result is transferred to U. The content of U is overwritten, but the content of V remains unchanged. The following is some examples of RTL statements that perform shift microoperations:

- $R0 \leftarrow lsl\ R1$

AA			BA			DA			AS	BS	FS				DS	RW	MW	Constant_{in}	Address_{in}
X	X	X	0	0	1	0	0	0	0	X	1	1	0	X	X	0	1	0	0xXXXXXXXX	0xXXXXXXXX

- $R2 \leftarrow lsr\ R5$

AA			BA			DA			AS	BS	FS				DS	RW	MW	Constant_{in}	Address_{in}
X	X	X	1	0	1	0	1	0	0	X	1	0	1	0	X	0	1	0	0xXXXXXXXX	0xXXXXXXXX

- $R3 \leftarrow asr\ R7$

AA			BA			DA			AS	BS	FS				DS	RW	MW	Constant_{in}	Address_{in}
X	X	X	1	1	1	0	1	1	0	X	1	0	1	1	X	0	1	0	0xXXXXXXXX	0xXXXXXXXX

- $R4 \leftarrow asr$ 0x80FF0001

AA			BA			DA			AS	BS	FS				DS	RW	MW	Constant_{in}	Address_{in}
X	X	X	X	X	X	1	0	0	X	1	1	0	1	1	X	0	1	0	0x80FF0001	0xXXXXXXXX

Memory transfer microoperations

A memory transfer microoperation performs a data transfer operation between the memory and a register. There are two memory transfer microoperations: read and write. A read microoperation transfers a data word within the memory to a register. An RTL statement that performs a memory read operation has the following form:

$$U \leftarrow M[V]$$

where U and V are possibly the same registers. The address of the desired word is given by the content of V. The content of U is overwritten, but the word in the memory remains unchanged.

A write microoperation transfers a data word stored a register to a memory word. An RTL statement that performs a memory write operation has the following form:

$$M[V] \leftarrow U$$

where U and V are possibly the same registers, but U may be a constant. The address of the desired word is the content of V. A write operation makes the content of a register U or a constant U to be transferred to the memory word specified by V. The content of the specified word in the memory is overwritten, but U remains unchanged. The following is some examples of RTL statements that perform memory transfer microoperations:

- $R0 \leftarrow M[R1]$

AA	BA	DA	AS	BS	FS	DS	RW	MW	$Constant_{in}$	$Address_{in}$
0 0 1	X X X	0 0 0	0	X	X X X X X	1	1	0	0xXXXXXXXX	0xXXXXXXXX

- $M[R2] \leftarrow R4$

AA	BA	DA	AS	BS	FS	DS	RW	MW	$Constant_{in}$	$Address_{in}$
0 1 0	1 0 0	X X X	0	0	X X X X X	X	0	1	0xXXXXXXXX	0xXXXXXXXX

- $M[R6] \leftarrow 24$

AA	BA	DA	AS	BS	FS	DS	RW	MW	$Constant_{in}$	$Address_{in}$
1 1 0	X X X	X X X	0	1	X X X X X	X	0	1	0x00000018	0xXXXXXXXX

4.4 Programming the Datapath

The datapath can be used for performing various data manipulations including integer computations. For example, we can perform the summation of 10 numbers from 1 to 10:

$$sum = \sum_{i=1}^{10} i$$

The above computation for the datapath can be described with a sequence of RTL statements as shown in Figure 4.6 (a). When we provide the datapath with the sequence of control words that are produced by the RTL statement sequence, the computation completes 22 clock cycles later, and we will have the result in register R0.

Suppose that the 10 numbers are arbitrary integers and they are stored in the memory at consecutive addresses ranging 100 to 109. The sequence of RTL statement shown in Figure 4.6 (b) performs this computation. In this case, it takes 32 clock cycles to obtain the result in register R0.

$$
\begin{aligned}
&R0 \leftarrow 0\\
&R1 \leftarrow 0\\
&R1 \leftarrow R1 + 1\\
&R0 \leftarrow R0 + R1\\
&R1 \leftarrow R1 + 1\\
&R0 \leftarrow R0 + R1\\
&R1 \leftarrow R1 + 1\\
&R0 \leftarrow R0 + R1\\
&R1 \leftarrow R1 + 1\\
&R0 \leftarrow R0 + R1\\
&R1 \leftarrow R1 + 1\\
&R0 \leftarrow R0 + R1\\
&R1 \leftarrow R1 + 1\\
&R0 \leftarrow R0 + R1\\
&R1 \leftarrow R1 + 1\\
&R0 \leftarrow R0 + R1\\
&R1 \leftarrow R1 + 1\\
&R0 \leftarrow R0 + R1\\
&R1 \leftarrow R1 + 1\\
&R0 \leftarrow R0 + R1\\
&R1 \leftarrow R1 + 1\\
&R0 \leftarrow R0 + R1
\end{aligned}
$$

(a)

$$
\begin{aligned}
&R0 \leftarrow 0\\
&R2 \leftarrow 99\\
&R2 \leftarrow R2 + 1\\
&R1 \leftarrow M[R2]\\
&R0 \leftarrow R0 + R1\\
&R2 \leftarrow R2 + 1\\
&R1 \leftarrow M[R2]\\
&R0 \leftarrow R0 + R1\\
&R2 \leftarrow R2 + 1\\
&R1 \leftarrow M[R2]\\
&R0 \leftarrow R0 + R1\\
&R2 \leftarrow R2 + 1\\
&R1 \leftarrow M[R2]\\
&R0 \leftarrow R0 + R1\\
&R2 \leftarrow R2 + 1\\
&R1 \leftarrow M[R2]\\
&R0 \leftarrow R0 + R1\\
&R2 \leftarrow R2 + 1\\
&R1 \leftarrow M[R2]\\
&R0 \leftarrow R0 + R1\\
&R2 \leftarrow R2 + 1\\
&R1 \leftarrow M[R2]\\
&R0 \leftarrow R0 + R1\\
&R2 \leftarrow R2 + 1\\
&R1 \leftarrow M[R2]\\
&R0 \leftarrow R0 + R1\\
&R2 \leftarrow R2 + 1\\
&R1 \leftarrow M[R2]\\
&R0 \leftarrow R0 + R1\\
&R2 \leftarrow R2 + 1\\
&R1 \leftarrow M[R2]\\
&R0 \leftarrow R0 + R1
\end{aligned}
$$

(b)

Figure 4.6: The sequence RTL statements that compute the sum of 10 integers: (a) the RTL statement sequence that performs the summation of 10 integers ranging from 1 to 10; (b) the RTL statement sequence that performs the summation of 10 arbitrary integers stored in the memory at consecutive addresses ranging from 100 to 109.

```
1  sum = 0;
2  i = -1;
3  i = i + 1;
4  sum = sum + a[i];
5  i = i + 1;
6  sum = sum + a[i];
7  i = i + 1;
8  sum = sum + a[i];
9  i = i + 1;
10 sum = sum + a[i];
11 i = i + 1;
12 sum = sum + a[i];
13 i = i + 1;
14 sum = sum + a[i];
15 i = i + 1;
16 sum = sum + a[i];
17 i = i + 1;
18 sum = sum + a[i];
19 i = i + 1;
20 sum = sum + a[i];
21 i = i + 1;
22 sum = sum + a[i];
```

(a)

```
1     sum = 0;
2     i = -1;
3  L: i = i + 1;
4     sum = sum + a[i];
5     if (i < 10) goto L;
```

(b)

Figure 4.7: The C code that computes the sum of 10 integers: (a) the code that does not use a branching mechanism; (b) the code that does use branching mechanisms.

What if we would like to compute the sum of 10,000 arbitrary integers stored in the memory? Then, we need to have a sequence of 30,002 RTL statements. If you carefully take a look at the RTL code in Figure 4.6 (b), the following group of three consecutive RTL statements is repeated 10 times:

$$R2 \leftarrow R2 + 1$$
$$R1 \leftarrow M[R2]$$
$$R0 \leftarrow R0 + R1$$

It would be good if we could have some mechanism to avoid such repetitions. This problem can be solved by using a branching mechanism available in high-level languages, such as C and Java. A branching mechanism alters *control flow*. Control flow refers to the order in which the individual statements are executed.

The computation of adding 10 integers can be represented with the C code as shown in Figure 4.7 (a). A *C statement* is a C expression delimited by a semicolon (`;`). This code corresponds to the RTL code in Figure 4.6 (b). An *assignment* operation (denoted by `=`) in C assigns the value of the right-hand operand to the storage location named by the left-hand operand. A *variable* is

a named storage location that contains some value. The variable name typically references the stored value. However, if it is the left-operand of an assignment operation, it refers to the storage location. In the C code, `sum` and `i` are variables whose storage locations are the registers R0 and R2, respectively.

An *array* in C is a collection of elements that have the same *type* (e.g., integers) under the same name. Each element of an array with n elements can be treated as a variable and is referenced by the array name and an index number ranging from 0 to $n-1$. In the C code, `a` is an array containing 10 integers. The array reference `a[i]` refers to the value of the i^{th} element in the array `a`. A consecutive group of n words in the memory is allocated to an array with n elements.

C statement	RTL statements
`sum = 0;`	R0 ← 0
`i = -1;`	R2 ← 99
`i = i + 1;`	R2 ← R2 + 1
`sum = sum + a[i];`	R1 ← M[R2] R0 ← R0 + R1

Figure 4.8: The correspondence between the C statements in Figure 4.7 (a) and the RTL statements in Figure 4.6 (b).

Assuming that the storage locations of the variables *sum* and i are R0 and R2, respectively, and 100 is the address of the first element `a[0]` in the memory, the table in Figure 4.8 shows the correspondence between the C statements in Figure 4.7 (a) and the RTL statements in Figure 4.6 (b).

An `if` statement in C is a conditional branching mechanism, and it has an expression in parentheses and another statement in the following way:

```
if ( E ) S
```

where `E` is an expression and `S` is a statement. If the expression `E` is evaluated to a non-zero value, the statement `S` gets executed. It controls *conditional branching*. Depending on the condition `E`, it alters control flow and either the statement `S` or the statement immediately after the `if` is executed. Contrast to the `if` statement, a `goto` statement is a branching mechanism that alters control flow unconditionally. It has the following form:

```
goto L;
```

where `L` is a label in C. A *label* is a name that identifies a location in source code. The `goto` statement always alters control flow, and control is transferred to a labeled statement (the statement in line 3 of Figure 4.7 (b)) whose label matches the label that appears in the `goto` statement.

Using branching mechanisms available in C, the summation process of 10 arbitrary integers is succinctly described in Figure 4.7 (b). The value of each

element of `a` is continuously added to the variable `sum` continuously until the value of `i` reaches 10.

4.5 Instruction Set

A machine instruction is a group of bits that specifies not only the operation, but also the registers or memory words in which the operands are found and the result is stored. The *operation code* (*opcode*) of an instruction is a group of bits in the instruction that specifies an operation. N-bit opcode can represent 2^n different operations. Machine instructions for a computer have either all the same size or different sizes. The way how bits are organized in a machine instruction varies with the type of the instruction and the machine. The *Instruction set* of a computer is the complete collection of instructions for the computer. The *instruction set architecture* (*ISA*) of a computer is a thorough description of its instruction set. The term *microarchitecture* refers to the design techniques used to implement the instruction set.

The instruction set provided by a CPU must be rich enough to implement all functions that are known to be computable. It usually includes the following types of machine instructions:

- Data transfer instructions
  - Load and store instructions that move data to and from memory and CPU registers (load and store instructions).
  - Input and output instructions that moves data to and from CPU registers and I/O devices.
- Arithmetic, logic, and shift instructions.
  - Addition, subtraction, multiply, and division instructions.
  - Bitwise AND, OR, and NOT instructions.
  - Logical and arithmetic shift instructions.
  - Comparison instructions that compare two values.
- Control flow instructions
  - Unconditional branch instructions that jump to another location in the program to execute instructions there.
  - Conditional branch instructions that jump to another location in the program when a certain condition holds.

4.6 The Control Unit

The key idea of the von Neumann architecture is the stored program concept. Not only are all data values used in the program stored in memory, but also are

machine instructions in the program. These machine instructions are placed in adjacent locations and fetched by the CU one by one.

A *fetch-decode-execute cycle* is the basic operation cycle of the CPU. The CU fetches an instruction from memory, determines what operations the instruction requires (decodes it), and executes it by activating the necessary sequence of microoperations (i.e., control words) to provide timing and control signals to the datapath and memory. The fetched instruction is decoded by the *instruciton decoder* in the CU. The decoder converts the instruction to control signals to the datapath and to the CU itself to perform the operation specified by the instruction. This cycle is repeated until the computer is powered down.

To execute instructions in sequence, it is necessary to provide the address of the instruction to be executed. The CU contains a register called a *program counter* (*PC*) to specify the address of the next instruction to be executed. The PC is either automatically incremented or loaded with a new address to change the sequence of instructions after an instruction is fetched. *Branch instructions* modify the PC to skip over some sequence of instructions or to go back to repeat the previous instruction sequence. A branch instruction typically contains an offset. This offset is added to the current PC to go to the branch target address. There are two types of branch instructions: conditional branches and unconditional branches. A *conditional branch* instruction modifies the PC when a certain condition is true. The CU evaluates the condition by checking the status signals from the datapath. If the condition is true, the branch is taken. An *unconditional branch* instruction always modifies the PC. It always jumps to the branch target address.

Figure 4.9 shows a simple CPU to which a memory unit is attached. The CU in the CPU controls the 32-bit datapath and memory introduced in this chapter. In addition to a PC, the CU has two other registers called an *instruction register* (*IR*) and a *processor status register* (*PSR*). The IR contains the current instruction fetched from memory. The PSR is used by the CU to keep track of the CPU state. The status signals N, Z, C, and V from the ALU are connected to PSR[31], PSR[30], PSR[29], and PSR[28] as inputs, respectively. These four bits in the PSR are called status flags and named in the same way as the status signals from the ALU. The status flags in the PSR are set by a comparison instruction. The status signals from the ALU indicate the status of the result of the last ALU operation performed by the CPU:

- N (negative): the result of the last ALU operation is negative (MSB = 1)
- Z (zero): the result of the last ALU operation is zero
- C (carry): the result of the last ALU operation has a carry-out
- V (oVerflow): the result of the last ALU operation overflows

As shown in Figure 4.10 (a), assume that a program is stored in the memory at address 0x8000, and the data accessed by the program are stored from address 0xA000. Initially, the PC is loaded with the address 0x8000 of the first

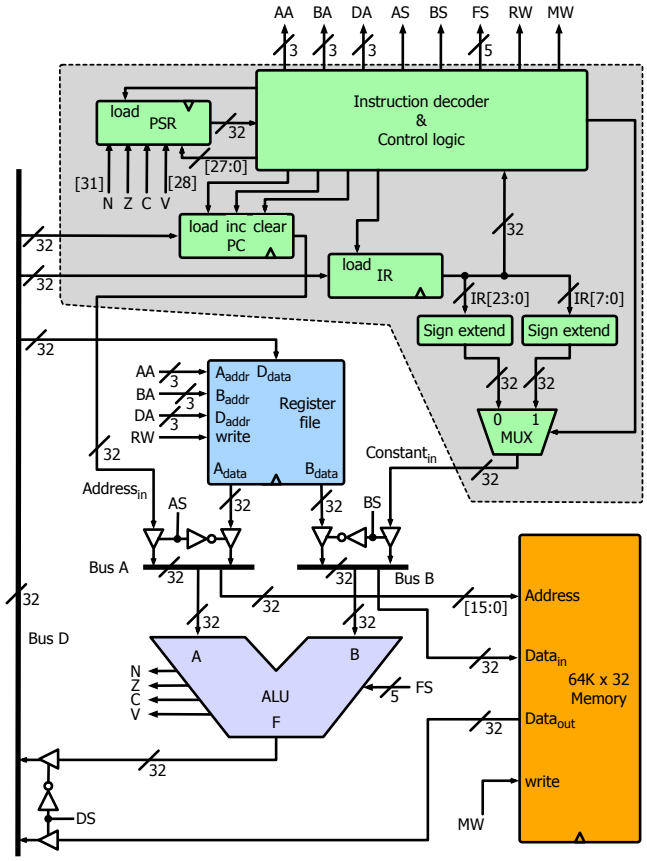


Figure 4.9: A simple CPU with a memory unit.

instruction of the program when power is applied to the system. Then the CPU repeats the fetch-decode-execute cycle.

In the fetch phase, the instruction whose address is specified by the PC is loaded into the IR. This process is the same for all instructions and described by the following RTL statement:

$$IR \leftarrow M[PC]$$

The instruction stored in the IR is decoded by the CU in the decode phase. For example, assume that the current instruction (i.e., the instruction that has been loaded into the IR) is an addition instruction that adds the contents of two registers R1 and R2 and stores the result to the register R3. The instruction has a 32-bit fixed length and four fields for the registers and opcode. Its instruction format is given in Figure 4.10 (b). Rd is a 4-bit field for the destination register R3. Rm and Rn are the two 4-bit fields for the two operand registers R1 and R2, respectively. Since the CPU has a total of 10 registers including the PC and

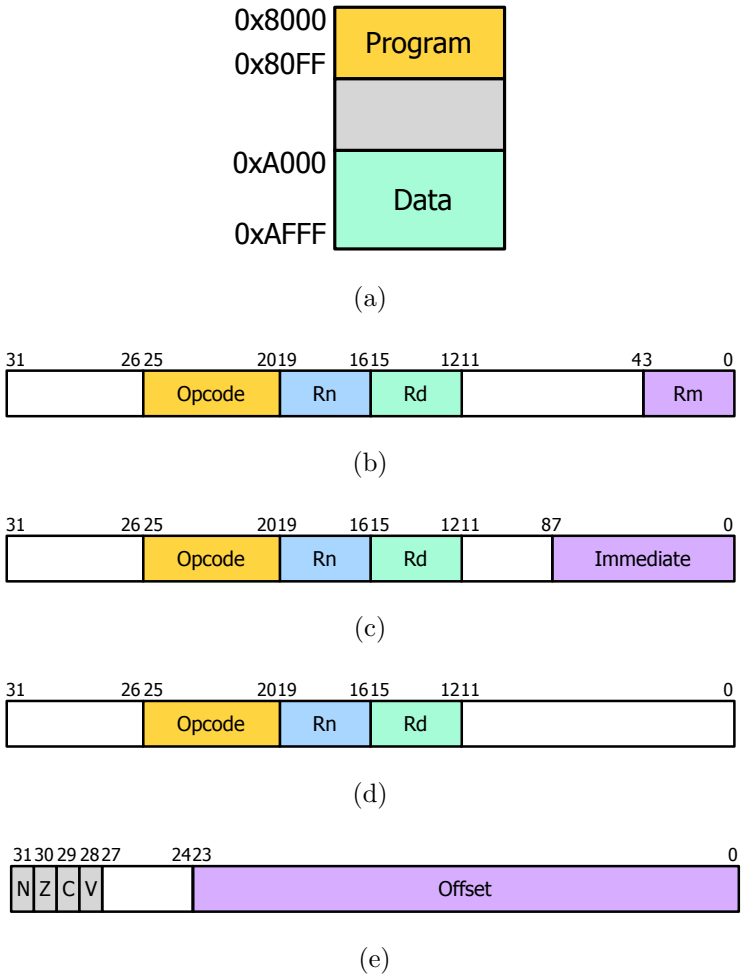


Figure 4.10: Memory map and instruction formats: (a) a memory map for the CPU in Figure 4.9; (b) the instruction format of an addition instruction; (c) the instruction format that contains an immediate constant; (d) the instruction format of a load or store instruction; (e) the instruction format of a branch instruction.

IR, at least 4 bits are required to specify a register. The instruction decoder in the CU reads the content of the IR, and the opcode and operands are being decoded. The CU generates appropriate control words to perform the addition operation in the execute phase:

$$R3 \leftarrow R1 + R2$$

Then, the CU increments the PC to fetch the next instruction:

$$PC \leftarrow PC + 1$$

The CU activates the input inc to the PC to perform this increment operation. Since the destination register of the instruction is not the PC, incrementing the PC can be performed in parallel with the addition operation:

$$R3 \leftarrow R1 + R2;\ PC \leftarrow PC + 1$$

In our RTL notation, RTL statements that are placed in the same line and separated with semicolons (;) are performed in parallel. In summary, the following microoperations are activated by the CU to execute the addition instruction:

$$IR \leftarrow M[PC]$$
$$R3 \leftarrow R1 + R2;\ PC \leftarrow PC + 1$$

It takes a total of two clock cycles to perform all the microoperations: one cycle for fetch and another one cycle for decode/execute and incrementing the PC.

Now, assume that the fetched instruction is an addition instruction that adds the content of register R1 and a constant 34, and stores the result to destination register R3. This type of instruction is called an *immediate instruction* because the instruction code contains the actual operand 34. The constant 34 is called an *immediate constant* or an *immediate*. The instruction format is given in Figure 4.10 (c). The immediate field is interpreted as an 8-bit signed binary number in the two's complement representation.

In the decode and execute phases, the immediate field IR[7:0] is sign-extended and connected to the 32-bit input $Constant_{in}$ of the datapath to execute the immediate instruction. The microoperations activated by the CU are as follows:

$$R3 \leftarrow R1 + Constant_{in};\ PC \leftarrow PC + 1$$

A load instruction makes a data word located in the memory to be transferred to a register. A store instruction transfers the content of a register to a memory location. The instruction format of load and store instructions is given in Figure 4.10 (d). The address of the memory location is contained Rn. Rd is the destination register when the instruction is a load instruction. It is the source register when the instruction is a store instruction. When the instruction is a load instruction, the CU activates the following microoperations:

$$Rd \leftarrow M[Rn];\ PC \leftarrow PC + 1$$

If the instruction is a store instruction, the following microoperations are activated by the CU in the execute phase:

$$M[Rn] \leftarrow Rd;\ PC \leftarrow PC + 1$$

As mentioned before, a branch instruction alters the content of the PC. The instruction format of branch instructions are given in Figure 4.10 (e). An unconditional branch instruction alters the PC when non-sequential execution is desired. The specified branch target address is computed by adding the value stored in the *offset* field to the PC. The offset field contains a 24-bit signed

binary number in the two's complement representation. After sign-extending the offset value IR[23:0], the CU put the result to the input $Constant_{in}$ of the datapath. The RTL statement for the unconditional branch instructions is:

$$PC \leftarrow PC + Constant_{in}$$

In the next clock cycle, the instruction located at the branch target address will be fetched. If the destination register of an instruction is the PC, the instruction is treated as an unconditional branch instruction.

When the fetched instruction is a conditional branch instruction, the N, Z, C, and V flags in Figure 4.10 (e) are used to make the decision to take the branch or not. The N, Z, C, and V flags in the instruction (IR[31:28]) are compared with the four status flags in the PSR by the CU. If they are the same, the branch is taken, and the following microoperation is activated:

$$PC \leftarrow PC + Constant_{in}$$

Otherwise, the branch is not taken and the next instruction to be fetched is the instruction that follows the current instruction in the memory:

$$PC \leftarrow PC + 1$$

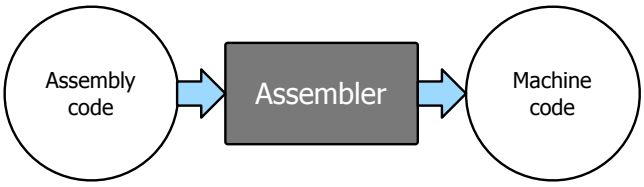


Figure 4.11: An assembler translates the assembly program into the machine code.

4.7 Assembly Language

Humans almost never write programs directly in machine code because it is very difficult to understand and write a program in patterns of 0 and 1. It may also be very much error prone because the opcode for every instruction is looked up or remembered to write a program. Instead, we use an *assembly language*. It is a low-level language and relatively easy to write a program compared to the machine language.

Assembly language uses *mnemonics* to symbolically represent the opcode of machine instructions. In addition, it uses symbolic names for locations in the program (labels), variables, and constants. An assembly language instruction usually consists of a mnemonic followed by a list of operands. Since a machine instruction has an equivalent assembly instruction, translation from an assembly

MOV R2, #-25	Move -25 to the destination register R2. An 8-bit integer constant is prefixed with a character #.
ADD R3, R1, R2	Add the value of register R2 to the value of register R1, and stores the result in the destination register R3.
LDR R1, [R3]	Make data located at the address contained in Rn to be loaded into the destination register Rd.
STR R0, [R2]	Make data from the register R0 to be stored to the memory location with the address contained in R2.
CMP R1, R2	Compare the value of register R2 with the value of register R1, and sets up the status flags N, Z, C, and V in the PSR. If the contents are equal, Z is set to 1, otherwise it is set to 0.
B L	Causes a jump to the target address labeled with "L:".
BNE L	Causes a jump to the target address if the Z flag in the PSR is not zero. NE stands for Not Equal.

Figure 4.12: Some examples of assembly language instructions for our computer.

instruction to the corresponding machine instruction is usually straightforward. However, there are some meta-instructions, such as *assembler directives* and *pseudo-instructions*, which may not be translated into a single machine instruction in a straightforward manner. An assembler directive is a command to the assembler that tells the assembler something to do in the assembly process. A pseudo-instruction does not actually exist in the machine instruction set. It is just an easy way of representing a group of machine instructions (possibly a single machine instruction), which makes the code easy to understand. An assembly program may also contains comments that facilitate understanding of the program.

A program written in assembly language is translated into the target computer's machine code by a utility program called an *assembler*. The assembler generates an object file by translating assembly instructions into machine instructions and by resolving symbolic names for memory locations and constants. A label is a symbolic name and specifies a location (address) in the program. When a branch instruction whose address a uses a label L as its target, the target address is computed by adding an offset $L-a$ to the address a of the branch instruction. The offset is encoded into the offset field of the branch instruction by the assembler.

Figure 4.12 shows some examples of assembly language instructions for our simple computer. The program adding 10 arbitrary numbers stored in the memory can be written in the assembly language. The assembly code is shown in

(a)

```
MOV  R0, #0
MOV  R2, #99
ADD  R2, R2, #1
LDR  R1, [R2]
ADD  R0, R0, R1
ADD  R2, R2, #1
LDR  R1, [R2]
ADD  R0, R0, R1
ADD  R2, R2, #1
LDR  R1, [R2]
ADD  R0, R0, R1
ADD  R2, R2, #1
LDR  R1, [R2]
ADD  R0, R0, R1
ADD  R2, R2, #1
LDR  R1, [R2]
ADD  R0, R0, R1
ADD  R2, R2, #1
LDR  R1, [R2]
ADD  R0, R0, R1
ADD  R2, R2, #1
LDR  R1, [R2]
ADD  R0, R0, R1
ADD  R2, R2, #1
LDR  R1, [R2]
ADD  R0, R0, R1
ADD  R2, R2, #1
LDR  R1, [R2]
ADD  R0, R0, R1
ADD  R2, R2, #1
LDR  R1, [R2]
ADD  R0, R0, R1
```

(b)

```
     MOV  R0, #0
     MOV  R2, #99
L :  ADD  R2, R2, #1
     LDR  R1, [R2]
     ADD  R0, R0, R1
     CMP  R2, #109
     BNE  L
```

Figure 4.13: The assembly code that computes the sum of 10 arbitrary integers: (a) the sequence of assembly language instructions that performs the summation of 10 arbitrary integers stored in the memory at consecutive addresses ranging from 100 to 109; (b) the code that performs the same computation with a comparison instruction and a branch instruction.

Figure 4.13. Figure 4.13 (a) is the straight line code, and Figure 4.13 (b) uses a branch instruction. The register R2 is loaded with the address of a number stored in the memory. R2 is incremented from 99 to 109 each time a new value is loaded to R1. The comparison instruction compares the content of R2 with 109. If they are not equal, the branch instruction makes control flow to be transferred to the instruction labeled with L.

Instruction	T_0	T_1
MOV R0, #0	$IR \leftarrow M[PC]$	$R0 \leftarrow 0$; $PC \leftarrow PC + 1$
MOV R2, #99	$IR \leftarrow M[PC]$	$R2 \leftarrow 99$ ($Constant_{in}$); $PC \leftarrow PC + 1$
ADD R2, R2, #1	$IR \leftarrow M[PC]$	$R2 \leftarrow R2 + 1$; $PC \leftarrow PC + 1$
LDR R1, [R2]	$IR \leftarrow M[PC]$	$R1 \leftarrow M[R2]$; $PC \leftarrow PC + 1$
ADD R0, R0, R1	$IR \leftarrow M[PC]$	$R0 \leftarrow R0 + R1$; $PC \leftarrow PC + 1$
CMP R2, #109	$IR \leftarrow M[PC]$	$R0 \leftarrow R2 - 109$ ($Constant_{in}$); $PSR[31:28] \leftarrow NZCV$; $PC \leftarrow PC + 1$
BNE L	$IR \leftarrow M[PC]$	$PC \leftarrow PC + Constant_{in}$ if PSR[31] = 0 $PC \leftarrow PC + 1$ otherwise

Figure 4.14: The microoperations that must be performed for each instruction used in Figure 4.13 (b).

Figure 4.14 lists the microoperations that must be performed for each instruction used in Figure 4.13 (b). The comparison instruction is implemented with a microoperation that performs subtraction. According to the result of the subtraction, the status flags in the PSR are set. Note that each microoperation takes a single cycle in the simple computer under discussion. The total time taken to fetch, decode, and execute an instruction is called the *instruction cycle time*. The microoperation to fetch an instruction is the same for all instructions. The CU generates appropriate control words in each clock cycle of the instruction cycle time depending on the opcode identified in the decode phase and the current status of the CPU.

4.8 Input and Output

Without input and output (I/O) devices, the computer based on the von Neumann architecture cannot receive information from outside and transmit the

result in a desired form. I/O devices attached to a computer is also called *peripherals*. Peripherals include keyboards, mice, display units, speakers, printers, hard disk drives, optical disk drives, solid state disk drives, network interface cards, etc. Peripherals that communicate with people typically transfer alphanumeric information to and from the CPU. The standard binary code for the alphanumeric information is ASCII.

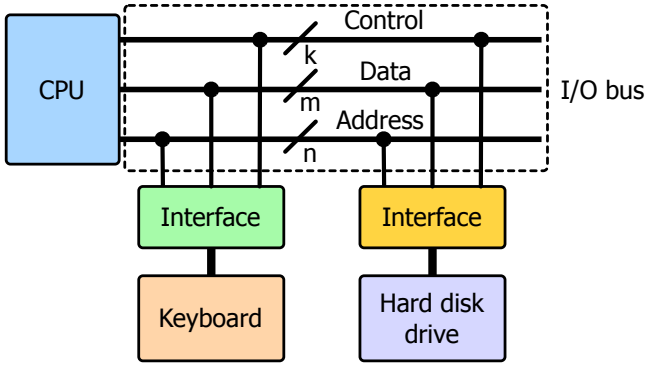


Figure 4.15: The I/O bus.

Peripherals are usually connected to the CPU through an I/O bus as shown in Figure 4.15. The I/O bus consists of data lines, address lines, and control lines. To connect a peripheral to the I/O bus, an *interface* is required to resolve differences between the peripheral and CPU. The interface contains an address decoder, a control unit, and registers for the device.

Each interface has a distinct address. To communicate with a specific peripheral device, the CPU places the address of the device on the address lines, which are continuously monitored by the interface for each device. If the interface for a device detects its own address on the bus, a communication link is established through the I/O bus between the device and the CPU. All other devices are disabled for the bus. The CPU and the selected device communicate with each other through the control and data lines.

Memory-mapped I/O

Memory-mapped I/O makes all I/O devices look exactly the same to the CPU. Each I/O device is allocated to an exclusive portion of the CPU's *address space*. When a CPU has n-bit addresses, its address space is the set of 2^n possible addresses. The memory and registers of the I/O device have the addresses in the exclusively allocated portion of the CPU's address space. These addresses must not be available for the physical main memory. Each of these addresses is called an *I/O port*. When the CPU accesses a location with an address using a load or store instruction, the location may be a register or a memory location of an I/O device. Since normal load or store instructions are used to communicate with I/O devices, the instruction set of the CPU does not need

to include special I/O instructions. To enable memory-mapped I/O, each I/O device needs to provide a hardware interface similar to that of memory, and is required to define an interaction contract (protocol).

The exclusive portion of the address space allocated to an I/O device continuously reflects the physical state of the device. For example, pressing a key on the keyboard makes a certain value (e.g., ASCII code of the key) to be written in the area allocated to the keyboard. Whenever a bit is changed in the area allocated to a physical screen, the associated pixel is drawn on the screen.