# Answers to Exercises

## Chapter 1

**1.1 :**

The internal mechanism of a digital computer can simply be denoted as a black box which is shown in Figure 1.1. From Figure 1.1, a digital computer can be thought of as a data processor. A digital program also can be thought of as a set of instructions written in a digital computer language that indicates the data processor what to do with the input data. The output data depend on the combination of two factors: the input data and the digital program. With the same digital program, you can produce different outputs if you change the input. Similarly, with the same input data, you can generate different outputs if you change the digital program.

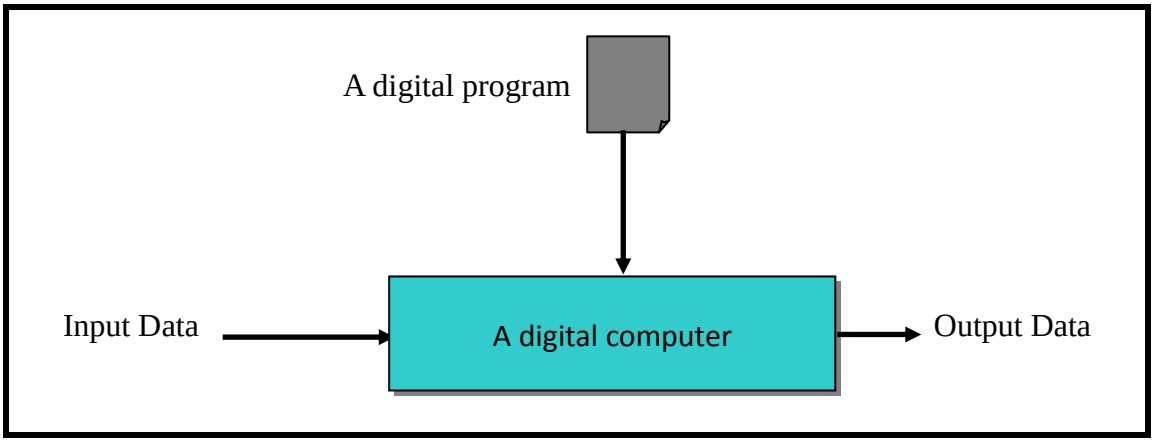


Figure 1.1: Computational model of a digital computer.

**1.2 :**

The internal mechanism of bio-molecular computer can simply be defined as another black box which is shown in Figure 1.2, where some robotics or electronic computing is used to carry out automatically the majority of the operations with the test tubes without the intervention of the user. From Figure 1.2, input data can be encoded in test tubes. Each encoded data in test tubes can be thought of a data processor. A bio-molecular program also can be thought of as a set of biological operations written in a high-level natural language that tells each data processor what to do. The output data also are based on the combination of two factors: the input data and the bio-molecular program. With the same bio-molecular program, you can produce different outputs if you change the input. Similarly, with the same input data, you can generate different outputs if you change the bio-molecular program. Finally, if the input data and the bio-molecular program remain the same, the output should be

the same.

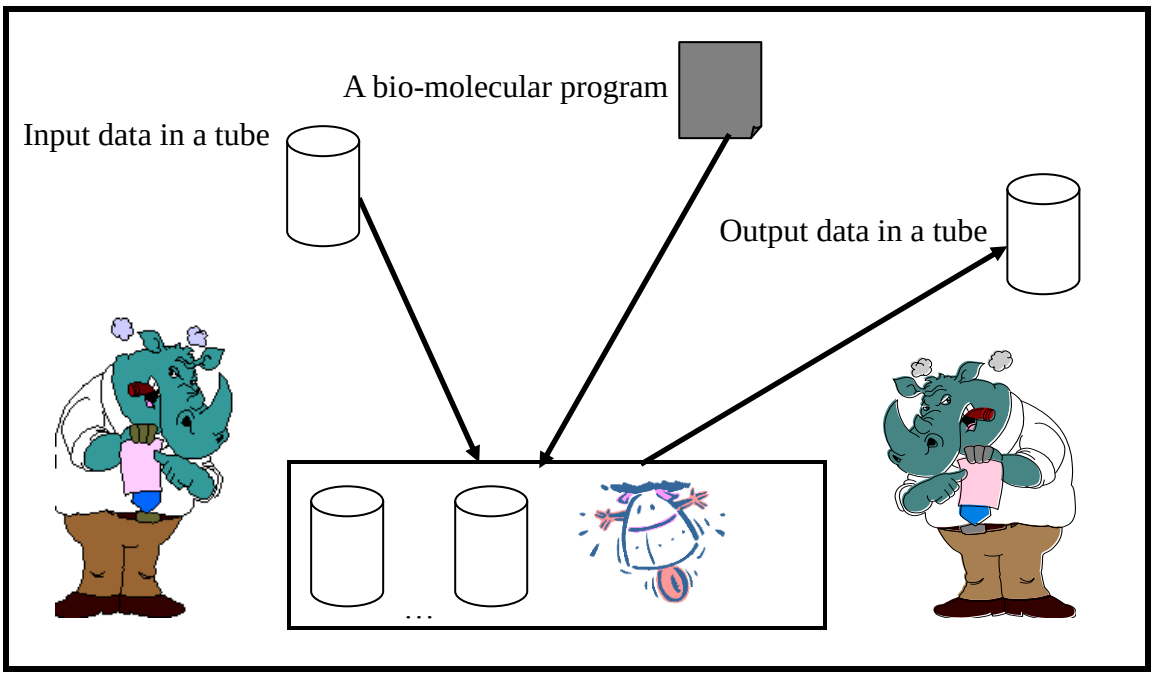


Figure 1.2: Some robotics or electronic computing in an advanced computational model of bio-molecular computer carries out automatically the majority of the operations with the test tubes without the intervention of the user.

**1.3 :**

A digital program in a digital computer can be thought of as a set of instructions written in a digital computer language that indicates the data processor what to do with the input data. A bio-molecular program in bio-molecular computer can be thought of as a set of biological operations written in a high-level natural language that tells each data processor what to do.

**1.4 :**

Memory is the main storage area in the inside of a digital computer. It is used to store data and digital programs during processing. This implies that both the data and programs should have the same format because they are stored in memory. They are, in fact, stored as binary patterns (a sequence of 0s and 1s) in memory.

Tubes in bio-molecular computer are devices in which input data and output data are stored and each biological operation is completed. The function of tubes in bio-molecular computer are actually the same that of memory and input/output devices in a digital computer.

**1.5 :**

An input/output subsystem in a digital computer is an auxiliary storage area and is also the communication between the digital computer and the outside world. Inputs are data received by a digital computer, and outputs are data sent from it. For instance, a keyboard, or a mouse is an input device for a computer, while a monitor or a printer is an output device for the computer. A hard disk or a tape is simultaneously input and output devices.

Input data that are encoded are stored in tubes in bio-molecular computer. Output data that are produced by a bio-molecular program are also stored in tubes. Tubes are the only storage device in bio-molecular computer.

**1.6 :**

A bit is the smallest unit of data that can be stored in a digital computer and bio-molecular computer; it is either 0 or 1. To a digital computer, bit 0 is encoded by the off state of a switch and bit 1 is encoded by the on state of the switch. For bio-molecular computer, different sequences of bio-molecules encode, respectively, bit 0 and bit 1.

**1.7 :**

The so-called von Neumann architecture is a model for a computing machine that uses a single storage structure to hold both the set of instructions on how to perform the computation and the data required or generated by the computation. A digital computer system of the von Neumann architecture is shown in Figure 1.3.

From Figure 1.3, the input subsystem accepts input data and the digital program from outside the digital computer and the output subsystem sends the result of processing to the outside. Memory is the main storage area in the inside of the digital computer system. The arithmetic logic unit is the core of the digital computer system and is applied to perform calculation and logical operations. The control unit is employed to control the operations of the memory, ALU, and the input/output subsystem.

A digital program in the von Neumann architecture is made of a finite number of instructions. In the architecture, the control unit fetches one instruction from memory, interprets it, and then excites it. In other words, the instructions in the digital program are executed one after another. Of course, one instruction may request the control unit to jump to some previous or following one instruction, but this does not mean that the instructions are not executed sequentially.

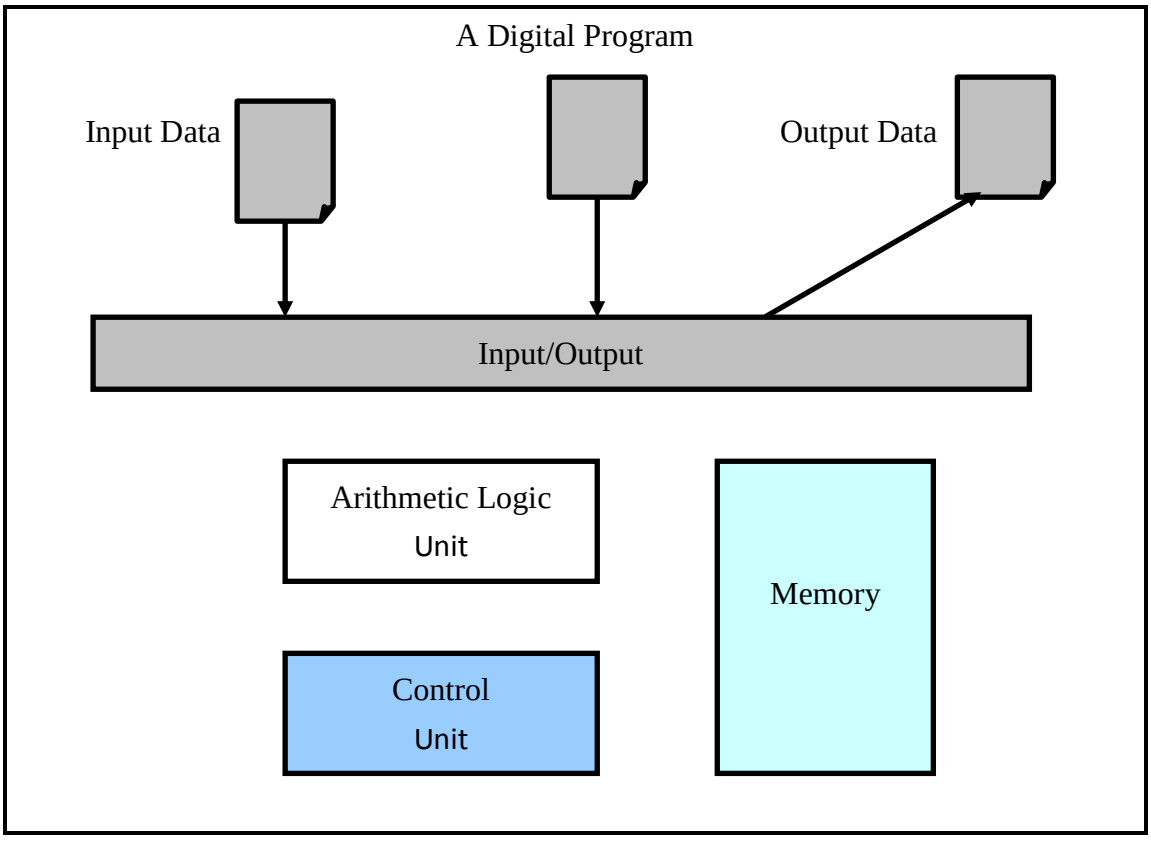


Figure 1.3: A digital computer system of the von Neumann architecture has four subsystems.

**1.8:**

In bio-molecular computer, data also are represented as binary patterns (a sequence of 0s and 1s). Those binary patterns are encoded by sequences of bio-molecules and are stored in a tube. This is to say that a tube is the only storage area in bio-molecular computer and is also the memory and the input/output subsystem of the von Neumann architecture. Bio-molecular programs are made of a set of bio-molecular operations and are used to perform calculation and logical operations. So, bio-molecular programs can be regarded as the arithmetic logic unit of the von Neumann architecture. A robot is used to automatically control the operations of a tube (the memory and the input/output subsystem) and bio-molecular programs (the ALU). This implies that the robot can be regarded as the control unit of the von Neumann architecture.

In Figure 1.4, bio-molecular computer of the von Neumann architecture is shown. From Figure 1.4, a robot fetches one bio-molecular operation from a bio-molecular program (the ALU), and then carries out the bio-molecular operation for those data stored in the tube (the memory). In other words, the bio-molecular operations are

executed one after another. Certainly, one bio-molecular operation perhaps requests the robot to perform some previous or following bio-molecular operations.

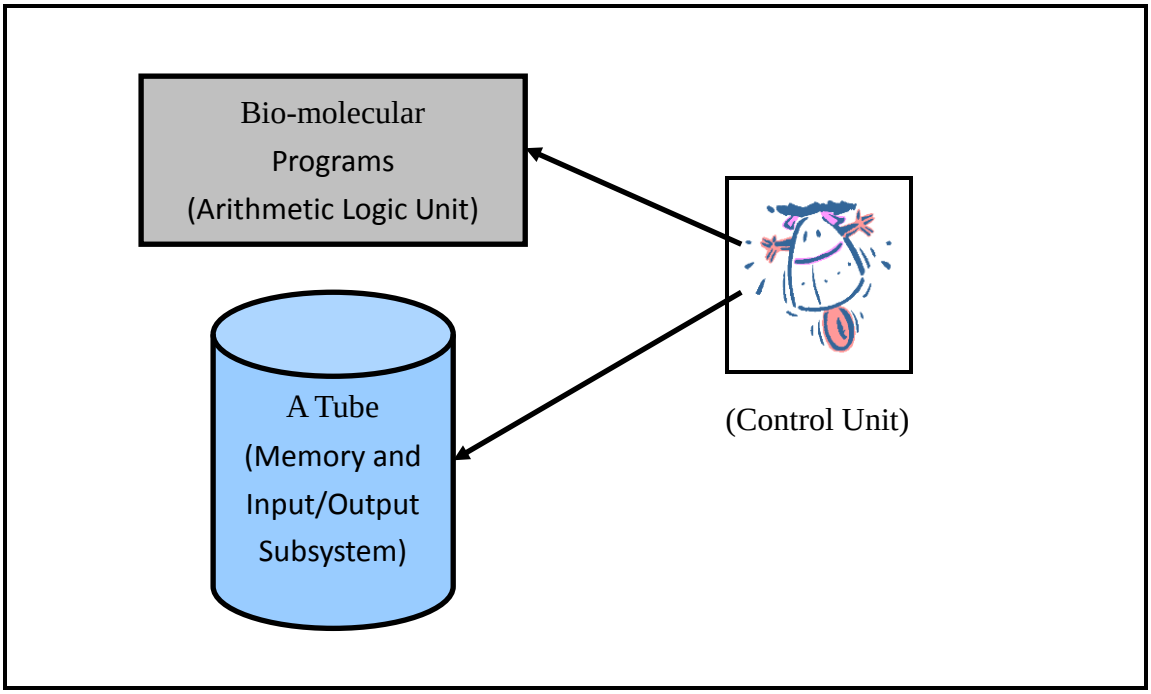


Figure 1.4: The bio-molecular computer of the von Neumann architecture.

## Chapter 2

**2.1:**

(a) In a digital computer, the on state and the off state of a switch subsequently encode the values 1 and 0 of a bit.

(b) In bio-molecular computer, different sequences of bio-molecules encode the values 1 and 0 of a bit. This indicates that two different sequences of bio-molecules can be regarded as the on state and the off state of a switch in a digital computer, where the on state is regarded as 1 and the off state is regarded as 0.

**2.2:**

(a) In a digital computer, a bit pattern that is a string of bits is also a combination of 0s and 1s. The values 1 and 0 of each bit in a bit pattern are encoded by the on state and the off state of a switch.

(b) In bio-molecular computer, a bit pattern is a combination of 0s and 1s. If a bit pattern that is made of $n$ bits can be stored in a tube in bio-molecular computer, then $(2 \times n)$ different sequences of bio-molecules are needed.

**2.3:** We would like to give many thanks to Louie Lu who wrote the following **C** programs.

```
#include <stdio.h>
#define MAX_LEN 100
char * hexadecimal-number-to-its-corresponding-binary-number(char *s)
{     int i, c, dec = 0, dec_tmp;
      static char b[MAX_LEN];
      while ((c = *s++) != NULL)
      {         dec = dec * 16 + (isdigit(c) ? c - '0' : c - 'A' + 10);       }
      i = 0;
      dec_tmp = dec;
      while (dec_tmp)
      {    i++;
           dec_tmp /= 2;
      }
```

```
        while (dec)
        {       b[--i] = dec % 2 + '0';
                dec /= 2;
          }
         return b;
}

int main()
{ char s[MAX_LEN];
   printf(“Please input a hexadecimal number: “);
   scanf("%s", s);
   printf("The corresponding binary number is: %s",
            hexadecimal-number-to-its-corresponding-binary-number (s));
}
```

**2.4:** We would like to give many thanks to Louie Lu who wrote the following **C** programs.

```
#include <stdio.h>
#define MAX_LEN 100
char * binary-number-to-its-corresponding-hexadecimal-number(char *s)
{       int c, j, i=0, dec=0;
        static char h[MAX_LEN];
        static char hex[] = {"0123456789ABCDEF"};
        while ((c = *s++) != NULL)
        {       dec <<= 1;
                if (c == '1') dec += 1;
          }
        while (dec)
       {        h[i++] = hex[dec % 16];
                dec /= 16;
        }
        for (j=0; j < i / 2; ++j)
        {       c = h[j];
                h[j] = h[i - j - 1];
                h[i - j - 1] = c;
        }
        return h;
```

```
}

int main()
{     char s[MAX_LEN];
      printf(“Please input a binary number: “);
      scanf("%s", s);
      printf("The corresponding hexadecimal number is : %s",
            binary-number-to-its-corresponding-hexadecimal-number(s));
}
```

**2.5:** We would like to give many thanks to Shang-De Jlang who wrote the following **C** programs.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{     char oct[1000];
      long int i = 0;
      printf("Please input an octal number: ");
      scanf("%s",&oct);
      while(oct[i])
      {
            switch(oct[i])
           {        case '0': printf("000"); break;
                    case '1': printf("001"); break;
                    case '2': printf("010"); break;
                    case '3': printf("011"); break;
                    case '4': printf("100"); break;
                    case '5': printf("101"); break;
                    case '6': printf("110"); break;
                    case '7': printf("111"); break;
                    default: printf("\n An invalid octal number %c",oct[i]); return 0;
            }
            i++;
       }
      system("pause");
      return 0;
}
```

**2.6:** We would like to give many thanks to Shang-De Jlang who wrote the following C programs.

```
#include <stdio.h>
#include <stdlib.h>
long int binarynumber, octalnumber = 0, j = 1, remainder;
int main()
{     printf("Please input a binary number: ");
      scanf("%ld",&binarynumber);
      while(binarynumber != 0)
      {    remainder = binarynumber % 10;
           octalnumber = octalnumber + remainder * j;
           j = j * 2;
           binarynumber = binarynumber / 10;
       }
      printf("The corresponding octal value is: %lo \n", octalnumber);
      system("pause");
      return 0;
}
```

## Chapter 3

### 3.1:

It is assumed that a binary number of a bit, $r$, is used to encode the first input to the **NOT** operation of two bits as shown in Table 3.12.1. Also it is supposed that a binary number of a bit $\bar{r}$ is employed to encode the first output to the **NOT** operation. For the sake of convenience, it is assumed that $r^1$ denotes the fact that the value of $r$ is 1 and $r^0$ denotes the fact that the value of $r$ is 0. Similarly, it is supposed that $\bar{r}^1$ denotes the fact that the value of $\bar{r}$ is 1 and $\bar{r}^0$ denotes the fact that the value of $\bar{r}$ is 0. The following algorithm is proposed to implement the **NOT** operation of two bits as shown in Table 3.12.1. Tubes $T_0$, $T_1$, and $T_2$ are subsequently the first, second and third parameters, and are set to empty tubes.

**Procedure NOT**($T_0$, $T_1$, $T_2$)
(1) Append-head($T_1$, $r^1$).
(2) Append-head($T_2$, $r^0$).
(3) Append-head($T_1$, $\bar{r}^0$).
(4) Append-head($T_2$, $\bar{r}^1$).
(5) $T_0 = \cup(T_1, T_2)$.
**EndProcedure**

**Proof of Correction:**

The algorithm, **NOT**($T_0$, $T_1$, $T_2$), is implemented by means of the *append-head* and *merge* operations. Step (1) and Step (2) use the *append-head* operations to append $r^1$ and $r^0$ onto tubes $T_1$ and $T_2$. This is to say that $T_1$ includes the first input that have $r$ = 1 and $T_2$ consists of the first input that have $r$ = 0, and two different inputs for the **NOT** operation of two bits as shown in Table 3.12.1 were poured into tubes $T_1$ through $T_2$, respectively. Next, Step (3) and Step (4) also use the *append-head* operations to append $\bar{r}^0$ and $\bar{r}^1$ onto tubes $T_1$ and $T_2$. This indicates that two different outputs to the **NOT** operation of two bits as shown in Table 3.12.1 are appended into tubes $T_1$ through $T_2$. Finally, on the first execution of Step (5), it applies the *merge* operation to pour tubes $T_1$ through $T_2$ into tube $T_0$. Tube $T_0$ contains the

result implementing the **NOT** operation of two bits as shown in Table 3.12.1. ■

**3.2:**

It is assumed that two binary numbers of a bit, $R_1$ and $R_2$, are used to, respectively, encode the first input and the second input for the **AND** operation of two bits as shown in Table 3.12.2. Also it is supposed that a binary number of a bit, $C_1$, is used to encode the output for the **AND** operation. For the sake of convenience, it is assumed that for $1 \le k \le 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is supposed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is offered to implement the **AND** operation of two bits as shown in Table 3.12.2. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes. The eighth parameter, $R_1$, is used to encode the first input to the **AND** operation of two bits and the ninth parameter, $R_2$, is applied to encode the second input to the **AND** operation of two bits.

**Procedure AND**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$, $R_1$, $R_2$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 2
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, $R_k^1$).
    (4c) Append-head($T_2$, $R_k^0$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
(5) $T_1 = +(T_0, R_1^1)$ and $T_2 = -(T_0, R_1^1)$.
(6) $T_3 = +(T_1, R_2^1)$ and $T_4 = -(T_1, R_2^1)$.
(7) $T_5 = +(T_2, R_2^1)$ and $T_6 = -(T_2, R_2^1)$.
(8) Append-head($T_3$, $C_1^1$).
(9) Append-head($T_4$, $C_1^0$).
(10) Append-head($T_5$, $C_1^0$).
(11) Append-head($T_6$, $C_1^0$).
(12) $T_0 = \cup(T_3, T_4, T_5, T_6)$.
**EndProcedure**

**Proof of Correction:**

The algorithm, **AND**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$, $R_1$, $R_2$), is implemented by means of the *extract*, *amplify*, *append-head* and *merge* operations. Steps (1) through (4d) use the *amplify*, *append-head* and *merge* operations to construct four different inputs to the **AND** operation of two bits as shown in Table 3.12.2. Next, Steps (5) through (7) use the *extract* operations to form some different test tubes including different inputs ($T_1$ to $T_6$). $T_1$ includes all of the inputs that have $R_1 = 1$, $T_2$ includes all of the inputs that have $R_1 = 0$, $T_3$ includes that input that has $R_1 = 1$ and $R_2 = 1$, $T_4$ includes that input that has $R_1 = 1$ and $R_2 = 0$, $T_5$ includes that input that has $R_1 = 0$ and $R_2 = 1$, and finally, $T_6$ includes that input that has $R_1 = 0$ and $R_2 = 0$. This indicates that four different inputs for the **AND** operation of two bits as shown in Table 3.12.2 were poured into tubes $T_3$ through $T_6$, respectively. Next, Steps (8) through (11) use the *append-head* operations to append $C_1^1$ and $C_1^0$ onto the head of every input in the corresponding tubes. This is to say that four different outputs to the **AND** operation of two bits as shown in Table 3.12.2 are appended into tubes $T_3$ through $T_6$. Finally, the execution of Step (12) applies the *merge* operation to pour tubes $T_3$ through $T_6$ into tube $T_0$. Tube $T_0$ contains the result implementing the **AND** operation of two bits as shown in Table 3.12.2. ■

**3.3:**

It is assumed that two binary numbers of a bit, $R_1$ and $R_2$, are used to, respectively, encode the first input and the second input for the **OR** operation of two bits as shown in Table 3.12.3. Also it is supposed that a binary number of a bit, $C_1$, is used to encode the output for the **OR** operation. For the sake of convenience, it is supposed that for $1 \le k \le 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is supposed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is proposed to implement the **OR** operation of two bits as shown in Table 3.12.3. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes. The eighth parameter, $R_1$, is employed to encode the first input to the **OR** operation of two bits and the ninth parameter, $R_2$, is used to encode the second input to the **OR** operation of two bits.

**Procedure OR**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$, $R_1$, $R_2$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 2

(4a) Amplify($T_0$, $T_1$, $T_2$).

(4b) Append-head($T_1$, $R_k^1$).

(4c) Append-head($T_2$, $R_k^0$).

(4d) $T_0 = \cup(T_1, T_2)$.

**EndFor**

(5) $T_1 = +(T_0, R_1^1)$ and $T_2 = -(T_0, R_1^1)$.

(6) $T_3 = +(T_1, R_2^1)$ and $T_4 = -(T_1, R_2^1)$.

(7) $T_5 = +(T_2, R_2^1)$ and $T_6 = -(T_2, R_2^1)$.

(8) Append-head($T_3$, $C_1^1$).

(9) Append-head($T_4$, $C_1^1$).

(10) Append-head($T_5$, $C_1^1$).

(11) Append-head($T_6$, $C_1^0$).

(12) $T_0 = \cup(T_3, T_4, T_5, T_6)$.

**EndProcedure**

**Proof of Correction**:

The algorithm, **OR**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$, $R_1$, $R_2$), is implemented by means of the *extract*, *append-head*, *amplify* and *merge* operations. Steps (1) through (4d) use the *amplify*, *append-head* and *merge* operations to construct four different inputs to the **OR** operation of two bits as shown in Table 3.12.3. Next, Steps (5) through (7) use the *extract* operations to form some different test tubes including different inputs ($T_1$ to $T_6$). $T_1$ includes all of the inputs that have $R_1 = 1$, $T_2$ includes all of the inputs that have $R_1 = 0$, $T_3$ includes that input that has $R_1 = 1$ and $R_2 = 1$, $T_4$ includes that input that has $R_1 = 1$ and $R_2 = 0$, $T_5$ includes that input that has $R_1 = 0$ and $R_2 = 1$, and finally, $T_6$ includes that input that has $R_1 = 0$ and $R_2 = 0$. This implies that four different inputs for the **OR** operation of two bits as shown in Table 3.12.3 were poured into tubes $T_3$ through $T_6$, respectively. Next, Steps (8) through (11) use the *append-head* operations to append $C_1^1$ and $C_1^0$ onto the head of every input in the corresponding tubes. This is to say that four different outputs to the **OR** operation of two bits as shown in Table 3.12.3 are appended into tubes $T_3$ through $T_6$. Finally, the execution of Step (12) applies the *merge* operation to pour tubes $T_3$ through $T_6$ into tube $T_0$. Tube $T_0$ contains the result implementing the **OR** operation of two bits as shown in Table 3.12.3. ■

**3.4:**

It is supposed that a binary number of a bit, *r*, is applied to encode the first input to the **BUFFER** operation of two bits as shown in Table 3.12.4. Also it is assumed that a binary number of a bit *b* is used to encode the first output to the **BUFFER**

operation. For the sake of convenience, it is supposed that $r^1$ denotes the fact that the value of $r$ is 1 and $r^0$ denotes the fact that the value of $r$ is 0. Similarly, it is also assumed that $b^1$ denotes the fact that the value of $b$ is 1 and $b^0$ denotes the fact that the value of $b$ is 0. The following algorithm is offered to implement the **BUFFER** operation of two bits as shown in Table 3.12.4. Tubes $T_0$, $T_1$, and $T_2$ are subsequently the first, second and third parameters, and are set to empty tubes.

**Procedure BUFFER**($T_0$, $T_1$, $T_2$)
(1) Append-head($T_1$, $r^1$).
(2) Append-head($T_2$, $r^0$).
(3) Append-head($T_1$, $b^1$).
(4) Append-head($T_2$, $b^0$).
(5) $T_0 = \cup(T_1, T_2)$.
**EndProcedure**

**Proof of Correction:**

The algorithm, **BUFFER**($T_0$, $T_1$, $T_2$), is implemented by means of the *append-head* and *merge* operations. Step (1) and Step (2) apply the *append-head* operations to append $r^1$ and $r^0$ onto tubes $T_1$ and $T_2$. This indicates that $T_1$ contains the first input that have $r = 1$ and $T_2$ includes the first input that have $r = 0$, and two different inputs for the **BUFFER** operation of two bits as shown in Table 3.12.4 were poured into tubes $T_1$ through $T_2$, respectively. Next, Step (3) and Step (4) also employ the *append-head* operations to append $b^1$ and $b^0$ onto tubes $T_1$ and $T_2$. This implies that two different outputs to the **BUFFER** operation of two bits as shown in Table 3.12.4 are appended into tubes $T_1$ through $T_2$. Finally, on the first execution of Step (5), it uses the *merge* operation to pour tubes $T_1$ through $T_2$ into tube $T_0$. Tube $T_0$ consists of the result implementing the **BUFFER** operation of two bits as shown in Table 3.12.4. ■

**3.5:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are applied to encode the first input and the second input for the **NAND** operation of two bits as shown in Table 3.12.5. Also it is assumed that a binary number of a bit, $C_1$, is employed to encode the output for the N**AND** operation. For the sake of convenience, it is supposed that for $1 \le k \le 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also assumed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The

following algorithm is presented to implement the **NAND** operation of two bits as shown in Table 3.12.5. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes.

**Procedure NAND**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 2
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, $R_k^1$).
    (4c) Append-head($T_2$, $R_k^0$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
(5) $T_1 = +(T_0, R_1^1)$ and $T_2 = -(T_0, R_1^1)$.
(6) $T_3 = +(T_1, R_2^1)$ and $T_4 = -(T_1, R_2^1)$.
(7) $T_5 = +(T_2, R_2^1)$ and $T_6 = -(T_2, R_2^1)$.
(8) Append-head($T_3$, $C_1^0$).
(9) Append-head($T_4$, $C_1^1$).
(10) Append-head($T_5$, $C_1^1$).
(11) Append-head($T_6$, $C_1^1$).
(12) $T_0 = \cup(T_3, T_4, T_5, T_6)$.
**EndProcedure**

**Proof of Correction:**

The algorithm, **NAND**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$), is implemented by means of the *extract*, *amplify*, *append-head* and *merge* operations. Steps (1) through (4d) apply the *amplify*, *append-head* and *merge* operations to construct four different inputs to the **NAND** operation of two bits as shown in Table 3.12.5. Next, Steps (5) through (7) apply the *extract* operations to form some different test tubes including different inputs ($T_1$ to $T_6$). $T_1$ consists of all of the inputs that have $R_1 = 1$, $T_2$ contains all of the inputs that have $R_1 = 0$, $T_3$ contains that input that has $R_1 = 1$ and $R_2 = 1$, $T_4$ contains that input that has $R_1 = 1$ and $R_2 = 0$, $T_5$ contains that input that has $R_1 = 0$ and $R_2 = 1$, and finally, $T_6$ contains that input that has $R_1 = 0$ and $R_2 = 0$. This is to say that four different inputs for the **NAND** operation of two bits as shown in Table 3.12.5 were poured into tubes $T_3$ through $T_6$, respectively. Next, Steps (8) through (11) use the *append-head* operations to append $C_1^0$ and $C_1^1$ onto the head of every input in the

corresponding tubes. This implies that four different outputs to the **NAND** operation of two bits as shown in Table 3.12.5 are appended into tubes $T_3$ through $T_6$. Finally, the execution of Step (12) uses the *merge* operation to pour tubes $T_3$ through $T_6$ into tube $T_0$. Tube $T_0$ includes the result implementing the **NAND** operation of two bits as shown in Table 3.12.5. ■

**3.6:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are employed to encode the first input and the second input for the **NOR** operation of two bits as shown in Table 3.12.6. Also it is assumed that a binary number of a bit, $C_1$, is used to encode the output for the **OR** operation. For the sake of convenience, it is assumed that for $1 \le k \le 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also assumed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is offered to implement the **NOR** operation of two bits as shown in Table 3.12.6. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes.

**Procedure NOR**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 2
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, $R_k^1$).
    (4c) Append-head($T_2$, $R_k^0$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
(5) $T_1 = +(T_0, R_1^1)$ and $T_2 = -(T_0, R_1^1)$.
(6) $T_3 = +(T_1, R_2^1)$ and $T_4 = -(T_1, R_2^1)$.
(7) $T_5 = +(T_2, R_2^1)$ and $T_6 = -(T_2, R_2^1)$.
(8) Append-head($T_3$, $C_1^0$).
(9) Append-head($T_4$, $C_1^0$).
(10) Append-head($T_5$, $C_1^0$).
(11) Append-head($T_6$, $C_1^1$).
(12) $T_0 = \cup(T_3, T_4, T_5, T_6)$.
**EndProcedure**

**Proof of Correction**:

The algorithm, **NOR**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$), is implemented by means of the *extract*, *append-head*, *amplify* and *merge* operations. Steps (1) through (4d) employ the *amplify*, *append-head* and *merge* operations to construct four different inputs to the **NOR** operation of two bits as shown in Table 3.12.6. Next, Steps (5) through (7) apply the *extract* operations to form some different test tubes containing different inputs ($T_1$ to $T_6$). $T_1$ consists of all of the inputs that have $R_1 = 1$, $T_2$ consists of all of the inputs that have $R_1 = 0$, $T_3$ consists of that input that has $R_1 = 1$ and $R_2 = 1$, $T_4$ consists of that input that has $R_1 = 1$ and $R_2 = 0$, $T_5$ consists of that input that has $R_1 = 0$ and $R_2 = 1$, and finally, $T_6$ consists of that input that has $R_1 = 0$ and $R_2 = 0$. This is to say that four different inputs for the **NOR** operation of two bits as shown in Table 3.12.6 were poured into tubes $T_3$ through $T_6$, respectively. Next, Steps (8) through (11) apply the *append-head* operations to append $C_1^0$ and $C_1^1$ onto the head of every input in the corresponding tubes. This indicates that four different outputs to the **NOR** operation of two bits as shown in Table 3.12.6 are appended into tubes $T_3$ through $T_6$. Finally, the execution of Step (12) uses the *merge* operation to pour tubes $T_3$ through $T_6$ into tube $T_0$. Tube $T_0$ includes the result implementing the **NOR** operation of two bits as shown in Table 3.12.6. ■

**3.7:**

It is assumed that two binary numbers of a bit, $R_1$ and $R_2$, are used to encode the first input and the second input for the **Exclusive-OR** operation of two bits as shown in Table 3.12.7. Also it is supposed that a binary number of a bit, $C_1$, is applied to encode the output for the **Exclusive-OR** operation. For the sake of convenience, it is assumed that for $1 \le k \le 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also supposed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is proposed to implement the **Exclusive-OR** operation of two bits as shown in Table 3.12.7. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes.

**Procedure EXCLUSIVE-OR**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 2
    (4a) Amplify($T_0$, $T_1$, $T_2$).

(4b) Append-head($T_1$, $R_k^1$).

(4c) Append-head($T_2$, $R_k^0$).

(4d) $T_0 = \cup(T_1, T_2)$.

**EndFor**

(5) $T_1 = +(T_0, R_1^1)$ and $T_2 = -(T_0, R_1^1)$.

(6) $T_3 = +(T_1, R_2^1)$ and $T_4 = -(T_1, R_2^1)$.

(7) $T_5 = +(T_2, R_2^1)$ and $T_6 = -(T_2, R_2^1)$.

(8) Append-head($T_3$, $C_1^0$).

(9) Append-head($T_4$, $C_1^1$).

(10) Append-head($T_5$, $C_1^1$).

(11) Append-head($T_6$, $C_1^0$).

(12) $T_0 = \cup(T_3, T_4, T_5, T_6)$.

**EndProcedure**

**Proof of Correction**:

The algorithm, **EXCLUSIVE-OR**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$), is implemented by means of the *extract*, *append-head*, *amplify* and *merge* operations. Steps (1) through (4d) use the *amplify*, *append-head* and *merge* operations to generate four different inputs to the **Exclusive-OR** operation of two bits as shown in Table 3.12.7. Next, Steps (5) through (7) employ the *extract* operations to form some different test tubes including different inputs ($T_1$ to $T_6$). $T_1$ includes all of the inputs that have $R_1 = 1$, $T_2$ includes all of the inputs that have $R_1 = 0$, $T_3$ includes that input that has $R_1 = 1$ and $R_2 = 1$, $T_4$ includes that input that has $R_1 = 1$ and $R_2 = 0$, $T_5$ includes that input that has $R_1 = 0$ and $R_2 = 1$, and finally, $T_6$ includes that input that has $R_1 = 0$ and $R_2 = 0$. This indicates that four different inputs for the **Exclusive-OR** operation of two bits as shown in Table 3.12.7 were poured into tubes $T_3$ through $T_6$, respectively. Next, Steps (8) through (11) apply the *append-head* operations to append $C_1^0$ and $C_1^1$ onto the head of every input in the corresponding tubes. This is to say that four different outputs to the **Exclusive-OR** operation of two bits as shown in Table 3.12.7 are appended into tubes $T_3$ through $T_6$. Finally, the execution of Step (12) applies the *merge* operation to pour tubes $T_3$ through $T_6$ into tube $T_0$. Tube $T_0$ contains the result implementing the **Exclusive-OR** operation of two bits as shown in Table 3.12.7. ■

**3.8:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are applied to encode the first input and the second input for the **Exclusive-NOR** operation of two bits as

shown in Table 3.12.8. Also it is assumed that a binary number of a bit, $C_1$, is employed to encode the output for the **Exclusive-NOR** operation. For the sake of convenience, it is supposed that for $1 \le k \le 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also assumed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is proposed to implement the **Exclusive-NOR** operation of two bits as shown in Table 3.12.8. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes.

**Procedure EXCLUSIVE-NOR**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 2
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, $R_k^1$).
    (4c) Append-head($T_2$, $R_k^0$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
(5) $T_1 = +(T_0, R_1^1)$ and $T_2 = -(T_0, R_1^1)$.
(6) $T_3 = +(T_1, R_2^1)$ and $T_4 = -(T_1, R_2^1)$.
(7) $T_5 = +(T_2, R_2^1)$ and $T_6 = -(T_2, R_2^1)$.
(8) Append-head($T_3$, $C_1^1$).
(9) Append-head($T_4$, $C_1^0$).
(10) Append-head($T_5$, $C_1^0$).
(11) Append-head($T_6$, $C_1^1$).
(12) $T_0 = \cup(T_3, T_4, T_5, T_6)$.
**EndProcedure**

**Proof of Correction**:

The algorithm, **EXCLUSIVE-NOR**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$), is implemented by means of the *extract*, *append-head*, *amplify* and *merge* operations. Steps (1) through (4d) use the *amplify*, *append-head* and *merge* operations to produce four different inputs to the **Exclusive-NOR** operation of two bits as shown in Table 3.12.8. Next, Steps (5) through (7) employ the *extract* operations to form some different test tubes containing different inputs ($T_1$ to $T_6$). $T_1$ consists of all of the inputs that have $R_1 = 1$,

$T_2$ consists of all of the inputs that have $R_1 = 0$, $T_3$ consists of that input that has $R_1 = 1$ and $R_2 = 1$, $T_4$ consists of that input that has $R_1 = 1$ and $R_2 = 0$, $T_5$ consists of that input that has $R_1 = 0$ and $R_2 = 1$, and finally, $T_6$ consists of that input that has $R_1 = 0$ and $R_2 = 0$. This is to say that four different inputs for the **Exclusive-NOR** operation of two bits as shown in Table 3.12.8 were poured into tubes $T_3$ through $T_6$, respectively. Next, Steps (8) through (11) apply the *append-head* operations to append $C_1^1$ and $C_1^0$ onto the head of every input in the corresponding tubes. This implies that four different outputs to the **Exclusive-NOR** operation of two bits as shown in Table 3.12.8 are appended into tubes $T_3$ through $T_6$. Finally, the execution of Step (12) applies the *merge* operation to pour tubes $T_3$ through $T_6$ into tube $T_0$. Tube $T_0$ consists of the result implementing the **Exclusive-NOR** operation of two bits as shown in Table 3.12.8. ■

**3.9:**

It is assumed that two binary numbers of a bit, $R_1$ and $R_2$, are used to encode the first input and the second input for the **NULL** operation of two bits as shown in Table 3.12.9. Also it is supposed that a binary number of a bit, $C_1$, is applied to encode the output for the **NULL** operation. For the sake of convenience, it is supposed that for $1 \le k \le 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also supposed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is proposed to implement the **NULL** operation of two bits as shown in Table 3.12.9. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes.

**Procedure NULL**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 2
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, $R_k^1$).
    (4c) Append-head($T_2$, $R_k^0$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
(5) $T_1 = +(T_0, R_1^1)$ and $T_2 = -(T_0, R_1^1)$.
(6) $T_3 = +(T_1, R_2^1)$ and $T_4 = -(T_1, R_2^1)$.
(7) $T_5 = +(T_2, R_2^1)$ and $T_6 = -(T_2, R_2^1)$.

(8) Append-head($T_3$, $C_1^0$).
(9) Append-head($T_4$, $C_1^0$).
(10) Append-head($T_5$, $C_1^0$).
(11) Append-head($T_6$, $C_1^0$).
(12) $T_0 = \cup(T_3, T_4, T_5, T_6)$.
**EndProcedure**

**Proof of Correction**:

The algorithm, **NULL**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$), is implemented by means of the *extract*, *append-head*, *amplify* and *merge* operations. Steps (1) through (4d) use the *amplify*, *append-head* and *merge* operations to yield four different inputs to the **NULL** operation of two bits as shown in Table 3.12.9. Next, Steps (5) through (7) use the *extract* operations to generate some different test tubes including different inputs ($T_1$ to $T_6$). $T_1$ contains all of the inputs that have $R_1 = 1$, $T_2$ contains all of the inputs that have $R_1 = 0$, $T_3$ contains that input that has $R_1 = 1$ and $R_2 = 1$, $T_4$ contains that input that has $R_1 = 1$ and $R_2 = 0$, $T_5$ contains that input that has $R_1 = 0$ and $R_2 = 1$, and finally, $T_6$ contains that input that has $R_1 = 0$ and $R_2 = 0$. This implies that four different inputs for the **NULL** operation of two bits as shown in Table 3.12.9 were poured into tubes $T_3$ through $T_6$, respectively. Next, Steps (8) through (11) use the *append-head* operations to append $C_1^0$ onto the head of every input in the corresponding tubes. This indicates that four different outputs to the **NULL** operation of two bits as shown in Table 3.12.9 are appended into tubes $T_3$ through $T_6$. Finally, the execution of Step (12) applies the *merge* operation to pour tubes $T_3$ through $T_6$ into tube $T_0$. Tube $T_0$ contains the result implementing the **NULL** operation of two bits as shown in Table 3.12.9. ■

**3.10:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are employed to encode the first input and the second input for the **IDENTITY** operation of two bits as shown in Table 3.12.10. Also it is assumed that a binary number of a bit, $C_1$, is used to encode the output for the **IDENTITY** operation. For the sake of convenience, it is assumed that for $1 \le k \le 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also assumed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is proposed to implement the **IDENTITY** operation of two bits as shown in Table 3.12.10. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes.

**Procedure IDENTITY**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 2
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, $R_k^1$).
    (4c) Append-head($T_2$, $R_k^0$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
(5) $T_1 = +(T_0, R_1^1)$ and $T_2 = -(T_0, R_1^1)$.
(6) $T_3 = +(T_1, R_2^1)$ and $T_4 = -(T_1, R_2^1)$.
(7) $T_5 = +(T_2, R_2^1)$ and $T_6 = -(T_2, R_2^1)$.
(8) Append-head($T_3$, $C_1^1$).
(9) Append-head($T_4$, $C_1^1$).
(10) Append-head($T_5$, $C_1^1$).
(11) Append-head($T_6$, $C_1^1$).
(12) $T_0 = \cup(T_3, T_4, T_5, T_6)$.
**EndProcedure**

**Proof of Correction**:

The algorithm, **IDENTITY**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$), is implemented by means of the *extract*, *append-head*, *amplify* and *merge* operations. Steps (1) through (4d) use the *amplify*, *append-head* and *merge* operations to produce four different inputs to the **IDENTITY** operation of two bits as shown in Table 3.12.10. Next, Steps (5) through (7) apply the *extract* operations to yield some different test tubes containing different inputs ($T_1$ to $T_6$). $T_1$ includes all of the inputs that have $R_1 = 1$, $T_2$ includes all of the inputs that have $R_1 = 0$, $T_3$ includes that input that has $R_1 = 1$ and $R_2 = 1$, $T_4$ includes that input that has $R_1 = 1$ and $R_2 = 0$, $T_5$ includes that input that has $R_1 = 0$ and $R_2 = 1$, and finally, $T_6$ includes that input that has $R_1 = 0$ and $R_2 = 0$. This is to say that four different inputs for the **IDENTITY** operation of two bits as shown in Table 3.12.10 were poured into tubes $T_3$ through $T_6$, respectively. Next, Steps (8) through (11) apply the *append-head* operations to append $C_1^1$ onto the head of every input in the corresponding tubes. This implies that four different outputs to the **IDENTITY** operation of two bits as shown in Table 3.12.10 are appended into tubes $T_3$ through $T_6$. Finally, the execution of Step (12) uses the *merge* operation to pour tubes $T_3$ through

$T_6$ into tube $T_0$. Tube $T_0$ consists of the result implementing the **IDENTITY** operation of two bits as shown in Table 3.12.10. ■

## Chapter 4

### 4.1:

For a decimal system, its base is 10. In the decimal system, there are ten digits, 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. For a decimal system, the first position is 10 raised to the power 0, the second position is 10 raised to the power 1 and the $n^{th}$ position is 10 raised to the power ($n$ – 1). For example, the relationship between the powers and the decimal number 128 is shown in Figure 4.1.

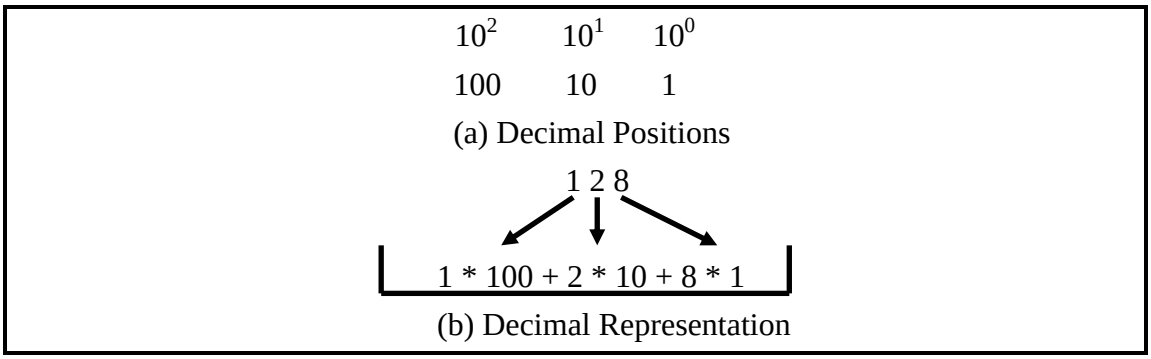


Figure 4.1: The relationship between the powers and the decimal number 128 is shown in a decimal system

For a binary system, its base is 2. There are only two digits in the binary system, 0 and 1. For a binary system, the first position is 2 raised to the power 0, the second position is 2 raised to the power 1 and the $n^{th}$ position is 2 raised to the power ($n$ – 1). The positional weights for a binary system and the value 128 in binary are shown in Figure 4.2. In the position table, each position is double the previous position. Again, this is because the base of the system is 2.

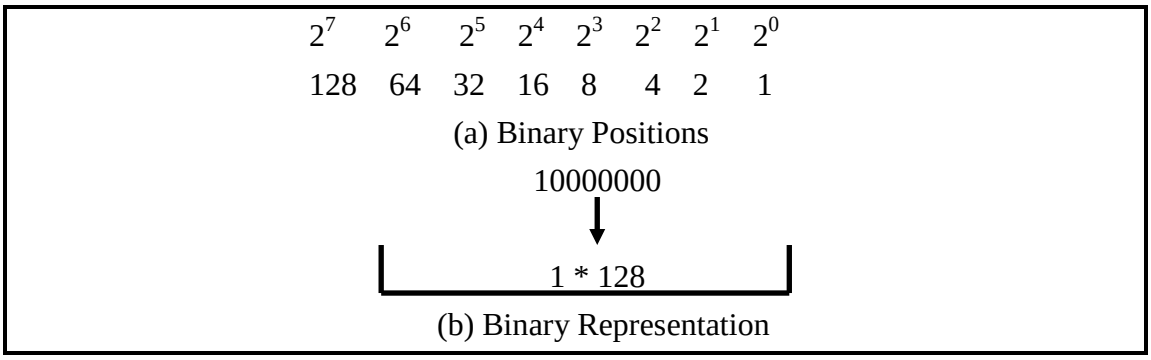


Figure 4.2: The positional weights for a binary system and the value 128 in binary are shown

### 4.2:

The corresponding binary number of the decimal number 128 is 10000000, and the corresponding binary number of the decimal number 33 is 00100001.

**4.3:**

The corresponding decimal number of the binary number 10000000 is 128, and the corresponding decimal number of the binary number 00100001 is 33.

**4.4:** We would like to give many thanks to Cai-Cheng Zhe who wrote the following **C** programs.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i, j, k, decimal, Ans[50]={0};
    i = 0;
    printf("Please enter a decimal number：");
    scanf("%d",&decimal);
    k = decimal;
    while(decimal)
    {   Ans[i] = decimal % 2;
        decimal = decimal / 2;
        i = i + 1;
    }
    printf("The decimal number is %d\n ", k);
    printf(“The corresponding binary number is”);
    for(j=i-1; j>=0; j--) { printf("%d", Ans[j]); }
    printf("\n");
    system("pause");
    return 0;
}
```

**4.5:** We would like to give many thanks to Cai-Cheng Zhe who wrote the following **C** programs.

```
#include <stdio.h>
#include <stdlib.h>
int main()
```

```
{     int i, j, k, binary, decimal;
      j = 1;
      decimal = 0;
      printf("Please enter a binary number：");
      scanf("%d",&binary);
      k = binary;
      while(binary)
      {     i = binary % 10 ;
            decimal += (i * j);
            j *= 2;
            binary /= 10;
      }
      printf("The binary number is %d \n", k);
      printf("The corresponding binary number is %d \n", decimal);
      system("pause");
      return 0;
}
```

**4.6:**

It is assumed that a three-bit binary number, $x_3\ x_2\ x_1$, is employed to encode an unsigned integer with three bits, where the value of each bit $x_k$ is either 1 or 0 for $1 \le k \le 3$. The bits $x_3$ and $x_1$ are used to encode, respectively, the most significant bit and the least significant bit for the unsigned integer with three bits. The following DNA-based algorithm is applied to yield the range of the value for an unsigned integer with three bits. Tubes $T_0$, $T_1$, $T_2$ are, subsequently, the first, second and third parameters, and are set to empty tubes.

**Procedure Yield-Unsigned-Integers-With-Three-Bits**($T_0$, $T_1$, $T_2$)
(1) Append-head($T_1$, $x_1^{\ 1}$).
(2) Append-head($T_2$, $x_1^{\ 0}$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 3
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, $x_k^{\ 1}$).
    (4c) Append-head($T_2$, $x_k^{\ 0}$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
**EndProcedure**

**Proof of Correction:**

After the first execution for Step (1) and the first execution for Step (2) are implemented, tube $T_1 = \{x_1^{\,1}\}$ and tube $T_2 = \{x_1^{\,0}\}$. Then, after the first execution of Step (3) is implemented, tube $T_0 = \{x_1^{\,1}, x_1^{\,0}\}$, tube $T_1 = \varnothing$ and tube $T_2 = \varnothing$. Step (4) is the main loop and the lower bound and the upper bound are, respectively, two and three, so Steps (4a) through (4d) will be executed two times. After the first execution of Step (4a) is implemented, tube $T_0 = \varnothing$, tube $T_1 = \{x_1^{\,1}, x_1^{\,0}\}$ and tube $T_2 = \{x_1^{\,1}, x_1^{\,0}\}$. Next, after the first execution for Step (4b) and Step (4c) is implemented, tube $T_1 = \{x_2^{\,1}\, x_1^{\,1}, x_2^{\,1}\, x_1^{\,0}\}$ and tube $T_2 = \{x_2^{\,0}\, x_1^{\,1}, x_2^{\,0}\, x_1^{\,0}\}$. After the first execution of Step (4d) is implemented, tube $T_0 = \{x_2^{\,1}\, x_1^{\,1}, x_2^{\,1}\, x_1^{\,0}, x_2^{\,0}\, x_1^{\,1}, x_2^{\,0}\, x_1^{\,0}\}$, tube $T_1 = \varnothing$ and tube $T_2 = \varnothing$.

Then, after the second execution of Step (4a) is implemented, tube $T_0 = \varnothing$, tube $T_1 = \{x_2^{\,1}\, x_1^{\,1}, x_2^{\,1}\, x_1^{\,0}, x_2^{\,0}\, x_1^{\,1}, x_2^{\,0}\, x_1^{\,0}\}$ and tube $T_2 = \{x_2^{\,1}\, x_1^{\,1}, x_2^{\,1}\, x_1^{\,0}, x_2^{\,0}\, x_1^{\,1}, x_2^{\,0}\, x_1^{\,0}\}$. After the second execution for Step (4b) and Step (4c) is implemented, tube $T_1 = \{x_3^{\,1}\, x_2^{\,1}\, x_1^{\,1}, x_3^{\,1}\, x_2^{\,1}\, x_1^{\,0}, x_3^{\,1}\, x_2^{\,0}\, x_1^{\,1}, x_3^{\,1}\, x_2^{\,0}\, x_1^{\,0}\}$ and tube $T_2 = \{x_3^{\,0}\, x_2^{\,1}\, x_1^{\,1}, x_3^{\,0}\, x_2^{\,1}\, x_1^{\,0}, x_3^{\,0}\, x_2^{\,0}\, x_1^{\,1}, x_3^{\,0}\, x_2^{\,0}\, x_1^{\,0}\}$. Next, after the second execution of Step (4d) is implemented, tube $T_0 = \{x_3^{\,1}\, x_2^{\,1}\, x_1^{\,1}, x_3^{\,1}\, x_2^{\,1}\, x_1^{\,0}, x_3^{\,1}\, x_2^{\,0}\, x_1^{\,1}, x_3^{\,1}\, x_2^{\,0}\, x_1^{\,0}, x_3^{\,0}\, x_2^{\,1}\, x_1^{\,1}, x_3^{\,0}\, x_2^{\,1}\, x_1^{\,0}, x_3^{\,0}\, x_2^{\,0}\, x_1^{\,1}, x_3^{\,0}\, x_2^{\,0}\, x_1^{\,0}\}$, tube $T_1 = \varnothing$ and tube $T_2 = \varnothing$. The result for tube $T_0$ is shown in Table 4.1. It is inferred from Table 4.1 that **Yield-Unsigned-Integers-With-Three-Bits**($T_0$, $T_1$, $T_2$) can be used to yield the range of the value for an unsigned integer with three bits. ■

| Tube | The result is yielded by **Yield-Unsigned-Integers-With-Three-Bits**($T_0$, $T_1$, $T_2$) |
|---|---|
| $T_0$ | $\{x_3^{\,1}\, x_2^{\,1}\, x_1^{\,1}, \quad x_3^{\,1}\, x_2^{\,1}\, x_1^{\,0}, \quad x_3^{\,1}\, x_2^{\,0}\, x_1^{\,1}, \quad x_3^{\,1}\, x_2^{\,0}\, x_1^{\,0},$ <br> $x_3^{\,0}\, x_2^{\,1}\, x_1^{\,1}, \quad x_3^{\,0}\, x_2^{\,1}\, x_1^{\,0}, \quad x_3^{\,0}\, x_2^{\,0}\, x_1^{\,1}, \quad x_3^{\,0}\, x_2^{\,0}\, x_1^{\,0}\}$ |

Table 4.1: The result for tube $T_0$ is generated by **Yield-Unsigned-Integers-With-Three-Bits**($T_0$, $T_1$, $T_2$).

**4.7:**

It is supposed that a three-bit binary number, $x_3\ x_2\ x_1$, is employed to encode a sign-and-magnitude integer with three bits, where the value of each bit $x_k$ is either 1 or 0 for $1 \le k \le 3$. The bit $x_3$ is applied to encode the sign, and the bits $x_2$ and $x_1$ is used to encode, respectively, the most significant bit and the least significant bit for a sign-and-magnitude integer with $n$ bits. From **Definition 4-2**, it is very clear that there

are two 0s in sign-and-magnitude representation: positive and negative. For example, in a three-bit allocation: “000” is used to encode “+0” and “100” is employed to encode “–0”. The following DNA-based algorithm is used to produce the range of the value for a sign-and-magnitude integer with three bits. Tubes $T_0$, $T_1$, $T_2$ are, subsequently, the first, second and third parameters, and are set to empty tubes.

**Procedure Yield-Sign-and-Magtitude-Integers-With-Three-Bits**($T_0$, $T_1$, $T_2$)
(1) Append-head($T_1$, ${x_1}^1$).
(2) Append-head($T_2$, ${x_1}^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 3
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, ${x_k}^1$).
    (4c) Append-head($T_2$, ${x_k}^0$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
**EndProcedure**

**Proof of Correction:**

After the first execution of Step (1) and the first execution of Step (2) are implemented, tube $T_1 = \{{x_1}^1\}$ and tube $T_2 = \{{x_1}^0\}$. Then, after the first execution of Step (3) is implemented, tube $T_0 = \{{x_1}^1, {x_1}^0\}$, tube $T_1 = \varnothing$ and tube $T_2 = \varnothing$. Step (4) is the main loop and the lower bound and the upper bound are, subsequently, are two and three, so Steps (4a) through (4d) will be implemented two times. After the first execution of Step (4a) is implemented, tube $T_0 = \varnothing$, tube $T_1 = \{{x_1}^1, {x_1}^0\}$ and tube $T_2 = \{{x_1}^1, {x_1}^0\}$. Next, after the first execution for Step (4b) and Step (4c) is implemented, tube $T_1 = \{{x_2}^1 {x_1}^1, {x_2}^1 {x_1}^0\}$ and tube $T_2 = \{{x_2}^0 {x_1}^1, {x_2}^0 {x_1}^0\}$. After the first execution of Step (4d) is implemented, tube $T_0 = \{{x_2}^1 {x_1}^1, {x_2}^1 {x_1}^0, {x_2}^0 {x_1}^1, {x_2}^0 {x_1}^0\}$, tube $T_1 = \varnothing$ and tube $T_2 = \varnothing$.

Then, after the second execution of Step (4a) is implemented, tube $T_0 = \varnothing$, tube $T_1 = \{{x_2}^1 {x_1}^1, {x_2}^1 {x_1}^0, {x_2}^0 {x_1}^1, {x_2}^0 {x_1}^0\}$ and tube $T_2 = \{{x_2}^1 {x_1}^1, {x_2}^1 {x_1}^0, {x_2}^0 {x_1}^1, {x_2}^0 {x_1}^0\}$. After the second execution for Step (4b) and Step (4c) is implemented, tube $T_1 = \{{x_3}^1 {x_2}^1 {x_1}^1, {x_3}^1 {x_2}^1 {x_1}^0, {x_3}^1 {x_2}^0 {x_1}^1, {x_3}^1 {x_2}^0 {x_1}^0\}$ and tube $T_2 = \{{x_3}^0 {x_2}^1 {x_1}^1, {x_3}^0 {x_2}^1 {x_1}^0, {x_3}^0 {x_2}^0 {x_1}^1, {x_3}^0 {x_2}^0 {x_1}^0\}$. Next, after the second execution of Step (4d) is implemented, tube $T_0 = \{{x_3}^1 {x_2}^1 {x_1}^1, {x_3}^1 {x_2}^1 {x_1}^0, {x_3}^1 {x_2}^0 {x_1}^1, {x_3}^1 {x_2}^0 {x_1}^0, {x_3}^0 {x_2}^1 {x_1}^1, {x_3}^0 {x_2}^1 {x_1}^0, {x_3}^0 {x_2}^0 {x_1}^1, {x_3}^0 {x_2}^0 {x_1}^0\}$, tube $T_1 = \varnothing$ and tube $T_2 = \varnothing$. The result for tube $T_0$ is shown in Table 4.2. It is derived

from Table 4.2 that **Yield-Sign-and-Magtitude-Integers-With-Three-Bits**($T_0$, $T_1$, $T_2$) can be used to produce the range of the value for a sign-and-magnitude integer with three bits. ■

| Tube | The result is generated by **Yield-Sign-and-Magtitude-Integers-With-Three-Bits**($T_0$, $T_1$, $T_2$) |
|---|---|
| $T_0$ | $\{x_3^{\,1}\, x_2^{\,1}\, x_1^{\,1},\quad x_3^{\,1}\, x_2^{\,1}\, x_1^{\,0},\quad x_3^{\,1}\, x_2^{\,0}\, x_1^{\,1},\quad x_3^{\,1}\, x_2^{\,0}\, x_1^{\,0},$ <br> $x_3^{\,0}\, x_2^{\,1}\, x_1^{\,1},\quad x_3^{\,0}\, x_2^{\,1}\, x_1^{\,0},\quad x_3^{\,0}\, x_2^{\,0}\, x_1^{\,1},\quad x_3^{\,0}\, x_2^{\,0}\, x_1^{\,0}\}$ |

Table 4.2: The result for tube $T_0$ is generated by **Yield-Sign-and-Magtitude-Integers-With-Three-Bits**($T_0$, $T_1$, $T_2$).

**4.8:**

It is assumed that a three-bit binary number, $x_3\ x_2\ x_1$, is employed to encode a one's complement integer with three bits, where the value of each bit $x_k$ is either 1 or 0 for $1 \le k \le 3$. From **Definition 4-3**, it is indicated that there are two 0s in one's complement representation: positive and negative. For example, in a three-bit allocation: "000" is used to encode "+0" and "111" is applied to encode "–0". The following DNA-based algorithm is used to produce the range of the value for a one's complement integer with three bits. Tubes $T_0$, $T_1$, $T_2$ are, subsequently, the first, second and third parameters, and are set to empty tubes.

**Procedure Yield-One's-Complement-Integers-With-Three-Bits**($T_0$, $T_1$, $T_2$)
(1) Append-head($T_1$, $x_1^{\,1}$).
(2) Append-head($T_2$, $x_1^{\,0}$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 3
  (4a) Amplify($T_0$, $T_1$, $T_2$).
  (4b) Append-head($T_1$, $x_k^{\,1}$).
  (4c) Append-head($T_2$, $x_k^{\,0}$).
  (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
**EndProcedure**

**Proof of Correction:**

After the first execution of Step (1) and the first execution of Step (2) are implemented, tube $T_1 = \{x_1^{\,1}\}$ and tube $T_2 = \{x_1^{\,0}\}$. Then, after the first execution of

Step (3) is implemented, tube $T_0 = \{x_1^{\,1}, x_1^{\,0}\}$, tube $T_1 = \varnothing$ and tube $T_2 = \varnothing$. Step (4) is the main loop and the lower bound and the upper bound are, respectively, two and three, so Steps (4a) through (4d) will be implemented two times. After the first execution of Step (4a) is implemented, tube $T_0 = \varnothing$, tube $T_1 = \{x_1^{\,1}, x_1^{\,0}\}$ and tube $T_2 = \{x_1^{\,1}, x_1^{\,0}\}$. Next, after the first execution for Step (4b) and Step (4c) is implemented, tube $T_1 = \{x_2^{\,1}\, x_1^{\,1}, x_2^{\,1}\, x_1^{\,0}\}$ and tube $T_2 = \{x_2^{\,0}\, x_1^{\,1}, x_2^{\,0}\, x_1^{\,0}\}$. After the first execution of Step (4d) is implemented, tube $T_0 = \{x_2^{\,1}\, x_1^{\,1}, x_2^{\,1}\, x_1^{\,0}, x_2^{\,0}\, x_1^{\,1}, x_2^{\,0}\, x_1^{\,0}\}$, tube $T_1 = \varnothing$ and tube $T_2 = \varnothing$.

Then, after the second execution of Step (4a) is implemented, tube $T_0 = \varnothing$, tube $T_1 = \{x_2^{\,1}\, x_1^{\,1}, x_2^{\,1}\, x_1^{\,0}, x_2^{\,0}\, x_1^{\,1}, x_2^{\,0}\, x_1^{\,0}\}$ and tube $T_2 = \{x_2^{\,1}\, x_1^{\,1}, x_2^{\,1}\, x_1^{\,0}, x_2^{\,0}\, x_1^{\,1}, x_2^{\,0}\, x_1^{\,0}\}$. After the second execution for Step (4b) and Step (4c) is implemented, tube $T_1 = \{x_3^{\,1}\, x_2^{\,1}\, x_1^{\,1}, x_3^{\,1}\, x_2^{\,1}\, x_1^{\,0}, x_3^{\,1}\, x_2^{\,0}\, x_1^{\,1}, x_3^{\,1}\, x_2^{\,0}\, x_1^{\,0}\}$ and tube $T_2 = \{x_3^{\,0}\, x_2^{\,1}\, x_1^{\,1}, x_3^{\,0}\, x_2^{\,1}\, x_1^{\,0}, x_3^{\,0}\, x_2^{\,0}\, x_1^{\,1}, x_3^{\,0}\, x_2^{\,0}\, x_1^{\,0}\}$. Next, after the second execution of Step (4d) is implemented, tube $T_0 = \{x_3^{\,1}\, x_2^{\,1}\, x_1^{\,1}, x_3^{\,1}\, x_2^{\,1}\, x_1^{\,0}, x_3^{\,1}\, x_2^{\,0}\, x_1^{\,1}, x_3^{\,1}\, x_2^{\,0}\, x_1^{\,0}, x_3^{\,0}\, x_2^{\,1}\, x_1^{\,1}, x_3^{\,0}\, x_2^{\,1}\, x_1^{\,0}, x_3^{\,0}\, x_2^{\,0}\, x_1^{\,1}, x_3^{\,0}\, x_2^{\,0}\, x_1^{\,0}\}$, tube $T_1 = \varnothing$ and tube $T_2 = \varnothing$. The result for tube $T_0$ is shown in Table 4.3. It is concluded from Table 4.3 that **Yield-One's-Complement-Integers-With-Three-Bits**($T_0$, $T_1$, $T_2$) can be employed to generate the range of the value for a one's complement integer with three bits. ■

| Tube | The result is generated by **Yield-One's-Complement-Integers-With-Three-Bits**($T_0$, $T_1$, $T_2$) |
|---|---|
| $T_0$ | $\{x_3^{\,1}\, x_2^{\,1}\, x_1^{\,1}, \quad x_3^{\,1}\, x_2^{\,1}\, x_1^{\,0}, \quad x_3^{\,1}\, x_2^{\,0}\, x_1^{\,1}, \quad x_3^{\,1}\, x_2^{\,0}\, x_1^{\,0},$ <br> $x_3^{\,0}\, x_2^{\,1}\, x_1^{\,1}, \quad x_3^{\,0}\, x_2^{\,1}\, x_1^{\,0}, \quad x_3^{\,0}\, x_2^{\,0}\, x_1^{\,1}, \quad x_3^{\,0}\, x_2^{\,0}\, x_1^{\,0}\}$ |

Table 4.3: The result for tube $T_0$ is generated by **Yield-One's-Complement-Integers-With-Three-Bits**($T_0$, $T_1$, $T_2$).

**4.9:**

It is supposed that a three-bit binary number, $x_3\, x_2\, x_1$, is applied to encode a two's complement integer with three bits, where the value of each bit $x_k$ is either 1 or 0 for $1 \le k \le 3$. From **Definition 4-4**, it is pointed out that there is only one 0 in two's complement representation. For example, in a three-bit allocation: "000" is used to encode "0". The following DNA-based algorithm is applied to construct the range of the value for a two's complement integer with three bits. Tubes $T_0$, $T_1$, $T_2$ are, subsequently, the first, second and third parameters, and are set to empty tubes.

**Procedure Yield-Two's-Complement-Integers-With-Three-Bits**($T_0$, $T_1$, $T_2$)

(1) Append-head($T_1$, $x_1^1$).
(2) Append-head($T_2$, $x_1^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 3
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, $x_k^1$).
    (4c) Append-head($T_2$, $x_k^0$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
**EndProcedure**

**Proof of Correction:**

After the first execution for Step (1) and Step (2) is implemented, tube $T_1 = \{x_1^1\}$ and tube $T_2 = \{x_1^0\}$. Next, after the first execution of Step (3) is implemented, tube $T_0 = \{x_1^1, x_1^0\}$, tube $T_1 = \varnothing$ and tube $T_2 = \varnothing$. Step (4) is the main loop and its lower and upper bounds are, respectively, two and three, so Steps (4a) through (4d) will be implemented two times. After the first execution of Step (4a) is implemented, tube $T_0 = \varnothing$, tube $T_1 = \{x_1^1, x_1^0\}$ and tube $T_2 = \{x_1^1, x_1^0\}$. Next, after the first execution for Step (4b) and Step (4c) is implemented, tube $T_1 = \{x_2^1\, x_1^1, x_2^1\, x_1^0\}$ and tube $T_2 = \{x_2^0\, x_1^1, x_2^0\, x_1^0\}$. After the first execution of Step (4d) is implemented, tube $T_0 = \{x_2^1\, x_1^1, x_2^1\, x_1^0, x_2^0\, x_1^1, x_2^0\, x_1^0\}$, tube $T_1 = \varnothing$ and tube $T_2 = \varnothing$.

Then, after the second execution of Step (4a) is implemented, tube $T_0 = \varnothing$, tube $T_1 = \{x_2^1\, x_1^1, x_2^1\, x_1^0, x_2^0\, x_1^1, x_2^0\, x_1^0\}$ and tube $T_2 = \{x_2^1\, x_1^1, x_2^1\, x_1^0, x_2^0\, x_1^1, x_2^0\, x_1^0\}$. After the second execution for Step (4b) and Step (4c) is implemented, tube $T_1 = \{x_3^1\, x_2^1\, x_1^1, x_3^1\, x_2^1\, x_1^0, x_3^1\, x_2^0\, x_1^1, x_3^1\, x_2^0\, x_1^0\}$ and tube $T_2 = \{x_3^0\, x_2^1\, x_1^1, x_3^0\, x_2^1\, x_1^0, x_3^0\, x_2^0\, x_1^1, x_3^0\, x_2^0\, x_1^0\}$. Next, after the second execution of Step (4d) is implemented, tube $T_0 = \{x_3^1\, x_2^1\, x_1^1, x_3^1\, x_2^1\, x_1^0, x_3^1\, x_2^0\, x_1^1, x_3^1\, x_2^0\, x_1^0, x_3^0\, x_2^1\, x_1^1, x_3^0\, x_2^1\, x_1^0, x_3^0\, x_2^0\, x_1^1, x_3^0\, x_2^0\, x_1^0\}$, tube $T_1 = \varnothing$ and tube $T_2 = \varnothing$. The result for tube $T_0$ is shown in Table 4.4. It is inferred from Table 4.4 that **Yield-Two's-Complement-Integers-With-Three-Bits**($T_0$, $T_1$, $T_2$) can be applied to construct the range of the value for a two's complement integer with three bits. ■

| Tube | The result is generated by **Yield-Two's-Complement-Integers-With-Three-Bits**($T_0$, $T_1$, $T_2$) |
|---|---|
| $T_0$ | $\{x_3^1\, x_2^1\, x_1^1,\quad x_3^1\, x_2^1\, x_1^0,\quad x_3^1\, x_2^0\, x_1^1,\quad x_3^1\, x_2^0\, x_1^0,$ |

| | |
|---|---|
| | $x_3^{\,0}\, x_2^{\,1}\, x_1^{\,1}$, $x_3^{\,0}\, x_2^{\,1}\, x_1^{\,0}$, $x_3^{\,0}\, x_2^{\,0}\, x_1^{\,1}$, $x_3^{\,0}\, x_2^{\,0}\, x_1^{\,0}\}$ |

Table 4.4: The result for tube $T_0$ is generated by **Yield-Two's-Complement-Integers-With-Three-Bits**($T_0$, $T_1$, $T_2$).

**4.10:**

It is assumed that a 32-bit binary number, $x_{32} \ldots x_1$ is applied to encode a floating-point number of 32 bits in form of single precision format based on Excess_127, where the value of each bit $x_k$ is either 1 or 0 for $1 \leq k \leq 32$. The following DNA-based algorithm is applied to yield the range of the value for a floating-point number with thirty-two bits in form of single precision format based on Excess_127. Tubes $T_0$, $T_1$, $T_2$ are, subsequently, the first, second and third parameters, and are set to empty tubes.

**Procedure Yield-Single-Precision-Floating-Point-Numbers**($T_0$, $T_1$, $T_2$)
(1) Append-head($T_1$, $x_1^{\,1}$).
(2) Append-head($T_2$, $x_1^{\,0}$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k$ = 2 **to** 32
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, $x_k^{\,1}$).
    (4c) Append-head($T_2$, $x_k^{\,0}$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
**EndProcedure**

**Proof of Correction:**

After the first execution for Step (1) and Step (2) is completed, tube $T_1 = \{x_1^{\,1}\}$ and tube $T_2 = \{x_1^{\,0}\}$. Then, after the first execution of Step (3) is implemented, tube $T_0 = \{x_1^{\,1}, x_1^{\,0}\}$, tube $T_1 = \varnothing$ and tube $T_2 = \varnothing$. Step (4) is the main loop and its lower bound and the upper bound are, respectively, two and thirty-two, so Steps (4a) through (4d) will be implemented thirty-one times. After the first execution of Step (4a) is implemented, tube $T_0 = \varnothing$, tube $T_1 = \{x_1^{\,1}, x_1^{\,0}\}$ and tube $T_2 = \{x_1^{\,1}, x_1^{\,0}\}$. Next, after the first execution for Step (4b) and Step (4c) is implemented, tube $T_1 = \{x_2^{\,1}\, x_1^{\,1}, x_2^{\,1}\, x_1^{\,0}\}$ and tube $T_2 = \{x_2^{\,0}\, x_1^{\,1}, x_2^{\,0}\, x_1^{\,0}\}$. After the first execution of Step (4d) is implemented, tube $T_0 = \{x_2^{\,1}\, x_1^{\,1}, x_2^{\,1}\, x_1^{\,0}, x_2^{\,0}\, x_1^{\,1}, x_2^{\,0}\, x_1^{\,0}\}$, tube $T_1 = \varnothing$ and tube $T_2 = \varnothing$.

Then, after the second execution of Step (4a) is implemented, tube $T_0 = \varnothing$, tube $T_1$

= $\{x_2^1 x_1^1, x_2^1 x_1^0, x_2^0 x_1^1, x_2^0 x_1^0\}$ and tube $T_2 = \{x_2^1 x_1^1, x_2^1 x_1^0, x_2^0 x_1^1, x_2^0 x_1^0\}$. After the rest of operations are implemented, tube $T_1 = \varnothing$, tube $T_2 = \varnothing$ and the result for tube $T_0$ is shown in Table 4.5. In Table 4.5, for this bit pattern, "$x_{32}^1 x_{31}^1 x_{30}^1 x_{29}^1 x_{28}^1 x_{27}^1 x_{26}^1 x_{25}^1 x_{24}^1 x_{23}^1 x_{22}^1 x_{21}^1 x_{20}^1 x_{19}^1 x_{18}^1 x_{17}^1 x_{16}^1 x_{15}^1 x_{14}^1 x_{13}^1 x_{12}^1 x_{11}^1 x_{10}^1 x_9^1 x_8^1 x_7^1 x_6^1 x_5^1 x_4^1 x_3^1 x_2^1 x_1^1$", the leftmost bit is the sign (–). The next 8 bits, "$x_{31}^1 x_{30}^1 x_{29}^1 x_{28}^1 x_{27}^1 x_{26}^1 x_{25}^1 x_{24}^1$", that subtract $127_{10}$ is the exponent ($128_{10}$). The next 23 bits are the mantissa. So, this bit pattern is used to encode $-(2^{128} \times 1.11111111111111111111111)$. Similarly, in Table 4.5, for that bit pattern, "$x_{32}^0 x_{31}^1 x_{30}^1 x_{29}^1 x_{28}^1 x_{27}^1 x_{26}^1 x_{25}^1 x_{24}^1 x_{23}^1 x_{22}^1 x_{21}^1 x_{20}^1 x_{19}^1 x_{18}^1 x_{17}^1 x_{16}^1 x_{15}^1 x_{14}^1 x_{13}^1 x_{12}^1 x_{11}^1 x_{10}^1 x_9^1 x_8^1 x_7^1 x_6^1 x_5^1 x_4^1 x_3^1 x_2^1 x_1^1$", it is also used to encode $+(2^{128} \times 1.11111111111111111111111)$. It is derived from Table 4.5 that **Yield-Single-Precision-Floating-Point-Numbers**($T_0$, $T_1$, $T_2$) can be used to construct the range of the value for a floating-point number with thirty-two bits in form of single precision format based on Excess_127. ■

| Tube | The result is generated by **Yield-Single-Precision-Floating-Point-Numbers**($T_0$, $T_1$, $T_2$) |
|---|---|
| $T_0$ | $\{x_{32}^1 x_{31}^1 x_{30}^1 x_{29}^1 x_{28}^1 x_{27}^1 x_{26}^1 x_{25}^1 x_{24}^1 x_{23}^1 x_{22}^1 x_{21}^1 x_{20}^1 x_{19}^1 x_{18}^1 x_{17}^1 x_{16}^1 x_{15}^1$ $x_{14}^1 x_{13}^1 x_{12}^1 x_{11}^1 x_{10}^1 x_9^1 x_8^1 x_7^1 x_6^1 x_5^1 x_4^1 x_3^1 x_2^1 x_1^1$ … $x_{32}^0 x_{31}^1 x_{30}^1 x_{29}^1 x_{28}^1 x_{27}^1 x_{26}^1 x_{25}^1 x_{24}^1 x_{23}^1 x_{22}^1 x_{21}^1 x_{20}^1 x_{19}^1 x_{18}^1 x_{17}^1 x_{16}^1 x_{15}^1$ $x_{14}^1 x_{13}^1 x_{12}^1 x_{11}^1 x_{10}^1 x_9^1 x_8^1 x_7^1 x_6^1 x_5^1 x_4^1 x_3^1 x_2^1 x_1^1\}$ |

Table 4.5: The result for tube $T_0$ is generated by **Yield-Single-Precision-Floating-Point-Numbers**($T_0$, $T_1$, $T_2$).

**4.11:**

It is supposed that a 64-bit binary number, $x_{64} \ldots x_1$ is employed to encode a floating-point number of 64 bits in form of double precision format based on Excess_1023, where the value of each bit $x_k$ is either 1 or 0 for $1 \le k \le 64$. The following DNA-based algorithm is used to construct the range of the value for a floating-point number with sixty-four bits in form of double precision format based on Excess_1023. Tubes $T_0$, $T_1$, $T_2$ are, subsequently, the first, second and third parameters, and are set to empty tubes.

**Procedure Yield-Double-Precision-Floating-Point-Numbers**($T_0$, $T_1$, $T_2$)
(1) Append-head($T_1$, $x_1^1$).
(2) Append-head($T_2$, $x_1^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 64

(4a) Amplify($T_0$, $T_1$, $T_2$).

(4b) Append-head($T_1$, $x_k{}^1$).

(4c) Append-head($T_2$, $x_k{}^0$).

(4d) $T_0 = \cup(T_1, T_2)$.

**EndFor**

**EndProcedure**

**Proof of Correction:**

After the first execution for Step (1) and Step (2) is implemented, tube $T_1 = \{x_1{}^1\}$ and tube $T_2 = \{x_1{}^0\}$. Next, after the first execution of Step (3) is implemented, tube $T_0 = \{x_1{}^1, x_1{}^0\}$, tube $T_1 = \varnothing$ and tube $T_2 = \varnothing$. Step (4) is the main loop and its lower and upper bounds are, respectively, two and sixty-four, so Steps (4a) through (4d) will be implemented sixty-three times. After the first execution of Step (4a) is implemented, tube $T_0 = \varnothing$, tube $T_1 = \{x_1{}^1, x_1{}^0\}$ and tube $T_2 = \{x_1{}^1, x_1{}^0\}$. Next, after the first execution for Step (4b) and Step (4c) is implemented, tube $T_1 = \{x_2{}^1\, x_1{}^1, x_2{}^1\, x_1{}^0\}$ and tube $T_2 = \{x_2{}^0\, x_1{}^1, x_2{}^0\, x_1{}^0\}$. After the first execution of Step (4d) is implemented, tube $T_0 = \{x_2{}^1\, x_1{}^1, x_2{}^1\, x_1{}^0, x_2{}^0\, x_1{}^1, x_2{}^0\, x_1{}^0\}$, tube $T_1 = \varnothing$ and tube $T_2 = \varnothing$.

Then, after the second execution of Step (4a) is implemented, tube $T_0 = \varnothing$, tube $T_1 = \{x_2{}^1\, x_1{}^1, x_2{}^1\, x_1{}^0, x_2{}^0\, x_1{}^1, x_2{}^0\, x_1{}^0\}$ and tube $T_2 = \{x_2{}^1\, x_1{}^1, x_2{}^1\, x_1{}^0, x_2{}^0\, x_1{}^1, x_2{}^0\, x_1{}^0\}$. After the rest of operations are implemented, tube $T_1 = \varnothing$, tube $T_2 = \varnothing$ and the result for tube $T_0$ is shown in Table 4.6. In Table 4.6, for this bit pattern, "$x_{64}{}^1\, x_{63}{}^1\, x_{62}{}^1\, x_{61}{}^1\, x_{60}{}^1\, x_{59}{}^1\, x_{58}{}^1\, x_{57}{}^1\, x_{56}{}^1\, x_{55}{}^1\, x_{54}{}^1\, x_{53}{}^1\, x_{52}{}^1\, x_{51}{}^1\, x_{50}{}^1\, x_{49}{}^1\, x_{48}{}^1\, x_{47}{}^1\, x_{46}{}^1\, x_{45}{}^1\, x_{44}{}^1\, x_{43}{}^1\, x_{42}{}^1\, x_{41}{}^1\, x_{40}{}^1\, x_{39}{}^1\, x_{38}{}^1\, x_{37}{}^1\, x_{36}{}^1\, x_{35}{}^1\, x_{34}{}^1\, x_{33}{}^1\, x_{32}{}^1\, x_{31}{}^1\, x_{30}{}^1\, x_{29}{}^1\, x_{28}{}^1\, x_{27}{}^1\, x_{26}{}^1\, x_{25}{}^1\, x_{24}{}^1\, x_{23}{}^1\, x_{22}{}^1\, x_{21}{}^1\, x_{20}{}^1\, x_{19}{}^1\, x_{18}{}^1\, x_{17}{}^1\, x_{16}{}^1\, x_{15}{}^1\, x_{14}{}^1\, x_{13}{}^1\, x_{12}{}^1\, x_{11}{}^1\, x_{10}{}^1\, x_9{}^1\, x_8{}^1\, x_7{}^1\, x_6{}^1\, x_5{}^1\, x_4{}^1\, x_3{}^1\, x_2{}^1\, x_1{}^1$", the leftmost bit is the sign (–). The next 11 bits that subtract $1023_{10}$ is the exponent ($1024_{10}$). The next 52 bits are the mantissa. So, this bit pattern is employed to encode $-(2^{1024} \times 1.1111111111111111111111111111111111111111111111111111)$. Similarly, in Table 4.6, for that bit pattern, "$x_{64}{}^0\, x_{63}{}^1\, x_{62}{}^1\, x_{61}{}^1\, x_{60}{}^1\, x_{59}{}^1\, x_{58}{}^1\, x_{57}{}^1\, x_{56}{}^1\, x_{55}{}^1\, x_{54}{}^1\, x_{53}{}^1\, x_{52}{}^1\, x_{51}{}^1\, x_{50}{}^1\, x_{49}{}^1\, x_{48}{}^1\, x_{47}{}^1\, x_{46}{}^1\, x_{45}{}^1\, x_{44}{}^1\, x_{43}{}^1\, x_{42}{}^1\, x_{41}{}^1\, x_{40}{}^1\, x_{39}{}^1\, x_{38}{}^1\, x_{37}{}^1\, x_{36}{}^1\, x_{35}{}^1\, x_{34}{}^1\, x_{33}{}^1\, x_{32}{}^1\, x_{31}{}^1\, x_{30}{}^1\, x_{29}{}^1\, x_{28}{}^1\, x_{27}{}^1\, x_{26}{}^1\, x_{25}{}^1\, x_{24}{}^1\, x_{23}{}^1\, x_{22}{}^1\, x_{21}{}^1\, x_{20}{}^1\, x_{19}{}^1\, x_{18}{}^1\, x_{17}{}^1\, x_{16}{}^1\, x_{15}{}^1\, x_{14}{}^1\, x_{13}{}^1\, x_{12}{}^1\, x_{11}{}^1\, x_{10}{}^1\, x_9{}^1\, x_8{}^1\, x_7{}^1\, x_6{}^1\, x_5{}^1\, x_4{}^1\, x_3{}^1\, x_2{}^1\, x_1{}^1$", it is also employed to encode $+(2^{1024} \times 1.1111111111111111111111111111111111111111111111111111)$. It is concluded from Table 4.6 that **Yield-Double-Precision-Floating-Point-Numbers**($T_0$, $T_1$, $T_2$) can be used to construct the range of the value for a floating-point number with sixty-four bits in form of double precision format based on Excess_1023. ■

| Tube | The result is generated by **Yield-Double-Precision-Floating-Point-Numbers**($T_0$, $T_1$, $T_2$) |
|---|---|
| $T_0$ | $\{x_{64}^{1}\ x_{63}^{1}\ x_{62}^{1}\ x_{61}^{1}\ x_{60}^{1}\ x_{59}^{1}\ x_{58}^{1}\ x_{57}^{1}\ x_{56}^{1}\ x_{55}^{1}\ x_{54}^{1}\ x_{53}^{1}\ x_{52}^{1}\ x_{51}^{1}\ x_{50}^{1}\ x_{49}^{1}\ x_{48}^{1}\ x_{47}^{1}$<br>$x_{46}^{1}\ x_{45}^{1}\ x_{44}^{1}\ x_{43}^{1}\ x_{42}^{1}\ x_{41}^{1}\ x_{40}^{1}\ x_{39}^{1}\ x_{38}^{1}\ x_{37}^{1}\ x_{36}^{1}\ x_{35}^{1}\ x_{34}^{1}\ x_{33}^{1}\ x_{32}^{1}\ x_{31}^{1}\ x_{30}^{1}\ x_{29}^{1}$<br>$x_{28}^{1}\ x_{27}^{1}\ x_{26}^{1}\ x_{25}^{1}\ x_{24}^{1}\ x_{23}^{1}\ x_{22}^{1}\ x_{21}^{1}\ x_{20}^{1}\ x_{19}^{1}\ x_{18}^{1}\ x_{17}^{1}\ x_{16}^{1}\ x_{15}^{1}\ x_{14}^{1}\ x_{13}^{1}\ x_{12}^{1}\ x_{11}^{1}$<br>$x_{10}^{1}\ x_{9}^{1}\ x_{8}^{1}\ x_{7}^{1}\ x_{6}^{1}\ x_{5}^{1}\ x_{4}^{1}\ x_{3}^{1}\ x_{2}^{1}\ x_{1}^{1}$<br>…<br>$x_{64}^{0}\ x_{63}^{1}\ x_{62}^{1}\ x_{61}^{1}\ x_{60}^{1}\ x_{59}^{1}\ x_{58}^{1}\ x_{57}^{1}\ x_{56}^{1}\ x_{55}^{1}\ x_{54}^{1}\ x_{53}^{1}\ x_{52}^{1}\ x_{51}^{1}\ x_{50}^{1}\ x_{49}^{1}\ x_{48}^{1}\ x_{47}^{1}$<br>$x_{46}^{1}\ x_{45}^{1}\ x_{44}^{1}\ x_{43}^{1}\ x_{42}^{1}\ x_{41}^{1}\ x_{40}^{1}\ x_{39}^{1}\ x_{38}^{1}\ x_{37}^{1}\ x_{36}^{1}\ x_{35}^{1}\ x_{34}^{1}\ x_{33}^{1}\ x_{32}^{1}\ x_{31}^{1}\ x_{30}^{1}\ x_{29}^{1}$<br>$x_{28}^{1}\ x_{27}^{1}\ x_{26}^{1}\ x_{25}^{1}\ x_{24}^{1}\ x_{23}^{1}\ x_{22}^{1}\ x_{21}^{1}\ x_{20}^{1}\ x_{19}^{1}\ x_{18}^{1}\ x_{17}^{1}\ x_{16}^{1}\ x_{15}^{1}\ x_{14}^{1}\ x_{13}^{1}\ x_{12}^{1}\ x_{11}^{1}$<br>$x_{10}^{1}\ x_{9}^{1}\ x_{8}^{1}\ x_{7}^{1}\ x_{6}^{1}\ x_{5}^{1}\ x_{4}^{1}\ x_{3}^{1}\ x_{2}^{1}\ x_{1}^{1}\}$ |

Table 4.6: The result for tube $T_0$ is generated by **Yield-Double-Precision-Floating-Point-Numbers**($T_0$, $T_1$, $T_2$).

## Chapter 5

**5.1:**

It is assumed that two binary numbers of a bit, $R_1$ and $R_2$, are used to encode the first input and the second input for $x \vee 0$ as shown in Table 5.6.1. Also it is supposed that a binary number of a bit, $C_1$, is applied to encode the output for the logical operation, $x \vee 0$. For the sake of convenience, it is supposed that for $1 \leq k \leq 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also assumed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is offered to implement, $x \vee 0$, as shown in Table 5.6.1. Tubes $T_0$, $T_1$ and $T_2$ are subsequently the first, second and third parameters, and are set to empty tubes.

**Procedure X-OR-ZERO**($T_0$, $T_1$, $T_2$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) Append-head($T_1$, $R_2^0$).
(4) Append-head($T_2$, $R_2^0$).
(5) Append-head($T_1$, $C_1^1$).
(6) Append-head($T_2$, $C_1^0$).
(7) $T_0 = \cup(T_1, T_2)$.
**EndProcedure**

**Proof of Correction**:

The algorithm, **X-OR-ZERO**($T_0$, $T_1$, $T_2$), is implemented by means of the *append-head* and *merge* operations. Steps (1) through (4) apply the *append-head* operations to yield two different inputs to $x \vee 0$ as shown in Table 5.6.1. This indicates that two different inputs for $x \vee 0$ as shown in Table 5.6.1 were poured into tubes $T_1$ and $T_2$, respectively. Next, Step (5) uses the *append-head* operation to append $C_1^1$ onto the head of every input in tube $T_1$, and Step (6) applies the *append-head* operation to append $C_1^0$ onto the head of every input in tube $T_2$. This is to say that two different outputs to $x \vee 0$ as shown in Table 5.6.1 are appended into tubes $T_1$ and $T_2$. Finally, the execution of Step (7) uses the *merge* operation to pour tubes $T_1$ and $T_2$ into tube $T_0$. Tube $T_0$ consists of the result implementing, $x \vee 0$, as shown in Table 5.6.1. ■

**5.2:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are employed to encode the first input and the second input for $x \wedge 1$ as shown in Table 5.6.2. Also it is assumed that a binary number of a bit, $C_1$, is used to encode the output for the logical operation, $x \wedge 1$. For the sake of convenience, it is assumed that for $1 \leq k \leq 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also supposed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is proposed to implement, $x \wedge 1$, as shown in Table 5.6.2. Tubes $T_0$, $T_1$ and $T_2$ are subsequently the first, second and third parameters, and are set to empty tubes.

**Procedure X-AND-ONE**($T_0$, $T_1$, $T_2$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) Append-head($T_1$, $R_2^1$).
(4) Append-head($T_2$, $R_2^1$).
(5) Append-head($T_1$, $C_1^1$).
(6) Append-head($T_2$, $C_1^0$).
(7) $T_0 = \cup(T_1, T_2)$.
**EndProcedure**

**Proof of Correction**:

The algorithm, **X-AND-ONE**($T_0$, $T_1$, $T_2$), is implemented by means of the *append-head* and *merge* operations. Steps (1) through (4) apply the *append-head* operations to produce two different inputs to $x \wedge 1$ as shown in Table 5.6.2. This is to say that two different inputs for $x \wedge 1$ as shown in Table 5.6.2 were poured into tubes $T_1$ and $T_2$, respectively. Next, Step (5) applies the *append-head* operation to append $C_1^1$ onto the head of every input in tube $T_1$, and Step (6) uses the *append-head* operation to append $C_1^0$ onto the head of every input in tube $T_2$. This implies that two different outputs to $x \wedge 1$ as shown in Table 5.6.2 are appended into tubes $T_1$ and $T_2$. Finally, the execution of Step (7) uses the *merge* operation to pour tubes $T_1$ and $T_2$ into tube $T_0$. Tube $T_0$ includes the result implementing, $x \wedge 1$, as shown in Table 5.6.2. ■

**5.3:**

It is assumed that two binary numbers of a bit, $R_1$ and $R_2$, are used to encode the first input and the second input for $x \vee x'$ as shown in Table 5.6.3. It is also supposed

that a binary number of a bit, $C_1$, is applied to encode the output for the logical operation, $x \vee x'$. For the sake of convenience, it is supposed that for $1 \leq k \leq 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also assumed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is presented to implement, $x \vee x'$, as shown in Table 5.6.3. Tubes $T_0$, $T_1$ and $T_2$ are subsequently the first, second and third parameters, and are set to empty tubes.

**Procedure X-OR-NEGATIVE-X**($T_0$, $T_1$, $T_2$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) Append-head($T_1$, $R_2^0$).
(4) Append-head($T_2$, $R_2^1$).
(5) Append-head($T_1$, $C_1^1$).
(6) Append-head($T_2$, $C_1^1$).
(7) $T_0 = \cup(T_1, T_2)$.
**EndProcedure**

**Proof of Correction**:

The algorithm, **X-OR-NEGATIVE-X**($T_0$, $T_1$, $T_2$), is implemented by means of the *append-head* and *merge* operations. Steps (1) through (4) use the *append-head* operations to generate two different inputs to $x \vee x'$ as shown in Table 5.6.3. This indicates that two different inputs for $x \vee x'$ as shown in Table 5.6.3 were poured into tubes $T_1$ and $T_2$, respectively. Next, Step (5) uses the *append-head* operation to append $C_1^1$ onto the head of every input in tube $T_1$, and Step (6) also uses the *append-head* operation to append $C_1^1$ onto the head of every input in tube $T_2$. This is to say that two different outputs to $x \vee x'$ as shown in Table 5.6.3 are appended into tubes $T_1$ and $T_2$. Finally, the execution of Step (7) uses the *merge* operation to pour tubes $T_1$ and $T_2$ into tube $T_0$. Tube $T_0$ contains the result implementing, $x \vee x'$, as shown in Table 5.6.3. ■

**5.4:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are employed to encode the first input and the second input for $x \wedge x'$ as shown in Table 5.6.4. It is also assumed that a binary number of a bit, $C_1$, is used to encode the output for the logical operation, $x \wedge x'$. For the sake of convenience, it is assumed that for $1 \leq k \leq 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is

0. Similarly, it is also supposed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is offered to implement, $x \wedge x'$, as shown in Table 5.6.4. Tubes $T_0$, $T_1$ and $T_2$ are subsequently the first, second and third parameters, and are set to empty tubes.

**Procedure X-AND-NEGATIVE-X**($T_0$, $T_1$, $T_2$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) Append-head($T_1$, $R_2^0$).
(4) Append-head($T_2$, $R_2^1$).
(5) Append-head($T_1$, $C_1^0$).
(6) Append-head($T_2$, $C_1^0$).
(7) $T_0 = \cup(T_1, T_2)$.
**EndProcedure**

**Proof of Correction**:

The algorithm, **X-AND-NEGATIVE-X**($T_0$, $T_1$, $T_2$), is implemented by means of the *append-head* and *merge* operations. Steps (1) through (4) employ the *append-head* operations to yield two different inputs to $x \wedge x'$ as shown in Table 5.6.4. This implies that two different inputs for $x \wedge x'$ as shown in Table 5.6.4 were poured into tubes $T_1$ and $T_2$, respectively. Next, Step (5) applies the *append-head* operation to append $C_1^0$ onto the head of every input in tube $T_1$, and Step (6) also applies the *append-head* operation to append $C_1^0$ onto the head of every input in tube $T_2$. This indicates that two different outputs to $x \wedge x'$ as shown in Table 5.6.4 are appended into tubes $T_1$ and $T_2$. Finally, the execution of Step (7) uses the *merge* operation to pour tubes $T_1$ and $T_2$ into tube $T_0$. Tube $T_0$ consists of the result implementing, $x \wedge x'$, as shown in Table 5.6.4. ■

**5.5:**

It is assumed that two binary numbers of a bit, $R_1$ and $R_2$, are used to encode the first input and the second input for $x \vee x$ as shown in Table 5.6.5. It is also supposed that a binary number of a bit, $C_1$, is applied to encode the output for the logical operation, $x \vee x$. For the sake of convenience, it is supposed that for $1 \leq k \leq 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also assumed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is proposed to implement, $x \vee x$, as shown in Table 5.6.5. Tubes $T_0$, $T_1$ and $T_2$ are subsequently the

first, second and third parameters, and are set to empty tubes.

**Procedure X-OR-X**($T_0$, $T_1$, $T_2$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) Append-head($T_1$, $R_2^1$).
(4) Append-head($T_2$, $R_2^0$).
(5) Append-head($T_1$, $C_1^1$).
(6) Append-head($T_2$, $C_1^0$).
(7) $T_0 = \cup(T_1, T_2)$.
**EndProcedure**

**Proof of Correction**:

The algorithm, **X-OR-X**($T_0$, $T_1$, $T_2$), is implemented by means of the *append-head* and *merge* operations. Steps (1) through (4) use the *append-head* operations to produce two different inputs to $x \vee x$ as shown in Table 5.6.5. This is to say that two different inputs for $x \vee x$ as shown in Table 5.6.5 were poured into tubes $T_1$ and $T_2$, respectively. Next, Step (5) uses the *append-head* operation to append $C_1^1$ onto the head of every input in tube $T_1$, and Step (6) also uses the *append-head* operation to append $C_1^0$ onto the head of every input in tube $T_2$. This indicates that two different outputs to $x \vee x$ as shown in Table 5.6.5 are appended into tubes $T_1$ and $T_2$. Finally, the execution of Step (7) uses the *merge* operation to pour tubes $T_1$ and $T_2$ into tube $T_0$. Tube $T_0$ includes the result implementing, $x \vee x$, as shown in Table 5.6.5. ■

**5.6:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are employed to encode the first input and the second input for $x \wedge x$ as shown in Table 5.6.6. It is also assumed that a binary number of a bit, $C_1$, is employed to encode the output for the logical operation, $x \wedge x$. For the sake of convenience, it is assumed that for $1 \leq k \leq 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also supposed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is presented to implement, $x \wedge x$, as shown in Table 5.6.6. Tubes $T_0$, $T_1$ and $T_2$ are subsequently the first, second and third parameters, and are set to empty tubes.

**Procedure X-AND-X**($T_0$, $T_1$, $T_2$)
(1) Append-head($T_1$, $R_1^1$).

(2) Append-head($T_2$, $R_1^0$).
(3) Append-head($T_1$, $R_2^1$).
(4) Append-head($T_2$, $R_2^0$).
(5) Append-head($T_1$, $C_1^1$).
(6) Append-head($T_2$, $C_1^0$).
(7) $T_0 = \cup(T_1, T_2)$.
**EndProcedure**

**Proof of Correction**:

The algorithm, **X-AND-X**($T_0$, $T_1$, $T_2$), is implemented by means of the *append-head* and *merge* operations. Steps (1) through (4) apply the *append-head* operations to generate two different inputs to $x \wedge x$ as shown in Table 5.6.6. This indicates that two different inputs for $x \wedge x$ as shown in Table 5.6.6 were poured into tubes $T_1$ and $T_2$, respectively. Next, Step (5) applies the *append-head* operation to append $C_1^1$ onto the head of every input in tube $T_1$, and Step (6) also applies the *append-head* operation to append $C_1^0$ onto the head of every input in tube $T_2$. This implies that two different outputs to $x \wedge x$ as shown in Table 5.6.6 are appended into tubes $T_1$ and $T_2$. Finally, the execution of Step (7) uses the *merge* operation to pour tubes $T_1$ and $T_2$ into tube $T_0$. Tube $T_0$ contains the result implementing, $x \wedge x$, as shown in Table 5.6.6. ■

**5.7:**

It is assumed that two binary numbers of a bit, $R_1$ and $R_2$, are used to encode the first input and the second input for $x \vee 1$ as shown in Table 5.6.7. It is also supposed that a binary number of a bit, $C_1$, is used to encode the output for the logical operation, $x \vee 1$. For the sake of convenience, it is supposed that for $1 \leq k \leq 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also assumed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is proposed to implement, $x \vee 1$, as shown in Table 5.6.7. Tubes $T_0$, $T_1$ and $T_2$ are subsequently the first, second and third parameters, and are set to empty tubes.

**Procedure X-OR-ONE**($T_0$, $T_1$, $T_2$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) Append-head($T_1$, $R_2^1$).
(4) Append-head($T_2$, $R_2^1$).

(5) Append-head($T_1$, $C_1^1$).
(6) Append-head($T_2$, $C_1^1$).
(7) $T_0 = \cup(T_1, T_2)$.
**EndProcedure**

**Proof of Correction**:

The algorithm, **X-OR-ONE**($T_0$, $T_1$, $T_2$), is implemented by means of the *append-head* and *merge* operations. Steps (1) through (4) use the *append-head* operations to yield two different inputs to $x \vee 1$ as shown in Table 5.6.7. This implies that two different inputs for $x \vee 1$ as shown in Table 5.6.7 were poured into tubes $T_1$ and $T_2$, respectively. Next, Step (5) applies the *append-head* operation to append $C_1^1$ onto the head of every input in tube $T_1$, and Step (6) also applies the *append-head* operation to append $C_1^1$ onto the head of every input in tube $T_2$. This is to say that two different outputs to $x \vee 1$ as shown in Table 5.6.7 are appended into tubes $T_1$ and $T_2$. Finally, the execution of Step (7) uses the *merge* operation to pour tubes $T_1$ and $T_2$ into tube $T_0$. Tube $T_0$ consists of the result implementing, $x \vee 1$, as shown in Table 5.6.7. ■

**5.8:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are employed to encode the first input and the second input for $x \wedge 0$ as shown in Table 5.6.8. It is also assumed that a binary number of a bit, $C_1$, is applied to encode the output for the logical operation, $x \wedge 0$. For the sake of convenience, it is assumed that for $1 \leq k \leq 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also supposed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is offered to implement, $x \wedge 0$, as shown in Table 5.6.8. Tubes $T_0$, $T_1$ and $T_2$ are subsequently the first, second and third parameters, and are set to empty tubes.

**Procedure X-AND-ZERO**($T_0$, $T_1$, $T_2$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) Append-head($T_1$, $R_2^0$).
(4) Append-head($T_2$, $R_2^0$).
(5) Append-head($T_1$, $C_1^0$).
(6) Append-head($T_2$, $C_1^0$).
(7) $T_0 = \cup(T_1, T_2)$.

**EndProcedure**

**Proof of Correction**:

The algorithm, **X-AND-ZERO**($T_0$, $T_1$, $T_2$), is implemented by means of the *append-head* and *merge* operations. Steps (1) through (4) use the *append-head* operations to produce two different inputs to $x \wedge 0$ as shown in Table 5.6.8. This is to say that two different inputs for $x \wedge 0$ as shown in Table 5.6.8 were poured into tubes $T_1$ and $T_2$, respectively. Next, Step (5) uses the *append-head* operation to append $C_1^{\,0}$ onto the head of every input in tube $T_1$, and Step (6) also uses the *append-head* operation to append $C_1^{\,0}$ onto the head of every input in tube $T_2$. This indicates that two different outputs to $x \wedge 0$ as shown in Table 5.6.8 are appended into tubes $T_1$ and $T_2$. Finally, the execution of Step (7) uses the *merge* operation to pour tubes $T_1$ and $T_2$ into tube $T_0$. Tube $T_0$ includes the result implementing, $x \wedge 0$, as shown in Table 5.6.8. ■

**5.9:**

It is assumed that two binary numbers of a bit, $R_1$ and $R_2$, are applied to encode the first input and the second input for $(x')'$ as shown in Table 5.6.9. It is also supposed that a binary number of a bit, $C_1$, is employed to encode the output for the logical operation, $(x')'$. For the sake of convenience, it is supposed that for $1 \leq k \leq 2$ $R_k^{\,1}$ denotes the fact that the value of $R_k$ is 1 and $R_k^{\,0}$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also assumed that $C_1^{\,1}$ denotes the fact that the value of $C_1$ is 1 and $C_1^{\,0}$ denotes the fact that the value of $C_1$ is 0. The following algorithm is presented to implement, $(x')'$, as shown in Table 5.6.9. Tubes $T_0$, $T_1$ and $T_2$ are subsequently the first, second and third parameters, and are set to empty tubes.

**Procedure X-NEGATIVE-NEGATIVE**($T_0$, $T_1$, $T_2$)
(1) Append-head($T_1$, $R_1^{\,1}$).
(2) Append-head($T_2$, $R_1^{\,0}$).
(3) Append-head($T_1$, $R_2^{\,0}$).
(4) Append-head($T_2$, $R_2^{\,1}$).
(5) Append-head($T_1$, $C_1^{\,1}$).
(6) Append-head($T_2$, $C_1^{\,0}$).
(7) $T_0 = \cup(T_1, T_2)$.
**EndProcedure**

**Proof of Correction**:

The algorithm, **X-NEGATIVE-NEGATIVE**($T_0$, $T_1$, $T_2$), is implemented by means of the *append-head* and *merge* operations. Steps (1) through (4) apply the *append-head* operations to generate two different inputs to $(x')'$ as shown in Table 5.6.9. This indicates that two different inputs for $(x')'$ as shown in Table 5.6.9 were poured into tubes $T_1$ and $T_2$, respectively. Next, Step (5) applies the *append-head* operation to append $C_1^{\,1}$ onto the head of every input in tube $T_1$, and Step (6) also uses the *append-head* operation to append $C_1^{\,0}$ onto the head of every input in tube $T_2$. This is to say that two different outputs to $(x')'$ as shown in Table 5.6.9 are appended into tubes $T_1$ and $T_2$. Finally, the execution of Step (7) uses the *merge* operation to pour tubes $T_1$ and $T_2$ into tube $T_0$. Tube $T_0$ contains the result implementing, $(x')'$, as shown in Table 5.6.9. ■

**5.10:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are applied to encode the first input and the second input for $x \wedge y'$ as shown in Table 5.6.10. Also it is assumed that a binary number of a bit, $C_1$, is employed to encode the output for $x \wedge y'$. For the sake of convenience, it is supposed that for $1 \le k \le 2$ $R_k^{\,1}$ denotes the fact that the value of $R_k$ is 1 and $R_k^{\,0}$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also assumed that $C_1^{\,1}$ denotes the fact that the value of $C_1$ is 1 and $C_1^{\,0}$ denotes the fact that the value of $C_1$ is 0. The following algorithm is proposed to implement $x \wedge y'$ as shown in Table 5.6.10. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes.

**Procedure X-AND-NEGATIVE-Y**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$)
(1) Append-head($T_1$, $R_1^{\,1}$).
(2) Append-head($T_2$, $R_1^{\,0}$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 2
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, $R_k^{\,1}$).
    (4c) Append-head($T_2$, $R_k^{\,0}$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
(5) $T_1 = +(T_0, R_1^{\,1})$ and $T_2 = -(T_0, R_1^{\,1})$.
(6) $T_3 = +(T_1, R_2^{\,1})$ and $T_4 = -(T_1, R_2^{\,1})$.
(7) $T_5 = +(T_2, R_2^{\,1})$ and $T_6 = -(T_2, R_2^{\,1})$.
(8) Append-head($T_3$, $C_1^{\,0}$).

(9) Append-head($T_4$, $C_1^1$).
(10) Append-head($T_5$, $C_1^0$).
(11) Append-head($T_6$, $C_1^0$).
(12) $T_0 = \cup(T_3, T_4, T_5, T_6)$.
**EndProcedure**

**Proof of Correction:**

The algorithm, **X-AND-NEGATIVE-Y**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$), is implemented by means of the *extract*, *amplify*, *append-head* and *merge* operations. Steps (1) through (4d) use the *amplify*, *append-head* and *merge* operations to generate four different inputs to $x \wedge y'$ as shown in Table 5.6.10. Next, Steps (5) through (7) apply the *extract* operations to form some different tubes containing different inputs ($T_1$ to $T_6$). $T_1$ contains all of the inputs that have $R_1 = 1$, $T_2$ contains all of the inputs that have $R_1 = 0$, $T_3$ contains that input that has $R_1 = 1$ and $R_2 = 1$, $T_4$ contains that input that has $R_1 = 1$ and $R_2 = 0$, $T_5$ contains that input that has $R_1 = 0$ and $R_2 = 1$, and finally, $T_6$ contains that input that has $R_1 = 0$ and $R_2 = 0$. This is to say that four different inputs for $x \wedge y'$ as shown in Table 5.6.10 were poured into tubes $T_3$ through $T_6$, respectively. Next, Steps (8) through (11) employ the *append-head* operations to append $C_1^0$ and $C_1^1$ onto the head of every input in the corresponding tubes. This implies that four different outputs to $x \wedge y'$ as shown in Table 5.6.10 are appended into tubes $T_3$ through $T_6$. Finally, the execution of Step (12) uses the *merge* operation to pour tubes $T_3$ through $T_6$ into tube $T_0$. Tube $T_0$ consists of the result implementing, $x \wedge y'$, as shown in Table 5.6.10. ■

## Chapter 6

### 6.1:

It is assumed that two binary numbers of a bit, $R_1$ and $R_2$, are employed to encode the first input and the second input for $x \vee (x \wedge y)$ as shown in Table 6.10.1. Also it is supposed that a binary number of a bit, $C_1$, is used to encode the output for $x \vee (x \wedge y)$. For the sake of convenience, it is assumed that for $1 \leq k \leq 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also supposed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is offered to implement, $x \vee (x \wedge y)$, as shown in Table 6.10.1. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes.

**Procedure X-OR-X-AND-Y**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 2
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, $R_k^1$).
    (4c) Append-head($T_2$, $R_k^0$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
(5) $T_1 = +(T_0, R_1^1)$ and $T_2 = -(T_0, R_1^1)$.
(6) $T_3 = +(T_1, R_2^1)$ and $T_4 = -(T_1, R_2^1)$.
(7) $T_5 = +(T_2, R_2^1)$ and $T_6 = -(T_2, R_2^1)$.
(8) Append-head($T_3$, $C_1^1$).
(9) Append-head($T_4$, $C_1^1$).
(10) Append-head($T_5$, $C_1^0$).
(11) Append-head($T_6$, $C_1^0$).
(12) $T_0 = \cup(T_3, T_4, T_5, T_6)$.
**EndProcedure**

**Proof of Correction:**

The algorithm, **X-OR-X-AND-Y**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$), is implemented by

means of the *extract*, *amplify*, *append-head* and *merge* operations. Steps (1) through (4d) use the *amplify*, *append-head* and *merge* operations to yield four different inputs to $x \vee (x \wedge y)$ as shown in Table 6.10.1. Next, Steps (5) through (7) use the *extract* operations to produce some different tubes including different inputs ($T_1$ to $T_6$). $T_1$ includes all of the inputs that have $R_1 = 1$, $T_2$ includes all of the inputs that have $R_1 = 0$, $T_3$ includes that input that has $R_1 = 1$ and $R_2 = 1$, $T_4$ includes that input that has $R_1 = 1$ and $R_2 = 0$, $T_5$ includes that input that has $R_1 = 0$ and $R_2 = 1$, and finally, $T_6$ includes that input that has $R_1 = 0$ and $R_2 = 0$. This indicates that four different inputs for $x \vee (x \wedge y)$ as shown in Table 6.10.1 were poured into tubes $T_3$ through $T_6$, respectively. Next, Steps (8) through (11) apply the *append-head* operations to append $C_1^1$ and $C_1^0$ onto the head of every input in the corresponding tubes. This is to say that four different outputs to $x \vee (x \wedge y)$ as shown in Table 6.10.1 are appended into tubes $T_3$ through $T_6$. Finally, the execution of Step (12) applies the *merge* operation to pour tubes $T_3$ through $T_6$ into tube $T_0$. Tube $T_0$ contains the result implementing, $x \vee (x \wedge y)$, as shown in Table 6.10.1. ■

**6.2:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are used to encode the first input and the second input for $x \wedge (x \vee y)$ as shown in Table 6.10.2. Also it is assumed that a binary number of a bit, $C_1$, is applied to encode the output for $x \wedge (x \vee y)$. For the sake of convenience, it is supposed that for $1 \leq k \leq 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also assumed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is presented to implement, $x \wedge (x \vee y)$, as shown in Table 6.10.2. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes.

**Procedure X-AND-X-OR-Y**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 2
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, $R_k^1$).
    (4c) Append-head($T_2$, $R_k^0$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**

(5) $T_1 = +(T_0, R_1^1)$ and $T_2 = -(T_0, R_1^1)$.
(6) $T_3 = +(T_1, R_2^1)$ and $T_4 = -(T_1, R_2^1)$.
(7) $T_5 = +(T_2, R_2^1)$ and $T_6 = -(T_2, R_2^1)$.
(8) Append-head($T_3$, $C_1^1$).
(9) Append-head($T_4$, $C_1^1$).
(10) Append-head($T_5$, $C_1^0$).
(11) Append-head($T_6$, $C_1^0$).
(12) $T_0 = \cup(T_3, T_4, T_5, T_6)$.
**EndProcedure**

**Proof of Correction:**

The algorithm, **X-AND-X-OR-Y**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$), is implemented by means of the *extract*, *amplify*, *append-head* and *merge* operations. Steps (1) through (4d) use the *amplify*, *append-head* and *merge* operations to generate four different inputs to $x \wedge (x \vee y)$ as shown in Table 6.10.2. Next, Steps (5) through (7) apply the *extract* operations to yield some different tubes containing different inputs ($T_1$ to $T_6$). $T_1$ consists of all of the inputs that have $R_1 = 1$, $T_2$ consists of all of the inputs that have $R_1 = 0$, $T_3$ consists of that input that has $R_1 = 1$ and $R_2 = 1$, $T_4$ consists of that input that has $R_1 = 1$ and $R_2 = 0$, $T_5$ consists of that input that has $R_1 = 0$ and $R_2 = 1$, and finally, $T_6$ consists of that input that has $R_1 = 0$ and $R_2 = 0$. This implies that four different inputs for $x \wedge (x \vee y)$ as shown in Table 6.10.2 were poured into tubes $T_3$ through $T_6$, respectively. Next, Steps (8) through (11) employ the *append-head* operations to append $C_1^1$ and $C_1^0$ onto the head of every input in the corresponding tubes. This indicates that four different outputs to $x \wedge (x \vee y)$ as shown in Table 6.10.2 are appended into tubes $T_3$ through $T_6$. Finally, the execution of Step (12) uses the *merge* operation to pour tubes $T_3$ through $T_6$ into tube $T_0$. Tube $T_0$ includes the result implementing, $x \wedge (x \vee y)$, as shown in Table 6.10.2. ■

**6.3:**

It is assumed that two binary numbers of a bit, $R_1$ and $R_2$, are applied to encode the first input and the second input for $y \vee (y \wedge x)$ as shown in Table 6.10.3. Also it is supposed that a binary number of a bit, $C_1$, is employed to encode the output for $y \vee (y \wedge x)$. For the sake of convenience, it is assumed that for $1 \leq k \leq 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also supposed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is proposed to implement, $y \vee (y \wedge x)$, as shown in Table 6.10.3. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently

the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes.

**Procedure Y-OR-Y-AND-X**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 2
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, $R_k^1$).
    (4c) Append-head($T_2$, $R_k^0$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
(5) $T_1 = +(T_0, R_1^1)$ and $T_2 = -(T_0, R_1^1)$.
(6) $T_3 = +(T_1, R_2^1)$ and $T_4 = -(T_1, R_2^1)$.
(7) $T_5 = +(T_2, R_2^1)$ and $T_6 = -(T_2, R_2^1)$.
(8) Append-head($T_3$, $C_1^1$).
(9) Append-head($T_4$, $C_1^0$).
(10) Append-head($T_5$, $C_1^1$).
(11) Append-head($T_6$, $C_1^0$).
(12) $T_0 = \cup(T_3, T_4, T_5, T_6)$.
**EndProcedure**

**Proof of Correction:**

The algorithm, **Y-OR-Y-AND-X**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$), is implemented by means of the *extract*, *amplify*, *append-head* and *merge* operations. Steps (1) through (4d) use the *amplify*, *append-head* and *merge* operations to produce four different inputs to $y \vee (y \wedge x)$ as shown in Table 6.10.3. Next, Steps (5) through (7) use the *extract* operations to generate some different tubes consisting of different inputs ($T_1$ to $T_6$). $T_1$ contains all of the inputs that have $R_1 = 1$, $T_2$ contains all of the inputs that have $R_1 = 0$, $T_3$ contains that input that has $R_1 = 1$ and $R_2 = 1$, $T_4$ contains that input that has $R_1 = 1$ and $R_2 = 0$, $T_5$ contains that input that has $R_1 = 0$ and $R_2 = 1$, and finally, $T_6$ contains that input that has $R_1 = 0$ and $R_2 = 0$. This is to say that four different inputs for $y \vee (y \wedge x)$ as shown in Table 6.10.3 were poured into tubes $T_3$ through $T_6$, respectively. Next, Steps (8) through (11) apply the *append-head* operations to append $C_1^1$ and $C_1^0$ onto the head of every input in the corresponding tubes. This is to say that four different outputs to $y \vee (y \wedge x)$ as shown in Table 6.10.3 are appended into tubes

$T_3$ through $T_6$. Finally, the execution of Step (12) applies the *merge* operation to pour tubes $T_3$ through $T_6$ into tube $T_0$. Tube $T_0$ consists of the result implementing, $y \vee (y \wedge x)$, as shown in Table 6.10.3. ■

**6.4:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are used to encode the first input and the second input for $y \wedge (y \vee x)$ as shown in Table 6.10.4. Also it is assumed that a binary number of a bit, $C_1$, is used to encode the output for $y \wedge (y \vee x)$. For the sake of convenience, it is supposed that for $1 \leq k \leq 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also assumed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is offered to implement, $y \wedge (y \vee x)$, as shown in Table 6.10.4. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes.

**Procedure Y-AND-Y-OR-X**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 2
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, $R_k^1$).
    (4c) Append-head($T_2$, $R_k^0$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
(5) $T_1 = +(T_0, R_1^1)$ and $T_2 = -(T_0, R_1^1)$.
(6) $T_3 = +(T_1, R_2^1)$ and $T_4 = -(T_1, R_2^1)$.
(7) $T_5 = +(T_2, R_2^1)$ and $T_6 = -(T_2, R_2^1)$.
(8) Append-head($T_3$, $C_1^1$).
(9) Append-head($T_4$, $C_1^0$).
(10) Append-head($T_5$, $C_1^1$).
(11) Append-head($T_6$, $C_1^0$).
(12) $T_0 = \cup(T_3, T_4, T_5, T_6)$.
**EndProcedure**

**Proof of Correction:**

The algorithm, **Y-AND-Y-OR-X**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$), is implemented by

means of the *extract*, *amplify*, *append-head* and *merge* operations. Steps (1) through (4d) apply the *amplify*, *append-head* and *merge* operations to yield four different inputs to $y \wedge (y \vee x)$ as shown in Table 6.10.4. Next, Steps (5) through (7) apply the *extract* operations to produce some different tubes including different inputs ($T_1$ to $T_6$). $T_1$ includes all of the inputs that have $R_1 = 1$, $T_2$ includes all of the inputs that have $R_1 = 0$, $T_3$ includes that input that has $R_1 = 1$ and $R_2 = 1$, $T_4$ includes that input that has $R_1 = 1$ and $R_2 = 0$, $T_5$ includes that input that has $R_1 = 0$ and $R_2 = 1$, and finally, $T_6$ includes that input that has $R_1 = 0$ and $R_2 = 0$. This implies that four different inputs for $y \wedge (y \vee x)$ as shown in Table 6.10.4 were poured into tubes $T_3$ through $T_6$, respectively. Next, Steps (8) through (11) use the *append-head* operations to append ${C_1}^1$ and ${C_1}^0$ onto the head of every input in the corresponding tubes. This implies that four different outputs to $y \wedge (y \vee x)$ as shown in Table 6.10.4 are appended into tubes $T_3$ through $T_6$. Finally, the execution of Step (12) uses the *merge* operation to pour tubes $T_3$ through $T_6$ into tube $T_0$. Tube $T_0$ includes the result implementing, $y \wedge (y \vee x)$, as shown in Table 6.10.4. ■

**6.5:**

It is assumed that two binary numbers of a bit, $R_1$ and $R_2$, are applied to encode the first input and the second input for $x' \wedge y$ as shown in Table 6.10.5. Also it is supposed that a binary number of a bit, $C_1$, is employed to encode the output for $x' \wedge y$. For the sake of convenience, it is assumed that for $1 \le k \le 2$ ${R_k}^1$ denotes the fact that the value of $R_k$ is 1 and ${R_k}^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also supposed that ${C_1}^1$ denotes the fact that the value of $C_1$ is 1 and ${C_1}^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is proposed to implement, $x' \wedge y$, as shown in Table 6.10.5. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes.

**Procedure NEGATIVE-X-AND-Y**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$)
(1) Append-head($T_1$, ${R_1}^1$).
(2) Append-head($T_2$, ${R_1}^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 2
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, ${R_k}^1$).
    (4c) Append-head($T_2$, ${R_k}^0$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
(5) $T_1 = +(T_0, {R_1}^1)$ and $T_2 = -(T_0, {R_1}^1)$.

(6) $T_3 = +(T_1, R_2^1)$ and $T_4 = -(T_1, R_2^1)$.
(7) $T_5 = +(T_2, R_2^1)$ and $T_6 = -(T_2, R_2^1)$.
(8) Append-head($T_3$, $C_1^0$).
(9) Append-head($T_4$, $C_1^0$).
(10) Append-head($T_5$, $C_1^1$).
(11) Append-head($T_6$, $C_1^0$).
(12) $T_0 = \cup(T_3, T_4, T_5, T_6)$.
**EndProcedure**

**Proof of Correction:**

The algorithm, **NEGATIVE-X-AND-Y**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$), is implemented by means of the *extract*, *amplify*, *append-head* and *merge* operations. Steps (1) through (4d) employ the *amplify*, *append-head* and *merge* operations to produce four different inputs to $x' \wedge y$ as shown in Table 6.10.5. Next, Steps (5) through (7) use the *extract* operations to generate some different tubes containing different inputs ($T_1$ to $T_6$). $T_1$ contains all of the inputs that have $R_1 = 1$, $T_2$ contains all of the inputs that have $R_1 = 0$, $T_3$ contains that input that has $R_1 = 1$ and $R_2 = 1$, $T_4$ contains that input that has $R_1 = 1$ and $R_2 = 0$, $T_5$ contains that input that has $R_1 = 0$ and $R_2 = 1$, and finally, $T_6$ contains that input that has $R_1 = 0$ and $R_2 = 0$. This is to say that four different inputs for $x' \wedge y$ as shown in Table 6.10.5 were poured into tubes $T_3$ through $T_6$, respectively. Next, Steps (8) through (11) apply the *append-head* operations to append $C_1^0$ and $C_1^1$ onto the head of every input in the corresponding tubes. This indicates that four different outputs to $x' \wedge y$ as shown in Table 6.10.5 are appended into tubes $T_3$ through $T_6$. Finally, the execution of Step (12) applies the *merge* operation to pour tubes $T_3$ through $T_6$ into tube $T_0$. Tube $T_0$ consists of the result implementing, $x' \wedge y$, as shown in Table 6.10.5. ■

**6.6:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are employed to encode the first input and the second input for $(x \wedge y) \vee (x' \wedge y')$ as shown in Table 6.10.6. Also it is assumed that a binary number of a bit, $C_1$, is applied to encode the output for $(x \wedge y) \vee (x' \wedge y')$. For the sake of convenience, it is supposed that for $1 \leq k \leq 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also assumed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is presented to implement, $(x \wedge y) \vee (x' \wedge y')$, as shown in Table 6.10.6. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters,

and are set to empty tubes.

**Procedure X-AND-Y-OR-NEGATIVE-X-AND-NEGATIVE-Y**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 2
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, $R_k^1$).
    (4c) Append-head($T_2$, $R_k^0$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
(5) $T_1 = +(T_0, R_1^1)$ and $T_2 = -(T_0, R_1^1)$.
(6) $T_3 = +(T_1, R_2^1)$ and $T_4 = -(T_1, R_2^1)$.
(7) $T_5 = +(T_2, R_2^1)$ and $T_6 = -(T_2, R_2^1)$.
(8) Append-head($T_3$, $C_1^1$).
(9) Append-head($T_4$, $C_1^0$).
(10) Append-head($T_5$, $C_1^0$).
(11) Append-head($T_6$, $C_1^1$).
(12) $T_0 = \cup(T_3, T_4, T_5, T_6)$.
**EndProcedure**

**Proof of Correction:**

The algorithm, **X-AND-Y-OR-NEGATIVE-X-AND-NEGATIVE-Y**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$), is implemented by means of the *extract*, *amplify*, *append-head* and *merge* operations. Steps (1) through (4d) use the *amplify*, *append-head* and *merge* operations to yield four different inputs to $(x \wedge y) \vee (x' \wedge y')$ as shown in Table 6.10.6. Next, Steps (5) through (7) employ the *extract* operations to form some different tubes consisting of different inputs ($T_1$ to $T_6$). $T_1$ consists of all of the inputs that have $R_1 = 1$, $T_2$ consists of all of the inputs that have $R_1 = 0$, $T_3$ consists of that input that has $R_1 = 1$ and $R_2 = 1$, $T_4$ consists of that input that has $R_1 = 1$ and $R_2 = 0$, $T_5$ consists of that input that has $R_1 = 0$ and $R_2 = 1$, and finally, $T_6$ consists of that input that has $R_1 = 0$ and $R_2 = 0$. This indicates that four different inputs for $(x \wedge y) \vee (x' \wedge y')$ as shown in Table 6.10.6 were poured into tubes $T_3$ through $T_6$, respectively. Next, Steps (8) through (11) use the *append-head* operations to append $C_1^1$ and $C_1^0$ onto the head of every input in the corresponding tubes. This implies that four different outputs to $(x \wedge$

$y) \vee (x' \wedge y')$ as shown in Table 6.10.6 are appended into tubes $T_3$ through $T_6$. Finally, the execution of Step (12) uses the *merge* operation to pour tubes $T_3$ through $T_6$ into tube $T_0$. Tube $T_0$ includes the result implementing, $(x \wedge y) \vee (x' \wedge y')$, as shown in Table 6.10.6. ■

**6.7:**

It is assumed that two binary numbers of a bit, $R_1$ and $R_2$, are used to encode the first input and the second input for $y'$ as shown in Table 6.10.7. Also it is supposed that a binary number of a bit, $C_1$, is used to encode the output for $y'$. For the sake of convenience, it is assumed that for $1 \leq k \leq 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also supposed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is proposed to implement, $y'$, as shown in Table 6.10.7. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes.

**Procedure NEGATIVE-Y**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 2
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, $R_k^1$).
    (4c) Append-head($T_2$, $R_k^0$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
(5) $T_1 = +(T_0, R_1^1)$ and $T_2 = -(T_0, R_1^1)$.
(6) $T_3 = +(T_1, R_2^1)$ and $T_4 = -(T_1, R_2^1)$.
(7) $T_5 = +(T_2, R_2^1)$ and $T_6 = -(T_2, R_2^1)$.
(8) Append-head($T_3$, $C_1^0$).
(9) Append-head($T_4$, $C_1^1$).
(10) Append-head($T_5$, $C_1^0$).
(11) Append-head($T_6$, $C_1^1$).
(12) $T_0 = \cup(T_3, T_4, T_5, T_6)$.
**EndProcedure**

**Proof of Correction:**

The algorithm, **NEGATIVE-Y**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$), is implemented by means of the *extract*, *amplify*, *append-head* and *merge* operations. Steps (1) through (4d) apply the *amplify*, *append-head* and *merge* operations to produce four different inputs to $y'$ as shown in Table 6.10.7. Next, Steps (5) through (7) use the *extract* operations to generate some different tubes including different inputs ($T_1$ to $T_6$). $T_1$ includes all of the inputs that have $R_1 = 1$, $T_2$ includes all of the inputs that have $R_1 = 0$, $T_3$ includes that input that has $R_1 = 1$ and $R_2 = 1$, $T_4$ includes that input that has $R_1 = 1$ and $R_2 = 0$, $T_5$ includes that input that has $R_1 = 0$ and $R_2 = 1$, and finally, $T_6$ includes that input that has $R_1 = 0$ and $R_2 = 0$. This is to say that four different inputs for $y'$ as shown in Table 6.10.7 were poured into tubes $T_3$ through $T_6$, respectively. Next, Steps (8) through (11) apply the *append-head* operations to append $C_1^0$ and $C_1^1$ onto the head of every input in the corresponding tubes. This indicates that four different outputs to $y'$ as shown in Table 6.10.7 are appended into tubes $T_3$ through $T_6$. Finally, the execution of Step (12) applies the *merge* operation to pour tubes $T_3$ through $T_6$ into tube $T_0$. Tube $T_0$ contains the result implementing, $y'$, as shown in Table 6.10.7. ■

**6.8:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are employed to encode the first input and the second input for $x'$ as shown in Table 6.10.8. Also it is assumed that a binary number of a bit, $C_1$, is applied to encode the output for $x'$. For the sake of convenience, it is supposed that for $1 \le k \le 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also assumed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is offered to implement, $x'$, as shown in Table 6.10.8. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes.

**Procedure NEGATIVE-X**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 2
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, $R_k^1$).
    (4c) Append-head($T_2$, $R_k^0$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
(5) $T_1 = +(T_0, R_1^1)$ and $T_2 = -(T_0, R_1^1)$.

(6) $T_3 = +(T_1, R_2^1)$ and $T_4 = -(T_1, R_2^1)$.
(7) $T_5 = +(T_2, R_2^1)$ and $T_6 = -(T_2, R_2^1)$.
(8) Append-head($T_3$, $C_1^0$).
(9) Append-head($T_4$, $C_1^0$).
(10) Append-head($T_5$, $C_1^1$).
(11) Append-head($T_6$, $C_1^1$).
(12) $T_0 = \cup(T_3, T_4, T_5, T_6)$.
**EndProcedure**

**Proof of Correction:**

The algorithm, **NEGATIVE-X**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$), is implemented by means of the *extract*, *amplify*, *append-head* and *merge* operations. Steps (1) through (4d) use the *amplify*, *append-head* and *merge* operations to yield four different inputs to $x'$ as shown in Table 6.10.8. Next, Steps (5) through (7) apply the *extract* operations to produce some different tubes containing different inputs ($T_1$ to $T_6$). $T_1$ contains all of the inputs that have $R_1 = 1$, $T_2$ contains all of the inputs that have $R_1 = 0$, $T_3$ contains that input that has $R_1 = 1$ and $R_2 = 1$, $T_4$ contains that input that has $R_1 = 1$ and $R_2 = 0$, $T_5$ contains that input that has $R_1 = 0$ and $R_2 = 1$, and finally, $T_6$ contains that input that has $R_1 = 0$ and $R_2 = 0$. This indicates that four different inputs for $x'$ as shown in Table 6.10.8 were poured into tubes $T_3$ through $T_6$, respectively. Next, Steps (8) through (11) use the *append-head* operations to append $C_1^0$ and $C_1^1$ onto the head of every input in the corresponding tubes. This is to say that four different outputs to $x'$ as shown in Table 6.10.8 are appended into tubes $T_3$ through $T_6$. Finally, the execution of Step (12) uses the *merge* operation to pour tubes $T_3$ through $T_6$ into tube $T_0$. Tube $T_0$ consists of the result implementing, $x'$, as shown in Table 6.10.8. ■

**6.9:**

It is assumed that two binary numbers of a bit, $R_1$ and $R_2$, are applied to encode the first input and the second input for $x \vee y'$ as shown in Table 6.10.9. Also it is supposed that a binary number of a bit, $C_1$, is employed to encode the output for $x \vee y'$. For the sake of convenience, it is assumed that for $1 \le k \le 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also supposed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is presented to implement, $x \vee y'$, as shown in Table 6.10.9. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes.

**Procedure X-OR-NEGATIVE-Y**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 2
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, $R_k^1$).
    (4c) Append-head($T_2$, $R_k^0$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
(5) $T_1 = +(T_0, R_1^1)$ and $T_2 = -(T_0, R_1^1)$.
(6) $T_3 = +(T_1, R_2^1)$ and $T_4 = -(T_1, R_2^1)$.
(7) $T_5 = +(T_2, R_2^1)$ and $T_6 = -(T_2, R_2^1)$.
(8) Append-head($T_3$, $C_1^1$).
(9) Append-head($T_4$, $C_1^1$).
(10) Append-head($T_5$, $C_1^0$).
(11) Append-head($T_6$, $C_1^1$).
(12) $T_0 = \cup(T_3, T_4, T_5, T_6)$.
**EndProcedure**

**Proof of Correction:**

The algorithm, **X-OR-NEGATIVE-Y**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$), is implemented by means of the *extract*, *amplify*, *append-head* and *merge* operations. Steps (1) through (4d) apply the *amplify*, *append-head* and *merge* operations to generate four different inputs to $x \vee y'$ as shown in Table 6.10.9. Next, Steps (5) through (7) use the *extract* operations to yield some different tubes including different inputs ($T_1$ to $T_6$). $T_1$ includes all of the inputs that have $R_1 = 1$, $T_2$ includes all of the inputs that have $R_1 = 0$, $T_3$ includes that input that has $R_1 = 1$ and $R_2 = 1$, $T_4$ includes that input that has $R_1 = 1$ and $R_2 = 0$, $T_5$ includes that input that has $R_1 = 0$ and $R_2 = 1$, and finally, $T_6$ includes that input that has $R_1 = 0$ and $R_2 = 0$. This is to say that four different inputs for $x \vee y'$ as shown in Table 6.10.9 were poured into tubes $T_3$ through $T_6$, respectively. Next, Steps (8) through (11) apply the *append-head* operations to append $C_1^1$ and $C_1^0$ onto the head of every input in the corresponding tubes. This implies that four different outputs to $x \vee y'$ as shown in Table 6.10.9 are appended into tubes $T_3$ through $T_6$. Finally, the execution of Step (12) applies the *merge* operation to pour tubes $T_3$ through $T_6$ into tube $T_0$. Tube $T_0$ contains the result implementing, $x \vee y'$, as shown in

Table 6.10.9. ■

**6.10:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are used to encode the first input and the second input for $x' \vee y$ as shown in Table 6.10.10. Also it is assumed that a binary number of a bit, $C_1$, is applied to encode the output for $x' \vee y$. For the sake of convenience, it is supposed that for $1 \leq k \leq 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also assumed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is proposed to implement, $x' \vee y$, as shown in Table 6.10.10. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes.

**Procedure NEGATIVE-X-OR-Y**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$)
(1) Append-head($T_1$, $R_1^1$).
(2) Append-head($T_2$, $R_1^0$).
(3) $T_0 = \cup(T_1, T_2)$.
(4) **For** $k = 2$ **to** 2
    (4a) Amplify($T_0$, $T_1$, $T_2$).
    (4b) Append-head($T_1$, $R_k^1$).
    (4c) Append-head($T_2$, $R_k^0$).
    (4d) $T_0 = \cup(T_1, T_2)$.
**EndFor**
(5) $T_1 = +(T_0, R_1^1)$ and $T_2 = -(T_0, R_1^1)$.
(6) $T_3 = +(T_1, R_2^1)$ and $T_4 = -(T_1, R_2^1)$.
(7) $T_5 = +(T_2, R_2^1)$ and $T_6 = -(T_2, R_2^1)$.
(8) Append-head($T_3$, $C_1^1$).
(9) Append-head($T_4$, $C_1^0$).
(10) Append-head($T_5$, $C_1^1$).
(11) Append-head($T_6$, $C_1^1$).
(12) $T_0 = \cup(T_3, T_4, T_5, T_6)$.
**EndProcedure**

**Proof of Correction:**

The algorithm, **NEGATIVE-X-OR-Y**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$), is implemented by means of the *extract*, *amplify*, *append-head* and *merge* operations. Steps (1) through (4d) use the *amplify*, *append-head* and *merge* operations to yield four

different inputs to $x' \vee y$ as shown in Table 6.10.10. Next, Steps (5) through (7) apply the *extract* operations to produce some different tubes containing different inputs ($T_1$ to $T_6$). $T_1$ consists of all of the inputs that have $R_1 = 1$, $T_2$ consists of all of the inputs that have $R_1 = 0$, $T_3$ consists of that input that has $R_1 = 1$ and $R_2 = 1$, $T_4$ consists of that input that has $R_1 = 1$ and $R_2 = 0$, $T_5$ consists of that input that has $R_1 = 0$ and $R_2 = 1$, and finally, $T_6$ consists of that input that has $R_1 = 0$ and $R_2 = 0$. This implies that four different inputs for $x' \vee y$ as shown in Table 6.10.10 were poured into tubes $T_3$ through $T_6$, respectively. Next, Steps (8) through (11) use the *append-head* operations to append ${C_1}^1$ and ${C_1}^0$ onto the head of every input in the corresponding tubes. This is to say that four different outputs to $x' \vee y$ as shown in Table 6.10.10 are appended into tubes $T_3$ through $T_6$. Finally, the execution of Step (12) uses the *merge* operation to pour tubes $T_3$ through $T_6$ into tube $T_0$. Tube $T_0$ includes the result implementing, $x' \vee y$, as shown in Table 6.10.10. ■

## Chapter 7

**7.1:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are employed to encode the first input and the second input for $x \vee 1$ as shown in Table 7.9.1. It is also assumed that a binary number of a bit, $C_1$, is applied to encode the output for the logical operation, $x \vee 1$. For the sake of convenience, it is assumed that for $1 \leq k \leq 2$ $R_k^{\,1}$ denotes the fact that the value of $R_k$ is 1 and $R_k^{\,0}$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also supposed that $C_1^{\,1}$ denotes the fact that the value of $C_1$ is 1 and $C_1^{\,0}$ denotes the fact that the value of $C_1$ is 0. The following algorithm is offered to demonstrate $x \vee 1 = 1$. Tubes $T_0$, $T_1$ and $T_2$ are subsequently the first, second and third parameters, and are set to empty tubes. Tubes $T_3$ and $T_4$ that are used in the DNA-based algorithm are also set to empty tubes.

**Procedure X-OR-ONE-IS-EQUAL-TO-ONE**($T_0$, $T_1$, $T_2$)
(1) **X-OR-ONE**($T_0$, $T_1$, $T_2$).
(2) $T_3 = +(T_0, C_1^{\,1})$ and $T_4 = -(T_0, C_1^{\,1})$.
(3) **If** (detectct($T_4$) = = *false*) **Then**
    (3a) The proof to $x \vee 1 = 1$ is completed, and the algorithm is terminated.
(4) **Else**
    (4a) We fail to show $x \vee 1 = 1$, and terminate the algorithm.
**EndIf**
**EndProcedure**

**Proof of Correction**:

The algorithm, **X-OR-ONE-IS-EQUAL-TO-ONE**($T_0$, $T_1$, $T_2$), is implemented by means of the *append-head*, *merge*, *extract* and *detect* operations. On the first execution of Step (1), it invokes the DNA-based algorithm **X-OR-ONE**($T_0$, $T_1$, $T_2$) that is the solution of the exercise 5.7 for generating the result of $x \vee 1$ as shown in Table 7.9.1. This implies that after the DNA-based algorithm **X-OR-ONE**($T_0$, $T_1$, $T_2$) is implemented, $T_0 = \{C_1^{\,1} R_2^{\,1} R_1^{\,1}, C_1^{\,1} R_2^{\,1} R_1^{\,0}\}$, $T_1 = \varnothing$ and $T_2 = \varnothing$. Next, on the first execute of Step (2), it uses the *extract* operation to yield two different tubes ($T_3$ and $T_4$) including different results. Tubes $T_3 = \{C_1^{\,1} R_2^{\,1} R_1^{\,1}, C_1^{\,1} R_2^{\,1} R_1^{\,0}\}$ and $T_4 = \varnothing$ are obtained. If a *false* is returned from the *detect* operation for tube $T_4$ on the first execution of Step (3), then it is at once inferred that the proof for $x \vee 1 = 1$ is completed and the algorithm is terminated. Otherwise, it is at once concluded that we

fail to complete the proof for $x \vee 1 = 1$ and terminate the algorithm. ■

**7.2:**

It is assumed that two binary numbers of a bit, $R_1$ and $R_2$, are applied to encode the first input and the second input for $x \wedge 0$ as shown in Table 7.9.2. It is also supposed that a binary number of a bit, $C_1$, is employed to encode the output for the logical operation, $x \wedge 0$. For the sake of convenience, it is supposed that for $1 \leq k \leq 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also assumed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is presented to demonstrate $x \wedge 0 = 0$. Tubes $T_0$, $T_1$ and $T_2$ are subsequently the first, second and third parameters, and are set to empty tubes. Tubes $T_3$ and $T_4$ that are applied in the DNA-based algorithm are also set to empty tubes.

**Procedure X-AND-ZERO-IS-EQUAL-TO-ZERO**($T_0$, $T_1$, $T_2$)
(1) **X-AND-ZERO**($T_0$, $T_1$, $T_2$).
(2) $T_3 = +(T_0, C_1^1)$ and $T_4 = -(T_0, C_1^1)$.
(3) **If** (detectct($T_3$) = = *false*) **Then**
(3a) The proof to $x \wedge 0 = 0$ is completed, and the algorithm is terminated.
(4) **Else**
(4a) We fail to show $x \wedge 0 = 0$, and terminate the algorithm.
**EndIf**
**EndProcedure**

**Proof of Correction**:

The algorithm, **X-AND-ZERO-IS-EQUAL-TO-ZERO**($T_0$, $T_1$, $T_2$), is implemented by means of the *append-head*, *merge*, *extract* and *detect* operations. On the first execution of Step (1), it calls the DNA-based algorithm **X-AND-ZERO**($T_0$, $T_1$, $T_2$) that is the solution of the exercise 5.8 for producing the result of $x \wedge 0$ as shown in Table 7.9.2. This is to say that after the DNA-based algorithm **X-AND-ZERO**($T_0$, $T_1$, $T_2$) is implemented, $T_0 = \{C_1^0 R_2^0 R_1^1, C_1^0 R_2^0 R_1^0\}$, $T_1 = \varnothing$ and $T_2 = \varnothing$. Next, on the first execute of Step (2), it applies the *extract* operation to generate two different tubes ($T_3$ and $T_4$) containing different results. Tubes $T_3 = \varnothing$ and $T_4 = \{C_1^0 R_2^0 R_1^1, C_1^0 R_2^0 R_1^0\}$ are obtained. If a *false* is returned from the *detect* operation for tube $T_4$ on the first execution of Step (3), then it is immediately derived that the proof for $x \wedge 0 = 0$ is completed and the algorithm is terminated. Otherwise, it is immediately inferred that we fail to complete the proof for $x \wedge 0 = 0$ and terminate

the algorithm. ■

**7.3:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are employed to encode the first input and the second input for $x \vee 0$ and $x$ as shown in Table 7.9.3. Also it is assumed that a binary number of a bit, $C_1$, is used to encode the output for the logical operation, $x \vee 0$. For the sake of convenience, it is assumed that for $1 \le k \le 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also supposed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is proposed to prove $x \vee 0 = x$. Tubes $T_0$, $T_1$ and $T_2$ are subsequently the first, second and third parameters, and are set to empty tubes. Tubes $T_3$, $T_4$, $T_5$, $T_6$, $T_7$ and $T_8$ that are applied in the DNA-based algorithm are also set to empty tubes.

**Procedure X-OR-ZERO-IS-EQUAL-TO-X**($T_0$, $T_1$, $T_2$)
(1) **X-OR-ZERO**($T_0$, $T_1$, $T_2$).
(2) $T_3 = +(T_0, C_1^1)$ and $T_4 = -(T_0, C_1^1)$.
(3) $T_5 = +(T_3, R_1^1)$ and $T_6 = -(T_3, R_1^1)$.
(4) $T_7 = +(T_4, R_1^0)$ and $T_8 = -(T_4, R_1^0)$.
(5) **If** ((detectct($T_5$) = = *true*) **AND** (detectct($T_7$) = = *true*)) **Then**
(5a) The proof to $x \vee 0 = x$ is completed, and the algorithm is terminated.
(6) **Else**
(6a) We fail to show $x \vee 0 = x$, and terminate the algorithm.
**EndIf**
**EndProcedure**

**Proof of Correction**:

The algorithm, **X-OR-ZERO-IS-EQUAL-TO-X**($T_0$, $T_1$, $T_2$), is implemented by means of the *append-head*, *merge*, *extract* and *detect* operations. On the first execution of Step (1), it invokes the DNA-based algorithm **X-OR-ZERO**($T_0$, $T_1$, $T_2$) that is the solution of the exercise 5.1 for yielding the result of $x \vee 0$ as shown in Table 7.9.3. This indicates that after the DNA-based algorithm **X-OR-ZERO**($T_0$, $T_1$, $T_2$) is implemented, $T_0 = \{C_1^1 R_2^0 R_1^1, C_1^0 R_2^0 R_1^0\}$, $T_1 = \varnothing$ and $T_2 = \varnothing$. Next, Steps (2) through (4) use three *extract* operations to generate six different tubes ($T_3$ through $T_8$) consisting of different results. After those operations from Step (2) through Step (4) are implemented, tubes $T_3 = \varnothing$, $T_4 = \varnothing$, $T_5 = \{C_1^1 R_2^0 R_1^1\}$, $T_6 = \varnothing$, $T_7 = \{C_1^0 R_2^0 R_1^0\}$ and $T_8 = \varnothing$ are obtained. If a *true* is returned from the first execution of Step (5), then

it is at once concluded that the proof for $x \vee 0 = x$ is completed and the algorithm is terminated. Otherwise, it is at once derived that we fail to complete the proof for $x \vee 0 = x$ and terminate the algorithm. ■

**7.4:**

It is assumed that two binary numbers of a bit, $R_1$ and $R_2$, are used to encode the first input and the second input for $x \wedge 1$ and $x$ as shown in Table 7.9.4. Also it is supposed that a binary number of a bit, $C_1$, is employed to encode the output for the logical operation, $x \wedge 1$. For the sake of convenience, it is supposed that for $1 \leq k \leq 2$ $R_k^1$ denotes the fact that the value of $R_k$ is 1 and $R_k^0$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also assumed that $C_1^1$ denotes the fact that the value of $C_1$ is 1 and $C_1^0$ denotes the fact that the value of $C_1$ is 0. The following algorithm is offered to show $x \wedge 1 = x$. Tubes $T_0$, $T_1$ and $T_2$ are subsequently the first, second and third parameters, and are set to empty tubes. Tubes $T_3$, $T_4$, $T_5$, $T_6$, $T_7$ and $T_8$ that are employed in the DNA-based algorithm are also set to empty tubes.

**Procedure X-AND-ONE-IS-EQUAL-TO-X**($T_0$, $T_1$, $T_2$)
(1) **X-AND-ONE**($T_0$, $T_1$, $T_2$).
(2) $T_3 = +(T_0, C_1^1)$ and $T_4 = -(T_0, C_1^1)$.
(3) $T_5 = +(T_3, R_1^1)$ and $T_6 = -(T_3, R_1^1)$.
(4) $T_7 = +(T_4, R_1^0)$ and $T_8 = -(T_4, R_1^0)$.
(5) **If** ((detectct($T_5$) = = *true*) **AND** (detectct($T_7$) = = *true*)) **Then**
(5a) The proof to $x \wedge 1 = x$ is completed, and the algorithm is terminated.
(6) **Else**
(6a) We fail to show $x \wedge 1 = x$, and terminate the algorithm.
**EndIf**
**EndProcedure**

**Proof of Correction**:

The algorithm, **X-AND-ONE-IS-EQUAL-TO-X**($T_0$, $T_1$, $T_2$), is implemented by means of the *append-head*, *merge*, *extract* and *detect* operations. On the first execution of Step (1), it calls the DNA-based algorithm **X-AND-ONE**($T_0$, $T_1$, $T_2$) that is the solution of the exercise 5.2 for producing the result of $x \wedge 1$ as shown in Table 7.9.4. This implies that after the DNA-based algorithm **X-AND-ONE**($T_0$, $T_1$, $T_2$) is implemented, $T_0 = \{C_1^1\ R_2^1\ R_1^1, C_1^0\ R_2^1\ R_1^0\}$, $T_1 = \varnothing$ and $T_2 = \varnothing$. Next, Steps (2) through (4) apply three *extract* operations to yield six different tubes ($T_3$ through $T_8$) including different results. After those operations from Step (2) through Step (4) are

implemented, tubes $T_3 = \varnothing$, $T_4 = \varnothing$, $T_5 = \{C_1^{\,1}\ R_2^{\,1}\ R_1^{\,1}\}$, $T_6 = \varnothing$, $T_7 = \{C_1^{\,0}\ R_2^{\,1}\ R_1^{\,0}\}$ and $T_8 = \varnothing$ are obtained. If a *true* is returned from the first execution of Step (5), then it is immediately inferred that the proof for $x \wedge 1 = x$ is completed and the algorithm is terminated. Otherwise, it is immediately concluded that we fail to complete the proof for $x \wedge 1 = x$ and terminate the algorithm. ■

**7.5:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are employed to encode the first input and the second input for $x \vee x'$ as shown in Table 7.9.5. It is also assumed that a binary number of a bit, $C_1$, is used to encode the output for the logical operation, $x \vee x'$. For the sake of convenience, it is assumed that for $1 \le k \le 2$ $R_k^{\,1}$ denotes the fact that the value of $R_k$ is 1 and $R_k^{\,0}$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also supposed that $C_1^{\,1}$ denotes the fact that the value of $C_1$ is 1 and $C_1^{\,0}$ denotes the fact that the value of $C_1$ is 0. The following algorithm is proposed to demonstrate $x \vee x' = 1$. Tubes $T_0$, $T_1$ and $T_2$ are subsequently the first, second and third parameters, and are set to empty tubes. Tubes $T_3$ and $T_4$ that are used in the DNA-based algorithm are also set to empty tubes.

**Procedure X-OR-NEGATIVE-X-IS-EQUAL-TO-ONE**($T_0$, $T_1$, $T_2$)
(1) **X-OR-NEGATIVE-X**($T_0$, $T_1$, $T_2$).
(2) $T_3 = +(T_0, C_1^{\,1})$ and $T_4 = -(T_0, C_1^{\,1})$.
(3) **If** (detectct($T_4$) = = *false*) **Then**
    (3a) The proof to $x \vee x' = 1$ is completed, and the algorithm is terminated.
(4) **Else**
    (4a) We fail to show $x \vee x' = 1$, and terminate the algorithm.
**EndIf**
**EndProcedure**

**Proof of Correction**:

The algorithm, **X-OR-NEGATIVE-X-IS-EQUAL-TO-ONE**($T_0$, $T_1$, $T_2$), is implemented by means of the *append-head*, *merge*, *extract* and *detect* operations. On the first execution of Step (1), it invokes the DNA-based algorithm **X-OR-NEGATIVE-X**($T_0$, $T_1$, $T_2$) that is the solution of the exercise 5.3 for yielding the result of $x \vee x'$ as shown in Table 7.9.5. This is to say that after the DNA-based algorithm **X-OR-NEGATIVE-X**($T_0$, $T_1$, $T_2$) is implemented, $T_0 = \{C_1^{\,1}\ R_2^{\,0}\ R_1^{\,1}, C_1^{\,1}\ R_2^{\,1}\ R_1^{\,0}\}$, $T_1 = \varnothing$ and $T_2 = \varnothing$. Next, on the first execution of Step (2), it uses the *extract* operation to generate two different tubes ($T_3$ through $T_4$) containing different

results. After that operation from Step (2) is implemented, tubes $T_3 = \{C_1^{\,1}\, R_2^{\,0}\, R_1^{\,1}, C_1^{\,1}\, R_2^{\,1}\, R_1^{\,0}\}$ and $T_4 = \varnothing$ are obtained. If a *false* is returned from the *detect* operation for tube $T_4$ on the first execution of Step (3), then it is at once derived that the proof for $x \vee x' = 1$ is completed and the algorithm is terminated. Otherwise, it is at once inferred that we fail to complete the proof for $x \vee x' = 1$ and terminate the algorithm. ■

**7.6:**

It is assumed that two binary numbers of a bit, $R_1$ and $R_2$, are applied to encode the first input and the second input for $x \wedge x'$ as shown in Table 7.9.6. It is also supposed that a binary number of a bit, $C_1$, is employed to encode the output for the logical operation, $x \wedge x'$. For the sake of convenience, it is supposed that for $1 \le k \le 2$ $R_k^{\,1}$ denotes the fact that the value of $R_k$ is 1 and $R_k^{\,0}$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also assumed that $C_1^{\,1}$ denotes the fact that the value of $C_1$ is 1 and $C_1^{\,0}$ denotes the fact that the value of $C_1$ is 0. The following algorithm is presented to show $x \wedge x' = 0$. Tubes $T_0$, $T_1$ and $T_2$ are subsequently the first, second and third parameters, and are set to empty tubes. Tubes $T_3$ and $T_4$ that are applied in the DNA-based algorithm are also set to empty tubes.

**Procedure X-AND-NEGATIVE-X-IS-EQUAL-TO-ZERO**($T_0$, $T_1$, $T_2$)
(1) **X-AND-NEGATIVE-X**($T_0$, $T_1$, $T_2$).
(2) $T_3 = +(T_0, C_1^{\,1})$ and $T_4 = -(T_0, C_1^{\,1})$.
(3) **If** (detectct($T_4$) = = *false*) **Then**
    (3a) The proof to $x \wedge x' = 0$ is completed, and the algorithm is terminated.
(4) **Else**
    (4a) We fail to show $x \wedge x' = 0$, and terminate the algorithm.
**EndIf**
**EndProcedure**

**Proof of Correction**:

The algorithm, **X-AND-NEGATIVE-X-IS-EQUAL-TO-ZERO**($T_0$, $T_1$, $T_2$), is implemented by means of the *append-head*, *merge*, *extract* and *detect* operations. On the first execution of Step (1), it calls the DNA-based algorithm **X-AND-NEGATIVE-X**($T_0$, $T_1$, $T_2$) that is the solution of the exercise 5.4 for producing the result of $x \wedge x'$ as shown in Table 7.9.6. This indicates that after the DNA-based algorithm **X-AND-NEGATIVE-X**($T_0$, $T_1$, $T_2$) is implemented, $T_0 = \{C_1^{\,0}\, R_2^{\,0}\, R_1^{\,1}, C_1^{\,0}\, R_2^{\,1}\, R_1^{\,0}\}$, $T_1 = \varnothing$ and $T_2 = \varnothing$. Next, on the first execution of Step (2), it applies the *extract* operation to yield two different tubes ($T_3$ through $T_4$) consisting of

different results. After that operation from Step (2) is implemented, tubes $T_3 = \{C_1^{\,0}\ R_2^{\,0}\ R_1^{\,1},\ C_1^{\,0}\ R_2^{\,1}\ R_1^{\,0}\}$ and $T_4 = \varnothing$ are obtained. If a *false* is returned from the *detect* operation for tube $T_4$ on the first execution of Step (3), then it is right away concluded that the proof for $x \wedge x' = 0$ is completed and the algorithm is terminated. Otherwise, it is right away derived that we fail to complete the proof for $x \wedge x' = 0$ and terminate the algorithm. ■

**7.7:**

It is assumed that two binary numbers of a bit, $R_1$ and $R_2$, are used to encode the first input and the second input for $x \wedge x$ as shown in Table 7.9.7. It is also supposed that a binary number of a bit, $C_1$, is applied to encode the output for the logical operation, $x \wedge x$. For the sake of convenience, it is supposed that for $1 \leq k \leq 2$ $R_k^{\,1}$ denotes the fact that the value of $R_k$ is 1 and $R_k^{\,0}$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also assumed that $C_1^{\,1}$ denotes the fact that the value of $C_1$ is 1 and $C_1^{\,0}$ denotes the fact that the value of $C_1$ is 0. The following algorithm is offered to prove $x \wedge x = x$. Tubes $T_0$, $T_1$ and $T_2$ are subsequently the first, second and third parameters, and are set to empty tubes. Tubes $T_3$, $T_4$, $T_5$, $T_6$, $T_7$ and $T_8$ that are employed in the DNA-based algorithm are also set to empty tubes.

**Procedure X-AND-X-IS-EQUAL-TO-X**($T_0$, $T_1$, $T_2$)
(1) **X-AND-X**($T_0$, $T_1$, $T_2$).
(2) $T_3 = +(T_0, C_1^{\,1})$ and $T_4 = -(T_0, C_1^{\,1})$.
(3) $T_5 = +(T_3, R_1^{\,1})$ and $T_6 = -(T_3, R_1^{\,1})$.
(4) $T_7 = +(T_4, R_1^{\,0})$ and $T_8 = -(T_4, R_1^{\,0})$.
(5) **If** ((detectct($T_5$) = = *true*) **AND** (detectct($T_7$) = = *true*)) **Then**
(5a) The proof to $x \wedge x = x$ is completed, and the algorithm is terminated.
(6) **Else**
(6a) We fail to show $x \wedge x = x$, and terminate the algorithm.
**EndIf**
**EndProcedure**

**Proof of Correction**:

The algorithm, **X-AND-X-IS-EQUAL-TO-X**($T_0$, $T_1$, $T_2$), is implemented by means of the *append-head*, *merge*, *extract* and *detect* operations. On the first execution of Step (1), it calls the DNA-based algorithm **X-AND-X**($T_0$, $T_1$, $T_2$) that is the solution of the exercise 5.6 for producing the result of $x \wedge x$ as shown in Table 7.9.7. This implies that after the DNA-based algorithm **X-AND-X**($T_0$, $T_1$, $T_2$) is

implemented, $T_0 = \{C_1^{\,1}\ R_2^{\,1}\ R_1^{\,1},\ C_1^{\,0}\ R_2^{\,0}\ R_1^{\,0}\}$, $T_1 = \varnothing$ and $T_2 = \varnothing$. Next, Steps (2) through (4) apply three *extract* operations to yield six different tubes ($T_3$ through $T_8$) including different results. After those operations from Step (2) through Step (4) are implemented, tubes $T_3 = \varnothing$, $T_4 = \varnothing$, $T_5 = \{C_1^{\,1}\ R_2^{\,1}\ R_1^{\,1}\}$, $T_6 = \varnothing$, $T_7 = \{C_1^{\,0}\ R_2^{\,0}\ R_1^{\,0}\}$ and $T_8 = \varnothing$ are obtained. If a *true* is returned from the first execution of Step (5), then it is at once inferred that the proof for $x \wedge x = x$ is completed and the algorithm is terminated. Otherwise, it is at once concluded that we fail to complete the proof for $x \wedge x = x$ and terminate the algorithm. ■

**7.8:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are applied to encode the first input and the second input for $x \vee x$ as shown in Table 7.9.8. It is also assumed that a binary number of a bit, $C_1$, is employed to encode the output for the logical operation, $x \vee x$. For the sake of convenience, it is assumed that for $1 \le k \le 2$ $R_k^{\,1}$ denotes the fact that the value of $R_k$ is 1 and $R_k^{\,0}$ denotes the fact that the value of $R_k$ is 0. Similarly, it is also supposed that $C_1^{\,1}$ denotes the fact that the value of $C_1$ is 1 and $C_1^{\,0}$ denotes the fact that the value of $C_1$ is 0. The following algorithm is offered to demonstrate $x \vee x = x$. Tubes $T_0$, $T_1$ and $T_2$ are subsequently the first, second and third parameters, and are set to empty tubes. Tubes $T_3$, $T_4$, $T_5$, $T_6$, $T_7$ and $T_8$ that are used in the DNA-based algorithm are also set to empty tubes.

**Procedure X-OR-X-IS-EQUAL-TO-X**($T_0$, $T_1$, $T_2$)
(1) **X-OR-X**($T_0$, $T_1$, $T_2$).
(2) $T_3 = +(T_0, C_1^{\,1})$ and $T_4 = -(T_0, C_1^{\,1})$.
(3) $T_5 = +(T_3, R_1^{\,1})$ and $T_6 = -(T_3, R_1^{\,1})$.
(4) $T_7 = +(T_4, R_1^{\,0})$ and $T_8 = -(T_4, R_1^{\,0})$.
(5) **If** ((detectct($T_5$) = = *true*) **AND** (detectct($T_7$) = = *true*)) **Then**
(5a) The proof to $x \vee x = x$ is completed, and the algorithm is terminated.
(6) **Else**
(6a) We fail to show $x \vee x = x$, and terminate the algorithm.
**EndIf**
**EndProcedure**

**Proof of Correction**:

The algorithm, **X-OR-X-IS-EQUAL-TO-X**($T_0$, $T_1$, $T_2$), is implemented by means of the *append-head*, *merge*, *extract* and *detect* operations. On the first execution of Step (1), it invokes the DNA-based algorithm **X-OR-X**($T_0$, $T_1$, $T_2$) that is the solution

of the exercise 5.5 for generating the result of $x \vee x$ as shown in Table 7.9.8. This is to say that after the DNA-based algorithm **X-OR-X**($T_0$, $T_1$, $T_2$) is implemented, $T_0 = \{C_1^{\,1}\ R_2^{\,1}\ R_1^{\,1}, C_1^{\,0}\ R_2^{\,0}\ R_1^{\,0}\}$, $T_1 = \varnothing$ and $T_2 = \varnothing$. Next, Steps (2) through (4) employ three *extract* operations to produce six different tubes ($T_3$ through $T_8$) consisting of different results. After those operations from Step (2) through Step (4) are implemented, tubes $T_3 = \varnothing$, $T_4 = \varnothing$, $T_5 = \{C_1^{\,1}\ R_2^{\,1}\ R_1^{\,1}\}$, $T_6 = \varnothing$, $T_7 = \{C_1^{\,0}\ R_2^{\,0}\ R_1^{\,0}\}$ and $T_8 = \varnothing$ are obtained. If a *true* is returned from the first execution of Step (5), then it is immediately concluded that the proof for $x \vee x = x$ is completed and the algorithm is terminated. Otherwise, it is immediately inferred that we fail to complete the proof for $x \vee x = x$ and terminate the algorithm. ■

**7.9:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are used to encode the two inputs for the **OR** operation of two bits, $x \vee y$ and $y \vee x$, as shown in Table 7.9.9. Also it is assumed that a binary number of a bit, $C_1$, is applied to encode the output for the **OR** operation. For the sake of convenience, it is assumed that for $1 \le k \le 2$ $R_k^{\,1}$ denotes the fact that the value of $R_k$ is 1 and $R_k^{\,0}$ denotes the fact that the value of $R_k$ is 0. Similarly, it is supposed that $C_1^{\,1}$ denotes the fact that the value of $C_1$ is 1 and $C_1^{\,0}$ denotes the fact that the value of $C_1$ is 0. The following algorithm is presented to prove $x \vee y = y \vee x$ that satisfies the commutative law. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes. All of other tubes used in the following DNA-based algorithm are initially set to empty tubes.

**Procedure COMMUTATIVE-LAW-OF-OR**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$)

(1) **OR**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$, $R_1$, $R_2$).

(2) **OR**($T_{20}$, $T_{21}$, $T_{22}$, $T_{23}$, $T_{24}$, $T_{25}$, $T_{26}$, $R_2$, $R_1$).

(3) $T_7 = +(T_0, R_1^{\,1})$ and $T_8 = -(T_0, R_1^{\,1})$.

(4) $T_9 = +(T_7, R_2^{\,1})$ and $T_{10} = -(T_7, R_2^{\,1})$.

(5) $T_{11} = +(T_8, R_2^{\,1})$ and $T_{12} = -(T_8, R_2^{\,1})$.

(6) $T_{27} = +(T_{20}, R_2^{\,1})$ and $T_{28} = -(T_{20}, R_2^{\,1})$.

(7) $T_{29} = +(T_{27}, R_1^{\,1})$ and $T_{30} = -(T_{27}, R_1^{\,1})$.

(8) $T_{31} = +(T_{28}, R_1^{\,1})$ and $T_{32} = -(T_{28}, R_1^{\,1})$.

(9) $T_9^{\,ON} = +(T_9, C_1^{\,1})$ and $T_9^{\,OFF} = -(T_9, C_1^{\,1})$.

(10) $T_{29}^{\,ON} = +(T_{29}, C_1^{\,1})$ and $T_{29}^{\,OFF} = -(T_{29}, C_1^{\,1})$.

(11) $T_{10}^{\,ON} = +(T_{10}, C_1^{\,1})$ and $T_{10}^{\,OFF} = -(T_{10}, C_1^{\,1})$.

(12) $T_{30}^{\,ON} = +(T_{30}, C_1^{\,1})$ and $T_{30}^{\,OFF} = -(T_{30}, C_1^{\,1})$.

(13) $T_{11}^{\,ON} = +(T_{11}, C_1^{\,1})$ and $T_{11}^{\,OFF} = -(T_{11}, C_1^{\,1})$.

(14) $T_{31}^{ON} = +(T_{31}, C_1^1)$ and $T_{31}^{OFF} = -(T_{31}, C_1^1)$.
(15) $T_{12}^{ON} = +(T_{12}, C_1^1)$ and $T_{12}^{OFF} = -(T_{12}, C_1^1)$.
(16) $T_{32}^{ON} = +(T_{32}, C_1^1)$ and $T_{32}^{OFF} = -(T_{32}, C_1^1)$.
(17) **If** ((detectct($T_9^{ON}$) = = *true*) **AND** (detect($T_{29}^{ON}$) = = *true*) **AND**
(detectct($T_{10}^{ON}$) = = *true*) **AND** (detect($T_{30}^{ON}$) = = *true*) **AND**
(detectct($T_{11}^{ON}$) = = *true*) **AND** (detect($T_{31}^{ON}$) = = *true*) **AND**
(detectct($T_{12}^{OFF}$) = = *true*) **AND** (detect($T_{32}^{OFF}$) = = *true*)) **Then**
(17a) The proof to $x \vee y = y \vee x$ that satisfies the commutative law is completed, and the algorithm is terminated.
(18) **Else**
(18a) We fail to show $x \vee y = y \vee x$ that satisfies the commutative law, and terminate the algorithm.
**EndIf**
**EndProcedure**

**Proof of Correction**:

The algorithm, **COMMUTATIVE-LAW-OF-OR**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$), is implemented by means of the *append-head*, *merge*, *extract* and *detect* operations. On the first execution of Step (1), it calls the DNA-based algorithm **OR**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$, $R_1$, $R_2$) that is the solution of the exercise 3.3 for yielding the result of $x \vee y$ as shown in Table 7.9.9, where the *eighth* parameter, $R_1$, is used to encode the *first* input and the *ninth* parameter, $R_2$, is applied to encode the second input. This implies that after the DNA-based algorithm **OR**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$, $R_1$, $R_2$) is implemented, $T_0 = \{C_1^1 R_2^1 R_1^1, C_1^1 R_2^0 R_1^1, C_1^1 R_2^1 R_1^0, C_1^0 R_2^0 R_1^0\}$, $T_1 = \varnothing$, $T_2 = \varnothing$, $T_3 = \varnothing$, $T_4 = \varnothing$, $T_5 = \varnothing$ and $T_6 = \varnothing$. Next, on the first execution of Step (2), it invokes the DNA-based algorithm **OR**($T_{20}$, $T_{21}$, $T_{22}$, $T_{23}$, $T_{24}$, $T_{25}$, $T_{26}$, $R_2$, $R_1$) that is the solution of the exercise 3.3 for yielding the result of $y \vee x$ as shown in Table 7.9.9, where the *eighth* parameter, $R_2$, is used to encode the *first* input and the *ninth* parameter, $R_1$, is applied to encode the second input. This indicates that after the DNA-based algorithm **OR**($T_{20}$, $T_{21}$, $T_{22}$, $T_{23}$, $T_{24}$, $T_{25}$, $T_{26}$, $R_2$, $R_1$) is implemented, $T_{20} = \{C_1^1 R_1^1 R_2^1, C_1^1 R_1^0 R_2^1, C_1^1 R_1^1 R_2^0, C_1^0 R_1^0 R_2^0\}$, $T_{21} = \varnothing$, $T_{22} = \varnothing$, $T_{23} = \varnothing$, $T_{24} = \varnothing$, $T_{25} = \varnothing$ and $T_{26} = \varnothing$.

Next, Steps (3) through (5) use three *extract* operations to generate six different tubes ($T_7$ through $T_{12}$) including different results. After those operations from Step (3) through Step (5) are implemented, tubes $T_7 = \varnothing$, $T_8 = \varnothing$, $T_9 = \{C_1^1 R_2^1 R_1^1\}$, $T_{10} = \{C_1^1 R_2^0 R_1^1\}$, $T_{11} = \{C_1^1 R_2^1 R_1^0\}$ and $T_{12} = \{C_1^0 R_2^0 R_1^0\}$ are obtained. Next, on those

operations from Steps (6) through (8), they apply three *extract* operations to produce six different tubes ($T_{27}$ through $T_{32}$) containing different results. After those operations from Step (6) through Step (8) are implemented, tubes $T_{27} = \varnothing$, $T_{28} = \varnothing$, $T_{29} = \{C_1^{\,1} R_1^{\,1} R_2^{\,1}\}$, $T_{30} = \{C_1^{\,1} R_1^{\,0} R_2^{\,1}\}$, $T_{31} = \{C_1^{\,1} R_1^{\,1} R_2^{\,0}\}$ and $T_{32} = \{C_1^{\,0} R_1^{\,0} R_2^{\,0}\}$ are obtained.

Next, Steps (9) through (16) employ eight *extract* operations to yield sixteen different tubes consisting of different results. After those operations from Step (9) through Step (16) are implemented, tubes $T_9^{ON} = \{C_1^{\,1} R_2^{\,1} R_1^{\,1}\}$, $T_9^{OFF} = \varnothing$, $T_{10}^{ON} = \{C_1^{\,1} R_2^{\,0} R_1^{\,1}\}$, $T_{10}^{OFF} = \varnothing$, $T_{11}^{ON} = \{C_1^{\,1} R_2^{\,1} R_1^{\,0}\}$, $T_{11}^{OFF} = \varnothing$, $T_{12}^{ON} = \varnothing$, $T_{12}^{OFF} = \{C_1^{\,0} R_2^{\,0} R_1^{\,0}\}$, $T_{29}^{ON} = \{C_1^{\,1} R_1^{\,1} R_2^{\,1}\}$, $T_{29}^{OFF} = \varnothing$, $T_{30}^{ON} = \{C_1^{\,1} R_1^{\,0} R_2^{\,1}\}$, $T_{30}^{OFF} = \varnothing$, $T_{31}^{ON} = \{C_1^{\,1} R_1^{\,1} R_2^{\,0}\}$, $T_{31}^{OFF} = \varnothing$, $T_{32}^{ON} = \varnothing$ and $T_{32}^{OFF} = \{C_1^{\,0} R_1^{\,0} R_2^{\,0}\}$ are obtained. If a *true* is returned from the first execution of Step (17), then it is at once derived that the proof for $x \vee y = y \vee x$ that satisfies the commutative law is completed and the algorithm is terminated. Otherwise, it is right away concluded that we fail to complete the proof for $x \vee y = y \vee x$ that satisfies the commutative law and terminate the algorithm. ■

**7.10:**

It is supposed that two binary numbers of a bit, $R_1$ and $R_2$, are employed to encode two inputs for $x \wedge y$ and $y \wedge x$ as shown in Table 7.9.10. Also it is assumed that a binary number of a bit, $C_1$, is applied to encode the output for $x \wedge y$ and $y \wedge x$. For the sake of convenience, it is supposed that for $1 \le k \le 2$ $R_k^{\,1}$ denotes the fact that the value of $R_k$ is 1 and $R_k^{\,0}$ denotes the fact that the value of $R_k$ is 0. Similarly, it is assumed that $C_1^{\,1}$ denotes the fact that the value of $C_1$ is 1 and $C_1^{\,0}$ denotes the fact that the value of $C_1$ is 0. The following algorithm is presented to prove $x \wedge y = y \wedge x$ that satisfies the commutative law. Tubes $T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$ and $T_6$ are subsequently the first, second, third, fourth, fifth, sixth and seventh parameters, and are set to empty tubes. All of other tubes used in the following DNA-based algorithm are initially set to empty tubes.

**Procedure COMMUTATIVE-LAW-OF-AND**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$)
(1) **AND**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$, $R_1$, $R_2$).
(2) **AND**($T_{20}$, $T_{21}$, $T_{22}$, $T_{23}$, $T_{24}$, $T_{25}$, $T_{26}$, $R_2$, $R_1$).
(3) $T_7 = +(T_0, R_1^{\,1})$ and $T_8 = -(T_0, R_1^{\,1})$.
(4) $T_9 = +(T_7, R_2^{\,1})$ and $T_{10} = -(T_7, R_2^{\,1})$.
(5) $T_{11} = +(T_8, R_2^{\,1})$ and $T_{12} = -(T_8, R_2^{\,1})$.
(6) $T_{27} = +(T_{20}, R_2^{\,1})$ and $T_{28} = -(T_{20}, R_2^{\,1})$.
(7) $T_{29} = +(T_{27}, R_1^{\,1})$ and $T_{30} = -(T_{27}, R_1^{\,1})$.

(8) $T_{31} = +(T_{28}, R_1^{\,1})$ and $T_{32} = -(T_{28}, R_1^{\,1})$.
(9) $T_9^{\,ON} = +(T_9, C_1^{\,1})$ and $T_9^{\,OFF} = -(T_9, C_1^{\,1})$.
(10) $T_{29}^{\,ON} = +(T_{29}, C_1^{\,1})$ and $T_{29}^{\,OFF} = -(T_{29}, C_1^{\,1})$.
(11) $T_{10}^{\,ON} = +(T_{10}, C_1^{\,1})$ and $T_{10}^{\,OFF} = -(T_{10}, C_1^{\,1})$.
(12) $T_{30}^{\,ON} = +(T_{30}, C_1^{\,1})$ and $T_{30}^{\,OFF} = -(T_{30}, C_1^{\,1})$.
(13) $T_{11}^{\,ON} = +(T_{11}, C_1^{\,1})$ and $T_{11}^{\,OFF} = -(T_{11}, C_1^{\,1})$.
(14) $T_{31}^{\,ON} = +(T_{31}, C_1^{\,1})$ and $T_{31}^{\,OFF} = -(T_{31}, C_1^{\,1})$.
(15) $T_{12}^{\,ON} = +(T_{12}, C_1^{\,1})$ and $T_{12}^{\,OFF} = -(T_{12}, C_1^{\,1})$.
(16) $T_{32}^{\,ON} = +(T_{32}, C_1^{\,1})$ and $T_{32}^{\,OFF} = -(T_{32}, C_1^{\,1})$.
(17) **If** ((detectct($T_9^{\,ON}$) = = *true*) **AND** (detect($T_{29}^{\,ON}$) = = *true*) **AND**
(detectct($T_{10}^{\,OFF}$) = = *true*) **AND** (detect($T_{30}^{\,OFF}$) = = *true*) **AND**
(detectct($T_{11}^{\,OFF}$) = = *true*) **AND** (detect($T_{31}^{\,OFF}$) = = *true*) **AND**
(detectct($T_{12}^{\,OFF}$) = = *true*) **AND** (detect($T_{32}^{\,OFF}$) = = *true*)) **Then**
(17a) The proof to $x \wedge y = y \wedge x$ that satisfies the commutative law is completed, and the algorithm is terminated.
(18) **Else**
(18a) We fail to show $x \wedge y = y \wedge x$ that satisfies the commutative law, and terminate the algorithm.
**EndIf**
**EndProcedure**

**Proof of Correction**:

The algorithm, **COMMUTATIVE-LAW-OF-AND**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$), is implemented by means of the *append-head*, *merge*, *extract* and *detect* operations. On the first execution of Step (1), it invokes the DNA-based algorithm **AND**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$, $R_1$, $R_2$) that is the solution of the exercise 3.2 for producing the result of $x \wedge y$ as shown in Table 7.9.10, where the *eighth* parameter, $R_1$, is applied to encode the *first* input and the *ninth* parameter, $R_2$, is employed to encode the second input. This is to say that after the DNA-based algorithm **AND**($T_0$, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$, $R_1$, $R_2$) is implemented, $T_0 = \{C_1^{\,1} R_2^{\,1} R_1^{\,1}, C_1^{\,0} R_2^{\,0} R_1^{\,1}, C_1^{\,0} R_2^{\,1} R_1^{\,0}, C_1^{\,0} R_2^{\,0} R_1^{\,0}\}$, $T_1 = \varnothing$, $T_2 = \varnothing$, $T_3 = \varnothing$, $T_4 = \varnothing$, $T_5 = \varnothing$ and $T_6 = \varnothing$. Next, on the first execution of Step (2), it calls the DNA-based algorithm **AND**($T_{20}$, $T_{21}$, $T_{22}$, $T_{23}$, $T_{24}$, $T_{25}$, $T_{26}$, $R_2$, $R_1$) that is the solution of the exercise 3.2 for generating the result of $y \wedge x$ as shown in Table 7.9.10, where the *eighth* parameter, $R_2$, is employed to encode the *first* input and the *ninth* parameter, $R_1$, is applied to encode the second input. This implies that after the DNA-based algorithm **AND**($T_{20}$, $T_{21}$, $T_{22}$, $T_{23}$, $T_{24}$, $T_{25}$, $T_{26}$, $R_2$, $R_1$) is implemented, $T_{20} = \{C_1^{\,1} R_1^{\,1} R_2^{\,1}, C_1^{\,0} R_1^{\,0} R_2^{\,1}, C_1^{\,0} R_1^{\,1} R_2^{\,0}, C_1^{\,0} R_1^{\,0} R_2^{\,0}\}$, $T_{21} = \varnothing$, $T_{22} = \varnothing$, $T_{23} = \varnothing$, $T_{24} = \varnothing$, $T_{25} = \varnothing$

and $T_{26} = \varnothing$.

Next, Steps (3) through (5) employ three *extract* operations to yield six different tubes ($T_7$ through $T_{12}$) containing different results. After those operations from Step (3) through Step (5) are implemented, tubes $T_7 = \varnothing$, $T_8 = \varnothing$, $T_9 = \{{C_1}^1\ {R_2}^1\ {R_1}^1\}$, $T_{10} = \{{C_1}^0\ {R_2}^0\ {R_1}^1\}$, $T_{11} = \{{C_1}^0\ {R_2}^1\ {R_1}^0\}$ and $T_{12} = \{{C_1}^0\ {R_2}^0\ {R_1}^0\}$ are obtained. Next, on those operations from Steps (6) through (8), they use three *extract* operations to generate six different tubes ($T_{27}$ through $T_{32}$) consisting of different results. After those operations from Step (6) through Step (8) are implemented, tubes $T_{27} = \varnothing$, $T_{28} = \varnothing$, $T_{29} = \{{C_1}^1\ {R_1}^1\ {R_2}^1\}$, $T_{30} = \{{C_1}^0\ {R_1}^0\ {R_2}^1\}$, $T_{31} = \{{C_1}^0\ {R_1}^1\ {R_2}^0\}$ and $T_{32} = \{{C_1}^0\ {R_1}^0\ {R_2}^0\}$ are obtained.

Next, Steps (9) through (16) employ eight *extract* operations to produce sixteen different tubes including different results. After those operations from Step (9) through Step (16) are implemented, tubes ${T_9}^{ON} = \{{C_1}^1\ {R_2}^1\ {R_1}^1\}$, ${T_9}^{OFF} = \varnothing$, ${T_{10}}^{ON} = \varnothing$, ${T_{10}}^{OFF} = \{{C_1}^0\ {R_2}^0\ {R_1}^1\}$, ${T_{11}}^{ON} = \varnothing$, ${T_{11}}^{OFF} = \{{C_1}^0\ {R_2}^1\ {R_1}^0\}$, ${T_{12}}^{ON} = \varnothing$, ${T_{12}}^{OFF} = \{{C_1}^0\ {R_2}^0\ {R_1}^0\}$, ${T_{29}}^{ON} = \{{C_1}^1\ {R_1}^1\ {R_2}^1\}$, ${T_{29}}^{OFF} = \varnothing$, ${T_{30}}^{ON} = \varnothing$, ${T_{30}}^{OFF} = \{{C_1}^0\ {R_1}^0\ {R_2}^1\}$, ${T_{31}}^{ON} = \varnothing$, ${T_{31}}^{OFF} = \{{C_1}^0\ {R_1}^1\ {R_2}^0\}$, ${T_{32}}^{ON} = \varnothing$ and ${T_{32}}^{OFF} = \{{C_1}^0\ {R_1}^0\ {R_2}^0\}$ are obtained. If a *true* is returned from the first execution of Step (17), then it is immediately inferred that the proof for $x \wedge y = y \wedge x$ that satisfies the commutative law is completed and the algorithm is terminated. Otherwise, it is at once inferred that we fail to complete the proof for $x \wedge y = y \wedge x$ that satisfies the commutative law and terminate the algorithm. ■