# Computer Number Systems, Codes, and Digital Devices

Prologue

Before starting our discussion of microprocessors and microcomputers, we need to make sure that some key concepts of the number systems, codes, and digital devices used in microcomputers are fresh in your mind. If the short summaries of these concepts in this chapter are not enough to refresh your memory, then you may want to consult some of the chapters in *Digital Circuits and Systems*, McGraw-Hill, 1989, before going on in this book.

## Objectives


*At the conclusion of this chapter you should be able to:*

1. *Convert numbers between the following codes: binary, hexadecimal, and BCD.*
2. *Define the terms bit, nibble, byte, word, most significant bit, and least significant bit.*
3. *Use a table to find the ASCII or EBCDIC code for a given alphanumeric character.*
4. *Perform addition and subtraction of binary, hexadecimal, and BCD numbers.*
5. *Describe the operation of gates, flip-flops, latches, registers, ROMs, PALs, dynamic RAMs, static RAMs, and buses.*
6. *Describe how an arithmetic logic unit can be instructed to perform arithmetic or logical operations on binary words.*

## COMPUTER NUMBER SYSTEMS AND CODES

### Review of Decimal System

To understand the structure of the binary number system, the first step is to review the familiar decimal or base-10 number system. Here is a decimal number with the value of each place holder or digit expressed as a power of 10.

| 5 | 3 | 4 | 6 | 7 | 2 |
|---|---|---|---|---|---|
| $10^3$ | $10^2$ | $10^1$ | $10^0$ | $10^{-1}$ | $10^{-2}$ |

The digits in the decimal number 5346.72 thus tell you that you have 5 thousands, 3 hundreds, 4 tens, 6 ones, 7 tenths, and 2 hundredths. The number of symbols needed in any number system is equal to the base number. In the decimal number system, then, there are 10 symbols, 0 through 9. When the count in any digit position passes that of the highest-value symbol, the digit rolls back to 0 and the next higher digit is incremented by 1. A car odometer is a good example of this.

A number system can be built using powers of any number as place holders or digits, but some bases are more useful than others. It is difficult to build electronic circuits which can store and manipulate 10 different voltage levels but relatively easy to build circuits which can handle two levels. Therefore, a binary, or *base-2,* number system is used to represent numbers in digital systems.

### The Binary Number System

Figure P.1*a*, shows the value of each digit in a binary number. Each binary digit represents a power of 2. A binary digit is often called a *bit*. Note that digits to the right of the *binary point* represent fractions used for numbers less than 1. The binary system uses only two symbols, zero (0) and one (1), so in binary you count as follows: 0, 1, 10, 11, 100, 101, 110, 111, 1000, etc. For reference, Fig. P.1*b* shows the powers of 2 from $2^1$ to $2^{32}$.

Binary numbers are often called *binary words* or just *words*. Binary words with certain numbers of bits have also acquired special names. A 4-bit binary word is called a *nibble*, and an 8-bit binary word is called a *byte*. A 16-bit binary word is often referred to just as a *word*, and a 32-bit binary word is referred to as a *doubleword*. The rightmost or *least significant bit* of a binary word is usually referred to as the LSB. The leftmost or *most significant bit* of a binary word is usually called the MSB.

To convert a binary number to its equivalent decimal number, multiply each digit times the decimal value of the

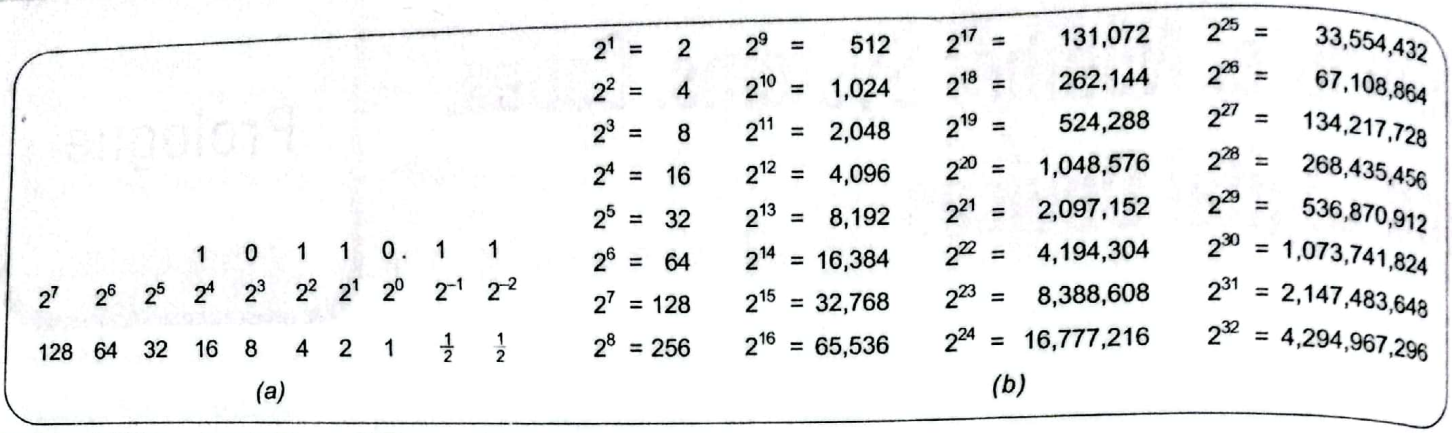



**Fig. P.1** (a) Digit values in binary, (b) Powers of 2.

digit and just add these up. The binary number 101, for example, represents: $(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$, or 4 + 0 + 1 = decimal 5. For the binary number 10110.11, you have:

$$(1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) + (1 \times 2^{-1}) + (1 \times 2^{-22}) =$$
$$16 + 0 + 4 + 2 + 0 + 0.5 + 0.25 = \text{decimal } 22.75$$

To convert a decimal number to binary, there are two common methods. The first (Fig. P.2*a*) is simply a reverse of the binary-to-decimal method. For example, to convert the decimal number 21 (sometimes written as $21_{10}$) to binary, first subtract the largest power of 2 that will fit in the number. For $21_{10}$ the largest power of 2 that will fit is 16 or $2^4$. Subtracting 16 from 21 gives a remainder of 5. Put a 1 in the $2^4$ digit position and see if the next lower power of 2 will fit in the remainder. Since $2^3$ is 8 and 8 will not fit in the remainder of 5, put a 0 in the $2^3$ digit position. Then try the next lower power of 2. In this case the next is $2^2$ or 4, which will fit in the remainder of 5. A 1 is therefore put in the $2^2$ digit position. When $2^2$ or 4 is subtracted from the old remainder of 5, a new remainder of 1 is left. Since $2^1$ or 2 will not fit into this remainder, a 0 is put in that position. A 1 is put in the $2^0$ position because $2^0$ is equal to 1 and this fits exactly into the remainder of 1. The result shows that $21_{10}$ is equal to 10101 in binary. This conversion process is somewhat messy to describe but easy to do. Try converting $46_{10}$ to binary. You should get 101110.

Another method of converting a decimal number to binary is shown in Fig. P.2*b*. Divide the decimal number by 2 and write the quotient and remainder as shown. Divide this quotient and following quotients by 2 until the quotient reaches 0. The column of remainders will be the binary equivalent of the given decimal number. Note that the MSD is on the bottom of the column and the LSD is on the top of the column if you perform the divisions in order from the top to the bottom of the page. You can demonstrate that the binary number is correct by reconverting from binary to decimal, as shown in the right-hand side of Fig. P.2*b*.

You can convert decimal numbers less than 1 to binary by successive multiplication by 2, recording carries until the quantity to the right of the decimal point becomes zero, as shown in Fig. P.2*c*. The carries represent the binary equivalent of the decimal number, with the *most significant bit* at the top of the column. Decimal 0.625 equals 0.101 in binary. For decimal values that do not convert exactly the way this one did (the quantity to the right of the decimal never becomes zero), you can continue the conversion process until you get the number of binary digits desired.

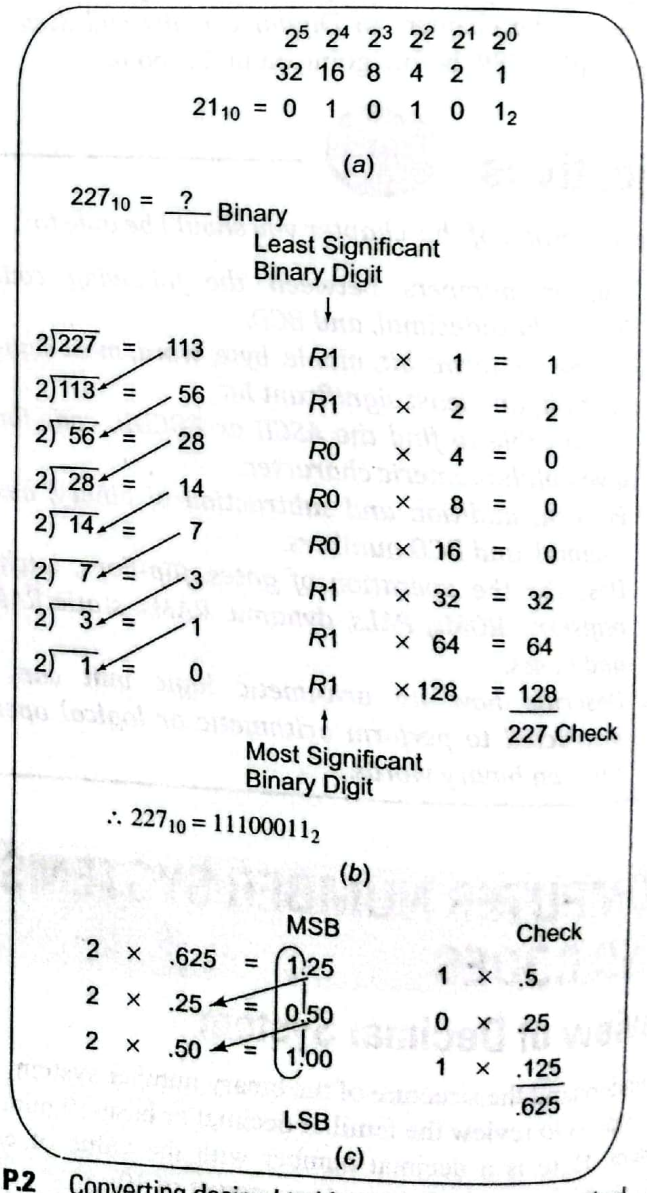



**Fig. P.2** Converting decimal to binary, (a) Digit value method, (b) Divide by 2 method, (c) Decimal fraction conversion.

**Table P.1** Common Number Codes

| Decimal | Binary | Octal | Hex | Binary-Coded Decimal: 8421 | Binary-Coded Decimal: BCD | Binary-Coded Decimal: EXCESS-3 | Reflected Gray Code | 7-Segment Display (1 = on): a | b | c | d | e | f | g | Display |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0000 | 0 | 0 | | 0000 | 0011 | 0000 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0001 | 1 | 1 | | 0001 | 0100 | 0001 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| 2 | 0010 | 2 | 2 | | 0010 | 0101 | 0011 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 2 |
| 3 | 0011 | 3 | 3 | | 0011 | 0110 | 0010 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 3 |
| 4 | 0100 | 4 | 4 | | 0100 | 0111 | 0110 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 4 |
| 5 | 0101 | 5 | 5 | | 0101 | 1000 | 0111 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 5 |
| 6 | 0110 | 6 | 6 | | 0110 | 1001 | 0101 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 6 |
| 7 | 0111 | 7 | 7 | | 0111 | 1010 | 0100 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 7 |
| 8 | 1000 | 10 | 8 | | 1000 | 1011 | 1100 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 8 |
| 9 | 1001 | 11 | 9 | | 1001 | 1100 | 1101 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 9 |
| 10 | 1010 | 12 | A | 0001 | 0000 | 0011 | 1111 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | A |
| 11 | 1011 | 13 | B | 0001 | 0001 | 0100 | 1110 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | B |
| 12 | 1100 | 14 | C | 0001 | 0010 | 0100 | 0101 | 1010 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | C |
| 13 | 1101 | 15 | D | 0001 | 0011 | 0100 | 0110 | 1011 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | D |
| 14 | 1110 | 16 | E | 0001 | 0100 | 0100 | 0111 | 1001 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | E |
| 15 | 1111 | 17 | F | 0001 | 0101 | 0100 | 1000 | 1000 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | F |

combinations as standard binary, but as you can see in the 4-bit example in Table P.1, they are arranged in a different order. Notice that only one binary digit changes at a time as you count up in this code.

If you need to construct a Gray-code table larger than that in Table P.1, a handy way to do so is to observe the pattern of 1's and 0's and just extend it. The least significant digit column starts with one 0 and then has alternating groups of two 1's and two 0's as you go down the column. The second most significant digit column starts with two 0's and then has alternating groups of four 1's and four 0's. The third column starts with four 0's, then has alternating groups of eight 1's and eight 0's. By now you should see the pattern. Try to Figure out the Gray code for the decimal number 16. You should get 11000.

## 7-Segment Display Code

Figure P.4*a* shows the segment identifiers for a 7- segment display such as those commonly used in digital instruments. Table P.1 shows the logic levels required to display 0 to 9 and A to F on a common-cathode LED display such as that shown in Fig. P.4*b*. For a commonanode LED display such as that in Fig. P.4*c*, simply invert the segment codes shown in Table P.1.

## Alphanumeric Codes

When communicating with or between computers, you need a binary-based code which can represent letters of the alphabet as well as numbers. Common codes used for this have 7 or 8 bits per word and are referred to as *alphanumeric codes*. To detect possible errors in these codes, an additional bit, called a *parity bit*, is often added as the most significant bit.

*Parity* is a term used to identify whether a data word has an odd or even number of 1's. If a data word contains an odd number of 1's, the word is said to have *odd parity*. The binary word 0110111 with five 1's has odd parity. The binary word 0110000 has an even number of 1's (two), so it has *even parity*.

In practice the parity bit is used as follows. The system that is sending a data word checks the parity of the word. If the

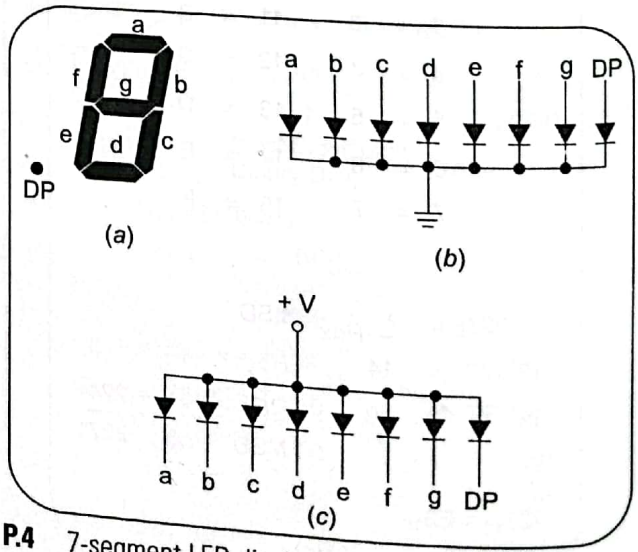



**Fig. P.4** 7-segment LED display, (a) Segment labels, (b) Schematic of common-cathode type, (c) Schematic of common-anode type.

parity of the data word is odd, the system will set the parity bit to a 1. This makes the parity of the data word plus parity bit even. If the parity of the data word is even, the sending system will reset the parity bit to a 0. This again makes the parity of the data word plus parity even. The receiving system checks the parity of the data word plus parity bit that it receives. If the receiving system detects odd parity in the received data word plus parity, it assumes an error has occurred and tells the sending system to send the data again. The system is then said to be using even parity. The system could have been set up to use (maintain) odd parity in a similar manner.

## ASCII

Table P.2 shows several alphanumeric codes. The first of these is *ASCII*, or American Standard Code for Information Interchange. This is shown in the table as a 7- bit code. With 7 bits you can code up to 128 characters, which is enough for

**Table P.2** Common Alphanumeric Codes

| *ASCII Symbol* | *HEX Code for 7-Bit ASCII* | *EBCDIC Symbol* | *HEX Code for EBCDIC* | *ASCII Symbol* | *HEX Code for 7-Bit ASCII* | *EBCDIC Symbol* | *HEX Code for EBCDIC* | *ASCII Symbol* | *HEX Code for 7-Bit ASCII* | *EBCDIC Symbol* | *HEX Code for EBCDIC* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| NUL | 00 | NUL | 00 | * | 2A | * | 5C | T | 54 | T | E3 |
| SOH | 01 | SOH | 01 | + | 2B | + | 4E | U | 55 | U | E4 |
| STX | 02 | STX | 02 | , | 2C | , | 6B | V | 56 | V | E5 |
| ETX | 03 | ETX | 03 | - | 2D | - | 60 | W | 57 | W | E6 |
| EOT | 04 | EOT | 37 | . | 2E | . | 4B | X | 58 | X | E7 |
| ENQ | 05 | ENQ | 2D | / | 2F | / | 61 | Y | 59 | Y | E8 |
| ACK | 06 | ACK | 2E | 0 | 30 | 0 | FO | Z | 5A | Z | E9 |
| BEL | 07 | BEL | 2F | 1 | 31 | 1 | Fl | [ | 5B | [ | AD |
| BS | 08 | BS | 16 | 2 | 32 | 2 | F2 | ȓ | 5C | NL | 15 |
| HT | 09 | HT | 05 | 3 | 33 | 3 | F3 | ] | 5D | [ | DD |
| LF | OA | LF | 25 | 4 | 34 | 4 | F4 | ^ | 5E | ⌝ | 5F |
| VT | OB | VT | OB | 5 | 35 | 5 | F5 | — | 5F | — | 6D |
| FF | OC | FF | OC | 6 | 36 | 6 | F6 | ` | 60 | RES | 14 |
| CR | OD | CR | OD | 7 | 37 | 7 | F7 | a | 61 | a | 81 |
| SO | OE | SO | OE | 8 | 38 | 8 | F8 | b | 62 | b | 82 |
| SI | OF | SI | OF | 9 | 39 | 9 | F9 | c | 63 | c | 83 |
| DLE | 10 | DLE | 10 | : | 3A | : | 7A | d | 64 | d | 84 |
| DC1 | 11 | DC1 | 11 | ; | 3B | ; | 5E | e | 65 | e | 85 |
| DC2 | 12 | DC2 | 12 | — | 3C | — | 4C | f | 66 | f | 86 |
| DC3 | 13 | DC3 | 13 | = | 3D | = | 7E | g | 67 | g | 87 |
| DC4 | 14 | DC4 | 35 | \ | 3E | \ | 6E | h | 68 | h | 88 |
| NAK | 15 | NAK | 3D | ? | 3F | ? | 6F | i | 69 | i | 89 |
| SYN | 16 | SYN | 32 | @ | 40 | @ | 7C | j | 6A | j | 91 |
| ETB | 17 | EOB | 26 | A | 41 | A | Cl | k | 6B | k | 92 |
| CAN | 18 | CAN | 18 | B | 42 | B | C2 | l | 6C | l | 93 |
| EM | 19 | EM | 19 | C | 43 | C | C3 | m | 6D | m | 94 |
| SUB | 1A | SUB | 3F | D | 44 | D | C4 | n | 6E | n | 95 |
| ESC | IB | BYP | 24 | E | 45 | E | C5 | o | 6F | 0 | 96 |
| FS | 1C | FLS | 1C | F | 46 | F | C6 | P | 70 | P | 97 |
| GS | ID | GS | ID | G | 47 | G | C7 | q | 71 | q | 98 |
| RS | IE | RDS | IE | H | 48 | H | C8 | r | 72 | r | 99 |

(*Continued*)

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| US | IF | US | IF | I | 49 | I | C9 | s | 73 | s | A2 |
| SP | 20 | SP | 40 | J | 4A | J | D1 | t | 74 | t | A3 |
| ! | 21 | ! | 5A | K | 4B | K | D2 | u | 75 | u | A4 |
| " | 22 | " | 7F | L | 4C | L | D3 | V | 76 | V | A5 |
| # | 23 | # | 7B | M | 4D | M | D4 | w | 77 | w | A6 |
| $ | 24 | $ | 5B | N | 4E | N | D5 | X | 78 | X | A7 |
| % | 25 | % | 6C | O | 4F | O | D6 | y | 79 | y | A8 |
| & | 26 | & | 50 | P | 50 | P | D7 | z | 7A | z | A9 |
| ' | 27 | ' | 7D | Q | 51 | Q | D8 | { | 7B | { | 8B |
| ( | 28 | ( | 4D | R | 52 | R | D9 | \| | 7C | 1 | 4F |
| ) | 29 | ) | 5D | S | 53 | S | E2 | • } | 7D | } | 9B |
| | | | | | | | | - | 7E | ¢ | 4A |
| | | | | | | | | DEL | 7F | DEL | 07 |

the full upper- and lowercase alphabet, numbers, punctuation marks, and control characters. The code is arranged so that if only uppercase letters, numbers, and a few control characters are needed, the lower 6 bits are all that are required. If a parity check is wanted, a parity bit is added to the basic 7-bit code in the MSB position. The binary word 1100 0100, for example, is the ASCII code for uppercase D with odd parity. Table P.3 gives the meanings of the control character symbols used in the ASCII code table.

## EBCDIC

Another alphanumeric code commonly encountered in IBM equipment is the Extended Binary-Coded Decimal Interchange Code or *EBCDIC*. This is an 8-bit code without parity. A ninth bit can be added for parity. To save space in Table P.2, the eight binary digits of EBCDIC are represented by their 2-digit hex equivalent.

**Table P.3** Definitions of Control Characters

| | | | |
|---|---|---|---|
| NULL | Null | DC1 | Direct control 1 |
| SOH | Start of heading | DC2 | Direct control 2 |
| STX | Start text | DC3 | Direct control 3 |
| ETX | End text | DC4 | Direct control 4 |
| EOT | End of transmission | NAK | Negative acknowledge |
| ENQ | Enquiry | SYN | Synchronous idle |
| ACK | Acknowledge | ETB | End transmission block |
| BELL | BS | CAN | Cancel |
| BS | Backspace | EM | End of medium |
| HT | Horizontal tab | SUB | Substitute |
| LF | Line feed | ESC | Escape |
| VT | Vertical tab | FS | Form separator |
| FF | Form feed | GS | Group separator |
| CR | Carriage return | RS | Record separator |
| SO | Shift out | US | Unit separator |
| SI | Shift in | | |
| DLE | Data link escape | | |

# ARITHMETIC OPERATIONS ON BINARY, HEX, AND BCD NUMBERS

## Binary Arithmetic

### ADDITION

Figure P.5*a* shows the truth table for addition of two binary digits and a carry in ($C_{IN}$) from addition of previous digits. Fig. P.5*b* shows the result of adding two 8-bit binary numbers together using these rules. Assuming that $C_{IN}$ = 1, 1 + 0 + $C_{IN}$ = a sum of 0 and a carry into the next digit, and 1 + 1 +

| INPUTS | | | OUTPUTS | |
|---|---|---|---|---|
| A | B | $C_{IN}$ | S | $C_{OUT}$ |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

(*a*)

$S = A \oplus B \oplus C_{IN}$

$C_{OUT} = A \bullet B + C_{IN}(A \oplus B)$

```
  10011010
+ 11011100
-----------
 [1] 01110110
  ↑
  Carry
```

(*b*)

**Fig. P.5** Binary addition, (a) Truth table for 2 bits plus carry, (b) Addition of two 8-bit words.

Likewise, the OR gates have several inputs, but are shown with a single input line to simplify the drawing. These devices are programmed by blowing out fuses, which are represented in the figure by Xs. An X in the figure indicates that the fuse is intact and makes a connection between, for example, the output of an AND gate and one of the inputs of an OR gate. A dot at the intersection of two wires indicates a hard-wired connection implemented during manufacture.

In a *programmable logic array (PLA)* or *field programmable logic array (FPLA)*, both the AND matrix and the OR matrix are programmable by leaving in fuses or blowing them out. The two programmable matrixes make FPLAs very flexible, but difficult to program.

In a *programmable read-only memory* or *PROM*, the AND matrix is fixed and just the OR matrix is programmable by leaving in fuses or blowing them out. PROMs implement all the possible product terms for the input variables, so they are useful as code converters.

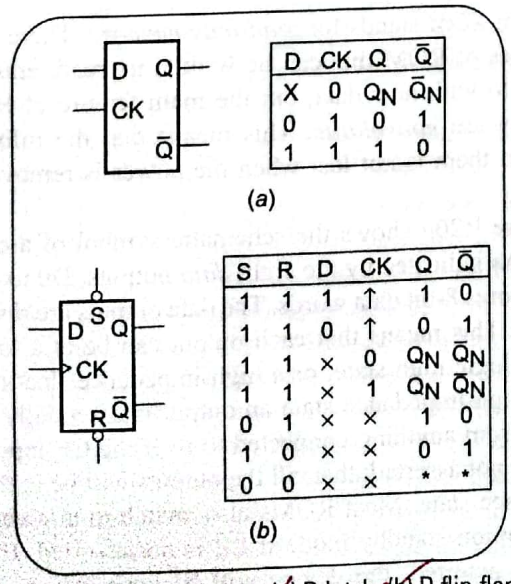
In a *programmable array logic device* or *PAL*, the connections in the OR matrix are fixed and the AND matrix connections are programmable. PALs are often used to implement combinational logic and address decoders in microcomputer systems.

A computer program is usually used to develop the fuse map for an FPLA, PROM, or PAL. Once developed, the fuse-map file is downloaded to a programmer which blows fuses or stores charges to actually program the device.

# Latches, Flip-Flops, Registers, and Counters

## THE D LATCH

A *latch* is a digital device that stores a 1 or a 0 on its output. Fig. P.17*a* shows the schematic symbol and truth table for a D latch. The device functions as follows. If the *enable* input CK is low, the logic level present on the D input will have no effect on the Q and $\overline{Q}$ outputs.




| D | CK | Q | $\overline{Q}$ |
|---|---|---|---|
| X | 0 | $Q_N$ | $\overline{Q}_N$ |
| 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

| S | R | D | CK | Q | $\overline{Q}$ |
|---|---|---|---|---|---|
| 1 | 1 | 1 | ↑ | 1 | 0 |
| 1 | 1 | 0 | ↑ | 0 | 1 |
| 1 | 1 | × | 0 | $Q_N$ | $\overline{Q}_N$ |
| 1 | 1 | × | 1 | $Q_N$ | $\overline{Q}_N$ |
| 0 | 1 | × | × | 1 | 0 |
| 1 | 0 | × | × | 0 | 1 |
| 0 | 0 | × | × | * | * |

**Fig. P.17** Latches and flip-flops. (a) D latch. (b) D flip-flop.

This is indicated in the truth table by an X in the D column. If the enable input is high, a high or a low on the D input will be passed to the Q output. In other words, the Q output will follow the D input as long as the enable input is high. The $\overline{Q}$ output will contain the complement of the logic state on Q. When the enable input is made low again, the state on Q at that time will be latched there. Any changes on D will have no effect on Q until the enable input is made high again. When the enable input goes low, then, the state present on D just before the enable goes low will be stored on the Q output. Keep this operation in mind as you read about the D flip-flop in the next section.

## THE D FLIP-FLOP

Figure P.17*b* shows the schematic symbol and the truth table for a typical D flip-flop. The small triangle next to the CK input of this device tells you that the Q and $\overline{Q}$ outputs are updated when a rising signal edge is applied to the CK input. The up arrows in the clock column of the truth table also indicate that a 1 or 0 on the D input will be copied to the Q output when the clock input goes from low to high. In other words, the D flip-flop takes a snapshot of whatever state is on the D input when the clock goes high, and displays the "photo" on the Q output. If the clock input is low, a change on D will have no effect on the output. Likewise, if the clock input is high, a change on D will have no effect on the Q output. Contrast this operation with that of the D latch to make sure you understand the difference between the two devices.

The D flip-flop in Fig. P.17*b* also has direct *set* (S) and *reset* (R) inputs. A flip-flop is considered set if its Q output is a 1. It is *reset* if its Q output is a 0. The bubbles on the set and reset inputs tell you that these inputs are active low. The truth table for the D flip-flop in Fig. P.17*b* indicates that the set and reset inputs are *asynchronous*. This means that if the set input is asserted low, the output will be set, regardless of the states on the D and the clock inputs. Likewise, if the reset input is asserted low, the Q output will be reset, regardless of the state of the D and clock inputs. The Xs in the D and CK columns of the truth table remind you that these inputs are "don't cares" if set or reset is asserted. The condition indicated by the asterisks (*) is a nonstable condition; that is, it will not persist when reset or clear inputs return to their inactive (high) level.

## REGISTERS

Flip-flops can be used individually or in groups to store binary data. A *register* is a group of D flip-flops connected in parallel, as shown in Fig. P.18*a*. A binary word applied to the data inputs of this register will be transferred to the Q outputs when the clock input is made high. The binary word will remain stored on the Q outputs until a new binary word

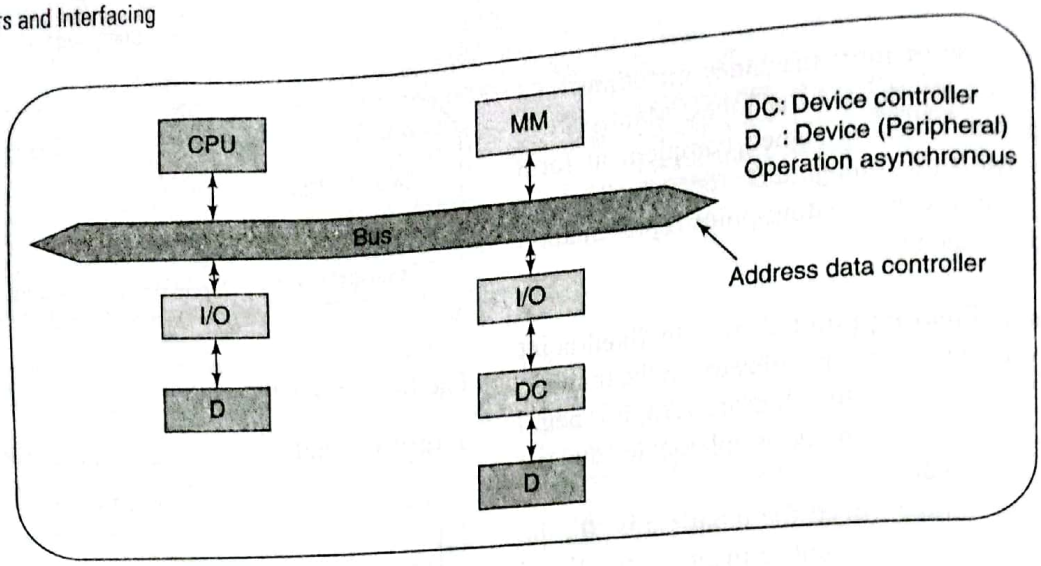



**Fig. 1.7** Bus-organized computer system configuration.

DMA is discussed in later chapters. In some devices, the Device Controller (DC) can control more than one device such as a disk and in others, the device controller and device will be integrated together such as printers. The binary coded information (program) which is sent through the input device to the main memory is converted into binary form before the CPU processes it.

The CPU fetches the machine instructions from the main memory and executes them on the data from the main memory, thereby altering the data as dictated by machine instructions.

Connecting various units as shown in Fig. 1.6 consumes lots of wires and it is not very convenient either to add additional units or debug the system in case of any failure. Hence, the concept of bus was introduced. A bus is a conducting wire carrying a signal from a transmitting source and delivering the signal to one or more receivers. There are two types of buses:

- Unidirectional
- Bidirectional

The frequencies at which the signals are transmitted on the bus make it important that the bus be treated as a transmission line that needs to be terminated at both ends for a bidirectional bus with impedances which equals the characteristic impedance of the wire. A unidirectional bus can be terminated with the characteristic impedance either at the end of the bus or in series with the transmitter. This termination avoids reflections on the bus and assures signal integrity. Most of the computers today are bus organized as shown in Fig. 1.7.

## ARCHITECTURE OF COMPUTERS

There are two computer architectures:

- von Neumann
- Harvard

### von Neumann Architecture

In the von Neumann architecture, there is only one main memory in which both instructions and data are stored together, the process being known as *stored program concept*. When information is fetched from the main memory location address through a register called *program counter*, that information is a machine instruction. If the information is fetched with the address specified from any other register, then that information is data. This *stored-program* concept was developed by designers of ENIAC, a vacuum-tube-based machine built for the US Army between 1943 and 1946. The concept was first expounded by von Neumann (1945), and incorporated into the IAS computer (at the Princeton Institute for Advanced Studies) which was completed in 1952. All general-purpose computers are now based on the key concepts of the von Neumann architecture (Fig. 1.8).

Though the von Neumann model is universal in general-purpose computing, it suffers from one clear problem. All information (instructions and data) must flow back and forth between the processor and memory through a single channel, and this channel has a finite bandwidth. When this bandwidth is fully

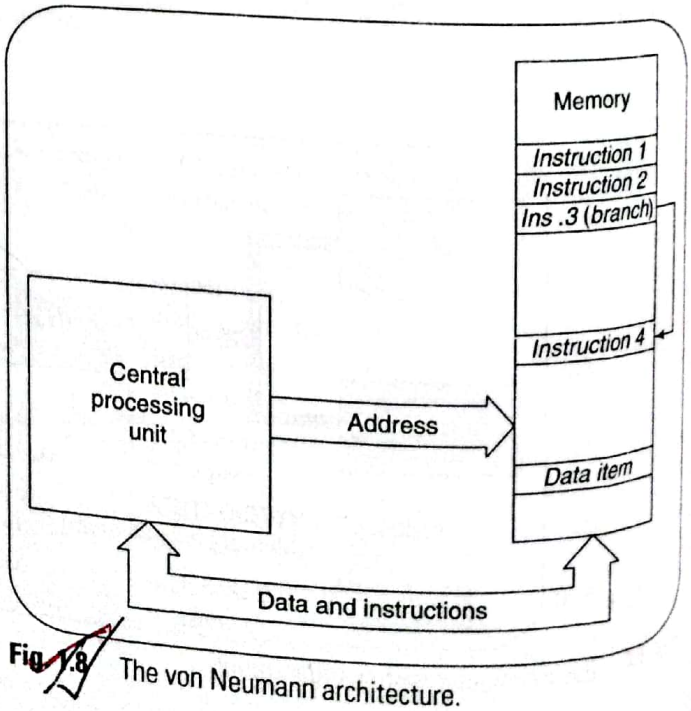



**Fig. 1.8** The von Neumann architecture.

used, the processor can go no faster. This performance-limiting factor is called *von Neumann bottleneck*.

## Harvard Architecture

The other architecture, known as Harvard architecture, shown in Fig. 1.9, has two memories—one for instructions and the other for data. The name comes from Harvard Mark 1, an electromechanical computer which pre-dates the stored-program concept of von Neumann, as does the architecture in this form. The advantage of this architecture is increased bandwidth available due to separate buses for instructions and data. The disadvantage is that the storage is allocated to instructions and data in a fixed ratio.

In current–day computers, von Neumann architecture is used in principle but within the CPU, there are two memories called *instruction cache* and *data cache*. The main memory will have both instructions and data together. The cache (hidden) memory is a very fast memory of limited size within the CPU. More details about the cache memory is discussed later. The modified Harvard architecture is shown in Fig. 1.10.

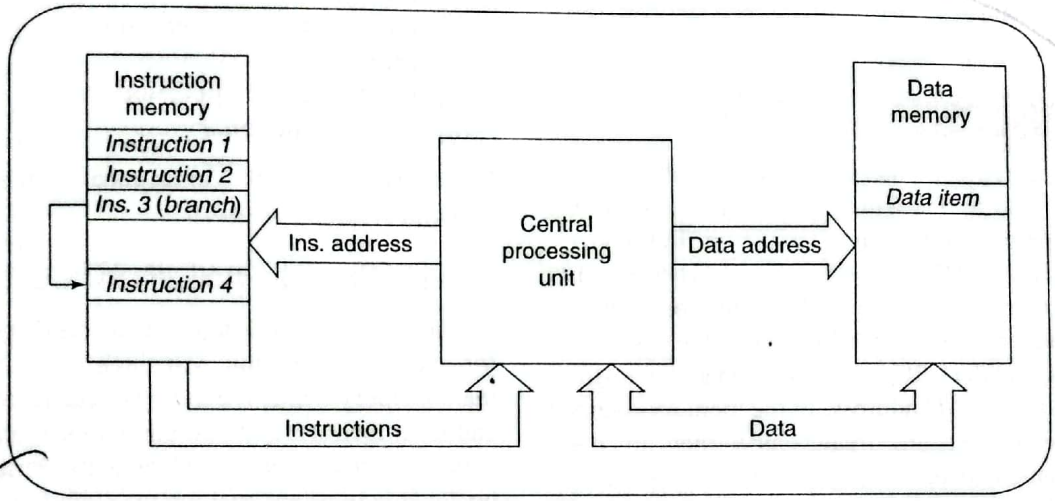



**Fig. 1.9** The Harvard architecture.

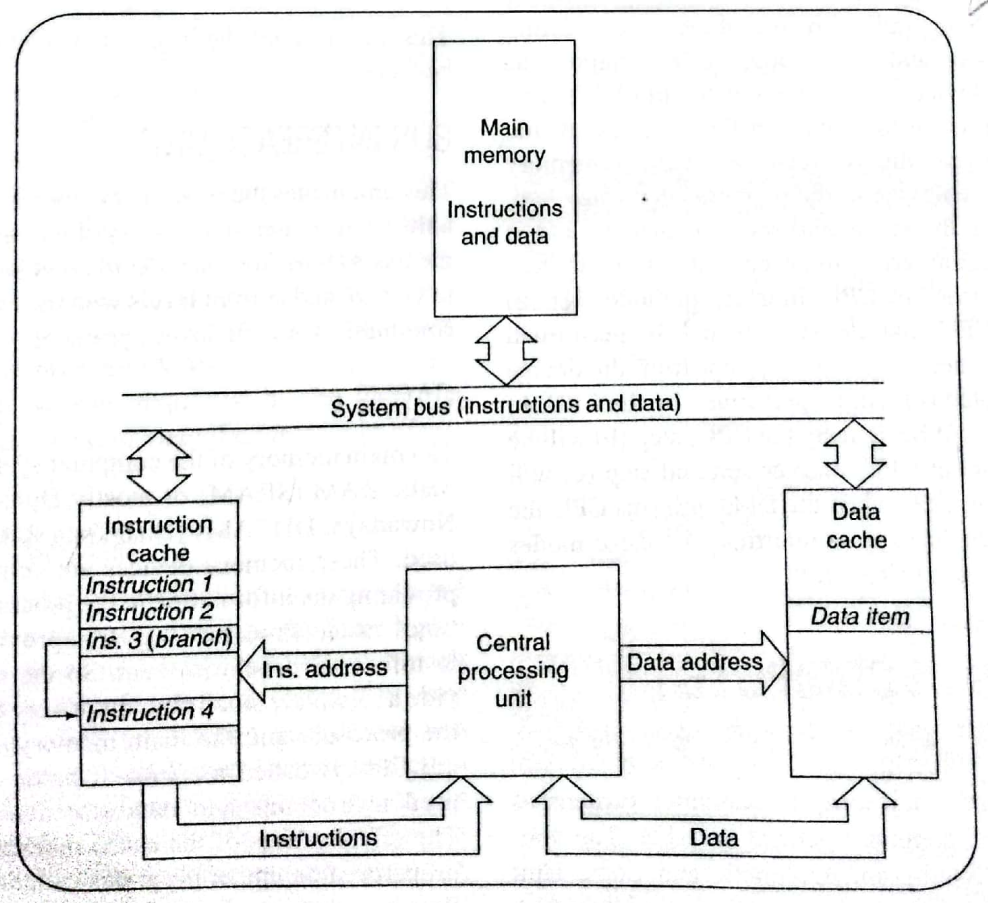



**Fig. 1.10** A modified Harvard architecture.

(DSP) can be incorporated.

Apart from these advances in the CPU, there are other advances such as pipelining, computer networking, and fast serial (gigabit) transmission in modern-day computers.

## RISC AND CISC

There are two classes of computers called Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC). The differences between these two are given below:

| | *CISC* | *RISC* |
|---|---|---|
| 1. | Control unit is hardwired and microprogrammed | Control Unit is hardwired only |
| 2. | Complex multiclock instructions | Simple single clock instructions |
| 3. | Memory reference instructions | Register reference instructions except LOAD and STORE memory reference instructions |
| 4. | Less registers in CPU | More in CPU |
| 5. | Maximum memory addressing modes | Minimum memory addressing modes |
| 6. | Small code sizes | Large code sizes |
| 7. | Variable instruction length | Fixed instruction length |