Chapter 2: Computer Arithmetic and Digital Logic

1. Why is binary arithmetic employed by digital computers?

SOLUTION

Binary arithmetic is used entirely because digital systems are constructed with two-state logic elements. If semiconductor manufacturing processes changed and allowed logic elements with five states, then base-5 arithmetic would be used.

2. We said that binary values have no intrinsic information (that is true of all other number representations). The Voyager I spacecraft, containing samples of human music and other messages, was the first human artifact to leave the solar system to travel to the stars. How is it possible to communicate with aliens in binary form if there is no intrinsic meaning to the data?

SOLUTION

It is, of course, true that binary data has no intrinsic information, and a pattern of 1s and 0s has no intrinsic meaning other than that given to it by the programmer; for example, we agree that the binary code 01000001 represents the letter 'A' in ASCII and represents 65 as an 8421-weighted, unsigned binary, integer.

Science Fiction writers have long thought about how communication might take place between different civilizations. One possible solution is to construct a language using information that can be shared. For example, there are universal constants such as the speed of light, the spectral lines of the hydrogen spectrum, the periodic table and the number of protons in each element. Similarly, there are mathematical concepts such as prime numbers, constants such as π and e, and so on. So, by transmitting, say, a series of prime numbers or the atomic numbers of members of the periodic table, an alien intelligence would be able to guess the sequence and then work out how the numbers have been encoded.

An interesting case of decoding in the face of little information took place in WW2. The British had German Enigma machines but needed to see a translation of known data before they could perform general decoding. So, they arranged for bombs to be dropped onto empty sea near a German submarine. The German submarine observed the bombs and transmitted their position back to base. The message was intercepted and the decoding performed by looking for the encoded position of the bombs using the known actual position.

3. How much more inaccurate is binary integer arithmetic than decimal integer arithmetic? Can the accuracy of binary computers be improved to make them as accurate as decimal computers?

SOLUTION

This is a trick question. In the absence of actual errors (faulty hardware or software), any digital logic system is perfectly accurate; that is, a calculation in one base yields the same result as a calculation in another base. Any so-called inaccuracies result from finite word-lengths (e.g., a 10-bit binary value represents a number to one part in 2^{10} (one in in 1,000) whereas a 10-bit decimal number allows a representation of 1 part in 10,000,000,000). Inaccuracies arise in fractional calculations (remember that a number like π cannot be represented in a finite number of digits). Similarly, not all decimal fractions can always be represented exactly in binary by a finite number of bits.

4. Why are computers byte-oriented?

SOLUTION

There is no logical reason – it is a matter of custom and historical development. During the development of computers many different wordlengths were used. Some early computers actually had a 6-bit byte. First-

generation microprocessors had 8-bit data registers. 8 bits were called bytes. By using the byte as a basic unit of data it means that 16, 32, and 64-bit addresses fit exactly in an integer number of bytes (which is, of course, why we choose these address widths). If an address were 34 bits wide in a byte-oriented world, then it would require a 40-bit word of 5 bytes to store the address with 6 bits unused.

More importantly, a byte is 8, bits which is 2^3 and that fits well in a binary word. Another reason for employing an 8-bit byte is that it allows $2^8 = 256$ different characters which fit in well with the extended ASCII character set.

I sometime wonder whether the basic unit of data in a computer should have been 12 bits. That would permit a wider range of representation using the basic unit (1 in $2^{12} = 4{,}096$) which would have provided for (a) greater precision in simple calculations and (b) a greater ability to encode alphabets. Moreover, a 12-bit unit would allow a feasible address in simple control applications (4K locations), whereas an 8-bit unit has to be used to create a 16-bit address.

5. Calculations are to be performed to a precision of 0.001%. How many bits does this require?

SOLUTION

A precision of 0.001% is one part in 100,000. The nearest power of two above this value is 2^{17}. Therefore, 17 bits are required.

6. What are the decimal equivalents of the following values (assume positional notation and unsigned integer formats):

a. 11001100_2
b. 11001100_3
c. 11001100_4
d. 11001100_{-2}

SOLUTION

a. 11001100_2 $\quad 1 \times 2^2 + 1 \times 2^3 + 1 \times 2^6 + 1 \times 2^7 = 4 + 8 + 64 + 128 = 204$
b. 11001100_3 $\quad 1 \times 3^2 + 1 \times 3^3 + 1 \times 3^6 + 1 \times 3^7 = 2{,}952$
c. 11001100_4 $\quad 1 \times 4^2 + 1 \times 4^3 + 1 \times 4^6 + 1 \times 4^7 = 20{,}560$
d. 11001100_{-2} $\quad 1 \times (-2)^2 + 1 \times (-2)^3 + 1 \times (-2)^6 + 1 \times (-2)^7 = 4 - 8 + 64 - 128 = -68$

7. Why do we have octal and hexadecimal arithmetic?

SOLUTION

Computers operate with base 2 arithmetic. However, it is difficult for people to handle binary numbers (for example 298_{10} is 100101010_2) because binary numbers involve long strings of bits and we are not good at remembering long sequences. Hexadecimal arithmetic has 16 digits from 0 to F representing 4 binary bits. Therefore, a binary number can be easily represented in hexadecimal form by replacing each four bits by a hexadecimal character. For example, 298_{10} is $12A_{16}$ (easier for people to remember than 000100101010).

Octal arithmetic uses base 8 with the digits 0 to 7. Each octal digit replaces three bits. It has the same advantage as hexadecimal numbers; for example the binary number 111101011 is 753 in octal. However, octal arithmetic is hardly used today.

8. Convert the following decimal numbers into (a) binary and (b) hexadecimal forms

a. 25
b. 250
c. 2500
d. 25555

SOLUTION

a.	25	11001	19
b.	250	11111010	FA
c.	2500	100111000100	9C4
d.	25555	110001111010011	63D3

9. Convert the following unsigned binary numbers into decimal form

a. 11
b. 1001
c. 10001
d. 10011001

SOLUTION

a.	11	3
b.	1001	9
c.	10001	17
d.	10011001	153

10. Convert the following hexadecimal numbers into decimal form

a. AB
b. A0B
c. 10A01
d. FFAAFF

SOLUTION

a.	AB	10101011	171
b.	A0B	101000001011	2571
c.	10A01	10000101000000001	68097
d.	FFAAFF	111111111010101011111111	16755455

11. Convert the following hexadecimal numbers into binary format

a. AC
b. DF0B
c. 10B11
d. FDEAF1

SOLUTION

a.	AC	10101100
b.	DF0B	1101111100001011
c.	10B11	10000101100010001
d.	FDEAF1	111111011110101011110001

12. Convert the following fractional decimal numbers into 16-bit unsigned binary form. Use eight bits of precision.

a. 0.2
b. 0.046875
c. 0.1111
d. 0.1234

SOLUTION

a.	0.2	0.0011001100110011	0.00110011
b.	0.046875	0.0000110000000000	0.00001100
c.	0.1111	0.0001110001110001	0.00011100
d.	0.1234	0.0001111110010111	0.00100000

13. Perform the following calculations in the stated bases

a. 00110111_2
$+01011011_2$

b. 00111111_2
$+01001001_2$

c. 00120121_{16}
$+0A015031_{16}$

d. $00ABCD1F_{16}$
$+ 0F00800F_{16}$

SOLUTION

a. 00110111_2
$+01011011_2$
10010010_2

b. 00111111_2
$+01001001_2$
10001000_2

c. 00120121_{16}
$+0A015031_{16}$
$0A135152_{16}$

d. $00ABCD1F_{16}$
$+0F00800F_{16}$
$0FAC4D2E_{16}$

14. What is arithmetic overflow? When does it occur and how can it be detected?

SOLUTION

Arithmetic overflow takes place when one or more two's complement numbers take part in an arithmetic operation and the sign-bit of the result is incorrect. For example, arithmetic overflow takes place when two positive integers are added and the result (when interpreted as a two's complement value) is negative, or when two negative numbers are added and the result is positive. In 4-bit arithmetic, the signed two's complement

values 1100 and 1000 are added to give 1 0100, where the most-significant bit is a carry out. The sign of the result, 0100, is 0 indicating a positive value. Since the numbers we added were both negative, arithmetic overflow has occurred.

In two's complement addition, arithmetic overflow can be detected by saying, "If the sign bits of the two source operands are the same, and differ from the sign bit of the result operand, then arithmetic overflow occurred."

15. The *n*-bit two's complement integer *N* is written a_{n-1}, a_{n-2}, . . . a_1, a_0. Prove that (in two's complement notation) the representation of a signed binary number in *n* + 1 bits may be derived from its representation in *n* bits by repeating the leftmost bit. For example, if *n* = -12 = 10100 in five bits, *n* = -12 = 110100 in six bits.

SOLUTION

In *n* bits the positive number *N* is represented by a_{n-1}, a_{n-2}, . . . a_1, a_0. We can extend this to *n*+1 bits by appending a 0 to the left without changing its value; that is 0, a_{n-1}, a_{n-2}, . . . a_1, a_0.

Now consider the value -*N* in *n* bits. This is represented as 2^n - *N*. If we extend this to *n* + 1 bits, it becomes 2^{n+1} - *N* or $2^n + 2^n$ - *N*. This is, of course, the original negative representation with a leading 1 to the left. Consequently, a positive number is extended by appending a 0, and a negative number by adding a 1; that is, by extending the sign bit.

16. Convert 1234.125 into 32-bit IEEE floating-point format.

SOLUTION

$1234 = 10011010010_2$ $0.125 = 0.001_2$ Therefore, $1234.125 = 10011010010.001_2$

This is $1.0011010010001 \times 2^{10}$ (normalized binary)

The IEEE floating point sign is 0 (positive)

The fractional mantissa in 23 bits (with suppressed leading 1) is 00110100100010000000000

The biased exponent is 10 + 127 = 137 or 10001001_2

The floating point number is 0 10001001 00110100100010000000000 or $449A4200_{16}$

17. What is the decimal equivalent of the 32-bit IEEE floating-point value CC4C0000?

SOLUTION

The binary equivalent of CC4C0000 is 11001100010011000000000000000000

This can be split into sign, biased exponent, and fractional mantissa. That is

S = 1, E = 10011000, F = 00110000000000000000000

The sign is negative, and the exponent is 10011000_2 - 127 = 11001_2 = 25 (remember the stored exponent is biased by 127, which has to be subtracted)

The mantissa, after the insertion of the leading 1, is 1.00110000000000000000000

Combining mantissa and exponent we get
$1.00110000000000000000000 \times 2^{25} = 11001100000000000000000000.0 = 53{,}477{,}376_{10}$

18. What is the difference between overflow in the context of two's complement numbers and overflow in the context of floating-point numbers?

SOLUTION

In two's complement arithmetic, arithmetic overflow occurs when a result goes out of range (that is, more positive than $2^{n-1}-1$ or more negative than -2^{n-1}. When arithmetic overflow occurs, the sign of the computed result is different to the correct result.

In floating-point arithmetic, overflow occurs when the exponent of a floating-point number becomes too large to be represented in the format in use. The number is now too big to be stored and, usually, an exception is generated.

19. In the *negabinary system* an i-bit binary integer, N, is expressed using positional notation as:

$$N = a_0 \times (-1)^0 \times 2^0 + a_1 \times (-1)^1 \times 2^1 + \ldots + a_{i-1} \times (-1)^{i-1} \times 2^{i-1}$$

This is the same as conventional natural 8421 binary weighted numbers, except that alternate positions have the additional weighting +1 and -1. For example, 1101 = (-1 × 1 × 8) + (+1 × 1 × 4) + (-1 × 0 × 2) + (+1 × 1 × 1) = -8 + 4 + 1 = -3. The following 4-bit numbers are represented in negabinary form. Convert them into their decimal equivalents.

a. 0000
b. 0101
c. 1010
d. 1111

SOLUTION

a. 0
b. +4 + +1 = 5
c. -8 + -2 = -10
d. -8 + +4 + -2 +1 = -5

20. Perform the following additions on 4-bit negabinary numbers. The result is a 6-bit negabinary value.

a. 0000 + 0001 = 000001
b. 1010 + 0101 = 001111
c. 1101 + 1011 = 110100
d. 1111 + 1111 = 001010

SOLUTION

a. 0000 + 0001 = 000001
b. 1010 + 0101 = 001111
c. 1101 + 1011 = 110100
d. 1111 + 1111 = 001010

21. Arithmetic overflow occurs during a two's complement addition if the result of adding two positive numbers yields a negative result, or if adding two negative numbers yields a positive result. If the sign bits of *A* and *B* are the same but the sign bit of the result is different, arithmetic overflow has occurred. If a_{n-1} is the sign bit of *A*, b_{n-1} is the sign bit of *B*, and s_{n-1} is the sign bit of the sum of *A* and *B*, then overflow is defined by

$$V = a_{n-1} \cdot b_{n-1} \cdot \overline{s_{n-1}} + \overline{a_{n-1}} \cdot \overline{b_{n-1}} \cdot s_{n-1}$$

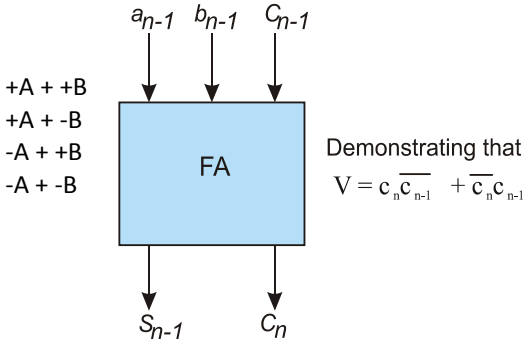
In practice, real systems detect overflow from $C_{in} \neq C_{out}$ to the last stage. That is, we detect overflow from

$$V = \overline{c_n} \cdot c_{n-1} + c_n \cdot \overline{c_{n-1}}$$

Demonstrate that this expression is correct.

SOLUTION

The diagram illustrates the most-significant stage of a parallel adder that adds together bits a_{n-1}, b_{n-1}, and c_{n-1} to generate a sum bit, s_{n-1}, and a carry-out, c_n. There are four possible combinations of *A* and *B* which can be added together:



Because adding two numbers of differing sign *cannot* result in arithmetic overflow, we need consider only the cases where A and B are both positive, and where A and B are both negative.

Case 1: *A* and *B* are positive; that is, $a_{n-1} = 0$, $b_{n-1} = 0$

The final stage adds $a_{n-1} + b_{n-1} + c_{n-1}$ to get c_n. Because a_{n-1} and b_{n-1} are both 0 (by definition if the numbers are positive), the carry-out, c_n, is 0, and $s_{n-1} = c_{n-1}$.

We know that overflow occurs if $s_{n-1} = 1$ (i.e., the sign bit of the sum is negative), therefore overflow occurs if $c_n \cdot \overline{C_{n-1}} = 1$.

Case 2: A and B are negative; that is, $a_{n-1} = 1$, $b_{n-1} = 1$.

The final stage adds $a_{n-1} + b_{n-1} + c_{n-1} = 1 + 1 + c_{n-1}$, to get a sum, $S_{n-1} = c_{n-1}$, and a carry-out $c_n = 1$.

Overflow occurs if $s_{n-1} = 0$. Consequently, overflow occurs when $c_{n-1} = 0$ and $c_n \cdot \overline{c_{n-1}} = 1$.

Considering both cases, overflow occurs if $\overline{c_n} \cdot c_{n-1} + c_n \cdot \overline{c_{n-1}} = 1$.

22. What is the difference between a *truncation* error and a *rounding* error?

SOLUTION

A truncation error occurs when a number is rounded by chopping off bits; for example, 0.1110111 is truncated to five bits as 0.11101 by chopping off the last two bits (11). Rounding is similar to truncation in the sense that bits are removed. When a number is truncated, the bits removed are examined. If they are less than half the value of the least-significant bit, then the bits are just dropped. If they are equal to or greater than half the least-significant digit, the least significant digit is rounded up.

For example 0.1110111 would be rounded up to 0.11101 + 1 = 0.11110, whereas 0.1110001 would be rounded down to 0.11100. Truncation leads to a systematic or biased error (the rounding is always down). Rounding leads to an unbiased error (the error is sometimes positive and sometimes negative). However, rounding is more difficult to perform as it requires an extra addition.

23. Positive and negative numbers can be represented in many ways in a computer. List some of the ways of representing signed numbers. Can you think of any other ways of representing signed values?

SOLUTION

There are four common ways of representing signed numbers in computing:

a. One's complement; the number begins with 0 for positive values and 1 for negative values. If the number has n bits, n-1 bits are used for the number and one for its polarity (i.e., sign). A negative number is obtained from a positive number by inverting bits; for example, if 12 in six bits is 001100 then -12 is 110011. One's complement is little used today. It has the disadvantage that there are two values for zero: 000…0 is +0 and 111…1 is -0.

b. Two's complement: like 1's complement, the most-significant bit is a sign bit. A two's complement number is formed by inverting the bits and then adding 1 (i.e., one's and two's complement negative values differ by 1). If 12 is 001100, -12 is 110011 + 1 = 110100. A two's complement number has a single value for zero, 000...0, and is a true complement because × + -x = 0 (e.g., 12 + -12 = 001100 + 110100 = 1 000000). The addition of two numbers generates a carry bit that is not part of the result. Two's complement representation is the standard form of representing signed integers. It is used because addition and subtraction are the same operation; that is x - y is evaluated by x + (-y).

c. Sign and magnitude representation is the simplest way of representing signed values. You take the most-significant bit and use it as a sign (0 = + and 1 = -); for example, +12 = 001100 and -12 = 101100. Sign and magnitude representation has two values for 0 and you can't use the same hardware for addition and subtraction. It is used largely to represent floating-point values.

d. Excess notation. One way of dealing with negative values is to get rid of negative numbers. Six bits can represent the unsigned integer range 0 to 63. Suppose we call these numbers -32 to 31 so that 000000 is -32, 000001 is 31, … , 100000 is 0, 100001 is 1, …, and 111111 is 31. Now we have a continuous sequence of positive numbers that represent -32 to 31 This representation is called 'excess', because we add a bias or constant to each number to convert it to its excess representation form. For example, in 6 bits -5 becomes -5 + 32 = 27 = 011011. The advantage of this representation is that numbers are monotonic from the most negative to most positive (monotonic means that if two numbers differ by 1, their representation differs by 1 in two's complement form, 0 is 0000..0 and -1 is 1111..11). This form of representing negative values is used to represent exponents in floating-point.

e. You can represent negative numbers in many ways; for example, negabinary numbers use positional weighting with the *n*th digit being weighted by $(-2)^n$. The position values would be +64 -32 +16 -8 +4 -2 +1; for example 0011011 would represent 0 + 0 + 16 -8 + 0 - 2 + 1 = +7.

24. What is a NaN and what is its significance in floating-point arithmetic?

SOLUTION

The IEEE 754 floating-point standard represents real numbers and three special entities: plus infinity, minus infinity, and *not a number*, NaN. The NaN cannot be interpreted as a normal floating-point value. That is, a NaN is a value that is outside the IEEE 754 standard. This mechanism allows the designer to use the bits of a floating-point value to represent anything that he or she wishes; it's a form of *escape* code. The NaN has the curious property that it is *unordered* and is not larger or smaller than another number including itself! A NaN is represented by an all 1s exponent and a non-zero mantissa.

Some literature divides NaNs into two categories: *quiet* NaNs that can be propagated in computer arithmetic, and *signalling* NaNs that can be used to force an exception.

25. Write down the largest base-5 positive integer in n digits and the largest base 7 number in m digits. It is necessary to represent n-digit base-5 numbers in base-7. What is the minimum number m of digits needed to represent all possible n-digit base-5 numbers? *Hint*—the largest m-digit base-7 number should be greater than, or equal to, the largest n-digit base-5 number.

SOLUTION

The largest base-5 number in n digits is 444…4 (n fours) which is 5^n-1. Similarly, the largest base 7 number in m digits is 7^m - 1. In order to represent all base 5 numbers we have

$7^m - 1 \geq 4^n - 1$ or $7^m \geq 4^n$ or $m.\log_{10}7 \geq n.\log_{10}4$ or $m \geq n.\log_{10}4 / \log_{10}7$

26. What is the largest three-digit number in base 13?

SOLUTION

$13^3 - 1 = 2197 - 1 = 2196$ or $CCC_{13} = 12 \times 13 \times 13 + 12 \times 13 + 12 = 2028 + 156 + 12 = 2196$

27. Consider $x^2 - y^2$ where $x = 12.1234$ and $y = 12.1111$. If arithmetic operations are carried out to six decimal significant figures , does it matter whether you evaluate this expression as $x^2 - y^2$ or $(x + y)(x - y)$?

SOLUTION

x = 12.1234 $x^2 = 146.97682756 = 146.977$ (six figures rounded up)
y = 12.1111 $y^2 = 146.67874321 = 146.679$ (six figures rounded up)
$x^2 - y^2 = 146.977 - 146.679 = 0.298000$ (only three significant figures available)

x + y = 12.1234 + 12.1111 = 24.2345 (six figures)
x - y = 12.1234 - 12.1111 = 0.0123
(x + y)(x - y) = 24.2345 × 0.0123 = 0.29808435 = 0.298084 (six figures)

Using a calculator the answer is 0.29808435. Clearly, the way in which we perform operations affects the result when using finite precision.

28. You are evaluating the function $x^4 + 4x^2 + 10x + 2$ at $x = 2$. What is the estimated error if the error in x is R_x?

SOLUTION

To calculate the error, differentiate the polynomial and substitute the value of x. That is: $4x^3 + 8x + 10$ at x = 2 is 4 x 8 + 8 x 2 + 10 = 58. Consequently, the total estimated error is $58R_x$ where R_x is the error in x.

29. What does the following ASCII-encoded message mean? Each character is given in hexadecimal form.

43, 6F, 6D, 70, 75, 74, 65, 72, 2E

SOLUTION

Simply read these values from the ASCII table to get Computer.

30. Floating-point arithmetic is seldom used to perform financial calculations. Why?

SOLUTION

Floating-point is generally used in engineering, scientific, and graphics calculations where wide-ranging numbers have represented (e.g., 5.97219×10^{24} or $9.10938188 \times 10^{-31}$). Floating-point arithmetic is intrinsically slower than fixed-point integer arithmetic (but not necessarily so in practice if hardware acceleration techniques and pipelining are used to implement a dedicated floating-point processor). Binary floating-point numbers do not exactly represent real numbers, and there is an error of conversion between real and floating-point forms. Very large numbers of financial transactions could lead to significant rounding errors in the long term.

31. Modern computers use unsigned integer arithmetic, fixed point arithmetic, two's complement arithmetic, and floating point arithmetic.

a. By examining a binary number, can you tell which number system it represents?
b. Why are there so many ways of representing numeric values?
c. Do we need them all?

SOLUTION

a. No. Remember that binary numbers have no intrinsic meaning and you cannot determine the meaning of a given number. However, you can guess the meaning of a string of binary values because the string may 'make sense' when interpreted in a particular way. For example, if you see a string of bits and notice that when interpreted as ASCII characters the string says 'Hello world", it is very likely to be ASCII text and not a floating-point number that looks like this string.

b. People invent things and then make improvements. Moreover, different representations have different properties; for example, two's complement representation makes the operation of addition and subtraction identical although this complicates division and multiplication.

c. A single binary format could be devised to deal with all number types – but that would complicate the logic circuits that are used to implement the system

32. For each of the following numbers, state the base in use; that is, what is p, q, r, s, t, u?

a. $100001_p = 33_{10}$
b. $25_q = 13_{10}$
c. $25_r = 23_{10}$
d. $25_s = 37_{10}$
e. $1010_t = 68_{10}$
f. $1001_u = 126_{10}$

SOLUTION

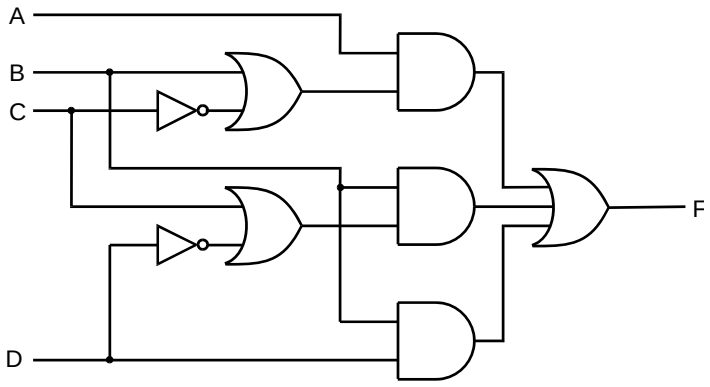
a. $100001_p = 33_{10}$ p = 2
b. $25_q = 13_{10}$ q = 4 (2x4 + 5 = 13)
c. $25_r = 23_{10}$ r = 9
d. $25_s = 37_{10}$ s = 16 (2 × 16 + 5 = 37)
e. $1010_t = 68_{10}$ t = 4 (4 × 4 × 4 + 1 × 4 = 68)
f. $1001_t = 126_{10}$ u = 5 (5 × 5 × 5 + 1 = 126)

33. A digital logic element represents the high state with an *output* of between 2.8 and 2.95 V. The same logic element will see an input high state as a voltage in the range 2.1 to 3.0 V. What is the reason for this difference? What are the practical implications?

SOLUTION

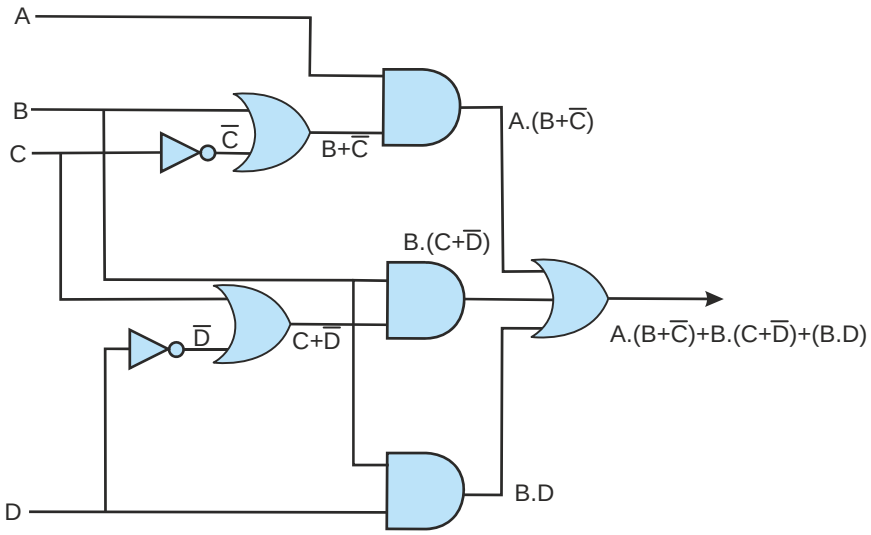
The output level for a high state is determined by the circuit of the gate and the electrical characteristics of the transistors. When a circuit is designed, the output high level is made as close to the high-voltage level in the circuit as possible. The range of inputs that are interpreted as a high level is made wider. This is done so that a high-level signal can suffer from some degradation due to noise and still be interpreted as a high level state. In this example, the lowest high-level output of a gate is 2.8V, whereas an input of 2.1V will be recognized as a high-level state. This means that a 2.1V output can be corrupted by noise or otherwise degraded by 0.7V and STILL be recognized as a high-level state.

34. Draw a truth table to represent the intermediate values and output of the circuit below.



SOLUTION

We can redraw the circuit with intermediate gate outputs and then create the following truth table.



A	B	C	D	C*	D*	B+C*	C+D*	A(B+C*)	B(C+D*)	B.D	A(B+C*) +B(C+D*)+B.D
0	0	0	0	1	1	1	1	0	0	0	0
0	0	0	1	1	0	1	0	0	0	0	0
0	0	1	0	0	1	0	1	0	0	0	0
0	0	1	1	0	0	0	1	0	0	0	0
0	1	0	0	1	1	1	1	0	1	0	1
0	1	0	1	1	0	1	0	0	0	1	1
0	1	1	0	0	1	1	1	0	1	0	1
0	1	1	1	0	0	1	1	0	1	1	1
1	0	0	0	1	1	1	1	1	0	0	1
1	0	0	1	1	0	1	0	1	0	0	1
1	0	1	0	0	1	0	1	0	0	0	0
1	0	1	1	0	0	0	1	0	0	0	0
1	1	0	0	1	1	1	1	1	1	0	1
1	1	0	1	1	0	1	0	1	0	1	1
1	1	1	0	0	1	1	1	1	1	0	1
1	1	1	1	0	0	1	1	1	1	1	1

35. In a digital system, what is the meaning of *negative logic*?

SOLUTION

In negative logic a low-level is interpreted as the true state and a high-level as the false state. Thus, a negative logic AND gate gives a low output if, and only if, both its inputs are low. A negative logic AND gate is identical to a positive logic OR gate.

36. The signal $\overline{\text{RESET}}$ is asserted. What does this statement mean?

SOLUTION

This means that a signal must be asserted active-low to cause a reset to occur. The term *asserted* means put in which ever state caused its named action to occur. Using the word *asserted* avoids the reader having to remember which level (1 or 0) is the active state.

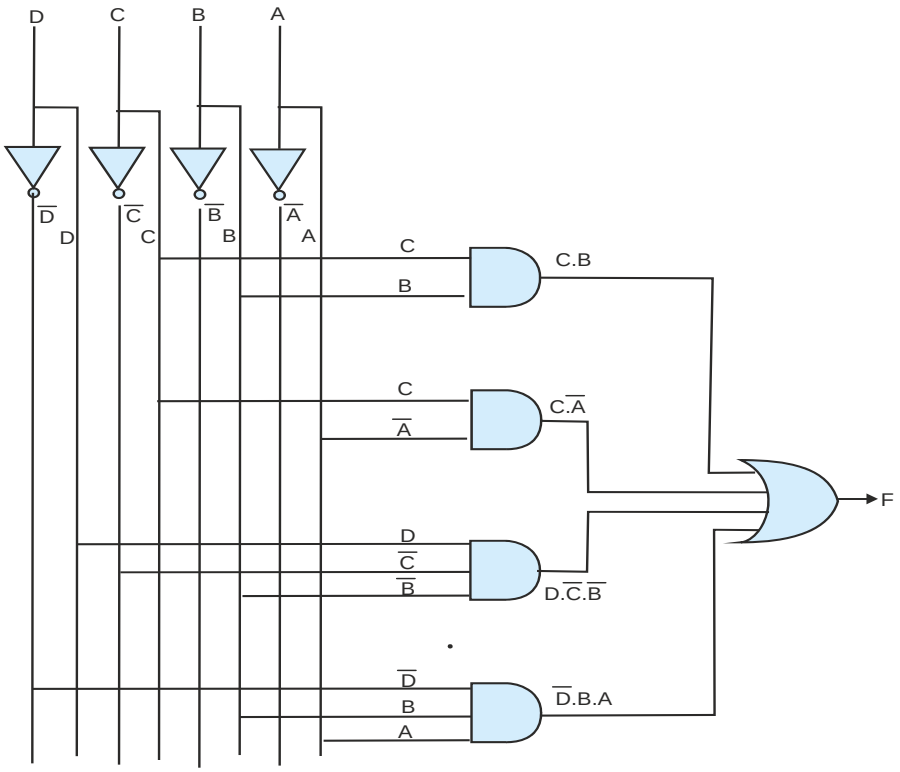
37. A digital system has four one-bit inputs *D*, *C*, *B*, *A,* and an output *F*. The input represents a 4-bit number in the range 0 to 15, where *A* denotes the least-significant bit. The output *F* is true if the binary input is divisible by 3, 4, or 7. Construct a truth table to represent this system and construct a logic circuit to implement it.

SOLUTION

DCBA	Number	÷3	÷4	÷7	Divisible by 3, 4, or 7
0000	0	0	0	0	0
0001	1	0	0	0	0
0010	2	0	0	0	0
0011	3	1	0	0	1
0100	4	0	1	0	1
0101	5	0	0	0	0
0110	6	1	0	0	1
0111	7	0	0	1	1
1000	8	0	1	0	1
1001	9	1	0	0	1
1010	10	0	0	0	0
1011	11	0	0	0	0
1100	12	1	1	0	1
1101	13	0	0	0	0
1110	14	0	0	1	1
1111	15	1	0	0	1

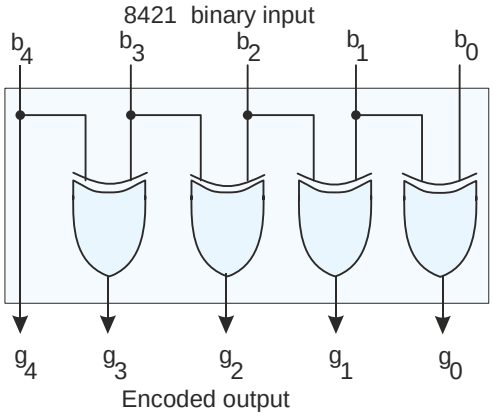
$\overline{D}\cdot\overline{C}\cdot B\cdot A + \overline{D}\cdot C\cdot\overline{B}\cdot\overline{A} + \overline{D}\cdot C\cdot B\cdot\overline{A} + \overline{D}\cdot C\cdot B\cdot A + D\cdot\overline{C}\cdot\overline{B}\cdot\overline{A} + D\cdot\overline{C}\cdot\overline{B}\cdot A + D\cdot C\cdot\overline{B}\cdot\overline{A} + D\cdot C\cdot B\cdot\overline{A} + D\cdot C\cdot B\cdot A$

$F = C\cdot B + C\cdot\overline{A} + \overline{D}\cdot B\cdot A + D\cdot\overline{C}\cdot\overline{B}$



38. Consider the following circuit that takes a 4-bit binary input and encodes it. Construct a truth table for the 16 inputs 0000 to 1111 and examine the output code. What is the characteristic feature of this code?

SOLUTION



The truth table for this circuit is:

Decimal value	Natural binary value	Gray code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

The output code is called a Gray code and is an unweighted binary code; that is, the bit positions do not have the binary weight 1,2,4,8. The important feature of this code is that two successive values (including from 15 to 0) differ in only one bit position, unlike natural binary where 7 (0111) and 8 (1000) differ in all four bit positions. This code is used when data always monotonically increases or decreases (i.e., numbers step through all values in sequence). Because only one bit changes at a time, there can never be any confusion between two consecutive numbers. In weighted codes where two bits may change so that 0011 (3) goes to 0100 (4) the sequence may change 0011 to 0000 to 0100 with the intermediate 0000 value being an error or glitch.

39. A logic circuit has two 2-bit natural binary inputs A and B. A is given by A_1, A_0 where A_1 is the most-significant bit. Similarly, B is given by B_1, B_0 where B_1 is the most-significant bit. The circuit has three outputs, X, Y, and Z. This circuit compares A with B and determines whether A is greater or less than B, or is the same as B. The relationship between inputs A and B, and outputs X, Y, Z is as follows.

Condition	X	Y	Z
A > B	1	0	0
A < B	0	1	0
A = B	0	0	1

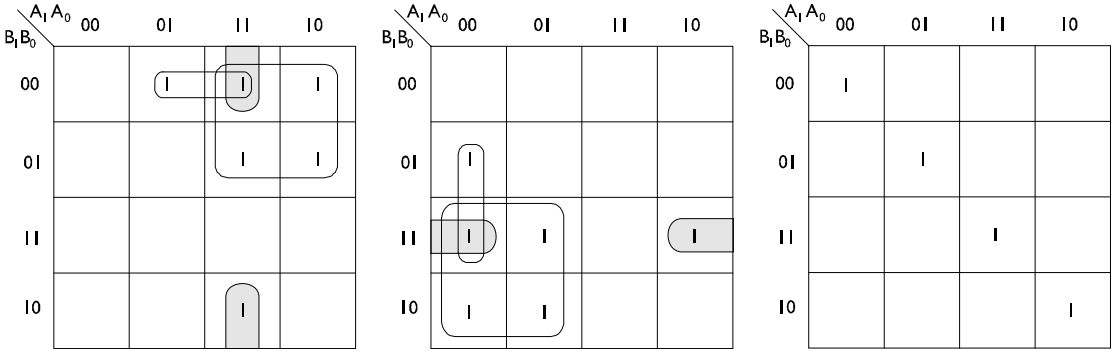
Design a circuit to implement this function.

SOLUTION

The truth table for this problem is obtained by treating the A and B inputs as 2-bit numbers (i.e., 0,0 = 0, 0,1 = 1, 1,0 = 2, and 1,1 = 3) and then comparing the A and B inputs.

Inputs				A > B	A < B	A = B
A_1	A_0	B_1	B_0	X	Y	Z
0	0	0	0	0	0	1
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	1	0	0
0	1	0	1	0	0	1
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	0	1
1	0	1	1	0	1	0
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	0	1

We can use a Karnaugh map to derive expressions for outputs X, Y, and Z. This example demonstrates *symmetry* in Boolean systems. Notice the relationship between the column corresponding to the X output (A > B) and the Y output (A < B). These outputs are complements for all cases except when A = B. You would expect this symmetry from the very nature of the problem.



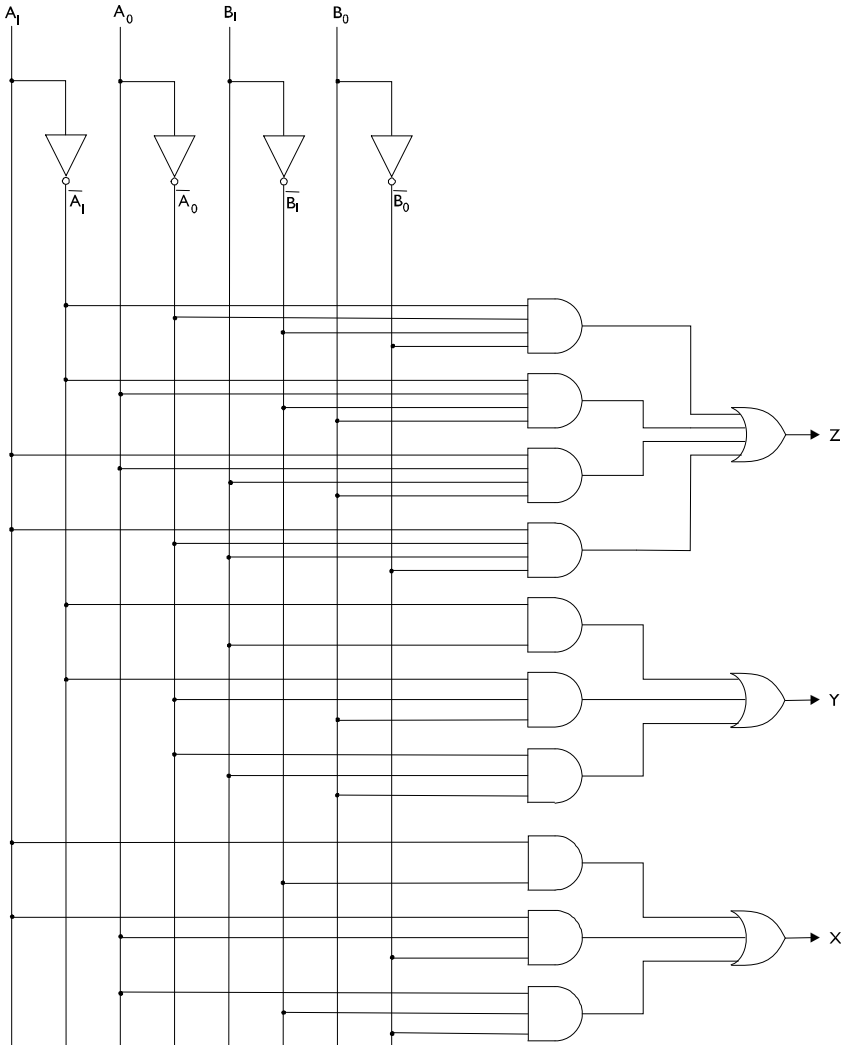
We can write down expressions for X, Y, and Z as

$X = A_1 \cdot \overline{B_1} + A_1 \cdot A_0 \cdot \overline{B_0} + A_0 \cdot \overline{B_1} \cdot \overline{B_0}$

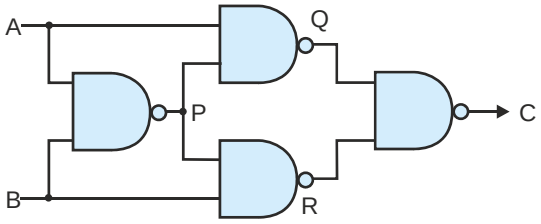
$Y = \overline{A_1} \cdot B_1 + \overline{A_1} \cdot \overline{A_0} \cdot B_0 + \overline{A_0} \cdot B_1 \cdot B_1$

$Z = \overline{A_1} \cdot \overline{A_0} \cdot \overline{B_1} \cdot \overline{B_0} + \overline{A_1} \cdot A_0 \cdot \overline{B_1} \cdot B_0 + A_1 \cdot A_0 \cdot B_1 \cdot B_0 + A_1 \cdot \overline{A_0} \cdot B_1 \cdot \overline{B_0}$

A circuit for the comparator constructed from AND, OR, and NOT gates is give below. Note that we have provided circuits for X and Y (although you might derive one from the other by means of an inverter).



40. Draw a truth table for the circuit below and explain what it does.

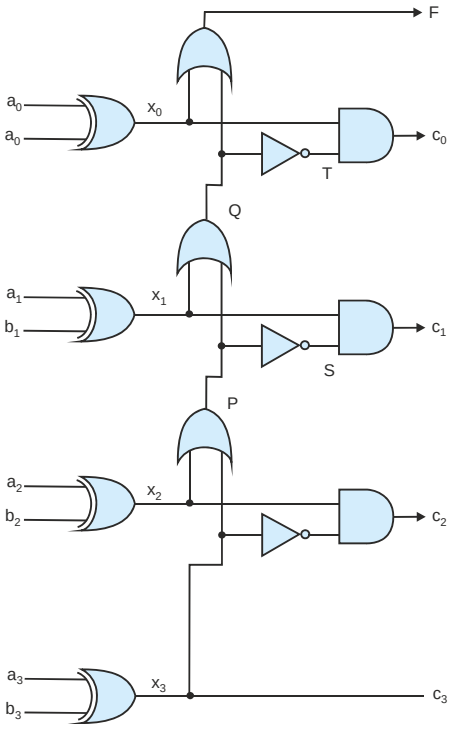


SOLUTION

A	B	P	Q	R	C
0	0	1	1	1	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	1	1	0

The output is true if one and only one input is true. This is the exclusive or function, XOR (EOR). It is also a two-bit binary adder without carry-out because it gives the sum of two bits.

41. The circuit below receives three pairs of inputs a_0, b_0, a_1, b_1, … a_3, b_3 and produces a four-bit output c_0, c_1, …, c_3. Analyze the circuit and explain what it does in plain English.



SOLUTION

Since all inputs are in pairs connected to XOR gates, we can simplify the truth table by just using the XOR outputs as variables. The diagram gives the intermediate variables to help us draw the truth table. As you can see the circuit is a priority encoder so that c3 is asserted if X3 is asserted, C2 is asserted if X2 is asserted (but not X3), C1 is asserted if X1 is asserted (but not C2 and C3). In practice this means that if two 4-bit words are matched, the C outputs correspond to the highest matching bits. Note that F is asserted only of there are no matching bits.

X3	X2	X1	X0	P	Q	F	S	T	C3	C2	C1	C0
0	0	0	0	0	0	0	1	1	0	0	0	0
0	0	0	1	0	0	1	1	1	0	0	0	1
0	0	1	0	0	1	1	1	0	0	0	1	0
0	0	1	1	0	1	1	1	0	0	0	1	0
0	1	0	0	1	1	1	0	0	0	1	0	0
0	1	0	1	1	1	1	0	0	0	1	0	0
0	1	1	0	1	1	1	0	0	0	1	0	0
0	1	1	1	1	1	1	0	0	0	1	0	0
1	0	0	0	1	1	1	0	0	1	0	0	0
1	0	0	1	1	1	1	0	0	1	0	0	0
1	0	1	0	1	1	1	0	0	1	0	0	0
1	0	1	1	1	1	1	0	0	1	0	0	0
1	1	0	0	1	1	1	0	0	1	0	0	0
1	1	0	1	1	1	1	0	0	1	0	0	0
1	1	1	0	1	1	1	0	0	1	0	0	0
1	1	1	1	1	1	1	0	0	1	0	0	0

42. Demonstrate that all logic circuits can be constructed from NOR gates by building an inverter, an *and* gate, and an *or* gate from one or more *nor* gates.

SOLUTION

The logic function for a NOR gate is $C = \overline{A + B}$.

If we connect its two inputs together so that B = A we get $C = A + \overline{A = A}$; that is the circuit acts as an inverter.

Suppose we put two inverters in the input path of a NOR gate (the inverters themselves constructed from NORs). We get

$C = \overline{\overline{A} + \overline{B}} = \overline{A} \cdot \overline{B} = A \cdot B$; that is, the circuit functions as an AND gate.

If we put an inverter on the output of a NOR gate we get

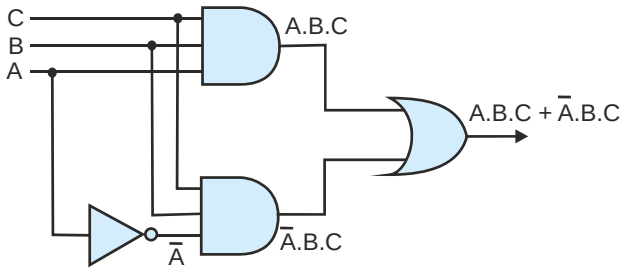
$C = \overline{\overline{A + B}} = A + B$ which is an OR gate.

So, with a NOR gate you can make a NOT gate, AND gate, and OR gate.

43. Design a logic circuit to implement the logical function $X = A \cdot B \cdot C + \overline{A} \cdot B \cdot C$

SOLUTION

The following circuit performs this operation. Note that this is a direct implementation of the Boolean algebra. We could have simplified the equation and, therefore, the circuit.



44. Simplify the following Boolean expressions.

a. $F = \overline{A}\cdot B\cdot C\cdot\overline{D} + A\cdot B\cdot C\cdot\overline{D} + A\cdot\overline{B}\cdot C\cdot D + A\cdot B\cdot C\cdot D$

b. $F = \overline{A}\cdot B\cdot C + A\cdot B\cdot C + A\cdot\overline{C}\cdot D + A\cdot B\cdot C\cdot D$

c. $F = \overline{A\cdot\overline{C}\cdot D + A\cdot B\cdot C\cdot D} + A\cdot B\cdot C$

SOLUTION

a. $F = \overline{A}\cdot B\cdot C\cdot\overline{D} + A\cdot B\cdot C\cdot\overline{D} + A\cdot\overline{B}\cdot C\cdot D + A\cdot B\cdot C\cdot D$
$= C\cdot D\cdot(A\cdot\overline{B} + A\cdot B) + \overline{A}\cdot B\cdot C\cdot\overline{D} + A\cdot B\cdot C\cdot\overline{D}$
$= C\cdot D\cdot A + B\cdot C\cdot\overline{D}\cdot(\overline{A} + A)$ *since* $A\cdot\overline{B} + A\cdot B = A$
$= C\cdot D\cdot A + B\cdot C\cdot\overline{D}$ *since* $\overline{A} + A = 1$

b. $F = \overline{A}\cdot B\cdot C + A\cdot B\cdot C + A\cdot\overline{C}\cdot D + A\cdot B\cdot C\cdot D$
$= B\cdot C\cdot(\overline{A} + A) + A\cdot\overline{C}\cdot D + A\cdot B\cdot C\cdot D$
$= B\cdot C + A\cdot D\cdot(\overline{C} + B\cdot C)$
$= B\cdot C + A\cdot D\cdot(\overline{C} + B)$
$= B\cdot C + A\cdot\overline{C}\cdot D + A\cdot B\cdot D$
$= B\cdot C + A\cdot\overline{C}\cdot D$

c. $F = \overline{A\cdot\overline{C}\cdot D + A\cdot B\cdot C\cdot D} + A\cdot B\cdot C$

$= \overline{\overline{A\cdot\overline{C}\cdot D}}\cdot\overline{A\cdot B\cdot C\cdot D} + A\cdot B\cdot C$ *(deMorgan)*
$= (\overline{A} + C + \overline{D})\cdot(\overline{A} + \overline{B} + \overline{C} + \overline{D}) + A\cdot B\cdot C$ *(deMorgan)*
$= \overline{A} + \overline{B}\cdot C + \overline{D} + A\cdot B\cdot C$
$= \overline{A} + \overline{B}\cdot C + \overline{D} + B\cdot C = \overline{A} + C + \overline{D}$

45. It is possible to have *n*-input AND, *or*, NAND, and NOR gates, where $n > 2$. Can you have an *n*-input XOR gate for $n > 2$? Explain your answer with a truth table.

SOLUTION

You can have AND, OR, NAND, and NOR gates with any number of inputs. They obey the rules of the simple 2-input gates (e.g., the output of a NAND gate is true if and only if each of its *n* inputs are true simultaneously).

The exclusive or gate is rather more complex. If you define it as a gate with two inputs whose output is true if one or both inputs are true, then you can have only a 2-input EOR (XOR) gate. However, if you say that the output of an XOR gate is y = a⊕b, then you can also have y = a⊕b⊕c⊕d. In this case the output is true if the number of inputs that are set to 1 is odd. This is a parity detector.

46. Draw the truth table of a full subtractor that directly subtracts bit *A* from *B* together with a borrow-in, to produce a difference *D* and a borrow-out.

SOLUTION

The truth table is below:

B_{in}	A	B	B_{out}	D
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	0	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

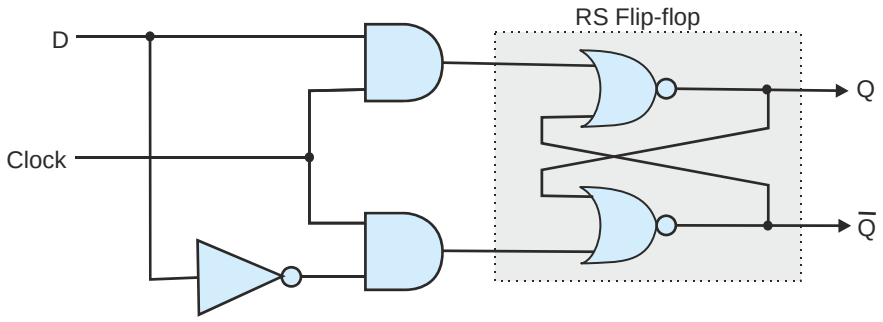
47. Design a D flip-flop using only an RS latch and simple gates. Include a circuit diagram, timing diagram, and truth table in your answer.

SOLUTION

There are several ways of tackling this problem. Let's start with an RS latch made from NOR gates. This has an R input and an S input that can be used to set or reset the flip-flop. If both inputs are 0, the outputs remain the same. If R = 1, S = 0, Q is cleared and if S = 1, R = 0, Q is set. The state S = R = 1 should be avoided.

So, we use two AND gates to feed the R,S inputs. One input to of each of the two AND gates is the clock. When the clock is 0, the AND gates both have 0 outputs and R = S = 0. Consequently Q remains what it was, unchanged.

When the clock is 1, the AND gates are enabled. One input is D and the other NOT D (via an inverter). This assumes that R,S = 0,1 or 1,0. Consequently, the Q output is set if D is 1 and cleared if D is 0. This is a D flip-flop.



48. Without the tristate gate, it would be almost impossible to design a modern computer. Why is this so?

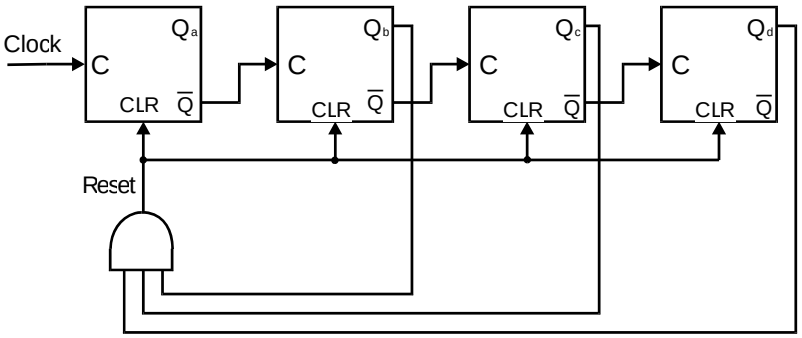
SOLUTION

Tristate gates are used to allow one of several devices to put data onto a common bus. A tristate device has three output states: high (driving the bus), low (driving the bus), and high-impedance (disconnected from the bus). The tristate gate makes it easy to design bused systems. The output of any tri-state device connected to a bus has tristate control logic. This allows one and only one device connected to a bus to drive the bus. All other devices connected to the bus are disconnected electrically by assuming the third *state*.

Without tristate gates you would have to drive buses with an alternative technology (e.g., open-collector or open-drain circuits). However, these are very similar to tristate logic in the sense that they allow one out of *n* devices to drive the bus.

If there were no tristate drives (or open-collector/open-drain) the only other way that a buses system would be possible would be to use multiplexers to control buses. This would be a rather cumbersome solution.

49. The circuit below consists of four JK flip-flops. Inputs J and K are not shown because it is assumed that they are both permanently connected to a logical 1. These JK flip-flops are positive edge triggered (i.e., they change state on the rising edge of the clock). Note that these flip-flops have a CLR (clear) input that sets Q to 1 when CLR = 1. What does this circuit do?

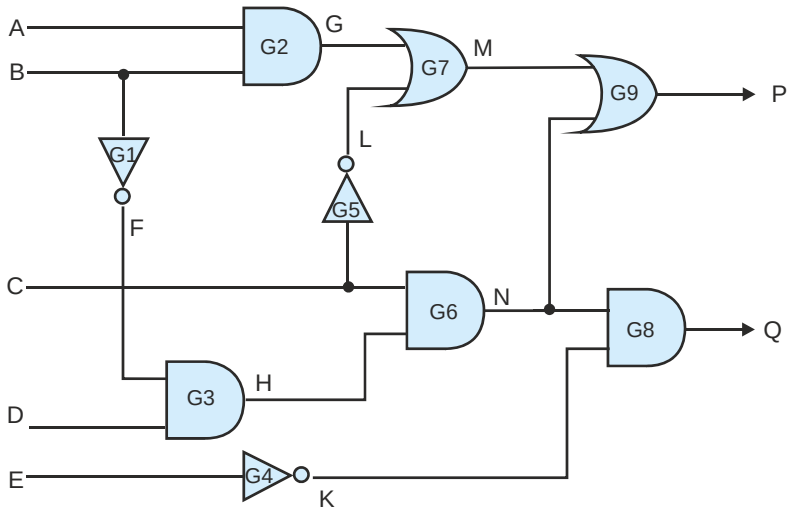


SOLUTION

The problem can be solved with a simple timing diagram. However, we can explain its action just by inspection. The JK flip-flops are configured as counters (J,K = 1,1 forces the output to toggle on each clock pulse). If JK flip-flops are positive-edge triggered and their Q outputs are connected to the clock input of the next stage, they will count down. If we connect the NOT Q outputs to the clocks they will count up. Consequently, we have a binary up counter.

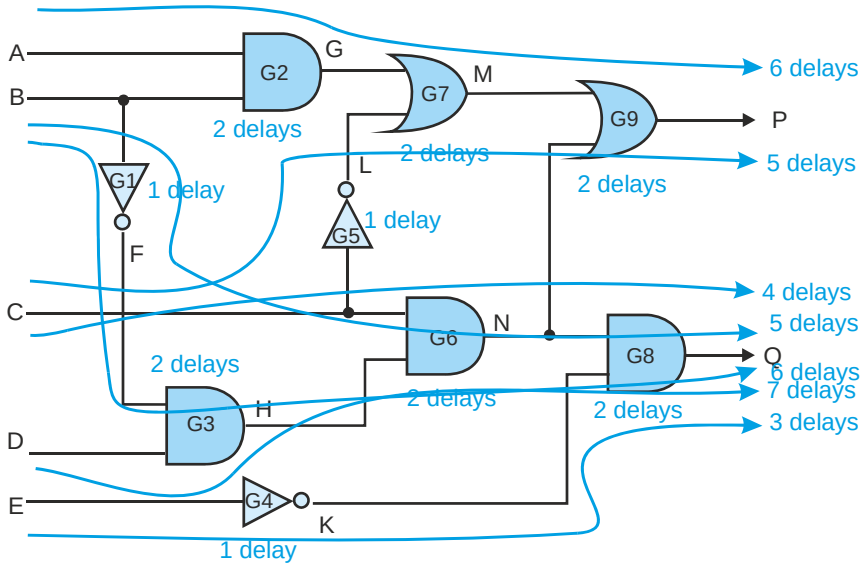
Note the AND gate. This detects Q_d,Q_c.Q_b = 1,1,1. The state of Q_a does not matter, so this value will be reached when Q_d,Q_c.Q_b,Q_a = 1110 which is 13. When that happens, all flip-flops will be set to zero. This is a modulo 13 counter that counts 0,1,2, …10,11,12, 0, 1, 2 , …

50. Consider the circuit below. Assume that all gates are implemented in silicon by NAND gates, NOR gates and inverters. Each NAND gate, NOR gate, or inverter has an internal delay of 0.4 ns. A logic transition at an input may cause a change at the output (depending on other inputs and the circuit). The time for a transition at the input to appear as a corresponding transition at an output depends on the circuit path and the nature of the gates. What is the longest circuit path through this circuit and what is the worst case delay that a signal experiences going through the circuit?

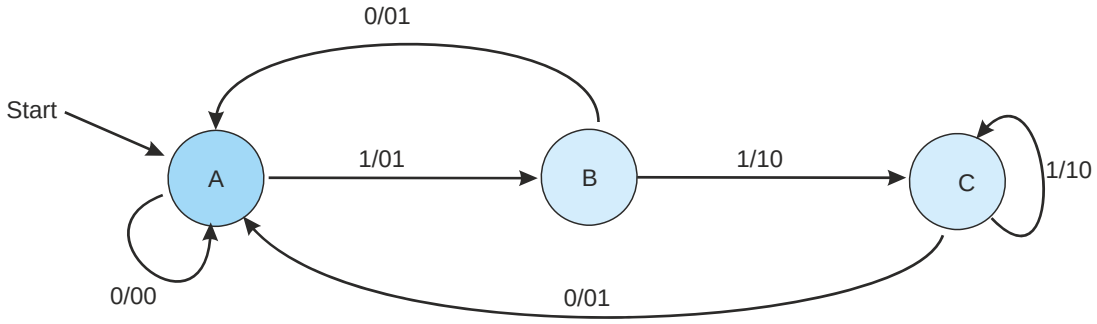


SOLUTION

The following diagram demonstrates all the paths through this circuit and their delays. Remember an AND gate is a NAND gate plus an inverter (and an OR gate is a NOR gate plus an inverter). Consequently, all AND and OR gates suffer two delay units. The longest path is from B via G1, G3, G6, G8 and is 7 delays or 7 × 0.4 = 2.8 ns.



51. The following state diagram describes a system with states A, B, and C. The system is initialized in State A. The state transition notation x/yz indicates that an input x causes a transition in the direction shown and the system outputs the value yz; for example, an input 1 in State A causes a transition to State B and the system outputs 01.



a. How many flip-flops would be required to construct this system?
b. If the system receives the input 000100110010101111110010, what would the output be?

SOLUTION

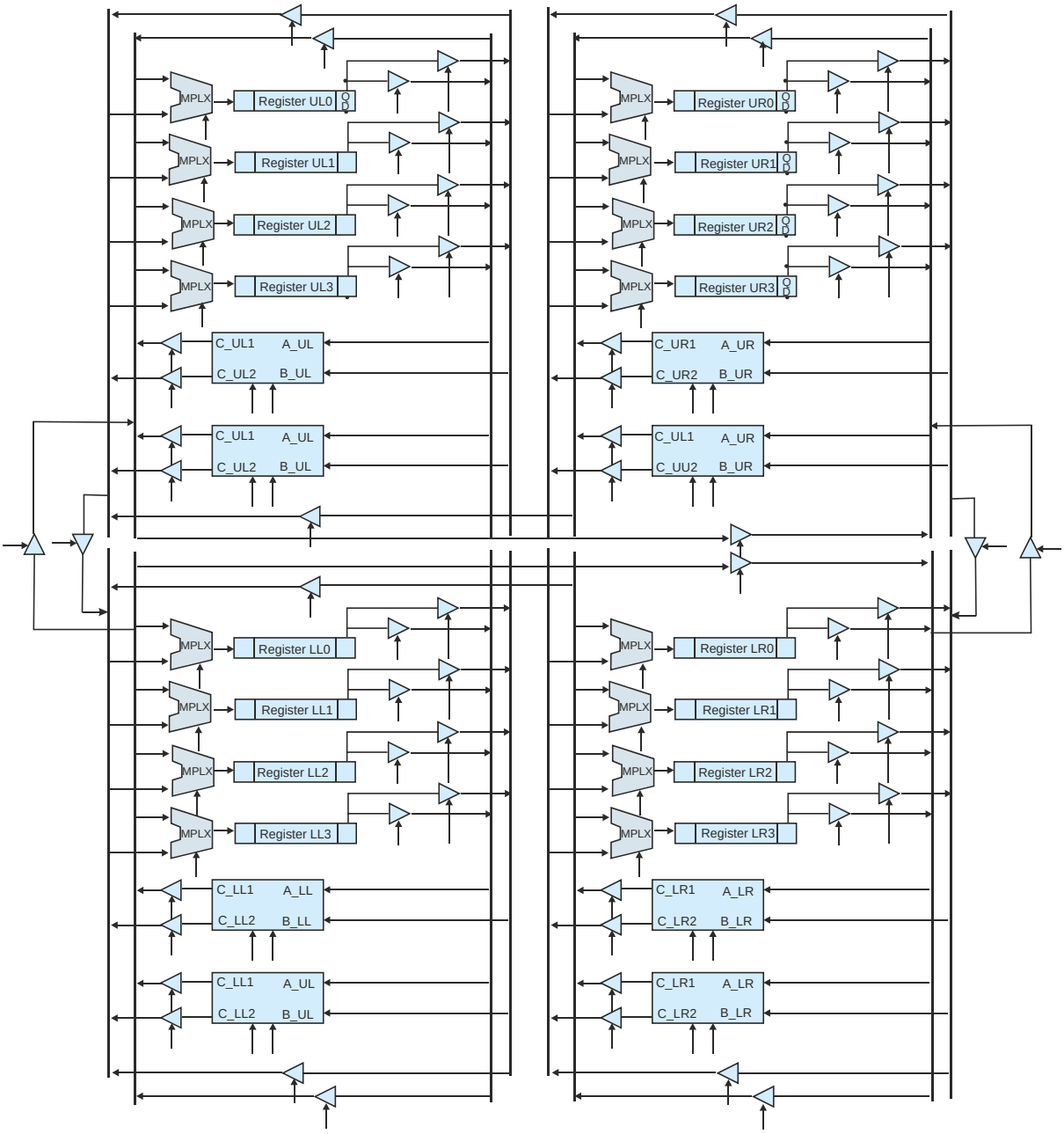
a. This system has three states and requires two flip-flops (which can store up to four states).

b. Consider the following table that gives the current state, input, next state, and output.

Current state	Input	Next state	Output
A	0	A	00
A	0	A	00
A	0	A	00
A	1	B	01
B	0	A	01
A	0	A	00
A	1	B	01
B	1	C	10
C	0	A	01
A	0	A	00
A	1	B	01
B	0	A	01
A	1	B	01
B	0	A	01
A	1	B	01
B	1	C	10
C	1	C	10
C	1	C	10
C	1	C	10
C	0	A	01
A	0	A	00
A	1	B	01
B	0	A	01

52. The following structure contains registers, ALUs, multiplexers, tristate gates and buses, and is essentially a more elaborate form of the register, ALU structure we introduced in this chapter. As a computing structure, what are the advantages of this over the simpler system we described earlier?

SOLUTION



Essentially, this replicates a basic register and ALU circuit four times. In each quadrant data can be moved from registers to the ALU and back to registers. Note that the four buses and two ALUs permit parallel operations (two ALU operations at a time).

By repeating the system four times, up to eight operations can take place simultaneously. This system is intended to demonstrate parallelism in digital systems.