ALLAMA IQBAL OPEN UNIVERSITY

AIOU LIBRARY

Name:

Registration No.:

Course Code: 3006

Program: BS

Semester: Autumn 2026

ALLAMA IQBAL OPEN UNIVERSITY, ISLAMABAD

(Department of Computer Science)

Course: Digital Logic Design (3006) • Level: BS

Semester: Autumn 2026

Assignment No. 1 (Units 1–4)

Total Marks: 100 • Pass Marks: 50

SOLVED ASSIGNMENT GUIDE

(For Study & Reference)

Note: This guide is prepared for study and reference purposes. Students are advised to read the relevant units of the course book, understand the method shown in each answer, and then write their assignments in their own words. Copying any solved guide word for word is against the university's rules, which are printed on the question booklet itself.

Transparency Note (course code): The question booklet prints the course as "Digital Logic Design (3006)". The source question file supplied for this booklet is named CS3006.doc in the assignment listing. This guide follows the code exactly as printed in the booklet — (3006) — in its title block, footer and file name.

Question 1: Number Conversions and Binary Subtraction Using 2's Complement (20 Marks)

Question as printed: (a). Convert the following numbers to decimal: (10) i). $(2331)_4$ ii). $(1215)_6$ iii). $(1245)_{14}$ iv). $(248)_9$ (b). Perform subtraction on the given unsigned binary numbers using the 2's complement of the subtrahend. Where the result should be negative, find its 2's complement and affix a minus sign. (10) i). 10001 – 0111 ii). 100 – 111100 iii). 111111 – 11011 iv). 1100011 – 10001

Introduction

Every digital computer works internally with the binary number system, yet the numbers a student meets in a course on digital logic design come written in many different bases — base 4, base 6, base 9, base 14 and others. Part (a) of this question therefore tests the most basic skill of the subject: reading a number written in an arbitrary base and restating its value in the familiar decimal (base-10) system. Part (b) tests the standard machine method of subtraction. Adding circuits are cheap and fast, so computers do not subtract by borrowing in the school manner; instead they convert the subtrahend into its 2's complement and add. This question asks us to carry out exactly that procedure by hand, including the important case in which the answer is negative and must be recognised and re-expressed with a minus sign. Both parts are worked below step by step, and every result is checked by an independent decimal calculation so that the method, and not merely the answer, is clear.

Part (a): The Positional (Place-Value) Method of Conversion

A number written in base r is a shorthand for a sum of powers of r. If the digits of the number are $d_n\ d_{n-1} \ldots d_1\ d_0$, then its decimal value is $d_n \times r^n + d_{n-1} \times r^{n-1} + \ldots + d_1 \times r^1 + d_0 \times r^0$. The leftmost digit is multiplied by the highest power and the units digit by $r^0 = 1$. Two points must be watched. First, every digit must be smaller than the base; all four numbers given satisfy this rule (for example, base 4 uses only the digits 0–3, and indeed $(2331)_4$ contains no digit of 4 or more). Second, the powers must be worked out carefully, because a single arithmetic slip spoils the whole conversion. The safest routine is to write the expansion in full, evaluate each term separately, and only then add.

(i) $(2331)_4 = 2 \times 4^3 + 3 \times 4^2 + 3 \times 4^1 + 1 \times 4^0 = 2 \times 64 + 3 \times 16 + 3 \times 4 + 1 \times 1 = 128 + 48 + 12 + 1 = 189$. So $(2331)_4 = (189)_{10}$.

(ii) $(1215)_6 = 1 \times 6^3 + 2 \times 6^2 + 1 \times 6^1 + 5 \times 6^0 = 1 \times 216 + 2 \times 36 + 1 \times 6 + 5 \times 1 = 216 + 72 + 6 + 5 = 299$. So $(1215)_6 = (299)_{10}$.

(iii) $(1245)_{14} = 1 \times 14^3 + 2 \times 14^2 + 4 \times 14^1 + 5 \times 14^0 = 1 \times 2744 + 2 \times 196 + 4 \times 14 + 5 \times 1 = 2744 + 392 + 56 + 5 = 3197$. So $(1245)_{14} = (3197)_{10}$. Base 14 is worth a moment of attention: bases above 10 normally use the letters A, B, C, D for the digit values 10, 11, 12, 13, but the given number uses only the ordinary digits 1, 2, 4 and 5, so no letter digits arise here.

(iv) $(248)_9 = 2 \times 9^2 + 4 \times 9^1 + 8 \times 9^0 = 2 \times 81 + 4 \times 9 + 8 \times 1 = 162 + 36 + 8 = 206$. So $(248)_9 = (206)_{10}$.

Given number	Expansion	Term values	Decimal value

$(2331)_4$	$2\times4^3 + 3\times4^2 + 3\times4^1 + 1\times4^0$	128 + 48 + 12 + 1	189
$(1215)_6$	$1\times6^3 + 2\times6^2 + 1\times6^1 + 5\times6^0$	216 + 72 + 6 + 5	299
$(1245)_{14}$	$1\times14^3 + 2\times14^2 + 4\times14^1 + 5\times14^0$	2744 + 392 + 56 + 5	3197
$(248)_9$	$2\times9^2 + 4\times9^1 + 8\times9^0$	162 + 36 + 8	206

Each result can be checked by dividing the decimal answer repeatedly by the base and collecting the remainders in reverse order. For example, 189 ÷ 4 = 47 remainder 1; 47 ÷ 4 = 11 remainder 3; 11 ÷ 4 = 2 remainder 3; 2 ÷ 4 = 0 remainder 2; reading the remainders upwards gives 2331, which is the original base-4 number. The same check applied to the other three answers reproduces $(1215)_6$, $(1245)_{14}$ and $(248)_9$, so all four conversions are confirmed.

Part (b): Subtraction by the 2's Complement Method

The rule for subtracting an unsigned binary number B from an unsigned binary number A by the 2's complement method has four steps. (1) Pad the shorter number with leading zeros so that both numbers have the same number of bits, n. (2) Form the 2's complement of the subtrahend B within these n bits: invert every bit (the 1's complement) and add 1. (3) Add this complement to A and keep only n bits of the sum. (4) Interpret the result by looking at the final carry out of the most significant position. If there is a carry out, discard it: the remaining n bits are the positive answer, because A was greater than or equal to B and the machine has in effect added

one extra 2^n which must be thrown away. If there is no carry out, the true answer is negative: the n bits obtained are the 2's complement form of the answer, so take their 2's complement again to get the magnitude and put a minus sign in front of it, exactly as the question instructs.

(i) 10001 − 0111. Pad the subtrahend to five bits: 0011 (decimal 3); the minuend is 10001 (decimal 17). The 2's complement of 0011 is 1100 + 1 = 1101, written with the leading bit, 11101. Add: 10001 + 11101 = 101110. There is a carry out of 1, which is discarded, leaving 01110. Result: 01110, that is, +14. Check: 17 − 3 = 14. Correct.

(ii) 100 − 111100. Here the minuend is the shorter number, so pad it to six bits: 000100 (decimal 4); the subtrahend is 111100 (decimal 60). The 2's complement of 111100 is 000011 + 1 = 000100. Add: 000100 + 000100 = 001000. This time there is no carry out, so the answer is negative. Take the 2's complement of 001000: invert to 110111 and add 1, giving 111000, whose value is 56. Result: − 111000, that is, − 56. Check: 4 − 60 = − 56. Correct. This is the one case among the four in which the subtrahend exceeds the minuend, and it is precisely the case the wording of the question warns about.

(iii) 111111 − 11011. Pad the subtrahend to six bits: 011011 (decimal 27); the minuend is 111111 (decimal 63). The 2's complement of 011011 is 100100 + 1 = 100101. Add: 111111 + 100101 = 10100100. Discard the carry out, leaving 100100. Result: 100100, that is, +36. Check: 63 − 27 = 36. Correct.

(iv) 1100011 − 10001. Pad the subtrahend to seven bits: 0010001 (decimal 17); the minuend is 1100011 (decimal 99). The 2's complement of 0010001 is 1101110 + 1 = 1101111. Add:

1100011 + 1101111 = 11010010. Discard the carry out, leaving 1010010. Result: 1010010, that is, +82. Check: 99 − 17 = 82. Correct.

Subtraction	Padded A – B	2's complement of B	Sum (n bits kept)	Carry out?	Result
10001 − 0111	10001 − 0011	11101	01110	Yes — discard	+01110 = +14
100 − 111100	000100 − 111100	000100	001000	No — negative	− 111000 = − 56
111111 − 11011	111111 − 011011	100101	100100	Yes — discard	+100100 = +36
1100011 − 10001	1100011 − 0010001	1101111	1010010	Yes — discard	+1010010 = +82

Two common mistakes deserve mention because they cost marks in examinations. The first is to form the 2's complement before padding: the complement depends on the number of bits, so 0111 in four bits has complement 1001, but padded to five bits as 00111 its complement is 11001, and only the padded version may be added to a five-bit minuend. The second mistake is to treat the result of the no-carry case as an ordinary positive binary number; 001000 in part (ii) is not +8 as the answer to the subtraction, it is the complement form from which the true answer − 56 still has to be recovered.

Conclusion

The positional expansion method converts any base-r number to decimal by summing digit-times-power terms, giving 189, 299, 3197 and 206 for the four numbers in part (a), each confirmed by the reverse division check. The 2's complement method of part (b) turns subtraction into addition: pad, complement the subtrahend, add, and read the carry. A carry out means a positive result (14, 36 and 82 in parts (i), (iii) and (iv)); no carry out means a negative result, whose magnitude is found by complementing again and affixing a minus sign (−56 in part (ii)). This is exactly the procedure a computer's arithmetic unit follows, which is why mastering it by hand is the real purpose of the question.

Question 2: Simplifying Boolean Expressions and Coding the 52 Playing Cards (20 Marks)

Question as printed: (a). Simplify the following Boolean expressions to a minimum number of literals (10) i). xyz + xyz' + xy' ii). (xy + z)(xy + z') iii). xyz' + xy'z + xz' iv) xyz + z(xy + x') (b) Assign a binary code in some orderly manner to the 52 playing cards. Use the minimum number of bits.. (10)

Introduction

Boolean algebra is the mathematics of two-valued logic, and its practical value in digital design is that a complicated switching expression can usually be replaced by a much simpler one that behaves identically but needs fewer gates, fewer inputs and less wiring. Part (a) asks for four such simplifications, each carried down to the minimum number of literals, where a literal is a

single variable in either its true or its complemented form. Part (b) turns from algebra to coding: it asks for an orderly binary code for the 52 cards of an ordinary pack, using as few bits as possible. The two parts are connected by a common theme — economy. In part (a) we economise on literals inside an expression; in part (b) we economise on the bits used to represent information, while keeping the code systematic enough that a card can be read off from its codeword at a glance.

Part (a): Laws Used and the Four Simplifications

The simplifications below use only the standard postulates and theorems of Boolean algebra: the complement law $X + X' = 1$ and $X \cdot X' = 0$; the identity laws $X + 0 = X$, $X \cdot 1 = X$, $X + 1 = 1$ and $X \cdot 0 = 0$; the idempotent law $X + X = X$; the distributive laws; and above all the combining (adjacency) theorem $XY + XY' = X$, which says that two product terms differing in exactly one literal merge into the common part. A useful companion form is $X + X'Y = X + Y$. Each reduction is shown step by step, with the law used stated at each step, and each final answer has been verified against the original expression over all eight combinations of x, y and z.

(i) $xyz + xyz' + xy'$. The first two terms differ only in the literal z, so by the combining theorem $xyz + xyz' = xy(z + z') = xy \cdot 1 = xy$. The expression becomes $xy + xy'$, whose two terms now differ only in y, so the combining theorem applies again: $xy + xy' = x(y + y') = x$. Final answer: x — a single literal. The result is easy to believe: whenever $x = 0$ every term of the original expression contains x and is therefore 0, and whenever $x = 1$ at least one term is 1 for every combination of y and z, so the function is simply a copy of the input x.

(ii) (xy + z)(xy + z′). This is a product of sums in which the same sum term xy appears in both factors. The distributive law in its less familiar form, (A + z)(A + z′) = A + zz′ = A + 0 = A, applies with A = xy: expanding, (xy + z)(xy + z′) = xy·xy + xy·z′ + z·xy + zz′ = xy + xyz′ + xyz + 0 = xy(1 + z′ + z) = xy. Final answer: xy — two literals. In words, the output is 1 exactly when both x and y are 1; the variable z, which appears in the original expression, in fact has no influence on the output at all.

(iii) xyz′ + xy′z + xz′. Factor x from all three terms: x(yz′ + y′z + z′). Inside the bracket, yz′ + z′ = z′(y + 1) = z′ by the identity law y + 1 = 1, so the bracket reduces to z′ + y′z. Now z′ + y′z = z′ + y′ by the theorem X + X′Y = X + Y with X = z′. Hence the whole expression is x(z′ + y′) = xz′ + xy′. Final answer: xz′ + xy′ — four literals. No further reduction is possible: the function is 1 only when x = 1, and then for every combination of y and z except y = z = 1, which is exactly what the sum of the two terms xz′ (covers y arbitrary, z = 0) and xy′ (covers y = 0, z arbitrary) expresses; each term is needed for one of the two arms of that pattern.

(iv) xyz + z(xy + x′). First distribute z over the bracket: z(xy + x′) = xyz + x′z, so the expression becomes xyz + xyz + x′z. By the idempotent law xyz + xyz = xyz, leaving xyz + x′z. Factor z: z(xy + x′). Now xy + x′ = x′ + xy, and by the theorem X′ + XY = X′ + Y (the same rule used in part (iii), with the roles exchanged) this equals x′ + y. Hence the expression is z(x′ + y) = x′z + yz. Final answer: x′z + yz — four literals. A direct check confirms it: the output is 1 only when z = 1, and with z = 1 it is 1 whenever x = 0 (any y) or y = 1 (any x), which is precisely the union covered by the two product terms.

Expression as printed	Key steps	Minimum form	Literals

xyz + xyz′ + xy′	Combine on z, then on y	x	1
(xy + z)(xy + z′)	(A + z)(A + z′) = A with A = xy	xy	2
xyz′ + xy′z + xz′	Factor x; z′ absorbs yz′; z′ + y′z = z′ + y′	xz′ + xy	4
xyz + z(xy + x′)	Distribute; xyz + xyz = xyz; x′ + xy = x′ + y	xz + yz	4

Part (b): A Minimum-Bit Binary Code for the 52 Playing Cards

The first task is to find the minimum number of bits. With n bits we can form 2^n different codewords. Four bits give only 16 codewords and five bits give 32, both fewer than 52, so five bits cannot name every card. Six bits give 64 codewords, which is enough, with 12 codewords to spare. The minimum is therefore 6 bits. A code that simply numbered the cards 0 to 51 in binary would use 6 bits, but it would not be “orderly” in any useful sense: nothing in the codeword would tell the reader the suit or the rank of the card. The standard orderly solution is a structured code with two fields — a suit field and a rank field — so that the codeword itself displays the identity of the card.

Use the first (two most significant) bits for the suit. There are four suits, and two bits give exactly the four combinations needed: 00 for Spades, 01 for Hearts, 10 for Diamonds and 11 for Clubs. Use the remaining four bits for the rank. There are thirteen ranks — Ace, 2, 3, …, 10,

Jack, Queen, King — and four bits give sixteen combinations, so the ranks are coded by their natural numbers: 0001 for the Ace, 0010 for the 2, and so on up to 1010 for the 10, 1011 for the Jack, 1100 for the Queen and 1101 for the King. The rank codes 0000, 1110 and 1111 are not used, and neither are they needed: 4 suits × 13 ranks = 52 code words are formed out of the 64 available, the 12 unused words being exactly the three unused rank values in each of the four suits.

Suit	Suit code (2 bits)
Spades	00
Hearts	01
Diamonds	10
Clubs	11

Rank	Rank code (4 bits)	Rank	Rank code (4 bits)
Ace	0001	8	1000
2	0010	9	1001
3	0011	10	1010
4	0100	Jack	1011
5	0101	Queen	1100
6	0110	King	1101

7	0111	(unused)	0000, 1110, 1111

The complete codeword of a card is the suit code followed by the rank code. Examples: the Ace of Spades is 00 0001 = 000001; the 7 of Hearts is 01 0111 = 010111; the Queen of Diamonds is 10 1100 = 101100; the King of Clubs is 11 1101 = 111101; and the 10 of Spades is 00 1010 = 001010. Decoding is just as direct: the codeword 011001 begins with 01, so the suit is Hearts, and 1001 is the rank 9, so the card is the 9 of Hearts. The code is orderly in a second sense as well: within any one suit, the codewords increase in binary counting order as the rank rises from Ace to King, and the four suits occupy four consecutive blocks of sixteen codewords, so the whole pack can be listed, sorted or searched by treating each codeword as an ordinary binary number.

It is worth stating why no 5-bit scheme, however cleverly arranged, can do the same job. The argument is a counting argument and does not depend on the arrangement at all: 52 distinct cards need 52 distinct codewords, and 5 bits supply only 32. Six bits is therefore not merely a convenient choice but a provable minimum, and the two-field arrangement shows that the minimum can be achieved while keeping the code readable.

Conclusion

Repeated application of the combining theorem and its companion rules reduces the four expressions of part (a) to x, xy, xz′ + xy′ and x′z + yz respectively, each verified against the original over every input combination. Part (b) needs a minimum of 6 bits for 52 cards, and splitting those bits into a 2-bit suit field and a 4-bit rank field gives an orderly code in which

every codeword can be decoded by inspection and the cards of each suit appear in natural rank order.

Question 3: K-Map Simplification and Two-Level NOR Implementation (20 Marks)

Question as printed: a). Simplify the following Boolean Expressions, using four-variable K-maps (10) i). F(w, x, y, z) = Σ(1, 3, 5, 7, 9, 11, 13, 15) ii). F(w, x, y, z) = Σ(0, 1, 2, 5, 8, 13) (b). Simplify the following functions, and implement them with two-level NOR gate circuits (10) i). F(w, x, y, z) = wx'y + xy'z' + w'yz' + xy ii). F(w, x, y, z) = Σ(4, 7, 11, 12, 15)

Introduction

The Karnaugh map (K-map) is the graphical form of the combining theorem: cells that are adjacent on the map — including cells that meet across the edges of the map — differ in exactly one variable, so a rectangular group of 1-cells of size 2, 4, 8 or 16 can be covered by a single product term containing only the variables that keep the same value throughout the group. Part (a) applies this method to two functions given in minterm form. Part (b) goes one step further into implementation. A two-level NOR circuit is the natural hardware form of a product-of-sums (POS) expression, just as a two-level NAND circuit is the natural form of a sum-of-products (SOP) expression; so for each function we first simplify, then obtain a minimum POS form by simplifying the complement on a K-map and applying De Morgan's theorem, and finally read off the NOR–NOR circuit.

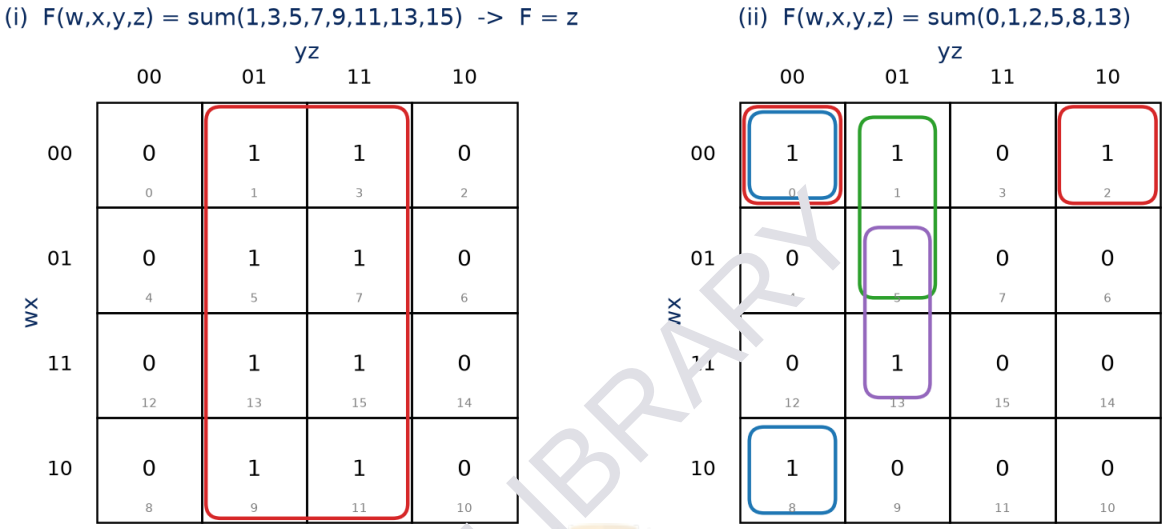
Part (a)(i): F = Σ(1, 3, 5, 7, 9, 11, 13, 15)

The eight minterms are exactly the odd numbers from 1 to 15, that is, exactly the combinations for which the last variable z = 1. On the four-variable map (Figure 1, left) the 1s fill the two complete columns headed yz = 01 and yz = 11 — the whole right-hand half of the map. This is a single group of eight cells. Within the group, w takes both values, x takes both values and y takes both values, so w, x and y all drop out; the only variable that is constant, at the value 1, is z. The simplified function is therefore F = z. The map has reduced an eight-minterm expression to a single literal: the circuit is just a wire from the input z to the output.

Part (a)(ii): F = Σ(0, 1, 2, 5, 8, 13)

Plotting the six minterms (Figure 1, right) gives a scattered pattern with no group larger than a pair, so four pair-groups are required. Minterm 2 can be paired only with minterm 0 (they differ only in y), giving the term w'x'z' — the group is essential. Minterm 8 can be paired only with minterm 0 (differing only in w), giving x'y'z' — also essential; note that this pair is a wrap-around pair, joining the top row of the map to the bottom row. Minterm 13 pairs with minterm 5 (differing only in w), giving xy'z — essential, since minterm 13 has no other neighbour among the 1s. These three groups cover minterms 0, 2, 5, 8 and 13, leaving only minterm 1 uncovered; it is paired with minterm 5 (differing only in x), giving w'y'z. The simplified function is F = w'x'z' + x'y'z' + xy'z + w'y'z. (Pairing minterm 1 with minterm 0 instead, giving w'x'y' in place of w'y'z, is an equally minimal alternative with the same four terms and twelve literals.) No three-term or larger-group covering exists, as the essential pairs already force three terms and minterm 1 cannot be absorbed by any of them.

Figure 1: Four-variable K-maps for Question 3(a)



Part (b)(i): F = wx'y + xy'z' + w'yz' + xy

First simplify as a sum of products. On the K-map the function has 1s at minterms 2, 4, 6, 7, 10, 11, 12, 14 and 15, and the minimum cover consists of four groups of four: the group with x = 1, z = 0 gives xz'; the group with w = 1, y = 1 gives wy; the group with y = 1, z = 0 gives yz'; and the group with x = 1, y = 1 gives xy. Hence F = xz' + wy + yz' + xy (eight literals, against thirteen in the expression as printed).

For the NOR implementation we need the product-of-sums form, and the reliable route is through the complement. The 0-cells of F are minterms 0, 1, 3, 5, 8, 9 and 13, and the K-map of F' (Figure 2, left) groups them as F' = x'y' + y'z + w'x'z: the four corners-pattern group x'y' covers 0, 1, 8 and 9; the group y'z covers 1, 5, 9 and 13; and the pair w'x'z covers 1 and 3. Complementing both sides and applying De Morgan's theorem term by term — (x'y')' = x + y, (y'z)' = y + z', (w'x'z)' = w + x + z' — gives the minimum POS form F = (x + y)(y + z')(w + x + z').

A two-level NOR circuit realises a POS expression directly, because a NOR gate fed with the literals of one sum term produces the complement of that sum, and a final NOR gate combining those outputs restores the product: ((x + y)′ + (y + z′)′ + (w + x + z′)′)′ = (x + y)(y + z′)(w + x + z′). The circuit therefore has a first level of three NOR gates — a 2-input NOR with inputs x and y, a 2-input NOR with inputs y and z′, and a 3-input NOR with inputs w, x and z′ — and a second level consisting of one 3-input NOR gate whose inputs are the three first-level outputs and whose output is F. One inverter (or a NOR gate with its inputs tied together) supplies the complemented literal z′. Total hardware: four NOR gates and one inverter.

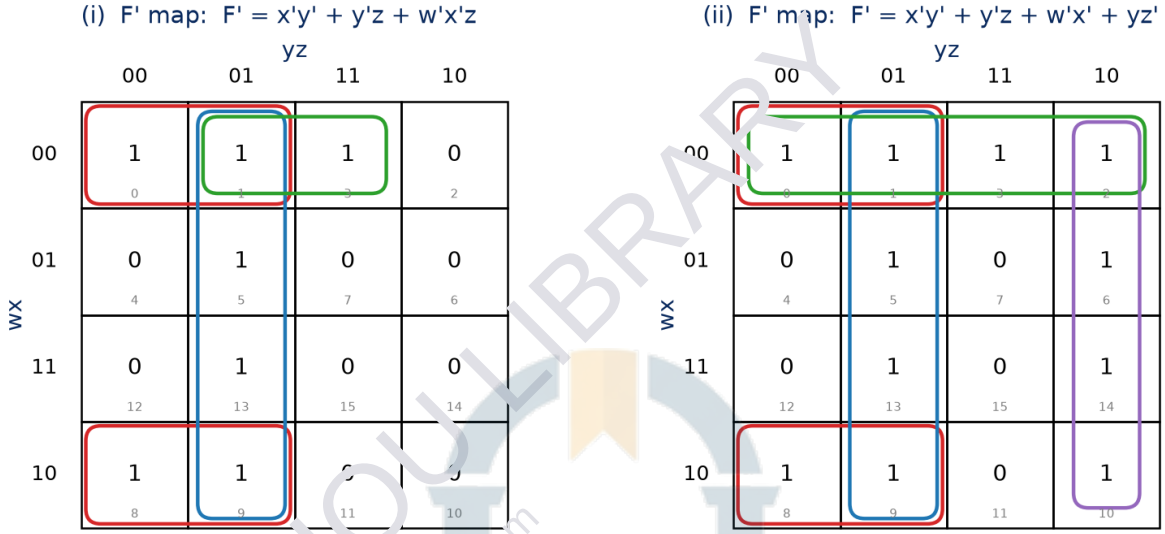
Part (b)(ii): F = Σ(4, 7, 11, 12, 15)

As a sum of products, the K-map groups are: minterm 15 with 7 and 11 forming, together with the pair structure of the map, the term xyz (minterms 7 and 15) and wyz (minterms 12 and 15), and minterm 4 pairing with 12 to give xy′z′; a minimum SOP form is F = xyz + xy′z′ + wyz. For the NOR form, the complement F′ has 1s at minterms 0, 1, 2, 3, 5, 6, 8, 9, 10, 13 and 14, and its K-map (Figure 2, right) gives F′ = x′y′ + y′z + w′x′ + yz′: the group x′y′ covers 0, 1, 8, 9; the group y′z covers 1, 5, 9, 13; the group w′x′ covers 0, 1, 2, 3; and the group yz′ covers 2, 6, 10, 14. Applying De Morgan's theorem gives the minimum POS form F = (x + y)(y + z′)(w + x)(y′ + z).

The two-level NOR circuit follows the same pattern as part (i): a first level of four NOR gates, one for each sum term — inputs (x, y), (y, z′), (w, x) and (y′, z) respectively — and a second level of one 4-input NOR gate combining the four first-level outputs to produce F. Two inverters are needed, one for z′ and one for y′. Total hardware: five NOR gates and two inverters. Both POS

forms in this part have been checked cell by cell against the original functions over all sixteen input combinations.

Figure 2: K-maps of the complement F' for the NOR implementations in Question 3(b)



Function	Minimum SOP	Minimum POS (for NOR)	Two-level NOR hardware
3(b)(i)	xz′ + wy + yz′ + xy	(x + y)(y + z′)(w + x + z′)	3 NOR (first level) + 1 NOR (output); 1 inverter for z′
3(b)(ii)	xyz + xy′z′ + wyz	(x + y)(y + z′)(w + x)(y′ + z)	4 NOR (first level) + 1 NOR (output); inverters for z′ and y′

Conclusion

The K-map reduces part (a)(i) to the single literal z, and part (a)(ii) to four product terms, w'x'z' + x'y'z' + xy'z + w'y'z. For part (b), simplifying the complement and applying De Morgan's theorem converts each function into a minimum product of sums form, which a two-level NOR–NOR circuit then implements directly — four NOR gates for part (i) and five for part (ii), plus the inverters needed for complemented input literals.

Question 4: Analysis of a Gate Circuit, and a Gray-Code-to-Binary Converter with a Verilog Model (20 Marks)

Question as printed: a) Obtain the simplified Boolean expressions for output F and G in terms of the input variables in the circuit given below [circuit printed in the booklet: input A passes through an inverter giving A'; B and C enter an AND gate; A' and D enter a NAND gate; the circuit is re-drawn and traced gate by gate in the answer]. b). Design a combinational circuit that converts a four-bit Gray code (Table 1.6) to a four-bit binary number. Implement the circuit with exclusive-OR gates. Using a case statement, write and verify a Verilog model of the circuit. (20)

Transparency Note (marks): No marks figure is printed against part (a) of this question, while part (b) is marked (20) and the booklet's general instructions state that all questions carry equal marks. The marks figures printed for the other parts of this assignment (10 + 10 for each of Questions 1–3 and 20 for Question 5) already sum to 100 only if part (a) is treated as carrying no separate printed figure. Both parts are solved in full below; students may confirm the intended split of marks with their tutor.

Introduction

This question covers the two directions of combinational logic work. Part (a) is analysis: a ready-made gate circuit is given, and the task is to work backwards from the diagram to the Boolean expressions it realises, simplified as far as possible. Part (b) is design: starting from the specification of a Gray-code-to-binary converter, we must produce the truth table, derive the output expressions, draw the circuit — here specifically with exclusive-OR gates — and then describe the same circuit in the Verilog hardware description language using a case statement, and verify the model by simulation. The Verilog model printed below has actually been compiled and simulated with a self-checking testbench; the simulation output quoted is the real output of that run, and it confirms all sixteen conversions.

Part (a): Tracing the Circuit Gate by Gate

Label the intermediate signals as follows. The inverter on input A produces A′, exactly as labelled in the printed diagram. The AND gate fed by B and C produces the product BC. The lower NAND gate has inputs A′ (taken from the vertical A′ line) and D, so its output is H = (A′ · D)′ = A + D′ by De Morgan's theorem. The OR gate has inputs A′ and BC, so its output is Y = A′ + BC. The upper NAND gate has inputs A′ and H (the output of the lower NAND gate, carried up by the vertical line), so its output is (A′ · H)′. The final AND gate for F combines this NAND output with Y, and the final AND gate for G combines Y with H. Hence, before simplification, F = (A′H)′ · (A′ + BC) and G = (A′ + BC) · H.

Gate (as printed)	Inputs	Output

Inverter	A	A′
AND (left)	B, C	BC
NAND (lower)	A′, D	H = (A′D)′ = A + D′
OR (middle)	A′, BC	Y = A′ + BC
NAND (upper)	A′, H	(A′H)′ = A + D (after simplification)
AND (output, for F)	(A′H)′, Y	F
AND (output, for G)	Y, H	G

Simplify F first. The upper NAND output is (A′H)′ = A + H′ (De Morgan), and H′ = (A + D′)′ = A′D, so (A′H)′ = A + A′D = A + D by the rule X + X′Y = X + Y. Therefore F = (A + D)(A′ + BC). Expanding, F = ABC + A′D + BCD; but the term BCD is redundant, because whenever B = C = D = 1 the output is already covered — by ABC if A = 1 and by A′D if A = 0 — so it can be dropped by the consensus theorem. The simplified expression is F = A′D + ABC. In words: F is 1 when A = 0 and D = 1 (whatever B and C are), or when A = B = C = 1 (whatever D is).

Now simplify G = Y · H = (A′ + BC)(A + D′). Expanding, G = A′A + A′D′ + ABC + BCD′ = A′D′ + ABC + BCD′, since A′A = 0. The term BCD′ is again redundant by consensus: when B = C = 1 and D = 0, the output is already 1 through ABC if A = 1 and through A′D′ if A = 0. The simplified expression is G = A′D′ + ABC. Comparing the two results is instructive: F and G share the term ABC, and differ only in the other term — A′D for F, A′D′ for G — so that together F + G = A′(D + D′) + ABC = A′ + ABC and F · G = ABC; the two outputs are never both produced by the A′-term at the same

time, because D cannot be both 0 and 1. Both simplifications have been verified against the gate-by-gate expressions over all sixteen combinations of A, B, C and D.

Part (b): Designing the Gray-Code-to-Binary Converter

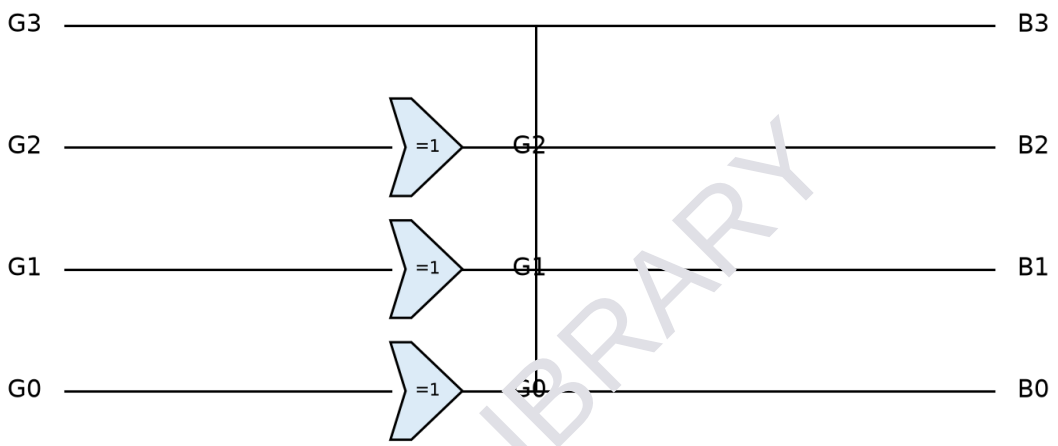
In the Gray code (Table 1.6 of the course book), successive code words differ in exactly one bit, which is why the code is used in shaft encoders and other electromechanical systems: a position read during a transition can be wrong in only one bit. Ordinary arithmetic, however, needs natural binary, so a converter is required. Let the Gray input be $G_3\ G_2\ G_1\ G_0$ (G_3 most significant) and the binary output $B_3\ B_2\ B_1\ B_0$. The truth table below lists all sixteen words; it is simply Table 1.6 read from the Gray column to the binary column.

Decimal	Gray code $G_3G_2G_1G_0$	Binary $B_3B_2B_1B_0$
0	0000	0000
1	0001	0001
2	0011	0010
3	0010	0011
4	0110	0100
5	0111	0101
6	0101	0110
7	0100	0111
8	1100	1000

9	1101	1001
10	1111	1010
11	1110	1011
12	1010	1100
13	1011	1101
14	1001	1110
15	1000	1111

Reading the table column by column reveals the pattern that makes the XOR implementation possible. The most significant binary bit equals the most significant Gray bit: $B_3 = G_3$. Each remaining binary bit is the exclusive-OR of the Gray bit in its own position with the binary bit just produced above it: $B_2 = B_3 \oplus G_2$, $B_1 = B_2 \oplus G_1$, and $B_0 = B_1 \oplus G_0$. Equivalently, each binary bit is the XOR (parity) of all the Gray bits from the top down to its own position: $B_2 = G_3 \oplus G_2$, $B_1 = G_3 \oplus G_2 \oplus G_1$, $B_0 = G_3 \oplus G_2 \oplus G_1 \oplus G_0$. This happens because the Gray code is formed from binary by the rule G = B ⊕(B shifted right by one), and XOR is its own inverse, so unwinding the rule from the top bit downwards recovers B. A K-map of each output confirms that no simpler AND–OR form exists; the XOR cascade is the minimal and natural implementation, and it needs just three 2-input exclusive-OR gates.

Figure 3: Gray-code-to-binary converter built from exclusive-OR gates (Question 4(b))



In the circuit of Figure 3, G_3 passes straight through as B_3. The first XOR gate combines B_3 with G_2 to give B_2; the second combines B_2 with G_1 to give B_1; the third combines B_1 with G_0 to give B_0. The gates are chained, so the total delay is three XOR-gate delays — a small price for such a compact circuit, and an acceptable one for a converter of this size.

The Verilog Model (Case Statement) and Its Verification

A case statement describes the converter directly from its truth table: each Gray word is a case item and the corresponding binary word is the value assigned. The model below is a combinational module with a 4-bit input G and a 4-bit output B. A default item is included so that the block assigns a value on every path and no latch is inferred.

```
module gray2bin (input  [3:0] G, output reg [3:0] B);

    always @ (G)

        case (G)

            4'b0000: B = 4'b0000;
```

```
            4'b0001: B = 4'b0001;
            4'b0011: B = 4'b0010;
            4'b0010: B = 4'b0011;
            4'b0110: B = 4'b0100;
            4'b0111: B = 4'b0101;
            4'b0101: B = 4'b0110;
            4'b0100: B = 4'b0111;
            4'b1100: B = 4'b1000;
            4'b1101: B = 4'b1001;
            4'b1111: B = 4'b1010;
            4'b1110: B = 4'b1011;
            4'b1010: B = 4'b1100;
            4'b1011: B = 4'b1101;
            4'b1001: B = 4'b1110;
            4'b1000: B = 4'b1111;
            default: B = 4'b0000;
        endcase
endmodule
```

To verify the model, a testbench applies the sixteen Gray codewords in counting order (generated as i ⊕(i shifted right by 1) for i = 0 … 15) and checks that the output equals the binary number i in every case. The model and testbench were compiled and simulated with the

Icarus Verilog simulator. The actual output of the simulation run is reproduced below; every line agrees with the truth table above, and the testbench reports that the test passed with no mismatches.

```
Gray = 0000  ->  Binary = 0000  (decimal 0)
Gray = 0001  ->  Binary = 0001  (decimal 1)
Gray = 0011  ->  Binary = 0010  (decimal 2)
Gray = 0010  ->  Binary = 0011  (decimal 3)
Gray = 0110  ->  Binary = 0100  (decimal 4)
Gray = 0111  ->  Binary = 0101  (decimal 5)
Gray = 0101  ->  Binary = 0110  (decimal 6)
Gray = 0100  ->  Binary = 0111  (decimal 7)
Gray = 1100  ->  Binary = 1000  (decimal 8)
Gray = 1101  ->  Binary = 1001  (decimal 9)
Gray = 1111  ->  Binary = 1010  (decimal 10)
Gray = 1110  ->  Binary = 1011  (decimal 11)
Gray = 1010  ->  Binary = 1100  (decimal 12)
Gray = 1011  ->  Binary = 1101  (decimal 13)
Gray = 1001  ->  Binary = 1110  (decimal 14)
Gray = 1000  ->  Binary = 1111  (decimal 15)
TEST PASSED: all 16 Gray code words converted correctly.
```

Conclusion

Tracing the printed circuit level by level and simplifying gives F = A'D + ABC and G = A'D' + ABC — a good example of how much simpler a circuit's behaviour can look after analysis than it does in the diagram. The Gray-to-binary converter of part (b) is built from three exclusive-OR gates in a cascade, $B_3 = G_3$, $B_2 = B_3 \oplus G_2$, $B_1 = B_2 \oplus G_1$, $B_0 = B_1 \oplus G_0$, and its Verilog case-statement model, compiled and simulated against a self-checking testbench, converts all sixteen Gray words correctly.

Question 5: Design of a BCD-to-Seven-Segment Decoder (20 Marks)

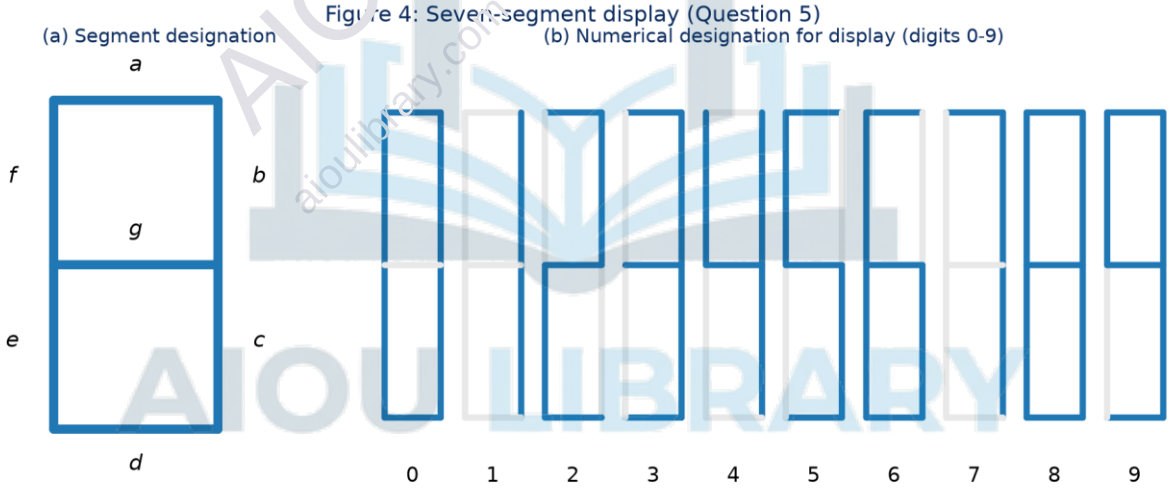
Question as printed: An ABCD-to-seven-segment decoder is a combinational circuit that converts a decimal digit in BCD to an appropriate code for the selection of segments in an indicator used to display the decimal digit in a familiar form. The seven outputs of the decoder (a, b, c, d, e, f, g) select the corresponding segments in the display, as shown in Fig. P4.9(a). The numeric display chosen to represent the decimal digit is shown in Fig. P4.9(b). Using a truth table and Karnaugh maps, design the BCD-to-seven-segment decoder using a minimum number of gates. The six invalid combinations should result in a blank display. The output F is 1 if the decimal equivalent of the binary input is an even number greater than 0. The output is 0 otherwise. (20)

Introduction

A seven-segment display shows a decimal digit by lighting a pattern of seven bar-shaped segments, labelled a to g as in Figure 4(a): a is the top bar, b the upper-right, c the lower-right, d the bottom, e the lower-left, f the upper-left and g the middle bar. The BCD-to-seven-

segment decoder is the combinational circuit that stands between a 4-bit BCD input ABCD (A most significant) and the display: for each decimal digit it must switch on exactly the segments that draw that digit, as shown in Figure 4(b). The design procedure is the standard one — write the truth table from the required digit patterns, transfer each output column to a Karnaugh map, and read off a minimum sum-of-products expression for each of the seven segment outputs, sharing product terms between outputs where possible to keep the total gate count low. The question adds two specific requirements that shape the design: the six unused BCD combinations (1010 to 1111) must produce a blank display, and there is an eighth output, F, which must detect the even non-zero inputs.

Figure 4: Seven-segment display (Question 5)



The Truth Table

For each valid digit, the segment column is filled in by inspecting the corresponding pattern in Figure 4(b): digit 0 lights every segment except g; digit 1 lights only b and c; digit 2 lights a, b, g, e and d; and so on. For the invalid combinations 1010–1111 every segment output is set to 0,

because the question requires a blank display for them. The output F follows its own printed definition: it is 1 exactly when the decimal equivalent of the 4-bit binary input is an even number greater than 0.

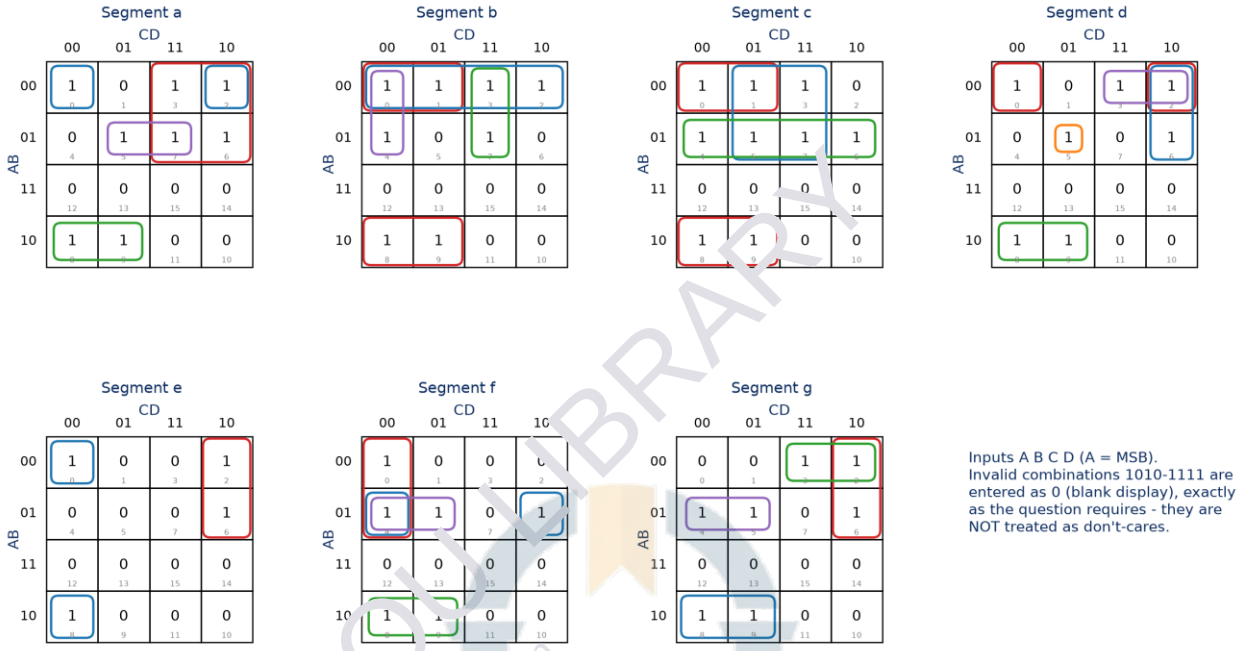
A B C D	Digit	a	b	c	d	e	f	g	F
0000	0	1	1	1	1	1	1	0	0
0001	1	0	1	1	0	0	0	0	0
0010	2	1	1	0	1	1	0	1	1
0011	3	1	1	1	1	0	0	1	0
0100	4	0	1	1	0	0	1	1	1
0101	5	1	0	1	1	0	1	1	0
0110	6	1	0	1	1	1	1	1	1
0111	7	1	1	1	0	0	0	0	0
1000	8	1	1	1	1	1	1	1	1
1001	9	1	1	1	1	0	1	1	0
1010	invalid	0	0	0	0	0	0	0	1
1011	invalid	0	0	0	0	0	0	0	0
1100	invalid	0	0	0	0	0	0	0	1
1101	invalid	0	0	0	0	0	0	0	0
1110	invalid	0	0	0	0	0	0	0	1

1111	invalid	0	0	0	0	0	0	0	0

Karnaugh Maps and the Minimum Expressions

One point of method must be stated plainly, because it decides the form of the answer. In many textbook treatments of this decoder, the six invalid combinations are entered in the K-maps as don't-care conditions, which allows larger groupings and shorter expressions. That shortcut is not available here: the question expressly requires the invalid combinations to result in a blank display, so their cells are entered as firm 0s, and every grouping must avoid them. The resulting expressions are therefore longer than the familiar don't-care versions — that is the honest price of guaranteeing a blank display — but they are the true minima under the stated requirement, and each one has been verified against the truth table over all sixteen input combinations. The seven maps are collected in Figure 5; reading the groups from them gives the following minimum sum-of-products expressions, in which A is the most significant input bit.

Figure 5: K-maps for the seven segment outputs (Question 5)



- a = A’C + A’B’D’ + AB’C’ + A’BD — the group A’C covers digits 2, 3, 6, 7; A’B’D’ covers 0 and 8; AB’C’ covers 8 and 9; A’BD covers 5 and 7.
- b = B’C’ + A’B’ + A’CD + A’C’D’ — b is off only for digits 5 and 6 among the valid digits, which is why four groups are needed once the invalid cells are barred.
- c = B’C’ + A’D + A’B — the shortest of the seven: c is off only for digit 2 among the valid digits.
- d = A’B’D’ + A’CD’ + AB’C’ + A’B’C + A’BC’D — d is the most irregular output (off for 1, 4 and 7), so it needs five groups, one of them a single-cell group at digit 5 (A’BC’D).
- e = A’CD’ + B’C’D’ — e is on only for digits 0, 2, 6 and 8, and two groups of two suffice: A’CD’ covers 2 and 6, B’C’D’ covers 0 and 8.
- f = A’C’D’ + A’BD’ + AB’C’ + A’BC’ — f is on for digits 0, 4, 5, 6, 8 and 9.

- g = A’CD’ + AB’C’ + A’B’C + A’BC’ — g is on for every digit from 2 to 9 except 7; the term A’CD’ is shared with output e and the term AB’C’ is shared with outputs a, d and f, which is where the gate saving of a multi-output design comes from.

Implemented with shared product terms, the decoder needs one AND gate for each distinct product term appearing in the seven expressions (terms such as A’C, B’C’D’ and AB’C’ are each built once and fanned out to every output that uses them), one OR gate per output to sum its terms, and inverters for A’, B’, C’ and D’ — four inverters in all, since every complemented literal comes from these. Counting distinct product terms across all seven expressions gives fifteen AND gates, plus seven OR gates and four inverters. Any attempt to shorten an individual expression by grouping through an invalid cell would light a segment for that invalid input and break the blank-display requirement, so this count is the minimum consistent with the question as printed.

The Additional Output F

The output F is defined over the whole 4-bit binary input, not merely over the ten valid BCD digits: F = 1 when the decimal equivalent of the input is an even number greater than 0, and 0 otherwise. The even 4-bit values greater than 0 are 2, 4, 6, 8, 10, 12 and 14, so F has 1s at minterms 2, 4, 6, 8, 10, 12 and 14. On the K-map these are exactly the cells with D = 0 except the cell 0000, which gives the compact result F = AD’ + BD’ + CD’ = D’(A + B + C): the factor D’ enforces “even” (an even binary number ends in 0), and the factor (A + B + C) enforces “greater than 0” by excluding the all-zero input. Restricted to the valid BCD digits, the same expression is 1 exactly for the digits 2, 4, 6 and 8, as one would expect; it is also 1 for the invalid inputs 10, 12

and 14, whose decimal equivalents are even and greater than 0, while the display itself remains blank for those inputs because all seven segment outputs are 0. One AND gate for the product D'(A + B + C) — or three small AND gates summed by an OR gate, reusing the existing D' inverter — implements F.

Conclusion

The decoder has been designed exactly to the printed specification: a sixteen-row truth table fixes the segment pattern of every digit and forces a blank display for the six invalid combinations; the K-maps of Figure 5 then give minimum expressions for the seven segment outputs, with product terms shared between outputs to minimise the gate count (fifteen AND gates, seven OR gates and four inverters in total); and the extra output is F = D'(A + B + C), which is 1 precisely for the even inputs greater than 0. Designs quoted in many books are shorter only because they treat the invalid combinations as don't-cares and may therefore light stray segments for them — a behaviour this question explicitly forbids.