ESD - Digital Systems Electronics

Solutions

Riccardo Berta

2025.12.11

1 Data Representation Exercises

1.1 Exercise 1

What is the largest 32-bit binary number that can be represented with:

1.1.1 (a) Unsigned numbers

Largest value: $1111 \ldots 1111$ (32 ones)
$= 2^{32} - 1 = 4.294.967.295$
$\approx 2^{30} \times 2^2 \approx 4GB$

1.1.2 (b) Two's complement numbers

Range: -2^{31} to $2^{31} - 1$
Largest value: $0111 \ldots 1111$ (31 ones after the leading 0)
$= 2^{31} - 1 = 2.147.483.647$
$\approx 2^{30} \times 2^1 \approx 2GB$

1.1.3 (c) Sign/magnitude numbers

1 bit for the sign, 31 bits for the magnitude
Largest positive: $0111 \ldots 1111$
$= 2^{31} - 1 = 2.147.483.647$
$\approx 2^{30} \times 2^1 \approx 2GB$

1.2 Exercise 2

What is the smallest (most negative) 16-bit binary number that can be represented with:

1.2.1 (a) Unsigned numbers

Unsigned representation cannot encode negative values.
Smallest value: $0000 \ldots 0000 = 0$

1.2.2 (b) Two's complement numbers

Range: -2^{15} to $+2^{15} - 1$
Smallest value: $1000 \ldots 0000$ (1 followed by 15 zeros)
$= -2^{15} = -32.768$

1.2.3 (c) Sign/magnitude numbers

1 bit for sign, 15 bits for magnitude
Smallest value: $1111 \ldots 1111$ (sign bit = 1, magnitude = max)
$= -(2^{15} - 1) = -32.767$

1.3 Exercise 3

What is the smallest (most negative) 32-bit binary number that can be represented with:

1.3.1 (a) Unsigned numbers

Unsigned representation cannot encode negative values.
Smallest value: $0000 \ldots 0000_2 = 0$

1.3.2 (b) Two's complement numbers

Range: -2^{31} to $+2^{31} - 1$
Smallest value: $1000 \ldots 0000_2$ (1 followed by 31 zeros)
$= -2^{31} = -2.147.483.648$

1.3.3 (c) Sign/magnitude numbers

1 bit for sign, 31 bits for magnitude
Smallest value: $1111 \ldots 1111_2$ (sign bit = 1, magnitude = max)
$= -(2^{31} - 1) = -2.147.483.647$

1.4 Exercise 4

Convert the following unsigned binary numbers to decimal and to hexadecimal:

1.4.1 (a) 1110_2

Decimal: $1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 8 + 4 + 2 = 14$
Hex: $14_{10} = E_{16}$

1.4.2 (b) 100100_2

Decimal: $1 \times 2^5 + 0 + 0 + 1 \times 2^2 = 32 + 4 = 36$
Hex: $36_{10} = 24_{16}$

1.4.3 (c) 11010111_2

Decimal: $128 + 64 + 16 + 4 + 2 + 1 = 215$
Hex: group as $(1101)(0111) = D7_{16}$

1.4.4 (d) 011101010100100_2

Decimal: $= 2^{13} + 2^{12} + 2^{11} + 2^9 + 2^7 + 2^5 + 2^2$
$= 8192 + 4096 + 2048 + 512 + 128 + 32 + 4 = 15012_{10}$ Hex: group as $(0011)(1010)(1010)(0100) = 3AA4_{16}$

1.4.5 (e) 0110_2

Decimal: $4 + 2 = 6$
Hex: $6_{10} = 6_{16}$

1.4.6 (f) 101101_2

Decimal: $32 + 8 + 4 + 1 = 45$
Hex: $45_{10} = 2D_{16}$

1.4.7 (g) 10010101_2

Decimal: $128 + 16 + 4 + 1 = 149$
Hex: group as $(1001)(0101) = 95_{16}$

1.4.8 (h) 110101001001_2

Decimal: $2048 + 1024 + 256 + 64 + 8 + 1 = 3401$
Hex: group as $(1101)(0100)(1001) = D49_{16}$

1.5 Exercise 5

Convert the following hexadecimal numbers to decimal and to unsigned binary:

1.5.1 (a) $4E_{16}$

$= 4 \times 16^1 + 14 \times 16^0 = 64 + 14 = 78_{10}$
Binary: $4 = 0100,\ E = 1110 \ \Rightarrow\ 0100\ 1110_2$

1.5.2 (b) $7C_{16}$

$= 7 \times 16^1 + 12 \times 16^0 = 112 + 12 = 124_{10}$
Binary: $7 = 0111,\ C = 1100 \ \Rightarrow\ 0111\ 1100_2$

1.5.3 (c) $ED3A_{16}$

$= 14 \times 16^3 + 13 \times 16^2 + 3 \times 16^1 + 10 \times 16^0$
$= 57.344 + 3.328 + 48 + 10 = 60.730_{10}$
Binary: $E = 1110,\ D = 1101,\ 3 = 0011,\ A = 1010 \ \Rightarrow\ 1110\ 1101\ 0011\ 1010_2$

1.5.4 (d) $403FB001_{16}$

$= 4 \times 16^7 + 0 \times 16^6 + 3 \times 16^5 + 15 \times 16^4 + 11 \times 16^3 + 0 \times 16^2 + 0 \times 16^1 + 1$
$= 1.073.741.824 + 3.145.728 + 61.440 + 45.056 + 1 = 1.077.915.649_{10}$
Binary: $4 = 0100,\ 0 = 0000,\ 3 = 0011,\ F = 1111,\ B = 1011,\ 0 = 0000,\ 0 = 0000,\ 1 = 0001$
$\Rightarrow 0100\ 0000\ 0011\ 1111\ 1011\ 0000\ 0000\ 0001_2$

1.5.5 (e) $2B_{16}$

$= 2 \times 16^1 + 11 \times 16^0 = 32 + 11 = 43_{10}$
Binary: $2 = 0010,\ B = 1011 \ \Rightarrow\ 0010\ 1011_2$

1.5.6 (f) $9F_{16}$

$= 9 \times 16^1 + 15 \times 16^0 = 144 + 15 = 159_{10}$
Binary: $9 = 1001,\ F = 1111 \ \Rightarrow\ 1001\ 1111_2$

1.5.7 (g) $42CE_{16}$

$= 4 \times 16^3 + 2 \times 16^2 + 12 \times 16^1 + 14 \times 16^0$
$= 16.384 + 512 + 192 + 14 = 17.102_{10}$
Binary: $4 = 0100,\ 2 = 0010,\ C = 1100,\ E = 1110 \ \Rightarrow\ 0100\ 0010\ 1100\ 1110_2$

1.5.8 (h) $E34F_{16}$

$= 14 \times 16^3 + 3 \times 16^2 + 4 \times 16^1 + 15 \times 16^0$
$= 57.344 + 768 + 64 + 15 = 58.191_{10}$
Binary: $E = 1110,\ 3 = 0011,\ 4 = 0100,\ F = 1111 \ \Rightarrow\ 1110\ 0011\ 0100\ 1111_2$

1.6 Exercise 6

Convert the following two's complement binary numbers to decimal:

1.6.1 (a) 1110_2 (4-bit)

MSB = 1 -> negative.
Invert 1110 → 0001, add 1 -> 0010 = 2.
Result = -2_{10}.

1.6.2 (b) 100011_2 (6-bit)

MSB = 1 -> negative.
Invert 100011 → 011100, add 1 -> 011101 = 29.
Result = -29_{10}.

1.6.3 (c) 01001110_2 (8-bit)

MSB = 0 -> positive.
Value = $64 + 8 + 4 + 2 = 78_{10}$.

1.6.4 (d) 10110101_2 (8-bit)

MSB = 1 -> negative.
Invert 10110101 → 01001010, add 1 -> 01001011 = 75.
Result = -75_{10}.

1.6.5 (e) 1001_2 (4-bit)

MSB = 1 -> negative.
Invert 1001 → 0110, add 1 -> 0111 = 7.
Result = -7_{10}.

1.6.6 (f) 110101_2 (6-bit)

MSB = 1 -> negative.
Invert 110101 → 001010, add 1 -> 001011 = 11.
Result = -11_{10}.

1.6.7 (g) 01100010_2 (8-bit)

MSB = 0 -> positive.
Value = $64 + 32 + 2 = 98_{10}$.

1.6.8 (h) 10111000_2 (8-bit)

MSB = 1 -> negative.
Invert 10111000 → 01000111, add 1 -> 01001000 = 72.
Result = -72_{10}.

1.7 Exercise 7

Convert the following decimal numbers to unsigned binary and to hexadecimal

1.7.1 (a) 42_{10}

$42 \div 2 = 21$ remainder 0
$21 \div 2 = 10$ remainder 1

$10 \div 2 = 5$ remainder 0
$5 \div 2 = 2$ remainder 1
$2 \div 2 = 1$ remainder 0
$1 \div 2 = 0$ remainder 1
Reading upwards -> 101010_2
Group: 0010 $1010_2 = 2A_{16}$

1.7.2 (b) 63_{10}

$63 \div 2 = 31$ r 1
$31 \div 2 = 15$ r 1
$15 \div 2 = 7$ r 1
$7 \div 2 = 3$ r 1
$3 \div 2 = 1$ r 1
$1 \div 2 = 0$ r 1
-> 111111_2
Group: 0011 $1111_2 = 3F_{16}$

1.7.3 (c) 229_{10}

$229 \div 2 = 114$ r 1
$114 \div 2 = 57$ r 0
$57 \div 2 = 28$ r 1
$28 \div 2 = 14$ r 0
$14 \div 2 = 7$ r 0
$7 \div 2 = 3$ r 1
$3 \div 2 = 1$ r 1
$1 \div 2 = 0$ r 1
-> 11100101_2
Group: 1110 $0101_2 = E5_{16}$

1.7.4 (d) 845_{10}

$845 \div 2 = 422$ r 1
$422 \div 2 = 211$ r 0
$211 \div 2 = 105$ r 1
$105 \div 2 = 52$ r 1

$52 \div 2 = 26$ r 0
$26 \div 2 = 13$ r 0
$13 \div 2 = 6$ r 1
$6 \div 2 = 3$ r 0
$3 \div 2 = 1$ r 1
$1 \div 2 = 0$ r 1
-> 1101001101_2
Group: $0011\ 0100\ 1101_2 = 34D_{16}$

1.7.5 (e) 56_{10}

$56 \div 2 = 28$ r 0
$28 \div 2 = 14$ r 0
$14 \div 2 = 7$ r 0
$7 \div 2 = 3$ r 1
$3 \div 2 = 1$ r 1
$1 \div 2 = 0$ r 1
$\rightarrow 111000_2$
Group: $0011\ 1000_2 = 38_{16}$

1.7.6 (f) 75_{10}

$75 \div 2 = 37$ r 1
$37 \div 2 = 18$ r 1
$18 \div 2 = 9$ r 0
$9 \div 2 = 4$ r 1
$4 \div 2 = 2$ r 0
$2 \div 2 = 1$ r 0
$1 \div 2 = 0$ r 1
-> 1001011_2
Group: $0100\ 1011_2 = 4B_{16}$

1.7.7 (g) 183_{10}

$183 \div 2 = 91$ r 1
$91 \div 2 = 45$ r 1
$45 \div 2 = 22$ r 1

$22 \div 2 = 11$ r 0
$11 \div 2 = 5$ r 1
$5 \div 2 = 2$ r 1
$2 \div 2 = 1$ r 0
$1 \div 2 = 0$ r 1
-> 10110111_2
Group: $1011\ 0111_2 = B7_{16}$

1.7.8 (h) 754_{10}

$754 \div 2 = 377$ r 0
$377 \div 2 = 188$ r 1
$188 \div 2 = 94$ r 0
$94 \div 2 = 47$ r 0
$47 \div 2 = 23$ r 1
$23 \div 2 = 11$ r 1
$11 \div 2 = 5$ r 1
$5 \div 2 = 2$ r 1
$2 \div 2 = 1$ r 0
$1 \div 2 = 0$ r 1
-> 1011110010_2
Group: $0010\ 1111\ 0010_2 = 2F2_{16}$

1.8 Exercise 8

Convert the following decimal numbers to 8-bit two's complement numbers or indicate overflow.

Range of 8-bit two's complement: $-128 \leq N \leq +127$.

1.8.1 (a) 24

128:0,
64:0,
32:0,
16:1 (remainder 8),
8:1 (0),
4:0,

2:0,
1:0
00011000

1.8.2 (b) -59

128:1 (69),
64:1 (5),
32:0,
16:0,
8:0,
4:1 (1),
2:0,
1:1 (0)
11000101

1.8.3 (c) 128

Outside interval $[-128, 127]$
overflow

1.8.4 (d) -150

$-150 < -128$
overflow

1.8.5 (e) 127

128:0,
64:1 (63),
32:1 (31),
16:1 (15),
8:1 (7),
4:1 (3),
2:1 (1),
1:1 (0)
01111111

1.8.6 (f) 48

128:0,
64:0,
32:1 (16),
16:1 (0),
8:0, 4:0,
2:0,
1:0
00110000

1.8.7 (g) -34

$256 - 34 = 222$
128:1 (94),
64:1 (30),
32:0,
16:1 (14),
8:1 (6),
4:1 (2),
2:1 (0),
1:0
11011110

1.8.8 (h) 133

$133 > 127$
overflow

1.8.9 (i) -129

$-129 < -128$
overflow

1.9 Exercise 9

How many bytes are in a 32-bit word? How many nibbles are in the 32-bit word? How many bytes are in a 64-bit word? How many nibbles are in the 64-bit word? How many bits are in 2

bytes? How many bits are in 6 bytes?

A **word** of 32 bits = 32/8 = 4 bytes.
Each byte = 2 nibbles -> 4 × 2 = 8 nibbles in a 32-bit word.

A **word** of 64 bits = 64/8 = 8 bytes.
Each byte = 2 nibbles -> 8 × 2 = 16 nibbles in a 64-bit word.

In **2 bytes**: 2 × 8 = 16 bits.
In **6 bytes**: 6 × 8 = 48 bits.

1.10 Exercise 10

Convert the following decimal numbers to IEEE 754 single-precision format:

1.10.1 (a) 45.375_{10}

Sign: positive -> $sign = 0$
Integer 45 -> 101101_2
Fraction .375 = 3/8 -> $.011_2$
Combined -> 101101.011_2
Normalize: $1.01101011_2 \times 2^5$ -> unbiased exponent $E = 5$
Exponent (bias 127): $E + 127 = 132 = 10000010_2$
Mantissa (drop leading 1): 01101011 then pad -> 01101011000000000000000
-> 0 10000010 01101011000000000000000

1.10.2 (b) -13.25_{10}

Sign: negative -> $sign = 1$
Integer 13 -> 1101_2
Fraction .25 = 1/4 -> $.01_2$
Combined -> 1101.01_2
Normalize: $1.10101_2 \times 2^3$ -> unbiased exponent $E = 3$
Exponent (bias 127): $E + 127 = 130 = 10000010_2$
Mantissa: drop leading 1 -> 10101 then pad -> 10101000000000000000000
-> 1 10000010 10101000000000000000000

1.10.3 (c) 0.1_{10}

Sign: positive -> $sign = 0$
Fraction (repeating): $0.0001100110011..._2$
Normalize -> $1.1001100110011..._2 \times 2^{-4}$ -> $E = -4$
Exponent (bias 127): $E + 127 = 123 = 01111011_2$
Mantissa: take 23 bits after the leading 1, with rounding: 10011001100110011001101
-> 0 01111011 10011001100110011001101
(*Note: 0.1 is not exactly representable in binary; this is the rounded IEEE single-precision value.*)

1.10.4 (d) -0.125_{10}

Sign: negative -> $sign = 1$
Binary: $0.125 = 1/8 = 2^{-3}$ Normalize: $1.0_2 \times 2^{-3}$ -> $E = -3$
Exponent (bias 127): $E + 127 = 124 = 01111100_2$
Mantissa: exactly zero (since significand is 1.000...)
-> 1 01111100 00000000000000000000000

1.11 Exercise 11

Convert the following IEEE 754 single-precision numbers into decimal values:

1.11.1 (a) 0 10000010 01100000000000000000000

Sign = 0 6. -> positive
Exponent = $10000010_2 = 130$ 6. -> unbiased $E = 130 - 127 = 3$
Mantissa = $1.011_2 = 1 + 0.25 + 0.125 = 1.375$
Value = $1.375 \times 2^3 = 11_{10}$

1.11.2 (b) 1 10000001 01000000000000000000000

Sign = 1 -> negative
Exponent = $10000001_2 = 129$ -> unbiased $E = 129 - 127 = 2$
Mantissa = $1.01_2 = 1 + 0.25 = 1.25$
Value = $-(1.25 \times 2^2) = -5_{10}$

1.11.3 (c) 0 01111101 10000000000000000000000

Sign = 0 -> positive
Exponent = $01111101_2 = 125$ -> unbiased $E = 125 - 127 = -2$
Mantissa = $1.1_2 = 1.5$
Value = $1.5 \times 2^{-2} = 1.5 \times 0.25 = 0.375_{10}$

1.11.4 (d) 1 01111100 00000000000000000000000

Sign = 1 -> negative
Exponent = $01111100_2 = 124$ -> unbiased $E = 124 - 127 = -3$
Mantissa = $1.0_2 = 1.0$
Value = $-(1.0 \times 2^{-3}) = -0.125_{10}$

1.12 Exercise 12

A particular modem operates at 768 Kb/sec. How many bytes can it receive in 1 minute?

Case 1: K = 1000 (decimal kilo, common in communications)

- Data rate: $768 \times 1000 = 768.000$ bits/sec
- Time: 60 seconds
  $768.000 \times 60 = 46.080.000$ bits
- Convert to bytes: $\frac{46.080.000}{8} = 5.760.000$ bytes (≈ 5.76 MB)

Case 2: K = 1024 (binary kilo, kibi)

- Data rate: $768 \times 1024 = 786.432$ bits/sec
- Time: 60 seconds
  $786.432 \times 60 = 47.185.920$ bits
- Convert to bytes: $\frac{47.185.920}{8} = 5.898.240$ bytes ≈ 5.63 MiB

1.13 Exercise 13

USB 3.0 can send data at 5 Gb/sec. How many bytes can it send in 1 minute?

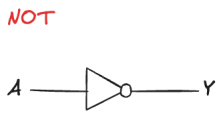
- Step 1 – Convert Gb/sec to bits/sec
  $5\,\text{Gb/sec} = 5 \times 10^9 = 5.000.000.000\ \text{bits/sec}$
- Step 2 – Multiply by time (60 seconds)
  $5.000.000.000 \times 60 = 300.000.000.000\ \text{bits}$
- Step 3 – Convert bits to bytes (8 bits = 1 byte) $\frac{300.000.000.000}{8} = 37.500.000.000\ \text{bytes}$
- In 1 minute, USB 3.0 can send **37.500.000.000 bytes**
  $\approx$ **37.5 GB (decimal)**

2 Logic gates and circuits Exercises

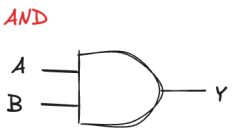
2.1 Exercise 1

Complete the truth tables of the following gates and use the DEEDS simulator to verify their correctness:

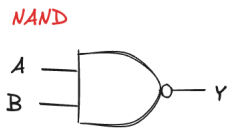
2.1.1 (a) NOT, AND, NAND



A	Y
0	1
1	0

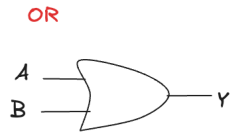


A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

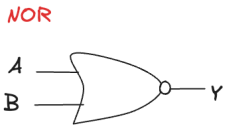


A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

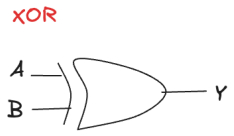
2.1.2 (b) OR, NOR, XOR



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

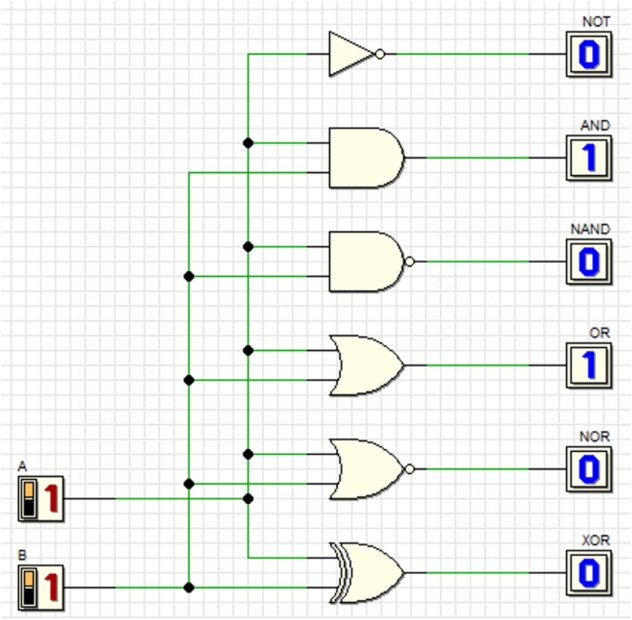


A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

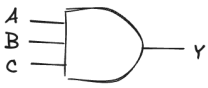
2.1.3 (c) DEEDS simulation



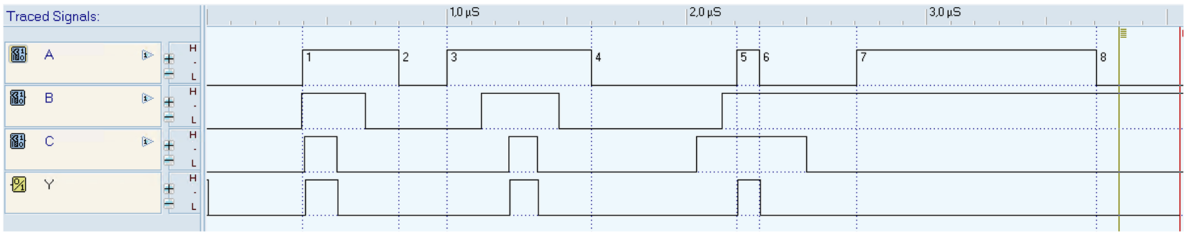
2.2 Exercise 2

Complete the truth table and the timing diagram of the following 3-inputs digital circuits, and use the DEEDS simulator to verify their correctness:

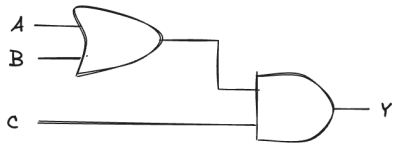
2.2.1 (a)



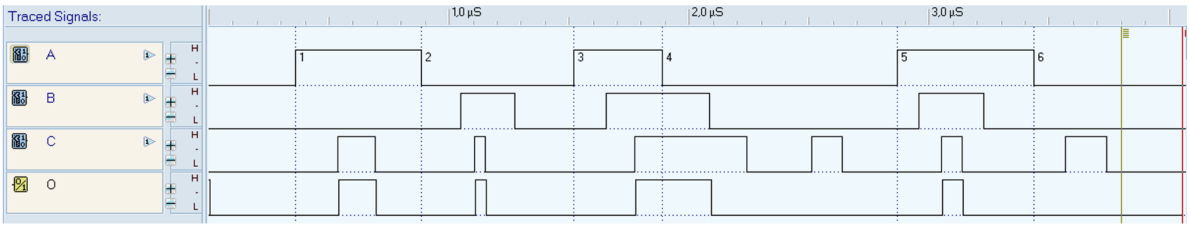
A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1



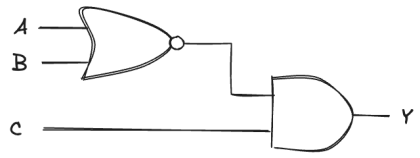
2.2.2 (b)



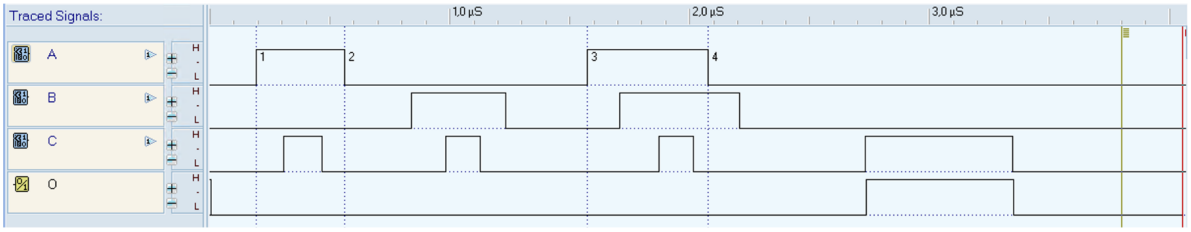
A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



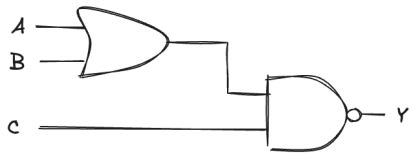
2.2.3 (c)



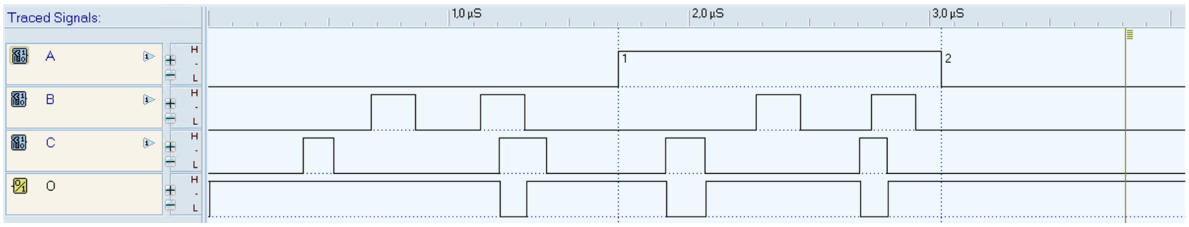
A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0



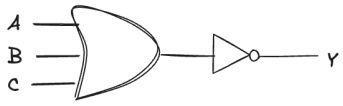
2.2.4 (d)



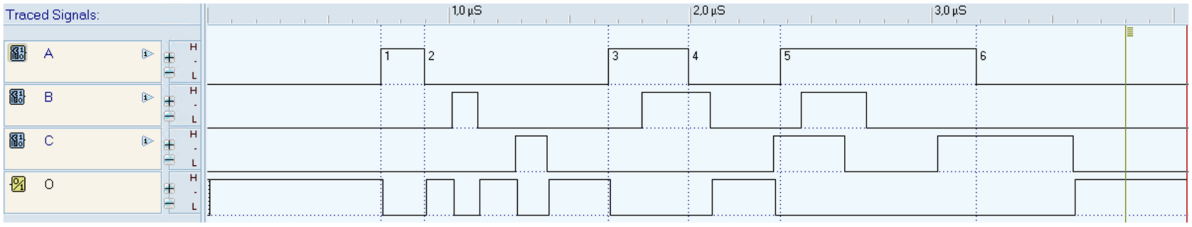
A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0



2.2.5 (e)



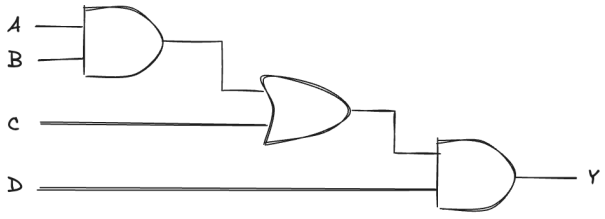
A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0



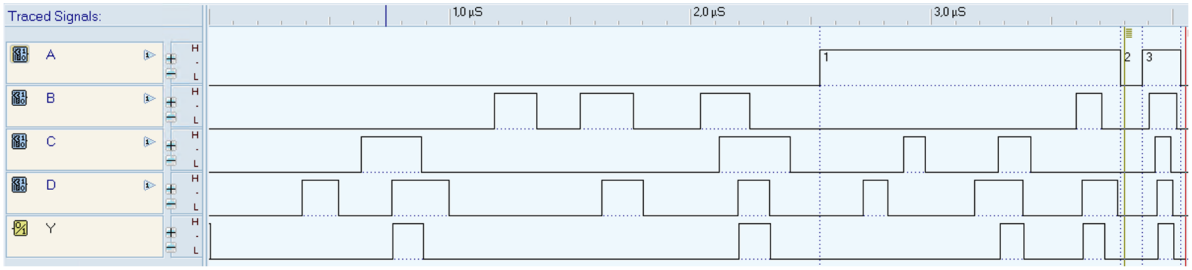
2.3 Exercise 3

Complete the truth table and the timing diagram of the following 4-inputs digital circuits, and use the DEEDS simulator to verify their correctness:

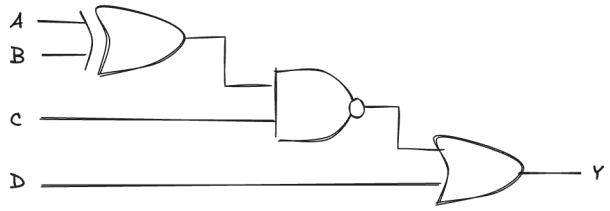
2.3.1 (a)



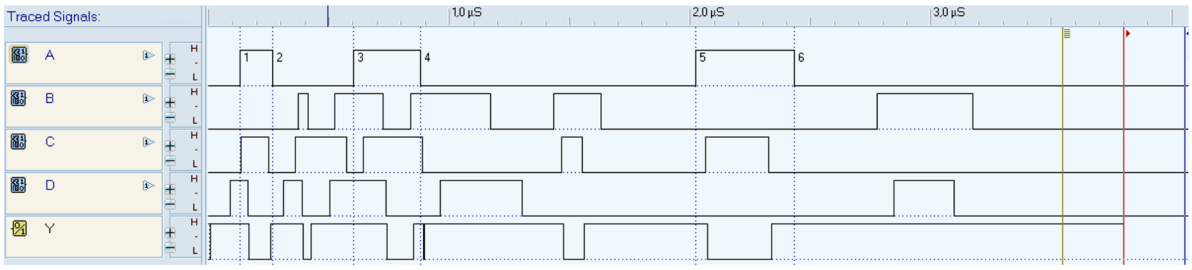
A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1



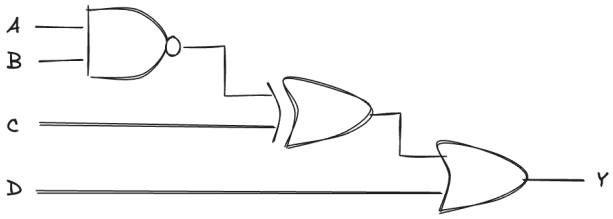
2.3.2 (b)



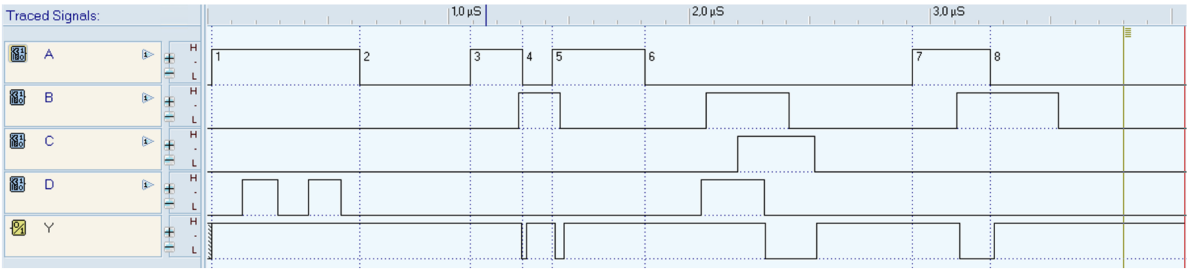
A	B	C	D	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1



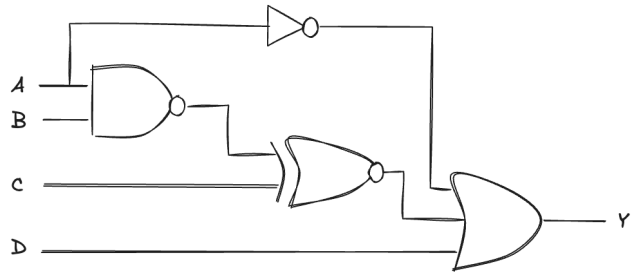
2.3.3 (c)



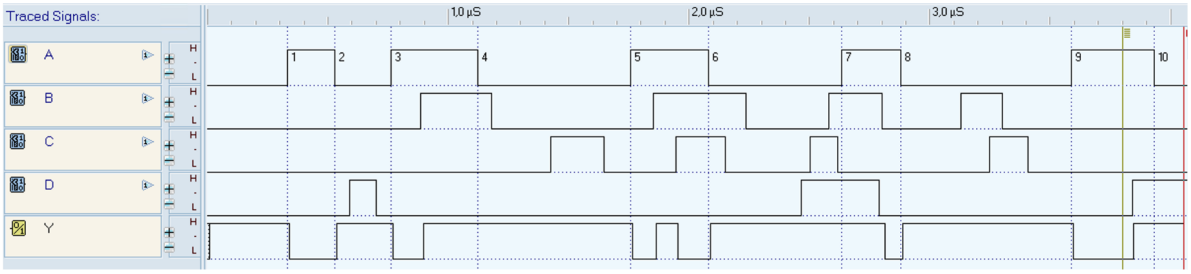
A	B	C	D	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1



2.3.4 (d)



A	B	C	D	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1



3 Boolean Algebra Exercises

3.1 Exercise 1

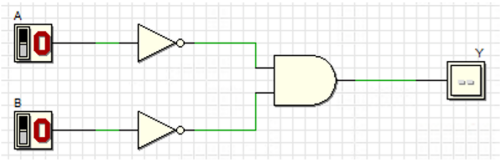
For each given truth table, write the Boolean equation in canonical **sum-of-products** form. Then simplify the expression to its minimal form using the theorems of Boolean algebra. Finally, implement the corresponding combinational circuit and verify its correctness by simulating it in DEEDS

3.1.1 (a)

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

$\overline{A}\,\overline{B}$

The expression cannot be simplified any further using Boolean algebra theorems.



A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

3.1.2 (b)

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

$Y = \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}C + AB\overline{C}$

Apply the Distributive Law
$= \overline{A}B(\overline{C} + C) + A\overline{B}C + AB\overline{C}$

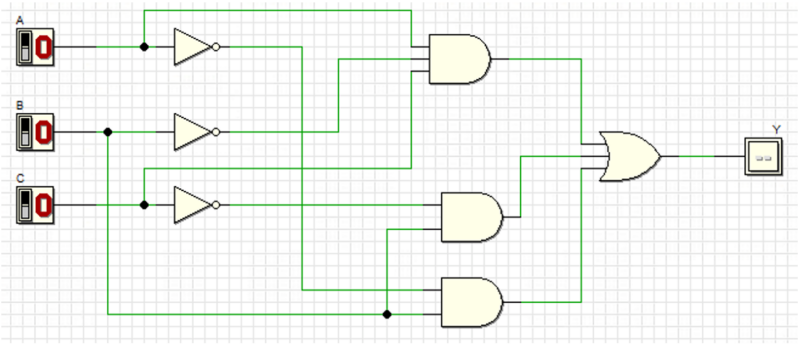
Apply the Complement Law:
$= \overline{A}B1 + A\overline{B}C + AB\overline{C}$

Apply the Identity Law:
$= \overline{A}B + A\overline{B}C + AB\overline{C}$

Apply the Distributive Law
$= A\overline{B}C + B(A\overline{C} + \overline{A})$

Apply the Absorption Law
$= A\overline{B}C + B(\overline{C} + \overline{A})$

Apply Distribution
$= A\overline{B}C + B\overline{C} + B\overline{A}$



C	B	A	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

3.1.3 (c)

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$Y = A\overline{B}C + AB\overline{C} + ABC$

Apply the Distributive Law:
$= AC(\overline{B} + B) + AB\overline{C}$

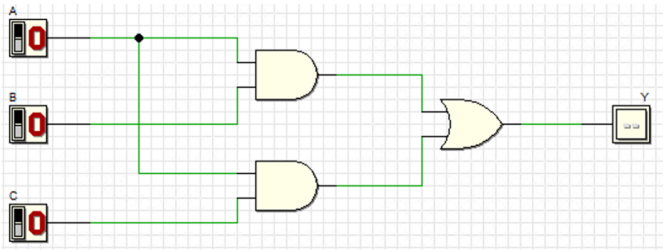
Apply the Complement Law:
$= AC1 + AB\overline{C}$

Apply the Identity Law:
$= AC + AB\overline{C}$

Apply the Distributive Law:
$= A(B\overline{C} + C)$

Apply the Absorption Law:
$= A(B + C)$

Apply Distribution
$= AB + AC$



A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

3.1.4 (d)

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

$Y = \overline{A}\,\overline{B}\,\overline{C}D + \overline{A}B\overline{C}D + AB\overline{C}\,\overline{D} + ABCD$

Apply the Distributive Law:
$= \overline{A}\,\overline{C}D(\overline{B} + B) + AB\overline{C}\,\overline{D} + ABCD$

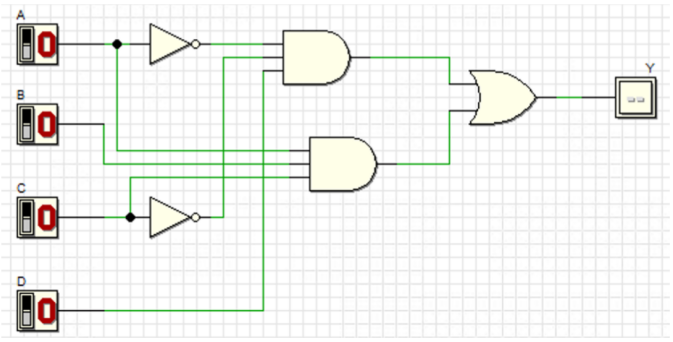
Apply the Complement Law:
$= \overline{A}\,\overline{C}D1 + AB\overline{C}\,\overline{D} + ABCD$

Apply the Identity Law:
$= \overline{A}\,\overline{C}D + AB\overline{C}\,\overline{D} + ABCD$

Apply the Distributive Law:
$= \overline{A}\,\overline{C}D + ABC(\overline{D} + D)$

Apply the Complement Law:
$= \overline{A}\,\overline{C}D + ABC1$

Apply the Identity Law:
$= \overline{A}\,\overline{C}D + ABC$



A	B	C	D	Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

3.1.5 (e)

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

$$Y = \overline{A}\,\overline{B}CD + \overline{A}BCD + A\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C}D + A\overline{B}C\overline{D}$$

Apply the Distributive Law:
$$= \overline{A}\,CD(\overline{B} + B) + A\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C}D + A\overline{B}C\overline{D}$$

Apply the Complement Law:
$$= \overline{A}\,CD1 + A\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C}D + A\overline{B}C\overline{D}$$

Apply the Identity Law:
$$= \overline{A}\,CD + A\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C}D + A\overline{B}C\overline{D}$$

Apply the Distributive Law:
$$= \overline{A}\,CD + A\overline{B}\,\overline{C}(\overline{D} + D) + A\overline{B}C\overline{D}$$

Apply the Complement Law:
$= \overline{A}\,CD + A\overline{B}\,\overline{C}1 + A\overline{B}C\overline{D}$

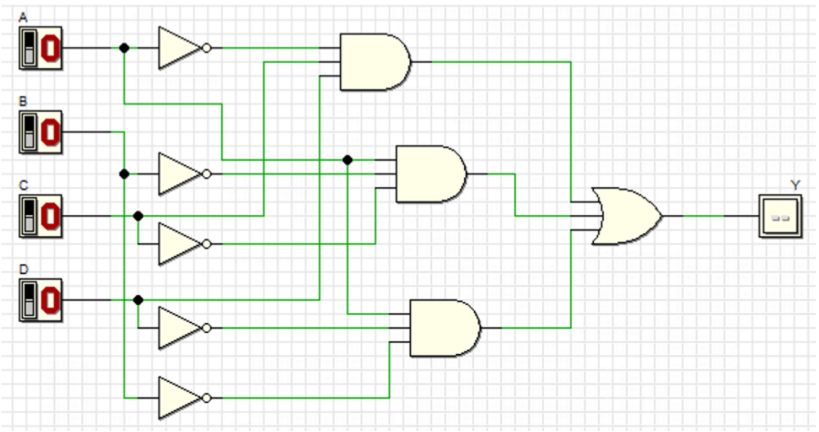
Apply the Identity Law:
$= \overline{A}\,CD + A\overline{B}\,\overline{C} + A\overline{B}C\overline{D}$

Apply the Distributive Law:
$= \overline{A}\,CD + A\overline{B}\,\overline{C} + A\overline{B}C\overline{D}$

Apply: Distribution Law:
$= \overline{A}\,CD + A\overline{B}(\overline{C} + C\overline{D})$

Apply the Absorption Law:
$= \overline{A}\,CD + A\overline{B}(\overline{C} + \overline{D})$

Apply: Distribution Law:
$= \overline{A}\,CD + A\overline{B}\,\overline{C} + A\overline{B}\,\overline{D}$

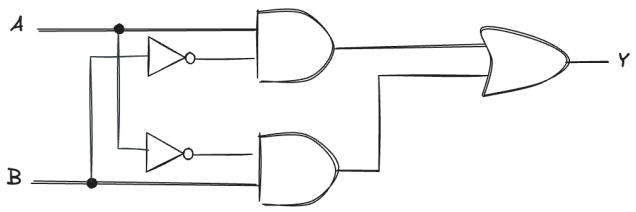


A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

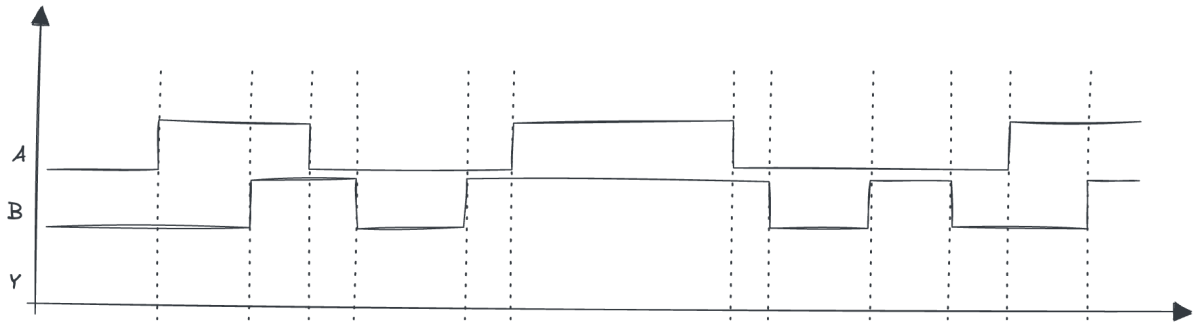
3.2 Exercise 2

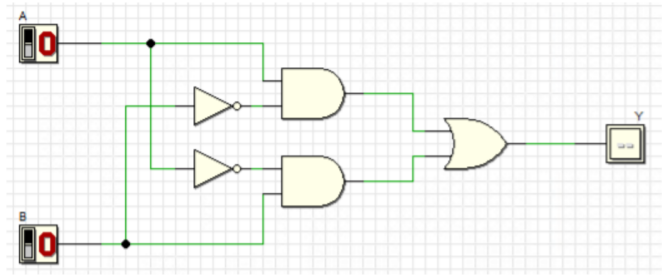
Complete the truth table and the timing diagram of the following digital circuits. Then derive the Boolean equation in canonical **sum-of-products** form and simplify it to its minimal expression using Boolean algebra theorems. Finally, implement the corresponding circuit in DEEDS and verify the correctness of your solution.

3.2.1 (a)



A	B	Y
0	0	
0	1	
1	0	
1	1	

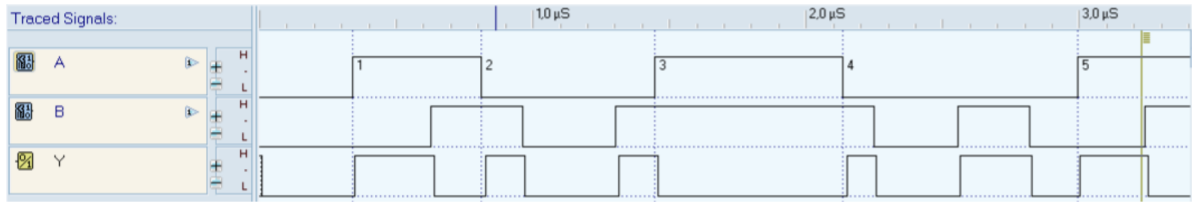




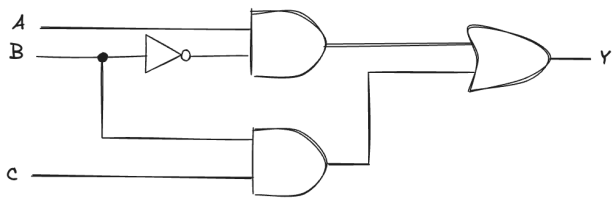
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

$Y = (A \cdot \overline{B}) + (\overline{A} \cdot B)$

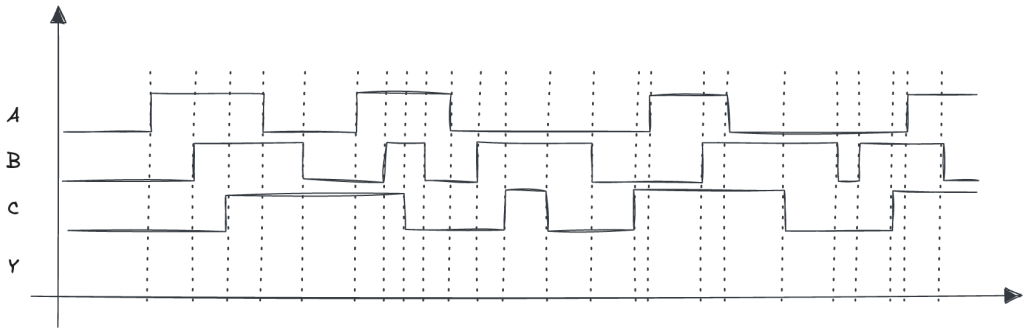
The expression cannot be simplified any further; however, by inspecting the truth table, we can see that it corresponds to an XOR

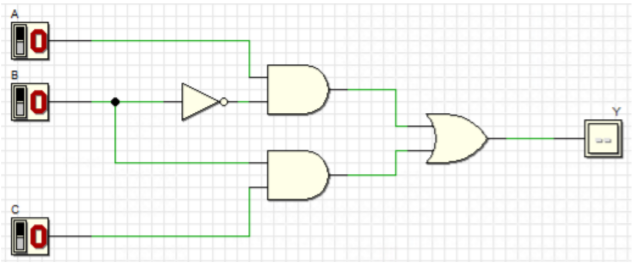


3.2.2 (b)



A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	





A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

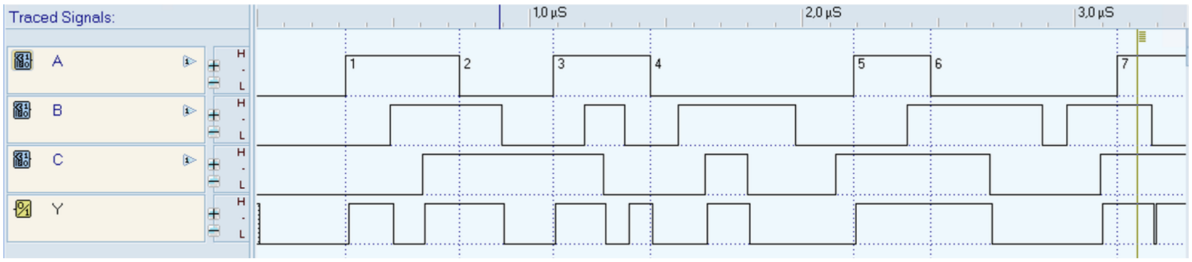
$$Y = \overline{A}BC + A\overline{B}\,\overline{C} + A\overline{B}C + ABC$$

$$= \overline{A}BC + ABC + A\overline{B}\,\overline{C} + A\overline{B}C$$

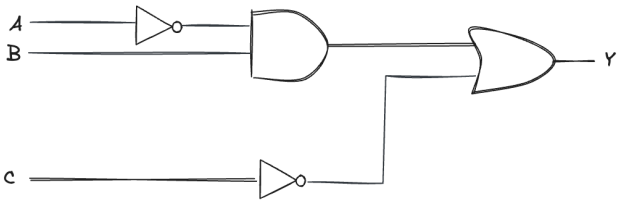
$$= BC(\overline{A} + A) + A\overline{B}(\overline{C} + C)$$

$$= BC \cdot 1 + A\overline{B} \cdot 1$$

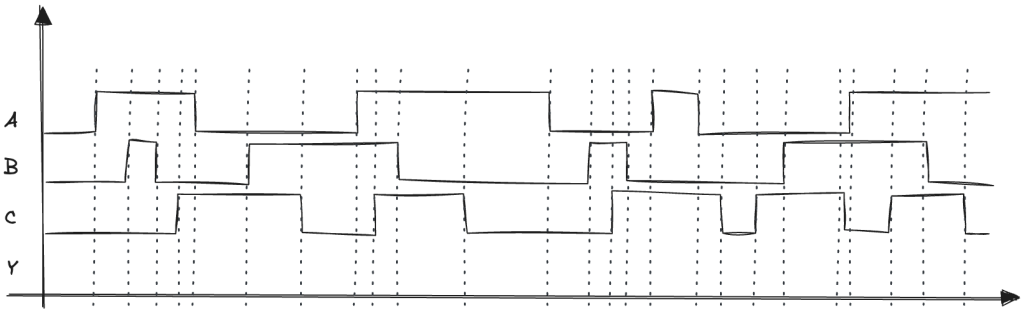
$$= BC + A\overline{B}$$

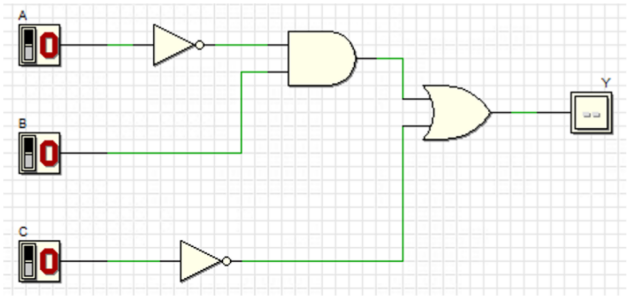


3.2.3 (c)



A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	





A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

$$Y = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}\,\overline{C} + AB\overline{C}$$

$$= \overline{A}\,\overline{C}(\overline{B} + B) + \overline{A}BC + A\overline{B}\,\overline{C} + AB\overline{C}$$

$$= \overline{A}\,\overline{C}1 + \overline{A}BC + A\overline{B}\,\overline{C} + AB\overline{C}$$

$$= \overline{A}\,\overline{C} + \overline{A}BC + A\overline{B}\,\overline{C} + AB\overline{C}$$

$$= \overline{A}(BC + \overline{C}) + A\overline{B}\,\overline{C} + AB\overline{C}$$

$$= \overline{A}(B + \overline{C}) + A\overline{B}\,\overline{C} + AB\overline{C}$$

$$= \overline{A}(B + \overline{C}) + A\overline{C}(B + \overline{B})$$

$$= \overline{A}(B + \overline{C}) + A\overline{C}1$$

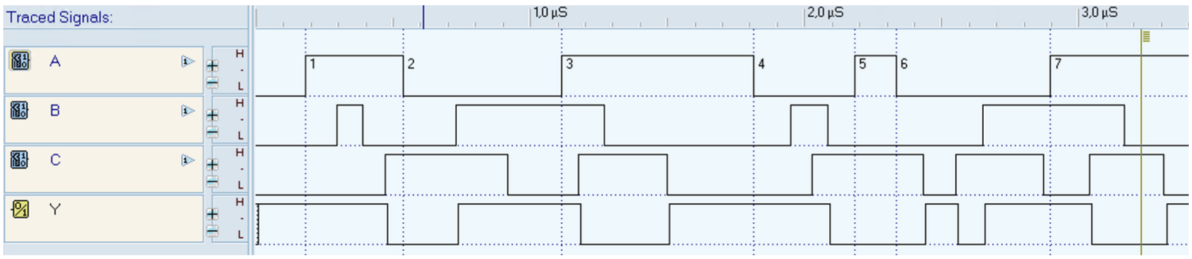
$$= \overline{A}(B + \overline{C}) + A\overline{C}$$

$$= \overline{A}B + \overline{A}\,\overline{C} + A\overline{C}$$

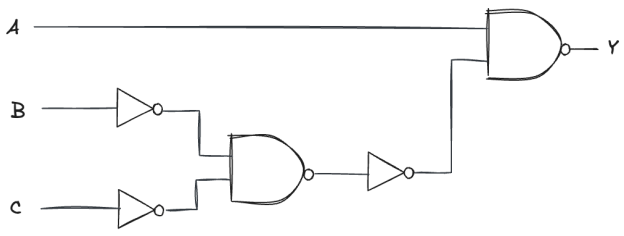
$$= \overline{A}B + \overline{C}(\overline{A} + A)$$

$$= \overline{A}B + \overline{C}1$$

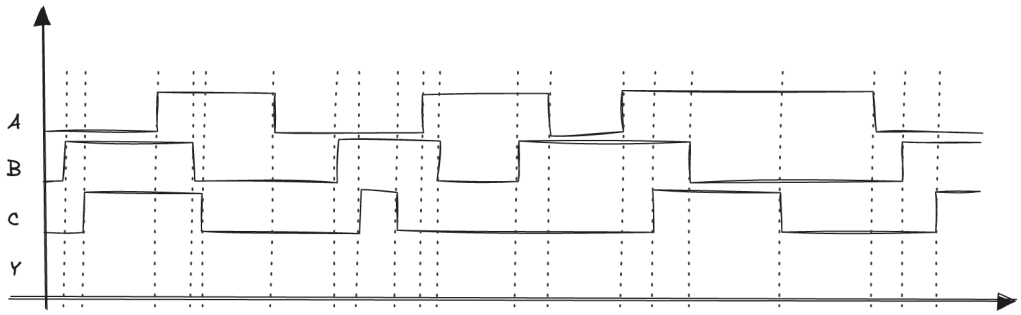
$$= \overline{A}B + \overline{C}$$

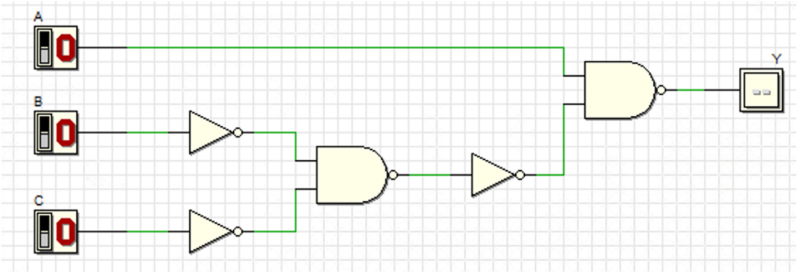


3.2.4 (d)



A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	





A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$$Y = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

$$= \overline{A}\,\overline{B}(C + \overline{C}) + \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

$$= \overline{A}\,\overline{B} + \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

$$= \overline{A}(\overline{B} + B\overline{C}) + \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

$$= \overline{A}(\overline{B} + \overline{C}) + \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

$$= \overline{A}(\overline{B} + \overline{C}) + BC(A + \overline{A}) + A\overline{B}C + AB\overline{C}$$

$$= \overline{A}(\overline{B} + \overline{C}) + BC + A\overline{B}C + AB\overline{C}$$

$$= \overline{A}(\overline{B} + \overline{C}) + C(A\overline{B} + B) + AB\overline{C}$$

$$= \overline{A}(\overline{B} + \overline{C}) + C(A + B) + AB\overline{C}$$

$$= \overline{A}\,\overline{B} + \overline{A}\,\overline{C} + C(A + B) + AB\overline{C}$$

$$= \overline{A}\,\overline{B} + C(A + B) + \overline{C}(AB + \overline{A})$$

$$= \overline{A}\,\overline{B} + C(A + B) + \overline{C}(B + \overline{A})$$

$$= \overline{A}\,\overline{B} + CA + CB + \overline{C}B + \overline{C}\,\overline{A}$$

$$= \overline{A}\,\overline{B} + CA + B(C + \overline{C}) + \overline{C}\,\overline{A}$$

$$= \overline{A}\,\overline{B} + CA + B + \overline{C}\,\overline{A}$$

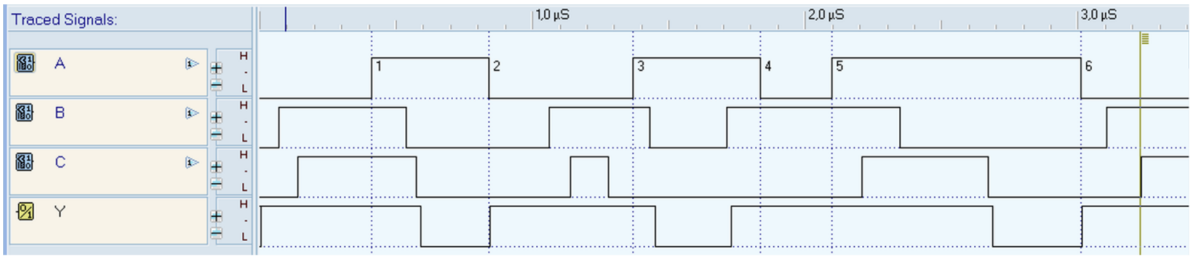
$$= \overline{A} + CA + B + \overline{C}\,\overline{A}$$

$$= \overline{A}(1 + \overline{C}) + CA + B$$

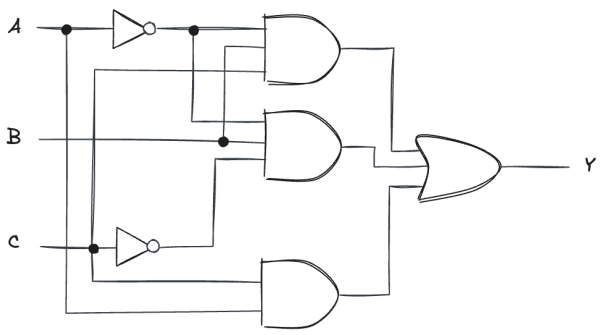
$= \overline{A} + CA + B$

$= \overline{A} + C + B$

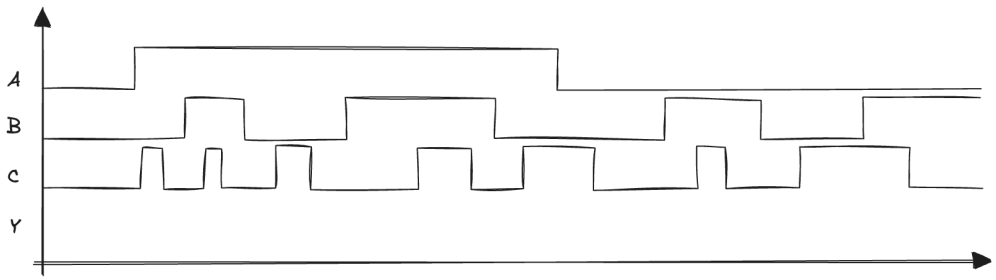
This is very tricky to simplify, however, if we start from the POS form (since we have just one 0 in the table and a lot of 1s), we can get the solution immediately.

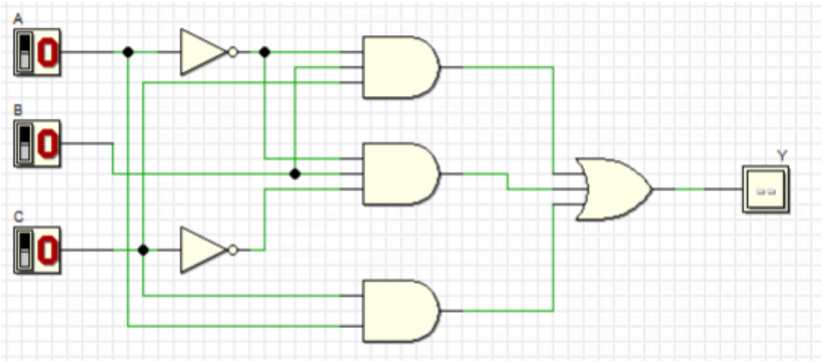


3.2.5 (e)



A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	





A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Check with the simulator what happens around 2.5 µs and 3.2 µs, and try to provide an explanation for the phenomenon.

$Y = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$

$= \overline{A}B(\overline{C} + C) + A\overline{B}C + ABC$

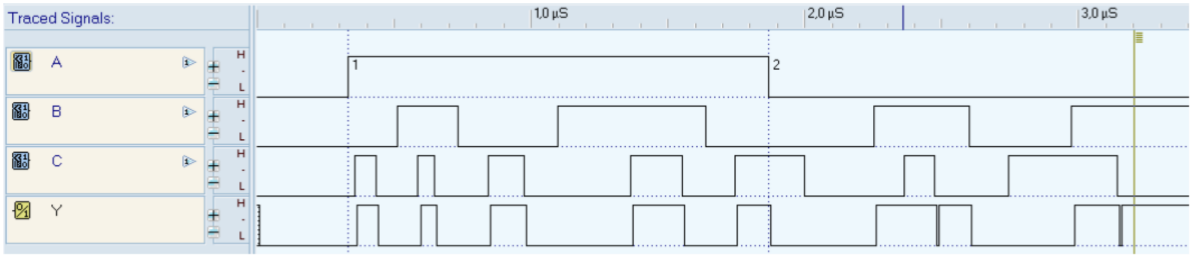
$= \overline{A}B(1) + A\overline{B}C + ABC$

$= \overline{A}B + A\overline{B}C + ABC$

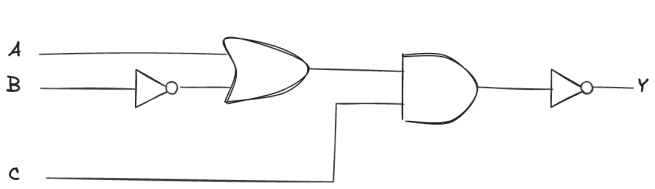
$= \overline{A}B + AC(\overline{B} + B)$

$= \overline{A}B + AC(1)$

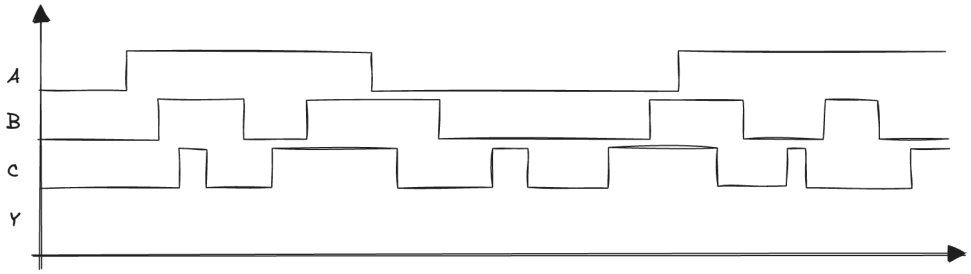
$= \overline{A}B + AC$

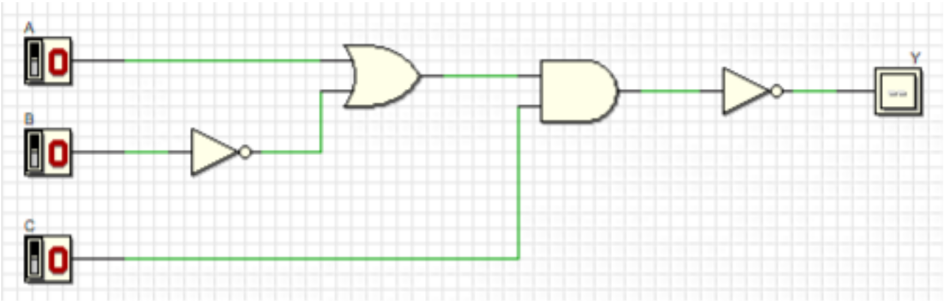


3.2.6 (f)



A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	





A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

$$Y = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}\,\overline{C} + AB\overline{C}$$

$$= \overline{A}\,\overline{C}(\overline{B} + B) + \overline{A}BC + A\overline{B}\,\overline{C} + AB\overline{C}$$

$$= \overline{A}\,\overline{C}1 + \overline{A}BC + A\overline{B}\,\overline{C} + AB\overline{C}$$

$$= \overline{A}\,\overline{C} + \overline{A}BC + A\overline{B}\,\overline{C} + AB\overline{C}$$

$$= \overline{A}(\overline{C} + BC) + A\overline{B}\,\overline{C} + AB\overline{C}$$

$$= \overline{A}(\overline{C} + B) + A\overline{B}\,\overline{C} + AB\overline{C}$$

$$= \overline{A}(\overline{C} + B) + A\overline{C}(\overline{B} + B)$$

$$= \overline{A}(\overline{C} + B) + A\overline{C}1$$

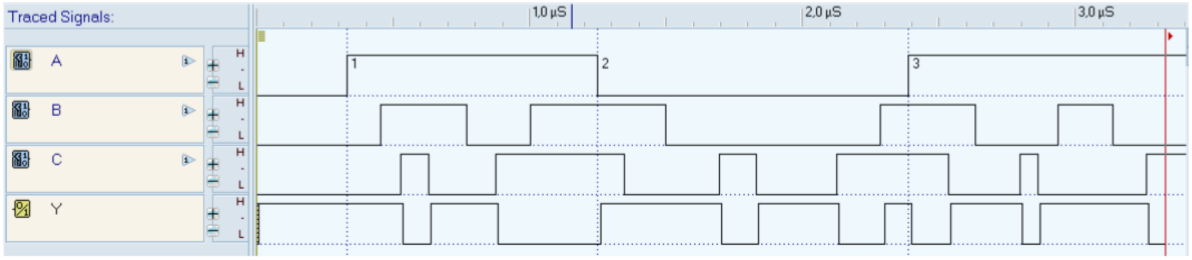
$$= \overline{A}(\overline{C} + B) + A\overline{C}$$

$= \overline{A}\,\overline{C} + \overline{A}B + A\overline{C}$

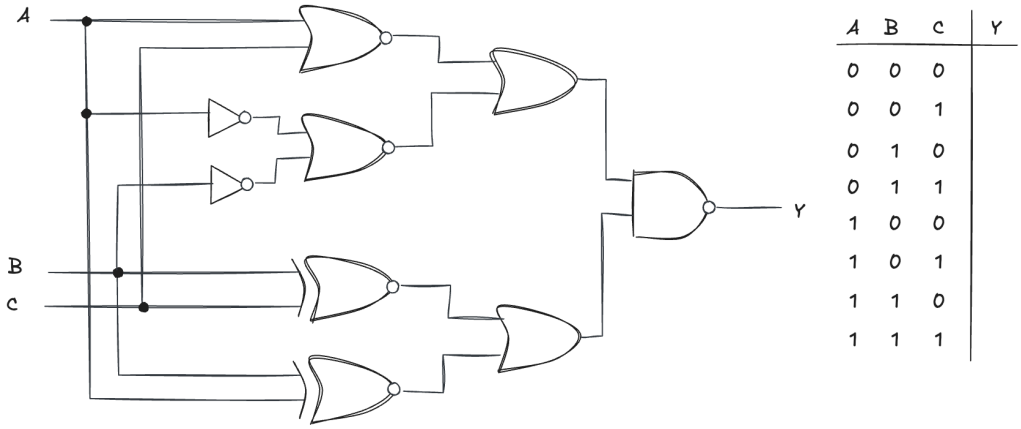
$= \overline{C}(\overline{A} + A) + \overline{A}B$

$= \overline{C}1 + \overline{A}B$

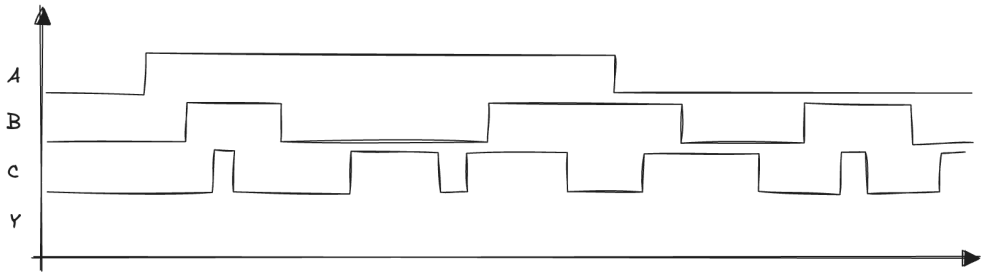
$= \overline{C} + \overline{A}B$

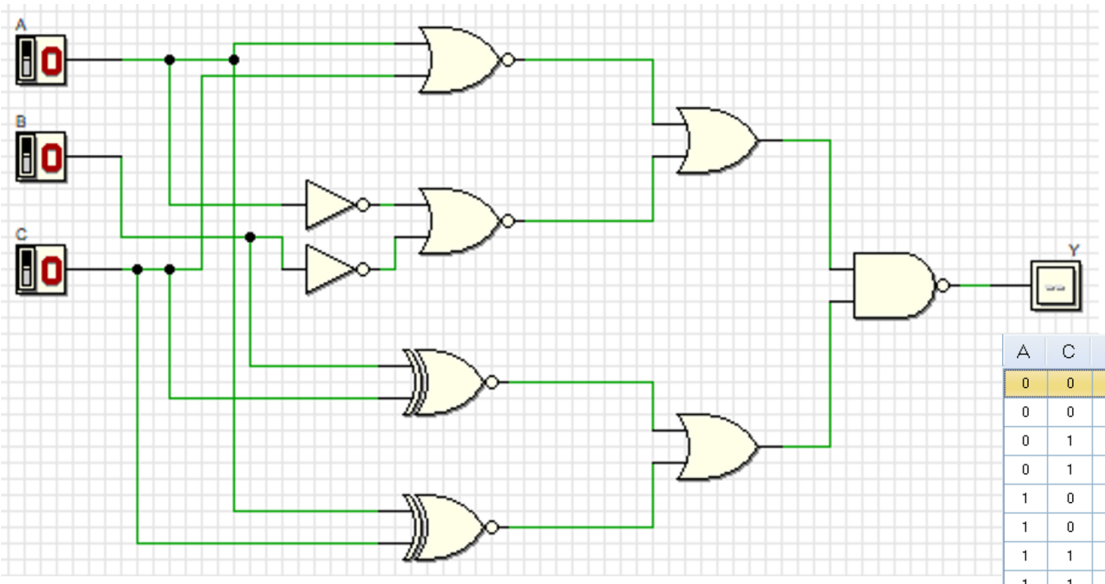


3.2.7 (g)



A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	





A	C	B	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

$$Y = \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}\,\overline{C} + A\overline{B}C + AB\overline{C}$$

$$= \overline{A}B(\overline{C} + C) + A\overline{B}\,\overline{C} + A\overline{B}C + AB\overline{C}$$

$$= \overline{A}B1 + A\overline{B}\,\overline{C} + A\overline{B}C + AB\overline{C}$$

$$= \overline{A}B + A\overline{B}\,\overline{C} + A\overline{B}C + AB\overline{C}$$

$$= \overline{A}B + A\overline{B}(\overline{C} + C) + AB\overline{C}$$

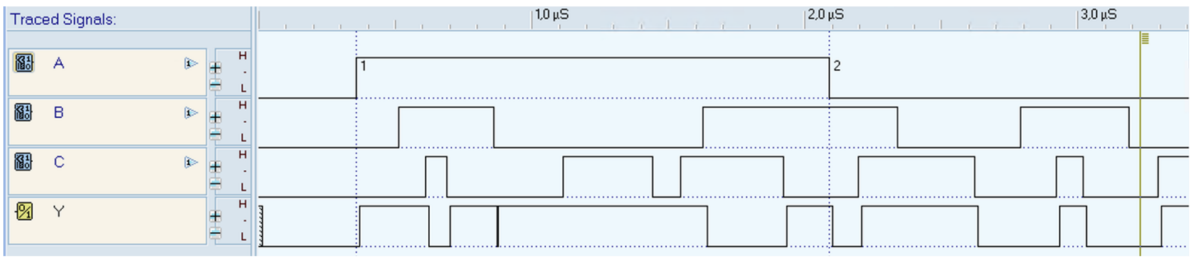
$$= \overline{A}B + A\overline{B}1 + AB\overline{C}$$

$$= \overline{A}B + A\overline{B} + AB\overline{C}$$

$$= A\overline{B} + B(A\overline{C} + \overline{A})$$

$$= A\overline{B} + B(\overline{C} + \overline{A})$$

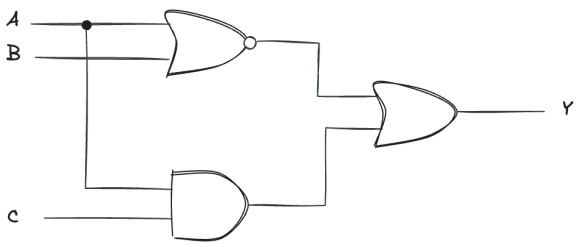
$$= A\overline{B} + B\overline{C} + B\overline{A}$$



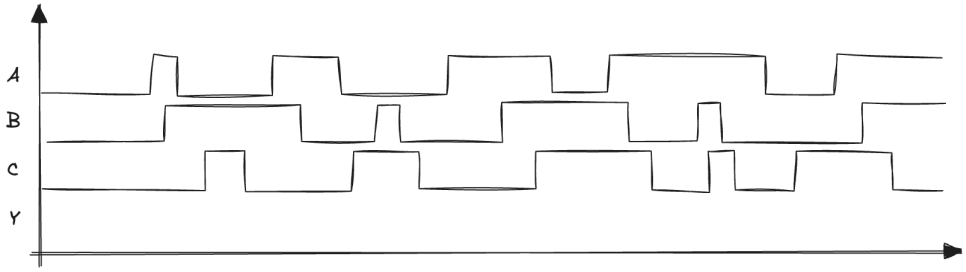
3.3 Exercise 3

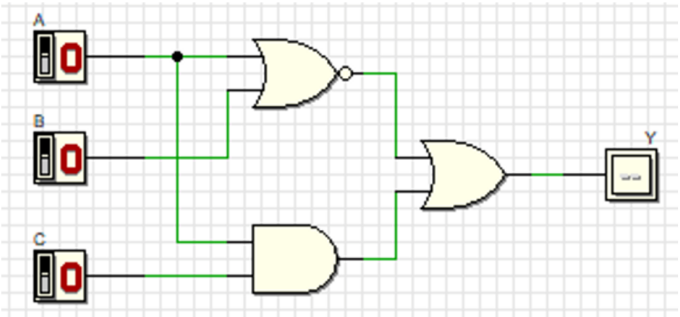
Complete the truth table and the timing diagram of the following digital circuits. Then derive the Boolean equation in canonical **product-of-sums** form and simplify it to its minimal expression using Boolean algebra theorems. Finally, implement the corresponding circuit in DEEDS and verify the correctness of your solution.

3.3.1 (a)



A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	



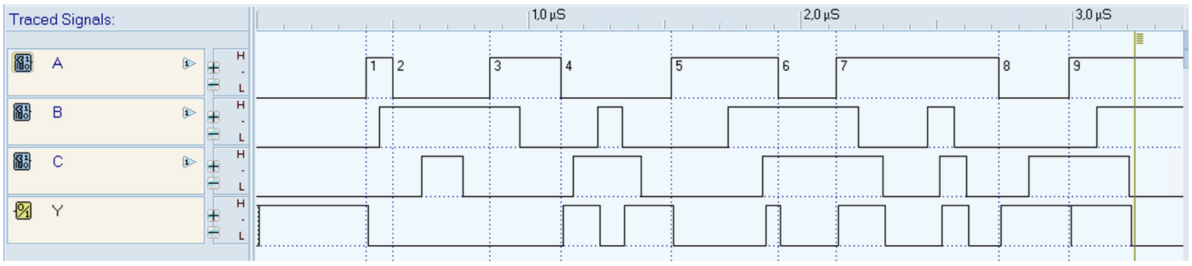


A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

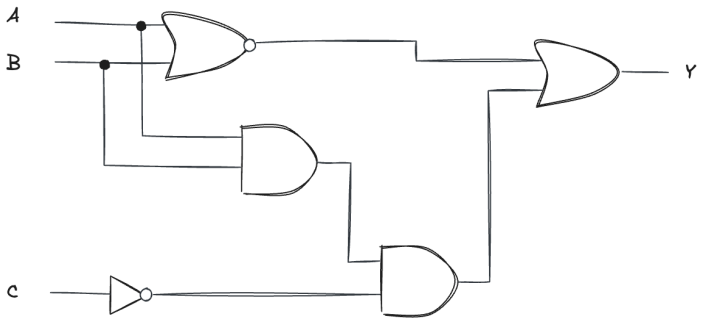
$$Y = (A + \overline{B} + C)(A + \overline{B} + \overline{C})(\overline{A} + B + C)(\overline{A} + \overline{B} + C)$$

$$= (A + \overline{B})(\overline{A} + B + C)(\overline{A} + \overline{B} + C)$$

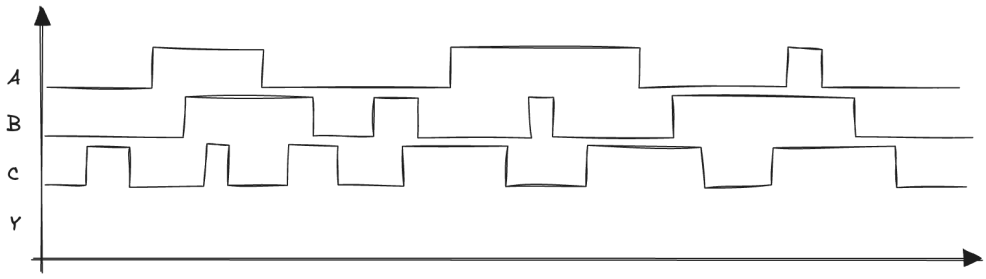
$$= (A + \overline{B})(\overline{A} + C)$$

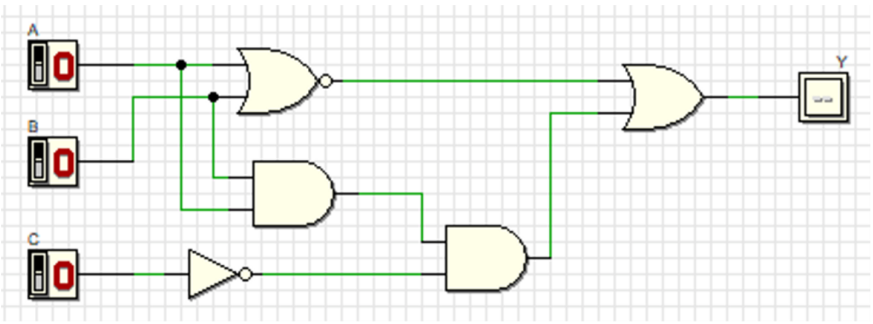


3.3.2 (b)



A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1





A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

$$Y = (A + \overline{B} + C)(A + \overline{B} + \overline{C})(\overline{A} + B + C)(\overline{A} + B + \overline{C})(\overline{A} + \overline{B} + \overline{C})$$

$$= (A + \overline{B})(\overline{A} + B + C)(\overline{A} + B + \overline{C})(\overline{A} + \overline{B} + \overline{C})$$

$$= (A + \overline{B})(\overline{A} + B)(\overline{A} + \overline{B} + \overline{C})$$

Convert so SOP:

$= (A\overline{A} + AB + \overline{B}\,\overline{A} + \overline{B}\,B)(\overline{A} + \overline{B} + \overline{C})$

$= (0 + AB + \overline{A}\,\overline{B} + 0)(\overline{A} + \overline{B} + \overline{C})$

$= (AB + \overline{A}\,\overline{B})(\overline{A} + \overline{B} + \overline{C})$

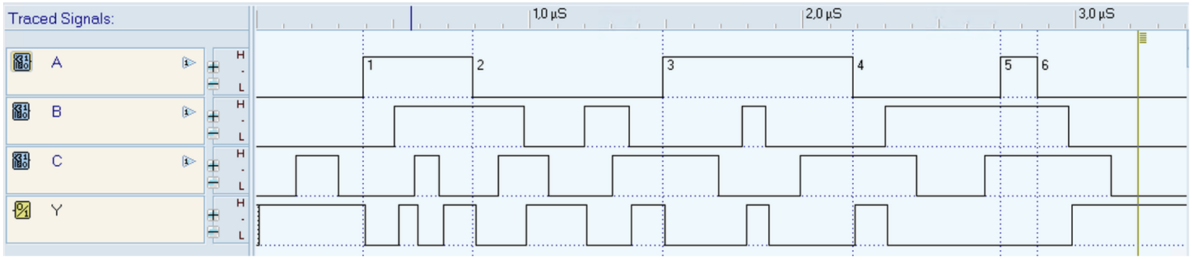
$= AB\overline{A} + AB\overline{B} + AB\overline{C} + \overline{A}\,\overline{B}\,\overline{A} + \overline{A}\,\overline{B}\,\overline{B} + \overline{A}\,\overline{B}\,\overline{C}$

$= 0 + 0 + AB\overline{C} + 0 + \overline{A}\,\overline{B}\,\overline{B} + \overline{A}\,\overline{B}\,\overline{C}$

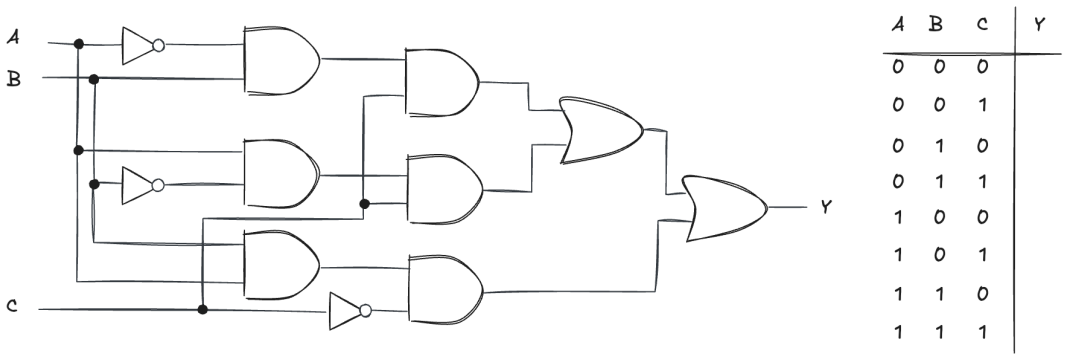
$= AB\overline{C} + \overline{A}\,\overline{B} + \overline{A}\,\overline{B}\,\overline{C}$

$= AB\overline{C} + \overline{A}\,\overline{B}(1 + \overline{C})$

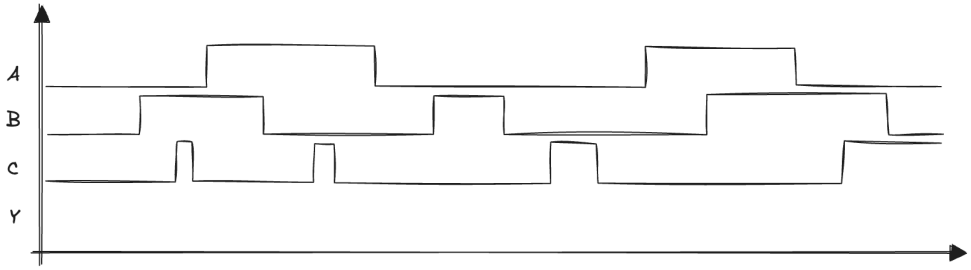
$= AB\overline{C} + \overline{A}\,\overline{B}$

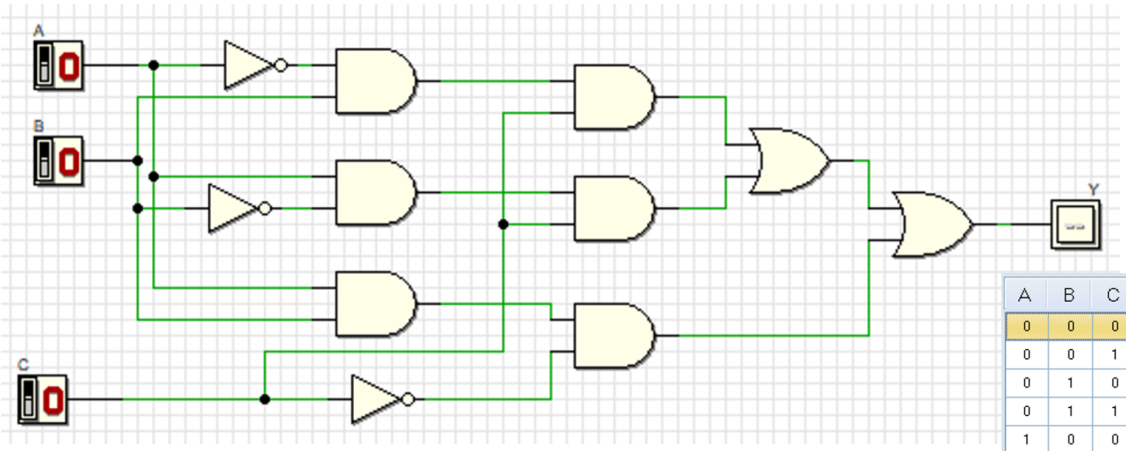


3.3.3 (c)



A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	





A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

$$Y = (A+B+C)(A+B+\overline{C})(A+\overline{B}+C)(\overline{A}+B+C)(\overline{A}+\overline{B}+\overline{C})$$

$$= (A+B)(A+\overline{B}+C)(\overline{A}+B+C)(\overline{A}+\overline{B}+\overline{C})$$

$$= (A+B)(C+(A+\overline{B})(\overline{A}+B))(\overline{A}+\overline{B}+\overline{C})$$

$$= (A+B)(C+AB+\overline{A}\,\overline{B})(\overline{A}+\overline{B}+\overline{C})$$

$$= ((A+B)C+(A+B)AB+(A+B)\overline{A}\,\overline{B})(\overline{A}+\overline{B}+\overline{C})$$

$$= (AC+BC+AAB+BAB+A\overline{A}\,\overline{B}+B\overline{A}\,\overline{B})(\overline{A}+\overline{B}+\overline{C})$$

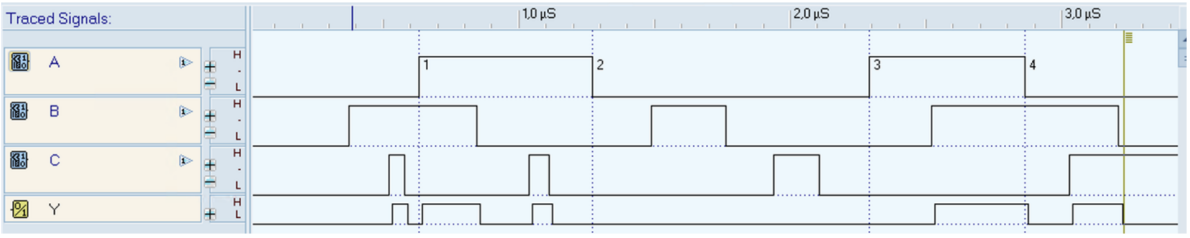
$$= (AC+BC+AB+AB+0+0)(\overline{A}+\overline{B}+\overline{C})$$

$$= (AC+BC+AB)(\overline{A}+\overline{B}+\overline{C})$$

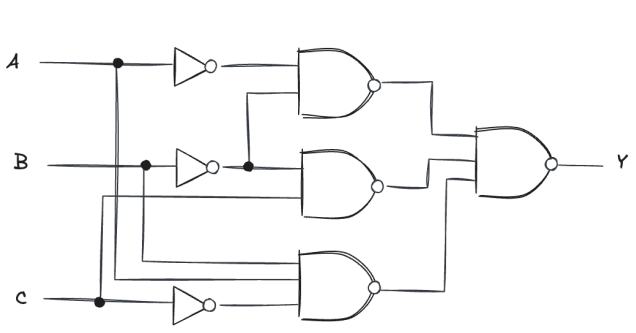
$$= AC\overline{A}+AC\overline{B}+AC\overline{C}+BC\overline{A}+BC\overline{B}+BC\overline{C}+AB\overline{A}+AB\overline{B}+AB\overline{C}$$

$$= 0+AC\overline{B}+0+BC\overline{A}+0+0+0+0+AB\overline{C}$$

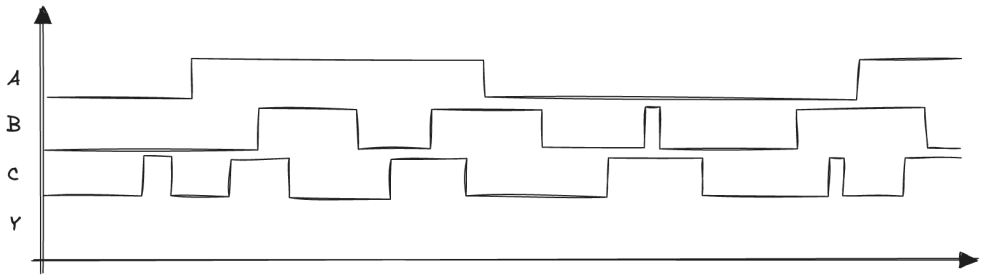
$$= AC\overline{B}+BC\overline{A}+AB\overline{C}$$

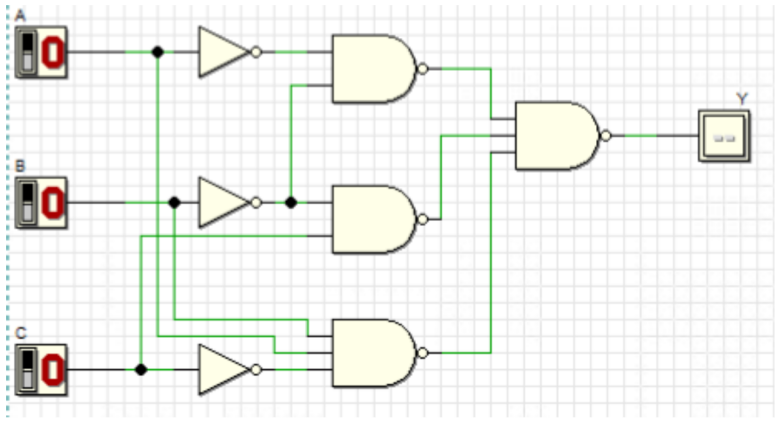


3.3.4 (d)



A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	





A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

$$Y = (A + \overline{B} + C)(A + \overline{B} + \overline{C})(\overline{A} + B + C)(\overline{A} + \overline{B} + \overline{C})$$

$$= (A + \overline{B})(\overline{A} + B + C)(\overline{A} + \overline{B} + \overline{C})$$

$$= (A + \overline{B})(\overline{A} + (B + C)(\overline{B} + \overline{C}))$$

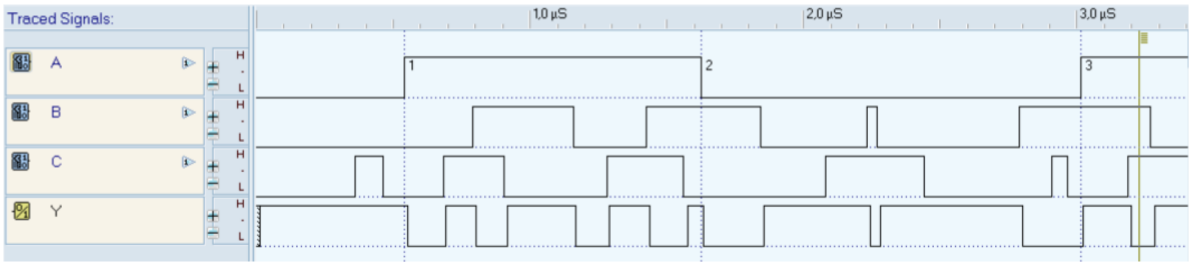
$$= (A + \overline{B})(\overline{A} + B\overline{B} + B\overline{C} + C\overline{B} + C\overline{C})$$

$$= (A + \overline{B})(\overline{A} + B\overline{C} + C\overline{B})$$

$$= A\overline{A} + AB\overline{C} + AC\overline{B} + \overline{B}\,\overline{A} + \overline{B}B\overline{C} + \overline{B}C\overline{B}$$

$$= 0 + AB\overline{C} + AC\overline{B} + \overline{A}\,\overline{B} + 0 + 0$$

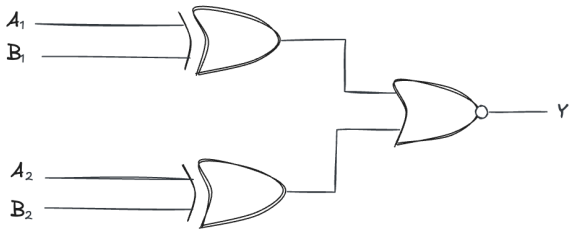
$= AB\overline{C} + AC\overline{B} + \overline{A}\,\overline{B}$



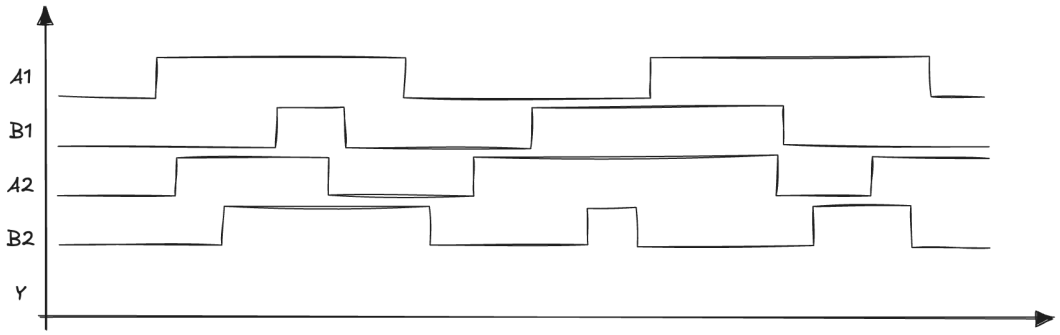
3.4 Exercise 4

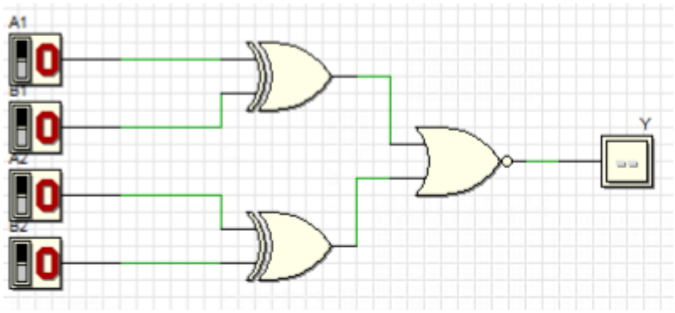
Consider the following circuits. The first one is a **2-bit comparator**, which determines whether the value of A is equal to the value of B. The second one is a **1-bit adder**, which adds A, B, and the input carry, generating both the sum and the carry outputs. Complete the truth table and the timing diagram. Explain the operation of the two circuits in light of the truth table, and verify that they behave as expected. Then implement the corresponding circuits in DEEDS and verify the correctness of your solutions through simulation.

3.4.1 (a) 2-bit comparator



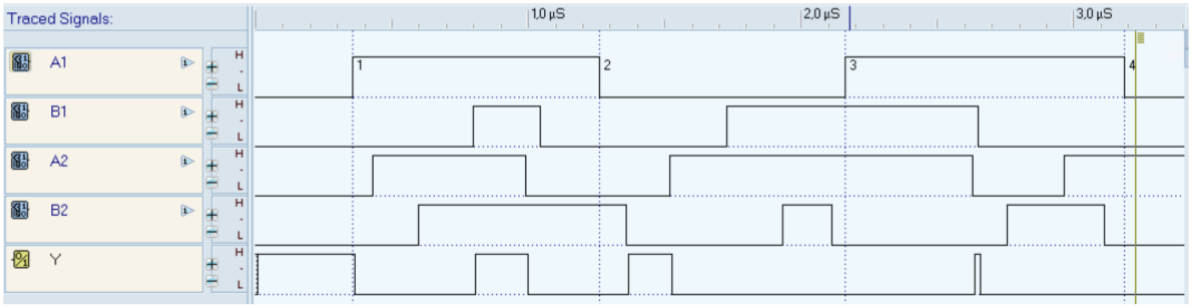
A1	B1	A2	B2	Y
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	



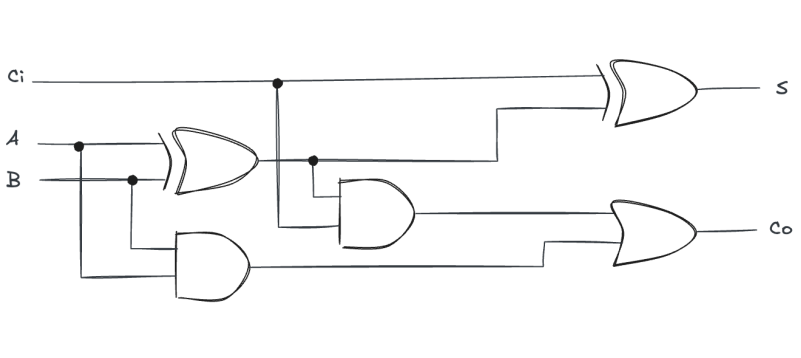


A1	B1	A2	B2	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

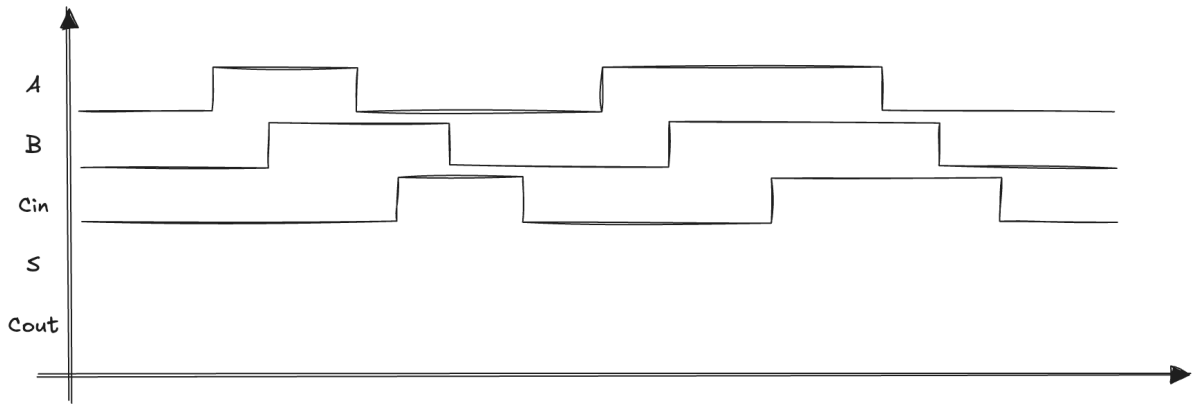
Each row lists all possible combinations of the two 2-bit inputs, A = A2A1 and B = B2B1. The output Y is 1 only when A and B are equal (A1=B1 and A2=B2). By comparing the output Y in the table, we confirm that it equals 1 only when both corresponding bits of A and B match — meaning the comparator works correctly and behaves as expected.

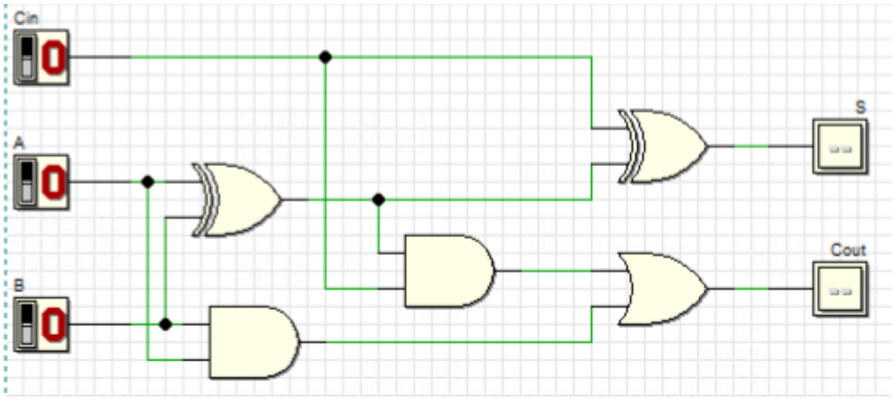


3.4.2 (b) 1-bit adder



A	B	Ci	S	Co
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

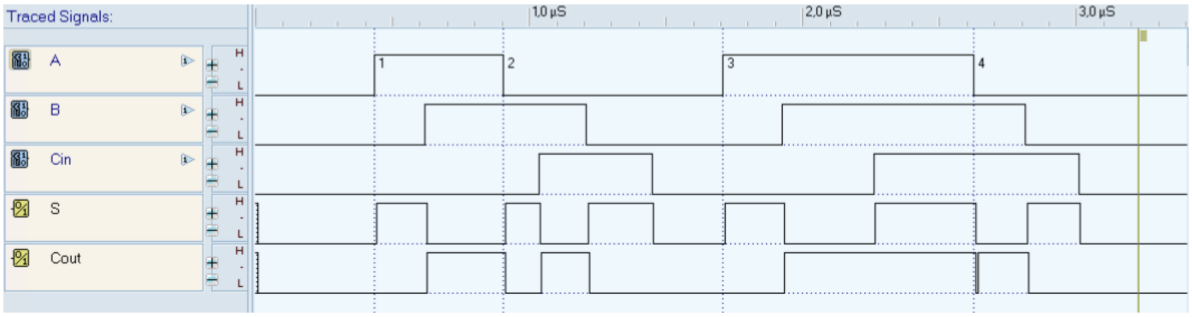




Cin	A	B	S
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

A	B	Cin	Cout
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

The sum output is 1 when an odd number of inputs among A, B, and Cin are 1. The carry output is 1 when at least two of the inputs are 1. The truth tables confirm that the circuit correctly performs binary addition of three inputs, producing the expected sum and carry outputs.



4 Graphical minimization methods Exercises

4.1 Exercise 1

Design the following circuits by proceeding through these steps: define the truth table based on the textual description, construct the corresponding Karnaugh map, minimize the Boolean function, draw the circuit schematic, and finally verify the correctness of the truth table using Deeds.

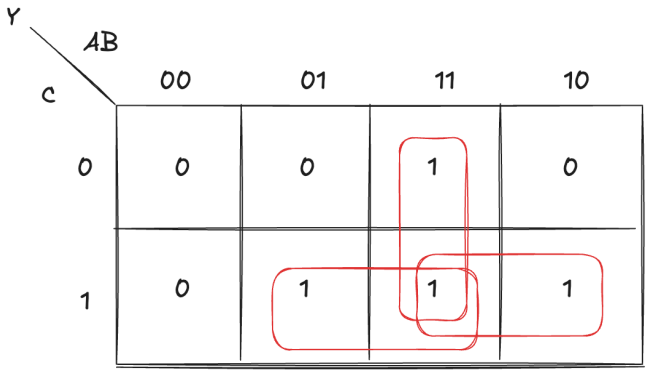
4.1.1 (a) Majority Detector

Design a circuit with three inputs A, B, C and one output Y, which is 1 whenever at least two inputs are 1.

Truth Table:

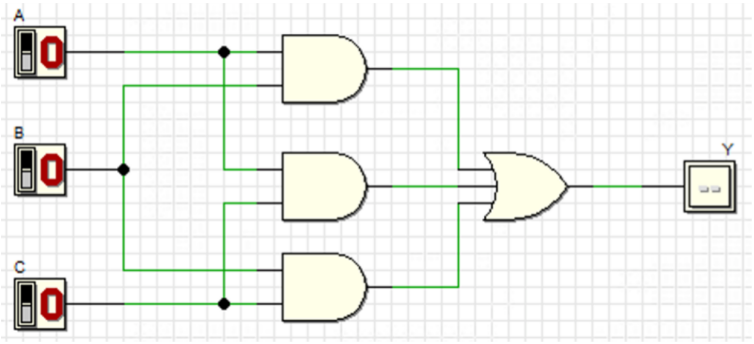
A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Karnaugh Map:



Minimized Boolean Function: $Y = AB + AC + BC$

Circuit Schematic and verification:



A	C	B	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

4.1.2 (b) Odd Parity Detector

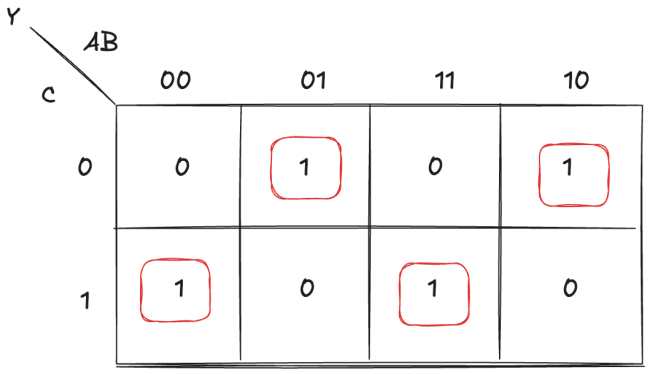
Design a circuit with three inputs A, B, C and output Y = 1 if the number of 1s is odd

Truth Table:

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0

A	B	C	Y
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

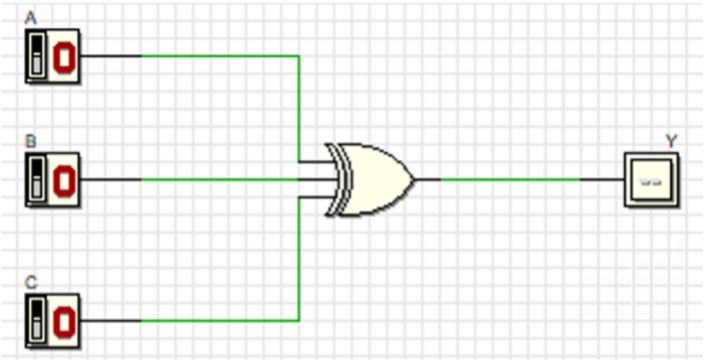
Karnaugh Map:



The Karnaugh maps for the parity checker clearly show a **symmetry** called **checkerboard pattern**. This pattern reflects the nature of the function: the output is 1 whenever the number of 1s among the inputs is odd. This pattern can be exploited to directly write the minimized Boolean function as an XOR operation among all the inputs:

$Y = A \oplus B \oplus C$

Schematic and verification:



A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

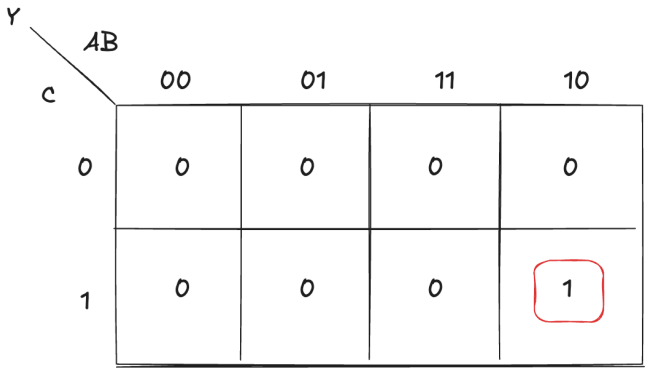
4.1.3 (c) Code Detector

Design a circuit with three inputs A, B, C. The output Y = 1 only when the input pattern is 101.

Truth Table:

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

Karnaugh Map:

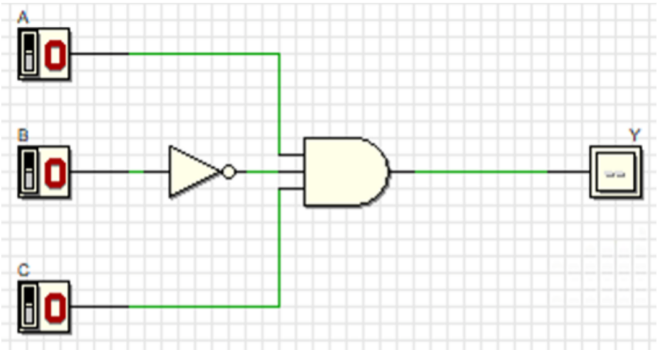


The function corresponds to a single minterm:

$Y = A\overline{B}C$

This circuit can be implemented with one AND gate taking inputs. It represents the simplest form of a combinational detector, producing a high output only for an exact binary pattern 101. For example, it can be used in digital systems to recognize specific command codes or data sequences.

Schematic and verification:



A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

4.1.4 (d) Greater Than 9

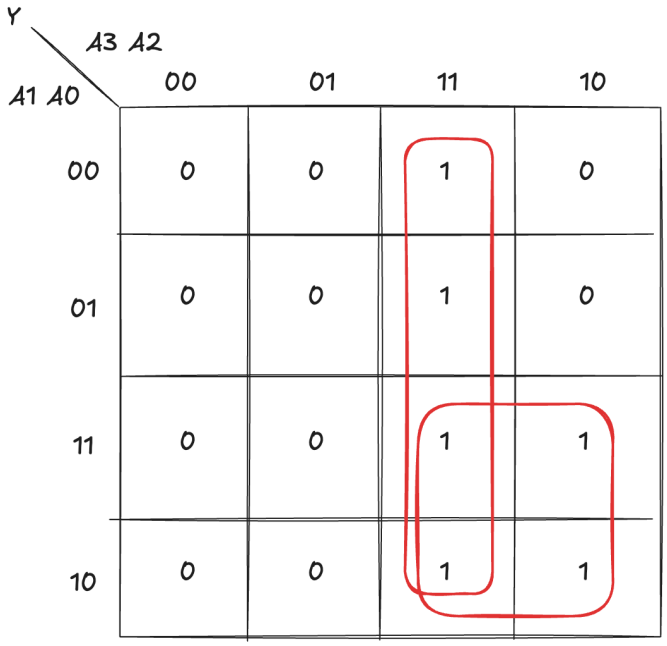
Design a circuit that takes a 4-bit input A3 A2 A1 A0 and outputs Y=1 if the unsigned decimal number represented by the binary input is greater than 9 (decimal 10–15).

Trueth Table:

A3	A2	A1	A0	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1

A3	A2	A1	A0	Y
1	1	1	0	1
1	1	1	1	1

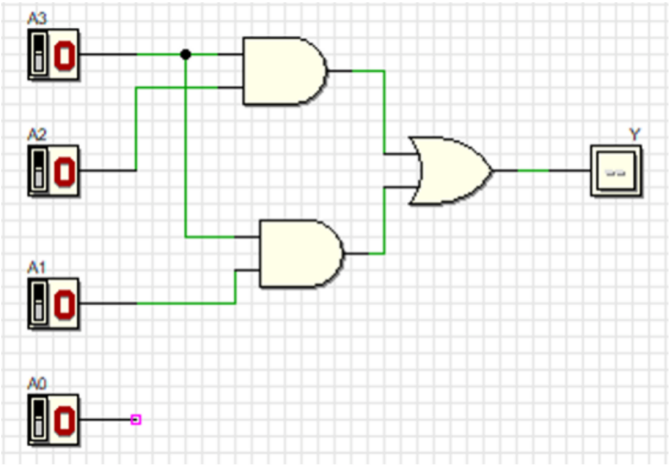
Karnaugh Map:



$Y = A_3A_2 + A_3A_1$

In this circuit, the variable A0 (the least significant bit) does not affect the output, since the condition "number > 9" depends only on the three most significant bits (A3, A2 and A_1). In other words, A0 is **irrelevant** (**a don't-care variable**) for this function, which is why it does not appear in the simplified Boolean expression.

Schematic and verification:



A3	A2	A1	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

4.1.5 (e) 2-bit Comparator

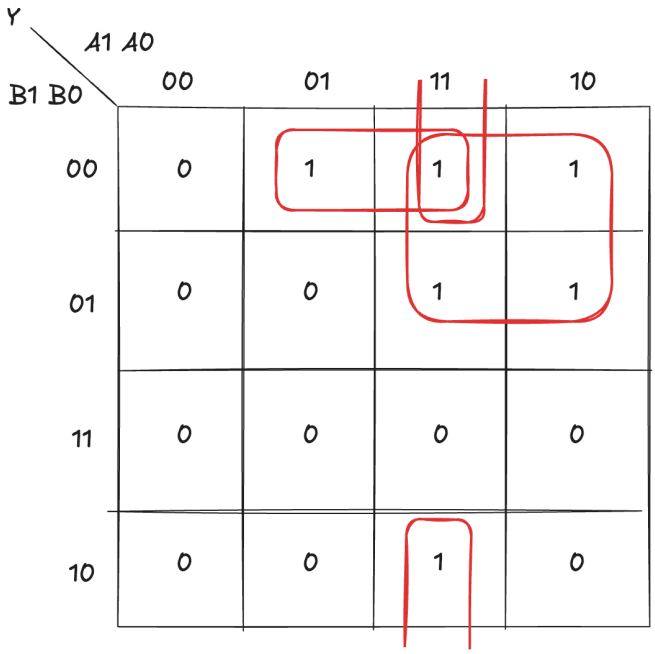
Design a circuit with two 2-bit binary numbers A1 A0 and B1 B0. The output Y = 1 if the unsigned decimal number represented by A is greater than the unsigned decimal number represented by B.

Truth Table:

A1	A0	B1	B0	A (decimal)	B (decimal)	Y
0	0	0	0	0	0	0
0	0	0	1	0	1	0
0	0	1	0	0	2	0
0	0	1	1	0	3	0
0	1	0	0	1	0	1
0	1	0	1	1	1	0
0	1	1	0	1	2	0
0	1	1	1	1	3	0
1	0	0	0	2	0	1
1	0	0	1	2	1	1
1	0	1	0	2	2	0
1	0	1	1	2	3	0

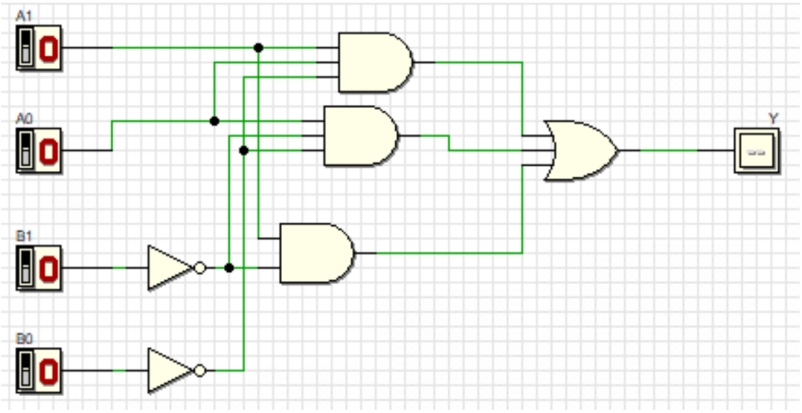
A1	A0	B1	B0	A (decimal)	B (decimal)	Y
1	1	0	0	3	0	1
1	1	0	1	3	1	1
1	1	1	0	3	2	1
1	1	1	1	3	3	0

Karnaugh Map:



$Y = A_1\overline{B_1} + A_0\overline{B_0}\,\overline{B_1} + A_1A_0\overline{B_0}$

Schematic and verification:



A1	A0	B1	B0	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

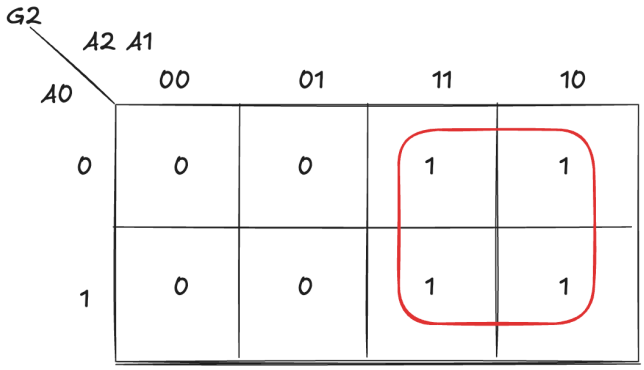
4.1.6 (f) Binary-to-Gray Code Converter

Design a circuit that converts a 3-bit binary input (A2 A1 A0) to its Gray code (G2 G1 G0)

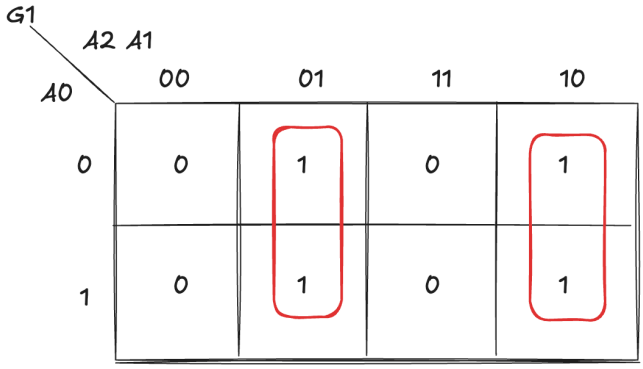
Truth Table:

A2	A1	A0	G2	G1	G0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

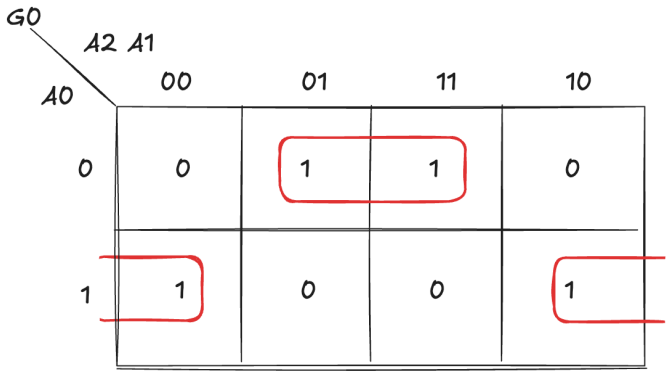
Karnaugh Map:



$G_2 = A_2$

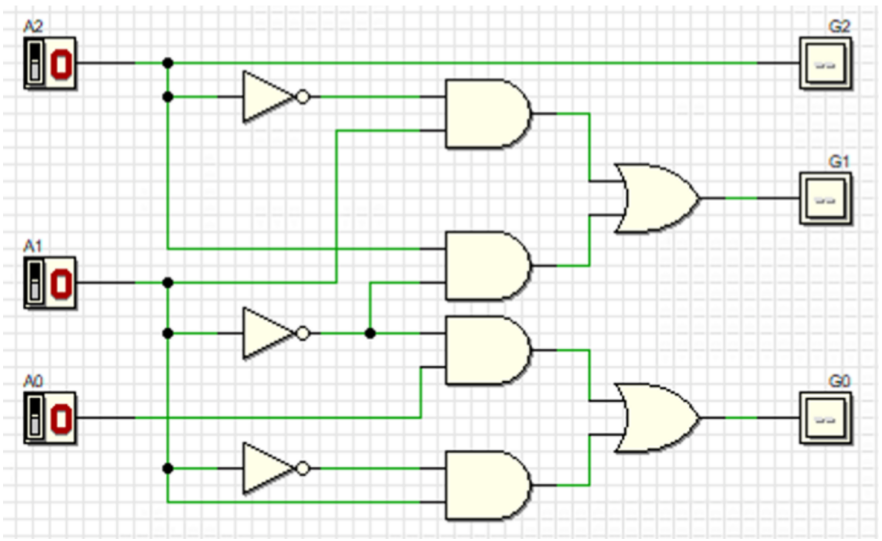


$G_1 = A_2\overline{A_1} + \overline{A_2}A_1$



$G_0 = A_1\overline{A_0} + \overline{A_1}A_0$

Schematic and verification:



A2	G2
0	0
1	1

A2	A1	G1
0	0	0
0	1	1
1	0	1
1	1	0

A1	A0	G0
0	0	0
0	1	1
1	0	1
1	1	0

4.2 Exercise 2

Simplify the following Boolean expressions using two different approaches: using Boolean algebra theorems and Karnaugh maps.

4.2.1 (a)

$G = ABC + B\overline{C}$

Boolean Algebra:

$G = B(AC + \overline{C})$

$G = B(A + \overline{C})$

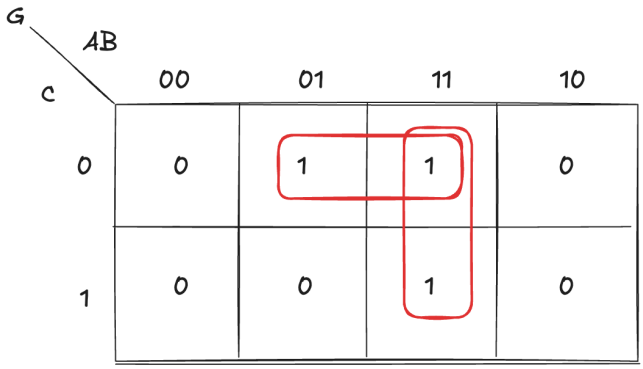
$G = AB + B\overline{C}$

Truth Table:

A	B	C	G
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0

A	B	C	G
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Karnaugh Map:



$G = AB + B\overline{C}$

4.2.2 (b)

$H = (A + \overline{B})(B + \overline{C})$

Boolean Algebra:

$H = AB + A\overline{C} + \overline{B}B + \overline{B}\,\overline{C}$

$H = AB + A\overline{C} + \overline{B}\,\overline{C}$

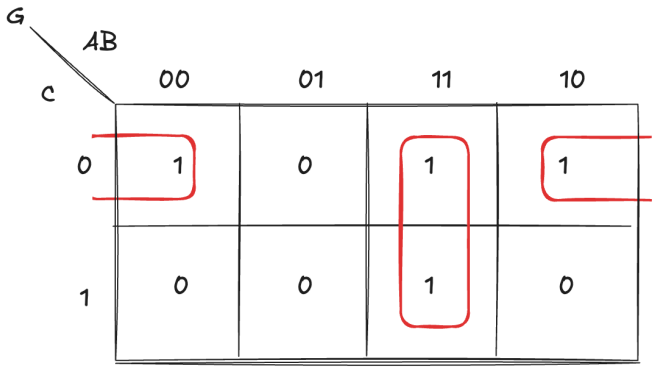
$H = AB + \overline{B}\,\overline{C}$

Truth Table:

A	B	C	H
0	0	0	1
0	0	1	0
0	1	0	0

A	B	C	H
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Karnaugh Map:



$H = AB + \overline{B}\,\overline{C}$

4.2.3 (c)

$Y = BC\overline{D} + CD + A\overline{B}C\overline{D} + \overline{A}\,\overline{B}C$

Boolean Algebra:

$Y = BC\overline{D} + CD + A\overline{B}C\overline{D} + \overline{A}\,\overline{B}C$

$Y = C(B\overline{D} + D) + A\overline{B}C\overline{D} + \overline{A}\,\overline{B}C$

$Y = C(B + D) + A\overline{B}C\overline{D} + \overline{A}\,\overline{B}C$

$Y = C(B + D) + \overline{B}C(A\overline{D} + \overline{A})$

$Y = C(B + D) + \overline{B}C(\overline{D} + \overline{A})$

$Y = CB + CD + \overline{B}C(\overline{D} + \overline{A})$

$Y = CD + C(\overline{B}(\overline{D} + \overline{A}) + B)$

$Y = CD + C(\overline{D} + \overline{A} + B)$

$Y = CD + C\overline{D} + C\overline{A} + CB$

$Y = C(D + \overline{D}) + C\overline{A} + CB$

$Y = C + C\overline{A} + CB$

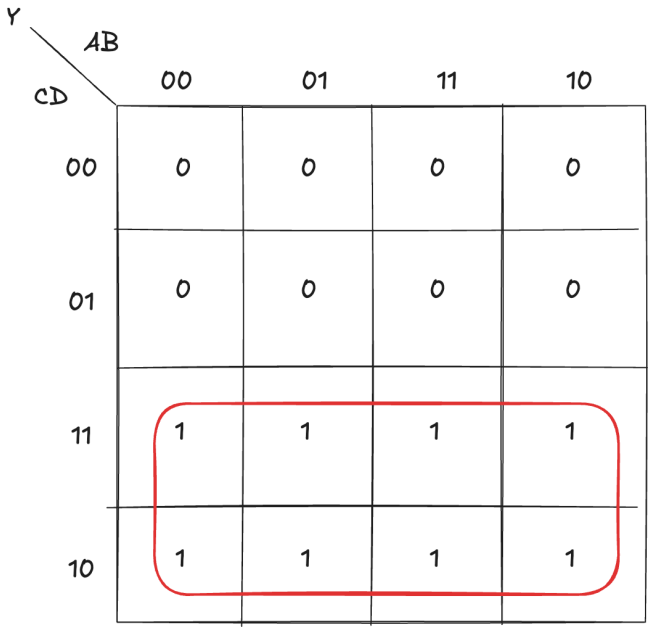
$Y = C(1 + \overline{A} + B)$

$Y = C$

Truth Table:

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

Karnaugh Map:



$Y = C$

4.2.4 (d)

$Y = AB + A\overline{B}\,\overline{C}\,\overline{D}$

Boolean Algebra:

$Y = AB + A\overline{B}\,\overline{C}\,\overline{D}$

$Y = A(B + \overline{B}\,\overline{C}\,\overline{D})$

$Y = A(B + \overline{C}\,\overline{D})$

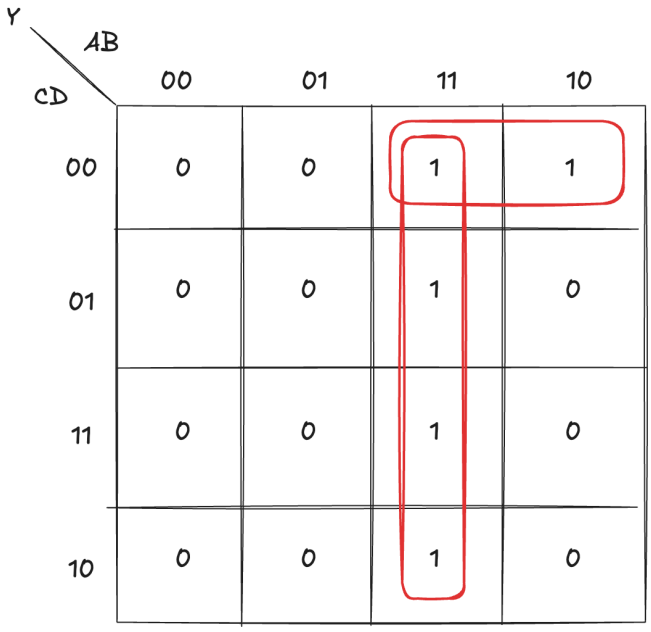
$Y = AB + A\overline{C}\,\overline{D}$

Truth Table:

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0

A	B	C	D	Y
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

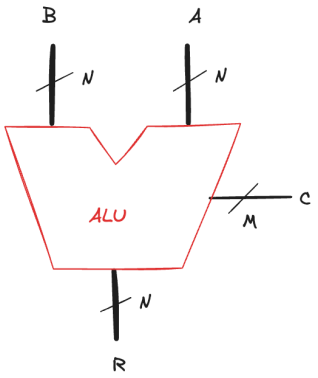
Karnaugh Map:



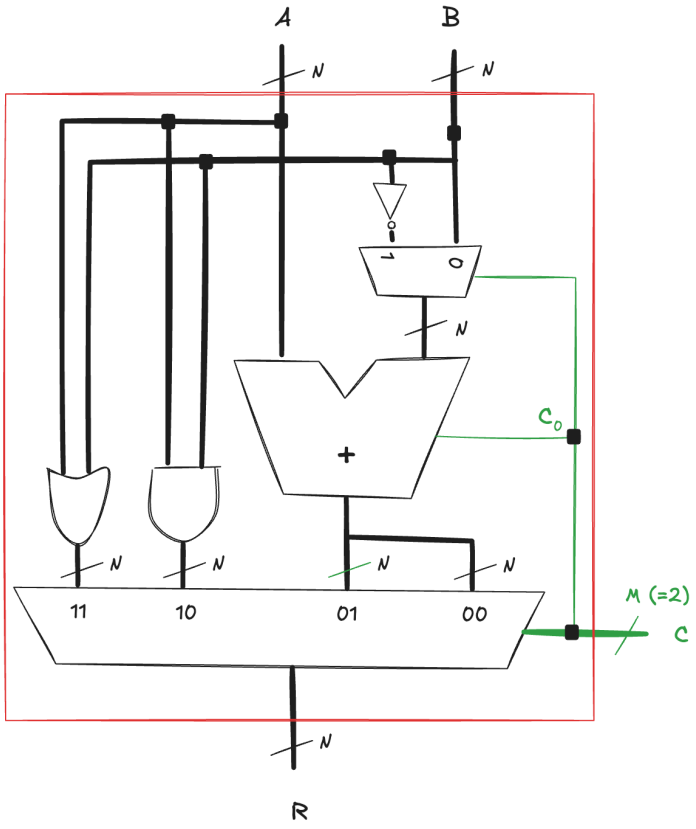
$Y = AB + A\overline{C}\,\overline{D}$

5 Arithmetic/Logical Unit Exercises

Realize using DEEDS and VHDL a simple Arithmetic/Logical Unit (ALU) as described during the lectures. The ALU should be able to perform 8-bit addition, subtraction, bitwise AND, and bitwise OR operations:



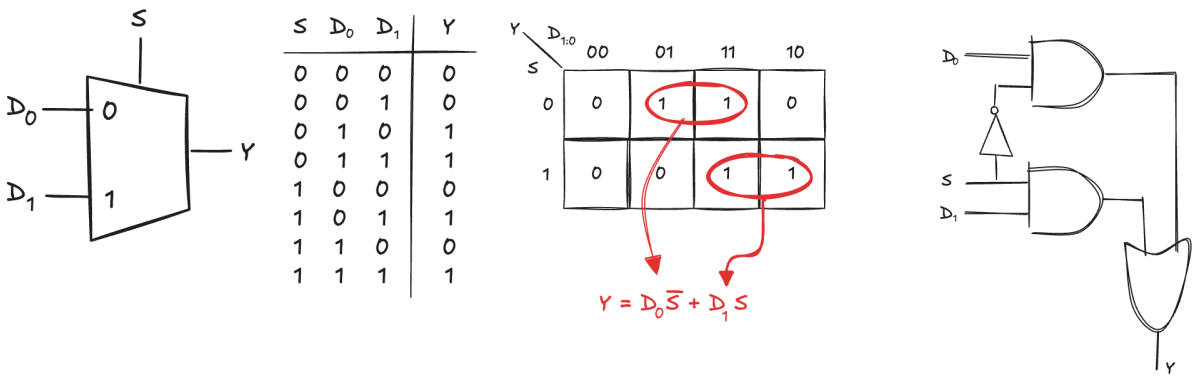
Control C_1	Control C_0	Funtion
0	0	Add
0	1	Subtract
1	0	AND
1	1	OR



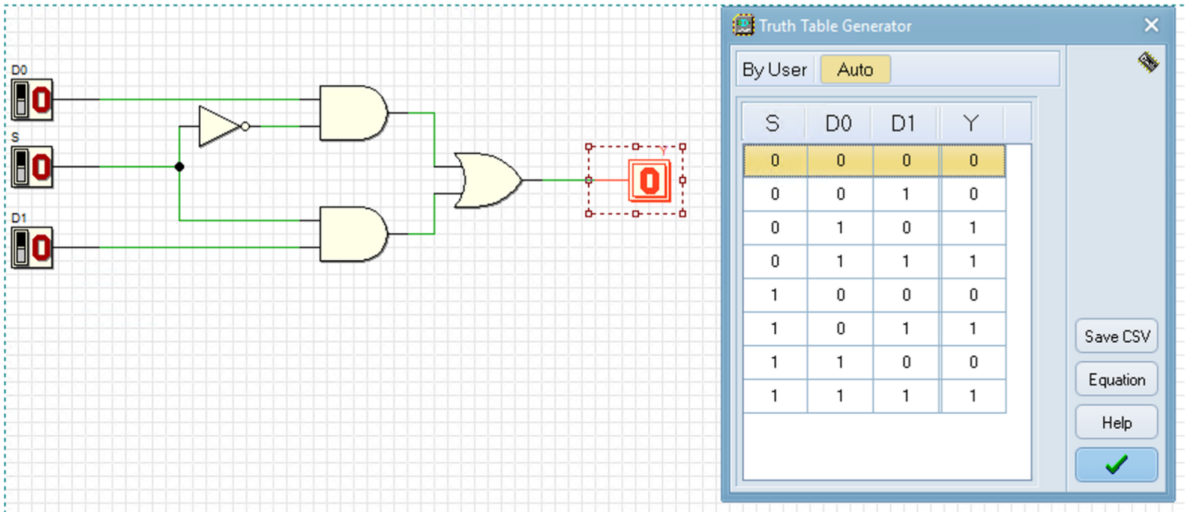
To build the ALU, we combine several key components: a 2-bit control decoder that selects the desired operation, a 4-to-1 multiplexer that chooses the final output, an 8-bit adder for arithmetic, and 8-bit AND and OR logic blocks for bitwise operations.

5.1 2-bit Multiplexer

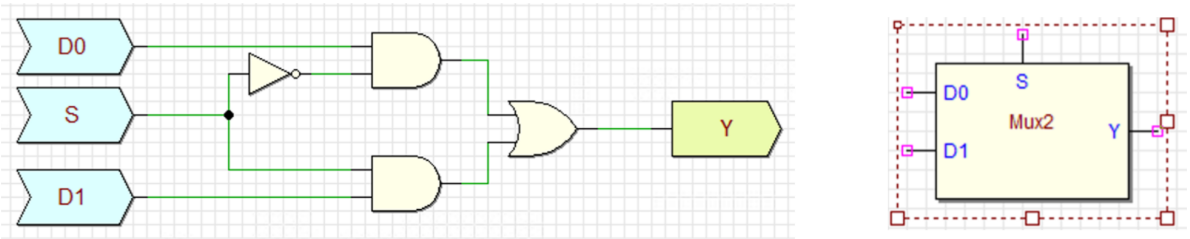
Create a 2-to-1 multiplexer (mux), a circuit that selects one of two input signals and forwards it to the output depending on the value of a single select line.



Design the circuit using DEEDS and simulate its truth table to verify its functionality.

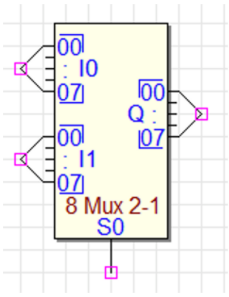


DEEDS allows us to create custom block components that can be reused in larger designs. Create a new block component for the multiplexer.



However, the multiplexer we need must handle two 8-bit inputs, so we must build an 8-bit version of the 2-to-1 mux. We can construct it by reusing the 1-bit mux as a building block and instantiating eight of them in parallel. DEEDS also provides bus-based components that simplify this task, allowing us to bundle the signals and connect the mux array more efficiently. Search the DEEDS component library and find a bus-based 2-to-1 multiplexer to use in our

design.



Develop the VHDL implementation of the 8-bit 2-to-1 multiplexer and add the source file to a Vivado project configured for the Artix-7 (xc7a200tfbg484-1) FPGA device.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Mux2 is
    port(
        D0 : in  std_logic_vector(7 downto 0);
        D1 : in  std_logic_vector(7 downto 0);
        S  : in  std_logic;
        Y  : out std_logic_vector(7 downto 0)
    );
end entity Mux2;

architecture logic of Mux2 is
begin
    Y <= D0 when S = '0' else
         D1;
end architecture logic;
```

Create a testbench to validate the behavior of the 2-bit multiplexer. Simulate the design and verify that it produces the correct output for every possible input combination.

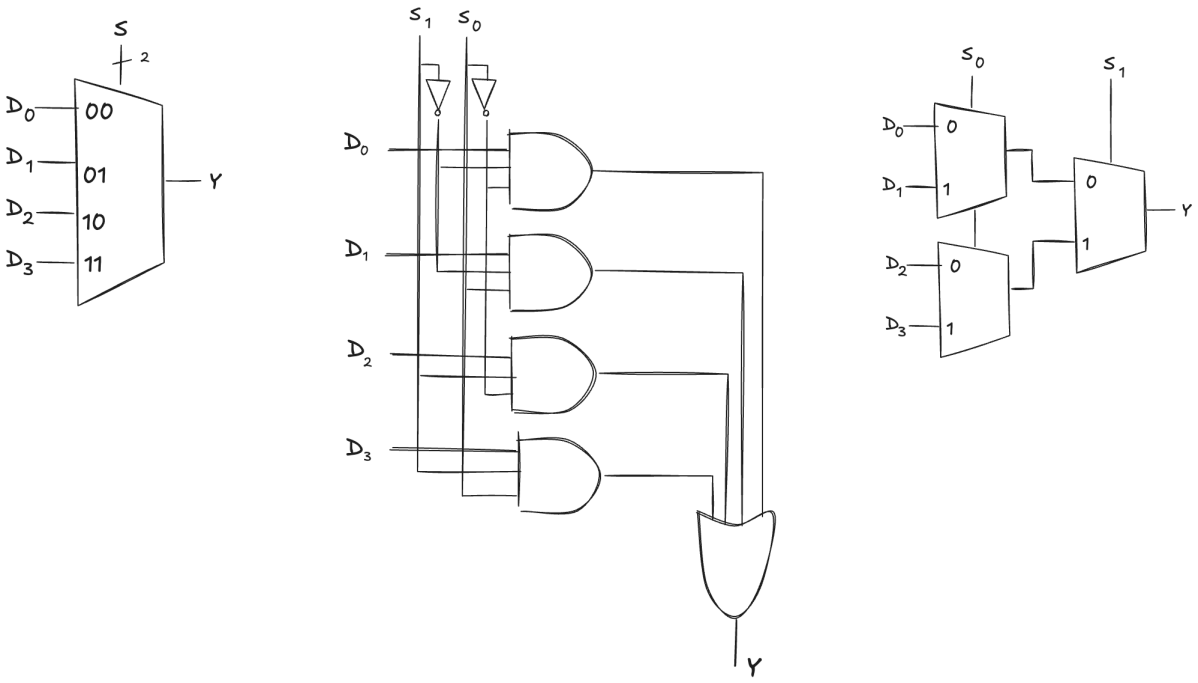
```
library IEEE;
use IEEE.std_logic_1164.all;

entity tb_Mux2 is
end entity tb_Mux2;

architecture behavior of tb_Mux2 is
```

```
    component Mux2
        port(
            D0 : in  std_logic_vector(7 downto 0);
            D1 : in  std_logic_vector(7 downto 0);
            S  : in  std_logic;
            Y  : out std_logic_vector(7 downto 0)
        );
    end component;

    signal D0_tb : std_logic_vector(7 downto 0);
    signal D1_tb : std_logic_vector(7 downto 0);
    signal S_tb  : std_logic;
    signal Y_tb  : std_logic_vector(7 downto 0);

begin

    UUT: Mux2
        port map(
            D0 => D0_tb,
            D1 => D1_tb,
            S  => S_tb,
            Y  => Y_tb
        );

    stim_proc: process
    begin
        -------------------------------------------------------------------
        -- Test 1
        -------------------------------------------------------------------
        D0_tb <= "00001111";    -- 0x0F
        D1_tb <= "11110000";    -- 0xF0
        S_tb  <= '0';
        wait for 10 ns;

        assert (Y_tb = D0_tb)
            report "ERROR: Test 1 failed (S=0, expected Y=D0)"
            severity error;

        -------------------------------------------------------------------
        -- Test 2
        -------------------------------------------------------------------
        S_tb <= '1';
```

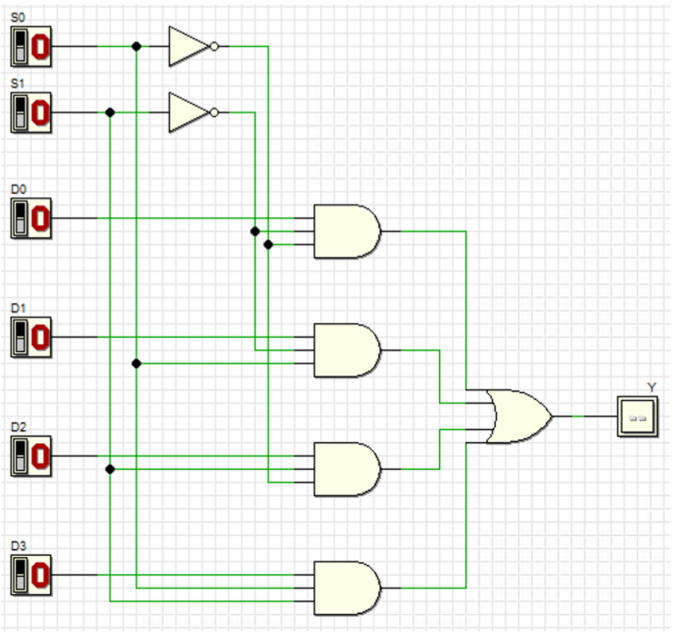
```
        wait for 10 ns;

        assert (Y_tb = D1_tb)
            report "ERROR: Test 2 failed (S=1, expected Y=D1)"
            severity error;

        ----------------------------------------------------------------------
        -- Test 3
        ----------------------------------------------------------------------
        D0_tb <= "10101010";    -- 0×AA
        D1_tb <= "01010101";    -- 0×55
        S_tb <= '0';
        wait for 10 ns;

        assert (Y_tb = D0_tb)
            report "ERROR: Test 3 failed (S=0, expected Y=D0)"
            severity error;

        ----------------------------------------------------------------------
        -- Test 4
        ----------------------------------------------------------------------
        S_tb <= '1';
        wait for 10 ns;

        assert (Y_tb = D1_tb)
            report "ERROR: Test 4 failed (S=1, expected Y=D1)"
            severity error;

        report "Mux2 tests passed successfully!" severity note;

        wait;
    end process;

end architecture behavior;
```

5.2 4-bit Multiplexer

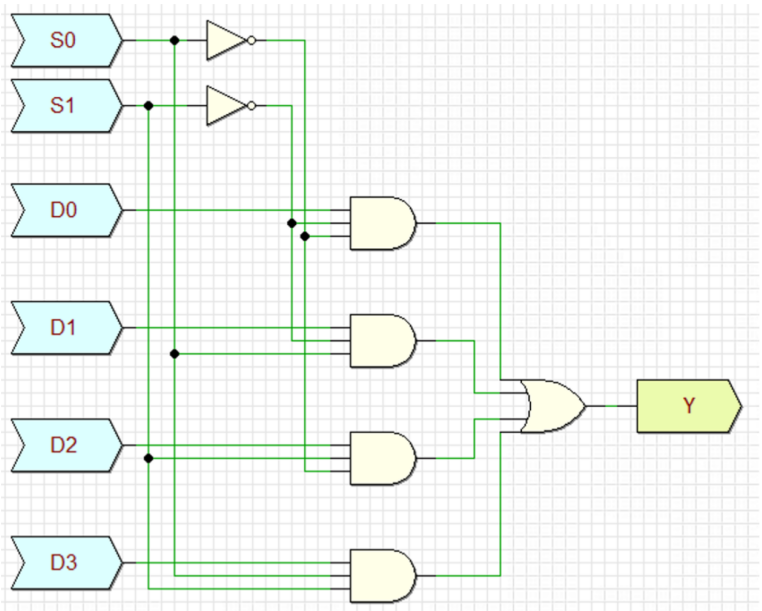
We need a 4-to-1 multiplexer to select the correct value to drive to the output based on the input control signals:

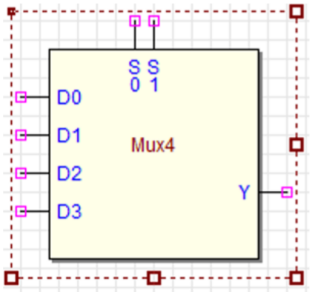


Design, using DEEDS, a 1-bit multiplexer that chooses one of four inputs, with the two select signals determining which input is routed to the output.

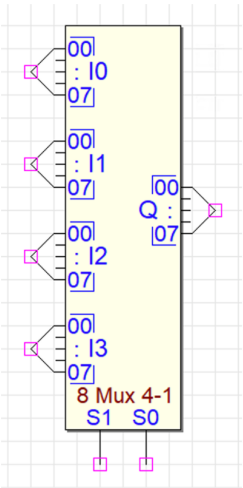


Build the DEEDS block component that implements the 4-to-1 multiplexer, so it can be reused as a module in more complex designs.





As before, we need to route four 8-bit input signals, so we must build an 8-bit version of the 4-to-1 multiplexer. This can be done by reusing the 1-bit 4-to-1 mux as a building block and instantiating eight of them in parallel. DEEDS also provides bus-based components that simplify this process and allow us to bundle and connect the signals efficiently. Find the bus-based 4-to-1 multiplexer in the DEEDS component library to incorporate into our design.



Develop the VHDL implementation of the 8-bit 4-to-1 multiplexer and include it in the Vivado project.

```
library IEEE;
use IEEE.std_logic_1164.all;
```

```
entity Mux4 is
    port(
        D0 : in std_logic_vector(7 downto 0);
        D1 : in std_logic_vector(7 downto 0);
        D2 : in std_logic_vector(7 downto 0);
        D3 : in std_logic_vector(7 downto 0);
        S0 : in std_logic;
        S1 : in std_logic;
        Y  : out std_logic_vector(7 downto 0)
    );
end entity Mux4;

architecture logic of Mux4 is
    signal sel : std_logic_vector(1 downto 0);
begin
    sel <= S1 & S0;
    -- 4-to-1 selection based on S1 S0
    with sel select
        Y <= D0 when "00",
             D1 when "01",
             D2 when "10",
             D3 when "11",
             (others => '0') when others;  -- safe default
end architecture logic;
```

Write a testbench to verify the functionality of the multiplexer. Run a simulation and check that the circuit produces the correct output for every possible combination of inputs.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity tb_Mux4 is
end entity tb_Mux4;

architecture behavior of tb_Mux4 is

    -- Component declaration (8-bit version)
    component Mux4 is
        port(
            D0 : in std_logic_vector(7 downto 0);
            D1 : in std_logic_vector(7 downto 0);
```

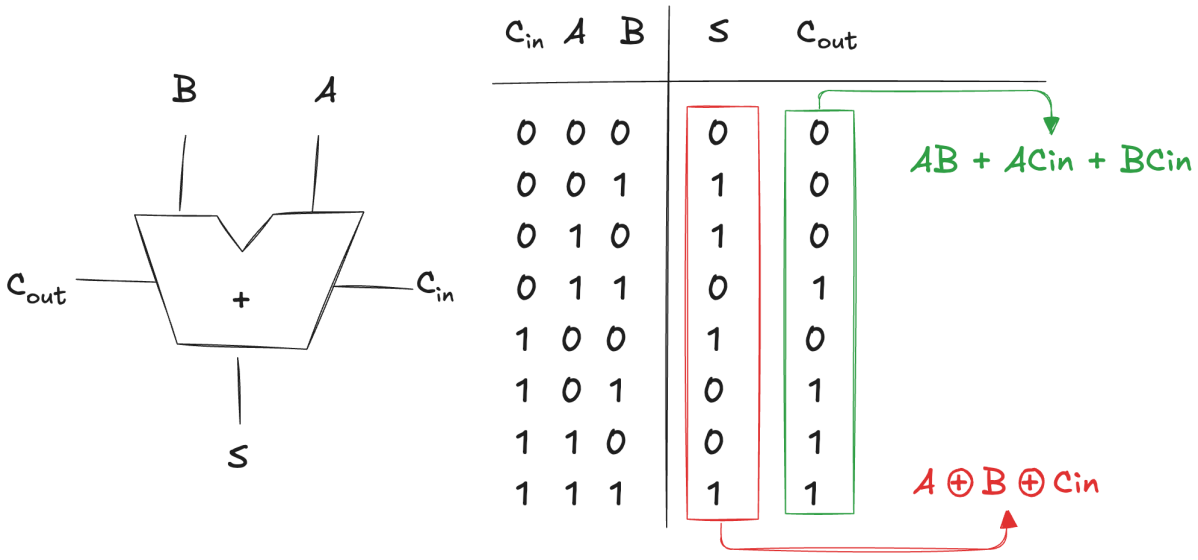
```
            D2 : in std_logic_vector(7 downto 0);
            D3 : in std_logic_vector(7 downto 0);
            S0 : in std_logic;
            S1 : in std_logic;
            Y  : out std_logic_vector(7 downto 0)
        );
    end component;

    -- Signals for test
    signal D0, D1, D2, D3 : std_logic_vector(7 downto 0);
    signal S0, S1         : std_logic;
    signal Y              : std_logic_vector(7 downto 0);

begin

    -- Instantiate UUT
    uut: Mux4
        port map(
            D0 => D0,
            D1 => D1,
            D2 => D2,
            D3 => D3,
            S0 => S0,
            S1 => S1,
            Y  => Y
        );

    -- Stimulus process
    process
    begin
        ---------------------------------------------------------------
        -- First test pattern
        ---------------------------------------------------------------
        D0 <= x"0F";   -- 0000 1111
        D1 <= x"F0";   -- 1111 0000
        D2 <= x"AA";   -- 1010 1010
        D3 <= x"55";   -- 0101 0101

        -- S1 S0 = 00 => Y = D0
        S1 <= '0'; S0 <= '0'; wait for 10 ns;
        assert (Y = D0)
            report "ERROR: S1S0=00 => expected Y=D0"
            severity error;
```

```
        -- S1 S0 = 01 ⇒ Y = D1
        S1 ⩽ '0'; S0 ⩽ '1'; wait for 10 ns;
        assert (Y = D1)
            report "ERROR: S1S0=01 ⇒ expected Y=D1"
            severity error;

        -- S1 S0 = 10 ⇒ Y = D2
        S1 ⩽ '1'; S0 ⩽ '0'; wait for 10 ns;
        assert (Y = D2)
            report "ERROR: S1S0=10 ⇒ expected Y=D2"
            severity error;

        -- S1 S0 = 11 ⇒ Y = D3
        S1 ⩽ '1'; S0 ⩽ '1'; wait for 10 ns;
        assert (Y = D3)
            report "ERROR: S1S0=11 ⇒ expected Y=D3"
            severity error;

        ----------------------------------------------------------------
        -- Second test pattern
        ----------------------------------------------------------------
        D0 ⩽ x"12";
        D1 ⩽ x"34";
        D2 ⩽ x"56";
        D3 ⩽ x"78";

        -- S1 S0 = 00 ⇒ Y = D0
        S1 ⩽ '0'; S0 ⩽ '0'; wait for 10 ns;
        assert (Y = D0)
            report "ERROR: S1S0=00 ⇒ expected Y=D0 (pattern 2)"
            severity error;

        -- S1 S0 = 01 ⇒ Y = D1
        S1 ⩽ '0'; S0 ⩽ '1'; wait for 10 ns;
        assert (Y = D1)
            report "ERROR: S1S0=01 ⇒ expected Y=D1 (pattern 2)"
            severity error;

        -- S1 S0 = 10 ⇒ Y = D2
        S1 ⩽ '1'; S0 ⩽ '0'; wait for 10 ns;
        assert (Y = D2)
            report "ERROR: S1S0=10 ⇒ expected Y=D2 (pattern 2)"
```

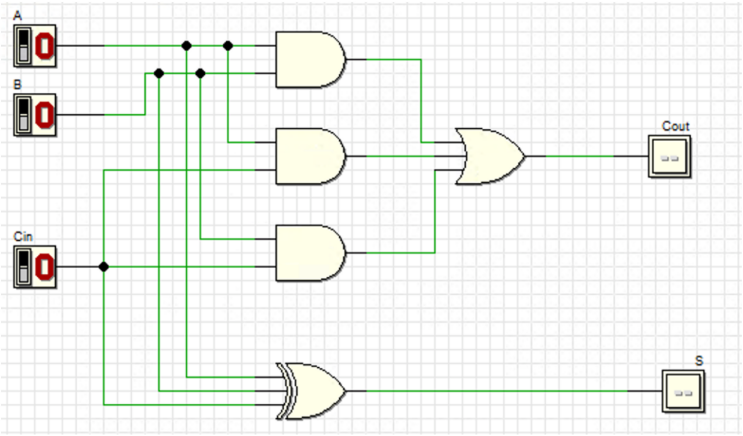
```
            severity error;

        -- S1 S0 = 11 ⇒ Y = D3
        S1 ⩽ '1'; S0 ⩽ '1'; wait for 10 ns;
        assert (Y = D3)
            report "ERROR: S1S0=11 ⇒ expected Y=D3 (pattern 2)"
            severity error;

        report "All Mux4 tests passed successfully!" severity note;
        wait;  -- stop simulation
    end process;

end architecture behavior;
```

5.3 8-bit Adder

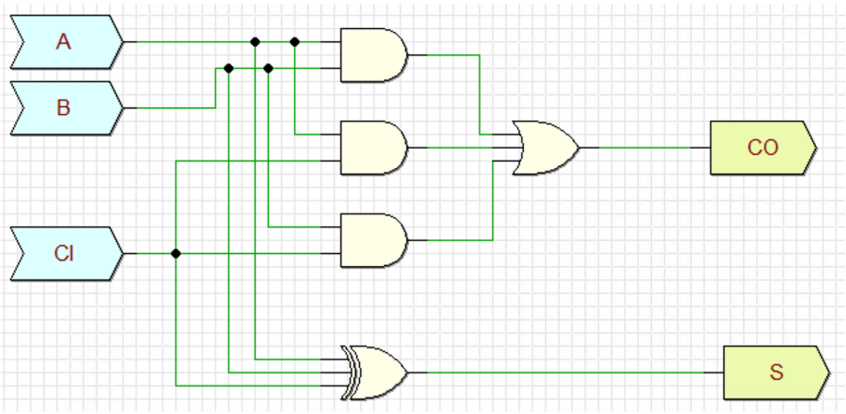
We need an 8-bit adder to support both addition and subtraction operations. First, we need to design a full adder— a circuit that adds two input bits and a carry-in bit, producing a sum bit and a carry-out bit.

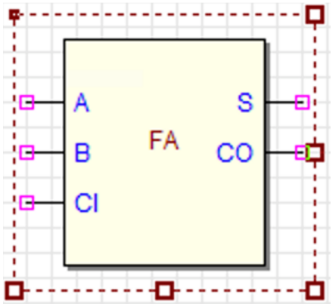


Design the full adder using DEEDS, simulate it, save it as a block diagram.



A	B	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1





Write the VHDL description of the full adder and include it in the Vivado project.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity FullAdder is
    port(
        A    : in  std_logic;
        B    : in  std_logic;
        Cin  : in  std_logic;
        Sum  : out std_logic;
        Cout : out std_logic
    );
end entity FullAdder;

architecture logic of FullAdder is
begin
    Sum  <= A xor B xor Cin;
```

```
    Cout <= (A and B) or (Cin and (A xor B));
end architecture logic;
```

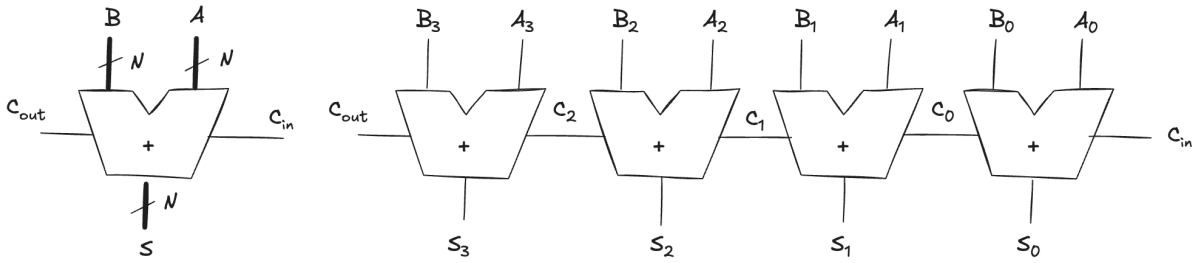
Write a testbench to verify the functionality of the full adder. Simulate the design and ensure that it behaves as expected for all possible input combinations.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity tb_FullAdder is
end entity;

architecture behavior of tb_FullAdder is

    -- Component declaration
    component FullAdder is
        port(
            A    : in  std_logic;
            B    : in  std_logic;
            Cin  : in  std_logic;
            Sum  : out std_logic;
            Cout : out std_logic
        );
    end component;

    signal A_tb, B_tb, Cin_tb : std_logic;
    signal Sum_tb, Cout_tb    : std_logic;

begin

    -- Instantiate the Unit Under Test
    UUT: FullAdder
        port map(
            A    => A_tb,
            B    => B_tb,
            Cin  => Cin_tb,
            Sum  => Sum_tb,
            Cout => Cout_tb
        );

    -- Stimulus process
    process
```

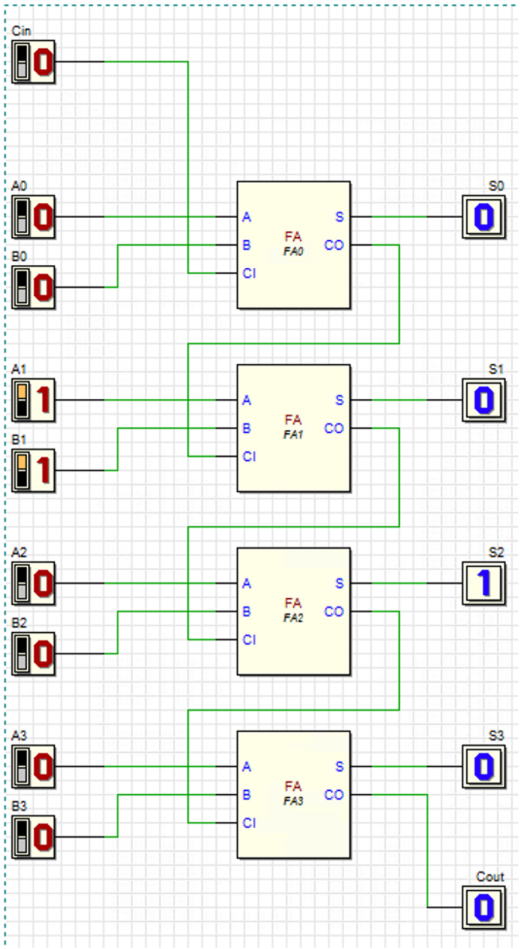
```
    begin

        -----------------------------------------------------------------
        -- Test 1: 0 + 0 + 0 = 0
        -----------------------------------------------------------------
        A_tb <= '0';  B_tb <= '0';  Cin_tb <= '0';
        wait for 10 ns;
        assert (Sum_tb = '0' and Cout_tb = '0')
            report "TEST 1 FAILED: 0 + 0 + 0 should be Sum=0 Cout=0" severity error;

        -----------------------------------------------------------------
        -- Test 2: 0 + 0 + 1 = 1
        -----------------------------------------------------------------
        A_tb <= '0';  B_tb <= '1';  Cin_tb <= '1';
        wait for 10 ns;
        assert (Sum_tb = '1' and Cout_tb = '0')
            report "TEST 2 FAILED: 0 + 0 + 1 should be Sum=1 Cout=0" severity error;

        -----------------------------------------------------------------
        -- Test 3: 0 + 1 + 0 = 1
        -----------------------------------------------------------------
        A_tb <= '0';  B_tb <= '1';  Cin_tb <= '0';
        wait for 10 ns;
        assert (Sum_tb = '1' and Cout_tb = '0')
            report "TEST 3 FAILED: 0 + 1 + 0 should be Sum=1 Cout=0" severity error;

        -----------------------------------------------------------------
        -- Test 4: 0 + 1 + 1 = 0 with carry
        -----------------------------------------------------------------
        A_tb <= '0';  B_tb <= '1';  Cin_tb <= '1';
        wait for 10 ns;
        assert (Sum_tb = '0' and Cout_tb = '1')
            report "TEST 4 FAILED: 0 + 1 + 1 should be Sum=0 Cout=1" severity error;

        -----------------------------------------------------------------
        -- Test 5: 1 + 0 + 0 = 1
        -----------------------------------------------------------------
        A_tb <= '1';  B_tb <= '0';  Cin_tb <= '0';
        wait for 10 ns;
        assert (Sum_tb = '1' and Cout_tb = '0')
            report "TEST 5 FAILED: 1 + 0 + 0 should be Sum=1 Cout=0" severity error;

        -----------------------------------------------------------------
```

```
        -- Test 6: 1 + 0 + 1 = 0 with carry
        ----------------------------------------------------------------------
        A_tb <= '1';  B_tb <= '0';  Cin_tb <= '1';
        wait for 10 ns;
        assert (Sum_tb = '0' and Cout_tb = '1')
            report "TEST 6 FAILED: 1 + 0 + 1 should be Sum=0 Cout=1" severity error;

        ----------------------------------------------------------------------
        -- Test 7: 1 + 1 + 0 = 0 with carry
        ----------------------------------------------------------------------
        A_tb <= '1';  B_tb <= '1';  Cin_tb <= '0';
        wait for 10 ns;
        assert (Sum_tb = '0' and Cout_tb = '1')
            report "TEST 7 FAILED: 1 + 1 + 0 should be Sum=0 Cout=1" severity error;

        ----------------------------------------------------------------------
        -- Test 8: 1 + 1 + 1 = 1 with carry
        ----------------------------------------------------------------------
        A_tb <= '1';  B_tb <= '1';  Cin_tb <= '1';
        wait for 10 ns;
        assert (Sum_tb = '1' and Cout_tb = '1')
            report "TEST 8 FAILED: 1 + 1 + 1 should be Sum=1 Cout=1" severity error;

        ----------------------------------------------------------------------
        report "FullAdder tests passed successfully!" severity note;

        wait;
    end process;

end architecture;
```

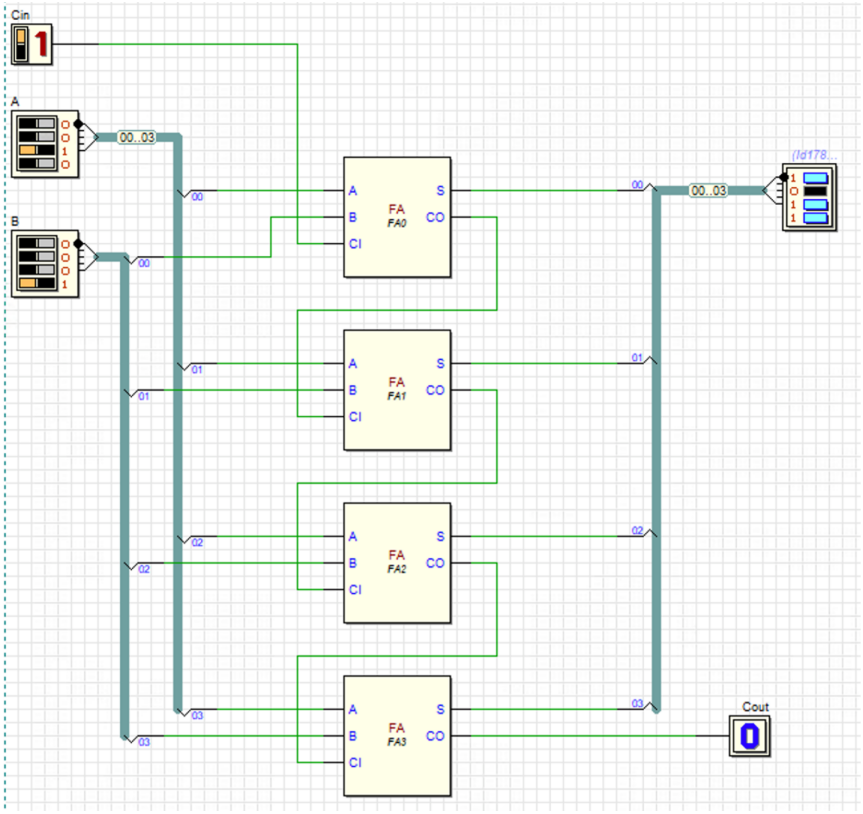
To support 8-bit addition, we must build an 8-bit adder. A straightforward approach is to use a ripple-carry structure, connecting eight full adders in series so that each carry-out feeds the next stage.



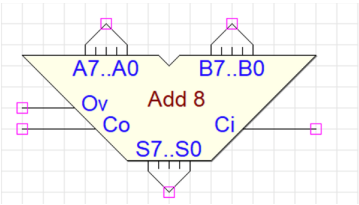
Using the full adder as a building block, design the 8-bit ripple-carry adder in DEEDS and simulate its behavior. As an intermediate step, begin by implementing a 4-bit version.



As the number of bits increases, using individual wires makes the schematic difficult to read. To keep the design clean and organized, redesign the 4-bit adder in DEEDS using bus connections.



Unfortunately, DEEDS does not allow creating new components that contain custom-made blocks. However, it already provides an adder component that we can use in our design. Search the DEEDS component library for an 8-bit adder and incorporate it into our design.



Write the VHDL description of the ripple-carry adder using the full adder you previously created, and add it to our Vivado project.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity RippleCarryAdder8 is
    port(
        A    : in  std_logic_vector(7 downto 0);
        B    : in  std_logic_vector(7 downto 0);
```

```
        Cin  : in  std_logic;
        S    : out std_logic_vector(7 downto 0);
        Cout : out std_logic
    );
end entity RippleCarryAdder8;

architecture structural of RippleCarryAdder8 is

    component FullAdder is
        port(
            A    : in  std_logic;
            B    : in  std_logic;
            Cin  : in  std_logic;
            Sum  : out std_logic;
            Cout : out std_logic
        );
    end component;

    signal C : std_logic_vector(7 downto 0);  -- internal carries

begin

    -- First bit (LSB) uses external Cin
    FA0: FullAdder port map(A(0), B(0), Cin, S(0), C(0));

    -- Bits 1 to 6
    FA1: FullAdder port map(A(1), B(1), C(0), S(1), C(1));
    FA2: FullAdder port map(A(2), B(2), C(1), S(2), C(2));
    FA3: FullAdder port map(A(3), B(3), C(2), S(3), C(3));
    FA4: FullAdder port map(A(4), B(4), C(3), S(4), C(4));
    FA5: FullAdder port map(A(5), B(5), C(4), S(5), C(5));
    FA6: FullAdder port map(A(6), B(6), C(5), S(6), C(6));

    -- Last bit (MSB)
    FA7: FullAdder port map(A(7), B(7), C(6), S(7), Cout);

end architecture structural;
```

Write a testbench to verify the functionality of the ripple-carry adder. Simulate the design and ensure that it behaves as expected for all possible input combinations.

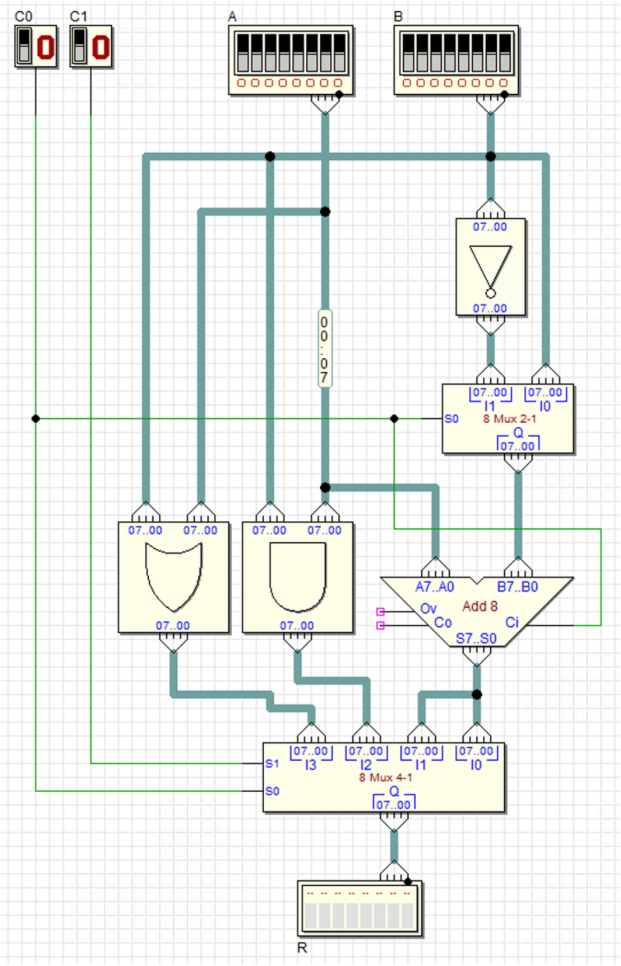
```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity tb_RippleCarryAdder8 is
end entity tb_RippleCarryAdder8;

architecture behavior of tb_RippleCarryAdder8 is

    component RippleCarryAdder8 is
        port(
            A    : in  std_logic_vector(7 downto 0);
            B    : in  std_logic_vector(7 downto 0);
            Cin  : in  std_logic;
            S    : out std_logic_vector(7 downto 0);
            Cout : out std_logic
        );
    end component;

    signal A_tb, B_tb : std_logic_vector(7 downto 0);
    signal Cin_tb     : std_logic;
    signal S_tb       : std_logic_vector(7 downto 0);
    signal Cout_tb    : std_logic;

begin

    UUT: RippleCarryAdder8
        port map(
            A    => A_tb,
            B    => B_tb,
            Cin  => Cin_tb,
            S    => S_tb,
            Cout => Cout_tb
        );

    ----------------------------------------------------------------------
    -- Testbench
    ----------------------------------------------------------------------
    process
        variable A_v, B_v : unsigned(7 downto 0);
        variable Cin_v    : unsigned(0 downto 0);
        variable expected : unsigned(8 downto 0);
    begin
```

```
        ------------------------------------------------------------------
        -- TEST 1
        ------------------------------------------------------------------
        A_v   := x"0A";
        B_v   := x"05";
        Cin_v := "0";

        A_tb   <= std_logic_vector(A_v);
        B_tb   <= std_logic_vector(B_v);
        Cin_tb <= Cin_v(0);

        wait for 10 ns;

        expected := ('0' & A_v) + ('0' & B_v) + Cin_v;

        assert unsigned(S_tb) = expected(7 downto 0)
            report "ERROR Test 1: wrong SUM" severity error;
        assert Cout_tb = expected(8)
            report "ERROR Test 1: wrong CARRY" severity error;

        ------------------------------------------------------------------
        -- TEST 2
        ------------------------------------------------------------------
        A_v   := x"10";
        B_v   := x"0F";
        Cin_v := "1";

        A_tb   <= std_logic_vector(A_v);
        B_tb   <= std_logic_vector(B_v);
        Cin_tb <= Cin_v(0);

        wait for 10 ns;

        expected := ('0' & A_v) + ('0' & B_v) + Cin_v;

        assert unsigned(S_tb) = expected(7 downto 0)
            report "ERROR Test 2: wrong SUM" severity error;
        assert Cout_tb = expected(8)
            report "ERROR Test 2: wrong CARRY" severity error;

        ------------------------------------------------------------------
        -- TEST 3
```

```
        ----------------------------------------------------------------
        A_v   := x"FF";
        B_v   := x"01";
        Cin_v := "0";

        A_tb   <= std_logic_vector(A_v);
        B_tb   <= std_logic_vector(B_v);
        Cin_tb <= Cin_v(0);

        wait for 10 ns;

        expected := ('0' & A_v) + ('0' & B_v) + Cin_v;

        assert unsigned(S_tb) = expected(7 downto 0)
            report "ERROR Test 3: wrong SUM" severity error;
        assert Cout_tb = expected(8)
            report "ERROR Test 3: wrong CARRY" severity error;

        ----------------------------------------------------------------
        report "RippleCarryAdder tests passed successfully" severity note;
        wait;

    end process;

end architecture behavior;
```

5.4 Putting everything together

Using the DEEDS adder, multiplexers, and the AND/OR gates, build the complete ALU design. Simulate the ALU in DEEDS to verify that each operation behaves correctly based on the control signals.



Using the VHDL enetities created in the previous steps, write the VHDL description of the complete ALU and add it to our Vivado project.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity ALU8 is
    port(
        A    : in  std_logic_vector(7 downto 0);
        B    : in  std_logic_vector(7 downto 0);
        C1   : in  std_logic;                    -- control bit MSB
        C0   : in  std_logic;                    -- control bit LSB
        R    : out std_logic_vector(7 downto 0); -- ALU result
        C    : out std_logic                     -- final carry-out
    );
end entity ALU8;
```

```
architecture structural of ALU8 is

    ---------------------------------------------------
    -- Components
    ---------------------------------------------------
    component RippleCarryAdder8 is
        port(
            A    : in  std_logic_vector(7 downto 0);
            B    : in  std_logic_vector(7 downto 0);
            Cin  : in  std_logic;
            S    : out std_logic_vector(7 downto 0);
            Cout : out std_logic
        );
    end component;

    component Mux2 is
        port(
            D0 : in  std_logic_vector(7 downto 0);
            D1 : in  std_logic_vector(7 downto 0);
            S  : in  std_logic;
            Y  : out std_logic_vector(7 downto 0)
        );
    end component;

    component Mux4 is
        port(
            D0 : in std_logic_vector(7 downto 0);
            D1 : in std_logic_vector(7 downto 0);
            D2 : in std_logic_vector(7 downto 0);
            D3 : in std_logic_vector(7 downto 0);
            S0 : in std_logic;
            S1 : in std_logic;
            Y  : out std_logic_vector(7 downto 0)
        );
    end component;

    ---------------------------------------------------
    -- Internal signals
    ---------------------------------------------------
    signal B_not   : std_logic_vector(7 downto 0);
    signal B_sel   : std_logic_vector(7 downto 0);
    signal SumRes  : std_logic_vector(7 downto 0);
```

```
    signal Cout    : std_logic;

    signal AndRes  : std_logic_vector(7 downto 0);
    signal OrRes   : std_logic_vector(7 downto 0);

begin

    ----------------------------------------------------
    -- Generate ~B (bitwise NOT of B)
    ----------------------------------------------------
    B_not <= not B;

    ----------------------------------------------------
    -- Mux2 selects B or ~B
    -- C0 = 0 → ADD mode: B_sel = B
    -- C0 = 1 → SUB mode: B_sel = ~B
    ----------------------------------------------------
    B_SELECTOR: Mux2
        port map(
            D0 => B,
            D1 => B_not,
            S  => C0,
            Y  => B_sel
        );

    ----------------------------------------------------
    -- Logical operations
    ----------------------------------------------------
    AndRes <= A and B;
    OrRes  <= A or  B;

    ----------------------------------------------------
    -- Ripple-carry adder:
    -- Cin = C0  (0 for ADD, 1 for SUB)
    ----------------------------------------------------
    ADDER: RippleCarryAdder8
        port map(
            A    => A,
            B    => B_sel,
            Cin  => C0,
            S    => SumRes,
            Cout => Cout
        );
```

```
    ---------------------------------------------------
    -- Mux4 selects the ALU output
    -- C1C0 = 00 → AND
    -- C1C0 = 01 → OR
    -- C1C0 = 10 → ADD
    -- C1C0 = 11 → SUB
    ---------------------------------------------------
    OUTPUT_SELECTOR: Mux4
        port map(
            D0 ⇒ AndRes,   -- 00
            D1 ⇒ OrRes,    -- 01
            D2 ⇒ SumRes,   -- 10 (ADD)
            D3 ⇒ SumRes,   -- 11 (SUB)
            S0 ⇒ C0,
            S1 ⇒ C1,
            Y  ⇒ R
        );

    ---------------------------------------------------
    -- Carry output (only meaningful for ADD/SUB)
    ---------------------------------------------------
    C ⩽ Cout;

end architecture structural;
```

Write a testbench to verify the funtionality of the complete ALU. Simulate the design and ensure that it behaves as expected for all possible input combinations and control signal configurations.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity tb_ALU8 is
end entity;

architecture behavior of tb_ALU8 is

    -- Component under test
    component ALU8 is
        port(
```

```
            A    : in  std_logic_vector(7 downto 0);
            B    : in  std_logic_vector(7 downto 0);
            C1   : in  std_logic;
            C0   : in  std_logic;
            R    : out std_logic_vector(7 downto 0);
            C    : out std_logic
        );
    end component;

    -- Signals
    signal A_tb, B_tb : std_logic_vector(7 downto 0);
    signal C1_tb, C0_tb : std_logic;
    signal R_tb : std_logic_vector(7 downto 0);
    signal C_tb : std_logic;

begin

    -- Instantiate the ALU
    UUT: ALU8
        port map(
            A => A_tb,
            B => B_tb,
            C1 => C1_tb,
            C0 => C0_tb,
            R => R_tb,
            C => C_tb
        );

    process
    begin

        ----------------------------------------------------------------------
        -- Test 1: AND
        ----------------------------------------------------------------------
        A_tb <= x"0F";        -- 0000 1111
        B_tb <= x"33";        -- 0011 0011
        C1_tb <= '0'; C0_tb <= '0';  -- AND
        wait for 10 ns;

        assert R_tb = (A_tb and B_tb)
            report "AND FAILED" severity error;

        ----------------------------------------------------------------------
```

```
        -- Test 2: OR
        ------------------------------------------------------------------
        C1_tb <= '0'; C0_tb <= '1';  -- OR
        wait for 10 ns;

        assert R_tb = (A_tb or B_tb)
            report "OR FAILED" severity error;

        ------------------------------------------------------------------
        -- Test 3: ADD
        ------------------------------------------------------------------
        A_tb <= x"10";        -- 16
        B_tb <= x"05";        --  5
        C1_tb <= '1'; C0_tb <= '0';  -- ADD
        wait for 10 ns;

        assert unsigned(R_tb) = unsigned(A_tb) + unsigned(B_tb)
            report "ADD FAILED" severity error;

        ------------------------------------------------------------------
        -- Test 4: SUB
        ------------------------------------------------------------------
        A_tb <= x"20";       -- 32
        B_tb <= x"10";       -- 16
        C1_tb <= '1'; C0_tb <= '1';  -- SUB
        wait for 10 ns;

        assert unsigned(R_tb) = unsigned(A_tb) - unsigned(B_tb)
            report "SUB FAILED" severity error;

        ------------------------------------------------------------------
        report "ALU tests passed!" severity note;
        wait;

    end process;

end architecture;
```

5.5 Next Steps

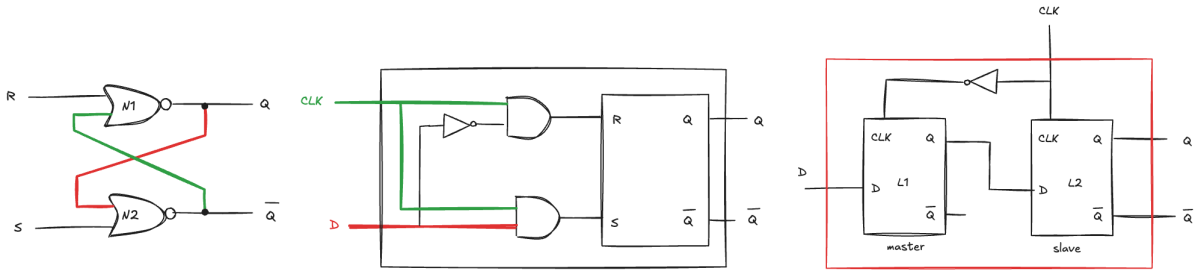
Explore additional operations to enhance the ALU's capabilities, such as adding flag outputs (zero, carry, overflow) or other bitwise operations like XOR and NAND. Another improvement

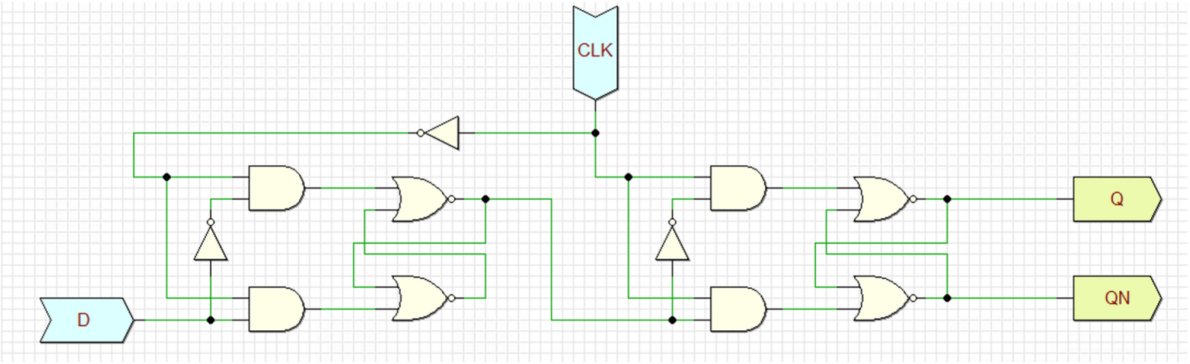
could be extend the ALU to more than 8 bits, such as 16 or 32 bits, to handle larger data sizes. In that case, consider using more advanced adder architectures, like carry-lookahead adders, to improve performance.

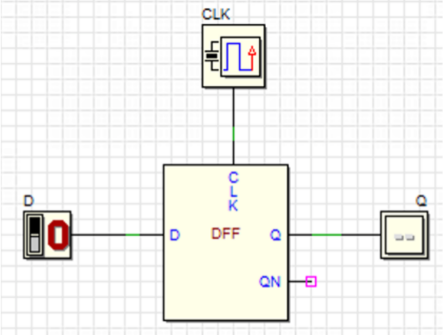
6 Latch and Flip-Flop Exercises

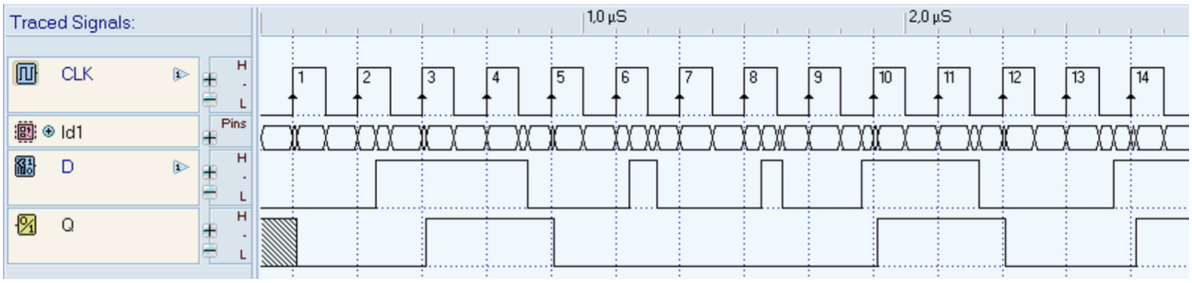
6.1 Exercise 1

Implement a D flip-flop in DEEDS and simulate its behavior using a timing diagram that shows it is transparent only at the rising edge of the clock and that, at all other times, it retains its previously stored value. Then use this circuit to create a reusable block that can be incorporated into other designs:







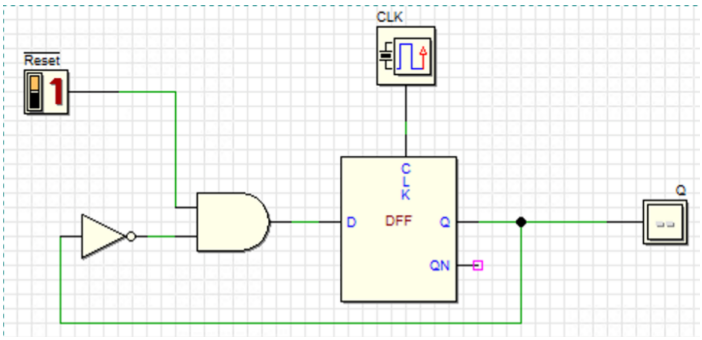


6.2 Exercise 2

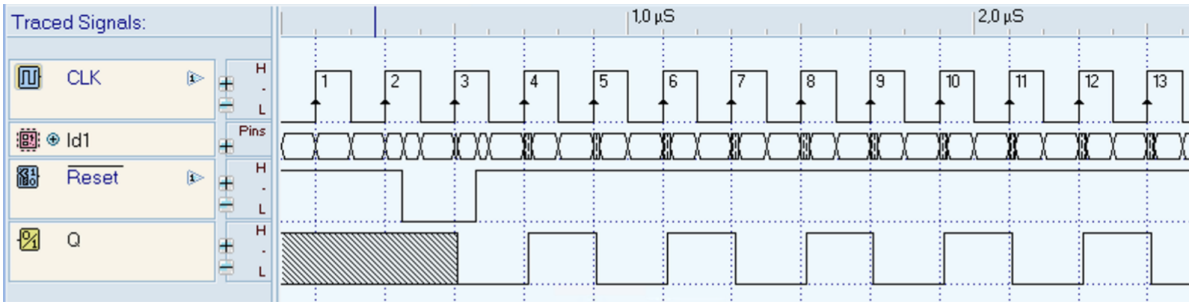
Consider the following behavioral descriptions of different types of flip-flops. For each type, design a possible implementation, build it in DEEDS, verify its operation using a timing diagram and write the VHDL code that describes its behavior.

6.2.1 (a) Toggle Flip-Flop

The toggle (T) flip-flop inverts its output on every active clock edge when T is high, producing a clean alternating sequence that effectively divides the clock frequency by two.



The T flip-flop needs a reset because its behavior depends entirely on its previous stored value; without initializing Q to a known state at power-up, the flip-flop begins in an undefined condition and produces unpredictable output sequences.

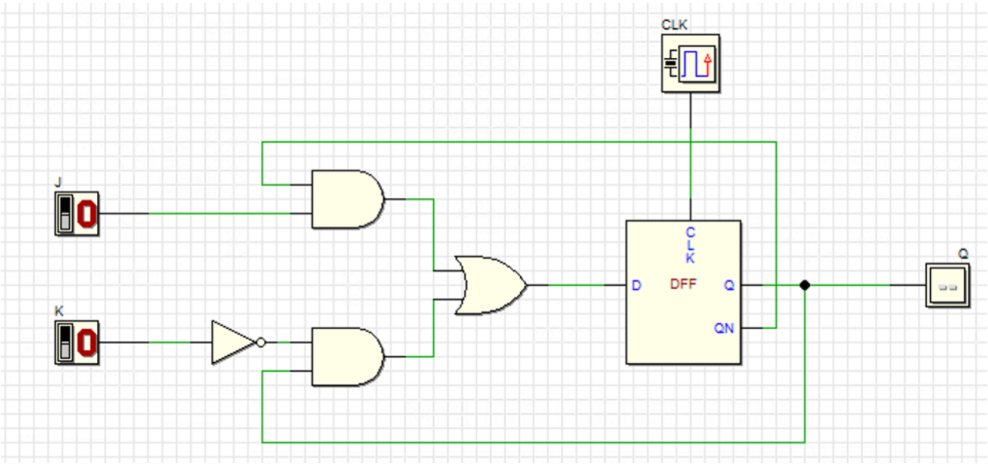



```vhdl
library ieee;
use ieee.std_logic.ALL;

entity TFF is
    Port (
        CLK   : in  std_logic;
        Reset : in  std_logic;
        Q     : out std_logic;
        QN    : out std_logic
    );
end TFF;

architecture Behavioral of TFF is
    signal D_internal : std_logic;
    signal Q_internal : std_logic;
    signal Reset_inv  : std_logic;
begin
    -- Inverter for Reset signal
    Reset_inv <= not Reset;

    -- AND gate combining inverted Reset with feedback from Q
    D_internal <= Reset_inv and Q_internal;

    -- D Flip-Flop process
    process(CLK, Reset)
    begin
        if Reset = '1' then
            -- Asynchronous reset
            Q_internal <= '0';
        elsif rising_edge(CLK) then
            -- On clock rising edge, transfer D to Q
            Q_internal <= D_internal;
        end if;
```

```
    end process;

    -- Output assignments
    Q  <= Q_internal;
    QN <= not Q_internal;

end Behavioral;
```

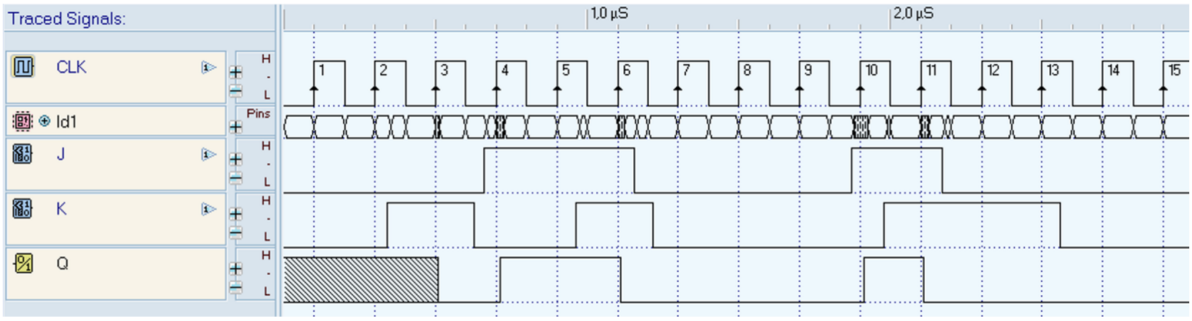
6.2.2 (b) JK Flip-Flop

A JK flip-flop receives a clock and two inputs, J and K. It updates its output on each clock edge by setting Q when J=1 and K=0, resetting it when J=0 and K=1, holding its value when both inputs are 0, and toggling when J and K are both 1, making it a versatile device that can act as a set/reset flip-flop, a memory cell, or a toggle depending on its inputs.



J	K	Clk	Q	$\overline{Q}$	
0	0	↑	Q_p	$\overline{Q}_p$	Previous state
1	0	↑	1	0	SET command
0	1	↑	0	1	RESET command
1	1	↑	$\overline{Q}_p$	Q_p	Toggle

A JK flip-flop can act as an SR flip-flop by using J and K as set and reset inputs, as a D flip-flop by driving J with D and K with the complement of D, and as a T flip-flop by tying J and K together so that a high input makes it toggle




```vhdl
library ieee;
use ieee.std_logic.ALL;

entity jk_flipflop is
    Port (
        J   : in  std_logic;
        K   : in  std_logic;
        CLK : in  std_logic;
        Q   : out std_logic;
        QN  : out std_logic
    );
end jk_flipflop;

architecture Behavioral of jk_flipflop is
    signal Q_internal  : std_logic := '0';
    signal QN_internal : std_logic := '1';
    signal D_input     : std_logic;
    signal K_inv       : std_logic;
    signal and1_out    : std_logic;
    signal and2_out    : std_logic;
begin
    -- Inverter for K input
    K_inv <= not K;

    -- Upper AND gate: J AND QN
    and1_out <= J and QN_internal;

    -- Lower AND gate: K_inv AND Q
    and2_out <= K_inv and Q_internal;

    -- OR gate to generate D input
    D_input <= and1_out or and2_out;
```

```
    -- D Flip-Flop process
    process(CLK)
    begin
        if rising_edge(CLK) then
            Q_internal <= D_input;
            QN_internal <= not D_input;
        end if;
    end process;

    -- Output assignments
    Q  <= Q_internal;
    QN <= QN_internal;

end Behavioral;
```

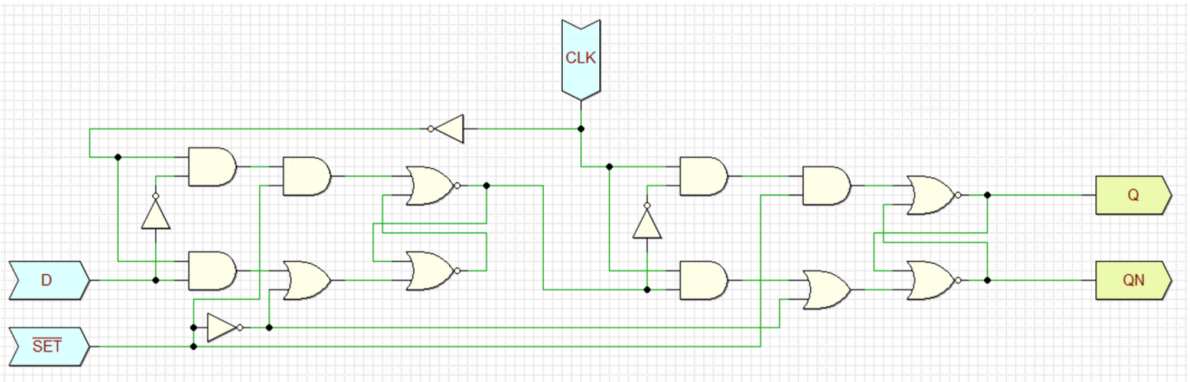
Alternative implementation using JK flip-flop behavior directly:

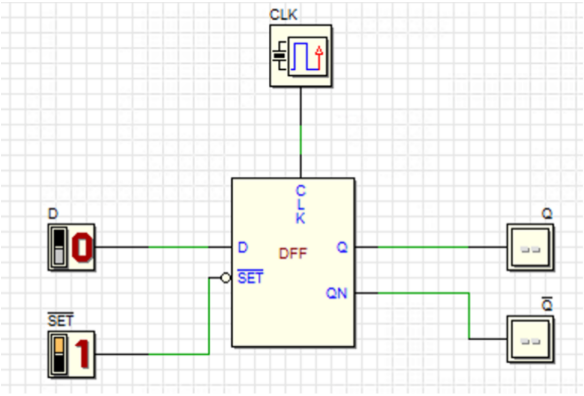
```
library ieee;
use ieee.STD_LOGIC_1164.ALL;

entity jk_flipflop_behavioral is
    Port (
        J   : in  std_logic;
        K   : in  std_logic;
        CLK : in  std_logic;
        Q   : out std_logic;
        QN  : out std_logic
    );
end jk_flipflop_behavioral;

architecture Behavioral of jk_flipflop_behavioral is
    signal Q_internal : std_logic := '0';
begin
    process(CLK)
    begin
        if rising_edge(CLK) then
            case (J & K) is
                when "00" => Q_internal <= Q_internal;     -- No change
                when "01" => Q_internal <= '0';           -- Reset
                when "10" => Q_internal <= '1';           -- Set
                when "11" => Q_internal <= not Q_internal; -- Toggle
                when others => Q_internal <= Q_internal;
```

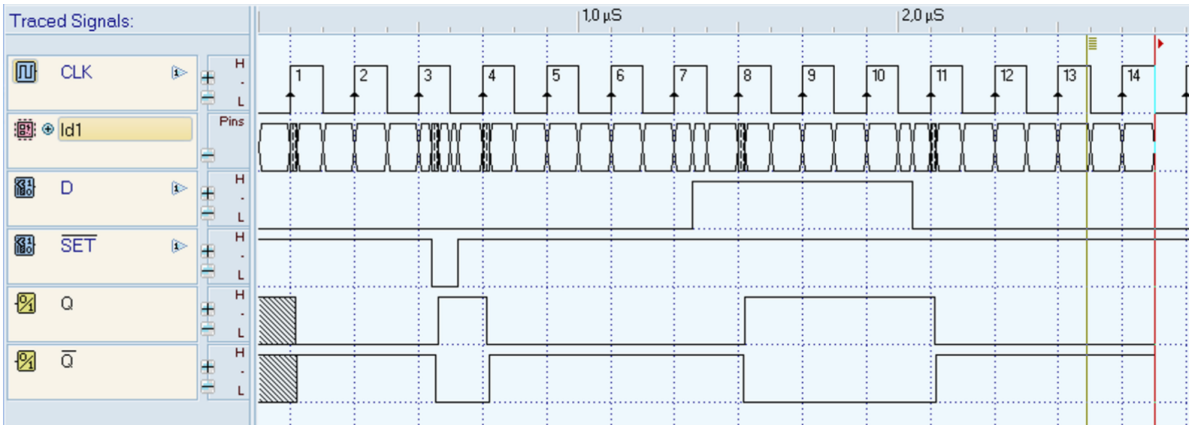
```
            end case;
        end if;
    end process;

    -- Output assignments
    Q  <= Q_internal;
    QN <= not Q_internal;

end Behavioral;
```

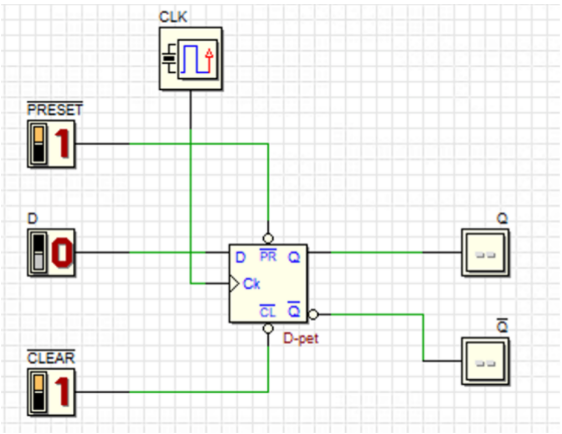
6.2.3 (c) Asynchronous Preset and Clear Flip-Flop

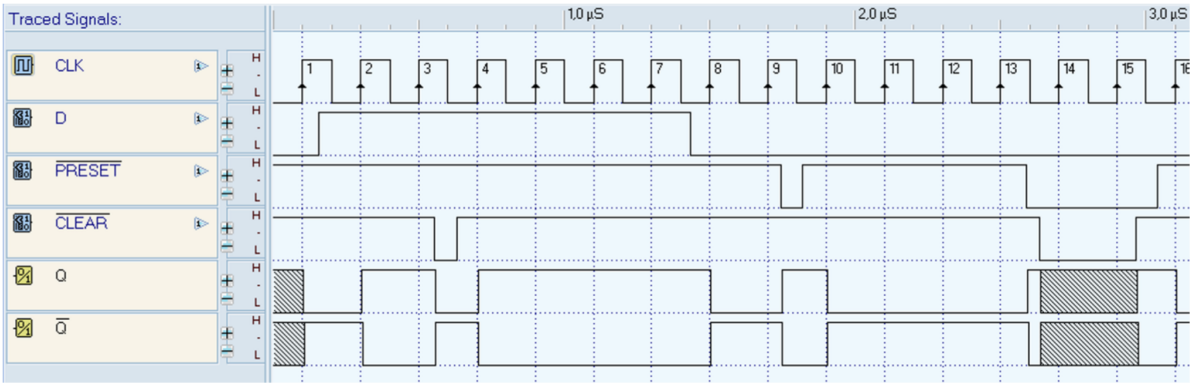
We already considered a reset input for flip-flops, which allows us to force the output Q to 0 regardless of the data inputs. However, we considered only a **synchronous reset**, which acts only on the active clock edge. In many applications, it is useful to have asynchronous inputs that can override the normal operation of the flip-flop at any time, without waiting for a clock event. Preset and Clear are **asynchronous inputs** that let us immediately force a flip-flop into a known state (Preset driving the output high and Clear driving it low), so the circuit can **start up correctly**, **recover from faults**, or be placed instantly in a controlled condition without waiting for a clock edge. Modify the D flip-flop block in order to add a Set inputs, and simulate its operation with a timing diagram that shows how this input override the normal D and clock behavior. Then try the DEEDS library flip-flop with preset and clear inputs and compare its behavior with your design.




CLK
C
L
K
D
DFF
Q
SET
QN
Q
SET


Traced Signals:
1,0 µS
2,0 µS
CLK
Id1
Pins
D
SET
Q
Q
1
2
3
4
5
6
7
8
9
10
11
12
13
14


CLK
PRESET
D
D
PR
Q
Ck
CL
Q
D-pet
Q
CLEAR



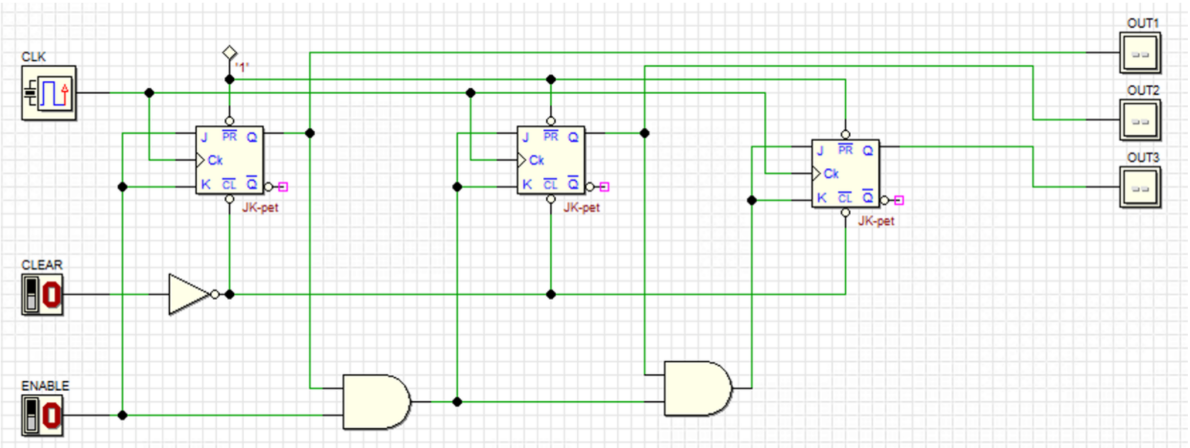

```
library ieee;
use ieee.std_logic_1164.all;

entity d_flipflop_clear_preset is
    Port (
        D       : in  std_logic;
        CLK     : in  std_logic;
        CLR_n   : in  std_logic;  -- Active-low Clear (asynchronous)
        PRE_n   : in  std_logic;  -- Active-low Preset (asynchronous)
        Q       : out std_logic;
        QN      : out std_logic
    );
end d_flipflop_clear_preset;

architecture Behavioral of d_flipflop_clear_preset is
    signal Q_internal : STD_LOGIC;
begin
    process(CLK, CLR_n, PRE_n)
    begin
        -- Check for illegal condition (both Clear and Preset active)
        if (CLR_n = '0' and PRE_n = '0') then
            -- Illegal state - typically set both Q and QN to '1' or handle as needed
            Q_internal <= '1';  -- Or could be 'X' for unknown

        elsif CLR_n = '0' then
            -- Asynchronous Clear (active-low) - has priority after illegal check
            Q_internal <= '0';

        elsif PRE_n = '0' then
            -- Asynchronous Preset (active-low)
            Q_internal <= '1';
```

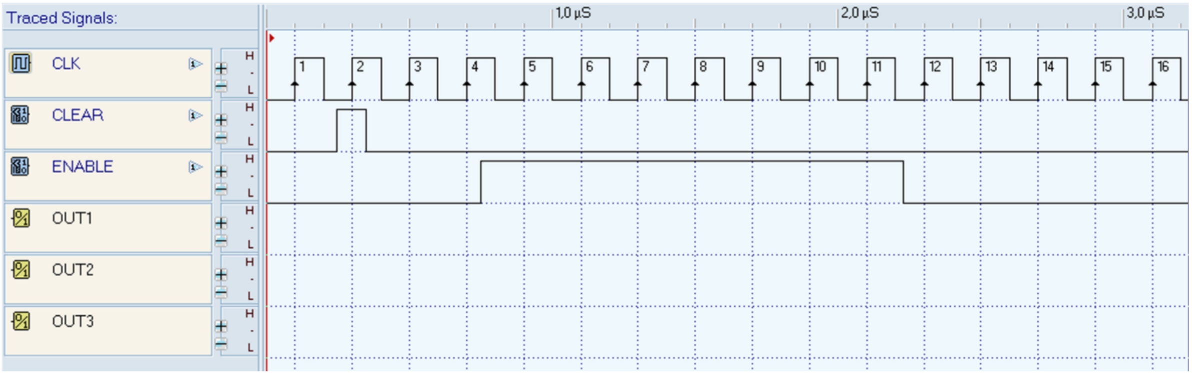
```
        elsif rising_edge(CLK) then
            -- Normal D flip-flop operation on clock rising edge
            Q_internal <= D;
        end if;
    end process;

    -- Output assignments
    Q  <= Q_internal;
    QN <= not Q_internal;

end Behavioral;
```

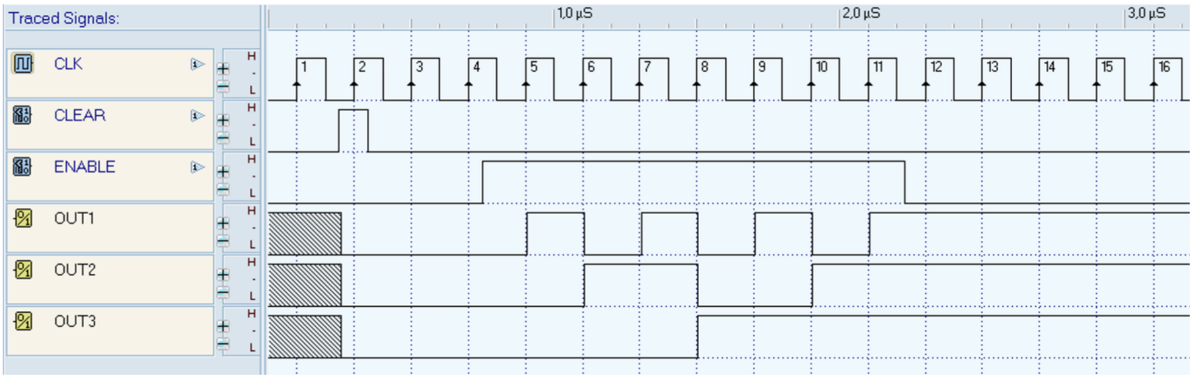
6.3 Exercise 3

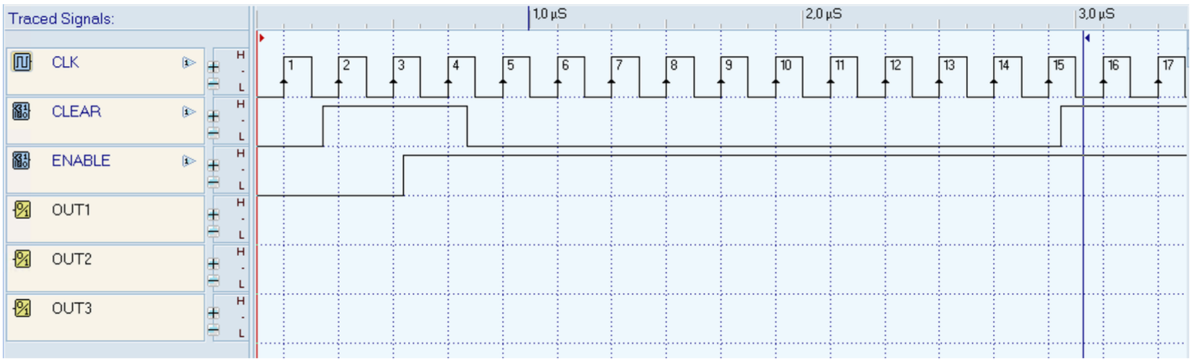
Analyze the following synchronous sequential circuits by completing the timing diagrams. First draw the waveforms by hand and then use DEEDS only to verify your solutions.

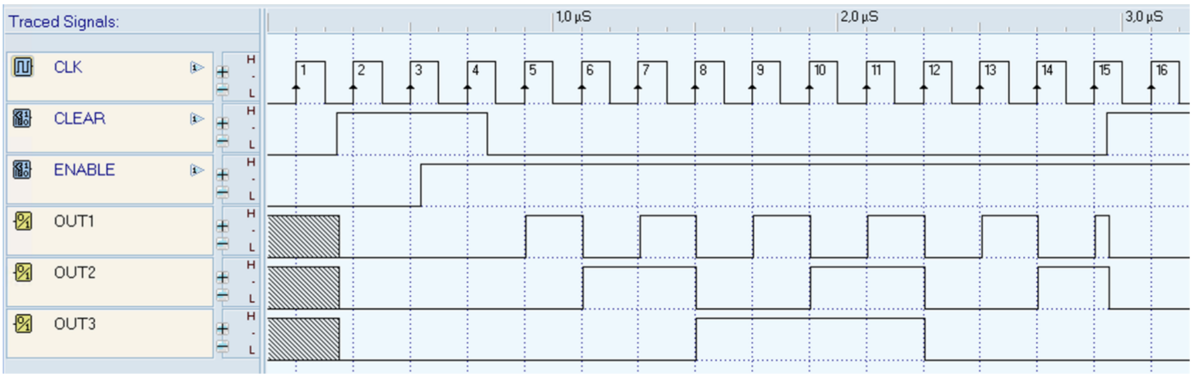
6.3.1 (a)



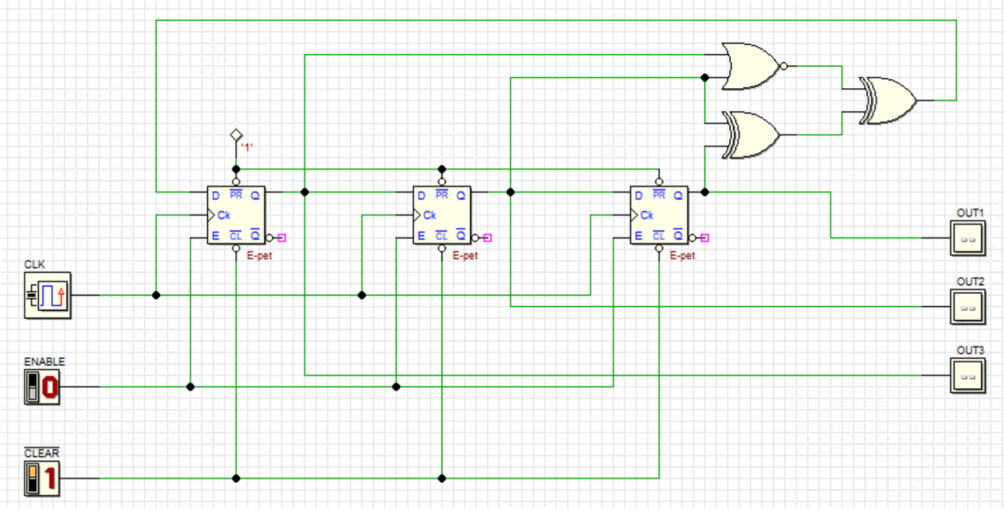

Traced Signals:
CLK
CLEAR
ENABLE
OUT1
OUT2
OUT3
1,0 μS
2,0 μS
3,0 μS

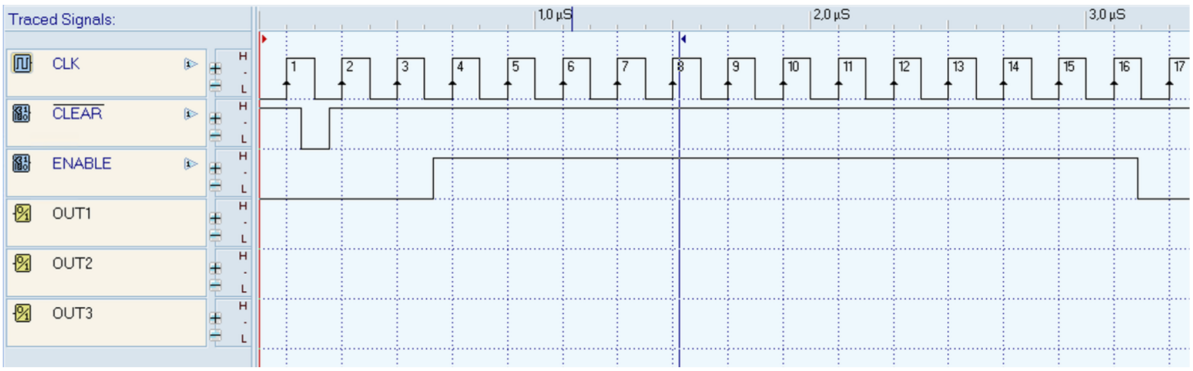

Traced Signals:
CLK
CLEAR
ENABLE
OUT1
OUT2
OUT3
1,0 μS
2,0 μS
3,0 μS

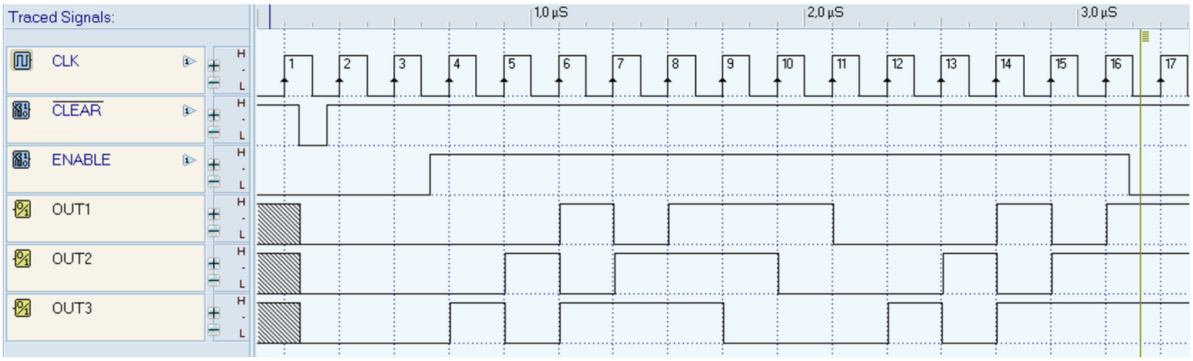

Traced Signals:
CLK
CLEAR
ENABLE
OUT1
OUT2
OUT3
1,0 μS
2,0 μS
3,0 μS

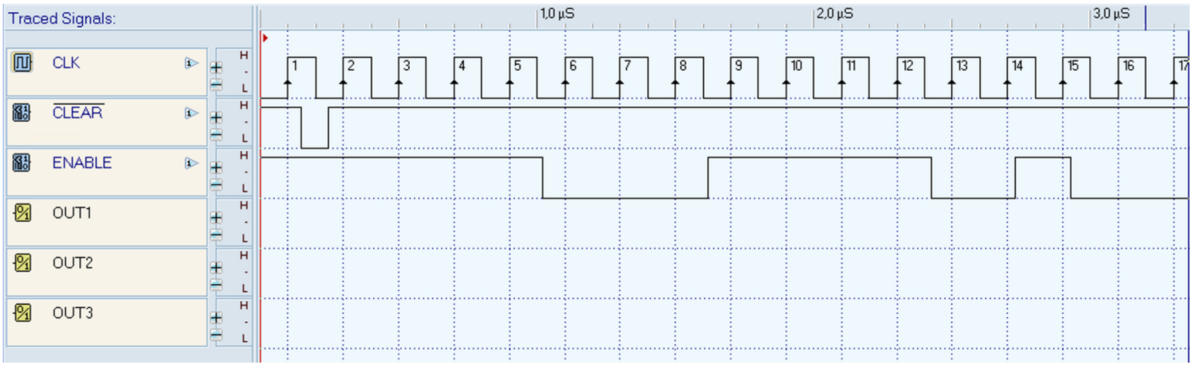

Traced Signals:
CLK
CLEAR
ENABLE
OUT1
OUT2
OUT3
1,0 μS
2,0 μS
3,0 μS

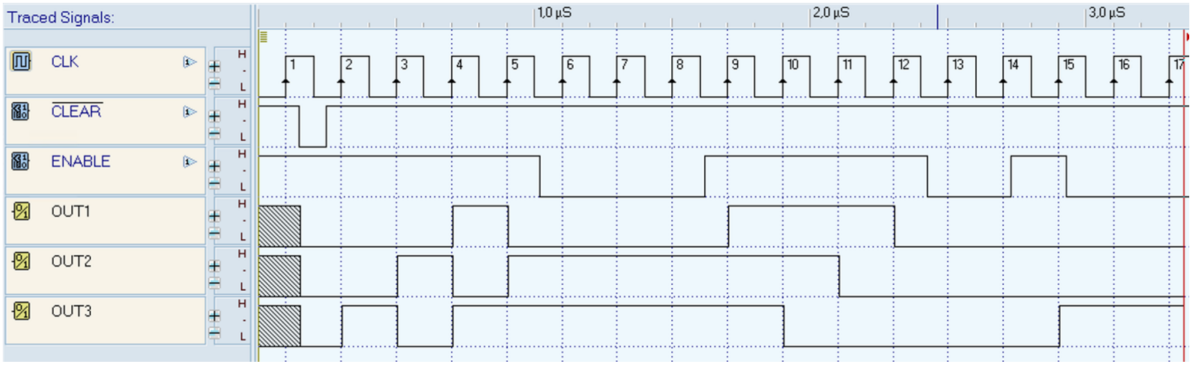
6.3.2 (b)


CLK
ENABLE
CLEAR
OUT1
OUT2
OUT3
E-pet
E-pet
E-pet

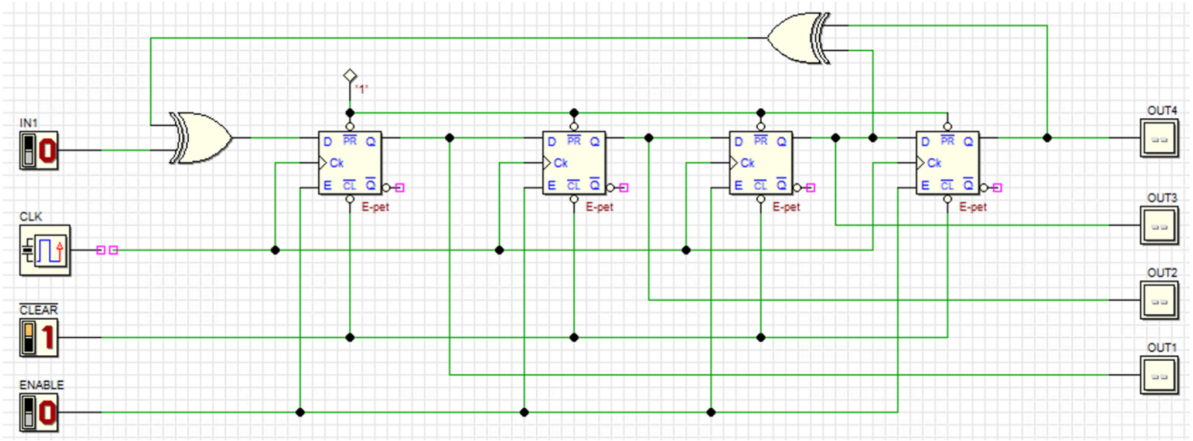

Traced Signals:
CLK
CLEAR
ENABLE
OUT1
OUT2
OUT3
1,0 µS
2,0 µS
3,0 µS

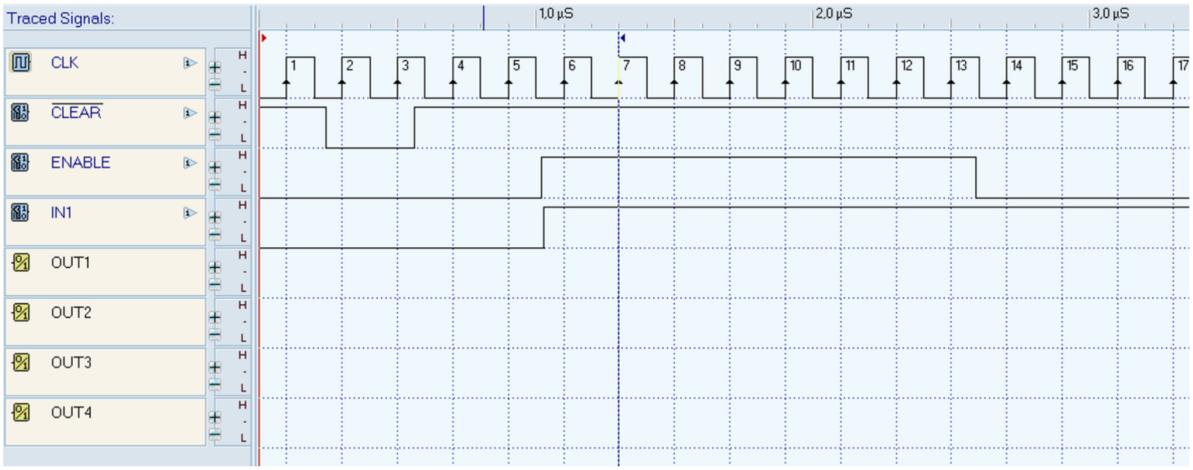

Traced Signals:
CLK
CLEAR
ENABLE
OUT1
OUT2
OUT3
1,0 µS
2,0 µS
3,0 µS

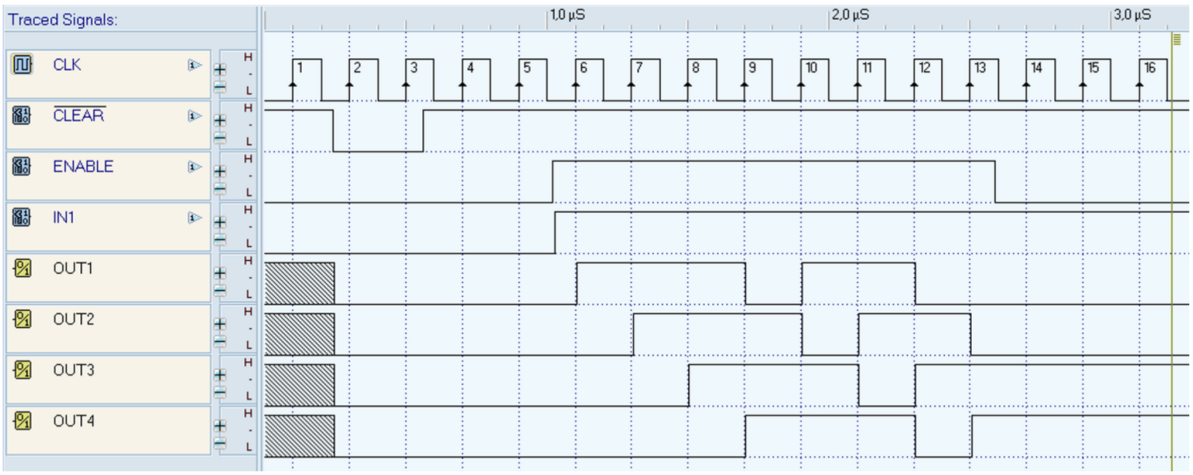

Traced Signals:
1,0 µS
2,0 µS
3,0 µS
CLK
CLEAR
ENABLE
OUT1
OUT2
OUT3


Traced Signals:
1,0 µS
2,0 µS
3,0 µS
CLK
CLEAR
ENABLE
OUT1
OUT2
OUT3

6.3.3 (c)


IN1
CLK
CLEAR
ENABLE
OUT4
OUT3
OUT2
OUT1
E-pet



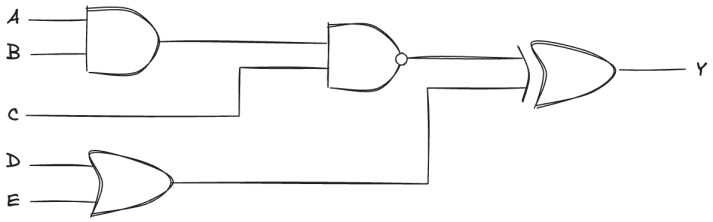


7 Timing Exercises

7.1 Exercise 1

In the circuits shown below, determine both the propagation delay and the contamination delay. Assume that every gate has a propagation delay of 100 ps and a contamination delay of 60 ps.

7.1.1 (a)



7.1.2 (b)