Exercise Book

CS-173 Fundamentals of Digital Systems

26th May 2025

Mirjana Stojilović
Parallel Systems Architecture Lab (PARSA)

Preface

This exercise book results from a team effort by the CS-173 course staff and many contributors who helped create new problems and improve existing ones. We gratefully acknowledge everyone who participated in shaping this resource.

The exercise book is a supplementary aid to help you understand the course material. The official lecture slides and course exercises remain the primary references for exams. We appreciate your feedback if you spot any issues or inconsistencies—it helps us continue refining this resource.

—The CS-173 Course Team

Part I: Number Systems

[Exercise 1] Basics of Different Positional Number Formats

a) Given n bits, how many different numbers can be represented?

b) Consider the given ranges below. For each range, if we want n-bit integers to be able to represent all the numbers in the range, what is the lowest value of n? **i)** $[0, 256)$ **ii)** $[-1024, 1024)$ **iii)** $[3325, 3581)$ **iv)** $[-657, 879)$

c) In general, how many bits are required to represent integers in the range of $[a, b)$, where a and b are integers, and $b \geq a$?

d) Fill the missing values in Table 1 by converting between the different listed formats. (Note: Consider all formats to be unsigned)

Decimal	Binary	Octal	Hexadecimal
139			
185			
	00111010		
	11001001		
		70	
		323	
			$1B$
			$6C$

Table 1: Conversion between decimal, binary and hexadecimal formats.

[Solution 1] Basics of Different Positional Number Formats

a) One bit can represent two different numbers. Therefore, n bits can represent 2^n different numbers.

b) Follow the formula outlined in the solution to question (b)
i) $[0, 256) = 8$ **ii)** $[-1024, 1024) = 11$ **iii)** $[3325, 3581) = 8$ **iv)** $[-657, 879) = 11$

c) There are a total of $(b - a)$ unique integers in the range $[a, b)$. To represent these unique integers, we need $\lceil \log_2(b - a) \rceil$ bits.

d) Rules for coversion:

Decimal to binary:	Divide the decimal number by 2 and record the remainder.
	Continue dividing the quotient by 2 until the quotient is 0.
	The binary number is the sequence of remainders in reverse order.
Binary to decimal:	Multiply each bit by 2^i, where i is the position of the bit.
	Add the products to get the decimal number.

The other number formats (Hex, Octal) are similar, follow the same steps as outlined above but replace the number 2 with 16 for Hex and 8 for Octal.

Decimal	Binary	Octal	Hexadecimal
139	10001011	213	$8B$
185	10111001	271	$B9$
58	00111010	72	$3A$
201	11001001	311	$C9$
56	00111000	70	38
211	11010011	323	$D3$
27	00011011	33	$1B$
108	01101100	154	$6C$

[Exercise 2] Signed and Unsigned Integers

a) Given n bits, what is the range of signed (two's complement) and unsigned integers that can be represented? In which scenario, would you prefer unsigned over signed?

b) Describe how overflow is handled differently for unsigned and two's complement signed integers.

c) Convert the following decimal numbers in Table 2 to 8-bit sign-magnitude and two's complement binary format.

Decimal	Sign-magnitude	Two's complement
-32		
73		
-98		
47		
-39		
86		

Table 2: Conversion of decimal numbers to signed binary formats.

d) Table 3 contains 4-bit binary numbers. Extend them to 8 bits and convert the 8-bit binary numbers to the different listed formats.

4-bit Sign-magnitude	8-bit Sign-magnitude	Hexadecimal	Decimal	8-bit Two's complement
0110				
1100				
1111				
0001				
4-bit Two's complement	8-bit Two's complement	Hexadecimal	Decimal	8-bit Sign-magnitude
1010				
0101				
1100				
1000				

Table 3: Sign extension and conversion between different formats.

e) List the advantages of two's complement over sign-magnitude form in terms of:
i) Difference in the way 0 is represented
ii) Algorithm for addition and subtraction
iii) Numerical range and handling overflows

[Solution 2] Signed and Unsigned Integers

a) Signed (Two's complement): $[-2^{n-1}, 2^{n-1})$, Unsigned: $[0, 2^n)$.
We would prefer to use unsigned integers when we know that the numbers will always be positive because it allows us to represent $2\times$ more positive numbers than signed integers.

b) For signed integers in two's complement, overflow results in the number being rolled over from the maximum representable positive number to the minimum representable negative number, and vice versa.
For unsigned integers, overflow results in the number being rolled over from the maximum representable number to 0, and vice versa.

c) For sign magnitude, convert the magnitude to 7-bit binary, and the MSB is 0 for positive and 1 for negative. For two's complement form, use the formula: $-x = \bar{x} + 1$.

Decimal	Sign-magnitude	Two's complement
-32	10100000	11100000
73	01001001	01001001
-98	11100010	10011110
47	00101111	00101111
-39	10100111	11011001
86	01010110	01010110

d) For sign magnitude, the MSB remains the same and the extra bits are all zero. For two's complement, the extra bits are same as the MSB.

4-bit Sign-magnitude	8-bit Sign-magnitude	Hexadecimal	Decimal	8-bit Two's complement
0110	0000 0110	06	6	0000 0110
1100	1000 0100	84	-4	1111 1100
1111	1000 0111	87	-7	1111 1001
0001	0000 0001	01	1	0000 0001
4-bit Two's complement	**8-bit Two's complement**	**Hexadecimal**	**Decimal**	**8-bit Sign-magnitude**
1010	1111 1010	FA	-6	1000 0110
0101	0000 0101	05	5	0000 0101
1100	1111 1100	FC	-4	1000 0100
1000	1111 1000	$F8$	-8	1000 1000

e)
i) In sign-magnitude form, 0 is represented as $+0$ and -0, which is redundant. In two's complement, 0 is represented as a single value.
ii) In two's complement, addition and subtraction can be performed using the same algorithm. In sign-magnitude form, separate algorithms are required for addition and subtraction.
iii) Two's complement has a larger numerical range than sign-magnitude form, since it can also represent -2^{n-1}, in contrast to sign-magnitude. Moreover, two's complement can handle overflows more efficiently.

[Exercise 3] Hamming Distance and Hamming Weight

a) Table 4 contains rows of two 8-bit binary numbers A and B. Fill in the table with the hamming weight (HW) of A and B, and the hamming distance (HD) between A and B.

A	HW(A)	B	HW(B)	HD(A, B)
01001010		11011110		
00010100		01000000		
01000010		01010101		
11010110		10010000		
10011111		01010010		

Table 4: Table for hamming distance and hamming weight

b) Consider a binary string of length 10. Answer the following questions:
i) If the hamming weight of the string is 7, how many 1s and 0s are in the string?
ii) How many different binary strings can be formed for a hamming weight of 7?

c) Given two binary strings, one of length 10 with a hamming weight of 6, and another of length 10 with a hamming weight of 4, what is the minimum and maximum possible hamming distance between them?

d) What is the hamming distance between any two consecutive numbers represented in Gray code?

e) Consider a set of binary strings $S = \{010, 101, 110, 001\}$. Determine the minimum and maximum hamming distances among all pairs of strings in S.

f) Consider two arbitrary binary strings of length n. Answer the following questions:
i) What is the total number of combinations such that the two binary strings are different from each other?
ii) How many distinct hamming distances are possible for two binary strings with length=n?

[Solution 3] Hamming Distance and Hamming Weight

a) Hamming weight is simply the number of 1s, and Hamming distance is the number of bits that differ between two binary numbers.

A	HW(A)	B	HW(B)	HD(A, B)
01001010	3	11011110	6	3
00010100	2	01000000	1	3
01000010	2	01010101	4	4
11010110	5	10010000	2	3
10011111	6	01010010	3	5

b) i) Number of 1s = 7, Number of 0s = 10 - 7 = 3.
ii) Out of 10 places, choose 7 places for 1s, so number of combinations = $\binom{10}{7} = 120$.

c) Maximum Hamming distance occurs when all four 1s in one string are at different positions from the six 1s in the other string, resulting in HD = 10. Minimum Hamming distance occurs when all four 1s in one string are at the same positions as four of the six 1s in the other string, resulting in HD = 2.

d) Consecutive numbers in Gray code differ by only 1 bit, so the Hamming distance between them is 1.

e) The Hamming distance among all pairs are shown in the table below. The min HD is 1, and the max HD is 3.

HD	010	101	110	001
010	x	3	1	2
101	3	x	2	1
110	1	2	x	3
001	2	1	3	x

f) i) Each string can have 2^n values, so there are a total of $2^n * 2^n = 2^{2n}$ possible pairs. Out of these pairs, 2^n pairs have the same value for both strings. So, number of combinations such that the two binary stings are different $= 2^{2n} - 2^n$.
ii) Hamming distances range from 0 to n. So, $(n + 1)$ distinct Hamming distances are possible.

[Exercise 4] Alternate Number Formats: BCD and Gray Code

a) Binary-Coded Decimal (BCD) is an alternate number format that represents each digit of a decimal number with the corresponding 4-bit binary number (Wiki). In the context of BCD, answer the following questions:
i) Convert the following decimal numbers to BCD: 236, 61 and 158
ii) Are the following BCD numbers valid? 00111001, 01101010, 11100011
iii) What are the disadvantages of using BCD?

b) Gray code is another alternate number format where two consecutive decimal numbers in their binary representation differ by only 1 bit. For example, 1, 2, and 3 are represented as 0001, 0011, and 0010, respectively (Wiki). In the context of Gray code, answer the following questions:
i) Determine if the sequence of consecutive numbers is valid: 0110, 1110, 0101
ii) Convert the following decimal numbers to Gray code: 236, 61 and 158 [Hint]

[Solution 4] Alternate Number Formats: BCD and Gray Code

a) i) Convert each digit of the decimal number to its corresponding 4-bit binary.
$236 = 0010\,0011\,0110$, $61 = 0110\,0001$, $158 = 0001\,0101\,1000$.

ii) Split the binary into 4-bit chunks and check if each 4-bit binary chunk is a number between 0 and 9.
$0011\,1001 = 3\,9$ is valid, $0110\,1010 = 6\,10$ is invalid, $1110\,0011 = 14\,3$ is invalid.

iii) 4-bit binary can represent $2^4 = 16$ different numbers but we are using them to represent only digits from 0 to 9, thus wasting bits and is inefficient.

b) i) The sequence is invalid as the binary representation of consecutive numbers should differ by only 1 bit but three bits are different between 1110 and 0101.

ii) Convert the decimal number to binary, and then convert the binary to Gray code. To convert a binary number to Gray code, follow the following steps:

- The MSB of the Gray code is the same as the MSB of the binary number.
- If the n-th bit of binary is different from the $(n-1)$-th bit of binary, then the $(n-1)$-th bit of the Gray code is 1, else 0.

$236_{10} = 1110\,1100_2 = 1001\,1010$
$61_{10} = 0011\,1101_2 = 0010\,0011$
$158_{10} = 1001\,1110_2 = 1101\,0001$

[Exercise 5] Arithmetic Operations on Signed and Unsigned Integers

a) Convert the decimal numbers in Table 5 to 8-bit binary numbers and perform the arithmetic shift operations.

Unsigned					
A	Bin(A)	A>>1	A>>3	A<<1	A<<3
39					
192					
75					
Signed (two's complement)					
96					
-38					
49					
-106					

Table 5: Binary Shift operations

b) Consider 16 unsigned binary numbers, each consisting of 8 bits.
i) How many bits are needed to represent the sum of these 16 numbers?
ii) How many bits are needed to represent the product of these 16 numbers in binary?

c) Each row of Table 6 contains two decimal numbers: A and B. Convert them to 4-bit binary numbers in two's complement form. Add the 4-bit binary numbers, and represent the sum using only 4 bits. Compare the result of the binary addition with the decimal sum of A and B, and most importantly, explain the difference if any.

A	B	A+B	Bin(A)	Bin(B)	Bin(A)+Bin(B)
6	7				
3	-8				
-2	-8				
5	-3				
1	5				

Table 6: Binary addition

d) Consider two decimal numbers, 12 and 6. Convert them to 4-bit unsigned binary numbers, multiply the two binary numbers, and show the steps involved.

e) Outline an algorithm for:
i) Multiplying two numbers represented in sign-magnitude form
ii) Multiplying two numbers represented in two's complement form

[Solution 5] Arithmetic Operations on Signed and Unsigned Integers

a) For unsigned integers, the new bits are always 0. For signed integers, when right shifting, the new bits are the same as the MSB. When left shifting, the new bits are 0.

Unsigned					
A	Bin(A)	A>>1	A>>3	A<<1	A<<3
39	00100111	00010011	00000100	01001110	00111000
192	11000000	01100000	00011000	10000000	00000000
75	01001011	00100101	00001001	10010110	01011000
Signed (two's complement)					
96	01100000	00110000	00001100	11000000	00000000
-38	11011010	11101101	11111011	10110100	11010000
49	00110001	00011000	00000110	01100010	10001000
-106	10010110	11001011	11110010	00101100	10110000

b) The maximum value of an unsigned 8-bit number is $2^8 - 1 = 255$.
The maximum value of sum can be $16 \times 255 = 4080$. Therefore, $\lceil log_2(4080) \rceil = 12$ bits are needed to represent the sum.
The maximum value of product can be 255^{16}. Therefore, $\lceil log_2(255^{16}) \rceil = 128$ bits are needed to represent the product.
Note that $\lceil x \rceil$ represents the smallest integer value greater than or equal to x.

c) The differences can be explained by overflow. As an example, $6 + 7 = 13$ but 13 is greater than the maximum possible representable value 7, so it rolls over and results in $-8 + (13 - 7 - 1) = -3$ which is the same result as the binary addition.

A	B	A+B	Bin(A)	Bin(B)	Bin(A)+Bin(B)
6	7	13	0110	0111	1101 = -3
3	-8	-5	0011	1000	1011 = -5
-2	-8	-10	1110	1000	0110 = 6
5	-3	2	0101	1101	0010 = 2
1	5	6	0001	0101	0110 = 6

d) $(12)_{10} = (1100)_2$ and $(6)_{10} = (0110)_2$.
The steps for the multiplication of the two numbers are depicted in Table 7.

e)

i) Multiplying two numbers represented in sign-magnitude form:

1. Identify the signs of the two numbers.

			1	1	0	0
		x	0	1	1	0
			0	0	0	0
		1	1	0	0	
	1	1	0	0		
0	0	0	0			
1	0	0	1	0	0	0

Table 7: Multiplication

2. Multiply the magnitudes of the two numbers.
3. Assign the sign of the result based on the signs of the two numbers. (The sign bit should be 1 if exactly one of the sign bits is one and 0 otherwise.)

ii) Multiplying two numbers represented in two's complement form:

1. Identify the signs of the two numbers.
2. Calculate the magnitude of each number.
3. Multiply the magnitudes of the two numbers.
4. Assign the sign of the result based on the signs of the two numbers.

[Exercise 6] Fixed-point Representation

* Unless otherwise stated, consider two's complement.

a) Convert the following numbers represented using fixed-point representation (Wiki) to their decimal form, binary numbers are in two's complement form.
i) 00001010.1101
ii) 10001010.1101
iii) 10101001.0001
iv) 01010101.1001
v) 01011010.1111

b) For a fixed-point format representation with n_1 integer bits and n_2 fractional bits, find the following.
i) the most positive number
ii) the least positive number
iii) the most negative number
iv) the least negative number

c) Convert the following decimal numbers into fixed-point representation with four integer bits in two's complement and four fractional bits.
i) 1.5
ii) 5.25
iii) -0.125
iv) -3.0
v) -6.75

[Solution 6] Fixed-point Representation

a) i) Given fixed-point representation in two's complement form: 00001010.1101

1. Determine the sign:

$$\text{Since the first bit is } 0, \text{ the number is positive}$$

2. Separate the integer and fractional parts:

$$\text{Integer part: } 00001010_2 = 10_{10}$$

$$\text{Fractional part: } 1101_2$$

3. Convert the integer part to decimal:

$$\text{Integer part: } 1010_2 = 10_{10}$$

4. Convert the fractional part to decimal:

$$\text{Fractional part: } 1101_2 = \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^4} = \frac{1}{2} + \frac{1}{4} + \frac{1}{16} = 0.5 + 0.25 + 0.0625 = 0.8125_{10}$$

5. Combine the integer and fractional parts:

$$\text{Number } = \text{Integer part } + \text{Fractional part}$$

$$\text{Number } = 10 + 0.8125$$

6. Calculate the result:

$$\text{Number } = 10 + 0.8125 = 10.8125$$

Therefore, the decimal representation of the given fixed-point representation 00001010.1101 is 10.8125.

ii) A shorter way to calculate the value:

$$\begin{aligned} 10001010.1101 &= -(2^7) + 2^3 + 2^1 + 2^{-1} + 2^{-2} + 2^{-4} \\ &= -(128) + 8 + 2 + 0.5 + 0.25 + 0.0625 \\ &= -117.1875 \end{aligned}$$

iii) -86.9375 **iv)** 85.5625 **v)** 90.9375

b) i) For the most positive number: make the sign bit 0, make all the bits 1 (before and after the decimal). Thus, the value of the number would be:

$$2^{-1} + 2^{-2} + ... + 2^{-n_2} + 2^0 + 2^1 + 2^2 + ... + 2^{n_1-2}$$

.

ii) For the least positive number, make the sign bit 0 and make all the bits before the binary point 0. Make the $(n_2 - 1)$ bits immediately after the binary point 0. Make the n_2^{th} bit after the binary point 1. Thus, the value of the number would be:

$$2^{-n_2}$$

iii) For the most negative number: make the sign bit 1, and rest of the bits 0. Thus, the value of the number would be:

$$-(2^{n_1-1})$$

.

iv) For the least negative number, make the sign bit 1 and rest of the digits 1 as well. Thus, the value of the number would be:

$$-2^{n_1-1} + 2^{n_1-2} + ... + 2^2 + 2^1 + 2^0 + 2^{-1} + 2^{-2} + ... + 2^{-n_2}$$

c) i) Given decimal number: 1.5

1. Determine the sign:

Since the number is positive, the sign bit is 0

2. Separate the integer and fractional parts:

Integer part: $1_{10} = 1_2$

Fractional part: $0.5_{10} = 0.1_2$

3. Convert the integer part to binary:

Integer part: $1_{10} = 0001_2$

4. Convert the fractional part to binary:

Fractional part: $0.5_{10} = 0.1000_2$

5. Combine the integer and fractional parts:

$$\text{Fixed-point representation: } 0001.1000$$

Therefore, the fixed-point representation of the decimal number 1.5 with 4 integer bits, and 4 fractional bits is 0001.1000.

ii) 0101.0100
iii) For negative number, use the idea of two's complement.

1. Write the binary representation of 0.125, which is 0000.0010
2. Convert to one's complement and add 1 to the least significant bit to get the two's complement form
3. As a result, we get 1111.1110, which is equal to -0.125

iv) 1101.0000 **v)** 1001.0100

[Exercise 7] Floating-point Representation

* Unless otherwise stated, the bits of fraction are after the binary point. That is, you have to add a 1 to the fraction for the calculations.

* The exponent provided has bias of $2^{k-1}-1$, where k is the number of bits in the exponent. Therefore, you have to always multiply $2^{exponent-bias}$ during your calculation.

a) In Table 8, convert the floating-point representation to their decimal form:

	Signed bit	Exponent	Fraction	Decimal
i)	0	0100	0100	
ii)	1	1100	1101	
iii)	0	1111	1100	
iv)	0	1110	0001	
v)	1	1011	1100	
vi)	0	0011	1111	

Table 8: Floating point representation

b) Convert the following decimal numbers into binary floating-point representation using the IEEE 754 (Wiki) standard with single-precision (32 bits):
i) 2.5 **ii)** 3.75 **iii)** -4.25 **iv)** -5.0 **v)** 6.25

c) Convert the floating-point number 0.75 to its binary representation according to the IEEE 754 standard in both single-precision and double-precision formats.

d) Below are fixed-point representation of numbers using sign-and-magnitude. Each fixed-point has 1 sign bit, 4 bits for the integer part, and 4 bits for the fractional part. Convert these numbers to IEEE 754 floating-point single-precision format.
i) 01101.0010
ii) 11010.1101
iii) 01111.1111
iv) 11001.1001

[Solution 7] Floating-point Representation

a)

i) For converting the floating-point representations to their decimal format, follow the following steps:

$$\text{Sign bit: } 0$$
$$\text{Exponent: } 0100$$
$$\text{Fraction: } 0100$$

1. Determine the sign:

$$\text{Sign bit } = 0 \implies \text{ Positive number}$$

2. Calculate the exponent:

$$\text{Exponent } = 0100_2 = 4_{10}$$

3. Apply bias to the exponent:

$$\text{Bias } = 2^{(k-1)} - 1$$
$$\text{For 4-bit exponent, } k = 4$$
$$\text{Bias } = 2^{(4-1)} - 1 = 2^3 - 1 = 7$$
$$\text{Exponent with bias } = 4_{10} - 7 = -3$$

4. Convert the fraction to decimal:

$$\text{Fraction } = 0.0100_2$$
$$\text{Fraction } = \frac{0}{2} + \frac{1}{4} + \frac{0}{8} + \frac{0}{16}$$
$$\text{Fraction } = \frac{1}{4}$$

5. Combine the sign, fraction, and exponent:

$$\text{Number } = 1 \times (2^{exponent-bias}) \times (1 + fraction)$$
$$\text{Number } = 1 \times 2^{-3} \times (1 + \frac{1}{4})$$

6. Perform the calculations:

$$\begin{aligned}\text{Number } &= 1 \times 2^{-3} \times (1 + \frac{1}{4}) \\ \text{Number } &= \frac{1}{8} \times (1 + \frac{1}{4}) \\ \text{Number } &= 0.15625\end{aligned}$$

Therefore, the decimal representation of the given floating-point representation is 0.15625.

ii) -58.0
iii) 448.0
iv) 136.0
v) -28.0
vi) 0.12109375

b)

i) Here is how you can convert a decimal number x to it's single-precision IEEE 754 standard (32 bits):

1. Convert $|x|$ to it's unsigned representation (divide the integer part by 2, recursively; multiply the fractional part by 2, recursively)
2. Shift the binary point such that it is after the left most 1
3. Sign bit: 1 if $x < 0$, 0 otherwise (if $x = 0$, the sign bit can be either 1 or 0)
4. Exponent: Add bias to the exponent, and convert to unsigned binary representation
5. Mantissa: The bits after the binary point

Again, let's do explicitly for the first example:

1. Sign Bit = 0 since $2.5 > 0$
2. Integer part (Divide recursively by 2 until reaching 0, and then read the remainders from bottom to top):

$$\begin{matrix} 2/2 \text{ gives remainder } 0 \\ 1/2 \text{ gives remainder } 1 \end{matrix} \Rightarrow (2)_{10} = (10)_2$$

3. Fractional part (Multiply by 2 until reaching 0 for the fractional part, read the integer part from top to bottom):

$$0.5 \cdot 2 \text{ gives integer part } 1 \Rightarrow (0.5)_{10} = (0.1)_2$$

4. Shift one to the left: $10.1 = 1.01 \cdot 2^1$
5. Exponent: $1 + 127 = 128 = 1000\ 0000$
6. Mantissa: $01 = 0100...00$ with 21 finishing 0s

Note: | are there to separate the 3 parts of the number

(i) $2.50 = 0|1000\ 0000|010\ 0000\ 0000\ 0000\ 0000\ 0000$

(ii) $3.75 = 0|1000\ 0000|111\ 0000\ 0000\ 0000\ 0000\ 0000$

(iii) $-4.25 = 1|1000\ 0001|000\ 1000\ 0000\ 0000\ 0000\ 0000$

(iv) $-5.00 = 1|1000\ 0001|010\ 0000\ 0000\ 0000\ 0000\ 0000$

(v) $6.25 = 0|1000\ 0001|100\ 1000\ 0000\ 0000\ 0000\ 0000$

c) Let's convert 0.75 into binary representation, as the integer part is zero we only need to convert the fractional part to binary:

1. Multiply 0.75 by 2, we get 1.5
2. The integer part of 1.5 is 1 and it becomes the first bit after binary point, and for the fractional part we again multiply by 2
3. Multiplying 0.5 by 2, we get 1.0
4. The integer part of 1.0 is 1, which becomes the second bit after binary point
5. The fractional part of 1.0 is 0, therefore, we terminate the conversion as all the remaining bits would be zero.

$$(0.75)_{10} = (0.11)_2 = (1.1)_2 \cdot (2^{-1})$$

Sign bit: 0

Exponent: $-1 + 127 = 126$ thus $0111\ 1110$

Mantissa: $100\ 0000\ 0000\ 0000\ 0000$

Therefore, single precision:
$0|0111\ 1110|100\ 0000\ 0000\ 0000\ 0000\ 0000$

Double Precision:
Double-precision format changes the bias (from 127 to 1023 and the size of the exponent and of the fraction, but the calculations stay similar):

$0|011\ 1111\ 1110|1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000$

Sign bit: 0

Exponent: $-1 + 1023 = 1022$ thus $011\ 1111\ 1110$

Mantissa: $1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000$

d) To convert fixed-point to floating-point representation, we need to calculate the sign bit, exponent bits, and mantissa bits. Follow the steps shown below:

1. Normalize the fixed-point representation by shifting the binary point to the right of leftmost 1
2. The mantissa is the entire binary string after the binary point
3. Calculate the exponent bits by adding the bias of 127 to the power of 2
4. Convert exponent to unsigned binary representation
5. If the number is positive, the sign bit is zero, otherwise it is 1

i) $01101.0010 = +1.1010\ 01 \cdot 2^3 \Rightarrow E = (127 + 3)_{10} = (1000\ 0010)_2 \Rightarrow$ $0|1000\ 0010|101\ 0010\ 0000\ 0000\ 0000\ 0000$

ii) $11010.1101 = -1.0101\ 101 \cdot 2^3 \Rightarrow E = (127 + 3)_{10} = (1000\ 0010)_2 \Rightarrow$ $1|1000\ 0010|010\ 1101\ 0000\ 0000\ 0000\ 0000$

iii) $01111.1111 = +1.1111\ 111 \cdot 2^3 \Rightarrow E = (127 + 3)_{10} = (1000\ 0010)_2 \Rightarrow$ $0|1000\ 0010|111\ 1111\ 0000\ 0000\ 0000\ 0000$

iv) $11001.1001 = -1.0011\ 001 \cdot 2^3 \Rightarrow E = (127 + 3)_{10} = (1000\ 0010)_2 \Rightarrow$ $1|1000\ 0010|001\ 1001\ 0000\ 0000\ 0000\ 0000$

[Exercise 8] Precision Math of Floating- and Fixed-point Representation

* Unless otherwise stated, consider two's complement.

a) Resolution is defined as the minimum difference between two consecutive numbers. So, it is equal to the difference between the smallest nonzero number and zero. In fixed-point representation, resolution becomes equal to the smallest nonzero number because zero can be represented as $(0.0)_2$. In IEEE 754 floating-point representation, however, zero has a special representation $(1.0)_2 \cdot 2^{-127}$. So, the difference (=resolution) is not same as the smallest nonzero number. IEEE 754 specifies two ways of representing fractional numbers:
(I) single-precision with one sign bit, eight exponent bits, and 23 bits for fraction
(II) double-precision with one sign bit, 11 exponent bits, and 52 bits for fraction.
Consider $(0.0)_{10}$ and infinity does not exist.
For both the formats, answer the following questions:
i) What is the range of values that can be represented?
ii) What is the resolution of the representation?

b) All the fractional numbers in decimal format cannot be represented using floating-point representation. For example, if you consider 0.1, then you will see that you cannot represent this number exactly using the IEEE 754 floating-point single-precision representation. The IEEE 754 floating-point single-precision representation closest to 0.1 is $00111101110011001100110011001101$ and this representation exceeds 0.1 by 1.4901×10^{-9}. The magnitude of difference between the true value in decimal format and its representation in floating-point is defined as round-off error.

Considering rounding off to the nearest representation (and to even when there is a tie), find the round-off error when using a sign-and-magnitude floating-point representation with one sign bit, two exponent bits, and four fraction bits for the following numbers.

Hint: Floating-point representation $= (-1)^{sign} \times (1 + fraction) \times 2^{exponent-bias}$, and $bias = 2^{k-1} - 1$ where k is the number of bits for the exponent)

i) 3.5625 **ii)** 3.9 **iii)** -0.52 **iv)** -0.67

c) Similar to the previous question, all the fractional numbers in decimal format cannot be represented using fixed-point representation. Again, for example, if you consider 0.1, then you will see that you cannot represent this number exactly using fixed-point representation with 4 integer bits and 4 decimal bits. Since you cannot represent the number exactly, you can use the number closest to 0.1 which will be represented as 0000.0010_2.

For each fractional number below, provide the fixed-point representation, which has four integer bits in two's complement, and four fractional bits, and is closest to the given decimal representation.

i) 7.2 **ii)** -6.42 **iii)** -3.67 **iv)** 5.33

d) Calculate the resolution and range of fixed-point representation in sign-and-magnitude form with one sign bit, 8 bits for the integer part and 23 bits for the fractional part.

[Solution 8] Precision of Floating- and Fixed-point Representation

a)

i) Single-precision: 1 sign bit, 8 exponent bits, and 23 bits for fraction
The range can be computed as follows (considering infinity and $(0.0)_{10}$ doesn't exist):

(a) In 32-bit floating-point representation, the exponent bits can represent values from $[0, 255]$. To represent negative and positive exponents, a value of 127 is subtracted from the exponent bits. Therefore, the value of exponents is in the range of $[-127, 128]$ (if we were not ignoring infinity and $(0.0)_{10}$, then -127 and 128 would hold a special value of 0.0_{10} and infinity, respectively). As a result, the maximum value of exponent we get is 2^{128}.

(b) The maximum value of fractional part is when all mantissa bits are 1, which implies the value of coefficient will be $(1 + 2^{-1} + 2^{-2} \cdots + 2^{-23})$. The sum of the values can be calculated using the formula for geometric series (Link for formula). After applying the formula, we get $(2 - 2^{-23})$.

(c) Putting sign, exponent, and mantissa together, we get $2^{128} \cdot (2 - 2^{-23})$.

(d) Rearranging the terms, we get $2^{105}(2^{24} - 1)$.

(e) Similarly, the minimum value can be calculated by just changing the sign bit. As a result, the minimum value we get is $-2^{105}(2^{24} - 1)$.

(f) Range of values: $[-(2^{24} - 1) \cdot 2^{105}, (2^{24} - 1) \cdot 2^{105}]$

Range = $(2^{24} - 1) \cdot 2^{106}$

Resolution is the difference between the smallest nonzero number $(1.000\ 0000\ 0000\ 0000\ 0000\ 0001 \cdot 2^{0-127})$ and the representation of zero, which is $1.0 \cdot 2^{-127}$. As a result, the difference is equal to $(1+2^{-23}) \cdot 2^{-127} - 1.0 \cdot 2^{-127} = 2^{-150}$.

ii) Double-precision: one sign bit, 11 exponent bits, and 52 bits for fraction
Similar to single-precision, we can compute the maximum for double-precision format (considering $(0.0)_{10}$ and infinity doesn't exist):
$1.1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111 \cdot 2^{2^{11}-1-1023} = (2^{53} - 1) \cdot 2^{972}$
Range of values: $[-(2^{53} - 1) \cdot 2^{972}, (2^{53} - 1) \cdot 2^{972}]$
Range = $(2^{53} - 1) \cdot 2^{973}$

Resolution is the difference between the smallest nonzero number

$(1.0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001\ \cdot 2^{0-1023})$ and the representation of zero, which is $1.0 \cdot 2^{-1023}$. As a result, the difference is equal to $(1 + 2^{-52}) \cdot 2^{-1023} - 1.0 \cdot 2^{-1023} = 2^{-1075}$.

b) To find the roundoff error when using a sign-and-magnitude floating-point representation with one sign bit, two exponent bits, and four fraction bits, follow the following steps:

i) $(3.5625)_{10}$

(a) 3.5625 can be represented as 11.1001

(b) Normalize 11.1001 by shifting binary point to the right of left most 1, as a result, we get $1.11001 \cdot 2^1$

(c) Given number is positive, sign bit is 0

(d) $bias = 2^{2-1} - 1 = 1$

(e) $exponent - bias = 1 \Rightarrow exponent = 2$

(f) Convert exponent to binary form: 10_2

(g) $mantissa = 11001$ so mantissa belongs to $[1100, 1100 + 0001] = [1100, 1101]$ (considering 4 bits)

(h) Rounding mantissa to nearest value gives us 1100. Note that there is a tie so we need to round to even. Generally, We took the max of the interval when the $n + 1$th bit is 1; otherwise, we would have taken the min of the interval. Note, in the case of negative numbers, if the $n + 1$th bit was one, we would have taken the min of the interval. Similarly, you can think of this as adding $(r^{-f})/2$ which is $2^{-(f+1)}$ for $r = 2$.

(i) Thus, the final result of the conversion is $0|10|1100$ which equals $1.11_2 \times (2^{2-1}) = 11.1_2 = 3.5$, since we "lost" a bit while rounding during the conversion.

Round-off error = $|3.5625 - 3.5| = 0.0625$.

Note: For the rest of the subquestions, the interval containing the number is calculated in floating-point representation and then the roundoff error is shown.

ii) $(3.9)_{10} \in [0|10|1111, 0|11|0000] \Rightarrow$ Round-off error = $3.9 - 3.875 = 0.025$

iii) $(-0.52)_{10} \in [1|00|0001, 1|00|0000] \Rightarrow$ Round-off error = $|-0.52+0.53125| = 0.01125$

iv) $(-0.67)_{10} \in [1|00|0110, 1|00|0101] \Rightarrow$ Round-off error = $|-0.67+0.65625| = 0.01375$

c) Provide the fixed-point representation, which has four integer bits, and four fractional bits, and is closest to the given decimal representation.

i) $(7.2)_{10} \in [(0111.0011)_2, (0111.0100)_2] \Rightarrow$ the number closest to $(7.2)_{10}$ is 0111.0011

ii) $(-6.42)_{10} \in [(1001.1001)_2, (1001.1010)_2] \Rightarrow$ the number closest to $(-6.42)_{10}$ is 1001.1001

iii) $(-3.67)_{10} \in [(1100.0101)_2, (1100.0110)_2] \Rightarrow$ the number closest to $(-3.67)_{10}$ is 1100.0101

iv) $(5.33)_{10} \in [(0101.0101)_2, (0101.0110)_2] \Rightarrow$ the number closest to $(5.33)_{10}$ is 0101.0101

d) Resolution for fixed-point representation is 2^{-f}, where f is the number of fractional bits. In this case, we have 23 fractional bits, therefore, the resolution is 2^{-23}.

The range is the difference between the most positive and the most negative number. The most positive number is with sign bit set to 0 and all other bits set to 1, which equals to the sum of $(2^8 - 1)$ from the integer part and $(1 - 2^{-23})$ from the fractional part. The most negative number is with sign bit set to 1 and all other bit set to 1, which is same as the most positive number but with a negative sign. Therefore, range is equal to $2 * ((2^8 - 1) + (1 - 2^{-23})) = 2^9 - 2^{-22}$.

[Exercise 9] Arithmetic Operations on Fractional Numbers

a) In Table 9, given the following fractional numbers in decimal format, fill in the IEEE 754 single-precision (SP) representation for each number and compute their sum. Then, verify if the sum of the IEEE 754 representations matches the sum of the original decimal numbers.

A	B	SP(A)	SP(B)	A + B	SP(A) + SP(B)
0.125	0.25				
-0.375	0.5				
0.625	0.75				
-0.875	-1.0				
1.125	-1.25				

Table 9: Addition in floating point representation

b) In Table 10, given the following fractional numbers in decimal format, fill in the fixed-point (FiP) representation with four integer bits in two's complement form and four fractional bits. Compute the sum of the fixed-representations. Verify if the sum of the fixed-point representations matches the sum of the original decimal numbers.

A	B	FiP(A)	FiP(B)	A + B	FiP(A) + FiP(B)
1.125	3.25				
-4.375	6.5				
2.625	-7.75				
-0.875	-1.125				
6.25	-3.875				

Table 10: Addition in fixed point representation

[Solution 9] Arithmetic Operations on Fractional Numbers

a) Let's say A and B are two single-precision floating point numbers to be added, and A has a larger exponent than B, follow the below steps to solve the sub-questions:

1. Subtract the exponent of A by the exponent of B (you obtain a number the difference d)
2. Shift right the significand of B by d
3. Add or Subtract the significands (depending on sign bits) and keep the resulting sign bit
4. Normalize the result by shifting and round off if necessary

Let's do it on the first example:

1. $SP(A) = 0|0111\ 1100|000\ 0000\ 0000\ 0000\ 0000\ 0000$
2. $SP(B) = 0|0111\ 1101|000\ 0000\ 0000\ 0000\ 0000\ 0000$
3. B has a larger exponent than A, therefore, $d = -2 - (-3) = 1$
4. $A = (1.0)_2 \cdot 2^{-3}$ after shifting $A = (0.1)_2 \cdot 2^{-2}$
5. $B = (1.0)_2 \cdot 2^{-2}$
6. $A + B = (1.1)_2 \cdot 2^{-2}$
7. Normalization is already done, and no rounding is required thus $SP(A) + SP(B) = 0|0111\ 1101|100\ 0000\ 0000\ 0000\ 0000\ 0000$

Here are all the conversions:

Decimal	Single-Precision
0.125	0|0111 1100|000 0000 0000 0000 0000 0000
0.25	0|0111 1101|000 0000 0000 0000 0000 0000
−0.375	1|0111 1101|100 0000 0000 0000 0000 0000
0.5	0|0111 1110|000 0000 0000 0000 0000 0000
0.625	0|0111 1110|010 0000 0000 0000 0000 0000
0.75	0|0111 1110|100 0000 0000 0000 0000 0000
−0.875	1|0111 1110|110 0000 0000 0000 0000 0000
−1.0	1|0111 1111|000 0000 0000 0000 0000 0000
1.125	0|0111 1111|001 0000 0000 0000 0000 0000
−1.25	1|0111 1111|010 0000 0000 0000 0000 0000

And here are the additions:

Decimal	Single-Precision
0.375	0|0111 1101|100 0000 0000 0000 0000 0000
0.125	0|0111 1100|000 0000 0000 0000 0000 0000
1.375	0|0111 1111|011 0000 0000 0000 0000 0000
-1.875	1|0111 1111|111 0000 0000 0000 0000 0000
-0.125	1|0111 1100|000 0000 0000 0000 0000 0000

You will notice with 32 bits, the sum of decimal numbers matches the sum from floating-point format.

b) Here are all the conversions:

Decimal	Fixed Point
1.125	0001.0010
3.25	0011.0100
-4.375	1011.1010
6.5	0110.1000
2.625	0010.1010
-7.75	1000.0100
-0.875	1111.0010
-1.125	1110.1110
6.25	0110.0100
-3.875	1100.0010

And here are the additions:

Decimal	Fixed-Point
4.375	0100.0110
2.125	0010.0010
-5.125	1010.1110
-2	1110.0000
2.375	0010.0110

You will notice with 4 bits of integer part and 4 bits of fractional part, the sum of decimal numbers matches the sum from fixed-point format.

[Exercise 10] Round-Off Error in Floating-Point Numbers

a) Consider x and y as the IEEE 754 single-precision representations of 12.345 and 0.00012345, respectively.

i) Calculate $x + y$ in IEEE 754.
ii) What is the round-off error of $x + y$?

[Solution 10] Round-Off Error in Floating-Point Numbers

a) First, we must find x and y by converting the decimal numbers to IEEE 754 single-precision numbers.

$$x = 0 \mid 10000010 \mid 10001011000010100011111$$
$$y = 0 \mid 01110010 \mid 00000010111001001011011$$

i) $x + y$ can be computed from the values we obtained in these steps:

- First, we must find the exponent values. For x, the value is $130 - bias = 130 - 127 = 3$. Repeating for y, we find -13.
- To align the exponents for addition, we adjust y's mantissa to match x's exponent value which is 3. For this, y's mantissa must be shifted right by $3 - (-13) = 16$ digits. Note that we can skip the first step, as we only need the *difference* between two exponents: $130 - bias - (114 - bias) = 16$.
- After alignment, y's mantissa becomes $0.000000000000000100000010111001001011011$. We must add this to x's mantissa, $1.10001011000010100011111$.
- The result is $1.100010110000101101000000111001001011011$ which does not require normalization, but must be rounded. The rounded result is $1.10001011000010110100000$.
- The leading 1 is hidden again to give us 10001011000010110100000. The exponent is calculated as $3 + bias = 3 + 127 = 130$ which is represented as 10000010. The sign bit is zero throughout our addition.
- Final result:
$$0 \mid 10000010 \mid 10001011000010110100000$$

ii) The round-off error is equal to the difference between the represented value and the actual value.

- The represented value is $0 \mid 10000010 \mid 10001011000010110100000$, which is equal to $2^{130-127} \times 1.10001011000010110100000$. In decimal: 12.345123291015625
- The actual value: $12.345 + 0.00012345 = 12.34512345$
- Round-off error is calculated as follows:
$$|12.345123291015625 - 12.34512345| = 1.58984376 \times 10^{-7}$$

Part II: Digital Logic and Design with Verilog

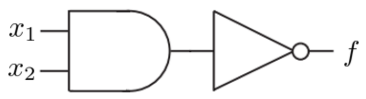
[Exercise 1] Truth Tables

a) Consider a digital logic circuit with n inputs and a single output. How many truth table entries are required to fully represent the functionality of this circuit?

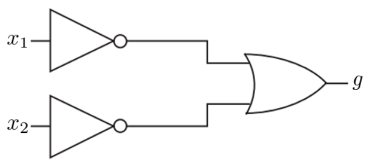
b) Consider a truth table describing the functionality of a digital logic circuit. Could this truth table equally correctly describe another digital circuit? In other words, can two logic circuits implemented differently (i.e., with a different combination of gates) have the same truth table or not?

c) Show the truth tables for the two logic circuits below, and compare them. What can you tell about these two logic circuits from the truth table comparison?

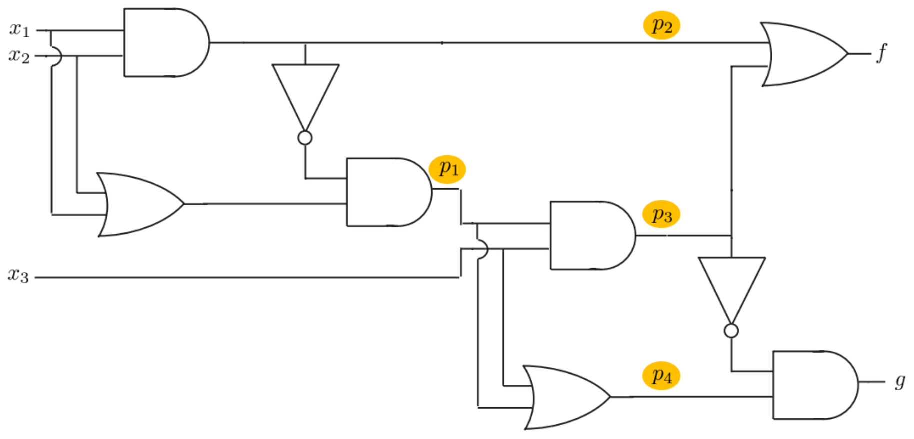
i)



ii)



d) Consider now a much more complex logic circuit illustrated below. It takes three inputs and produces two outputs. Construct the truth table for this circuit. Hint: To make your task simpler, we advise you include in the table the data corresponding to the intermediate values labeled as p_1, p_2, p_3, and p_4.



[Solution 1] Truth Tables

a) A circuit with n inputs and one output is fully represented with a truth table containing all possible input value combinations. As every input can take on two values (zero or one), the truth table contains 2^n entries.

b) Yes, one truth table can describe the functionality of different digital logic circuits.

In digital circuits, a truth table represents the relationship between inputs and outputs. It shows all possible combinations of inputs along with the corresponding outputs but does not put any limitations on the implementation of the circuit itself. In other words, it is just a way of identifying the behavior (expected output) of the black box circuit when different inputs are applied.

An example of having one truth table representing two different digital logic circuits is given in the subsequent question (c).

c) The way to get the truth table for any circuit is to supply all possible inputs to it and see the corresponding output for each combination of inputs. The truth tables are shown below. They are identical, even though the two circuits are different. Hence, the circuits implement the same logic function.

i)

x_1	x_2	f
0	0	1
0	1	1
1	0	1
1	1	0

ii)

x_1	x_2	g
0	0	1
0	1	1
1	0	1
1	1	0

d)

$$\begin{aligned} p_2 &= x_1 \cdot x_2 \\ p_1 &= \overline{p_2} \cdot (x_1 + x_2) \\ p_3 &= p_1 \cdot x_3 \\ p_4 &= p_1 + x_3 \\ f &= p_2 + p_3 \\ g &= \overline{p_3} \cdot p_4 \end{aligned}$$

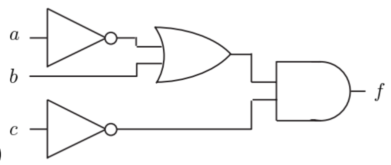
The truth table is shown below.

x_1	x_2	x_3	p_1	p_2	p_3	p_4	f	g
0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	1
0	1	0	1	0	0	1	0	1
0	1	1	1	0	1	1	1	0
1	0	0	1	0	0	1	0	1
1	0	1	1	0	1	1	1	0
1	1	0	0	1	0	0	1	0
1	1	1	0	1	0	1	1	1

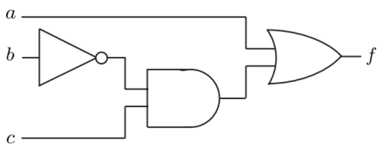
[Exercise 2] Logical Expressions and Boolean Algebra

a) Find the Boolean expressions describing the functionality of the circuits below.

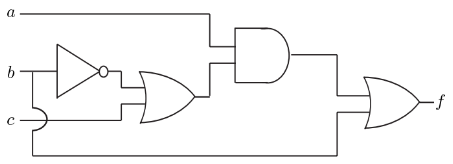
i)



ii)



iii)



b) Using Boolean algebra transformations, simplify the following logical expressions.

i) $f = ab(\bar{b}c + ac)$

ii) $f = ab + a(b + c) + b(b + c)$

iii) $f = (a + \bar{b} + \bar{c})(a + \bar{b} + c)(a + b + \bar{c})$

c) Use Boolean algebra transformations to verify the correctness of the logical equalities below. In every step, specify the transformation (i.e., the exact axiom, theorem, property) applied.

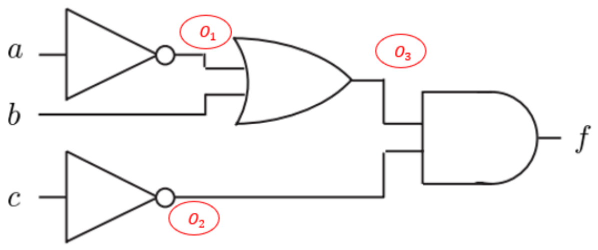
i) $a + bc = (a + b)(a + c)$

ii) $(a + b)(a + \bar{b}) = a$

[Solution 2] Logical Expressions and Boolean Algebra

a)

i) To get the algebraic/Boolean expression for a circuit consisting of multiple gates, you start by getting the Boolean expression for each individual gate. This circuit contains 3 different types of gates: **NOT**, **OR**, **AND** gates. Let the intermediate outputs be O_1, O_2, O_3 (marked in red in the figure below).



1. Get the Boolean expression for the top **NOT** gate: $O_1 = \overline{a}$
2. Get the Boolean expression for the bottom **NOT** gate: $O_2 = \overline{c}$
3. Get the Boolean expression for the **OR** gate: $O_3 = O_1 + b$
4. Substitute for O_1 in (3) with its expression in (1): $O_3 = \overline{a} + b$
5. Get the Boolean expression for the **AND** gate: $f = O_3 \cdot O_2$
6. Substitute for O_3 in (5) with its expression in (4) and for O_2 in (5) with its expression in (2): $f = (\overline{a} + b) \cdot \overline{c}$

ii) Following the same techniques as in (i), the Boolean expression is expressed as follows: $f = a + \overline{b} \cdot c$

iii) Again following the same technique, $f = (\overline{b} + c) \cdot a + b$

b)

i)

$$\begin{aligned} f &= ab(\overline{b}c + ac) \\ &= ab\overline{b}c + abac \quad (x \cdot x = x) \\ &= ab\overline{b}c + abc \quad (x \cdot \overline{x} = 0) \\ &= 0 + abc \\ &= abc \end{aligned}$$

ii)

$$\begin{aligned} f &= ab + a(b+c) + b(b+c) \\ &= ab + ab + ac + bb + bc \qquad (x + x = x; x \cdot x = x) \\ &= ab + ac + b + bc \\ &= ac + b \cdot (a + 1 + c) \qquad (x + 1 = 1) \\ &= ac + b \cdot 1 \\ &= ac + b \end{aligned}$$

iii)

$$\begin{aligned} f &= (a + \bar{b} + \bar{c})(a + \bar{b} + c)(a + b + \bar{c}) \\ &= a(a + \bar{b} + c)(a + b + \bar{c}) + \bar{b}(a + \bar{b} + c)(a + b + \bar{c}) + \bar{c}(a + \bar{b} + c)(a + b + \bar{c}) \end{aligned}$$

using $x = x \cdot x,$ we can write $a = a \cdot a, \bar{b} = \bar{b} \cdot \bar{b},$ and $\bar{c} = \bar{c} \cdot \bar{c}$

$$\begin{aligned} &= a(a + \bar{b} + c)a(a + b + \bar{c}) + \bar{b}(a + \bar{b} + c)\bar{b}(a + b + \bar{c}) + \bar{c}(a + \bar{b} + c)\bar{c}(a + b + \bar{c}) \\ &= (aa + a\bar{b} + ac)(aa + ab + a\bar{c}) + (a\bar{b} + \bar{b}\bar{b} + \bar{b}c)(a\bar{b} + b\bar{b} + \bar{b}\bar{c}) + (a\bar{c} + \bar{b}\bar{c} + c\bar{c})(a\bar{c} + b\bar{c} + \bar{c}\bar{c}) \end{aligned}$$

using $x \cdot x = x; \overline{x} \cdot \overline{x} = \overline{x}; x \cdot \overline{x} = 0$

$$\begin{aligned} &= (a + a\bar{b} + ac)(a + ab + a\bar{c}) + (a\bar{b} + \bar{b} + \bar{b}c)(a\bar{b} + 0 + \bar{b}\bar{c}) + (a\bar{c} + \bar{b}\bar{c} + 0)(a\bar{c} + b\bar{c} + \bar{c}) \\ &= a(1 + \bar{b} + c)a(1 + b + \bar{c}) + \bar{b}(a + 1 + c)\bar{b}(a + \bar{c}) + \bar{c}(a + \bar{b})\bar{c}(a + b + 1) \end{aligned}$$

using $x + 1 = 1; \overline{x} \cdot \overline{x} = \overline{x}$

$$= aa + \bar{b}(a + \bar{c}) + \bar{c}(a + \bar{b})$$

using $x \cdot x = x$

$$\begin{aligned} &= a + \bar{b}(a + \bar{c}) + \bar{c}(a + \bar{b}) \\ &= a + \bar{b}a + \bar{b}\bar{c} + \bar{c}a + \bar{c}\bar{b} \\ &= a(1 + \bar{b} + \bar{c}) + \bar{b}(\bar{c} + \bar{c}) \end{aligned}$$

using $x + 1 = 1; x + x = x$

$$= a + \bar{b}\bar{c}$$

c)
i) The proof is as follows.

$$\begin{aligned}(a+b)\cdot(a+c) &= aa+ac+ab+bc \quad (x\cdot x=x)\\ &= a+ac+ab+bc\\ &= a(1+c+b)+bc \quad (x+1=1)\\ &= a\cdot 1+bc\\ &= a+bc\end{aligned}$$

ii) The proof is as follows.

$$\begin{aligned}(a+b)\cdot(a+\bar{b}) &= aa+ab+a\bar{b}+b\bar{b} \quad (x\cdot\bar{x}=0)\\ &= a+ab+a\bar{b}+0\\ &= a(1+b+\bar{b}) \qquad (x+1=1)\\ &= a\cdot 1\\ &= a\end{aligned}$$

[Exercise 3] Logic Networks

a) A circuit that controls a given digital system has three inputs: x_1, x_2, and x_3. It has to recognize three different conditions:

- Condition A is true if x_3 is true and either x_1 is true or x_2 is false.
- Condition B is true if x_1 is true and either x_2 or x_3 is false.
- Condition C is true if x_2 is true and either x_1 is true or x_3 is false.

Please note that for clauses with *"either or"* use an OR gate. For example, for either x_1 is true or x_2 is false, we can write $(x_1 + \overline{x_2})$.

The control circuit must produce an output of 1 if at least two of the conditions A, B, and C are true. Design a simple circuit that can be used for this purpose. Hint: start by defining logic expressions for each of the conditions. Apply Boolean algebra transformations to reduce the circuit size. The final circuit should not use more than three basic logic gates (AND/OR/NOT).

b) NAND gates

i) Show the truth table for a 3-input NAND gate.

ii) Using only the basic logic gates (AND/OR/NOT) with one or two inputs, draw a logic circuit equivalent to a three-input NAND gate.

iii) Using only the basic logic gates (AND/OR/NOT) with one or two inputs, draw a logic circuit equivalent to a four-input NAND gate.

c) An n-input **XOR** gate implements the following logical function:

- The output is 1 when it has an odd number of ones at the input.
- The output is 0 when it has an even number of ones at the input. Zero is considered an even number.

i) Show the truth table for a three-input XOR gate.

ii) Draw a logic circuit implementing the functionality of the three-input XOR logic gate, using only the basic logic gates (AND/OR/NOT). In this question, the number of inputs of the basic logic gates is not limited to one or two only.

[Solution 3] Logic Networks

a) Using 1 for true and 0 for false, we can express the three conditions as follows:

$$\begin{aligned}
A &= x_3\left(x_1+\bar{x}_2\right) = x_3x_1 + x_3\bar{x}_2\\
B &= x_1\left(\bar{x}_2+\bar{x}_3\right) = x_1\bar{x}_2 + x_1\bar{x}_3\\
C &= x_2\left(x_1+\bar{x}_3\right) = x_2x_1 + x_2\bar{x}_3
\end{aligned}$$

Then, the desired output of the circuit can be expressed as $f = AB + AC + BC$. These product terms can be determined as:

$$\begin{aligned}
AB &= \left(x_3x_1 + x_3\bar{x}_2\right)\left(x_1\bar{x}_2 + x_1\bar{x}_3\right)\\
&= x_3x_1x_1\bar{x}_2 + x_3x_1x_1\bar{x}_3 + x_3\bar{x}_2x_1\bar{x}_2 + x_3\bar{x}_2x_1\bar{x}_3\\
&= x_3x_1\bar{x}_2 + 0 + x_3\bar{x}_2x_1 + 0\\
&= x_1\bar{x}_2x_3
\end{aligned}$$

$$\begin{aligned}
AC &= \left(x_3x_1 + x_3\bar{x}_2\right)\left(x_2x_1 + x_2\bar{x}_3\right)\\
&= x_3x_1x_2x_1 + x_3x_1x_2\bar{x}_3 + x_3\bar{x}_2x_2x_1 + x_3\bar{x}_2x_2\bar{x}_3\\
&= x_3x_1x_2 + 0 + 0 + 0\\
&= x_1x_2x_3
\end{aligned}$$

$$\begin{aligned}
BC &= \left(x_1\bar{x}_2 + x_1\bar{x}_3\right)\left(x_2x_1 + x_2\bar{x}_3\right)\\
&= x_1\bar{x}_2x_2x_1 + x_1\bar{x}_2x_2\bar{x}_3 + x_1\bar{x}_3x_2x_1 + x_1\bar{x}_3x_2\bar{x}_3\\
&= 0 + 0 + x_1\bar{x}_3x_2 + x_1\bar{x}_3x_2\\
&= x_1x_2\bar{x}_3
\end{aligned}$$

Therefore, f can be written as

$$\begin{aligned}
f &= x_1\bar{x}_2x_3 + x_1x_2x_3 + x_1x_2\bar{x}_3\\
&= x_1\bar{x}_2x_3 + x_1x_2x_3 + x_1x_2x_3 + x_1x_2\bar{x}_3 \quad [\text{using } x + x = x]\\
&= x_1\left(\bar{x}_2 + x_2\right)x_3 + x_1x_2\left(x_3 + \bar{x}_3\right)\\
&= x_1x_3 + x_1x_2\\
&= x_1\left(x_3 + x_2\right)
\end{aligned}$$

Figure 1 shows the circuit for $f = x_1 \cdot (x_2 + x_3)$.

b)
i) Table 11 represents three-input NAND gate.
ii) The truth table shown in Table 11 represents a gate which is the opposite of a 3-input AND gate. An equivalent Boolean expression would be $\overline{(a \cdot b \cdot c)}$. Using associativity,

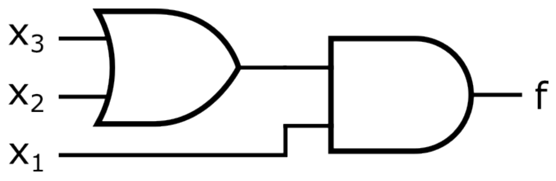


Figure 1: The circuit representing $f = x_1 \cdot (x_2 + x_3)$.

a	b	c	f
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	0

Table 11: Truth table for three-input NAND

we can rewrite this as $\overline{((a \cdot b) \cdot c)}$, and draw a circuit with three logic gates as shown in Figure 2.

iii) To construct a four-input NAND gate, we can write $\overline{(a \cdot b \cdot c \cdot d)}$ as $\overline{(a \cdot b) \cdot (c \cdot d)}$ using the associative property. As a result, we can use two two-input AND gates to compute $(a \cdot b)$ and $(c \cdot d)$, and use another AND gate to compute the product of $(a \cdot b)$ and $(c \cdot d)$. Finally, an inverter at the end can be used to get the complement. See Figure 3 for the final circuit.

c)
i)

a	b	c	f
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

Table 12: Truth table for three input XOR gate.

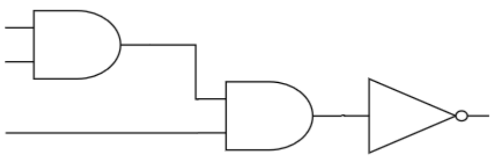


Figure 2: The circuit representing three-input NAND.

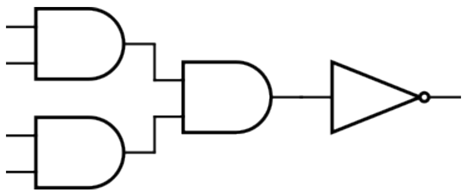


Figure 3: The circuit representing four-input NAND gate.

ii) To get the circuit for three-input XOR gate, start by writing the Boolean expression using the truth table shown in Table 12. Write the product of inputs for which the output is 1 in the truth table, and then sum these products. As a result, we get $(\overline{a} \cdot \overline{b} \cdot c) + (\overline{a} \cdot b \cdot \overline{c}) + (a \cdot \overline{b} \cdot \overline{c}) + (a \cdot b \cdot c)$. The calculated expression can be expressed using four three-input AND gates and one four-input OR gate, as shown in Figure 4. Note that to get the complement of an input you can pass it through an inverter.

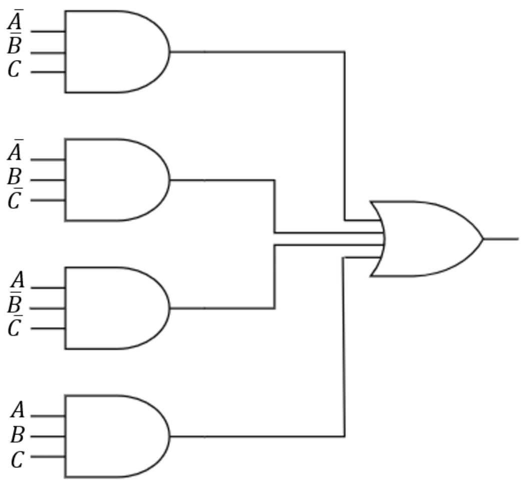


Figure 4: The circuit representing three-input XOR logic gate.

[Exercise 4] Timing Diagrams

a) Draw timing diagrams for the circuit shown in Figure 5. Assume the input sequence is the same as in the corresponding truth table (i.e., (abc) = {000, 001, 010, ..., 111}). Show the waveforms at the inputs, the internal connections (p_1, p_2, p_3, p_4), and the outputs.

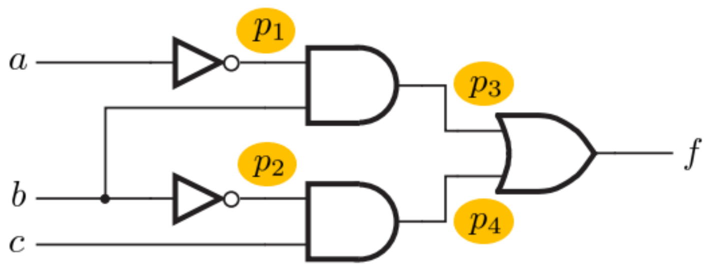


Figure 5: A three-input, one-output logic circuit.

b) Draw timing diagrams for the below circuit. Assume the input sequence is the same as in the corresponding truth table (i.e., (abc) = {000, 001, 010, ..., 111}). Show the waveforms at the inputs, the internal connections (p_1, p_2, p_3, p_4), and the outputs.

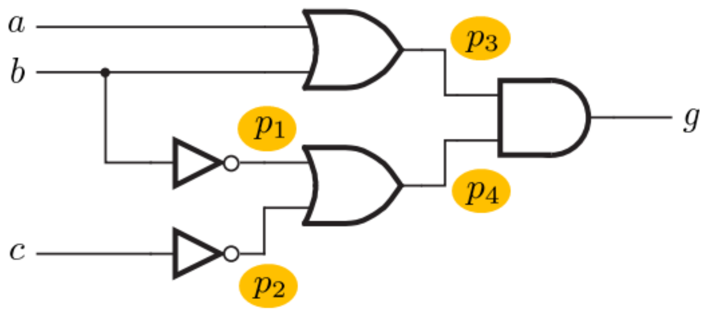


Figure 6: A three-input, one-output logic circuit.

[Solution 4] Timing Diagrams

a) To draw any timing diagram, apply square waves to inputs such that you get the same input sequence as in the truth table.

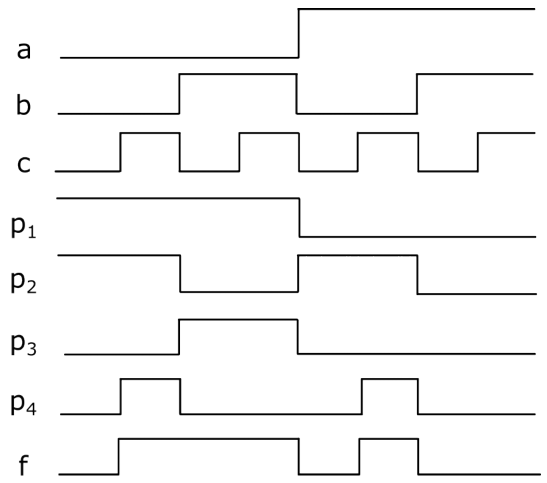


Figure 7: Timing diagram for the logic circuit.

b) Similarly to the previous question, apply square waves to inputs such that you get the same input sequence as in the truth table.

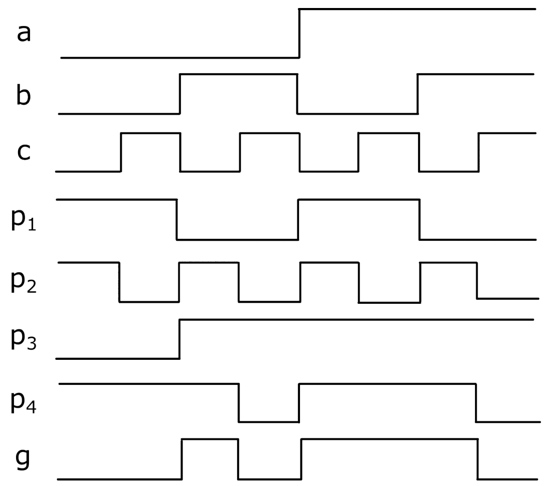


Figure 8: Timing diagram for the logic circuit.

[Exercise 5] Venn Diagrams

Use Venn diagrams to verify the correctness of the logical equalities below. Show the Venn diagrams of the intermediate steps.

a) Distributive property $x + yz = (x + y)(x + z)$

b) DeMorgan's theorem (i.e., $\overline{x \cdot y} = \bar{x} + \bar{y}$)

c) $(x_1 + x_2 + x_3) \cdot (x_1 + x_2 + \bar{x}_3) = x_1 + x_2$

[Solution 5] Venn Diagrams

a) The Venn diagrams corresponding to the Left Hand Side (LHS) and the Right Hand Side (RHS) are shown in Fig. 9. As they are identical, the equality of the two logical expressions is proven correct.

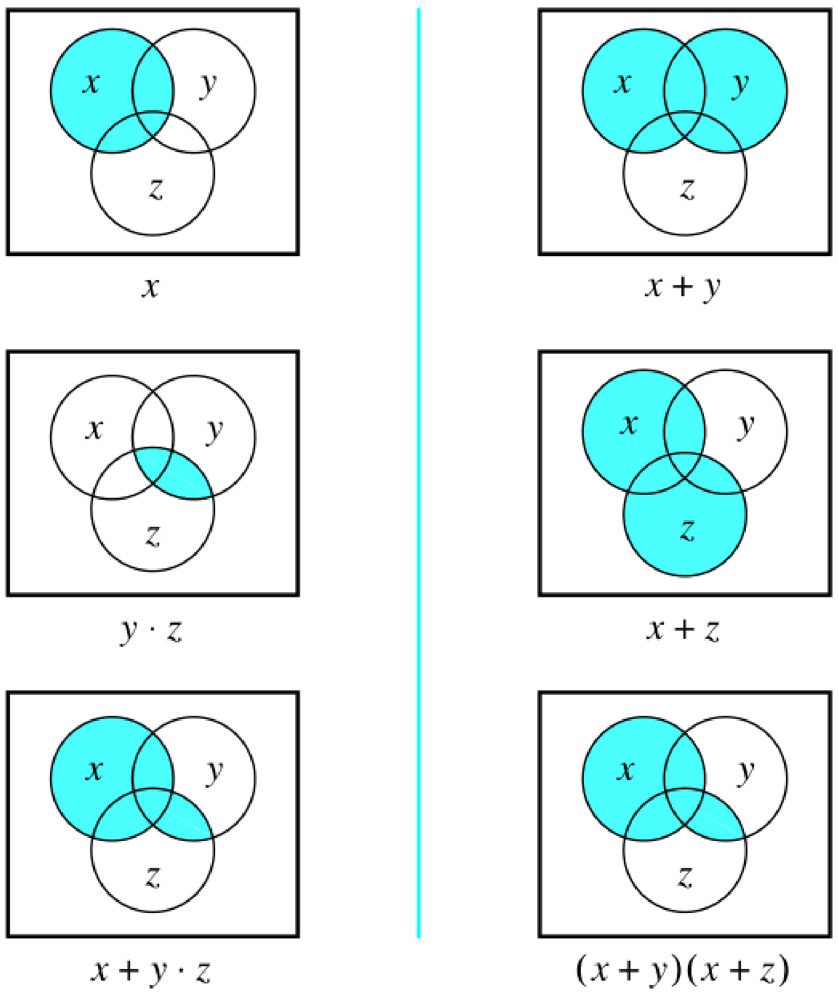


Figure 9: Venn diagram illustrating the distributive property.

b) The Venn diagrams corresponding to the Left Hand Side (LHS) and the Right Hand Side (RHS) are shown in Fig. 10. As they are identical, the equality of the two logical expressions is proven correct.

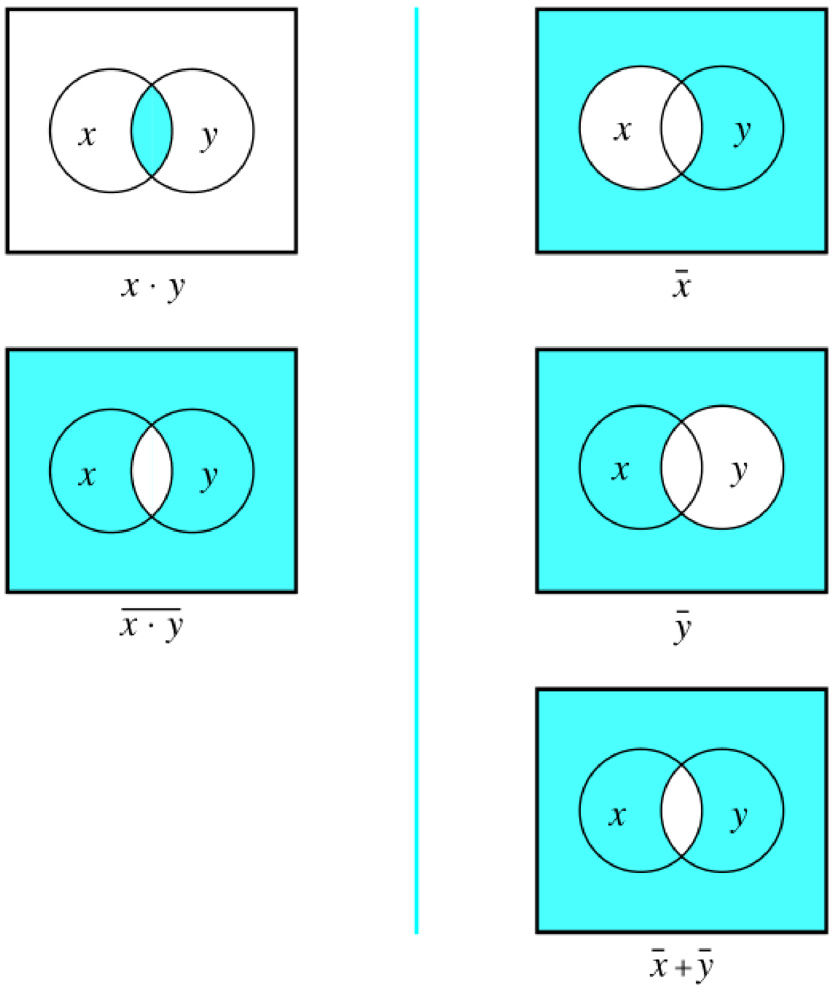


Figure 10: Venn diagram illustrating the De Morgan's theorem.

c) The Venn diagrams corresponding to the Left Hand Side (LHS) and the Right Hand Side (RHS) are shown in Fig. 11. As they are identical, the equality of the two logical expressions is proven correct.

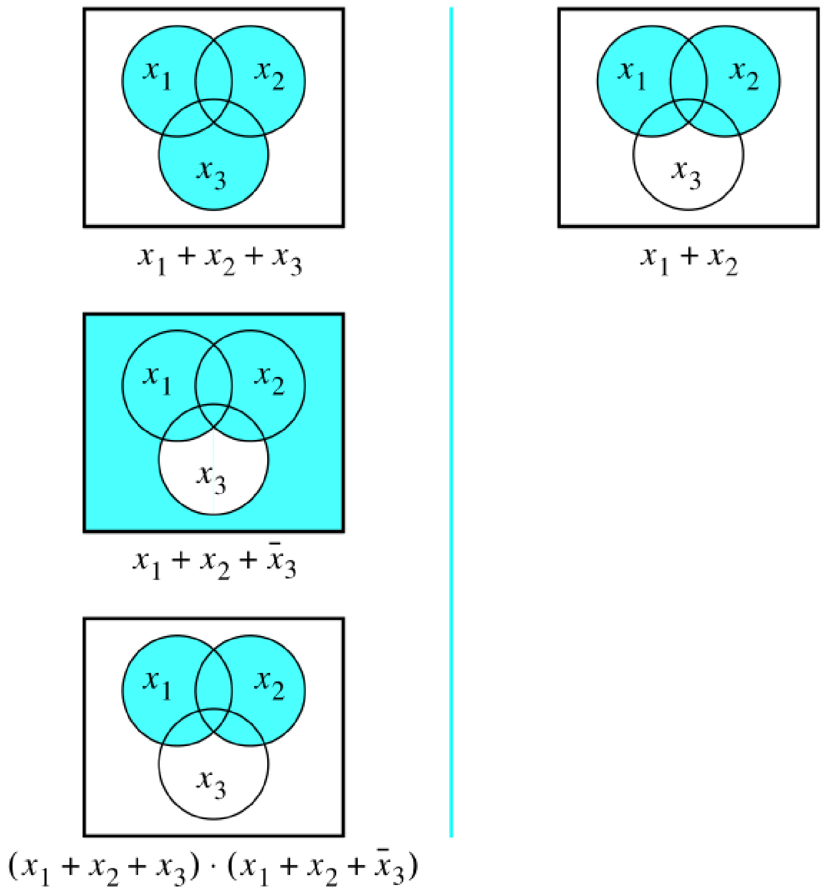


Figure 11: Venn diagram corresponding to $(x_1 + x_2 + x_3) \cdot (x_1 + x_2 + \bar{x}_3) = x_1 + x_2$.

[Exercise 6] Logic Gates in Logisim

a) Consider the digital logic circuit shown in Figure 12, constructed using Logisim-evolution. The circuit consists of NOT, AND, and OR gates, along with two input pins a and b on the left and one output pin f on the right.

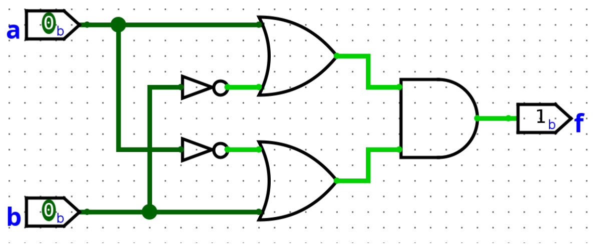


Figure 12: A two-input, one-output logic circuit.

i) Build the same circuit in Logisim-evolution and then use Logisim's "poke" (hand) tool to change the values of the input pins. Subsequently, construct the circuit's truth table by observing and noting down how different combinations of input values affect the output.

ii) Logisim has a library of predefined logic gates, which can be found in the explorer pane on the left, on the `Design` tab, under the `Gates` folder. Which of these gates share the same truth table as the circuit above? Verify whether the truth table matches by interacting with the gate's inputs.

b) In Logisim, construct the digital logic circuits corresponding to the two expressions given below. Then, compare the outputs of the two circuits to determine whether the circuits implement the same function.

Hint: The number of inputs to a gate can be configured from the properties pane at the bottom-left.

Hint: You can connect the same three input pins to both circuits. This way, you only need to change inputs once in order to compare both outputs.

i) $f = (a + b)\, c + \bar{a} b \bar{c}$

ii) $g = ac + \bar{b} c + \bar{a} b \bar{c}$

[Solution 6] Logic Gates in Logisim

a) i) By trying each combination of values for the input pins a and b, and recording the corresponding value of the output pin f, we arrive at the truth table shown in Table 13:

a	b	f
0	0	1
0	1	0
1	0	0
1	1	1

Table 13: A truth table with two inputs and one output.

ii) The same truth table is shared by the XNOR gate shown in Figure 13:

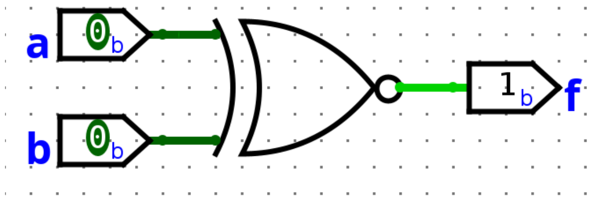


Figure 13: A two-input XNOR logic gate.

b) The two circuits f and g are **not** equivalent. One counterexample is when $a = 0, b = 0, c = 1$. For the full schematic of the circuit, see Figure 14.

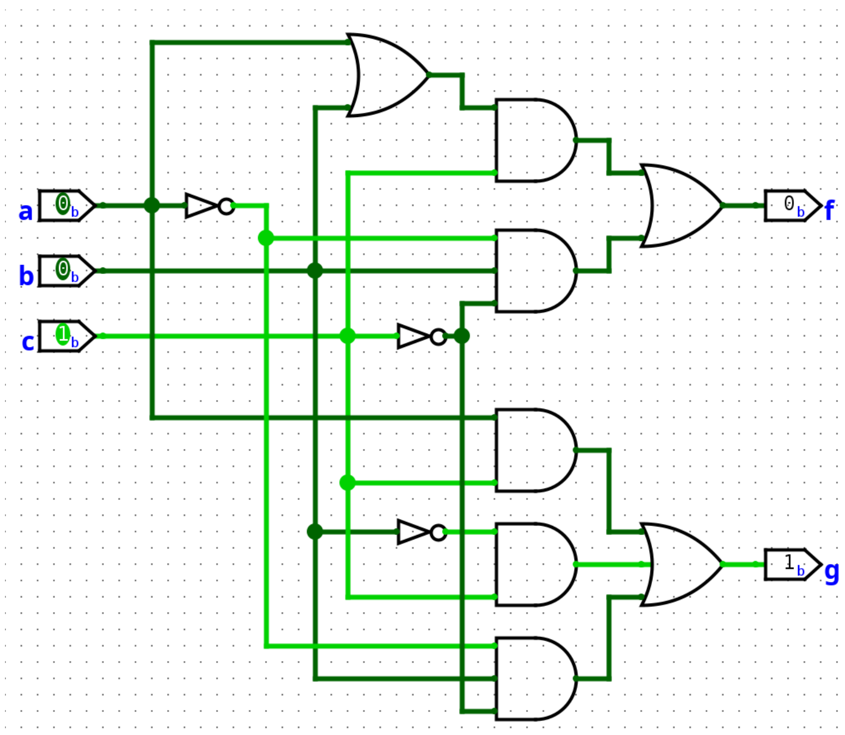


Figure 14: A circuit with three inputs and two outputs.

[Exercise 7] Adders/Subtractors in Logisim-evolution

Note: Before solving this exercise, it will be helpful to watch the online tutorial on *Subcircuits in Logisim-evolution*: [Click on this Link].

a) 1-bit addition

i) Realize a half adder in Logisim-evolution using logic gates (AND, OR, NOT, or XOR). By configuring the inputs to various binary combinations, verify the half-adder's correct functionality.

Suggestion: Create a **subcircuit** (in Logisim-evolution terms) so that you can easily replicate the half adder later.

ii) Realize a full-adder in Logisim-evolution using half adders. By configuring the inputs to various binary combinations, verify the full-adder's correct functionality.

Suggestion: Create a **subcircuit** (in Logisim-evolution terms) so that you can easily replicate the full-adder later.

b) Ripple-Carry Adders

i) Realize a 4-bit ripple-carry adder by replicating the full-adder subcircuit created in question **a-ii** (in Logisim-evolution terms) and connecting them appropriately.

ii) So far, we assumed logic gates take virtually no time to compute the output given the inputs. However, in practice, real gates do take some time to compute the output: we refer to that time as *gate delay*.

Consider the gates have the following delays: $d(\text{NOT}) = 1$ ns, $d(\text{AND}) = 2$ ns, $d(\text{OR}) = 2$ ns, and $d(\text{XOR}) = 3$ ns. Assuming the inputs to be added are available at the same time (e.g., time 0 ns) and that delays of wires connecting the gates can be neglected, compute the worst-case delay of the 4-bit ripple-carry adder, in nanoseconds. If instead of the 4-bit ripple-carry adder you had an 8-bit ripple-carry adder, how would the worst-case delay change?

c) Adders/Subtractors

i) 1-bit addition/subtraction: Design a circuit that can perform 1-bit addition or subtraction in two's complement format (ADDSUB). This circuit should have an additional input op, which controls the type of operation performed as follows:

- $op = 1$: ADDSUB performs addition;
- $op = 0$: ADDSUB performs subtraction.

Hint: You are free to reuse full-adder subcircuit built earlier.

ii) Using the full-adder circuit, realize a 4-bit ripple-carry adder/subtractor and verify its correct operation by applying a few combinations of inputs and setting the control signal op appropriately.

[Solution 7] Adders/Subtractors in Logisim-evolution

a) i) A half-adder circuit is shown in Figure 15.

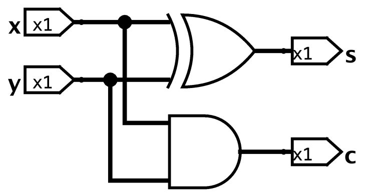


Figure 15: A half-adder circuit.

ii) A full-adder circuit is shown in Figure 16.

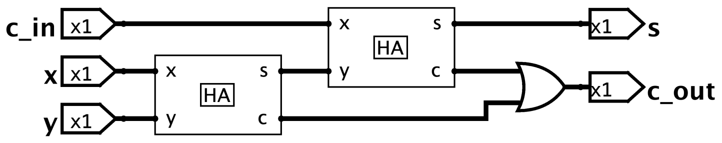


Figure 16: A full-adder circuit.

b) i) A 4-bit ripple-carry adder circuit is shown in Figure 17.

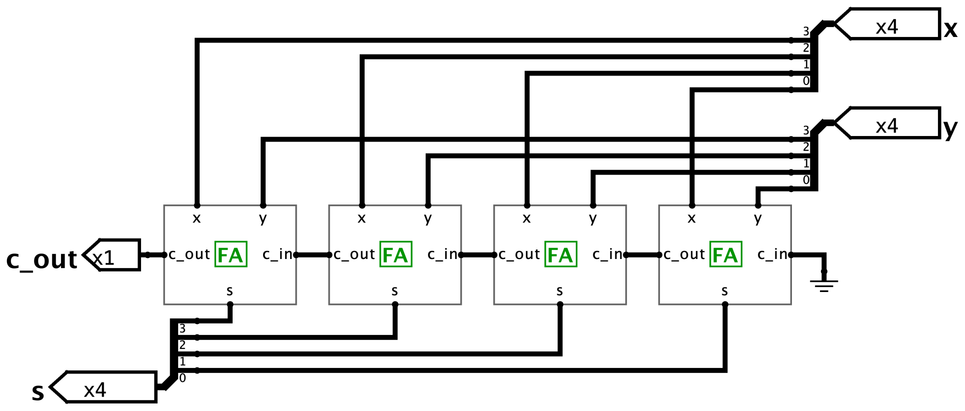


Figure 17: A 4-bit ripple-carry adder circuit.

ii)

$d(\text{NOT}) = 1$ ns, $d(\text{AND}) = 2$ ns, $d(\text{OR}) = 2$ ns, and $d(\text{XOR}) = 3$ ns

Consider the Figure 18, which has the paths listed below:

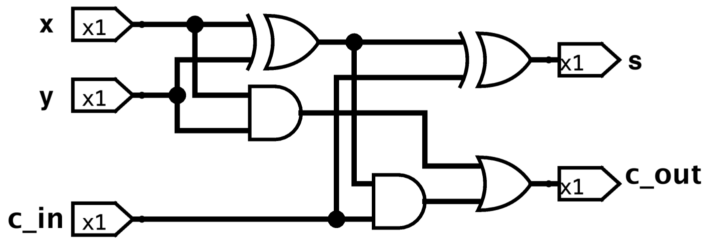


Figure 18: A gate-level view of the full-adder circuit.

Maximum delay between inputs (x and y) and sum (s): $t(x, s) = t(y, s) = 2 \cdot t(XOR) = 6\ ns$

Maximum delay between inputs (x and y) and carry out (c_out): $t(x, c\_out) = t(XOR) + t(AND) + t(OR) = 7\ ns$

Maximum delay between carry input (c_in) and carry out (c_out): $t(c\_in, c\_out) = t(AND) + t(OR) = 4\ ns$

Maximum delay between carry input (c_in) and sum (s): $t(c\_in, s) = t(XOR) = 3\ ns$

The critical path of a full-adder circuit is the path that produces the maximum delay, and in this case it is the path from inputs (x and y) to carry out (c_out) with delay 7 ns.

With the given gate delays, the critical path of the 4-bit ripple carry adder is the path that ripples the carry from the first adder to the last, as shown in Figure 19. The total delay for n-bit ripple-carry adder can be written as $t(x, c\_out) + (n - 2) \cdot t(c\_in, c\_out) + max(t(c\_in, c\_out), t(c\_in, s))$, where the first term represents the delay from first full-adder, second term represents the delay from second adder to second last adder, and the third term represents the delay from the last adder. In our case, delay from first adder is 7 ns, delay from the second adder to the second last adder is $2 \cdot 4\ ns$, and the delay from the last adder 4 ns, which results in a total delay of $7\ ns + (4-2) \cdot 4\ ns + 4\ ns = 19\ ns$.

In case of an 8-bit ripple carry adder, the total critical path delay would be $7\ ns + (8 - 2) \cdot 4\ ns + 4\ ns = 35\ ns$, which is higher than the 4-bit ripple carry adder.

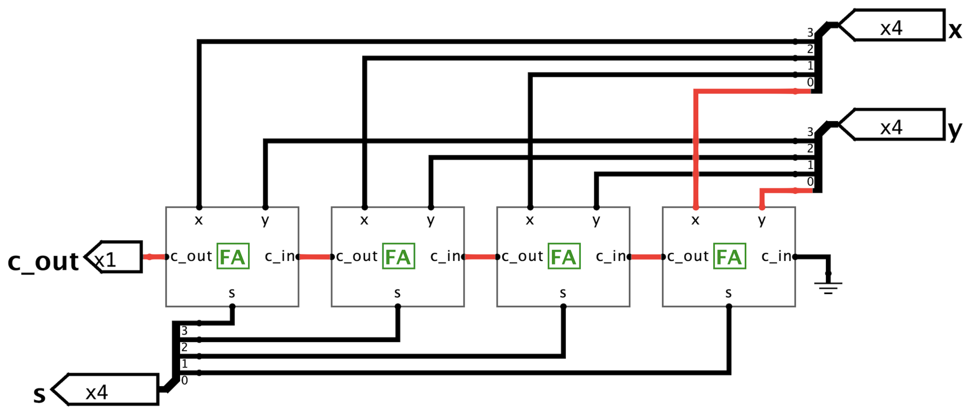


Figure 19: The critical path of the 4-bit ripple adder circuit.

c) i) An ADDSUB circuit is shown in Figure 20.

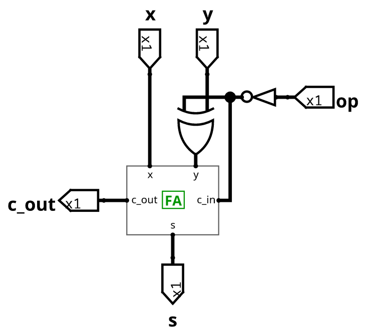


Figure 20: An ADDSUB circuit.

ii) A 4-bit ripple-carry ADDSUB circuit is shown in Figure 21.

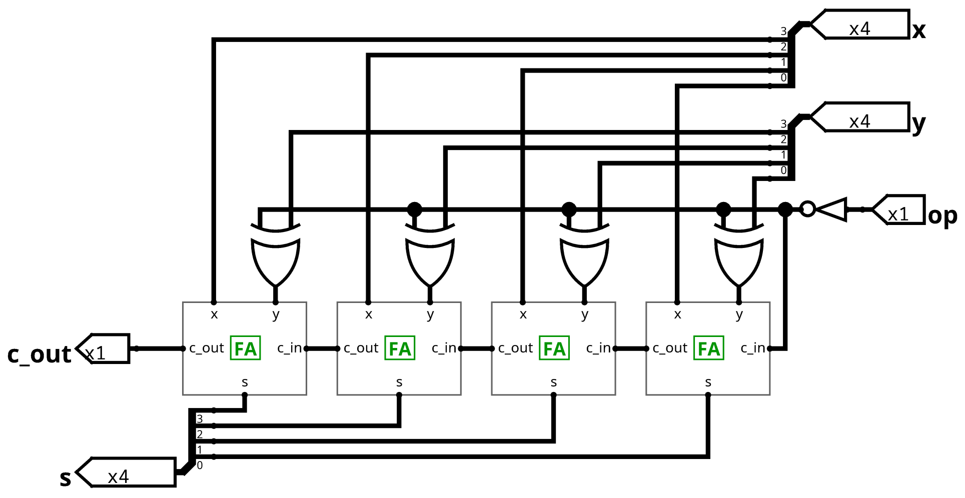


Figure 21: A 4-bit ripple-carry ADDSUB circuit.

[Exercise 8] Sum of Products and Product of Sums

a) Write the logical expression for each of the following truth tables in both canonical SoP and canonical PoS forms.

i)

a	b	c	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

ii)

a	b	c	f
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	1
1	1	1	1

b) Write the logical expression for each of the following algebraic functions in both canonical SoP and canonical PoS forms.

Hint: To convert an algebraic function to its canonical form, start by expanding it using the distributive property and Boolean theorems (e.g., $x \cdot \overline{x} = 0$, $x + \overline{x} = 1$). Moreover, you can convert one canonical form to the other by using the minterms and maxterms.

i) $f = \overline{a}(\overline{b}c + bc + b\overline{c}) + abc$

ii) $f = \overline{a} + a(a + \overline{b})(b + \overline{c})$

c) Design and draw a circuit diagram for each of the logical expressions you found in parts **a** and **b**. Use the simpler one of the SoP and PoS representation while designing the circuit. In other words, use SoP representation if it results in a fewer number of gates than PoS, and vice versa. If both representations require the same number of gates, you can use either version. You can only use **AND**, **OR**, and **NOT** gates with any number of inputs.

[Solution 8] Sum of Products and Product of Sums

a) The simplest way to create the Sum of Products from a truth table is to look at the rows where the output is 1 and write down the product of the literals in those rows, and sum all the product terms. For example, the first row in the truth table given in part **i** has the output 1, and the product of the literals in that row is $\overline{a}\overline{b}\overline{c}$. This term will be present in the Sum of Products.

For the Product of Sums, look at the rows where the output is 0 in a truth table and write down the sum of the inverted literals in those rows, and multiply all the sum terms. For example, the second row in the truth table given in part **i** has the output 0, and the sum of the inverted literals in that row is $a + b + \overline{c}$. This term will be present in the Product of Sums.

i)

Sum of Products: $\overline{a}\overline{b}\overline{c} + \overline{a}b\overline{c} + a\overline{b}c + ab\overline{c}$
Product of Sums: $(a + b + \overline{c})(a + \overline{b} + \overline{c})(\overline{a} + b + c)(\overline{a} + \overline{b} + \overline{c})$

ii)

Sum of Products: $\overline{a}\overline{b}c + ab\overline{c} + abc$
Product of Sums: $(a + b + c)(a + \overline{b} + c)(a + \overline{b} + \overline{c})(\overline{a} + b + c)(\overline{a} + b + \overline{c})$

b) The technique for converting an algebraic function to its sum of products form is to use distributive property to expand the function into a sum of products and remove redundant terms. You might need to expand some terms into their minterms to have the canonical form.

There are multiple ways to get product of sums form. One way is getting the sum of products form and using maxterm indices that are not present in the minterms of sum of products. Another way is using complements and De Morgan's Theorem. The sum of product and product of sums are complementary representations of each other. Therefore, using the complements and De Morgan's theorem, you can convert a sum of products to a product of sums and vice versa. In other words, $f = \overline{(\overline{f})}$ which means if we find the sum of products form of $\overline{f}$ we can convert it to product of sums form by taking the complement and using De Morgan's theorem.

Creating truth tables and using them to find the sum of products and product of sums forms is another way of doing this.

i)

Sum of products:

$$\begin{aligned}
f &= \overline{a}(\overline{b}c + bc + b\overline{c}) + abc \\
&= \overline{a}\overline{b}c + \overline{a}bc + \overline{a}b\overline{c} + abc && \text{(Distributive Property)}
\end{aligned}$$

Product of sums:

The sum of products has the following minterms: m_1, m_3, m_2, m_7. The missing minterms are m_0, m_4, m_5, m_6. Therefore, the product of sums should include the maxterms M_0, M_4, M_5, M_6.
Therefore, the product of sums is $(a + b + c)(\overline{a} + b + c)(\overline{a} + b + \overline{c})(\overline{a} + \overline{b} + c)$

ii)

Sum of products:

$$\begin{aligned}
f &= \overline{a} + a(a + \overline{b})(b + \overline{c}) \\
&= \overline{a} + (aa + a\overline{b})(b + \overline{c}) && \text{(Distributive Property)} \\
&= \overline{a} + (a + a\overline{b})(b + \overline{c}) && (x \cdot x = x) \\
&= \overline{a} + a(b + \overline{c}) && (x + xy = x) \\
&= \overline{a} + ab + a\overline{c} && \text{(Distributive Property)} \\
&= \overline{a}(\overline{b} + b)(\overline{c} + c) + ab(\overline{c} + c) + a(\overline{b} + b)\overline{c} && (x \cdot 1 = x) \\
&= \overline{a}\overline{b}\overline{c} + \overline{a}\overline{b}c + \overline{a}b\overline{c} + \overline{a}bc + ab\overline{c} + abc + a\overline{b}\overline{c} + ab\overline{c} && \text{(Distributive Property)} \\
&= \overline{a}\overline{b}\overline{c} + \overline{a}\overline{b}c + \overline{a}b\overline{c} + \overline{a}bc + ab\overline{c} + abc + a\overline{b}\overline{c} && (x + x = x)
\end{aligned}$$

Product of Sums: We can use the missing minterms in the SoP form to find the maxterms in the PoS form. The only missing minterm is m_5. Therefore, the product of sums should include the maxterm M_5. Therefore, the product of sums is $\overline{a} + b + \overline{c}$. Or you can use the algebraic manipulations to get PoS form as shown below.

$$
\begin{aligned}
f &= \overline{a} + a(a+\overline{b})(b+\overline{c}) && \\
\overline{f} &= \overline{\overline{a} + a(a+\overline{b})(b+\overline{c})} && \text{(Inverting Both Sides)}\\
\overline{f} &= a \cdot (\overline{a} + \overline{(a+\overline{b})} + \overline{(b+\overline{c})}) && \text{(De Morgan's Theorem)}\\
\overline{f} &= a \cdot (\overline{a} + \overline{a}b + \overline{b}c) && \text{(De Morgan's Theorem)}\\
\overline{f} &= a \cdot (\overline{a} + \overline{b}c) && (x + xy = x)\\
\overline{f} &= a\overline{a} + a\overline{b}c && \text{(Distributive Property)}\\
\overline{f} &= 0 + a\overline{b}c && (x \cdot \overline{x} = 0)\\
\overline{f} &= a\overline{b}c && (x + 0 = x)\\
f &= \overline{(a\overline{b}c)} && \text{(Inverting Both Sides)}\\
f &= \overline{a} + b + \overline{c} && \text{(De Morgan's Theorem)}
\end{aligned}
$$

c)

i) Circuit for question **a**, subquestion **i**. The SoP and PoS circuits require the same number of gates for the given truth table, so we can use both of the forms. We choose to use the SoP form. We are using an **AND** gate for each minterm and an **OR** gate to combine the minterms. The sum of products from is $\overline{a}\overline{b}\overline{c} + \overline{a}b\overline{c} + a\overline{b}c + ab\overline{c}$. The circuit is given below. To simplify the visualization, all inputs of **AND** gates are ordered as a, b, c and the wires are colored red if they are complemented. The circuit is given in Figure 22.

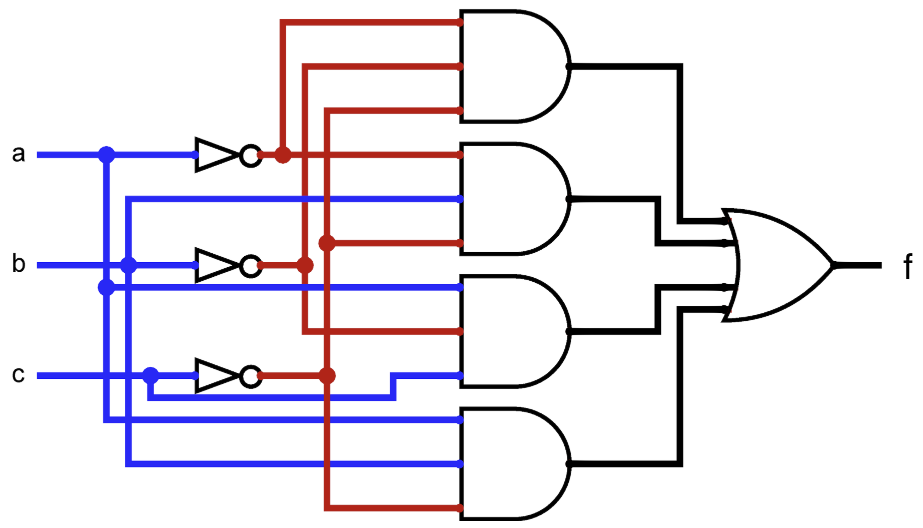


Figure 22: Circuit for question **a**, subquestion **i**.

ii) Circuit for question **a**, subquestion **ii**. Following the same reasoning in the previous question, we find the SoP form to be the smaller circuit, so we will draw the circuit for $\overline{a}\overline{b}c + ab\overline{c} + abc$. The circuit is given in Figure 23.

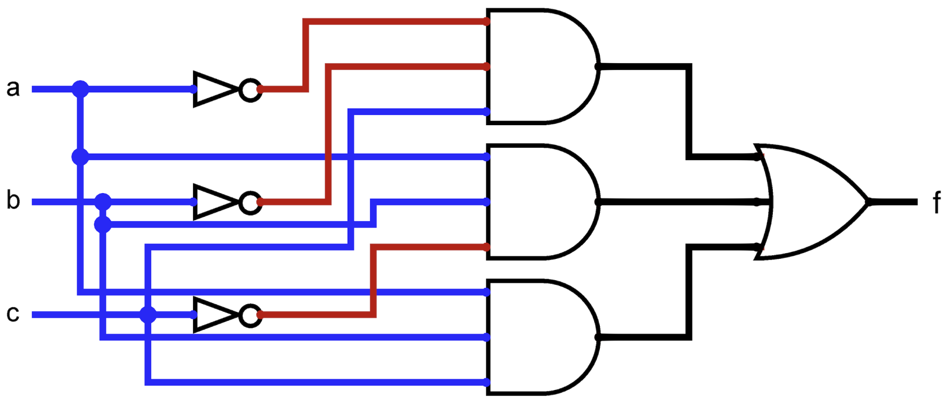


Figure 23: Circuit for question **a**, subquestion **ii**.

iii) Circuit for question **b**, subquestion **i**. The sum of products and product of sums have the same number of gates, so we can use both of the forms. We choose to use the sum of products form, i.e., $\overline{a}\overline{b}c + \overline{a}bc + \overline{a}b\overline{c} + abc$. The circuit is given in Figure 24.

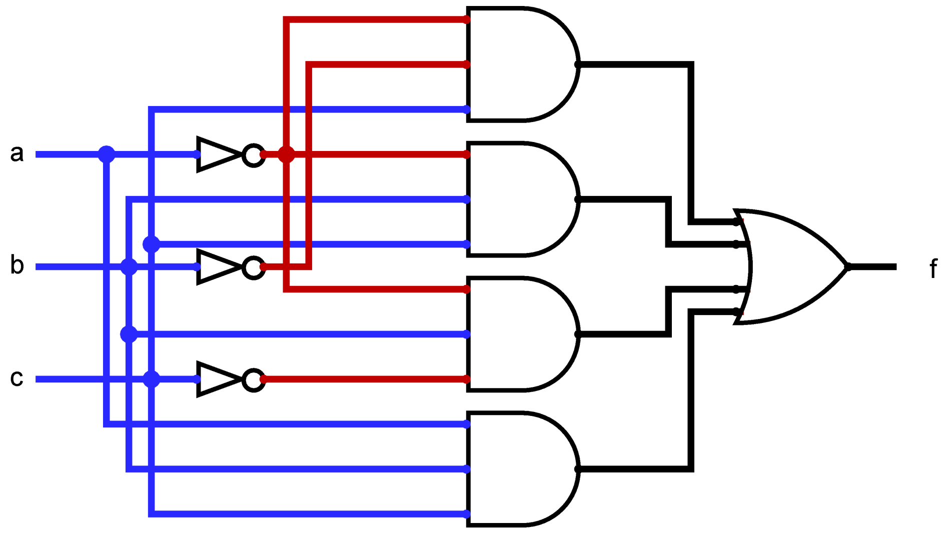


Figure 24: Circuit for question **b**, subquestion **i**.

iv) Circuit for question **b**, subquestion **ii**. The PoS version have the smaller number of gates, so we will use the that form, i.e., $\overline{a} + b + \overline{c}$. The circuit is given in Figure 25.

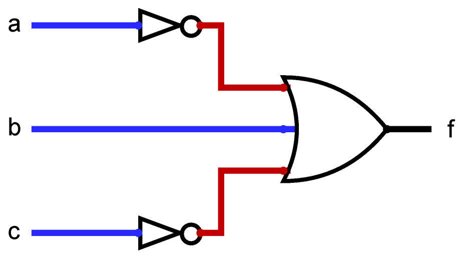


Figure 25: Circuit for question **b**, subquestion **ii**.

[Exercise 9] Canonical Form Comparison

The canonical forms of a logical expression is a unique representation of the expression. That means we can check and compare the equality of two expressions by comparing their canonical forms.

Using canonical SoP or PoS form, determine whether or not the following expressions are valid. In other words, determine whether the left- and right-hand sides represent the same function.

<u>Hint:</u> To convert an algebraic function to its canonical form, start by expanding it using the distributive property and Boolean theorems (e.g., $x \cdot \overline{x} = 0$, $x + \overline{x} = 1$).

a) $\overline{a}\,\overline{c} + bc + a\overline{b} = \overline{a}b + ac + \overline{b}\overline{c}$

b) $\overline{a}c + ab\overline{c} + \overline{a}b + a\overline{b} = \overline{b}c + a\overline{c} + b\overline{c} + \overline{a}bc$

c) $a\overline{c} + bc + \overline{b}\overline{c} = \left(a + \overline{b} + c\right)(a + b + \overline{c})(\overline{a} + b + \overline{c})$

d) $(a + c)\left(\overline{a} + \overline{b} + \overline{c}\right)(\overline{a} + b) = (a + b)(b + c)(\overline{a} + \overline{c})$

[Solution 9] Canonical Form Comparison

a) The equation is valid if the expressions on the left- and right-hand sides represent the same function. To perform the comparison, we could construct a truth table for each side and see if the truth tables are the same. An algebraic approach is to derive a canonical sum of products form for each expression and compare them.

$$\begin{aligned}
\text{LHS} &= \overline{a}\,\overline{c} + bc + a\overline{b} \\
&= \overline{a}\left(b + \overline{b}\right)\overline{c} + \left(a + \overline{a}\right)bc + a\overline{b}\left(c + \overline{c}\right) && (x \cdot 1 = x) \\
&= \overline{a}b\overline{c} + \overline{a}\overline{b}\overline{c} + abc + \overline{a}bc + a\overline{b}c + a\overline{b}\overline{c} && \text{(Distributive Property)}
\end{aligned}$$

These product terms represent the minterms $2, 0, 7, 3, 5$, and 4, respectively. For the right-hand side we have

$$\begin{aligned}
\text{RHS} &= \overline{a}b + ac + \overline{b}\overline{c} \\
&= \overline{a}b\left(c + \overline{c}\right) + a\left(b + \overline{b}\right)c + \left(a + \overline{a}\right)\overline{b}\overline{c} && (x \cdot 1 = x) \\
&= \overline{a}bc + \overline{a}b\overline{c} + abc + a\overline{b}c + a\overline{b}\overline{c} + \overline{a}\overline{b}\overline{c} && \text{(Distributive Property)}
\end{aligned}$$

These product terms represent the minterms 3, 2, 7, 5, 4, and 0, respectively. Since both expressions specify the same minterms, they represent the same function; therefore, the equation is valid. Another way of representing this function is by $\sum m(0, 2, 3, 4, 5, 7)$.

b)

$$\begin{aligned}
\text{LHS} &= \overline{a}c + ab\overline{c} + \overline{a}b + a\overline{b} \\
&= \overline{a}(b+\overline{b})c + ab\overline{c} + \overline{a}b(c+\overline{c}) + a\overline{b}(c+\overline{c}) && (x \cdot 1 = x) \\
&= \overline{a}bc + \overline{a}\overline{b}c + ab\overline{c} + \overline{a}bc + \overline{a}b\overline{c} + a\overline{b}c + a\overline{b}\overline{c} && \text{(Distributive Property)} \\
&= \overline{a}bc + \overline{a}\overline{b}c + ab\overline{c} + \overline{a}b\overline{c} + a\overline{b}c + a\overline{b}\overline{c} && (x + x = x) \\
&= \sum m(1,2,3,4,5,6) \\
\text{RHS} &= \overline{b}c + a\overline{c} + b\overline{c} + \overline{a}bc \\
&= (a+\overline{a})\overline{b}c + a(b+\overline{b})\overline{c} + (a+\overline{a})b\overline{c} + \overline{a}bc && (x \cdot 1 = x) \\
&= a\overline{b}c + \overline{a}\overline{b}c + ab\overline{c} + a\overline{b}\overline{c} + ab\overline{c} + \overline{a}b\overline{c} + \overline{a}bc && \text{(Distributive Property)} \\
&= a\overline{b}c + \overline{a}\overline{b}c + ab\overline{c} + a\overline{b}\overline{c} + \overline{a}b\overline{c} + \overline{a}bc && (x + x = x) \\
&= \sum m(1,2,3,4,5,6)
\end{aligned}$$

Both expressions have the same minterms, so they represent the same function. Therefore, the equation is valid.

c)

$$\begin{aligned}
\text{LHS} &= a\overline{c} + bc + \overline{b}\overline{c} \\
&= a(b+\overline{b})\overline{c} + (a+\overline{a})bc + (a+\overline{a})\overline{b}\overline{c} && (x \cdot 1 = x) \\
&= ab\overline{c} + a\overline{b}\overline{c} + abc + \overline{a}bc + a\overline{b}\overline{c} + \overline{a}\overline{b}\overline{c} && \text{(Distributive Property)} \\
&= ab\overline{c} + a\overline{b}\overline{c} + abc + \overline{a}bc + \overline{a}\overline{b}\overline{c} && (x + x = x) \\
&= \sum m(0,3,4,6,7) \\
\text{RHS} &= (a+\overline{b}+c)(a+b+\overline{c})(\overline{a}+b+\overline{c}) \\
&= \prod M(1,2,5) && \text{(PoS Form)} \\
&= \sum m(0,3,4,6,7)
\end{aligned}$$

Both expressions have the same minterms, so they represent the same function. Therefore, the equation is valid.

d)

$$
\begin{aligned}
\text{LHS} &= (a+c)(\overline{a}+\overline{b}+\overline{c})(\overline{a}+b) \\
&= (a+b\overline{b}+c)(\overline{a}+\overline{b}+\overline{c})(\overline{a}+b+c\overline{c}) && (x+0=x) \\
&= ((a+b)(a+\overline{b})+c)(\overline{a}+\overline{b}+\overline{c})(\overline{a}+(b+c)(b+\overline{c})) && \text{(Distributive Property)} \\
&= (a+b+c)(a+\overline{b}+c)(\overline{a}+\overline{b}+\overline{c})(\overline{a}+b+c)(\overline{a}+b+\overline{c}) && \text{(Distributive Property)} \\
&= \prod M(0,2,4,5,7) \\
\text{RHS} &= (a+b)(b+c)(\overline{a}+\overline{c}) \\
&= (a+b+c\overline{c})(a\overline{a}+b+c)(\overline{a}+b\overline{b}+\overline{c}) && (x+0=x) \\
&= (a+(b+c)(b+\overline{c}))((a+b)(\overline{a}+b)+c)((\overline{a}+b)(\overline{a}+\overline{b})+\overline{c}) \\
& && \text{(Distributive Property)} \\
&= (a+b+c)(a+b+\overline{c})(a+b+c)(\overline{a}+b+c)(\overline{a}+b+\overline{c})(\overline{a}+\overline{b}+\overline{c}) \\
& && \text{(Distributive Property)} \\
&= (a+b+c)(a+b+\overline{c})(\overline{a}+b+c)(\overline{a}+b+\overline{c})(\overline{a}+\overline{b}+\overline{c}) && (x\cdot x=x) \\
&= \prod M(0,1,4,5,7)
\end{aligned}
$$

The maxterms for the left-hand side and right-hand side are different, so the equation is not valid.

[Exercise 10] Design with NAND and NOR gates

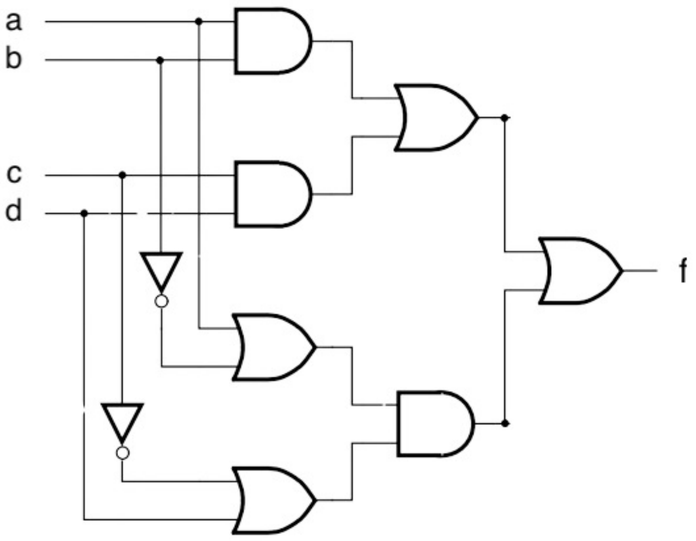
NAND and NOR gates are called universal gates because they can be used to implement any Boolean function. You can refer to the Wikipedia page for NAND logic [link] to see how some gates can be formed using only NAND gates. And similarly for the NOR logic [link].

a) Design and draw circuit diagrams for the following functions using only NAND gates with any number of inputs. Please note that there could be many solutions and we do not require a minimum gate solution.

i)

a	b	c	f
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

ii)



iii) $f = \overline{a}bc + a\overline{b}\overline{c} + ab\overline{c} + abc$

b) Repeat the previous question using only NOR gates with any number of inputs.

[Solution 10] Design with NAND and NOR gates

a) You can implement this function using **AND**, **OR**, and **NOT** gates and then convert it into **NAND** gates using the transformations in Figure 26.

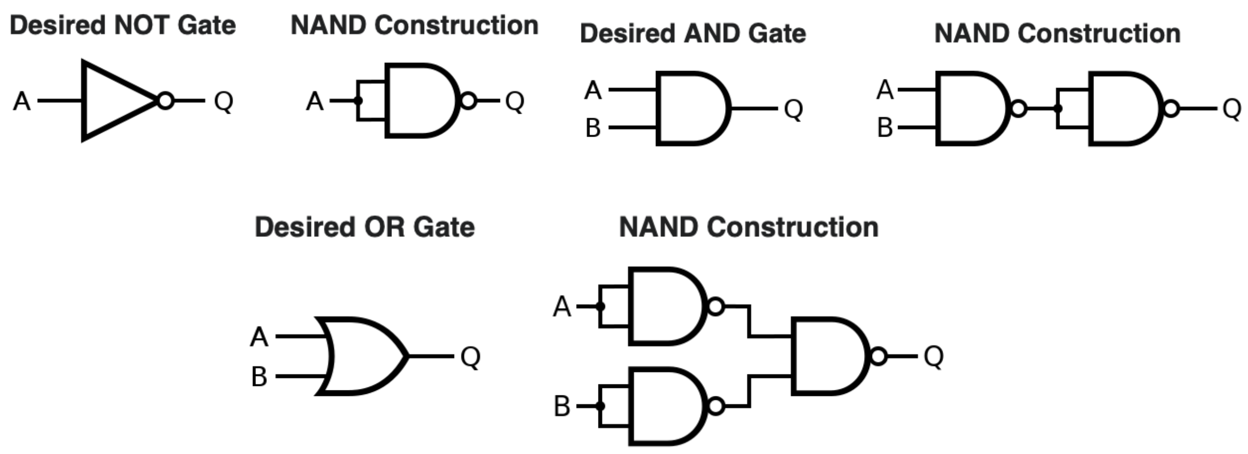


Figure 26: The transformations from **NOT**, **AND**, and **OR** gates to **NAND** gates.

You can either use these visual transformations or use the algebraic transformations to convert the given function into a circuit that only has **NAND** gates.

i) The algebraic method is shown below.

$$
\begin{aligned}
f &= \sum m(1,2,4,7) \\
&= \overline{a}\overline{b}c + \overline{a}b\overline{c} + a\overline{b}\overline{c} + abc \\
&= \overline{\overline{(\overline{a}\overline{b}c + \overline{a}b\overline{c} + a\overline{b}\overline{c} + abc)}} && (\overline{\overline{x}} = x) \\
&= \overline{\overline{(\overline{a}\overline{b}c)} \cdot \overline{(\overline{a}b\overline{c})} \cdot \overline{(a\overline{b}\overline{c})} \cdot \overline{(abc)}} && \text{(DeMorgan's Theorem)}
\end{aligned}
$$

Now, we show the visual method. The function is $f = \bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}\bar{c} + abc$. A simple circuit for this function is given in Figure 27.

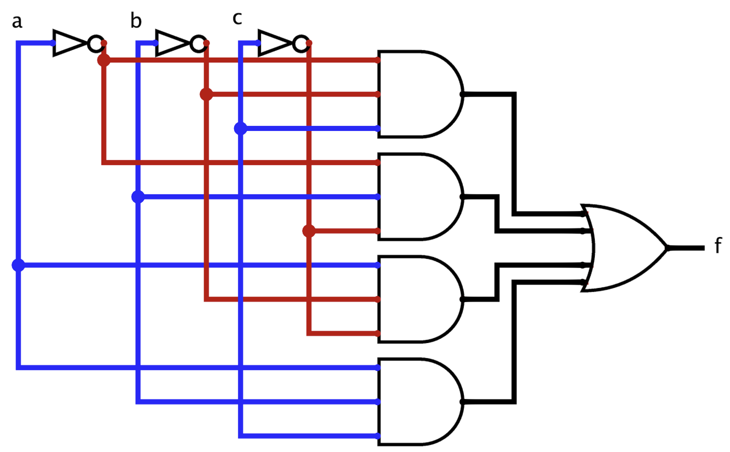


Figure 27: The circuit for the function $f = \bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}\bar{c} + abc$.

Then, we can use the **OR** transformation to eliminate the **OR** gate and get the circuit in Figure28.

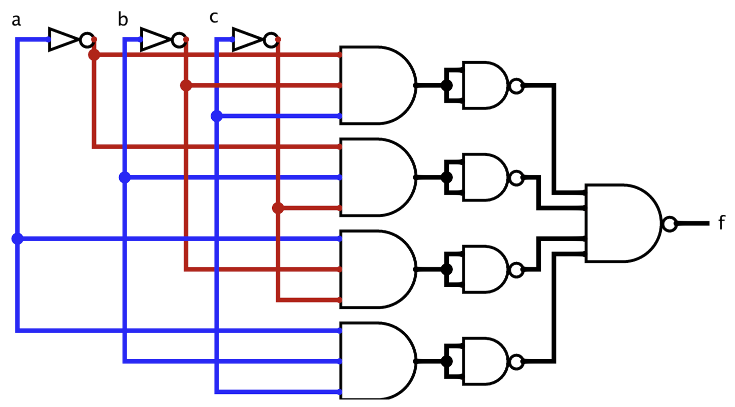


Figure 28: The circuit diagram after replacing the last **OR** gate.

Finally, we can merge **AND** gates with the **NOT** gates, and convert **NOT** gates into **NAND** gates to get the following circuit that only has **NAND** gates. The final circuit is shown in Figure 29.

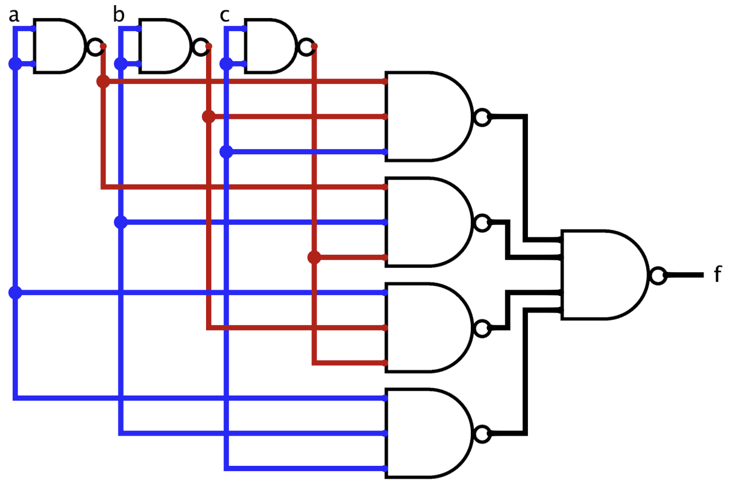


Figure 29: The final circuit diagram for function $f = \overline{a}\overline{b}c + \overline{a}b\overline{c} + a\overline{b}\overline{c} + abc$.

ii) The function $f = (ab + cd) + ((a + \overline{b})(\overline{c} + d))$ can be construed from the circuit. The algebraic method to obtain NAND only expression is shown below.

$$
\begin{aligned}
f &= (ab + cd) + ((a + \overline{b}) \cdot (\overline{c} + d)) \\
&= \overline{\overline{(ab + cd) + ((a + \overline{b}) \cdot (\overline{c} + d))}} && (\overline{\overline{x}} = x) \\
&= \overline{\overline{(ab + cd)} \cdot \overline{((a + \overline{b}) \cdot (\overline{c} + d))}} && \text{(DeMorgan's Theorem)} \\
&= \overline{\overline{(ab + cd)} \cdot \overline{(\overline{\overline{(a + \overline{b})}} \cdot \overline{\overline{(\overline{c} + d)}})}} && (\overline{\overline{x}} = x) \\
&= \overline{(\overline{(ab)} \cdot \overline{(cd)}) \cdot \overline{\overline{(\overline{a}b)} \cdot \overline{(c\overline{d})}}} && \text{(DeMorgan's Theorem)}
\end{aligned}
$$

The final circuit with only NAND gates is shown in Figure 30.

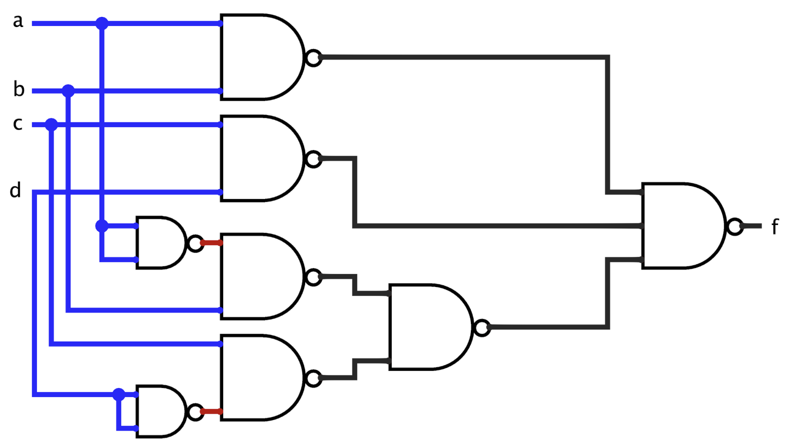


Figure 30: Circuit diagram for $f = ab + cd + (a + \overline{b})(\overline{c} + d)$ using only **NAND** gates.

iii) The algebraic method to obtain NAND only expression is shown below.

$$\begin{aligned}
f &= \overline{a}bc + a\overline{b}\overline{c} + ab\overline{c} + abc \\
&= \overline{\overline{\overline{a}bc + a\overline{b}\overline{c} + ab\overline{c} + abc}} && (\overline{\overline{x}} = x) \\
&= \overline{\overline{(\overline{a}bc)} \cdot \overline{(a\overline{b}\overline{c})} \cdot \overline{(ab\overline{c})} \cdot \overline{(abc)}} && \text{(DeMorgan's Theorem)}
\end{aligned}$$

The final circuit is shown in Figure 31.

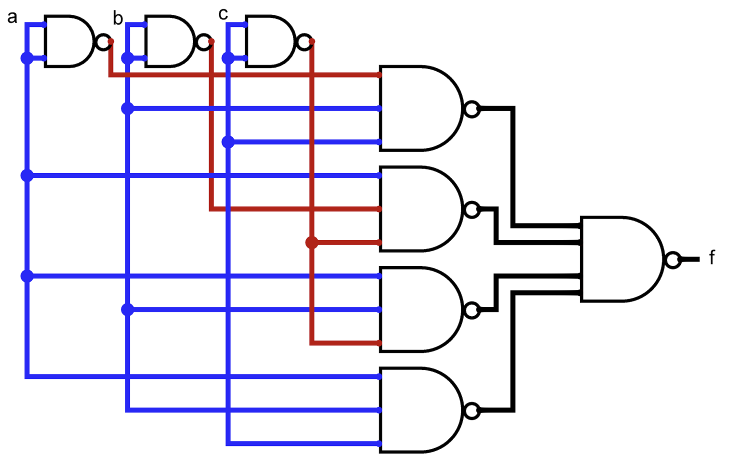


Figure 31: Circuit diagram for $f = \overline{a}bc + a\overline{b}\overline{c} + ab\overline{c} + abc$ using only **NAND** gates.

b) Similarly, we will use the transformations in Figure32 to convert the given function into a circuit that only has **NOR** gates. Please note that these circuits are not unique. There are multiple ways to implement the same function using **NOR** gates.

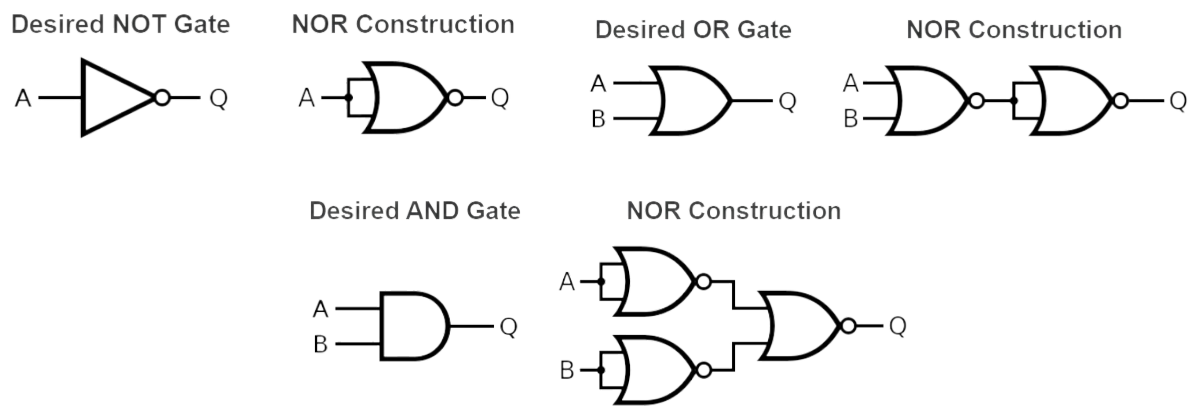


Figure 32: Transformations from **NOT**, **OR**, and **AND** gates to **NOR** gates.

i) Using product of sums form is easier for **NOR** gates. That is why we will use PoS representation to start with. The algebraic method to obtain NOR only expression is shown below.

$$
\begin{aligned}
f &= \prod M(0,3,5,6) \\
f &= (a+b+c)(a+\overline{b}+\overline{c})(\overline{a}+b+\overline{c})(\overline{a}+\overline{b}+c) \\
f &= \overline{\overline{(a+b+c)(a+\overline{b}+\overline{c})(\overline{a}+b+\overline{c})(\overline{a}+\overline{b}+c)}} && (\overline{\overline{x}} = x) \\
f &= \overline{\overline{(a+b+c)}+\overline{(a+\overline{b}+\overline{c})}+\overline{(\overline{a}+b+\overline{c})}+\overline{(\overline{a}+\overline{b}+c)}} && \text{(DeMorgan's Theorem)}
\end{aligned}
$$

The final circuit is shown in Figure 33.

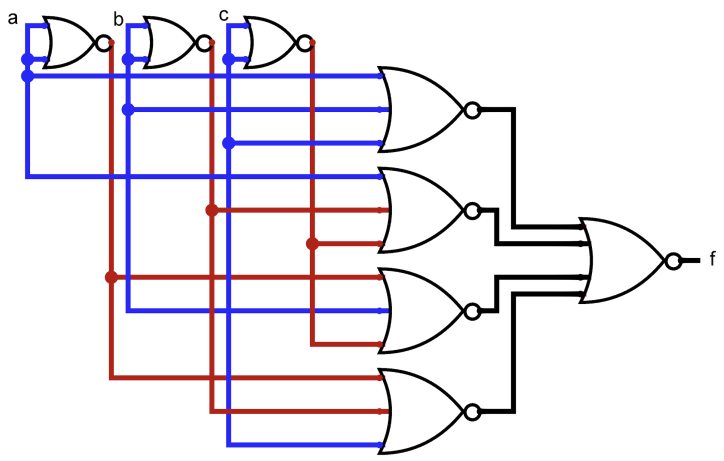


Figure 33: The circuit diagram for $f = (a+b+c)(a+\overline{b}+\overline{c})(\overline{a}+b+\overline{c})(\overline{a}+\overline{b}+c)$ using only **NOR** gates.

ii) The function is $f = (ab+cd)+((a+\overline{b})(\overline{c}+d))$. The algebraic transformations to convert the given function into a circuit that only has **NOR** gates is shown below.

$$
\begin{aligned}
f &= (ab+cd)+((a+\overline{b})(\overline{c}+d)) \\
f &= (\overline{\overline{ab}}+\overline{\overline{cd}})+\overline{\overline{((a+\overline{b})(\overline{c}+d))}} && (\overline{\overline{x}}=x) \\
f &= ((\overline{(\overline{a}+\overline{b})}+\overline{(\overline{c}+\overline{d})}))+\overline{(\overline{(a+\overline{b})}+\overline{(\overline{c}+d)})} && \text{(DeMorgan's Theorem)} \\
f &= \overline{\overline{((\overline{(\overline{a}+\overline{b})}+\overline{(\overline{c}+\overline{d})}))+\overline{(\overline{(a+\overline{b})}+\overline{(\overline{c}+d)})}}} && (\overline{\overline{x}}=x)
\end{aligned}
$$

The final circuit using only NOR gates is shown in Figure 34.

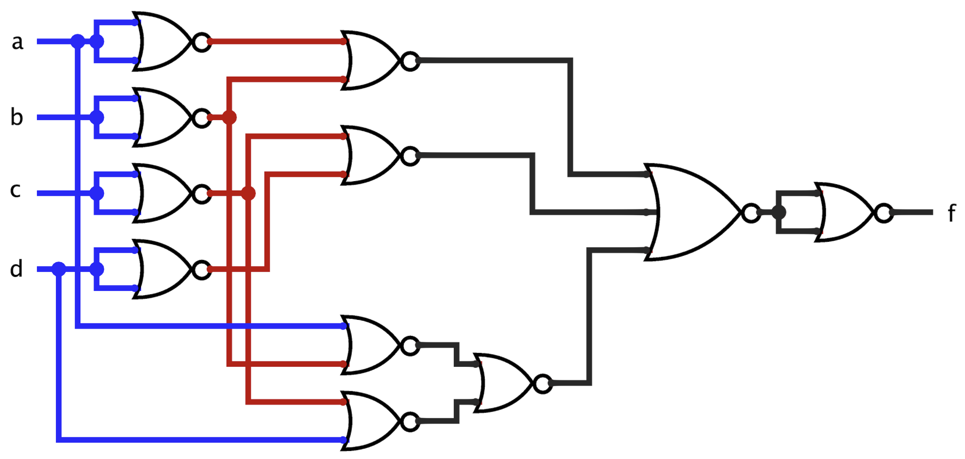


Figure 34: Circuit diagram for the circuit in question **a** subquestion **ii** using only **NOR** gates.

iii) Again, to simplify the process, we will use the PoS form. The function is $f = \overline{a}bc + a\overline{b}\overline{c} + ab\overline{c} + abc$. The algebraic method is shown below.

$$
\begin{aligned}
f &= \overline{a}bc + a\overline{b}\overline{c} + ab\overline{c} + abc \\
&= \sum m(3,4,6,7) \\
&= \prod M(0,1,2,5) \\
&= (\overline{a}+b+\overline{c})(a+\overline{b}+c)(a+b+\overline{c})(a+b+c) \\
&= \overline{\overline{(\overline{a}+b+\overline{c})(a+\overline{b}+c)(a+b+\overline{c})(a+b+c)}} && (\overline{\overline{x}} = x) \\
&= \overline{\overline{(\overline{a}+b+\overline{c})}+\overline{(a+\overline{b}+c)}+\overline{(a+b+\overline{c})}+\overline{(a+b+c)}} && \text{(DeMorgan's Theorem)}
\end{aligned}
$$

And the final circuit using only NOR gates is shown in Figure 35.

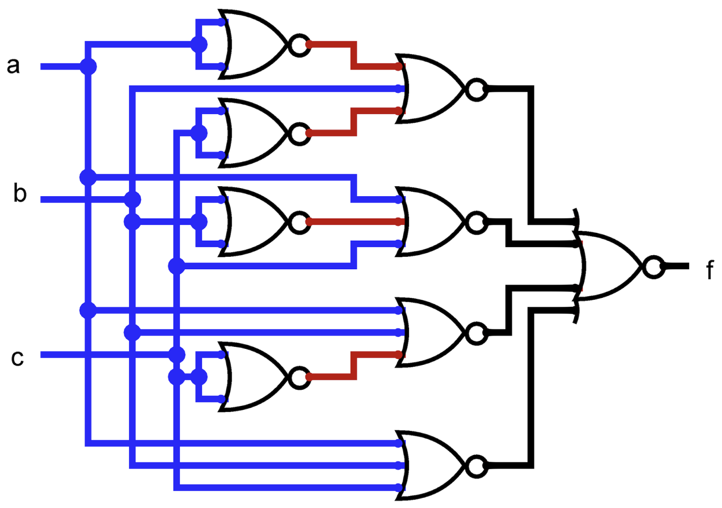


Figure 35: Circuit diagram for $f = \overline{a}bc + a\overline{b}\overline{c} + ab\overline{c} + abc$ using only **NOR** gates.

[Exercise 11] 7-Segment Display

Consider a circuit that drives a 7-segment display shown in Fig. 36. Its operation can be described as follows:

- The circuit has 4 inputs (W, X, Y, and Z) and 7 outputs (a, b, c, d, e, f, and g).
- Each of the outputs controls one segment of the display. The segment is switched on (i.e., it is lit) if the signal controlling it is high (i.e., logic 1). Otherwise, the segment is switched off.
- The display shows the decimal digit corresponding to the binary number formed by the inputs WXYZ. For example, if the inputs are WXYZ = $(0110)_2$, the display shows the decimal digit 6. When the inputs take a combination higher than $(1001)_2$ (i.e., 9_{10}), the output can be considered a *don't care condition* and, thus, be arbitrarily defined. Fig. 37 visualizes how decimal digits are shown on the display.

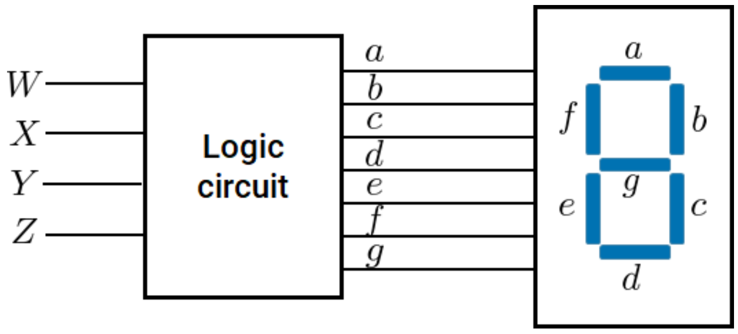


Figure 36: Seven-segment display.

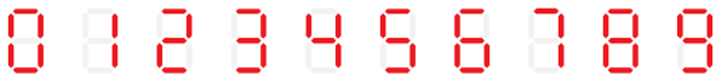


Figure 37: Decimal digits shown on the display.

a) Assume now that, when the inputs take a combination higher than $(1001)_2$ (i.e., 9_{10}), the display shows nothing: all segments of the display are switched off.

i) Give the truth table for the digital logic circuit.

ii) The minimized (i.e., optimized) SoP forms for all seven outputs are given below. Starting from the canonical SoP forms for outputs b and c, perform Boolean algebra transformations to reduce the complexity of your expressions until you reach the minimized SoP forms.

Optional: You are encouraged to perform Boolean algebra transformations on the canonical SoP forms describing other outputs, and design and test your circuit in Logisim-evolution.

$$
\begin{aligned}
a &= W\overline{X}\,\overline{Y} + \overline{W}XZ + \overline{W}\,\overline{X}\,\overline{Z} + \overline{W}Y \\
b &= \overline{W}\,\overline{X} + \overline{W}YZ + \overline{W}\,\overline{Y}\,\overline{Z} + \overline{X}\,\overline{Y} \\
c &= \overline{W}X + \overline{W}Z + \overline{X}\,\overline{Y} \\
d &= W\overline{X}\,\overline{Y} + \overline{W}X\overline{Y}Z + \overline{W}\,\overline{X}Y + \overline{W}\,\overline{X}\,\overline{Z} + \overline{W}Y\overline{Z} \\
e &= \overline{W}Y\overline{Z} + \overline{X}\,\overline{Y}\,\overline{Z} \\
f &= W\overline{X}\,\overline{Y} + \overline{W}X\overline{Y} + \overline{W}X\overline{Z} + \overline{W}\,\overline{Y}\,\overline{Z} \\
g &= W\overline{X}\,\overline{Y} + \overline{W}X\overline{Y} + \overline{W}X\overline{Z} + \overline{W}\,\overline{X}Y
\end{aligned}
$$

b) Assume now that the input combinations higher than $(1001)_2$ (i.e., 9_{10}) generate outputs listed in Table 14.

Table 14: Alternative output signals for input combinations higher than $(1001)_2$.

W	X	Y	Z	a	b	c	d	e	f	g
1	0	1	0	1	1	0	1	1	1	1
1	0	1	1	1	1	1	1	0	1	1
1	1	0	0	1	1	1	1	0	1	1
1	1	0	1	1	0	1	1	0	1	1
1	1	1	0	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1	0	1	1

i) Give the truth table for the digital logic circuit.

ii) The minimized SoP forms for all seven outputs are given below. Starting from the canonical SoP forms for outputs b and c, perform Boolean algebra transformations to reduce the complexity of your expressions until you reach the minimized SoP forms.

Optional: You are encouraged to perform Boolean algebra transformations on the canonical SoP forms describing other outputs as well.

$$
\begin{aligned}
a &= \overline{X}\,\overline{Z} + Y + XZ + W \\
b &= \overline{X} + \overline{Y}\,\overline{Z} + YZ \\
c &= \overline{Y} + Z + X \\
d &= \overline{X}\,\overline{Z} + \overline{X}Y + Y\overline{Z} + X\overline{Y}Z + W \\
e &= \overline{X}\,\overline{Z} + Y\overline{Z} \\
f &= \overline{Y}\,\overline{Z} + X\overline{Y} + X\overline{Z} + W \\
g &= \overline{X}Y + X\overline{Y} + X\overline{Z} + W
\end{aligned}
$$

iii) Compare the complexity (e.g., number of gates, number of gate inputs) of the digital circuits corresponding to the minimized SoP forms in this and the previous question. Discuss the reasons behind the differences.

iv) In Logisim-evolution, implement the digital logic circuit that controls the 7-segment display and test it. You can make use of the 7-segment display module inside Logisim-evolution.

[Solution 11] 7-Segment Display

a) i) The truth table for the circuit described in the question is given in Table 15.

Table 15: Truth table for 7-segment display

W	X	Y	Z	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

ii) We will start from the canonical SoP forms and then apply Boolean algebra transformations to simplify the logic expressions.

$$
\begin{aligned}
b &= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}\,\overline{Y}Z + \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z \\
&\qquad\qquad (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z \qquad (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y} \qquad (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X} + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y} \qquad (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X} + \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y} \qquad (x = x + xy) \\
&= \overline{W}\,\overline{X} + \overline{W}\,\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y} \qquad (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X} + \overline{W}\,\overline{X}YZ + \overline{W}\,\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y} \qquad (x = x + xy) \\
&= \overline{W}\,\overline{X} + \overline{W}\,\overline{Y}\,\overline{Z} + \overline{W}YZ + W\overline{X}\,\overline{Y} \qquad (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X} + \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{Y}\,\overline{Z} + \overline{W}YZ + W\overline{X}\,\overline{Y} \qquad (x = x + xy) \\
&= \overline{W}\,\overline{X} + \overline{W}\,\overline{Y}\,\overline{Z} + \overline{W}YZ + \overline{X}\,\overline{Y} \qquad (xy + x\overline{y} = x)
\end{aligned}
$$

$$
\begin{aligned}
c &= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}\,\overline{Y}Z + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y}\,\overline{Z} \\
&\quad + W\overline{X}\,\overline{Y}Z \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z \\
&\qquad\qquad (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y} + \overline{W}XY\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z \\
&\qquad\qquad (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y} + \overline{W}XY + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z \qquad (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y} + \overline{W}XY + W\overline{X}\,\overline{Y} \qquad (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}YZ + \overline{W}X + W\overline{X}\,\overline{Y} \qquad (xy + x\overline{y} = x) \\
&= \overline{X}\,\overline{Y} + \overline{W}\,\overline{X}YZ + \overline{W}X \qquad (xy + x\overline{y} = x) \\
&= \overline{X}\,\overline{Y} + \overline{W}\,\overline{X}\,\overline{Y}Z + \overline{W}\,\overline{X}YZ + \overline{W}X \qquad (x = x + xy) \\
&= \overline{X}\,\overline{Y} + \overline{W}\,\overline{X}Z + \overline{W}X \qquad (xy + x\overline{y} = x) \\
&= \overline{X}\,\overline{Y} + \overline{W}\,\overline{X}Z + \overline{W}X + \overline{W}XZ \qquad (x = x + xy) \\
&= \overline{X}\,\overline{Y} + \overline{W}Z + \overline{W}X \qquad (xy + x\overline{y} = x)
\end{aligned}
$$

b) **i)** The truth table for the circuit described in the question is given in Table 16.

Table 16: Truth table for 7-segment display

W	X	Y	Z	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	1	1	0	1	1	1	1
1	0	1	1	1	1	1	1	0	1	1
1	1	0	0	1	1	1	1	0	1	1
1	1	0	1	1	0	1	1	0	1	1
1	1	1	0	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1	0	1	1

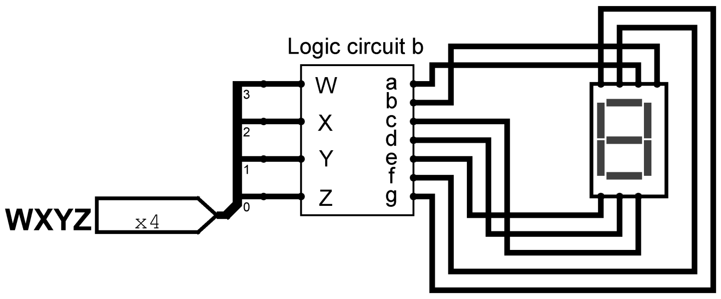
ii) We will start from the canonical SoP forms and then apply Boolean algebra transformations to simplify the logic expressions.

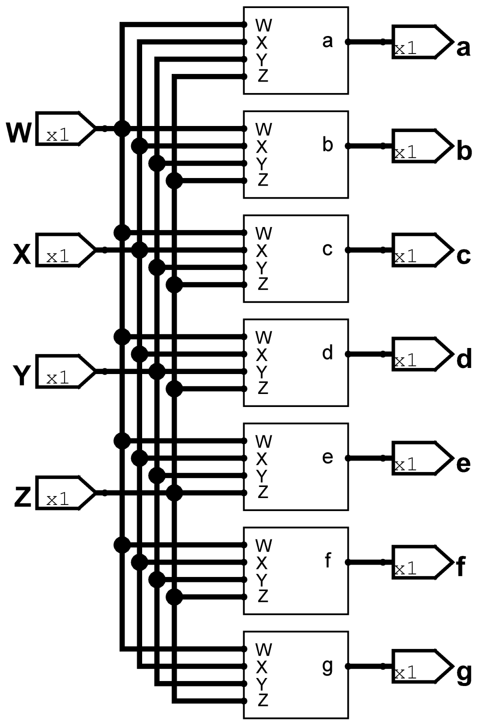
$$\begin{aligned}
b = {} & \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}\,\overline{Y}Z + \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y}\,\overline{Z} \\
& + W\overline{X}\,\overline{Y}Z + W\overline{X}Y\overline{Z} + W\overline{X}YZ + WX\overline{Y}\,\overline{Z} + WXYZ \\
& = \textcolor{red}{\overline{W}\,\overline{X}\,\overline{Y}} + \textcolor{red}{\overline{W}\,\overline{X}Y} + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + \textcolor{red}{W\overline{X}\,\overline{Y}} + \textcolor{red}{W\overline{X}Y} + WX\overline{Y}\,\overline{Z} + WXYZ \\
& \qquad (xy + x\overline{y} = x) \\
& = \textcolor{red}{\overline{W}\,\overline{X}} + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + \textcolor{red}{W\overline{X}} + WX\overline{Y}\,\overline{Z} + WXYZ \qquad (xy + x\overline{y} = x) \\
& = \textcolor{red}{\overline{X}} + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + WX\overline{Y}\,\overline{Z} + WXYZ \qquad (xy + x\overline{y} = x) \\
& = \overline{X} + \textcolor{red}{\overline{W}\,\overline{Y}\,\overline{Z}} + \textcolor{red}{\overline{W}YZ} + \textcolor{red}{W\overline{Y}\,\overline{Z}} + \textcolor{red}{WYZ} \qquad (x + \overline{x}y = x + y) \\
& = \overline{X} + \textcolor{red}{\overline{Y}\,\overline{Z}} + \textcolor{red}{YZ} \qquad (xy + x\overline{y} = x)
\end{aligned}$$

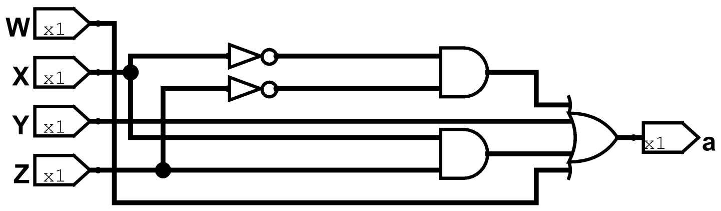
$$
\begin{aligned}
c &= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}\,\overline{Y}Z + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + \overline{W}XYZ \\
&\quad + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z + W\overline{X}YZ + WX\overline{Y}\,\overline{Z} + WX\overline{Y}Z + WXY\overline{Z} + WXYZ \\
&= \color{red}{\overline{W}\,\overline{X}\,\overline{Y}} + \overline{W}\,\overline{X}YZ + \color{red}{\overline{W}X\overline{Y}} + \color{red}{\overline{W}XY} + \color{red}{W\overline{X}\,\overline{Y}} + W\overline{X}YZ + \color{red}{WX\overline{Y}} + \color{red}{WXY} \\
&\hspace{20em} (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}YZ + \color{red}{\overline{W}X} + W\overline{X}\,\overline{Y} + W\overline{X}YZ + \color{red}{WX} \qquad (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}YZ + \color{red}{X} + W\overline{X}\,\overline{Y} + W\overline{X}YZ \qquad (xy + x\overline{y} = x) \\
&= \color{red}{\overline{W}\,\overline{Y}} + \color{red}{\overline{W}YZ} + X + \color{red}{W\overline{Y}} + \color{red}{WYZ} \qquad (x + \overline{x}y = x + y) \\
&= \color{red}{\overline{Y}} + \overline{W}YZ + X + WYZ \qquad (xy + x\overline{y} = x) \\
&= \overline{Y} + \color{red}{\overline{W}Z} + X + \color{red}{WZ} \qquad (x + \overline{x}y = x + y) \\
&= \overline{Y} + \color{red}{Z} + X \qquad (xy + x\overline{y} = x)
\end{aligned}
$$

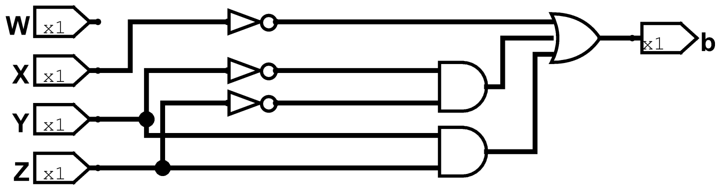
iii) We can observe that the corresponding logic circuit is of significantly lower complexity than the circuit obtained in the previous question. The reason this implementation is more efficient lies in a better choice for *don't care conditions* (the outputs corresponding to the inputs higher than digit 9). This example demonstrates how don't care conditions allow obtaining more efficient circuit implementations.

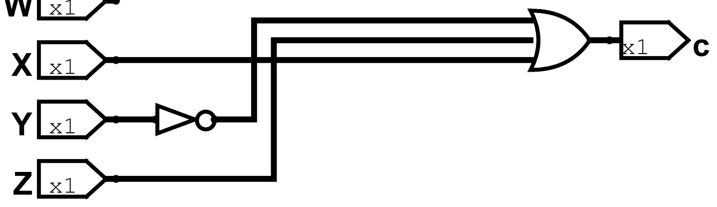
iv) The 7-segment display driver circuit implementation in Logisim-evolution is given in the figures below.

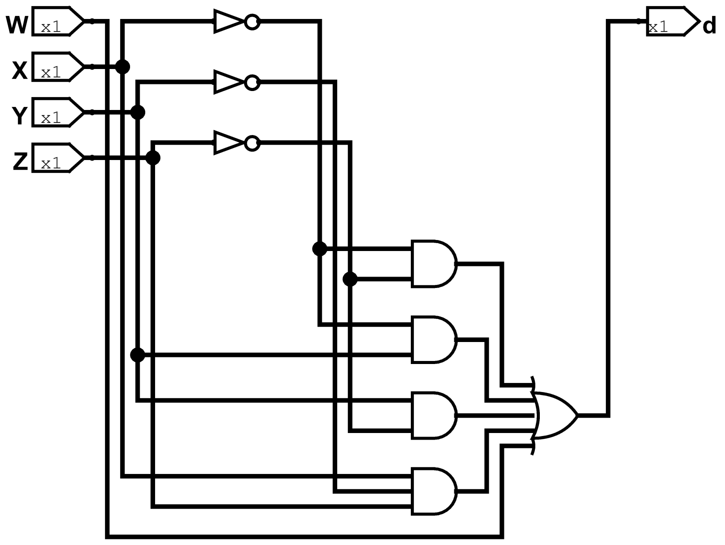


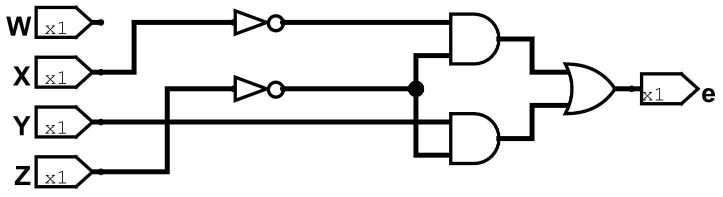

W
X
Y
Z
a
b
c
d
e
f
g
x1

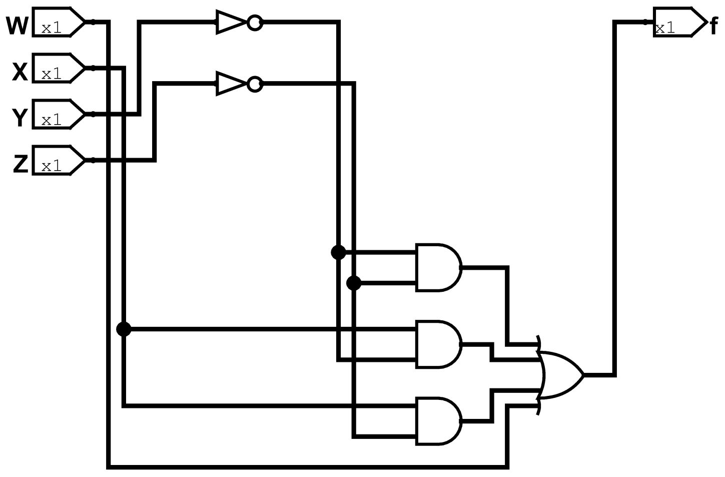

W
X
Y
Z
a
x1


W
X
Y
Z
b
x1


W
X
Y
Z
c
x1


W
X
Y
Z
x1
d


W
X
Y
Z
x1
e


W
X
Y
Z
x1
f

[Exercise 12] 7-Segment Display (Extended version)

a) You are required to design a circuit that drives a 7-segment display shown in Fig. 38. The design should satisfy the following criteria:

- The circuit has 4 inputs (W, X, Y, and Z) and 7 outputs (a, b, c, d, e, f, and g).
- Each of the outputs controls one segment of the display. The segment switches on (i.e., lights up) if the signal controlling it is high (i.e., logic 1).
- The display should show the decimal digit corresponding to the binary number formed by the inputs WXYZ. For example, if the inputs are WXYZ = 0110, the display should show the decimal digit 6. You can assume that the inputs will never take a combination higher than 1001 (i.e., decimal digit 9). If the inputs take a combination higher than 1001 (i.e., 9), the display should show nothing: all segments of the display should be switched off.

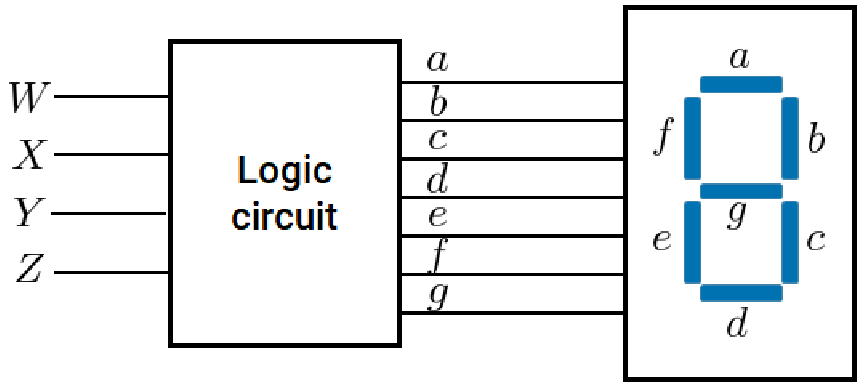


Figure 38: Seven-segment display.

i) Give the truth table for the digital logic circuit.

ii) Starting from the canonical SoP forms for all seven outputs, perform Boolean algebra transformations to reduce, as much as you can, the complexity of your circuit.

<u>Note:</u> You're encouraged to design and test your circuit in Logisim-evolution.

b) Repeat the same exercise, but this time use the outputs in Table 17 for input combinations higher than 1001 (i.e., 9). Compare the complexity of the digital logic circuit you implemented in this and the previous question. Discuss the differences.

Note: You're encouraged to design and test your circuit in Logisim-evolution.

Table 17: Alternative output signals for input combinations higher than 1001.

W	X	Y	Z	a	b	c	d	e	f	g
1	0	1	0	1	1	0	1	1	1	1
1	0	1	1	1	1	1	1	0	1	1
1	1	0	0	1	1	1	1	0	1	1
1	1	0	1	1	0	1	1	0	1	1
1	1	1	0	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1	0	1	1

[Solution 12] High Level Design

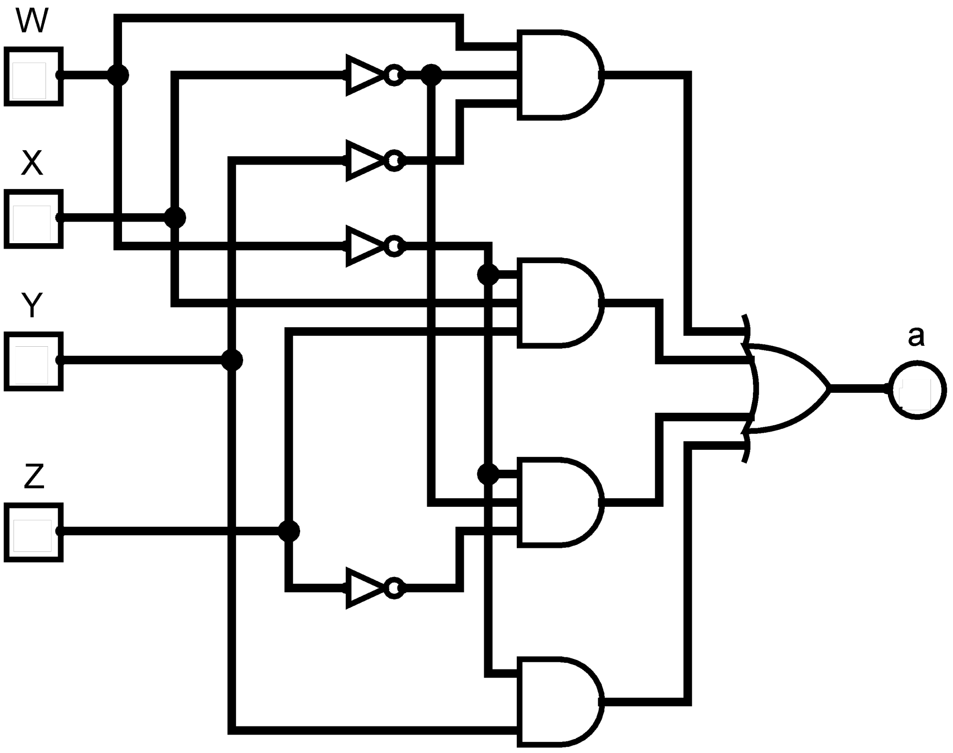
a) i)

W	X	Y	Z	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

ii) We will start from the canonical SoP forms and then apply Boolean algebra transformations to simplify the logic expressions.

$$
\begin{aligned}
a &= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y}Z + \overline{W}XY + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z \qquad (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y}Z + \overline{W}XY + W\overline{X}\,\overline{Y} \qquad (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y}Z + \overline{W}XYZ + \overline{W}XY + W\overline{X}\,\overline{Y} \qquad (x = x + xy) \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y + \overline{W}XZ + \overline{W}XY + W\overline{X}\,\overline{Y} \qquad (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}Y + \overline{W}XZ + W\overline{X}\,\overline{Y} \qquad (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}Y + \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}XZ + W\overline{X}\,\overline{Y} \qquad (x = x + xy) \\
&= \overline{W}\,\overline{X}\,\overline{Z} + \overline{W}Y + \overline{W}XZ + W\overline{X}\,\overline{Y} \qquad (xy + x\overline{y} = x)
\end{aligned}
$$

As an example, we show below the circuit diagram for output **a**.



$$
\begin{aligned}
b &= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}\,\overline{Y}Z + \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z \\
& && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y} && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X} + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y} && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X} + \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y} && (x = x + xy) \\
&= \overline{W}\,\overline{X} + \overline{W}\,\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y} && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X} + \overline{W}\,\overline{X}YZ + \overline{W}\,\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y} && (x = x + xy) \\
&= \overline{W}\,\overline{X} + \overline{W}\,\overline{Y}\,\overline{Z} + \overline{W}YZ + W\overline{X}\,\overline{Y} && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X} + \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{Y}\,\overline{Z} + \overline{W}YZ + W\overline{X}\,\overline{Y} && (x = x + xy) \\
&= \overline{W}\,\overline{X} + \overline{W}\,\overline{Y}\,\overline{Z} + \overline{W}YZ + \overline{X}\,\overline{Y} && (xy + x\overline{y} = x)
\end{aligned}
$$

$$\begin{aligned}
c &= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}\,\overline{Y}Z + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y}\,\overline{Z} \\
&\quad + W\overline{X}\,\overline{Y}Z \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z \\
&\qquad\qquad (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y} + \overline{W}XY\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z \\
&\qquad\qquad (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y} + \overline{W}XY + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y} + \overline{W}XY + W\overline{X}\,\overline{Y} && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}YZ + \overline{W}X + W\overline{X}\,\overline{Y} && (xy + x\overline{y} = x) \\
&= \overline{X}\,\overline{Y} + \overline{W}\,\overline{X}YZ + \overline{W}X && (xy + x\overline{y} = x) \\
&= \overline{X}\,\overline{Y} + \overline{W}\,\overline{X}\,\overline{Y}Z + \overline{W}\,\overline{X}YZ + \overline{W}X && (x = x + xy) \\
&= \overline{X}\,\overline{Y} + \overline{W}\,\overline{X}Z + \overline{W}X && (xy + x\overline{y} = x) \\
&= \overline{X}\,\overline{Y} + \overline{W}\,\overline{X}Z + \overline{W}X + \overline{W}XZ && (x = x + xy) \\
&= \overline{X}\,\overline{Y} + \overline{W}Z + \overline{W}X && (xy + x\overline{y} = x)
\end{aligned}$$

$$\begin{aligned}
d &= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y} && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y} && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y + \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y} && (x = x + xy) \\
&= \overline{W}\,\overline{X}\,\overline{Z} + \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y} && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Z} + \overline{W}\,\overline{X}Y + \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y} && (x = x + xy) \\
&= \overline{W}\,\overline{X}\,\overline{Z} + \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y}Z + \overline{W}Y\overline{Z} + W\overline{X}\,\overline{Y} && (xy + x\overline{y} = x)
\end{aligned}$$

$$\begin{aligned}
e &= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y}\,\overline{Z} \\
&= \overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}XY\overline{Z} && (xy + x\overline{y} = x) \\
&= \overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}Y\overline{Z} && (xy + x\overline{y} = x)
\end{aligned}$$

$$\begin{aligned}
f &= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y} && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}X\overline{Z} + \overline{W}X\overline{Y}Z + W\overline{X}\,\overline{Y} && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}X\overline{Z} + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y}Z + W\overline{X}\,\overline{Y} && (x = x + xy) \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}X\overline{Z} + \overline{W}X\overline{Y} + W\overline{X}\,\overline{Y} && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}X\overline{Z} + \overline{W}X\overline{Y} + \overline{W}X\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y} && (x = x + xy) \\
&= \overline{W}\,\overline{Y}\,\overline{Z} + \overline{W}X\overline{Z} + \overline{W}X\overline{Y} + W\overline{X}\,\overline{Y} && (xy + x\overline{y} = x)
\end{aligned}$$

$$\begin{aligned}
g &= \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z \\
&= \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y} + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y} + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y} && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y} + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y} && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y} + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y} && (x = x + xy) \\
&= \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y} + \overline{W}X\overline{Z} + W\overline{X}\,\overline{Y} && (xy + x\overline{y} = x)
\end{aligned}$$

The resulting simplified algebraic equations are summarized below.

$$\begin{aligned}
a &= W\overline{X}\,\overline{Y} + \overline{W}XZ + \overline{W}\,\overline{X}\,\overline{Z} + \overline{W}Y \\
b &= \overline{W}\,\overline{X} + \overline{W}YZ + \overline{W}\,\overline{Y}\,\overline{Z} + \overline{X}\,\overline{Y} \\
c &= \overline{W}X + \overline{W}Z + \overline{X}\,\overline{Y} \\
d &= W\overline{X}\,\overline{Y} + \overline{W}X\overline{Y}Z + \overline{W}\,\overline{X}Y + \overline{W}\,\overline{X}\,\overline{Z} + \overline{W}Y\overline{Z} \\
e &= \overline{W}Y\overline{Z} + \overline{X}\,\overline{Y}\,\overline{Z} \\
f &= W\overline{X}\,\overline{Y} + \overline{W}X\overline{Y} + \overline{W}X\overline{Z} + \overline{W}\,\overline{Y}\,\overline{Z} \\
g &= W\overline{X}\,\overline{Y} + \overline{W}X\overline{Y} + \overline{W}X\overline{Z} + \overline{W}\,\overline{X}Y
\end{aligned}$$

b) i)

W	X	Y	Z	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	1	1	0	1	1	1	1
1	0	1	1	1	1	1	1	0	1	1
1	1	0	0	1	1	1	1	0	1	1
1	1	0	1	1	0	1	1	0	1	1
1	1	1	0	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1	0	1	1

ii)

$$
\begin{aligned}
a = {} & \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y}\,\overline{Z} \\
& + W\overline{X}\,\overline{Y}Z + W\overline{X}Y\overline{Z} + W\overline{X}YZ + WX\overline{Y}\,\overline{Z} + WX\overline{Y}Z + WXY\overline{Z} + WXYZ \\
= {} & \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y}Z + \overline{W}XY + W\overline{X}\,\overline{Y} + W\overline{X}Y + WX\overline{Y} + WXY \\
& && (xy + x\overline{y} = x) \\
= {} & \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y}Z + \overline{W}XY + W\overline{X} + WX && (xy + x\overline{y} = x) \\
= {} & \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y}Z + \overline{W}XY + W && (xy + x\overline{y} = x) \\
= {} & \overline{X}\,\overline{Y}\,\overline{Z} + \overline{X}Y + X\overline{Y}Z + XY + W && (x + \overline{x}y = x + y) \\
= {} & \overline{X}\,\overline{Y}\,\overline{Z} + Y + X\overline{Y}Z + W && (xy + x\overline{y} = x) \\
= {} & \overline{X}\,\overline{Z} + Y + XZ + W && (x + \overline{x}y = x + y)
\end{aligned}
$$

$$\begin{aligned}
b &= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}\,\overline{Y}Z + \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y}\,\overline{Z} \\
&\quad + W\overline{X}\,\overline{Y}Z + W\overline{X}Y\overline{Z} + W\overline{X}YZ + WX\overline{Y}\,\overline{Z} + WXYZ \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X}\,\overline{Y} + W\overline{X}Y + WX\overline{Y}\,\overline{Z} + WXYZ \\
&& (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X} + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + W\overline{X} + WX\overline{Y}\,\overline{Z} + WXYZ && (xy + x\overline{y} = x) \\
&= \overline{X} + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}XYZ + WX\overline{Y}\,\overline{Z} + WXYZ && (xy + x\overline{y} = x) \\
&= \overline{X} + \overline{W}\,\overline{Y}\,\overline{Z} + \overline{W}YZ + W\overline{Y}\,\overline{Z} + WYZ && (x + \overline{x}y = x + y) \\
&= \overline{X} + \overline{Y}\,\overline{Z} + YZ && (xy + x\overline{y} = x)
\end{aligned}$$

$$\begin{aligned}
c &= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}\,\overline{Y}Z + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + \overline{W}XYZ \\
&\quad + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z + W\overline{X}YZ + WX\overline{Y}\,\overline{Z} + WX\overline{Y}Z + WXY\overline{Z} + WXYZ \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y} + \overline{W}XY + W\overline{X}\,\overline{Y} + W\overline{X}YZ + WX\overline{Y} + WXY \\
&& (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}YZ + \overline{W}X + W\overline{X}\,\overline{Y} + W\overline{X}YZ + WX && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y} + \overline{W}\,\overline{X}YZ + X + W\overline{X}\,\overline{Y} + W\overline{X}YZ && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{Y} + \overline{W}YZ + X + W\overline{Y} + WYZ && (x + \overline{x}y = x + y) \\
&= \overline{Y} + \overline{W}YZ + X + WYZ && (xy + x\overline{y} = x) \\
&= \overline{Y} + \overline{W}Z + X + WZ && (x + \overline{x}y = x + y) \\
&= \overline{Y} + Z + X && (xy + x\overline{y} = x)
\end{aligned}$$

$$
\begin{aligned}
d &= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z \\
&\quad + W\overline{X}Y\overline{Z} + W\overline{X}YZ + WX\overline{Y}\,\overline{Z} + WX\overline{Y}Z + WXY\overline{Z} + WXYZ \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y} + W\overline{X}Y + WX\overline{Y} + WXY \\
& && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + W\overline{X} + WX && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + W && (xy + x\overline{y} = x) \\
&= \overline{X}\,\overline{Y}\,\overline{Z} + \overline{X}Y + X\overline{Y}Z + XY\overline{Z} + W && (x + \overline{x}y = x + y) \\
&= \overline{X}(\overline{Y}\,\overline{Z} + Y) + X\overline{Y}Z + XY\overline{Z} + W && \text{(Distributive Property)} \\
&= \overline{X}(\overline{Z} + Y) + X\overline{Y}Z + XY\overline{Z} + W && (x + \overline{x}y = x + y) \\
&= \overline{X}\,\overline{Z} + \overline{X}Y + X\overline{Y}Z + XY\overline{Z} + W && \text{(Distributive Property)} \\
&= \overline{X}\,\overline{Z} + Y(\overline{X} + X\overline{Z}) + X\overline{Y}Z + W && \text{(Distributive Property)} \\
&= \overline{X}\,\overline{Z} + Y(\overline{X} + \overline{Z}) + X\overline{Y}Z + W && (x + \overline{x}y = x + y) \\
&= \overline{X}\,\overline{Z} + \overline{X}Y + Y\overline{Z} + X\overline{Y}Z + W && \text{(Distributive Property)}
\end{aligned}
$$

$$
\begin{aligned}
e &= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}Y\overline{Z} + WXY\overline{Z} \\
&= \overline{X}\,\overline{Y}\,\overline{Z} + \overline{X}Y\overline{Z} + XY\overline{Z} && (xy + x\overline{y} = x) \\
&= \overline{X}\,\overline{Z} + XY\overline{Z} && (xy + x\overline{y} = x) \\
&= \overline{Z}(\overline{X} + XY) && \text{(Distributive Property)} \\
&= \overline{Z}(\overline{X} + Y) && (x + \overline{x}y = x + y) \\
&= \overline{X}\,\overline{Z} + Y\overline{Z} && \text{(Distributive Property)}
\end{aligned}
$$

$$
\begin{aligned}
f &= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}\,\overline{Y}Z \\
&\quad + W\overline{X}Y\overline{Z} + W\overline{X}YZ + WX\overline{Y}\,\overline{Z} + WX\overline{Y}Z + WXY\overline{Z} + WXYZ \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y} + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y} + W\overline{X}Y + WX\overline{Y} + WXY && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y} + \overline{W}XY\overline{Z} + W\overline{X} + WX && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y} + \overline{W}XY\overline{Z} + W && (xy + x\overline{y} = x) \\
&= \overline{X}\,\overline{Y}\,\overline{Z} + X\overline{Y} + XY\overline{Z} + W && (x + \overline{x}y = x + y) \\
&= \overline{X}\,\overline{Y}\,\overline{Z} + X(\overline{Y} + Y\overline{Z}) + W && \text{(Distributive Property)} \\
&= \overline{X}\,\overline{Y}\,\overline{Z} + X(\overline{Y} + \overline{Z}) + W && (x + \overline{x}y = x + y) \\
&= \overline{X}\,\overline{Y}\,\overline{Z} + X\overline{Y} + X\overline{Z} + W && \text{(Distributive Property)} \\
&= \overline{Y}(\overline{X}\,\overline{Z} + X) + X\overline{Z} + W && \text{(Distributive Property)} \\
&= \overline{Y}(\overline{Z} + X) + X\overline{Z} + W && (x + \overline{x}y = x + y) \\
&= \overline{Y}\,\overline{Z} + X\overline{Y} + X\overline{Z} + W && \text{(Distributive Property)}
\end{aligned}
$$

$$
\begin{aligned}
g &= \overline{W}\,\overline{X}Y\overline{Z} + \overline{W}\,\overline{X}YZ + \overline{W}X\overline{Y}\,\overline{Z} + \overline{W}X\overline{Y}Z + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y}\,\overline{Z} \\
&\quad + W\overline{X}\,\overline{Y}Z + W\overline{X}Y\overline{Z} + W\overline{X}YZ + WX\overline{Y}\,\overline{Z} + WX\overline{Y}Z + WXY\overline{Z} + WXYZ \\
&= \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y} + \overline{W}XY\overline{Z} + W\overline{X}\,\overline{Y} + W\overline{X}Y + WX\overline{Y} + WXY && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y} + \overline{W}XY\overline{Z} + W\overline{X} + WX && (xy + x\overline{y} = x) \\
&= \overline{W}\,\overline{X}Y + \overline{W}X\overline{Y} + \overline{W}XY\overline{Z} + W && (xy + x\overline{y} = x) \\
&= \overline{X}Y + X\overline{Y} + XY\overline{Z} + W && (x + \overline{x}y = x + y) \\
&= \overline{X}Y + X(\overline{Y} + Y\overline{Z}) + W && \text{(Distributive Property)} \\
&= \overline{X}Y + X(\overline{Y} + \overline{Z}) + W && (x + \overline{x}y = x + y) \\
&= \overline{X}Y + X\overline{Y} + X\overline{Z} + W && \text{(Distributive Property)}
\end{aligned}
$$

The resultant simplified algebraic equations are summarized below. We can observe that the corresponding logic circuit is of significantly lower complexity than the circuit obtained in the previous question. The reason this implementation is more efficient lies in a better choice for *don't care conditions* (the outputs corresponding to the inputs higher than digit 9). This example demonstrates how don't care conditions allow obtaining more efficient circuit implementations.

$$\begin{aligned}
a &= \overline{X}\,\overline{Z} + Y + XZ + W \\
b &= \overline{X} + \overline{Y}\,\overline{Z} + YZ \\
c &= \overline{Y} + Z + X \\
d &= \overline{X}\,\overline{Z} + \overline{X}Y + Y\overline{Z} + X\overline{Y}Z + W \\
e &= \overline{X}\,\overline{Z} + Y\overline{Z} \\
f &= \overline{Y}\,\overline{Z} + X\overline{Y} + X\overline{Z} + W \\
g &= \overline{X}Y + X\overline{Y} + X\overline{Z} + W
\end{aligned}$$

[Exercise 13] Timing Hazards

a) Consider the circuit in Figure 39.

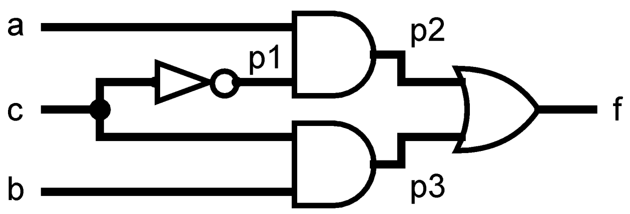


Figure 39: A digital circuit.

i) Derive the Boolean expression for the output f in terms of the inputs a, b, and c.

ii) Assume an ideal scenario where gates have no delay. Fill the timing diagram in Figure 40 for the circuit in Figure 39.

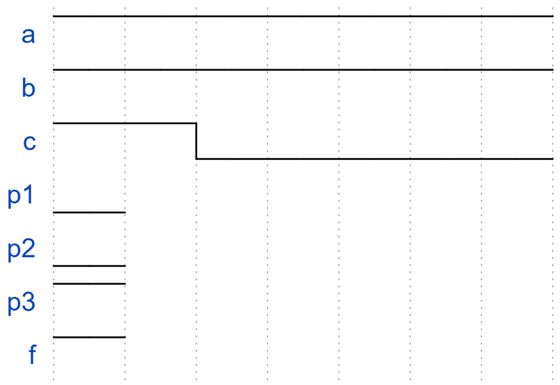


Figure 40: The timing diagram.

iii) Now consider the same circuit and input signals, but each gate has a delay of 1 time unit. Each dotted vertical line shows one (single) time unit. Fill the timing diagram in Figure 40 for the circuit in Figure 39. Is the output correct at all times? If not, identify the reason for the incorrect output.

b) Now consider the circuit in Figure 41.

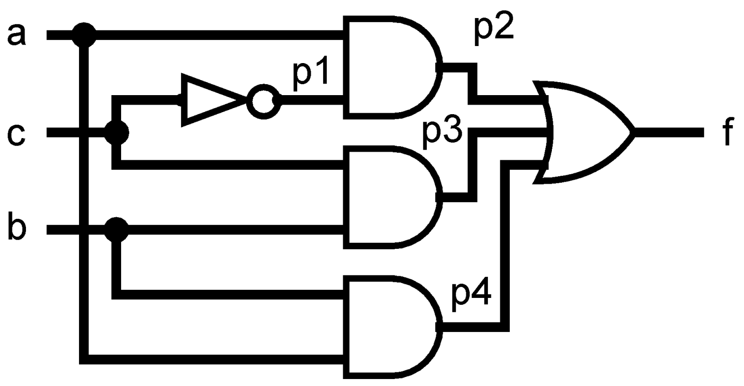


Figure 41: A digital circuit.

i) Derive the Boolean expression for the output f in terms of the inputs a, b, and c. Is this expression functionally equivalent to the expression you obtained in question **a**?

ii) Assume an ideal scenario where gates have no delay. Fill the timing diagram in Figure 42 for the circuit in Figure 41.

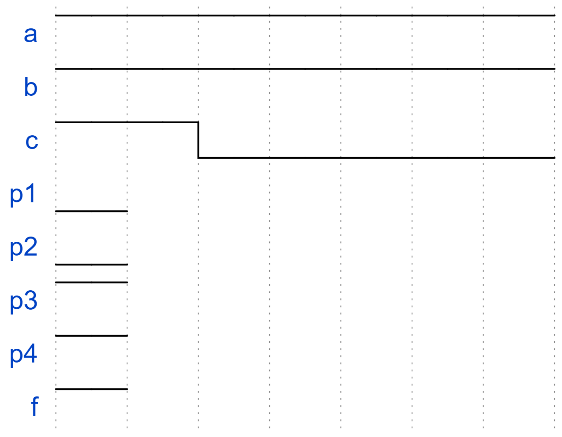


Figure 42: The timing diagram.

iii) Now consider the same circuit and input signals, but each gate has a delay of 1 time unit. Fill the timing diagram in Figure 42 for the circuit in Figure 41. Is the output correct at all times? If not, identify the reason for the incorrect output.

[Solution 13] Timing Hazards

a)

i) The Boolean expression for the output f in terms of the inputs a, b, and c is:

$$\begin{aligned} p1 &= \overline{c} \\ p2 &= a \cdot p1 \\ p3 &= b \cdot c \\ f &= p2 + p3 \\ &= a \cdot \overline{c} + b \cdot c \end{aligned}$$

ii) The timing diagram with zero delay gates for the circuit in Figure 39 is shown in Figure 43.

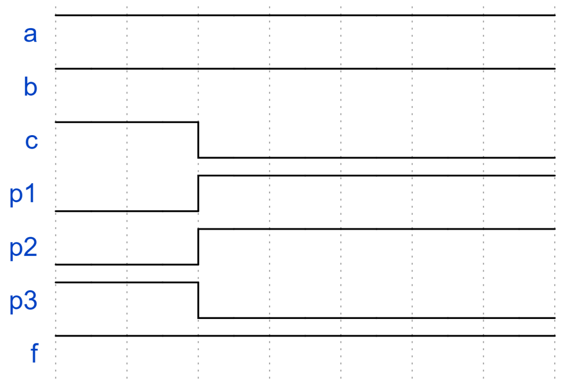


Figure 43: The timing diagram without any delay.

iii) The timing diagram with one time-unit delay for each gate is shown in Figure 44.

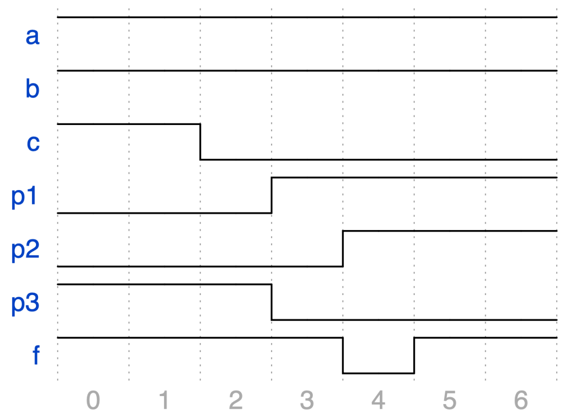


Figure 44: The timing diagram with delays.

At time t = 4, the output has a glitch because the output changes from 1 to 0 and then back to 1. There are two parallel data paths from input c to the output and the delays of these paths are different. Glitches like these (also called timing hazards) typically occur when the inputs change from one product term to another and there is a difference in delay these product terms take to generate the output.

b)

i) The Boolean expression for the output f in terms of the inputs a, b, and c is:

$$\begin{aligned}
p1 &= \overline{c} \\
p2 &= a \cdot p1 \\
p3 &= b \cdot c \\
p4 &= a \cdot b \\
f &= p2 + p3 + p4 \\
&= a \cdot \overline{c} + b \cdot c + a \cdot b \\
&= a \cdot \overline{c} + b \cdot c + a \cdot b \cdot (c + \overline{c}) && (x \cdot 1 = x) \\
&= a \cdot \overline{c} + b \cdot c + a \cdot b \cdot c + a \cdot b \cdot \overline{c} && \text{(Distributive Property)} \\
&= a \cdot \overline{c} \cdot (1 + b) + b \cdot c \cdot (1 + a) && \text{(Distributive Property)} \\
&= a \cdot \overline{c} + b \cdot c && (1 + x = 1)
\end{aligned}$$

The expression is functionally equivalent to the one obtained in question (a).

ii) The timing diagram with zero delay gates for the circuit in Figure 41 is shown in Figure 45.

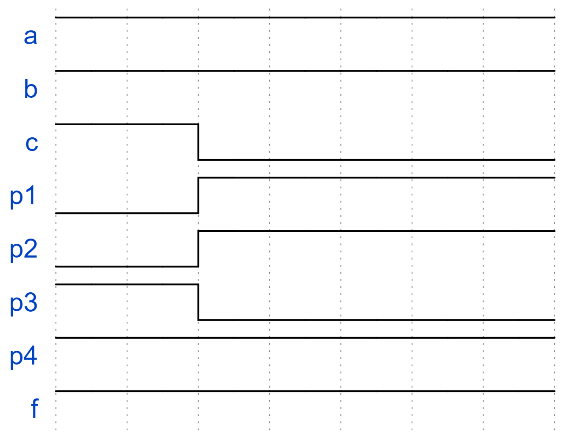


Figure 45: The timing diagram without delays.

iii) The timing diagram with one time-unit delay for each gate is shown in Figure 46.

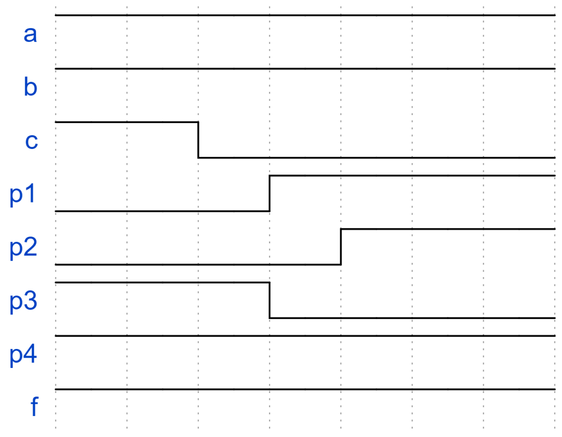


Figure 46: The timing diagram with delays.

There are no glitches in the output. The output is correct at all times. The redundant term $a \cdot b$ in the Boolean expression helps to eliminate the glitches in the output while c transitions: the redundant term generates a signal independent of c, that keeps the output unaffected by the glitch created by c transitions.

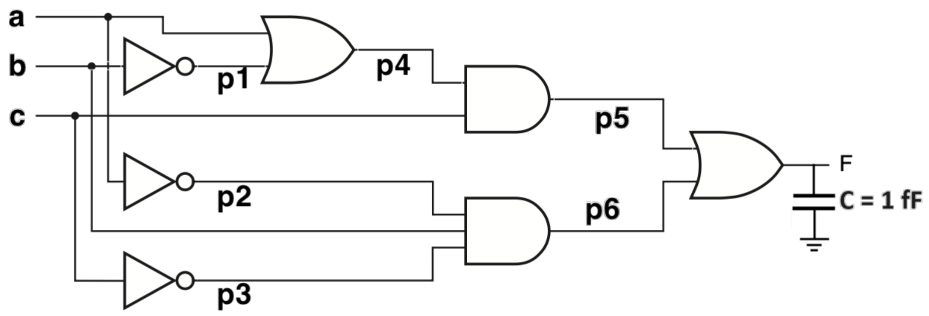
Note: In general, avoiding hazards is not a trivial task. It requires careful design and analysis of the circuit. However, a well-designed, *synchronous* digital system is structured so that hazard analysis is not needed for most of its elements. In a synchronous system, all inputs to a *combinational* circuit, such as the one we analyzed earlier, are changed at a particular time, and the outputs are not *looked at* until they have had time to settle to a steady-state (i.e., glitch-free) value.

[Exercise 14] Logic Circuit Propagation Delay

Consider the following circuit. The inputs become available at $t = 0$. If any logic gate drives a load capacitance of $C_{\text{load}} = 1$ fF, its propagation delay is 10 ns. The capacitance of an input of a logic gate depends on the total number of inputs of that gate (i.e., of its fan-in). Assume that the following holds:

- the capacitance of an input of the NOT gate is 1 fF,
- the capacitance of an input of a two-input gate (AND or OR) is 2 fF, and
- the capacitance of an input of a three-input gate (AND or OR) is 3 fF.

Finally, the load capacitance of the final OR gate is 1 fF.



a) Compute the delay of every gate. Find the worst-case delay (the critical path delay) of the circuit, in nanoseconds.

b) Which path is the **critical path**? Highlight in red the critical path in the circuit. If there are multiple paths with the same delay, color them all.

[Solution 14] Logic Circuit Propagation Delay

a)

Note: Recall that, when a logic gate drives a load capacitance of $C_{\text{load}} = 1$ fF, its propagation delay is 10 ns.

The inverter is driving an input of a two-input OR gate (wire $p1$), which has the capacitance of 2 fF. Therefore, the inverter delay is $2fF \times \frac{10ns}{1fF} = 2 \times 10ns = 20ns$.

The inverter is driving an input of a three-input AND gate (wire $p2$), which has the capacitance of 3 fF. Therefore, the inverter delay is $3 \times 10ns = 30ns$.

The inverter is driving an input of a three-input AND gate (wire $p3$), which has the capacitance of 3 fF. Therefore, the inverter delay is $3 \times 10ns = 30ns$.

The OR gate is driving an input of a two-input AND gate (wire $p4$), which has the capacitance of 2 fF. Therefore, the OR gate delay is $d_{OR} = 2 \times 10ns = 20ns$. The OR gate receives inputs from a and $p1$. The input from a is available at 0 ns, whereas the input from $p1$ becomes available at 20 ns. Therefore, the output of the OR gate (wire $p4$) will be ready at time $t = d_{OR} + \max(0ns, 20ns) = 20ns + 20ns = 40ns$.

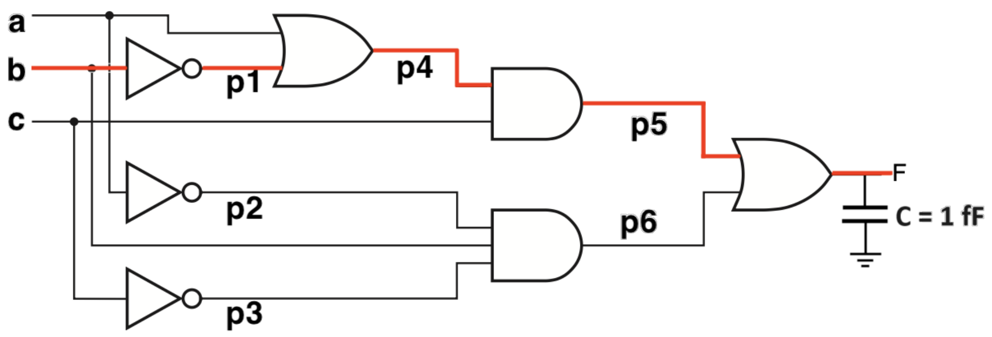
The top AND gate is driving an input of a two-input OR gate (wire $p5$), which has the capacitance of 2 fF. Therefore, the AND gate delay is $2 \times 10ns = 20ns$. The AND gate receives inputs from $p4$ and c. The signal on $p4$ becomes ready at 40 ns and c is available at 0 ns. Therefore, the output of the AND gate (wire $p5$) will be ready at time $t = 20 + \max(40, 0) = 20 + 40 = 60ns$.

The bottom AND gate is driving an input of a two-input OR gate (wire $p6$), which has the capacitance of 2 fF. Therefore, the AND gate delay is $2 \times 10ns = 20ns$. The AND gate receives inputs from $p2$, b, and $p3$. The signal on $p2$ becomes ready at 30 ns, b is available at 0 ns, and the signal on $p3$ becomes ready at 30 ns. Therefore, the output of the AND gate (wire $p6$) will be ready at time $t = 20 + \max(30, 0, 30) = 20 + 30 = 50ns$.

Finally, the last OR gate drives an output (load) capacitance of 1 fF. Therefore, the OR gate delay is $1 \times 10ns = 10ns$. The OR gate receives inputs from $p5$ and $p6$. The signal on $p5$ becomes ready at 60 ns. The signal on $p6$ becomes ready at 50 ns. Therefore, the output F will be ready at time $t = 10 + \max(60, 50) = 10 + 60 = 70ns$.

b)

The circuit has one critical path, highlighted in the figure below.



[Exercise 15] Multiplexer Design

a) Design an 8-to-1 multiplexer using only 2-to-1 multiplexers.

b) Design an 8-to-1 multiplexer using any combination of 2-to-1 and 4-to-1 multiplexers.

[Solution 15] Multiplexer Design

a) The inputs of the multiplexer are a, b, c, d, e, f, g, and h. The select lines are x, y and z. The output is m. The circuit using only 2-to-1 multiplexers is shown in Figure 47.

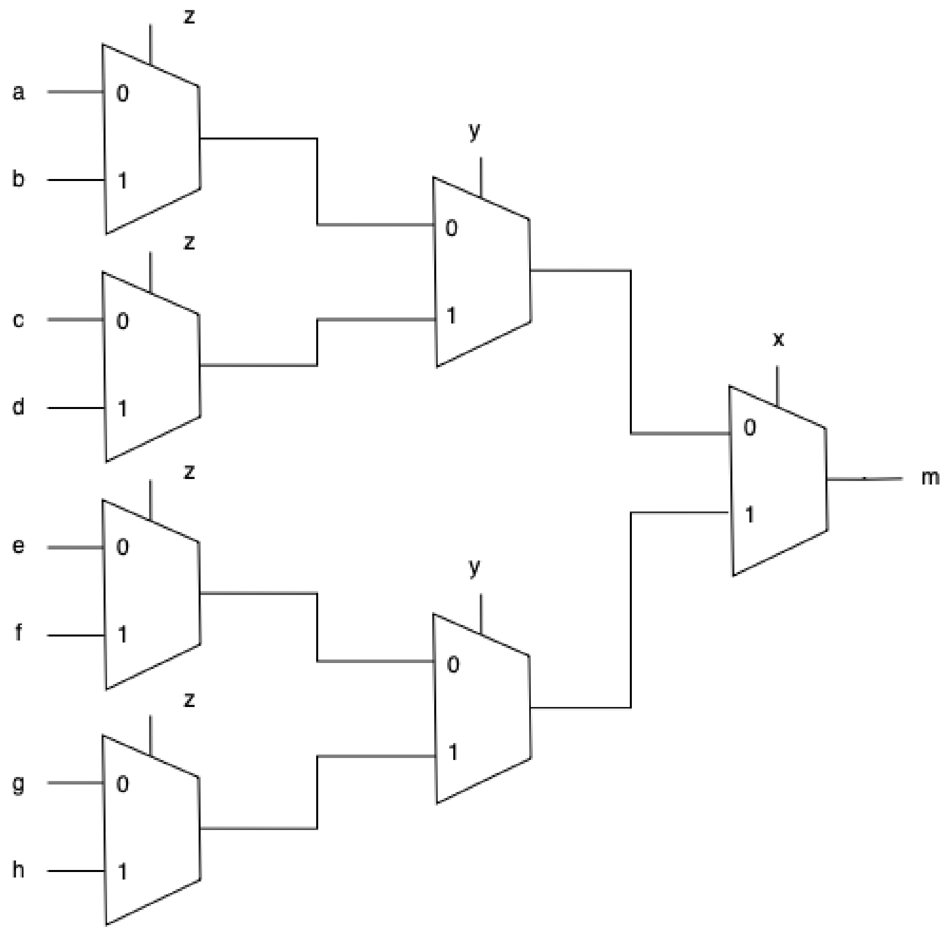


Figure 47: 8-to-1 multiplexer circuit using only 2-to-1 multiplexers.

b) Combining the last 2 stages of the circuit in Figure 47 using a 4-to-1 multiplexer, we get the circuit shown in Figure 48.

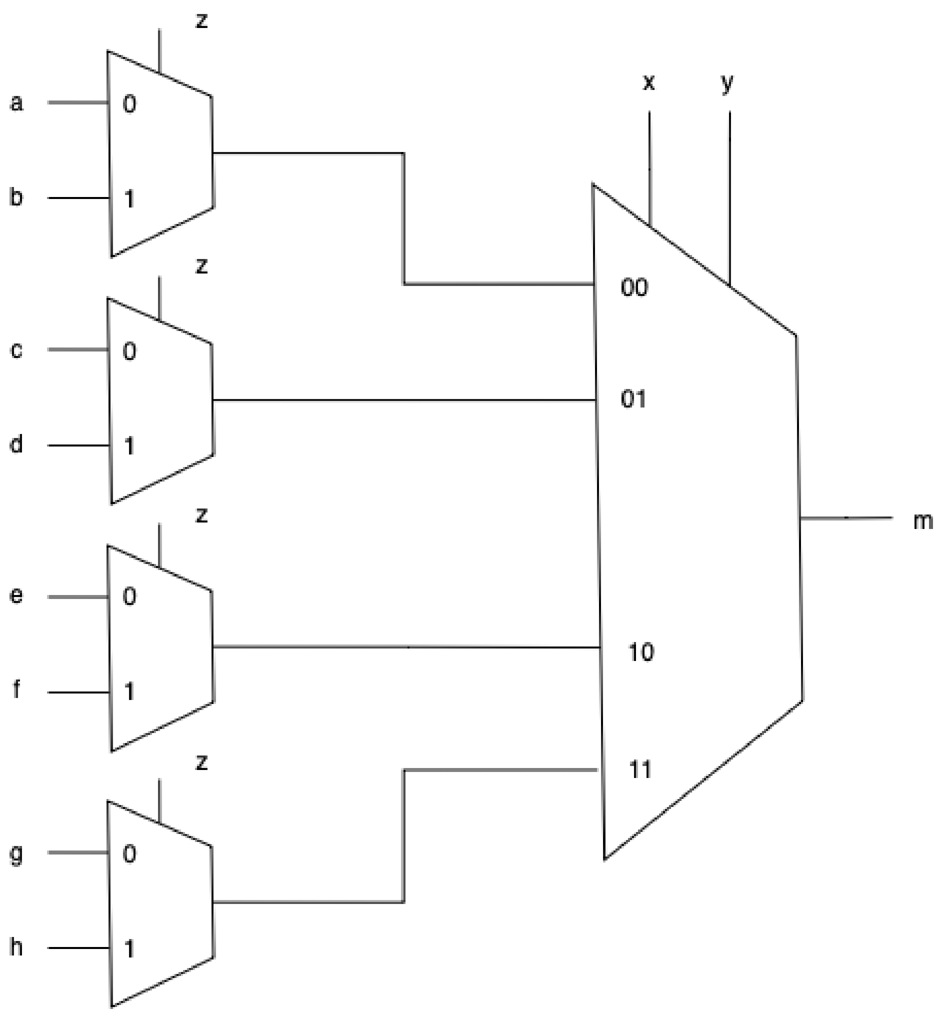


Figure 48: 8-to-1 multiplexer circuit using 2-to-1 and 4-to-1 multiplexers.

[Exercise 16] Dynamic Power Dissipation

a) Find the dynamic power dissipation P_1 of a CMOS inverter operated from a 1.05 V supply and having a load capacitance of 100 fF. Let the inverter be switched at 320 MHz.

b) Given the same inverter as in question **a**, find the dynamic power P_2 assuming the inverter has a load capacitance of 300 fF. What can you tell about the impact of load capacitance on dynamic power dissipation?

c) Find the dynamic power dissipation P_3 of a CMOS inverter operated from a 1.05 V supply and having a load capacitance of 100 fF. Let the inverter be switched at 1 GHz. What can you tell about the impact of switching frequency on dynamic power dissipation?

d) Find the dynamic power dissipation P_4 of a CMOS inverter having a load capacitance of 100 fF and switching at 320 MHz. Let the inverter be powered from 0.9 V supply. What can you tell about the impact of power supply on dynamic power dissipation?

[Solution 16] Dynamic Power Dissipation

a) The dynamic power dissipation is given by the equation: $P_1 = C_{out}V_{dd}^2 f$. Plugging in the given values, $C_{out} = 100fF = 10^{-13}F$, $V_{dd} = 1.05V$, and $f = 320MHz = 3.2 \times 10^8 Hz$, we get $P_1 = 10^{-13}F \times (1.05V)^2 \times 3.2 \times 10^8 Hz = 3.528 \times 10^{-5}W = 35.28\mu W$.

b) Using the same formula with the new capacitance value $C_{out} = 300fF = 3 \times 10^{-13}F$, we get $P_2 = 3 \times 10^{-13}F \times (1.05V)^2 \times 3.2 \times 10^8 Hz = 10.584 \times 10^{-5}W = 105.84\mu W$. The dynamic power P_2 tripled with tripled capacitance, compared to P_1, because the dynamic power is directly proportional to the capacitance.

c) Using the same formula with the new frequency value $f = 1GHz = 10^9 Hz$, we get $P_3 = 10^{-13}F \times (1.05V)^2 \times 10^9 Hz = 1.1025 \times 10^{-4}W = 110.25\mu W$. The dynamic power P_3 increased by a factor of 3.125 (i.e., the same factor of increase in frequency) compared to P_1, because it is directly proportional to the frequency.

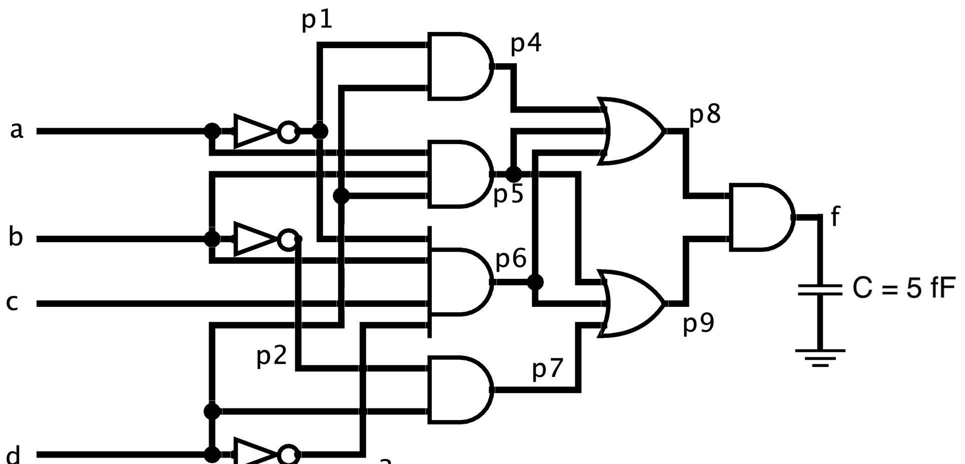
d) Using the same formula with the new voltage value $V_{dd} = 0.9V$, we get $P_4 = 10^{-13}F \times (0.9V)^2 \times 3.2 \times 10^8 Hz = 2.592 \times 10^{-5}W = 25.92\mu W$. The dynamic power P_4 decreased by a factor of 1.36 compared to P_1; the dynamic power is directly proportional to the **square** of the power supply voltage.

[Exercise 17] Fanin & Fanout and Propagation Delay

Consider the following circuit. The inputs become available at $t = 0$. If any logic gate drives a load capacitance of $C_{\text{load}} = 1$ fF, its propagation delay is 10 ns. The load capacitance and therefore the delay of a gate depends on its fanout. If a gate drives multiple gates, the equivalent load capacitance is the sum of the input capacitance of each fanout. The capacitance of an input of a logic gate depends on the total number of inputs of that gate (i.e., of its fan-in). Assume that the following holds:

- the capacitance of an input of the NOT gate is 1 fF,
- the capacitance of an input of a two-input gate (AND or OR) is 2 fF, and
- the capacitance of an input of a three-input gate (AND or OR) is 3 fF.
- the capacitance of an input of a four-input gate (AND or OR) is 4 fF.

For example, if a gate drives one two-input gate and one three-input gate, then the equivalent load capacitance is $C_{\text{load}} = 2 \text{ fF} + 3 \text{ fF} = 5$ fF and, consequently, the delay of that gate increases five times: $10 \times 5 = 50$ ns. The load capacitance of the final AND gate equals 5 fF.



a) Compute the delay of each gate. What is the worst-case delay (the critical path delay) of the circuit, in nanoseconds?

b) Which path is the **critical path**? Highlight in red the critical path in the circuit. If there are multiple paths with the same delay, color them all.

[Solution 17] Fanin & Fanout and Propagation Delay

a)

The inverter driving wire p1 has a fanout of 2. The first gate is a two input gate while the second gate is a four input gate. So, the total load capacitance is 2 fF $+$ 4 fF $=$ 6 fF. Therefore, its propagation delay is $\frac{10ns}{1fF} \times 6fF = 10ns \times 6 = 60ns$.

The inverter driving wire p2 has a fanout of 1. The gate it drives has 2 inputs, so the total load capacitance is 2 fF. Therefore, its propagation delay is $10ns \times 2 = 20ns$.

The inverter driving wire p3 has a fanout of 1. The gate it drives has 4 inputs, so the total load capacitance is 4 fF. Therefore, its propagation delay is $10ns \times 4 = 40ns$.

The AND gate driving wire p4 has a fanout of 1. The gate it drives has 3 inputs, so the total load capacitance is 3 fF. Therefore, its propagation delay is $10ns \times 3 = 30ns$. Moreover, it gets its inputs from wires p1 and d. p1 becomes available at 60ns and d becomes available at 0ns. Therefore, p4 becomes available at time $t = max(60ns, 0ns) + 30ns = 90ns$.

The AND gate driving wire p5 has a fanout of 2. The first gate it drives has 3 inputs and the second gate has 3 inputs, so the total load capacitance is 6 fF. Therefore, its propagation delay is $10ns \times 6 = 60ns$. Moreover, it gets its inputs from wires a, b, and d. They all become available at time $t = 0ns$. Therefore, p5 becomes available at time $t = 60ns$.

The AND gate driving wire p6 has a fanout of 2. The first gate it drives has 3 inputs and the second gate has 3 inputs, so the total load capacitance is 6 fF. Therefore, its propagation delay is $10ns \times 6 = 60ns$. Moreover, it gets its inputs from wires p1, b, c, and p3. p1 becomes available at 60ns, b and c become available at 0ns, and p3 becomes available at 40ns. Therefore, p6 becomes available at time $t = max(60, 0, 0, 40) + 60 = 120ns$.

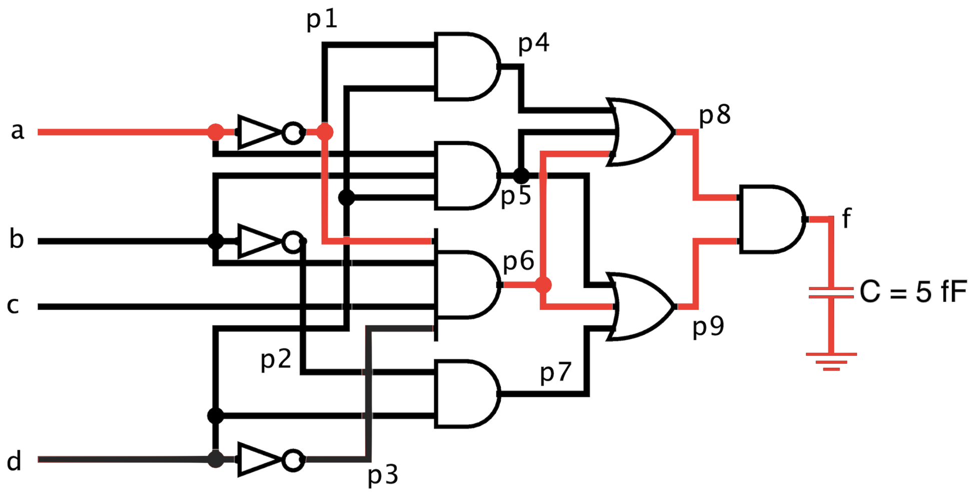
The AND gate driving wire p7 has a fanout of 1. The gate it drives has 3 inputs, so the total load capacitance is 3 fF. Therefore, its propagation delay is $10ns \times 3 = 30ns$. Moreover, it gets its inputs from wires p2, and d. p2 becomes available at 20ns and d becomes available at 0ns. Therefore, p7 becomes available at time $t = max(20, 0) + 30 = 50ns$.

The OR gate driving wire p8 has a fanout of 1. The gate it drives has 2 inputs, so the total load capacitance is 2 fF. Therefore, its propagation delay is $10ns \times 2 = 20ns$. Moreover, it gets its inputs from wires p4, p5, and p6. p4 becomes available at 90ns, p5 becomes available at 60ns, and p6 becomes available at 120ns. Therefore, p8 becomes available at time $t = max(90, 60, 120) + 20 = 140ns$.

The OR gate driving wire p9 has a fanout of 1. The gate it drives has 2 inputs, so the total load capacitance is 2 fF. Therefore, its propagation delay is $10ns \times 2 = 20ns$. Moreover, it gets its inputs from wires p5, p6, and p7. p5 becomes available at 60ns, p6 becomes available at 120ns, and p7 becomes available at 50ns. Therefore, p9 becomes available at time $t = max(60, 120, 50) + 20 = 140ns$.

The final AND gate driving a load capacitance of 5 fF. Therefore, its propagation delay is $10ns \times 5 = 50ns$. Moreover, it gets its inputs from wires p8 and p9. Both p8 and p9 become available at 140ns. Therefore, the output becomes available at time $t = max(140, 140) + 50 = 190ns$.

b)



[Exercise 18] Logic Circuits and Verilog: Gate-Level Modeling

Consider the logic circuit shown in Figure 49. The circuit has three inputs a, b, c, one output f, and one intermediate wire p (shown here with a Logisim probe).

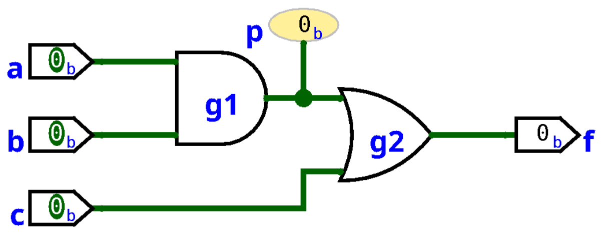


Figure 49: A simple combinational circuit with three inputs and one output.

Listing 2.1 shows one way of modeling the same circuit in Verilog structurally, at the gate level. Listing 2.2 shows an example Verilog testbench for simulating the operation of the circuit.

Listing 2.1: A structural Verilog description of the circuit in Figure 49.

```
module structural_example (
  // Three input signals.
  input  a,
  input  b,
  input  c,
  // One output signal.
  output f
);

  // One intermediate wire.
  wire p;

  // Two gates specifying name, output, and inputs.
  and g1 (p, a, b);
  or  g2 (f, p, c);

endmodule
```

Listing 2.2: Verilog testbench for the circuit in Figure 49.

```
module test_structural_example;

  // Define three inputs that can procedurally
  // be assigned values.
  reg  a, b, c;
  // And one output that responds to them.
  wire f;

  // Connect them to an instance of the module being tested.
  structural_example ex (.a(a), .b(b), .c(c), .f(f));

  // Loop variable.
  integer i;

  initial begin
    // Write this test's data to a .vcd file
    // that GTKWave can read.
    $dumpfile ("structural_example.vcd");
    $dumpvars;

    // Print values whenever they change.
    $monitor ("Time %2t, a=%b, b=%b, c=%b, f=%b",
              $time, a, b, c, f);

    // Exhaust all input combinations,
    // each time followed by a delay of 1 time unit.
    for (i = 0; i < 8; i += 1) begin
      // Each variable gets one bit from i.
      // First 0, 0, 0
      // then  0, 0, 1, etc.
      {a, b, c} = i;
      #1;
    end

    // Done.
    $finish;
  end

endmodule
```

a) Start by creating two files:

- Verilog source file, called `structural_example.v`, containing the gate-level model given in Listing 2.1; and
- Verilog testbench file, called `structural_example_tb.v`, containing the code for generating various input combinations (see Listing 2.2), so that you can test the Verilog model of your circuit.

Note: The two files above are also available for download from Moodle.

Run circuit simulation and generate the waveforms. To do so, feel free to follow the sequence of commands given in Listing 2.3 below. Inspect the timing waveforms to verify that the Verilog description matches the expected behavior of the logic circuit in Figure 49.

Listing 2.3: Inspecting the waveform simulation in GTKWave.

```
$ iverilog -o example structural_example.v \
                      structural_example_tb.v
$ ./example
$ gtkwave structural_example.vcd
```

b) Consider a slightly different logic circuit shown in Figure 50, which has three inputs a, b, c, and one output f. Write a structural (i.e., gate-level) Verilog description of the circuit. Use only Verilog gate primitives (e.g., `not`, `and`, `nor`).

Simulate your circuit (i.e., generate waveforms to verify its correct operation).
Hint: You can use the same testbench as in the previous question.

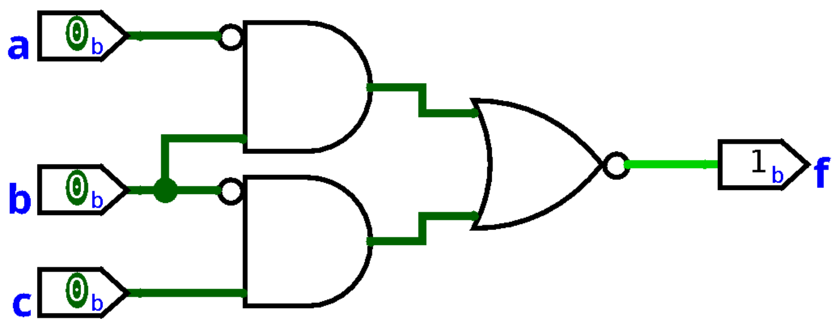


Figure 50: A more involved combinational circuit with three inputs and one output.

c) Recall that a Full-Adder is a one-bit adder that can be realized with two Half-Adders (see Figure 51). A Full-Adder has three binary inputs and two binary outputs:

- one-bit inputs x, y;
- one-bit input carry-in c_{in};
- one-bit output s, which is the one-bit binary sum of $x + y$; and
- one-bit output carry-out c_{out}.

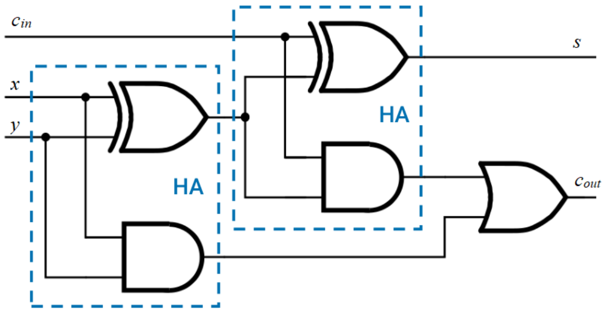


Figure 51: A logic circuit performing the function of a Full-Adder.

Write a structural (gate-level) Verilog description of a Full-Adder.

i) Variant 1: For a moment, ignore the fact that some gates can be grouped into a Half-Adder. In other words, make your Full-Adder be entirely described with basic Verilog gates. Name your `module full_adder1` and give it three inputs `x`, `y`, `cin`, and two outputs `s`, `cout`. As before, check that your Verilog model is correct by inspecting the waveforms in GTKWave. This time, use the testbench file `full_adder1_tb.v` (available for download from Moodle).

ii) Variant 2: Create a Verilog module `half_adder` that describes the functionality of a Half-Adder using the gate-level modeling approach. Then, use `half_adder` as a submodule to simplify the description of your Full-Adder. Name your Full-Adder `module full_adder2` and give it three inputs `x`, `y`, `cin`, and two outputs `s`, `cout`. You can define both modules `half_adder` and `full_adder2` in the same file. Otherwise, if `half_adder` is in its own file, remember to include that file name as an argument to `iverilog` when compiling. As before, check that your Verilog model is correct by inspecting the waveforms in GTKWave. This time, use the testbench file `full_adder2_tb.v` (available for download from Moodle).

[Solution 18] Logic Circuits and Verilog: Gate-Level Modeling

a) Comparing the circuit in Figure 49 with its structural Verilog model in Listing 2.1 should help to highlight the straightforward means by which one can arrive at one from the other: first, identify the inputs and outputs; then, the logic gates; and finally the intermediate wires (other than the input/output ports) that are needed to connect the gates.

After saving Listing 2.1 as the file `structural_example.v` and Listing 2.2 as `structural_example_tb.v`, and running the commands in Listing 2.3, you should see the GTKWave output shown in Figure 52.

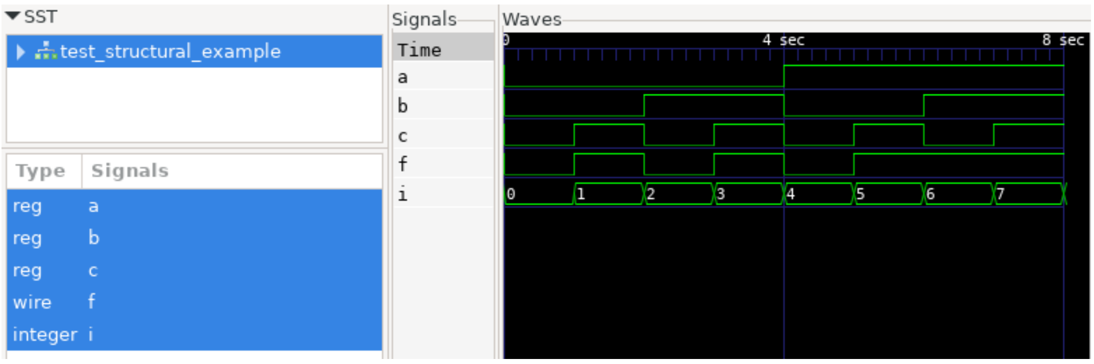


Figure 52: Testbench waveform for `structural_example`.

b) This circuit is slightly trickier to describe than the example in Figure 49 because of the negated gate inputs, which require additional wires and `not` gates. Nevertheless, the steps involved are largely the same.

Listing 2.4 shows one possible solution. This module is also named `structural_example` for convenience, in order to reuse the testbench in Listing 2.2 verbatim. Naming gates (e.g., `and1`, `and2`) is optional, but here it can help distinguish the two `and` gates.

Listing 2.4: A structural Verilog description of the circuit in Figure 50.

```
module structural_example (
  // Three input signals.
  input  a, b, c,
  // One output signal.
  output f
);

  // Four intermediate wires.
  wire not_a, not_b;         // Negated inputs.
  wire and1_out, and2_out;   // AND gate outputs.

  // Negated inputs to AND gates.
  not (not_a, a);
  not (not_b, b);

  // Intermediate AND gates.
  and and1 (and1_out, not_a, b);
  and and2 (and2_out, not_b, c);

  // Output NOR gate.
  nor (f, and1_out, and2_out);

endmodule
```

c) i) Listing 2.5 shows one way of structurally describing a Full-Adder, using only Verilog gate primitives. It is a relatively straightforward translation of the Full-Adder schematic in Figure 51, but some care is needed to avoid mistakes and identify which intermediate wires are needed. Writing gate-level descriptions like this at scale can be tedious and error-prone, which is why higher-level behavioral descriptions are often preferred in practice. Running the provided testbench `full_adder1_tb.v` should give the printed output in Listing 2.6.

Listing 2.5: A structural Verilog description of a Full-Adder using gate primitives.

```
module full_adder1 (
  input  x, y, cin,
  output s, cout
);

  // Intermediate wires.
  wire s1;  // First Half-Adder sum.
  wire c1;  // First Half-Adder carry.
  wire c2;  // Second Half-Adder carry.

  // First Half-Adder.
  xor (s1, x, y);
  and (c1, x, y);

  // Second Half-Adder.
  xor (s, cin, s1);
  and (c2, cin, s1);

  // Carry-out.
  or  (cout, c2, c1);

endmodule
```

Listing 2.6: Testbench output for `full_adder1`.

```
$ iverilog -o full_adder1 full_adder1.v full_adder1_tb.v
$ ./full_adder1
VCD info: dumpfile full_adder1.vcd opened for output.
Time  0, x=0, y=0, cin=0, s=0, cout=0
Time  1, x=0, y=0, cin=1, s=1, cout=0
Time  2, x=0, y=1, cin=0, s=1, cout=0
Time  3, x=0, y=1, cin=1, s=0, cout=1
Time  4, x=1, y=0, cin=0, s=1, cout=0
Time  5, x=1, y=0, cin=1, s=0, cout=1
Time  6, x=1, y=1, cin=0, s=0, cout=1
Time  7, x=1, y=1, cin=1, s=1, cout=1
full_adder1_tb.v:34: $finish called at 8 (1s)
```

ii) Listing 2.7 shows one way of structurally describing the same Full-Adder after abstracting and reusing the Half-Adder as a separate module. Note that, in this case, it does not matter whether the Half-Adder module is defined in the same or a separate `.v` Verilog source file. Running the provided testbench `full_adder2_tb.v` should give printed output similar to that in Listing 2.6.

Listing 2.7: A structural Verilog description of a Full-Adder using Half-Adders.

```
// Reusable Half-Adder module.
module half_adder (
  input  x, y,
  output s, c
);

  xor (s, x, y);  // Sum.
  and (c, x, y);  // Carry.

endmodule

// Full-Adder module built upon half_adder.
module full_adder2 (
  input  x, y, cin,
  output s, cout
);

  // Intermediate wires.
  wire s1;  // First Half-Adder sum.
  wire c1;  // First Half-Adder carry.
  wire c2;  // Second Half-Adder carry.

  // Two Half-Adder instances.
  half_adder ha1 (.x(x),   .y(y),  .s(s1), .c(c1));
  half_adder ha2 (.x(cin), .y(s1), .s(s),  .c(c2));

  // Carry-out.
  or (cout, c2, c1);

endmodule
```

[Exercise 19] Logic Circuits and Verilog: Behavioral Modeling

a) Recall that an n-to-1 Multiplexer (MUX) is a circuit which takes $n + 1$ inputs:

- n data inputs $x_1, \ldots, x_n$; and
- an m-bit selection signal s whose value determines which of the n data inputs to select as the circuit's output.

An example of a 2-to-1 MUX is shown in Figure 53.

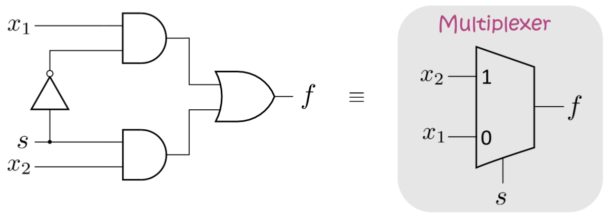


Figure 53: A 2-to-1 MUX.

In this question, you are asked to write behavioral Verilog descriptions of multiplexers using procedural statements (e.g., `if`-`else`) instead of structural descriptions using gates (e.g., `and`). Recall that procedural code should be wrapped in an `always` block. In particular, combinational circuits such as multiplexers can react to all module inputs by starting the block with the sensitivity list `always @*`, and they can set variables using the = blocking assignment operator.

Listing 2.8 shows a basic example of this syntax with a NOT gate.

Listing 2.8: A behavioral Verilog description of a NOT gate.

```
module behavioral_not (
  input       a,
  // Output declared as a reg variable so that it can
  // procedurally be assigned to within always.
  output reg f
);
  always @* begin
    f = ~a; // Executed every time the input changes.
  end
endmodule
```

i) Write a Verilog description of a 2-to-1 MUX using an `if`-`else` statement rather than gate primitives. Name your `module` `mux_2to1`, give it two data inputs `x1`, `x2`, a select input `s`, and one output `f`, and save it in a file named `mux_2to1.v`.

Check that your description is correct by inspecting its waveform in GTKWave. Listing 2.9 shows how you can achieve this with the testbench file `mux_2to1_tb.v` (available for download on Moodle).

Listing 2.9: Inspecting the 2-to-1 MUX waveform simulation in GTKWave.

```
$ iverilog -o mux_2to1 mux_2to1.v mux_2to1_tb.v
$ ./mux_2to1
$ gtkwave mux_2to1.vcd
```

ii) Write a Verilog description of a 3-bit 8-to-1 MUX using a `case` statement rather than gate primitives. Name your `module` `mux_8to1`. Give it eight 3-bit data inputs `x1`, ..., `x8`, a select input `s`, and one 3-bit output `f`.

Hint: Recall that the number of select bits in `s` changes with the number of inputs n.

As before, check that your description is correct by inspecting its waveform in GTKWave. This time, use the testbench file `mux_8to1_tb.v` (available for download on Moodle).

[Solution 19] Logic Circuits and Verilog: Behavioral Modeling

a) i) Listing 2.10 shows one way of describing a 2-to-1 MUX using an `if`-`else` statement. Running the provided testbench `mux_2to1_tb.v` as per Listing 2.9 should give the printed output in Listing 2.11 and the GTKWave waveforms in Figure 54.

Listing 2.10: A behavioral Verilog description of the circuit in Figure 53.

```
module mux_2to1 (
  // Two data inputs and one selection input.
  input      x1, x2, s,
  // One procedurally assignable reg output.
  output reg f
);

  // Select x1 when s=0, and x2 when s=1.
  always @* begin
    f = 0;
    if (s == 0) f = x1;
    else        f = x2;
  end

endmodule
```

Listing 2.11: Testbench output for `mux_2to1`.

```
$ iverilog -o mux_2to1 mux_2to1.v mux_2to1_tb.v
$ ./mux_2to1
VCD info: dumpfile mux_2to1.vcd opened for output.
Time  0, sel=0, x1=0, x2=0, out=0
Time  1, sel=0, x1=0, x2=1, out=0
Time  2, sel=0, x1=1, x2=0, out=1
Time  3, sel=0, x1=1, x2=1, out=1
Time  4, sel=1, x1=0, x2=0, out=0
Time  5, sel=1, x1=0, x2=1, out=1
Time  6, sel=1, x1=1, x2=0, out=0
Time  7, sel=1, x1=1, x2=1, out=1
mux_2to1_tb.v:33: $finish called at 8 (1s)
```

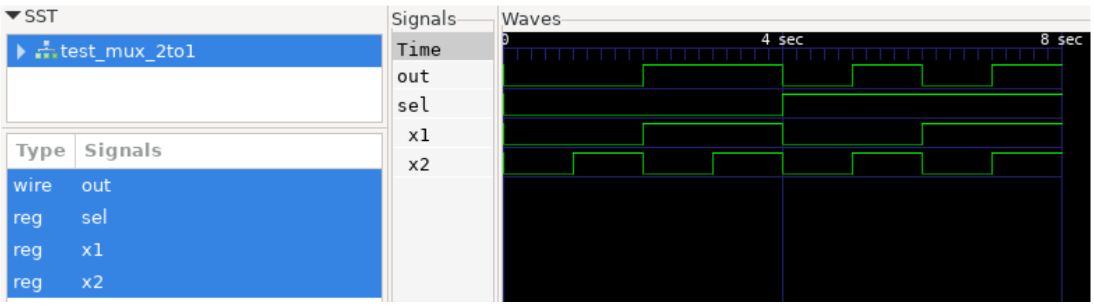


Figure 54: Testbench waveform for `mux_2to1`.

ii) Listing 2.12 shows one way of describing a 3-bit 8-to-1 MUX using a `case` statement. This is quite similar to the 2-to-1 MUX, except all ports are 3-bit vectors instead of single bits, and `case` provides a simpler syntax than `if-else` for comparing the same variable against multiple values. Running the provided testbench `mux_8to1_tb.v` should give the printed output in Listing 2.13.

Listing 2.12: A behavioral Verilog description of a 3-bit 8-to-1 MUX.

```
module mux_8to1 (
  // Eight 3-bit data inputs.
  input      [2:0] x1, x2, x3, x4,
  input      [2:0] x5, x6, x7, x8,
  // One 3-bit selection input.
  input      [2:0] s,
  // One procedurally assignable reg output.
  output reg [2:0] f
);

  always @* begin
    f = 0;
    case (s)
      // 3 stands for the bit width, b stands for binary.
      3'b000: f = x1;
      3'b001: f = x2;
      3'b010: f = x3;
      3'b011: f = x4;
      3'b100: f = x5;
      3'b101: f = x6;
      3'b110: f = x7;
      3'b111: f = x8;
    endcase
  end

endmodule
```

Listing 2.13: Testbench output for `mux_8to1`.

```
$ iverilog -o mux_8to1 mux_8to1.v mux_8to1_tb.v
$ ./mux_8to1
VCD info: dumpfile mux_8to1.vcd opened for output.
Time  0, sel=7, out=7
Time  1, sel=6, out=6
Time  2, sel=5, out=5
Time  3, sel=4, out=4
Time  4, sel=3, out=3
Time  5, sel=2, out=2
Time  6, sel=1, out=1
Time  7, sel=0, out=0
mux_8to1_tb.v:46: $finish called at 8 (1s)
```

[Exercise 20] 3-to-8 Binary Decoder

Write a Verilog module for a **3-to-8 binary decoder** (Wikipedia article). A binary decoder takes a binary input vector and activates one of output bits based on the input value. The 3-to-8 binary decoder will have an input vector of 3 bits (`in[2:0]`) and an output vector of 8 bits (`out[7:0]`). Each output bit corresponds to one of the possible binary combinations of the input. For example, a binary input of 010 will output 00000100, where the third bit is set to 1. Table 18 highlights the outputs for all inputs.

in[2]	**in[1]**	**in[0]**	**out[7]**	**out[6]**	**out[5]**	**out[4]**	**out[3]**	**out[2]**	**out[1]**	**out[0]**
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Table 18: 3-to-8 binary decoder

The module should have the following interface and name:

```
module binary_decoder (
    input [2:0] in,
    output reg [7:0] out
);
```

Hint: Think of using a `case` statement.

Note: Make sure to use meaningful signal names and adhere to proper Verilog syntax and coding conventions. Please refer to Verilog HDL Coding - Example of style.

[Solution 20] 3-to-8 Binary Decoder

The Verilog code for 3-to-8 binary decoder can be found below:

```
module binary_decoder (
    input [2:0] in,
    output reg [7:0] out
);

always @* begin
    out = 8'b00000000;
    case (in)
        3'b000: out = 8'b00000001;
        3'b001: out = 8'b00000010;
        3'b010: out = 8'b00000100;
        3'b011: out = 8'b00001000;
        3'b100: out = 8'b00010000;
        3'b101: out = 8'b00100000;
        3'b110: out = 8'b01000000;
        3'b111: out = 8'b10000000;
        default: out = 8'b00000000; // All outputs off by default
    endcase
end

endmodule
```

Note: Even when all the cases are covered, it is good practice to include `default` statement. The `default` statement helps to cover the case when input is unassigned.

The waveform from GTKWave is shown below:

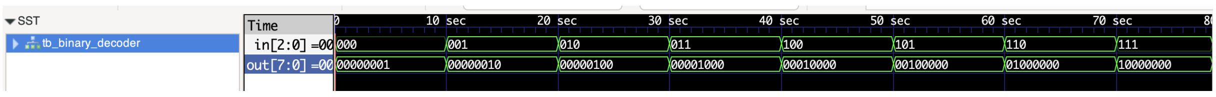


Figure 55: Waveform for 3-to-8 binary decoder.

[Exercise 21] Gray Code Encoder

Gray code is a number format where two consecutive decimal numbers in their binary representation differ by only 1 bit. For example, 1, 2, and 3 are represented as 001, 011, and 010, respectively (Wiki). A Gray code encoder converts an input binary vector into its Gray code representation.

Binary			**Gray code**		
B2	**B1**	**B0**	**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

Table 19: Binary to Gray code

Implement a Verilog module `gray_encoder` that takes as input a three-bit binary vector (`binary_in`) and a one-bit select signal (`select`), and outputs either the input binary vector unmodified or the Gray code. When the select signal is 0, the module should output the binary vector received at the input, and when it is 1, the module should output the Gray code.

The module should have the following interface and name:

```
module gray_encoder (
    input [2:0] binary_in,
    input select,
    output reg [2:0] out
);
```

Hint: Use an `if-else` statement to select between the binary input vector and Gray code. Use a `case` statement to select between outputs of Gray code.

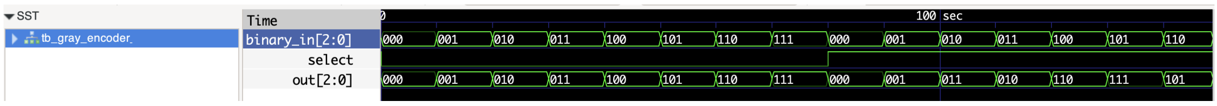
[Solution 21] Gray Code Encoder

The Verilog code for Gray code encoder can be found below:

```
module gray_encoder (
    input [2:0] binary_in,
    input select,
    output reg [2:0] out
);

always @* begin
    out = 3'b000;
    if (select == 0) begin
        // Output binary value if select is 0
        out = binary_in;
    end
    else begin
        // Otherwise, output Gray code value
        case (binary_in)
            3'b000: out = 3'b000; // Output 000 for input 000
            3'b001: out = 3'b001; // Output 001 for input 001
            3'b010: out = 3'b011; // Output 011 for input 010
            3'b011: out = 3'b010; // Output 010 for input 011
            3'b100: out = 3'b110; // Output 110 for input 100
            3'b101: out = 3'b111; // Output 111 for input 101
            3'b110: out = 3'b101; // Output 101 for input 110
            3'b111: out = 3'b100; // Output 100 for input 111
            default: out = 3'b000; // Default output 000
        endcase
    end
end

endmodule
```

Note: Even when all the cases are covered, it is good practice to include `default` statement. The `default` statement helps to cover the case when input is unassigned.

The waveform from GTKWave is shown below:



```
binary_in[2:0]  000 001 010 011 100 101 110 111 000 001 010 011 100 101 110
select
out[2:0]        000 001 010 011 100 101 110 111 000 001 011 010 110 111 101
```

Figure 56: Waveform for Gray code encoder.

[Exercise 22] Absolute Difference Calculator using Comparator

A comparator is a fundamental digital circuit component used to compare two binary numbers or signals and determine if they are equal to each other or greater than or less than the other. Wiki.

a) Design a module named `comparator` that compares two 4-bit inputs `A` and `B`. It should have three output signals: `A_eq_B`, `A_gt_B`, and `A_lt_B`, representing if `A` is equal to, greater than, or less than `B`, respectively.

b) Use the 4-bit comparator module in another module named `absolute_difference_calculator` to calculate the absolute difference between `A` and `B`. This module should take inputs `A` and `B` and output the result based on the comparison results. The module should have two 4-bit inputs `A` and `B` and one 4-bit output `result`. If `A` is equal to `B`, output should be 0; if `A` is greater than `B`, output should be `A` - `B`; otherwise, output should be `B` - `A`.

You can use the following code snippet as a template for your Verilog module:

```
module absolute_difference_calculator (
    // Your code here
);

    // Your code here

    // Instantiate the comparator module
    comparator comp(.A(A), .B(B), .A_eq_B(A_eq_B),
                    .A_gt_B(A_gt_B), .A_lt_B(A_lt_B));

    // Your code here

endmodule
```

[Solution 22] Absolute Difference Calculator using Comparator

a) The Verilog module for 4-bit comparator:

```
module comparator (
    input [3:0] A,
    input [3:0] B,
    output reg A_eq_B,
    output reg A_gt_B,
    output reg A_lt_B
);

    always @* begin
        A_eq_B = 0;
        A_gt_B = 0;
        A_lt_B = 0;

        if (A == B)
            A_eq_B = 1;
        else if (A > B)
            A_gt_B = 1;
        else
            A_lt_B = 1;
    end
endmodule
```

b) The Verilog module for `absolute_difference_calculator`:

```
module absolute_difference_calculator (
    input [3:0] A,
    input [3:0] B,
    output reg [3:0] result
);

    wire A_eq_B, A_gt_B, A_lt_B;
    comparator comp(.A(A), .B(B), .A_eq_B(A_eq_B),
                    .A_gt_B(A_gt_B), .A_lt_B(A_lt_B));

    // Multiplexer logic
    always @* begin
        result = 4'b0000;
        if (A_eq_B)
            result = 4'b0000;
        else if (A_gt_B)
            result = A - B;
        else
            result = B - A;
    end

endmodule
```

The waveform from GTKWave is shown below:

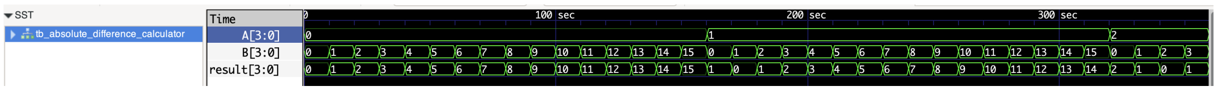

SST
tb_absolute_difference_calculator
Time
A[3:0]
B[3:0]
result[3:0]
0
100 sec
200 sec
300 sec
A: 0 1 2
B: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3
result: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 2 1 0 1

Figure 57: Waveform for absolute difference calculator.

[Exercise 23] Testbench for a Gray Code Encoder

Please write a Verilog testbench to validate the functionality of the Gray code encoder that you designed. As a part of this exercise, you will need to implement the following steps:

- Create a testbench module that instantiates the Gray code encoder module, and contains the required input and output wires.
- For *select* = 0, generate all possible input combinations of *binary_in* and check the output against the expected output (which should be the same as the input) for each case.
- For *select* = 1, generate all possible input combinations of *binary_in* and check the output against the expected output (which should be the Gray code of the input) for each case.

If the output matches the expected output for a particular input combination, output the following message to the console:

[*select_val binary_in_val*] Correct Output!

If the output does not match the expected output for a particular input combination, output the following message to the console:

[*select_val binary_in_val*] Error! Wrong Output!

Here, *select_val* is the value of *select* signal and *binary_in_val* is the value of *binary_in*.

[Solution 23] Testbench for a Gray Code Encoder

```
module tb_gray_encoder;

reg [2:0] binary_in;
reg select;
wire [2:0] out;

gray_encoder DUT(
    .binary_in(binary_in),
    .select(select),
    .out(out)
);

initial begin
    select = 1'b0;
    for(integer i = 0; i < 8; i = i + 1) begin
        binary_in = i;
        #10;
        if (out != i) begin
            $display("[0 %d] Error! Wrong Output!", i);
        end else begin
            $display("[0 %d] Correct Output!", i);
        end
    end

    select = 1'b1;
    binary_in = 3'b000;
    #10;
    if (out != 3'b000) begin
        $display("[1 0] Error! Wrong Output!");
    end else begin
        $display("[1 0] Correct Output!");
    end

    binary_in = 3'b001;
    #10;
    if (out != 3'b001) begin
        $display("[1 1] Error! Wrong Output!");
    end else begin
        $display("[1 1] Correct Output!");
    end

    binary_in = 3'b010;
    #10;
    if (out != 3'b011) begin
```

```
        $display("[1 2] Error! Wrong Output!");
    end else begin
        $display("[1 2] Correct Output!");
    end

    binary_in = 3'b011;
    #10;
    if (out != 3'b010) begin
        $display("[1 3] Error! Wrong Output!");
    end else begin
        $display("[1 3] Correct Output!");
    end

    binary_in = 3'b100;
    #10;
    if (out != 3'b110) begin
        $display("[1 4] Error! Wrong Output!");
    end else begin
        $display("[1 4] Correct Output!");
    end

    binary_in = 3'b101;
    #10;
    if (out != 3'b111) begin
        $display("[1 5] Error! Wrong Output!");
    end else begin
        $display("[1 5] Correct Output!");
    end

    binary_in = 3'b110;
    #10;
    if (out != 3'b101) begin
        $display("[1 6] Error! Wrong Output!");
    end else begin
        $display("[1 6] Correct Output!");
    end

    binary_in = 3'b111;
    #10;
    if (out != 3'b100) begin
        $display("[1 7] Error! Wrong Output!");
    end else begin
        $display("[1 7] Correct Output!");
    end
end
endmodule
```

[Exercise 24] Testbench for Absolute Difference Calculator using Comparator

Please write a Verilog testbench to validate the functionality of the absolute difference calculator that you designed. As a part of this exercise, you will need to implement the following steps:

- Create a testbench module that instantiates the absolute difference calculator module, and contains the required input and output wires.
- Generate all possible input combinations of *A* and *B* and check the output against the expected output (which should be the absolute difference of the inputs) for each case.

If the output matches the expected output for a particular input combination, output the following message to the console:

[*a_val b_val*] Correct Output!

If the output does not match the expected output for a particular input combination, output the following message to the console:

[*a_val b_val*] Error! Wrong Output!

Here, *a_val* is the value of *A* and *b_val* is the value of *B*.

[Solution 24] Testbench for Absolute Difference Calculator using Comparator

```
module tb_absolute_difference_calculator;

    reg [3:0] A;
    reg [3:0] B;
    wire [3:0] result;
    reg [3:0] exp_result;

    absolute_difference_calculator DUT(
        .A(A),
        .B(B),
        .result(result)
    );

    initial begin
        for(integer i = 0; i < 16; i = i + 1) begin
            for(integer j = 0; j < 16; j = j + 1) begin
                A = i;
                B = j;
                #10;
                exp_result = (A > B) ? A - B : B - A;
                if (result !== exp_result) begin
                    $display("[%d %d] Error! Wrong Output!", i, j);
                end else begin
                    $display("[%d %d] Correct Output!", i, j);
                end
            end
        end
    end

endmodule
```

[Exercise 25] Creating a Sequential Circuit using D Flip-Flop

a) Consider the expression $Y = A \cdot B + C$. Add an extra condition: If the value of Y in the previous clock cycle was 1, then the output Y should be 0 in the current clock cycle, irrespective of the current values of A, B, and C. Otherwise, if the value of Y in the previous clock cycle was 0, then the output Y in the current clock cycle should be equal to $A \cdot B + C$.

Design this circuit (on pen and paper) using a D flip-flop.
Assume that the CLK, R and S pins are driven from external sources.

Complete the following steps:

- Write the Boolean expression for the output $Y(T)$.
- Draw the circuit diagram for the sequential circuit.
  Hint: Use one D flip-flop and combinational logic.
- Complete the timing diagram shown in Fig. 58 for this circuit.
  Note: Assume that the reset signal (R) is active-high and synchronous.

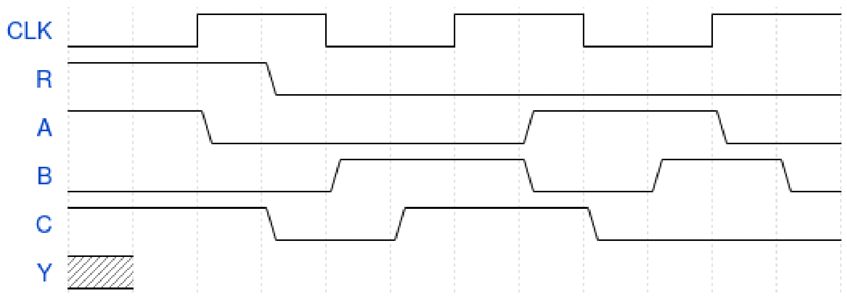


Figure 58: Timing Diagram for the custom sequential circuit

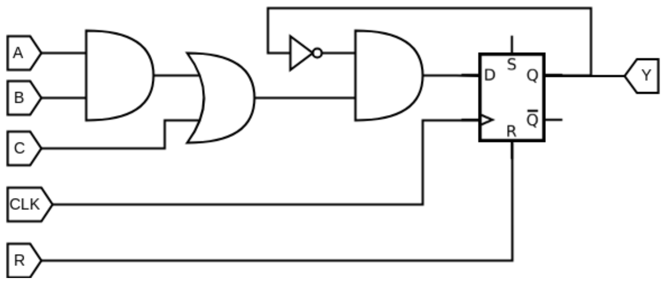
[Solution 25] Creating a Sequential Circuit using D Flip-Flop

If $Y(T-1) = 1$, then $Y(T) = 0$.
If $Y(T-1) = 0$, then $Y(T) = A \cdot B + C$.

We can formulate these two conditions with a characteristic table (equivalent to the concept of a truth table in combinational circuits) as shown in Table 59a, considering $Y(T-1)$ as an input. We can then derive a reduced Boolean expression for $Y(T)$ from the characteristic table. $Y(T) = \overline{Y(T-1)} \cdot (A \cdot B + C)$. Once you have the Boolean expression, you can draw the circuit diagram, as shown in Fig. 59b.

Y(T-1)	A	B	C	Y(T)
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
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	0
1	1	1	1	0

(a) Characteristic table



(b) Custom sequential circuit using D Flip-Flop

Figure 59: Example usage of D Flip-Flop in a sequential circuit

The timing diagram for the custom sequential circuit is shown in Fig. 60. You can use the Boolean expression/circuit in Fig. 59b to find the value of Y at each clock cycle.

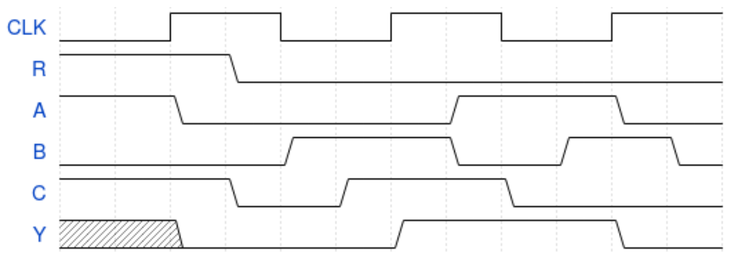


Figure 60: Timing Diagram for the custom sequential circuit

[Exercise 26] Creating a D Flip-Flop in Verilog

As explained previously, D flip-flops are the basic building blocks of sequential circuits. For this exercise, please write a verilog module for a D flip-flop.

The module should have the following interface and name:

```
module D_FF (
    input clk,
    input reset,
    input set,
    input D,
    output reg Q,
    output reg Q_n
);
```

The module should behave exactly as the D flip-flop described in the tutorial.
The functionality of the module should be as follows:

- The `reset` and `set` inputs are used to clear/set the output Q respectively.
- The `reset` and `set` inputs are synchronous.
- The `reset` signal should have higher priority over the `set` signal.
- The output `Q` should be equal to the input `D` at the rising edge of the clock signal, if neither `reset` and `set` signals are set.
- The output `Q_n` should be the complement of `Q` at all times.

Hint: Think of using `always` block with `posedge` conditions.

You are encouraged to write your own testbench to test your own design. You can also use the testbench provided in the next page and follow the instructions to check if your design is working correctly.

Execute the following testbench to test your D flip-flop module:

```
module tb_D_FF;
    reg clk, reset, set, D;
    wire Q, Q_n;
    D_FF D_FF_0 (.clk(clk), .reset(reset), .set(set),
                 .D(D), .Q(Q), .Q_n(Q_n));
    initial begin
        $dumpfile("tb_D_FF.vcd");
        $dumpvars(0, tb_D_FF);
        clk = 0; reset = 1; set = 0; D = 1;
        #7;
        D = 0;
        #2;
        reset = 0;
        #3;
        D = 1;
        #5;
        D = 0;
        #10;
        set = 1;
        #13;
        $finish;
    end
    always begin
        #5 clk = ~clk;
    end
endmodule
```

Visualize the waveforms on GTKWave, the waveforms should match Fig. 61.

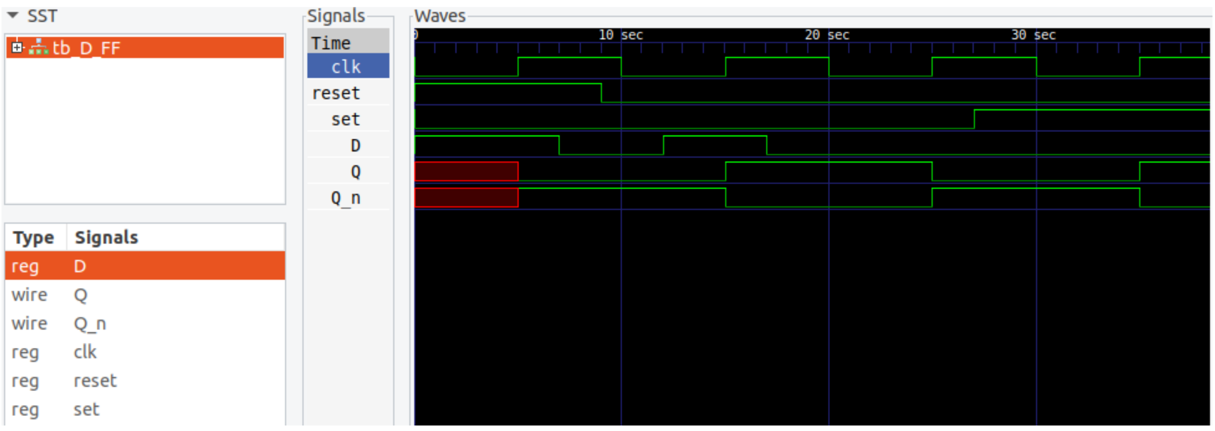


Figure 61: Waveforms for the D Flip-Flop

[Solution 26] Creating a D Flip-Flop in Verilog

Below is the Verilog for DFF:

```
module D_FF (
    input clk,
    input reset,
    input set,
    input D,
    output reg Q,
    output reg Q_n
);

    always@(posedge clk) begin
        if(reset) begin
            Q <= 1'b0;
            Q_n <= 1'b1;
        end
        else if(set) begin
            Q <= 1'b1;
            Q_n <= 1'b0;
        end
        else begin
            Q <= D;
            Q_n <= ~D;
        end
    end

endmodule
```

[Exercise 27] Timing diagrams of different types of D flip-flops

Consider the timing diagram in Fig. 63. Assuming that the `D` and `CLK` inputs shown are applied to the circuit in Fig. 62, draw waveforms for the Q_a and Q_b signals.

(Hint: The top flip-flop is triggered on the rising edge of the clock signal, while the bottom flip-flop is triggered on the falling edge of the clock signal.)

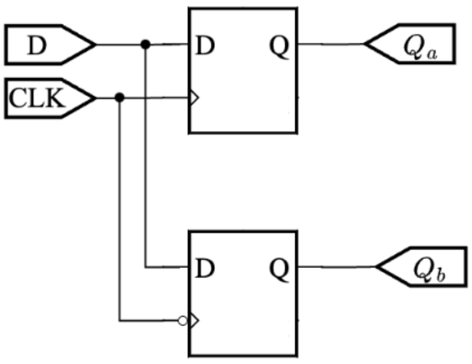


Figure 62: Sequential Circuit with two D flip-flops

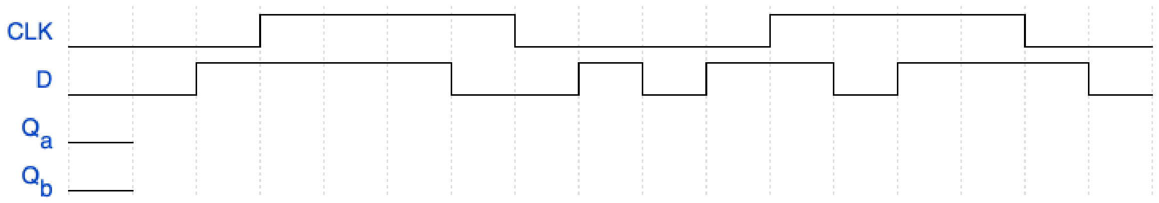


Figure 63: Timing Diagram for waveforms

[Solution 27] Timing diagrams of different types of D flip-flops

The first D Flip-Flop updates its output on the rising edge of the clock signal. The second D Flip-Flop updates its output on the falling edge of the clock signal. The waveforms of the outputs of the D Flip-Flops are shown in Figure 64.

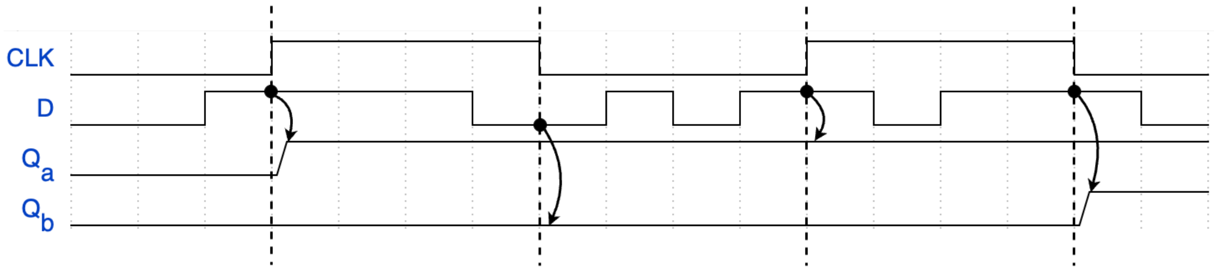


Figure 64: Timing Diagram of the outputs of the D Flip-Flops

[Exercise 28] Analysing the behavior of sequential circuits

Consider the sequential circuit shown in Fig. 65. The circuit consists of two 4-bit counters and a few logic gates. The functionality of each 4-bit counter is as follows:

1. The `Enable` input is used to enable the counter, i.e., if the `Enable` input is 0, the counter is disabled and should not increment its value.
2. The `Load` input is used to load the counter, i.e., if the `Load` input is 1, the counter should load the value present at $\{D_0, D_1, D_2, D_3\}$.
3. In the absence of the `Load` signal, the counter increments by 1 at each clock cycle.

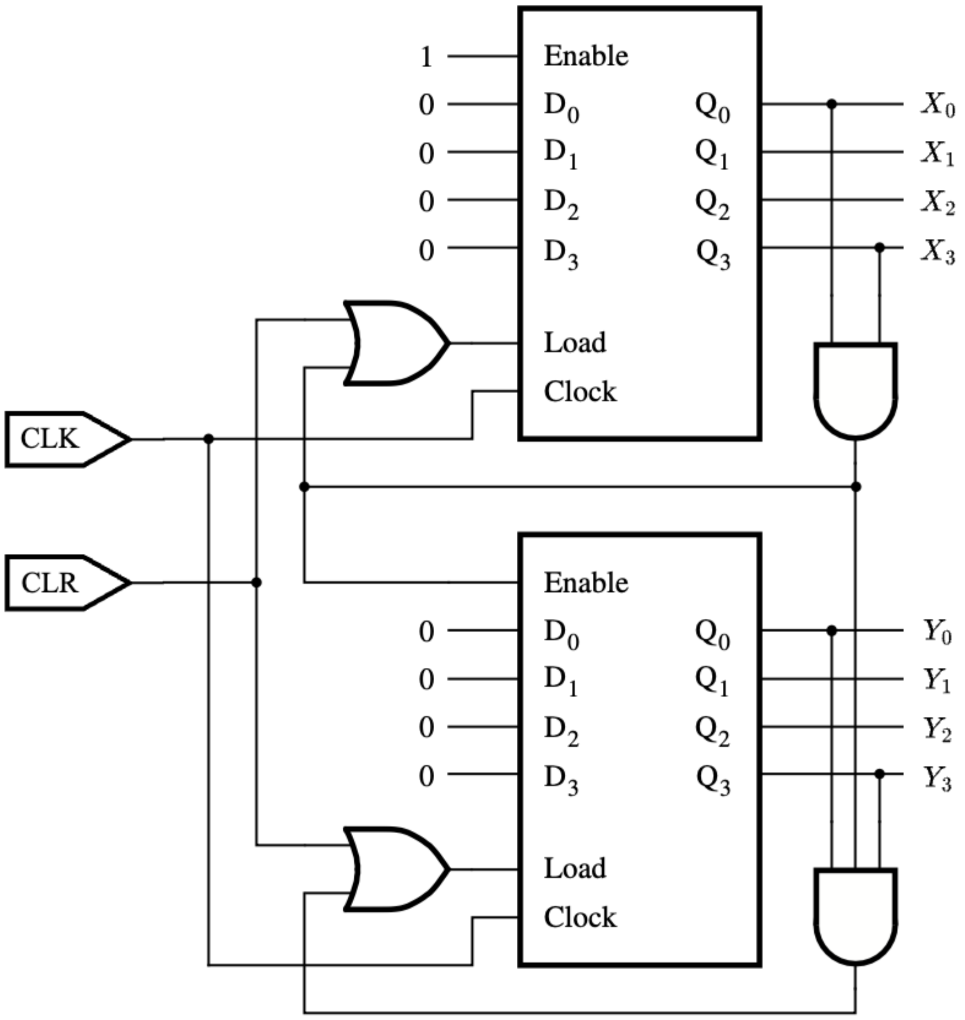


Figure 65: Sequential Circuit consisting of two counters

Answer the following questions with respect to the circuit shown in Fig. 65

(Note: For Parts 1, 2, and 3, you can assume that the clear signal (`CLR`) is active only in the first clock cycle and then inactive forever. The `CLK` signal behaves as a regular clock signal.)

1. Given that both counters start from Q_0, Q_1, Q_2, Q_3 = 0000, what is the sequence of values arising at $X(X_0, X_1, X_2, X_3)$ and $Y(Y_0, Y_1, Y_2, Y_3)$?
2. Calculate the number of possible states for:
   (a) One of the 4-bit counters in isolation
   (b) The complete circuit (including the two 4-bit counters and the logic gates)
3. Based on your answer to Part 1, what is the functionality of this sequential circuit?
4. Implement the sequential circuit shown in Fig. 65 using Verilog.
   The module should have the following interface and name:

```
module counter_circuit(
    input clk,
    input clr,
    output reg [3:0] X,
    output reg [3:0] Y
);
```

   Test your implementation using the testbench (tb_counter_circuit.v) provided on Moodle. Try to understand how the testbench works and check if the output of the testbench matches the sequence you found in Part 1.

[Solution 28] Analysing the behavior of sequential circuits

First, derive the boolean expressions the Load and Enable signals of the two counters. For Counter X,

1. Load signal: $L_X = (X_0 \cdot X_3) + CLR$
2. Enable signal: $E_X = 1$

For Counter Y,

1. Load signal: $L_Y = (Y_0 \cdot Y_3 \cdot (X_0 \cdot X_3)) + CLR$
2. Enable signal: $E_Y = (X_0 \cdot X_3)$

In the beginning, $X = 0000$ and $Y = 0000$, so $L_X = 0$, $L_Y = 0$ and $E_Y = 0$. (Note that the CLR signal is used only in the first clock cycle to load the counters to 0000, and then it is always 0, so it doesn't affect the boolean expressions.) Counter X increments by 1 ($X = 0001$), and Counter Y remains the same ($Y = 0000$). The L_X, L_Y and E_Y signals do not change and Counter X keeps incrementing by 1 until it reaches $X = 1001$. Y remains fixed ($= 0000$) for this entire duration.

When $X = 1001$, $L_X = 1$ and $E_Y = 1$. Hence, in the next clock cycle, Counter X will load 0000 ($X = 0000$) and Counter Y will increment by 1 ($Y = 0001$). As a result, again $L_X = 0$, $L_Y = 0$ and $E_Y = 0$. X will keep incrementing by 1 until it reaches $X = 1001$, and Y will remain fixed at $Y = 0001$. This process will keep repeating.

At the very end when, when $X = 1001$ and $Y = 1001$, $L_X = 1$, $L_Y = 1$ and $E_Y = 1$. So, both X and Y will load 0000 in the next clock cycle. And the entire process starting from $X = 0000$ and $Y = 0000$ will repeat.

1. The sequence (Y, X) observed is as follows:
   0000 0000 (= 0 0)
   0000 0001 (= 0 1)
   0000 0010 (= 0 2)
   $\vdots$
   0000 1001 (= 0 9)
   0001 0000 (= 1 0)
   0001 0001 (= 1 1)
   0001 0010 (= 1 2)
   $\vdots$
   1001 1001 (= 9 9)
   0000 0000 (= 0 0)

2. (a) Each 4-bit counter in isolation can take any value between 0000 and 1111, so 16 states are possible.

   (b) However, when combined with logic gates as shown in Fig. 65, each counter can only take values between 0000 and 1001, so 10 states are possible for each counter, and in total $10 * 10 = 100$ states are possible. Note that these 100 states correspond to the same sequence found in Part 1.

3. The sequence found in Part 1 is the sequence of BCD numbers from 00 to 99, with the value of each digit represented by the output of each counter. Hence, this circuit functions as a BCD counter.

4. The verilog code for the sequential circuit as shown in Fig. 65 is as follows:

```
module counter_circuit(
    input clk,
    input clr,
    output reg [3:0] X,
    output reg [3:0] Y
);
    // Boolean logic for the load and enable signals
    wire load_X, load_Y, enable_X, enable_Y;
    assign enable_X = 1'b1;
    assign enable_Y = X[0] & X[3];
    assign load_X = clr | (X[0] & X[3]);
    assign load_Y = clr | ((X[0] & X[3]) & Y[0] & Y[3]);

    always@(posedge clk) begin // Counter for X
        if(load_X == 1'b1) begin
            X <= 4'b0000;
        end else begin
            if(enable_X == 1'b1) begin
                X <= X + 1;
            end
        end
    end
    always@(posedge clk) begin // Counter for Y
        if(load_Y == 1'b1) begin
            Y <= 4'b0000;
        end else begin
            if(enable_Y == 1'b1) begin
                Y <= Y + 1;
            end
        end
    end
endmodule
```

   You should expect the following output after running the testbench:

```
Y =  0, X =  0
Y =  0, X =  1
Y =  0, X =  2
⋮
Y =  9, X =  8
Y =  9, X =  9
Y =  0, X =  0
Y =  0, X =  1
Y =  0, X =  2
Y =  0, X =  3
```

The testbench generates the same sequence of BCD numbers as found in Part 1. You can also visualise the input and output waveforms using GTKWave. The waveforms should look like the ones shown in Fig. 66.

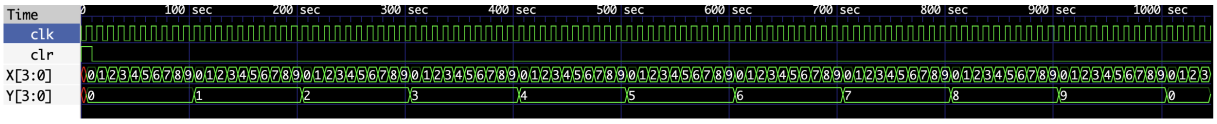


Figure 66: Waveforms of the sequential circuit

[Exercise 29] Linear-Feedback Shift Register

Linear-feedback shift registers (LFSRs) are N-bit counters exhibiting *pseudo-random* behavior. LFSRs are realized as shift registers, with input bits as linear functions of the output bits. At the output, LFSRs generate a periodic sequence of pseudo-random numbers. The period of the sequence (i.e., the number of clock cycles before the sequence repeats) does not exceed $2^N - 1$ clock cycles, where N is the number of flip-flops in the LFSR. LFSRs are commonly used in cryptography, computer graphics, automatic testing, and error detection and correction. To learn more about LFSRs, visit the Wiki: Link.

Figure 67 shows an example three-bit LFSR. The inputs are the clock and the asynchronous active-high power-on reset (not shown). There are three flip-flops; their outputs are Q_2, Q_1, and Q_0. The output of the LFSR is a three-bit binary vector $Q_2Q_1Q_0$. The flip-flop input bits are linear functions of the output bits. To generate a sequence of pseudo-random numbers, the LFSR must be initialized with a value different than all zeros. This initial value is commonly referred to as the *seed* of the LFSR.

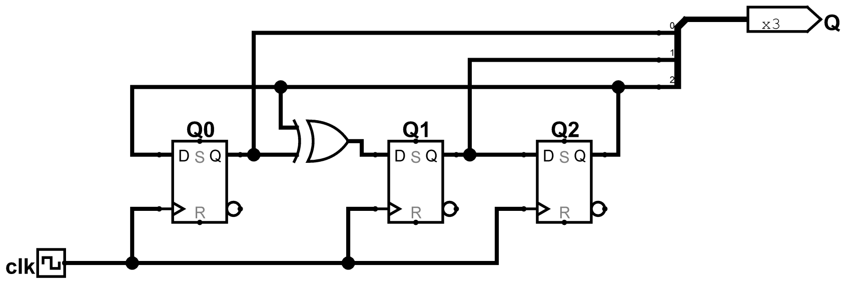


Figure 67: Three-bit linear-feedback shift register

a) Assuming the seed $(001)_2$, what is the sequence of pseudo-random numbers generated by the LFSR in the subsequent seven clock cycles?

b) Describe what is missing in the LFSR circuit in Figure 67 to allow loading the LFSR seed in a single clock cycle (i.e., parallel loading of the seed).

c) Write the Verilog model of the complete three-bit LFSR module (`lfsr`).

d) Write a testbench to verify the functionality of the LFSR.

[Solution 29] Linear-Feedback Shift Register

a) At each clock cycle, Q_2 is assigned Q_1, Q_1 is assigned Q_2 XOR Q_0, and Q_0 is assigned Q_2. The sequence of the LFSR for the first seven clock cycles is as follows:

- Clock cycle 0: $Q_2Q_1Q_0 = 001$
- Clock cycle 1: $Q_2Q_1Q_0 = 010$
- Clock cycle 2: $Q_2Q_1Q_0 = 100$
- Clock cycle 3: $Q_2Q_1Q_0 = 011$
- Clock cycle 4: $Q_2Q_1Q_0 = 110$
- Clock cycle 5: $Q_2Q_1Q_0 = 111$
- Clock cycle 6: $Q_2Q_1Q_0 = 101$
- Clock cycle 7: $Q_2Q_1Q_0 = 001$ (repeats)

b) To allow loading the seed value into the LFSR, we need to add a multiplexer at the input of the flip-flops and a 1-bit load signal. The multiplexer selects between the seed value and the feedback value based on the load signal. Therefore, seed value and feedback act as inputs of the multiplexer and load acts as the select signal. Figure 68 shows the circuit diagram with multiplexers to load the seed.

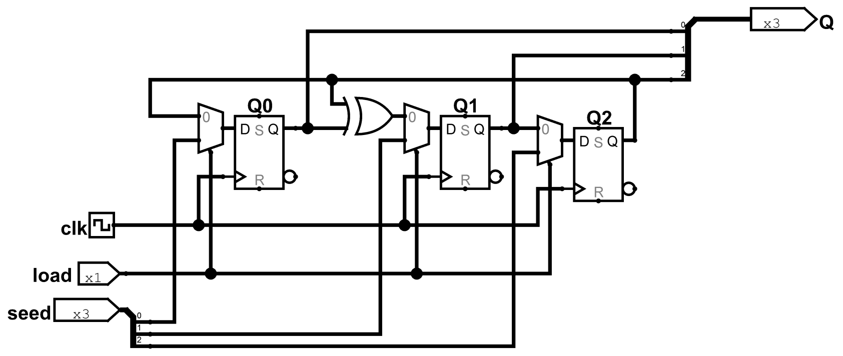


Figure 68: Three-bit linear-feedback shift register with seed loading

c) Below is the Verilog description of the module:

Note: If you copy the Verilog code below, please retype the caret character (ˆ) as special characters are sometimes misinterpreted or corrupted during copying.

```
module lfsr(
    input clk, reset, load,
    input [2:0] seed,
    output reg [2:0] out
);

  always @(posedge clk or posedge reset) begin
    if (reset == 1) begin
      out <= 0;
    end else if (load == 1) begin
      out <= seed;
    end else begin
      out <= {
              out[1],             // D2 = Q1
              out[2] ^ out[0],    // D1 = Q2 XOR Q0
              out[2]};            // D0 = Q2
    end
  end

endmodule
```

d) Below is a testbench to verify the functionality of the LFSR on the sequence after initialization with the seed value 001:

```
module testbench;
  reg clk, reset, load;
  reg [2:0] seed;
  wire [2:0] out;

  lfsr dut (.clk(clk), .reset(reset), .load(load), .seed(seed), .out(out));

  initial begin
    $dumpfile("tb_lfsr.vcd");
    $dumpvars(0, testbench);

    clk = 0;
    reset = 0;
    load = 1;
    seed = 3'b001;
    #20 load = 0;

    if (out != 3'b001)
        $error("Cycle 0 failed; expected 3'b001, got %b", out);
    #10 if (out != 3'b010)
        $error("Cycle 1 failed; expected 3'b010, got %b", out);
    #10 if (out != 3'b100)
        $error("Cycle 2 failed; expected 3'b100, got %b", out);
    #10 if (out != 3'b011)
        $error("Cycle 3 failed; expected 3'b011, got %b", out);
    #10 if (out != 3'b110)
        $error("Cycle 4 failed; expected 3'b110, got %b", out);
    #10 if (out != 3'b111)
        $error("Cycle 5 failed; expected 3'b111, got %b", out);
    #10 if (out != 3'b101)
        $error("Cycle 6 failed; expected 3'b101, got %b", out);
    #10 if (out != 3'b001)
        $error("No wrap-around; expected 3'b001, got %b", out);
    $finish;
  end

  always begin
    #5 clk = ~clk;
  end
endmodule
```

[Exercise 30] Up/Down counter

a) Design a 4-bit synchronous counter in Verilog that can count both up or down starting from any value provided by the user. The functionality of this module should be as follows:

- There should be a synchronous active-high `reset` input to clear the counter.
- There should be a `load` and a 4-bit `load_val` input such that the user should be able to load a 4-bit value into the counter by making the `load` input high.
- There should be an `up` input to select the direction of the counter. If `up` is high, the counter should increment; otherwise, it should decrement.
- The counter should handle overflows, i.e., if the counter is at its maximum value and is incremented, it should go back to zero, and if it is at zero and is decremented, it should go to its maximum value.

You are advised to use the ternary operator to select between the up and down counter functionality. The ternary operator is a conditional operator that provides a shorter syntax for the if-else statement. The syntax is as follows:

```
<condition> ? <expression1> : <expression2>
```

If the condition is true, the operator returns the value of `expression1`; otherwise, it returns the value of `expression2`.

b) Create a comprehensive testbench to verify the functionality of the counter.

[Solution 30] Up/Down counter

a) Below is a Verilog description of the module:

```
module up_down_counter(
    input clk,
    input reset,
    input load,
    input [3:0] load_val,
    input up,
    output reg [3:0] count
);

    always @(posedge clk) begin
        if (reset == 1) begin
            count <= 4'b0000;
        end else begin
            if (load == 1) begin
                count <= load_val;
            end else begin
                count <= count + ((up == 1) ? 1 : -1);
            end
        end
    end

endmodule
```

In the module above, we used the ternary operator to select between incrementing and decrementing the counter based on the value of the select signal. It is equivalent to the following if...else statement:

```
if (up == 1)
    count = count + 1;
else
    count = count - 1;
```

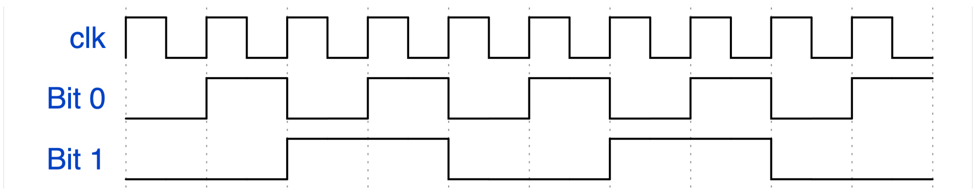
b) Below is a comprehensive testbench to verify the functionality of the counter:

```
module testbench;

    reg clk, reset, load, up;
    reg [3:0] load_val;
    wire [3:0] count;

    up_down_counter dut (
        .clk(clk),
        .reset(reset),
        .load(load),
        .load_val(load_val),
        .up(up),
        .count(count)
    );

    initial begin
        $dumpfile("tb_up_down_counter.vcd");
        $dumpvars(0, testbench);

        clk = 0;
        reset = 0;
        up = 0;
        load = 0;
        load_val = 0;

        #10 reset = 1;
        #10 reset = 0;
        if (count == 4'b0000) begin
            $display("Reset test passed");
        end else begin
            $error("Reset test failed");
        end

        up = 1;
        #100

        if (count == 4'b1010) begin
            $display("Increment test passed");
        end else begin
            $error("Increment test failed");
        end

        up = 0;
        #100
```

```
            if (count == 4'b0000) begin
                $display("Decrement test passed");
            end else begin
                $error("Decrement test failed");
            end

            #10 load = 1;
            load_val = 4'b1111;
            #10 load = 0;
            if (count == 4'b1111) begin
                $display("Parallel load test passed");
            end else begin
                $error("Parallel load test failed");
            end
            $finish;
        end

        always begin
            #5 clk = ~clk;
        end
    endmodule
```

[Exercise 31] Generating a Periodic Output

Consider an N-bit counter that increments on the rising edge of the clock signal and goes back to zero after reaching the maximum value or with a synchronous active-high reset signal. If we check the waveform of each bit of the counter, we see that they all are periodic signals with different frequencies. Bit zero toggles at every clock cycle, so it generates a periodic square wave with a frequency of $f_{\mathrm{CLK}}/2$. Similarly, bit one toggles at every two clock cycles, so it generates a periodic square wave with a frequency of $f_{\mathrm{CLK}}/4$ and so on. The Figure below shows the waveform for bit zero and bit one.



To create such a waveform with $f_{\mathrm{CLK}}/4$, a 2-bit counter can be used as it would create the following sequence: $(00)_2, (01)_2, (10)_2, (11)_2, (00)_2 \ldots$ and this sequence's most significant bit would be zero for two cycles and one for two cycles. In other words, a 2-bit counter's most significant bit would have the period of four cycles, and as a result a frequency of $f_{\mathrm{CLK}}/4$.

Design a counter whose most significant bit can be used to generate a periodic signal with a frequency of $f_{\mathrm{CLK}}/4096$.

a) What is the number of flip-flops required to design this counter?

b) Implement this counter in Verilog.

c) Create a comprehensive testbench to verify the functionality of the counter.

[Solution 31] Generating a Periodic Output

a) The counter should count up to 4096 to generate a periodic signal with a frequency of $f_{\text{CLK}}/4096$. The MSB will be low for the first 2048 clock cycles, high for the next 2048 clock cycles, low for the next 2048 clock cycles, and so on.

To count up to 4096, we will need 12 bits ($\lceil \log_2(4096) \rceil = 12$ bits), which results in a 12-bit counter, and to store each bit, we will need 12 flip-flops.

b) Below is the Verilog description of the module:

```verilog
module periodic_output(
    input clk,
    input rst,
    output reg out
);

    reg [11:0] count;

    always @(posedge clk)
    begin
        if (rst == 1) begin
            count <= 12'b0000_0000_0000;
        end else begin
            count <= count + 1;
        end
    end

    always @(*) begin
        out = count[11];
    end

endmodule
```

c) Below is the testbench for the module:

```
module tb_periodic_output;

    reg clk, rst;
    wire out;

    integer expected;

    periodic_output dut (
        .clk(clk),
        .rst(rst),
        .out(out)
    );

    initial begin
        $dumpfile("dump.vcd");
        $dumpvars(0, tb_periodic_output);

        clk = 0;
        rst = 1;
        #10;
        rst = 0;
        for (integer i = 1; i < 5000; i = i + 1) begin
            #10;
            expected = (i%4096)/2048;
            if (out != expected) begin
                $display("Error: out = %d, expected = %d, at time = %d",
                            out, expected, i);
                $finish;
            end
        end
        $display("Pass");
        $finish;
    end

    always begin
        #5 clk = ~clk;
    end

endmodule
```

[Exercise 32] Modulo Six Counter

In general, a k-bit counter can count from zero to $2^k - 1$, overflow back to zero, and repeat the cycle. For example, a 2-bit counter counts in the sequence zero, one, two, three, zero, one, two, three, and so on.

This 2-bit counter is called a modulo four counter because it counts from zero to three. The value of the counter at clock cycle n is $n \mod 4$. So, a k-bit counter can be referred to as a modulo 2^k counter.

However, how can a modulo N counter be designed such that N is not a power of two? Consider a modulo six counter that counts in the sequence zero, one, two, three, four, five, zero, one, two, three, four, five, and so on.

In this exercise, we explore the design of a modulo six counter.

a) What is the minimum number of flip-flops required to design the counter?

b) Implement a synchronous modulo six counter in Verilog. The module should use the minimum number of bits (that was found in part a) to store the counter value. The Verilog module should have the following functionality:

- There should be a synchronous active-high `reset` input to clear the counter.
- There should be an `enable` input to enable the counter, i.e., if the `enable` input is zero, the counter is disabled and should not change its value, otherwise, the counter increments.
- The counter should overflow back to zero after reaching five.

c) Create a comprehensive testbench to verify the functionality of the modulo six counter.

[Solution 32] Modulo Six Counter

a) The counter can take six possible values (zero, one, two, three, four, five). Hence, we need $\lceil \log_2(6) \rceil = 3$ bits to represent the values. As each flip-flop can only store one bit, the minimum number of flip-flops required are three.

b) Below is the Verilog description of the module

```
module modulo_6_counter(
    input clk,
    input reset,
    input enable,
    output reg [2:0] count
);

    always @(posedge clk) begin
        if (reset == 1) begin
            count <= 3'b000;
        end else begin
            if (enable == 1) begin
                if (count == 3'b101) begin
                    count <= 3'b000;
                end else begin
                    count <= count + 1;
                end
            end else begin
                count <= count;
            end
        end
    end

endmodule
```

c) Below is the testbench for the module:

```
module testbench;

    reg clk, reset, enable;
    wire [2:0] count;

    modulo_6_counter dut (
        .clk(clk),
        .reset(reset),
        .enable(enable),
        .count(count)
    );

    initial begin
        $dumpfile("tb_modulo_6_counter.vcd");
        $dumpvars(0, testbench);

        clk = 0;
        reset = 1;
        enable = 0;
        #10;
        reset = 0;
        enable = 1;
        for(integer i = 1; i < 10; i = i + 1) begin
            #10;
            if (count == i%6) begin
                $display("Pass: count = %d, expected = %d", count, i%6);
            end else begin
                $display("Error: count = %d, expected = %d", count, i%6);
            end
        end
        enable = 0;
        #10;
        if (count == 3) begin
            $display("Pass: count = %d, expected = %d", count, 3);
        end else begin
            $display("Error: count = %d, expected = %d", count, 3);
        end
        $finish;
    end

    always begin
        #5 clk = ~clk;
    end

endmodule
```

[Exercise 33] Mystery Circuit

Consider the following synchronous circuit with one input b and one output x. The reset is not shown in the diagram.

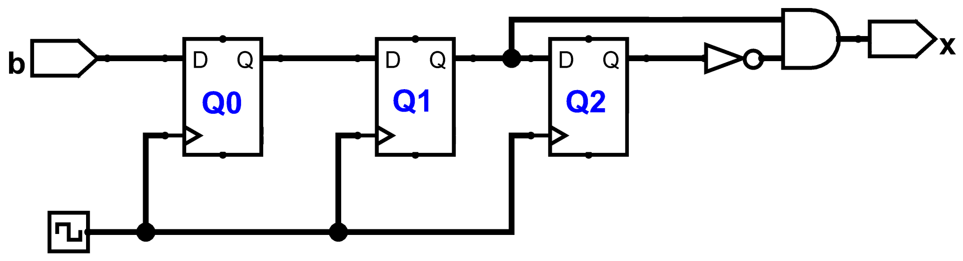


Figure 69: Mystery Circuit

a) Given the waveform template below, fill in the values of the signals assuming zero delay for both the gates and flip-flops.

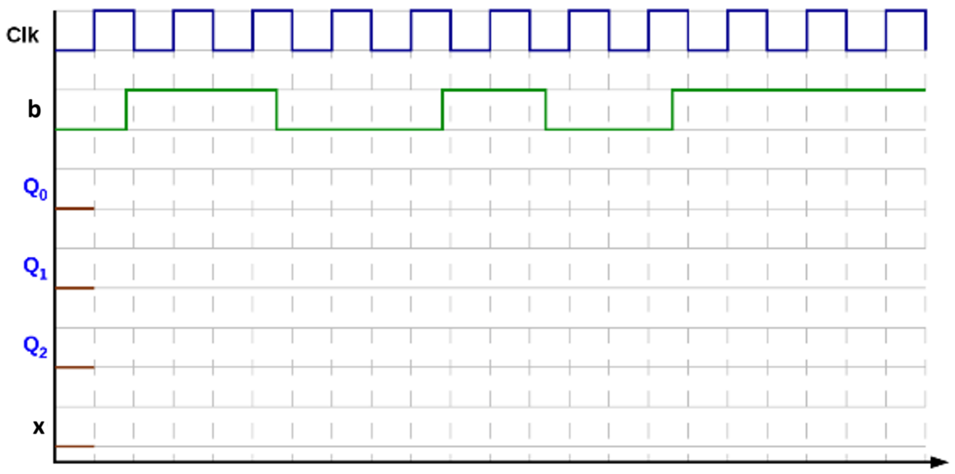


Figure 70: Waveform Template

b) Briefly describe with words the functionality of this circuit. Also, explain the purpose of using three flip-flops.

[Solution 33] Mystery Circuit

a) The waveform is as follows:

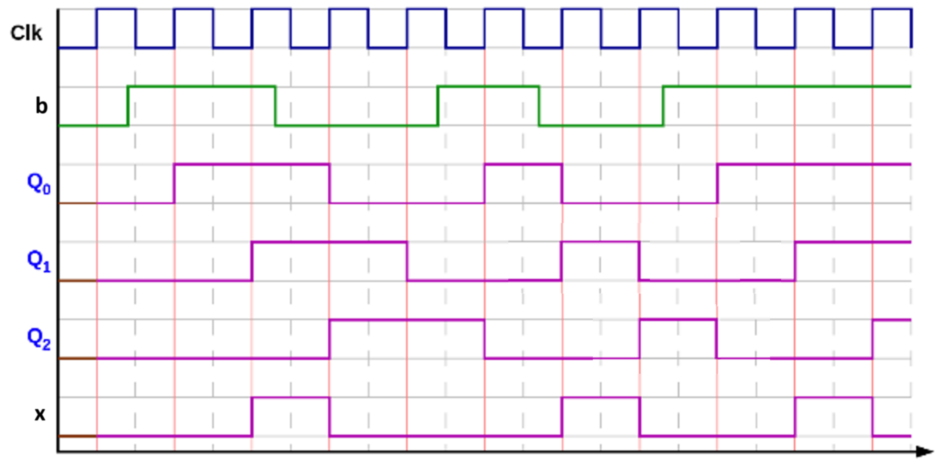


Figure 71: Waveform Solution

b) The circuit is a rising edge detector, which generates a short pulse when the input signal `b` transitions from low to high. The way it works is as follows:

1. When the input b transitions from zero to one, the first D flip-flop captures the value of `b` at the rising edge of the clock signal. This stores the current state of `b` in `Q0`.
2. On the next rising clock edge, the second D flip-flop captures the value of `Q0`, storing it in `Q1`. This means `Q1` now holds the previous state of `b`.
3. On the clock edge after that, the third D flip-flop captures the value of `Q1`, storing it in `Q2`. This means `Q2` now holds the state of b two clock cycles ago.
4. The final combinatorial logic, a two-input AND gate, compares `Q1` and the negation of `Q2`. This will result in a high output `x` only when `Q1` is high and `Q2` is low indicating that the signal was low two cycles ago and it was high one cycle ago, which indicates a rising edge.
5. This high output `x` will last for one clock cycle, two clock cycles after the rising edge on `b` occurred.

The reason why there are three flip-flops is to avoid any metastability issues. The metastability is more likely to occur at the first flip-flop, and then the possibility decreases with each flip-flop. As the output of the first flip-flop (Q0) is more likely to have metastability issues, we do not use it for the output, but instead, we use the output of the second flip-flop (Q1) and the output of the third flip-flop (Q2) to detect the rising edge.

[Exercise 34] Analysing Finite State Machines (FSMs)

Consider the sequential circuit shown in Fig. 72. Answer the following questions:

1. Calculate the Boolean expressions for Y_1, Y_2 and Z.
2. Derive the state/output table for the sequential circuit. (Hint: Use the Boolean expressions derived in Part 1. y_1 and y_2 represent the current state, Y_1 and Y_2 represent the next state, W represents the input, and Z represents the output.)
3. Based on the state/output table you derived, what is the functionality of this sequential circuit?

Note: You can assume that the `CLK` signal behaves as a regular clock signal, and the `R` signal behaves simply as an asynchronous reset (power-on reset) of the FFs.

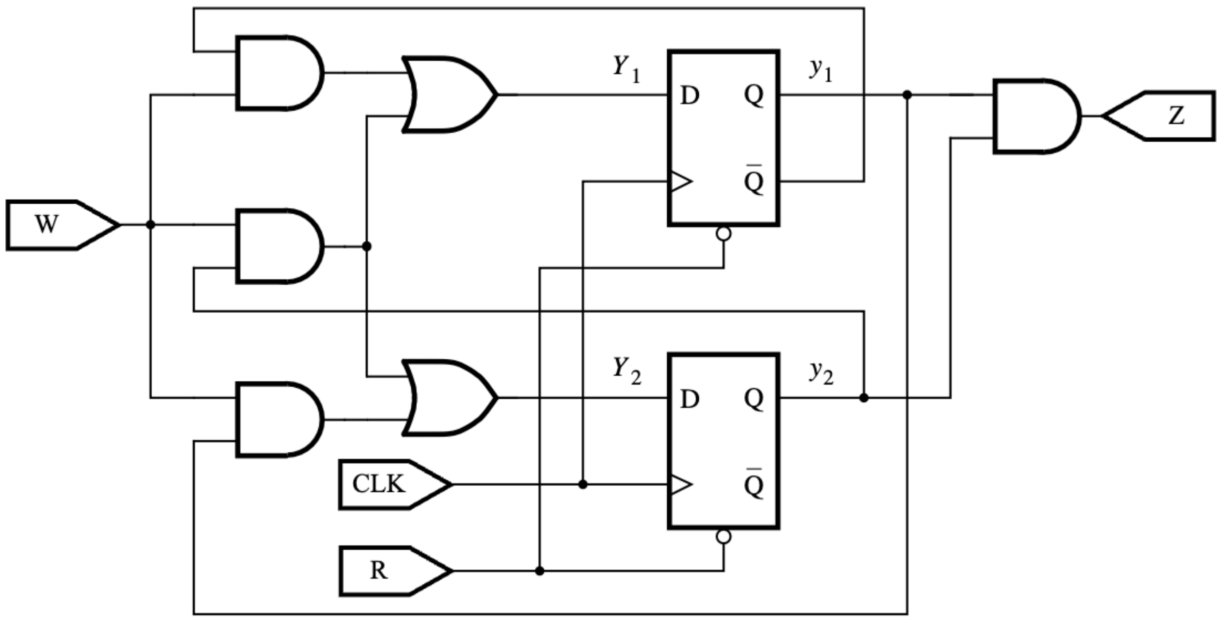


Figure 72: Sequential Circuit

[Solution 34] Analysing Finite State Machines (FSMs)

1. The boolean expressions are as follows:
   $Y_1 = \overline{y_1} \cdot W + y_2 \cdot W$
   $Y_2 = y_2 \cdot W + y_1 \cdot W$
   $Z = y_1 \cdot y_2$

2. The state/output table is as follows:

Present State	Next State		Output (Z)
	$W = 0$	$W = 1$	
y_2y_1	Y_2Y_1	Y_2Y_1	
00	00	01	0
01	00	10	0
10	00	11	0
11	00	11	1

Table 20: State/output table

3. As seen from the state/output table (Table 20), no matter which state the FSM is currently in, if the input W is 0 then the next state is 00. Starting from state 00, if the input (W) is 1, it transitions to 01. From state 01, if the input is 1, it transitions to 10. Again from state 10, if the input is 1, it transitions to 11. The output (Z) of the FSM is 1 only in the state 11, which can only be reached if W is 1 for three consecutive clock cycles as explained. Hence, the sequential circuit that implements this FSM is a sequence detector that detects the sequence 111.

[Exercise 35] Sequence Detector

a) Design a Moore FSM that has a 1-bit input W and a 1-bit output Z.
The FSM functions as a sequence detector that produces $Z = 1$ when the previous two values of W were 00 or 11; otherwise $Z = 0$.

Complete the following tasks:

1. How many states does the FSM have?
2. Draw the state diagram of the FSM.
3. Write the state/output table of the FSM.
4. Encode the states with the minimum number of bits possible. The i-th state should be represented by the binary encoding of i.
5. Derive the Boolean expressions for the next state and output functions.

b) An alternate encoding scheme popular for encoding states is the one-hot encoding scheme. The traditional approach (as done in Part 4) needs $\lceil \log_2 N \rceil$ bits to represent N states. One hot encoding uses N bits to represent N states. Each bit represents one state. Only one of the bits is set to 1, and the rest are set to 0. The index of the set bit represents the state. For example, consider an FSM that has 4 states (State 0, 1, 2 and 3). To represent State 3, the binary representation would be 11. However, the one-hot representation would be 1000. The bit with index 3 (indexing starting from zero) is set to 1 corresponding to State 3 and all other bits are 0. Similarly, to represent the state 0, the binary representation would be 00. But, the one-hot representation would be 0001.

Complete the following tasks:

1. Encode the states using a one hot encoding.
2. Derive the Boolean expressions for the next state and output functions.
3. How does the choice of encoding affect the complexity of the next state and output functions? Answer in terms of
   (a) How does the number of D Flip-Flops required to implement the circuit change?
   (b) How many NOT gates and 2-input AND/OR gates in the circuit are required?

[Solution 35] Sequence Detector

a)

1. One state (State A) represents the initial state when no inputs have been applied yet. Two states (State B and State D) are required to store the information what was the value of input W in the previous clock cycle. Two more states (State C and State E) are required to store the information if the previous two values of W were 00 or 11. So, in total, 5 states are required. The reasoning behind choosing these 5 states is explained next.

2. (a) To draw the state diagram, lets first see how the FSM operates. The FSM is initially in state A. If $W = 0$, it transitions to state B, and if $W = 1$, it transitions to state D.

   (b) Now, if the FSM is in state B, and $W = 0$, then it means $W = 0$ for two clock cycles, and the FSM transitions to state C. However, if the FSM is in state B, and $W = 1$, then it means $W = 01$ in two clock cycles. This sequence is not useful, and the FSM transitions to state D to represent the fact that W was equal to 1.

   (c) Similarly, if the FSM is in state D, and $W = 1$, then it means $W = 1$ for two clock cycles, and the FSM transitions to state E. If $W = 0$, then the sequence 10 is not useful and the FSM transitions to state B.

   (d) If the FSM is in state C, and $W = 0$, then it still means $W = 00$ in the last two clock cycles, and the FSM remains in state C. However if $W = 1$, then the sequence 01 is not useful and the FSM transitions to state D. Similarly, if the FSM is in state E, and $W = 1$, then it remains in state E, otherwise it transitions to state B.

   (e) The output of the FSM is 1 simply when the FSM is in state C or state E.

   This behavior is represented through the state diagram shown in Fig. 73.

3. The state/output table, shown in Table 21, can be calculated from the state diagram shown in Fig. 73.

4. State A is encoded as 000. State B is encoded as 001. State C is encoded as 010. State D is encoded as 011. State E is encoded as 100. The state/output table is shown in Table 22.

5. The next state and output Boolean expressions can then be derived in SOP form:

   - $Y_1 = \overline{W}\cdot\overline{y_3}\cdot\overline{y_2}\cdot\overline{y_1}+W\cdot\overline{y_3}\cdot\overline{y_2}\cdot\overline{y_1}+W\cdot\overline{y_3}\cdot\overline{y_2}\cdot y_1+W\cdot\overline{y_3}\cdot y_2\cdot\overline{y_1}+\overline{W}\cdot\overline{y_3}\cdot y_2\cdot y_1+\overline{W}\cdot y_3\cdot\overline{y_2}\cdot\overline{y_1}$
     $= \overline{y_3}\cdot\overline{y_2}\cdot\overline{y_1} + W\cdot\overline{y_3}\cdot\overline{y_2}\cdot y_1 + W\cdot\overline{y_3}\cdot y_2\cdot\overline{y_1} + \overline{W}\cdot\overline{y_3}\cdot y_2\cdot y_1 + \overline{W}\cdot y_3\cdot\overline{y_2}\cdot\overline{y_1}$
     $= \overline{y_3}\cdot\overline{y_2}\cdot\overline{y_1} + W\cdot(\overline{y_3}\cdot\overline{y_2}\cdot y_1 + \overline{y_3}\cdot y_2\cdot\overline{y_1}) + \overline{W}\cdot(\overline{y_3}\cdot y_2\cdot y_1 + y_3\cdot\overline{y_2}\cdot\overline{y_1})$

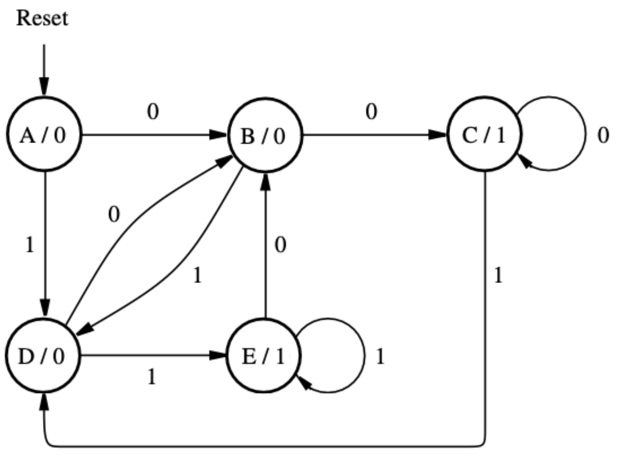


Figure 73: State Diagram for the FSM

Present State	Next State		Output (Z)
	$W=0$	$W=1$	
A	B	D	0
B	C	D	0
C	C	D	1
D	B	E	0
E	B	E	1

Table 21: State/output table

- $Y_2 = W\cdot\overline{y_3}\cdot\overline{y_2}\cdot\overline{y_1} + \overline{W}\cdot\overline{y_3}\cdot\overline{y_2}\cdot y_1 + W\cdot\overline{y_3}\cdot\overline{y_2}\cdot y_1 + \overline{W}\cdot\overline{y_3}\cdot y_2\cdot\overline{y_1} + W\cdot\overline{y_3}\cdot y_2\cdot\overline{y_1}$
  $= W\cdot\overline{y_3}\cdot\overline{y_2}\cdot\overline{y_1} + \overline{y_3}\cdot\overline{y_2}\cdot y_1 + \overline{y_3}\cdot y_2\cdot\overline{y_1}$
- $Y_3 = W\cdot\overline{y_3}\cdot y_2\cdot y_1 + W\cdot y_3\cdot\overline{y_2}\cdot\overline{y_1}$
  $= W\cdot(\overline{y_3}\cdot y_2\cdot y_1 + y_3\cdot\overline{y_2}\cdot\overline{y_1})$
- $Z = \overline{y_3}\cdot y_2\cdot\overline{y_1} + y_3\cdot\overline{y_2}\cdot\overline{y_1}$

b)

1. In one-hot encoding, State A is encoded as 00001. State B is encoded as 00010. State C is encoded as 00100. State D is encoded as 01000. State E is encoded as 10000. The corresponding state/output table is now shown in Table. 23:

2. The next state and output Boolean expressions can then be derived from the state/output table using SOP form:

   - $Y_1 = 0$

Present State	Next State		Output (Z)
	$W = 0$	$W = 1$	
$y_3y_2y_1$	$Y_3Y_2Y_1$	$Y_3Y_2Y_1$	
000	001	011	0
001	010	011	0
010	010	011	1
011	001	100	0
100	001	100	1

Table 22: State/output table using binary encoding

Present State	Next State		Output (Z)
	$W = 0$	$W = 1$	
$y_5y_4y_3y_2y_1$	$Y_5Y_4Y_3Y_2Y_1$	$Y_5Y_4Y_3Y_2Y_1$	
00001	00010	01000	0
00010	00100	01000	0
00100	00100	01000	1
01000	00010	10000	0
10000	00010	10000	1

Table 23: State/output table using one-hot encoding

- $Y_2 = \overline{W} \cdot (y_1 + y_4 + y_5)$
- $Y_3 = \overline{W} \cdot (y_2 + y_3)$
- $Y_4 = W \cdot (y_1 + y_2 + y_3)$
- $Y_5 = W \cdot (y_4 + y_5)$
- $Z = y_3 + y_5$

Note that the fact that only one bit of the binary vector representing the state can be one greatly simplifies the circuit design. As an example, $\overline{y_5} \cdot \overline{y_4} \cdot \overline{y_3} \cdot \overline{y_2} \cdot y_1$ is equivalent to simply y_1 because $y_1 = 1$ automatically implies $y_5, y_4, y_3, y_2 = 0000$.

3. Comparison of one-hot encoding with binary encoding:

   (a) One-hot encoding needs N bits for N states whereas, binary encoding only needs $\lceil \log_2 N \rceil$ bits. Each bit corresponds to a D Flip-Flop, so circuits using one-hot encoding use more D Flip Flops.

   (b) To count the gates, we count all the $+$ and $\cdot$ in the logic functions that account for OR and AND gate, respectively (note that we are not sharing the gates for simplification). In addition, we need a NOT gate to invert the input (W). Note that we do not need a NOT gate to invert state values (y_1, y_2, y_3)

because we can get them from the $\overline{Q}$ pin of the flip-flop. As a result, the combinational logic for next state transitions and output value needs 37 gates when using binary encoding, and only 12 gates when using one-hot encoding. Hence, the circuit for one-hot encoding is simpler.

[Exercise 36] Implementing FSMs in Verilog

Consider an FSM that has a 1-bit input W and a 1-bit output Z. The current state of the FSM is denoted by y_2y_1, and the next state is denoted by Y_2Y_1. The FSM has four possible states, which are encoded as: 00, 01, 10, and 11. The state/output table of the FSM is given in Table 1.

Present State	Next State		Output (Z)
	W = 0	W = 1	
y_2y_1	Y_2Y_1	Y_2Y_1	
00	10	11	0
01	01	00	0
10	11	00	0
11	10	01	1

Table 24: State/output table of the FSM

1. Derive the Boolean expressions for the next state and output functions.
2. Is this FSM a Moore or Mealy machine? Justify your answer.
3. Draw the circuit diagram of a sequential circuit that implements this FSM. (Note: The circuit should include a standard clk signal and a synchronous `reset` signal that resets the state to 00 if set high.)
4. Design a Verilog module that implements this FSM. The module should have the following interface and name:

```
module fsm(
    input clk,
    input reset,
    input W,
    output reg Z
);
```

5. Write a testbench to verify the functionality of the Verilog module. The `reset` signal should first reset the FSM state. Next, the testbench can apply different input squences and check if the output matches the expected output in Table 1. Ensure that the testbench covers all possible state transitions.

[Solution 36] Implementing FSMs in Verilog

1. The next state and output Boolean expressions can then be derived in SOP form:
   - $Y_1 = W \cdot \overline{y_2} \cdot \overline{y_1} + \overline{W} \cdot \overline{y_2} \cdot y_1 + \overline{W} \cdot y_2 \cdot \overline{y_1} + W \cdot y_2 \cdot y_1$
   - $Y_2 = \overline{y_2} \cdot \overline{y_1} + \overline{W} \cdot y_2$
   - $Z = y_2 \cdot y_1$
2. The FSM is a Moore machine because the output Z only depends on the current state y_2y_1, as $Z = y_2 \cdot y_1$ and does not depend on the input (W) at all.
3. The circuit diagram is shown in Fig. 74

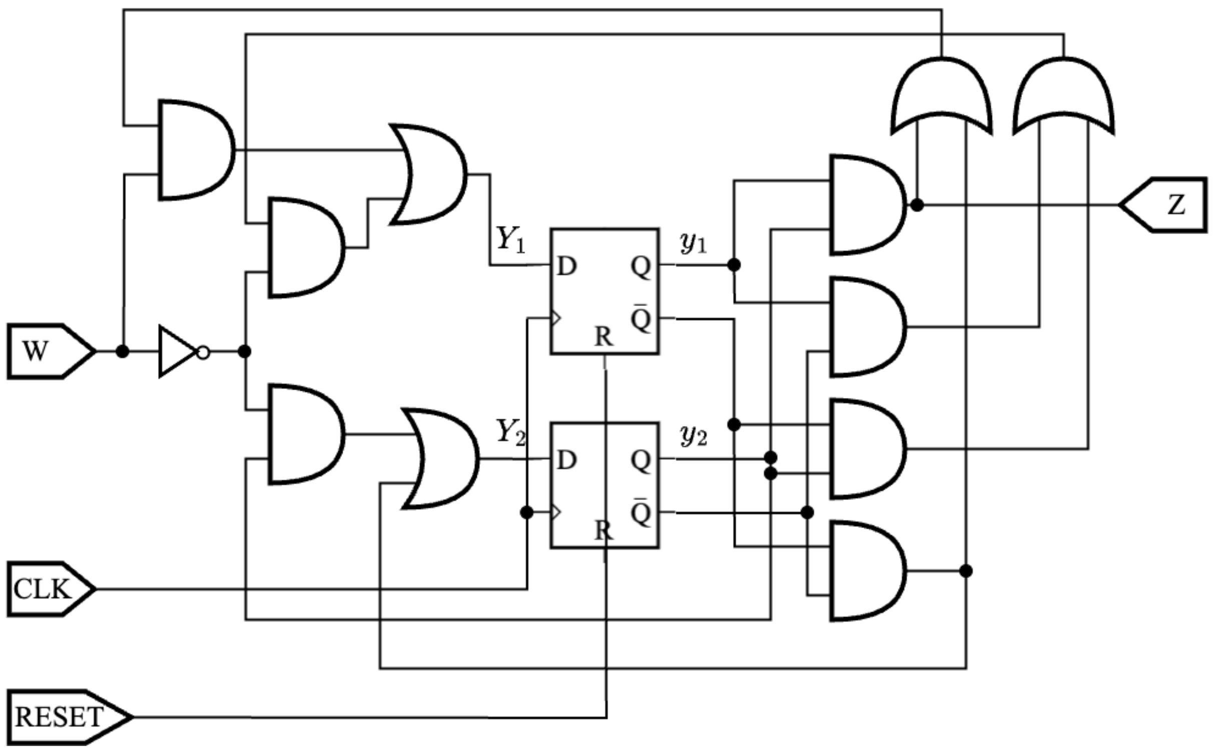


Figure 74: Sequential circuit implenenting FSM

4. The Verilog module can be implemented in three ways.

   Solution 1: Using the derived Boolean expressions. We do not advise this solution for the course, where three always blocks are not used. We advise you to implement it using three always blocks, two purely combinational and one sequential. The implementation with three always blocks is easy to read, manage, and implement.

```
module fsm(
    input clk,
    input reset,
    input W,
    output reg Z
);
    reg [1:0] S;

    always@(posedge clk) begin
        if(reset == 1'b1) begin
            S <= 2'b00;
        end else begin
            S[0] <= (W & ((!S[0] & !S[1]) |
                        (S[0] & S[1]))) |
                        (!W & ((!S[0] & S[1]) |
                        (S[0] & !S[1])));
            S[1] <= (!S[0] & !S[1]) |
                        (!W & S[1]);
        end
    end

    always@(*) begin
        Z = S[0] & S[1];
    end
endmodule
```

Solution 2: Using the derived Boolean expressions with the recommended guidelines, that is, to use three always blocks as it is easy to read, manage, and implement.

```
module fsm(
    input clk,
    input reset,
    input W,
    output reg Z
);
    reg [1:0] D, S;

    // Next-state logic
    always@(*) begin
        D[0] = (W & ((!S[0] & !S[1]) |
                (S[0] & S[1]))) |
                (!W & ((!S[0] & S[1]) |
                (S[0] & !S[1])));
        D[1] = (!S[0] & !S[1]) |
                (!W & S[1]);
    end

    // State memory
    always@(posedge clk) begin
        if(reset == 1'b1) begin
            S <= 2'b00;
        end else begin
            S <= D;
        end
    end

    // Output logic
    always@(*) begin
        Z = S[0] & S[1];
    end
endmodule
```

Solution 3: Using the state/output table specification with the recommended guidelines, that is, to use three always blocks as it is easy to read, manage, and implement.

```
module fsm(
    input clk,
    input reset,
    input W,
    output reg Z
);
    reg [1:0] S_next, S;

    //Next-state logic
    always@(*) begin
        S_next = 2'b00;
        case(S)
            2'b00: if (W == 1'b0) S_next = 2'b10;
                   else           S_next = 2'b11;
            2'b01: if (W == 1'b0) S_next = 2'b01;
                   else           S_next = 2'b00;
            2'b10: if (W == 1'b0) S_next = 2'b11;
                   else           S_next = 2'b00;
            2'b11: if (W == 1'b0) S_next = 2'b10;
                   else           S_next = 2'b01;
            default:              S_next = 2'b00;
        endcase
    end

    // State memory
    always@(posedge clk) begin
        if(reset == 1'b1) begin
            S <= 2'b00;
        end else begin
            S <= S_next;
        end
    end

    // Output logic
    always@(*) begin
        Z = S[0] & S[1];
    end

endmodule
```

5. The testbench can be written as follows:

```
module tb_fsm;
reg clk, reset, W;
wire Z;

fsm DUT(.clk(clk), .reset(reset),
        .W(W), .Z(Z));

initial begin
    $dumpfile("tb_fsm.vcd");
    $dumpvars(0, tb_fsm);
    reset = 1;
    clk = 0;
    W = 0;
    #10 reset = 0; // State -> 00
    $display("Internal State = %b, Output = %b", DUT.S, Z);
    #10; // 00 -> 10
    $display("Internal State = %b, Output = %b", DUT.S, Z);
    #10; // 10 -> 11
    $display("Internal State = %b, Output = %b", DUT.S, Z);
    #10 W = 1;  // 11 -> 10
    $display("Internal State = %b, Output = %b", DUT.S, Z);
    #10; // 10 -> 00
    $display("Internal State = %b, Output = %b", DUT.S, Z);
    #10; // 00 -> 11
    $display("Internal State = %b, Output = %b", DUT.S, Z);
    #10 W = 0; // 11 -> 01
    $display("Internal State = %b, Output = %b", DUT.S, Z);
    #10 W = 1; // 01 -> 01
    $display("Internal State = %b, Output = %b", DUT.S, Z);
    #10; // 01 -> 00
    $display("Internal State = %b, Output = %b", DUT.S, Z);
    #10 $finish;
end

always begin
    #5 clk = ~clk;
end

endmodule
```

This testbench cycles through all 8 possible state transitions, and prints the internal state and output at each step. (Note: If the name of the state variable inside your FSM module is different, you should replace `DUT.S` with `DUT.XYZ` where XYZ is the name of your state variable.)

On successful execution, the output should be:

```
VCD info: dumpfile tb_fsm.vcd opened for output.
Internal State = 00, Output = 0
Internal State = 10, Output = 0
Internal State = 11, Output = 1
Internal State = 10, Output = 0
Internal State = 00, Output = 0
Internal State = 11, Output = 1
Internal State = 01, Output = 0
Internal State = 01, Output = 0
Internal State = 00, Output = 0
tb_fsm.v:27: $finish called at 100 (1s)
```

[Exercise 37] FSM for a Candy Vending Machine

Consider a vending machine that dispenses candy bars. Each candy bar costs 8 CHF (it is a very fancy candy). The candy bar accepts coins of 1 CHF, 2 CHF, and 5 CHF. A user can insert as many coins into the vending machine as they want. The vending machine will dispense a candy bar when the user has inserted a total of atleast 8 CHF. Any extra money will count towards the next candy bar.

For example, a user can enter a 5 CHF, 2 CHF and a 1 CHF coin to get a candy bar. A user can also enter a 5 CHF and 5 CHF coin. In that case, the user will get one candy bar. The vending machine will still have 2 CHF left inside, and the user can just insert a new 5 CHF coin and a 1 CHF coin to get another candy bar.

Design a finite state machine that models the behavior of this vending machine.

Complete the following tasks:

1. What should be the input(s), output(s), and internal state(s) of the FSM?
2. Encode the input(s), output(s) and internal state(s) of the FSM with a suitable representation of your choice.
3. Draw the state diagram of the finite state machine.
4. Write the state/output table of the finite state machine.

[Solution 37] FSM for a Candy Vending Machine

1. The input to the finite state machine is the value of the coin inserted by the user. The internal state of the finite state machine is the total amount of money already present inside the vending machine. The output of the finite state machine is 1 if the vending machine should dispense a candy bar, and 0 otherwise.
2. There can be 4 possible inputs: no coin or 1 CHF or 2 CHF or 5 CHF. These 4 inputs can be encoded as 00, 01, 10, and 11 respectively. The vending machine can have 0 CHF or 1 CHF or 2 CHF or 3 CHF or 4 CHF or 5 CHF or 6 CHF or 7 CHF. These 8 states can be encoded as 000, 001, 010, 011, 100, 101, 110, and 111 respectively. (Note: The vending machine can never have more than 7 CHF inside it) The output can either be 0 or 1 as explained earlier.
3. The state diagram of the finite state machine is shown in Figure 75.

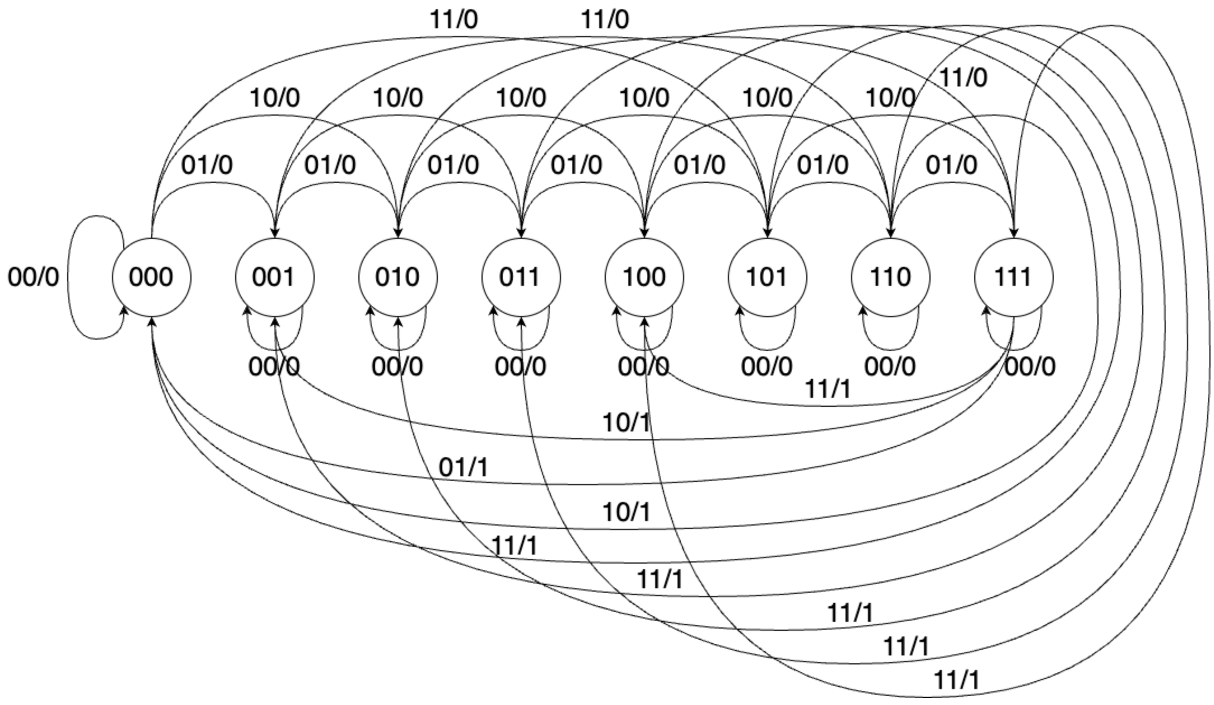


Figure 75: State diagram of the finite state machine for a candy vending machine

4. The logic behind the state diagram and the state/output table is shown next. If the vending machine has X CHF inside it, and the user inserts Y CHF, then the new state of the vending machine will be $(X + Y) \pmod 8$ CHF. As an example, if the vending machine has 5 CHF inside it, and the user inserts 2 CHF, then the new state of the vending machine will be $(5 + 2) \pmod 8 = 7$ CHF.

However, if the vending machine has 5 CHF inside it, and the user inserts 5 CHF, then the new state of the vending machine will be $(5+5) \pmod 8 = 2$ CHF. This is because the total money the user has inserted is 10 CHF, which is enough for 1 candy bar (which costs 8 CHF), hence 2 CHF is leftover inside the machine. If the user doesn't insert any coin (i.e., $Y = 0$), then the state of the vending machine remains unchanged. The output of the FSM is 1 if $X + Y \geq 8$, to indicate that a candy bar is available and 0 otherwise.

The state/output table of the finite state machine is shown in Table 5.

Present	Next State (CHF)				Output			
State (CHF)	In = 0	In = 1	In = 2	In = 5	In = 0	In = 1	In = 2	In = 5
0	0	1	2	5	0	0	0	0
1	1	2	3	6	0	0	0	0
2	2	3	4	7	0	0	0	0
3	3	4	5	0	0	0	0	1
4	4	5	6	1	0	0	0	1
5	5	6	7	2	0	0	0	1
6	6	7	0	3	0	0	1	1
7	7	0	1	4	0	1	1	1

Table 25: State/output table of the finite state machine for a candy vending machine

By using the encoded values of the inputs, outputs, and states, the state/output table can be used to create the state transition diagram shown in Fig. 75.

Note: Although we used a Mealy FSM, we recommend you to implement it with a Moore FSM to ensure that the outputs are not a function of the inputs. This independence of outputs from inputs increases stability of the circuit, as outputs are not impacted by input glitches. However, a Moore FSM may require more number of states than a Mealy FSM, but this is generally acceptable given the improved reliability against input glitches.

[Exercise 38] Timing Analysis of Synchronous Circuits

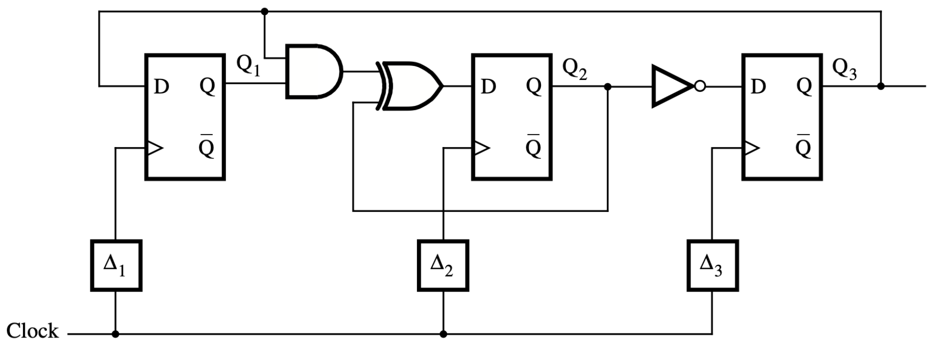


Figure 76: Circuit diagram.

Consider the circuit in Figure 76. It consists of three D flip-flops (DFFs) and combinational logic. The flip-flop inputs are D_1, D_2, and D_3, and outputs are Q_1, Q_2, and Q_3, respectively. The clock delays to these flip-flops are shown as Δ_1, Δ_2, and Δ_3, respectively.

The timing properties of the DFFs are as follows:

- Setup time (t_{setup}) is 0.6 ns;
- Hold time (t_{hold}) is 0.4 ns;
- Clock-to-Q delay (t_{cQ}) between 0.8 ns (min) and 1 ns (max).

The propagation delay of a combinational logic gate is given by $t_{gatedelay} = 1 + 0.1k$ ns, where k is the number of inputs to the gate.

Find the maximum operating clock frequency (f_{max}) and determine whether there are any hold time violations for the following three cases:

a) $\Delta_1 = \Delta_2 = \Delta_3 = 0$ ns

b) $\Delta_1 = \Delta_3 = 0$ ns, $\Delta_2 = 0.7$ ns

c) $\Delta_1 = 1$ ns, $\Delta_2 = 0$ ns, and $\Delta_3 = 0.5$ ns

[Solution 38] Timing Analysis of Synchronous Circuits

a) $\Delta_1 = \Delta_2 = \Delta_3 = 0$ ns

Logic gates have the following delays: t_{NOT} = 1.1 ns, t_{AND} = t_{XOR} = 1.2 ns.

The maximum clock frequency is determined by the longest path between a flip-flop output and a flip-flop input. To find it, we need to enumerate (identify) all paths and find their max delays (using the maximum value of t_{cQ}). The sum of the path delay and the setup time of the FF determines the corresponding minimum clock period. Below, we list all paths and compute their delay, adding to each the setup time of the FF.

- Q_1 to D_2: $t_{cQ,\max} + t_{XOR} + t_{AND} + t_{setup} = 1 + 1.2 + 1.2 + 0.6 = 4$ ns
- Q_2 to D_2: $t_{cQ,\max} + t_{XOR} + t_{setup} = 1 + 1.2 + 0.6 = 2.8$ ns
- Q_2 to D_3: $t_{cQ,\max} + t_{NOT} + t_{setup} = 1 + 1.1 + 0.6 = 2.7$ ns
- Q_3 to D_1: $t_{cQ,\max} + t_{setup} = 1 + 0.6 = 1.6$ ns
- Q_3 to D_2: $t_{cQ,\max} + t_{XOR} + t_{AND} + t_{setup} = 1 + 1.2 + 1.2 + 0.6 = 4$ ns

The min clock period this circuit can support is determined by the longest-delay path of all. In this circuit, two longest-delay paths exist: from Q_1 to D_2 and from Q_3 to D_2. Therefore, the min clock period is 4 ns. Finally, the corresponding clock frequency is $f_{max} = 1/4$ ns $= 250$ MHz. This is the highest frequency on which the circuit can correctly operate. At a higher clock frequency, setup-time constraints would no longer be satisfied on at least one of the paths listed above.

To check whether any hold-time violations exist, we need to again enumerate all paths. This time, we need to find their min delays (using the minimum value of t_{cQ}).

- Q_1 to D_2: $t_{cQ,\min} + t_{XOR} + t_{AND} = 0.8 + 1.2 + 1.2 = 3.2$ ns
- Q_2 to D_2: $t_{cQ,\min} + t_{XOR} = 0.8 + 1.2 = 2$ ns
- Q_2 to D_3: $t_{cQ,\min} + t_{NOT} = 0.8 + 1.1 = 1.9$ ns
- Q_3 to D_1: $t_{cQ,\min} = 0.8$ ns
- Q_3 to D_2: $t_{cQ,\min} + t_{XOR} + t_{AND} = 0.8 + 1.2 + 1.2 = 3.2$ ns

The delay of the shortest path of all is 0.8 ns. As this delay is longer than the hold time (t_{hold} =0.4 ns), there are no hold-time violations in the circuit.

b) $\Delta_1 = \Delta_3 = 0$ ns, $\Delta_2 = 0.7$ ns

There is clock skew for the paths between flip-flops Q1 and Q2, as well as Q3 and Q2. Adjusting the max path delays calculated above to account for clock skew, we have:

- Q_1 to D_2: $4 - t_{skew} = 4 - (\Delta_2 - \Delta_1) = 4 - 0.7 = 3.3$ ns
- Q_2 to D_2: 2.8 ns
- Q_2 to D_3: $2.7 - t_{skew} = 2.7 - (\Delta_3 - \Delta_2) = 2.7 - (0 - 0.7) = 3.4$ ns
- Q_3 to D_1: 1.6 ns
- Q_3 to D_2: $4 - t_{skew} = 4 - (\Delta_2 - \Delta_3) = 4 - 0.7 = 3.3$ ns

The corresponding maximum clock frequency is $f_{max} = 1/3.4$ ns $= 294$ MHz.

Similarly, to find if there are hold-time violations, we need to adjust the min path delays to account for the clock skew.

- Q_1 to D_2: $3.2 - t_{skew} = 3.2 - (\Delta_2 - \Delta_1) = 3.2 - 0.7 = 2.5$ ns
- Q_2 to D_2: 2 ns
- Q_2 to D_3: $1.9 - t_{skew} = 1.9 - (\Delta_3 - \Delta_2) = 1.9 - (0 - 0.7) = 2.6$ ns
- Q_3 to D_1: 0.8 ns
- Q_3 to D_2: $3.2 - t_{skew} = 3.2 - (\Delta_2 - \Delta_3) = 3.2 - 0.7 = 2.5$ ns

The delay of the shortest path of all is 0.8 ns, which is greater than the hold time. Therefore, there are no hold-time violations in the circuit.

c) $\Delta_1 = 1$ ns, $\Delta_2 = 0$ ns, and $\Delta_3 = 0.5$ ns

There is clock skew on all paths between flip-flops. Adjusting the max path delays calculated above to account for clock skew, we get:

- Q_1 to D_2: $4 - t_{skew} = 4 - (\Delta_2 - \Delta_1) = 4 - (0 - 1) = 5$ ns

- Q_2 to D_2: 2.8 ns
- Q_2 to D_3: $2.7 - t_{skew} = 2.7 - (\Delta_3 - \Delta_2) = 2.7 - (0.5 - 0) = 2.2$ ns
- Q_3 to D_1: $1.6 - t_{skew} = 1.6 - (\Delta_1 - \Delta_3) = 1.6 - (1 - 0.5) = 1.1$ ns
- Q_3 to D_2: $4 - t_{skew} = 4 - (\Delta_2 - \Delta_3) = 4 - (0 - 0.5) = 4.5$ ns

The corresponding maximum clock frequency is $f_{max} = 1/5$ ns $= 200$ MHz.

Similarly, to find if there are hold-time violations, we need to adjust the min path delays to account for the clock skew.

- Q_1 to D_2: $3.2 - t_{skew} = 3.2 - (\Delta_2 - \Delta_1) = 3.2 - (0 - 1) = 4.2$ ns
- Q_2 to D_2: 2 ns
- Q_2 to D_3: $1.9 - t_{skew} = 1.9 - (\Delta_3 - \Delta_2) = 1.9 - (0.5 - 0) = 1.4$ ns
- Q_3 to D_1: $0.8 - t_{skew} = 0.8 - (\Delta_1 - \Delta_3) = 0.8 - (1 - 0.5) = 0.3$ ns
- Q_3 to D_2: $3.2 - t_{skew} = 3.2 - (\Delta_2 - \Delta_3) = 3.2 - (0 - 0.5) = 3.7$ ns

The shortest path of all is the path between Q_3 and D_1. Its delay is 0.3 ns, which is lower than the hold time (0.4 ns). Therefore, there is a hold-time violation in the circuit and the circuit may not function reliably regardless of the clock frequency.

[Exercise 39] Moore Finite State Machine

A state diagram of a Moore finite state machine (FSM) is shown in Figure 77. The FSM contains a counter, counting the elapsed clock cycles and acting as a simple **timer**.

While **In** is 0, the FSM is in the IDLE state. When **In** becomes 1, the timer is initiated (reset) and the FSM transitions to state A. The FSM stays in state A for 10 cycles. After that, the FSM transitions to state B and stays in it for one cycle before returning to the IDLE state. While the FSM is in states IDLE or B, the timer is not advancing. Implement this FSM.

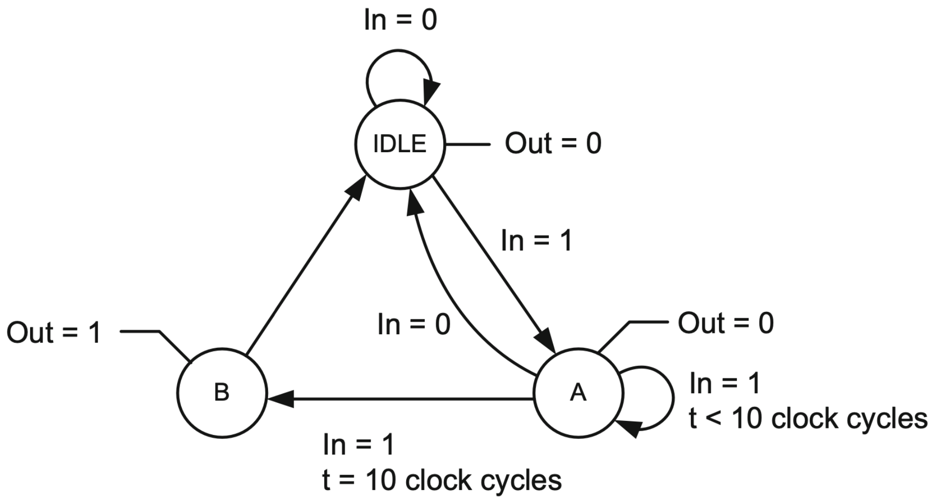


Figure 77: State diagram of a Moore FSM with a timer.

a) Fill the state/output table.

b) Implement the FSM in Verilog. Your design should include an **active low** synchronous reset. The reset clears the counter (i.e., resets all bits to zero) and initializes the FSM to the IDLE state.

[Solution 39] Moore Finite State Machine

a)

In this design, we have two variables that contribute to the current state: the state itself and the counter. The counter is used to count the number of cycles in state A until counter reaches 9 and we do not care about the counter value when the machine is in other states. As the question requires, we keep the counter value unchanged when the machine is in other states.

The state transition logic is given in Table 26. The X in the table represents a don't care value.

Table 26: State transition logic for the Moore machine.

Current State	Current Counter	In	Next State	Next Counter	Out
IDLE	X	0	IDLE	X	0
IDLE	X	1	A	0	0
A	X	0	IDLE	X	0
A	0	1	A	1	0
A	1	1	A	2	0
A	2	1	A	3	0
A	3	1	A	4	0
A	4	1	A	5	0
A	5	1	A	6	0
A	6	1	A	7	0
A	7	1	A	8	0
A	8	1	A	9	0
A	9	1	B	9	0
B	9	X	IDLE	9	1

b)

```
module outEveryTen (
    input clk,
    input in,
    input resetn, // "n" for negated, active low
    output reg out
);

    parameter IDLE = 2'b00;
    parameter A = 2'b01;
    parameter B = 2'b10;
    reg [1:0] current_state;
    reg [1:0] next_state;
    reg [3:0] count;

    // State transition logic
    always @(*) begin
        next_state = current_state;
        case (current_state)
            IDLE:
                if (in) begin
                    next_state = A;
                end else begin
                    next_state = IDLE;
                end
            A:
                if (in && count == 9) begin
                    next_state = B;
                end else if (in) begin
                    next_state = A;
                end else begin
                    next_state = IDLE;
                end
            B:
                next_state = IDLE;
            default:
                next_state = IDLE;
        endcase
    end

    // output assignment, Out=1 if current_state is B
    always @(*) begin
        out = 0;
        if (current_state == B) begin
            out = 1;
        end else begin
            out = 0;
        end
    end

    // Next state assignment at rising clk edge
    always @(posedge clk) begin
        if (resetn == 1'b0) begin
            current_state <= IDLE;
        end else begin
            current_state <= next_state;
        end
    end

    // increment count in state A until it reaches 9
    // and when in is high;
    // reset count when in is high in IDLE state;
    // reset count when reset is active
    always @(posedge clk) begin
        if (resetn == 1'b0) begin
            count <= 0;
        end else if (current_state == A && in && count < 9) begin
            count <= count + 1;
        end else if (current_state == IDLE && in) begin
            count <= 0;
        end
    end


endmodule
```

[Exercise 40] Memory

Consider the design shown in Fig. 78. It has four inputs: clock, Init, DataInA, and an asynchronous Reset (not shown), and no outputs. It contains two memories, two counters, a comparator, an adder, a subtractor, a multiplexer, and a controller. This digital design reads bytes from one memory, compares them, adds or subtracts them, and writes the result to another memory.

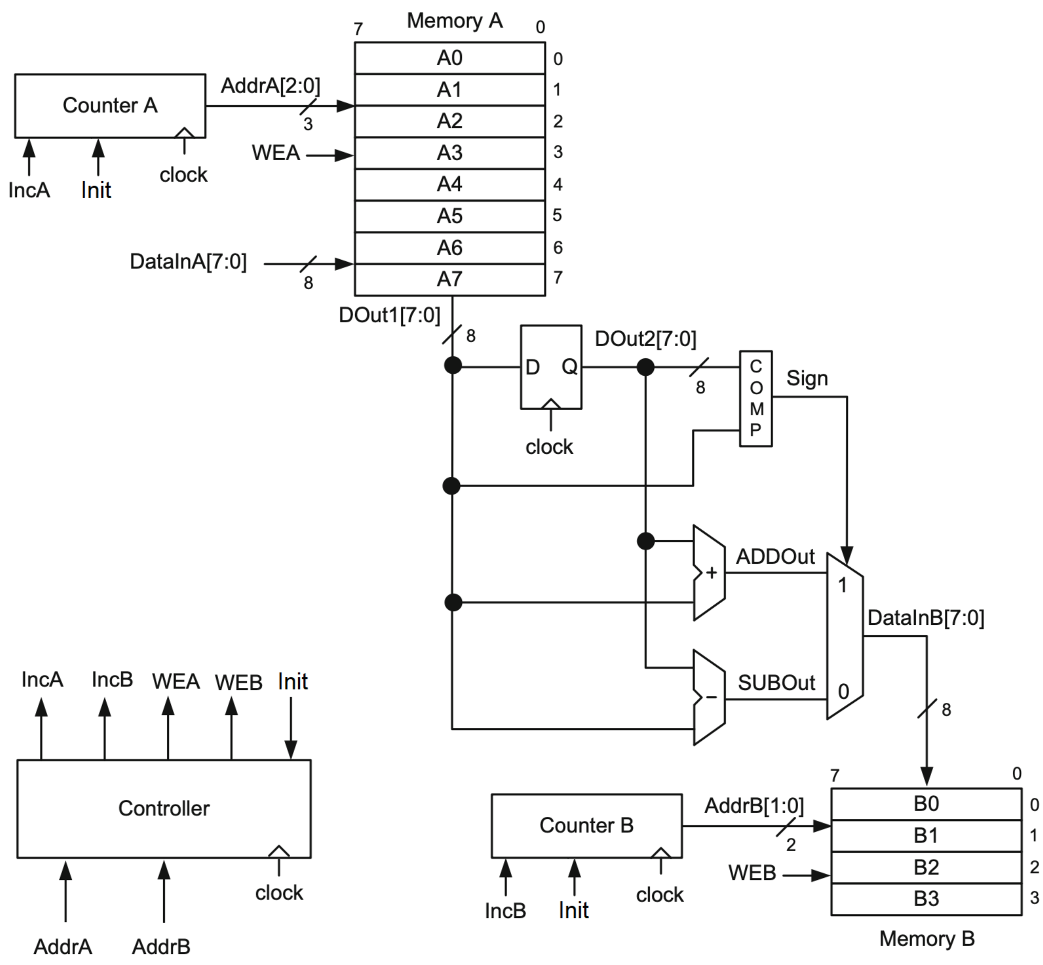


Figure 78: Circuit diagram

PART I: DESCRIPTION OF THE CIRCUIT DIAGRAM

Memory A is an 8×8 memory. In other words, it has eight rows, each containing one byte of data. **Memory B** is similar: it has four rows, each containing one byte of data. Both memories have a data input port (**DataInA** and **DataInB**, respectively), a write enable input (**WEA** and **WEB**, respectively), and a data output port (**Dout1** for Memory A). The clock input and the data output of Memory B are not shown. Memory write and read operations are synchronous with the rising edge of the **clock**.

Up counter A generates the address for Memory A. Similarly, **up counter B** generates the address for Memory B. The counters have an enable input (**IncA** and **IncB**, respectively). When Init is active, the counters reset to zero.

The **Controller** is the design's finite state machine, responsible for generating signals IncA, IncB, WEA, and WEB.

Besides the above components, the design contains a comparator **COMP**, an adder (**+**), a subtractor (**-**), and a multiplexer. The comparator takes two 8-bit inputs, compares them, and produces a one-bit output. The adder (subtractor) adds (subtracts) two 8-bit inputs and generates an 8-bit output. The multiplexer receives the 8-bit outputs from the adder and the subtractor and sends one of them to the data input of Memory B.

The **comparator** receives two bytes from Memory A: **Dout2** and **Dout1**. If Dout2 is greater than Dout1, the comparator outputs a logic 0; otherwise, it outputs a logic 1.

The **adder** and **subtractor** take Dout2 and Dout1 as inputs, perform addition and subtraction (Dout2 - Dout1), and output the result.

The **multiplexer** sends either the result of the addition or the result of the subtraction to the data input port of Memory B (DataInB). The output of the comparator acts as the select input of the multiplexer.

PART II: DESIGN FUNCTIONALITY

This circuit operates in three states: READA, COMP, and HALT.

The initial state is the READA state. As long as **Init** input is active, the circuit remains in this state. Once Init becomes inactive, counter A counts from 0 to 7, and the data present at the DataInA input gets written into Memory A. In other words, memory A gets filled and initialized. The initialization lasts eight clock cycles.

The circuit enters the second state, COMP, in the following clock cycle. It reads the data bytes from memory A sequentially (one after another) and processes them in pairs. The processing is done on the pairs of bytes (Dout2 and Dout1) at memory addresses

0 and 1, 2 and 3, 4 and 5, and 6 and 7. DOut2 and DOut1 are compared, and if DOut2 is greater than DOut1, then DOut1 is subtracted from DOut2. Otherwise, DOut1 is summed with DOut2. The computation result is written in memory B once every two clock cycles. When the result is written in Memory B (i.e., once every two clock cycles), the input address of Memory B is incremented. The up counter B counts from 0 to 3. COMP state terminates once all bytes in Memory A are read, and all bytes in Memory B are updated. If Init is active, the circuit returns to the start of READA state.

In the following clock cycle, the circuit enters the last (third) phase, HALT, which halts the computation. The counters stop counting, and reading from and writing to memories stop as well. If Init is active, the circuit returns to the start of READA state.

Answer the questions below.

a) Fill the timing diagrams in Figure 79. You can assume that Init and Reset are inactive. The diagram is split into two parts for space reasons. It covers 20 clock cycles. Labels A0, A1, ..., and A7 are the placeholders for a sequence of values appearing at the DataInA input (in cycles 0, 1, ..., 7). They should not be interpreted as hexadecimal values.

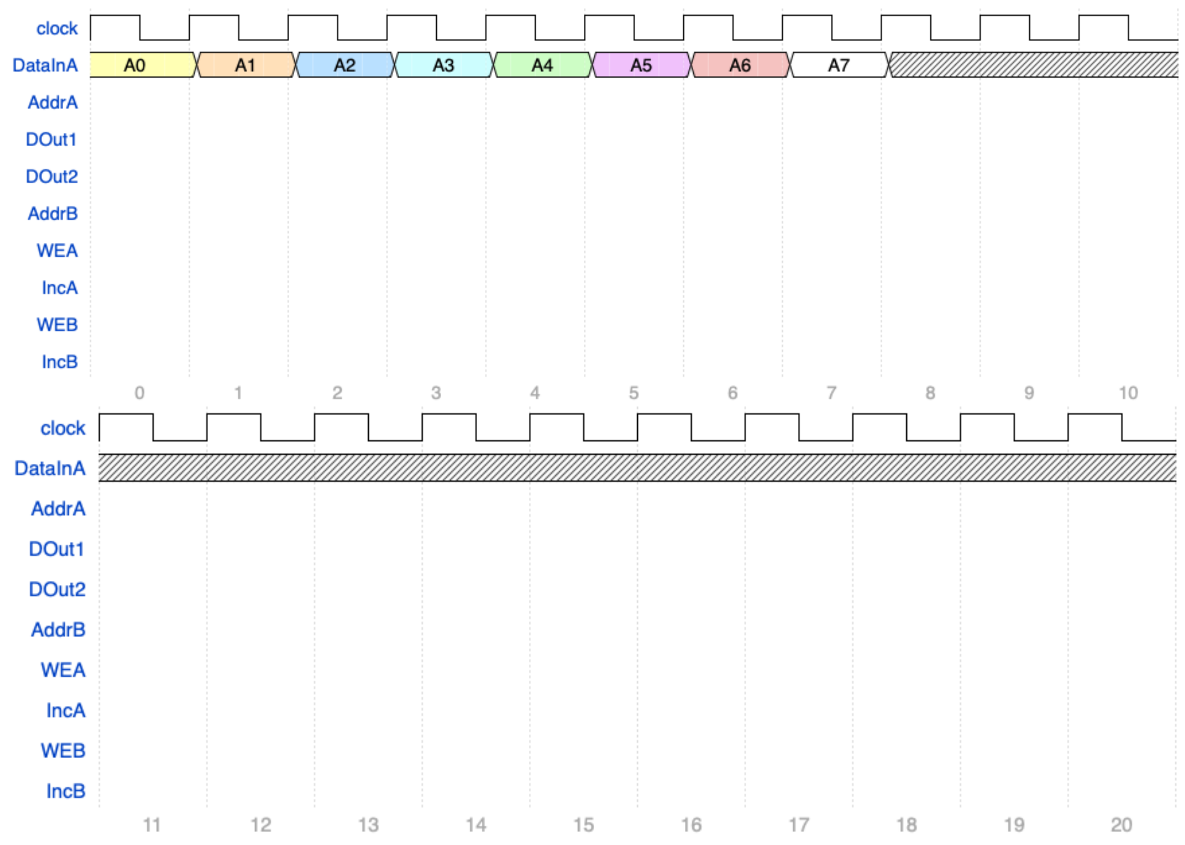


Figure 79: Timing diagram for the memory processing design.

b) Draw the state diagram of this memory processing design.

c) Draw the state transition table for the Controller module.

d) Write the Verilog description of the design in Fig. 78. To model the memory, write a single *parameterized* module, containing an internal memory array variable called `mem`. The flip-flops in the counters and the controller state register have an asynchronous power-on reset, to which Reset signal is connected (not shown in Fig. 78).

Hint: In Verilog, addition and subtraction can be easily modeled with + and −.

The testbench is available for download on Moodle. It requires the module to have the interface below. You are free to add more test cases in the testbenches to test your design comprehensively.

```
module memProcessing (
    input clock,
    input Reset,
    input Init,
    input [7:0] DataInA,
);
```

It also expects Memory B to be named `MemoryB`.

[Solution 40] Memory

a) The timing diagram can be seen in Figure 80.

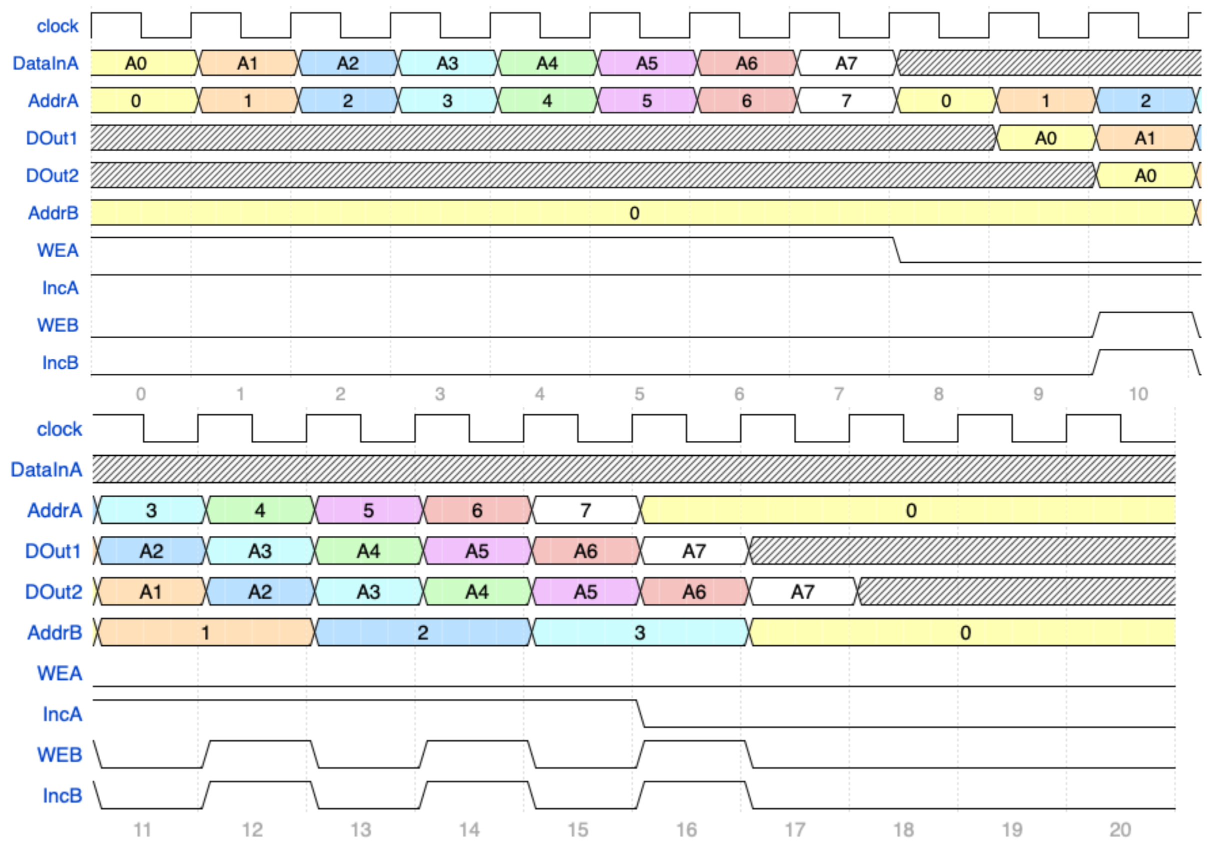


Figure 80: Solution of the timing diagram for the memory processing design.

b) The state diagram is given in Figure 81.

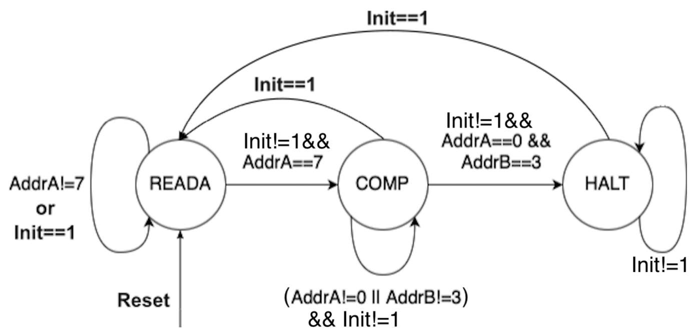


Figure 81: State diagram for the memory processing design.

c) The state transition table of the Controller is given in Table 27. CntA and CntB stand for the outputs of the counters, i.e., AddrA and AddrB in Fig. 78.

Table 27: State transition table of the Controller.

Current	Input	Current		Next			Outputs			
State	Init	CntA	CntB	State	CntA	CntB	IncA	IncB	WEA	WEB
READA	0	0	0	READA	1	0	1	0	1	0
READA	0	1	0	READA	2	0	1	0	1	0
READA	0	2	0	READA	3	0	1	0	1	0
READA	0	3	0	READA	4	0	1	0	1	0
READA	0	4	0	READA	5	0	1	0	1	0
READA	0	5	0	READA	6	0	1	0	1	0
READA	0	6	0	READA	7	0	1	0	1	0
READA	0	7	0	COMP	0	0	1	0	1	0
COMP	0	0	0	COMP	1	0	1	0	0	0
COMP	0	1	0	COMP	2	0	1	0	0	0
COMP	0	2	0	COMP	3	1	1	1	0	1
COMP	0	3	1	COMP	4	1	1	0	0	0
COMP	0	4	1	COMP	5	2	1	1	0	1
COMP	0	5	2	COMP	6	2	1	0	0	0
COMP	0	6	2	COMP	7	3	1	1	0	1
COMP	0	7	3	COMP	0	3	1	0	0	0
COMP	0	0	3	HALT	0	0	0	1	0	1
HALT	0	0	0	HALT	0	0	0	0	0	0
READA	1	X	X	READA	0	0	X	X	X	X
COMP	1	X	X	READA	0	0	X	X	X	X
HALT	1	X	X	READA	0	0	X	X	X	X

d) The Verilog code for the design is given below.

```
module mem (addr, data_in, we, clock, data_out);
  parameter Nawidth = 3;    // default 2^3 = 8 lines
  parameter Ndwidth = 8;    // default 8 bits per line
  input we, clock;          // write enable and clock
  input [Nawidth-1:0] addr;
  input [Ndwidth-1:0] data_in;
  output reg [Ndwidth-1:0] data_out;

  // memory array
  reg [Ndwidth-1:0] mem [2**Nawidth-1:0];

  always @(posedge clock) begin
      if (we) begin
          // synchronous write
          mem[addr] <= data_in;
      end
      // synchronous read
      data_out <= mem[addr];
  end
endmodule

module memProcessing (
  input clock,
  input Reset,
  input Init,
  input [7:0] DataInA
);

  // State definitions
  parameter READA = 2'b00;
  parameter COMP = 2'b01;
  parameter HALT = 2'b10;

  // Wire and register definitions
  reg [2:0] AddrA;
  reg [1:0] AddrB;
  wire [7:0] DOut1;
  reg [7:0] DOut2;
  reg [7:0] DataInB;
  reg [7:0] ADDOut;
  reg [7:0] SUBOut;
  reg Sign;

  reg IncA;
  reg IncB;
  reg WEA;
  reg WEB;

  reg [1:0] state;
  reg [1:0] next_state;

  // Memory A
  mem #(.Nawidth(3), .Ndwidth(8)) MemoryA (
      .clock(clock),
      .addr(AddrA),
      .data_in(DataInA),
      .we(WEA),
      .data_out(DOut1)
  );

  // Memory B
  mem #(.Nawidth(2), .Ndwidth(8)) MemoryB (
      .clock(clock),
      .addr(AddrB),
      .data_in(DataInB),
      .we(WEB)
  );

  // State transition logic
  always @(*) begin
      // Initialize
      next_state = state;

      // Assign
      case (state)
          READA:
              if (Init) next_state = READA;
              else if (AddrA == 7)
                next_state = COMP;
              else
                next_state = READA;
          COMP:
              if (Init) next_state = READA;
              else if (AddrA == 0 && AddrB == 3)
                  next_state = HALT;
              else
                  next_state = COMP;
          HALT:
              if (Init) next_state = READA;
              else next_state = HALT;
          default:
              next_state = HALT;
      endcase
  end

  // State assignment logic
  always @(posedge clock or posedge Reset) begin
      if (Reset) state <= READA;
      else state <= next_state;
  end

  // Controller output logic
  always @(*) begin
    // Initialize
      WEA = 0;
      IncA = 0;
      WEB = 0;
      IncB = 0;
```

```
        // Assign
        case (state)
            READA: begin
                // Write and increment A
                WEA = 1;
                IncA = 1;
                WEB = 0;
                IncB = 0;
            end
            COMP: begin
                // Never write to A
                WEA = 0;
                // Increment A until we finish reading
                // i.e. check if AddrA wrapped around
                if (AddrA == 0 && AddrB == 3)
                    IncA = 0;
                else
                    IncA = 1;
                // Write and increment B every 2 cycles
                // i.e. check the last bit of AddrA
                // and dont write and increment in the first cycle
                if (AddrA[0] == 0 && !(AddrA == 0
                        && AddrB == 0)) begin
                    WEB = 1;
                    IncB = 1;
                end else begin
                    WEB = 0;
                    IncB = 0;
                end
            end
            HALT: begin
                // Halt and do nothing
                WEA = 0;
                IncA = 0;
                WEB = 0;
                IncB = 0;
            end
            default: begin
                WEA = 0;
                IncA = 0;
                WEB = 0;
                IncB = 0;
            end
        endcase
    end
```

```
  // Counter logic for AddrA
  always @(posedge clock or posedge Reset) begin
      if (Reset) AddrA <= 0;
      else if (Init) AddrA <= 0;
      else if (IncA) AddrA <= AddrA + 1;
  end

  // Counter logic for AddrB
  always @(posedge clock or posedge Reset) begin
      if (Reset) AddrB <= 0;
      else if (Init) AddrB <= 0;
      else if (IncB) AddrB <= AddrB + 1;
  end

  // DOut2 logic
  always @(posedge clock or posedge Reset) begin
      if (Reset) DOut2 <= 0;
      else if (Init) DOut2 <= 0;
      else DOut2 <= DOut1;
  end

  // Data processing logic
  always @(*) begin
      // Initialize
      ADDOut = 0;
      SUBOut = 0;
      Sign = 0;
      DataInB = 0;

      // Assign
      ADDOut = DOut2 + DOut1;
      SUBOut = DOut2 - DOut1;
      Sign = DOut2 <= DOut1;

      if (Sign) DataInB = ADDOut;
      else DataInB = SUBOut;
  end
endmodule
```

[Exercise 41] Two Input Sequence Detector

Given two 1-bit inputs `w1` and `w2`, your task is to design a Moore finite state machine (FSM) that outputs a high signal when `w1` is equal to `w2` for four consecutive clock cycles or more. The initial output of the FSM is `0`, and an asynchronous power-on active-high reset signal is used to reset the FSM to its initial state. Below is an example of sequence detection:

w1:	0	1	1	0	1	1	1	0	0	0	1	1	0
w2:	1	1	1	0	1	0	1	0	0	0	1	1	1
output:	0	0	0	0	1	0	0	0	0	1	1	1	0

a) How many states are required to design the FSM?

b) Draw the state diagram of the FSM by clearly indicating the states and transitions.

c) Derive the state/output table of the FSM.

d) Implement the FSM in Verilog. Use the state/output table specification with the recommended guidelines, that is, to use three always blocks as it is easy to read, manage, and implement.

[Solution 41] Two Input Sequence Detector

a) The FSM requires five states to detect the sequence:

- State A: Zero matches (also the initial state)
- State B: Detected one match
- State C: Detected two matches
- State D: Detected three matches
- State E: Detected four or more matches

b) First, let us define an intermediate signal `w` that is high when `w1` is equal to `w2` in a particular clock cycle and low otherwise. The state diagram of the FSM is as follows:

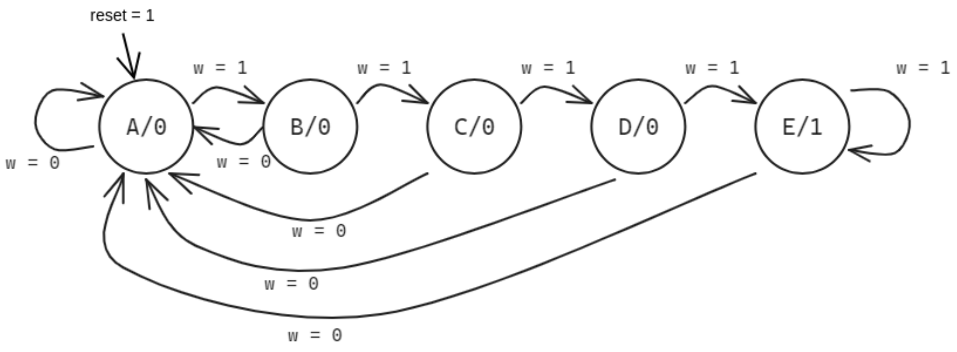


Figure 82: State diagram of the FSM

c) The state/output table of the FSM is as follows:

Present State	Next State		Output (Z)
	`w` = 0	`w` = 1	
A	A	B	0
B	A	C	0
C	A	D	0
D	A	E	0
E	A	E	1

Table 28: State/output table

d) Below is the Verilog description of the module:

```
module sequence_detector(
    input clk,
    input reset,
    input w1,
    input w2,
    output reg Z
);
    reg w;
    reg [2:0] state, next_state;
    parameter A = 3'b000, B = 3'b001,
              C = 3'b010, D = 3'b011, E = 3'b100;

    always @(*) begin
        next_state = A;
        w = (w1 == w2) ? 1 : 0;
        if (w == 1) begin
            case (state)
                A: next_state = B;
                B: next_state = C;
                C: next_state = D;
                D: next_state = E;
                E: next_state = E;
                default: next_state = A;
            endcase
        end else begin
            next_state = A;
        end
    end

    always @(posedge clk, reset)
    begin
        if (reset) begin
            state <= A;
        end else begin
            state <= next_state;
        end
    end

    always @(*) begin
        Z = (state == E) ? 1'b1: 1'b0;
    end
endmodule
```

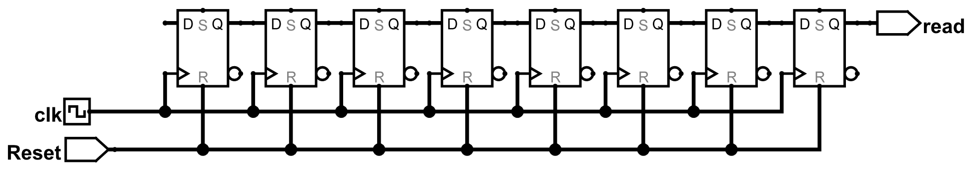
e) Below is a testbench to verify the functionality of the FSM that uses the example sequence provided in the question:

```
module testbench;

    reg clk, w1, w2, reset;
    wire Z;

    sequence_detector dut (
        .clk(clk),
        .reset(reset),
        .w1(w1),
        .w2(w2),
        .Z(Z)
    );

    initial begin
        $dumpfile("tb_sequence_detector.vcd");
        $dumpvars(0, testbench);

        clk = 0; reset = 1;
        w1 = 0; w2 = 1; #10; reset = 0;
        if (Z != 0)
            $error("Cycle 0 failed; expected 0, got %b", Z);
        w1 = 1; w2 = 1; #10;
        if (Z != 0)
            $error("Cycle 1 failed; expected 0, got %b", Z);
        w1 = 1; w2 = 1; #10; // same
        if (Z != 0)
            $error("Cycle 2 failed; expected 0, got %b", Z);
        w1 = 0; w2 = 0; #10; // same
        if (Z != 0)
            $error("Cycle 3 failed; expected 0, got %b", Z);
        w1 = 1; w2 = 1; #10; // same
        if (Z != 1)
            $error("Cycle 4 failed; expected 1, got %b", Z);
        w1 = 1; w2 = 0; #10; // different
        if (Z != 0)
            $error("Cycle 5 failed; expected 0, got %b", Z);
        w1 = 1; w2 = 1; #10; // same
        if (Z != 0)
            $error("Cycle 6 failed; expected 0, got %b", Z);
        w1 = 0; w2 = 0; #10; // same
        if (Z != 0)
            $error("Cycle 7 failed; expected 0, got %b", Z);
        w1 = 0; w2 = 0; #10; // same
        if (Z != 0)
```

```
                $error("Cycle 8 failed; expected 0, got %b", Z);
            w1 = 0; w2 = 0; #10; // same
            if (Z != 1)
                $error("Cycle 9 failed; expected 1, got %b", Z);
            w1 = 1; w2 = 1; #10; // same
            if (Z != 1)
                $error("Cycle 10 failed; expected 1, got %b", Z);
            w1 = 1; w2 = 1; #10; // same
            if (Z != 1)
                $error("Cycle 11 failed; expected 1, got %b", Z);
            w1 = 0; w2 = 1; #10; // different
            if (Z != 0)
                $error("Cycle 12 failed; expected 0, got %b", Z);
            $finish;
        end

        always begin
            #5 clk = ~clk;
        end
    endmodule
```

[Exercise 42] Bit counting

a) Write the Verilog description of a Moore FSM designed to count the number of bits with value one stored in an 8-bit shift register. The shift register consists of eight D flip-flops connected together in a chain as follows:



An 8-bit value can be loaded into the shift register in one clock cycle, but only the LSB (i.e., the value stored in the rightmost D flip-flop) can be read in one clock cycle. When enabled, the shift register shifts its stored value right by one position, i.e., each D flip-flop reads the value of the D flip-flop to its left.

The Verilog module should have the following interface and name:

```
module bit_counter(
    input clk,
    input reset,
    input [7:0] load_data,
    input load,
    input enable,
    output reg [3:0] count,
    output reg done
);
```

The functionality of this module should be as follows:

- The `reset` is an active-high asynchronous signal and used to reset the FSM.
- The `load_data` value should be loaded when the `load` signal is high.
- The `enable` signal is used to enable the FSM, i.e, if the `enable` signal is low, this entire operation should be paused.
- The `count` output should store the number of bits with value one in the 8-bit vector.
- The `done` output should be high when the counting is done.

The operation of this module can be described by the following logic:

After the circuit is reset, FSM is set to state `S1`. In state `S1`, when the `load` signal is set to one, the data on the `load_data` input is loaded into the internal 8-bit register. When the `enable` signal is set to one, the next rising edge of the clock causes the FSM to change to state `S2`. In state `S2`, at each rising edge of the clock, if the value of the LSB of the internal 8-bit register is one, the value of the `count` output is incremented by one. The value of the internal 8-bit register is then shifted to the right. When the internal 8-bit register becomes zero, the FSM goes to state `S3`, which means the counting has finished, and the `done` signal is set to one. To start counting again with new data, the entire process is repeated.

b) Use the testbench below to verify the functionality of your Verilog module. Analyze the waveforms of your module to see if it acts as expected.

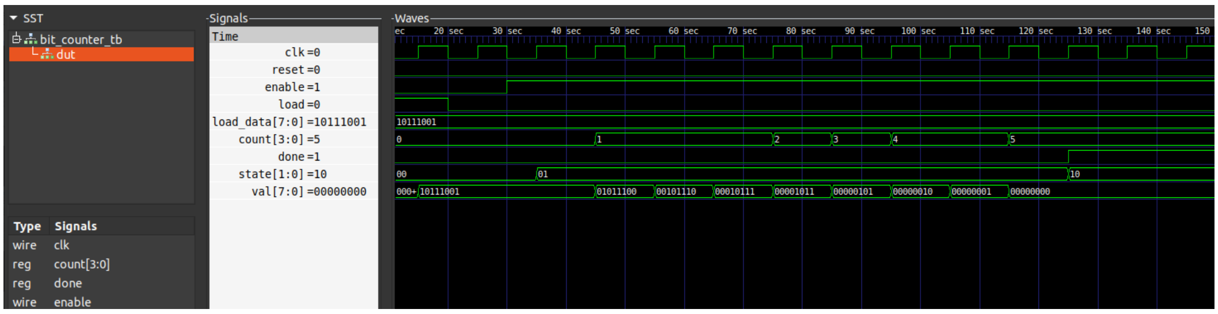
```
module bit_counter_tb;
    reg [7:0] load_data;
    reg clk, reset, load, enable;
    wire [3:0] count;
    wire done;

    bit_counter dut (.clk(clk), .reset(reset),
        .load_data(load_data), .load(load),
        .enable(enable), .count(count), .done(done)
    );

    initial begin
        $dumpfile("dump.vcd");
        $dumpvars(0, bit_counter_tb);
        clk = 0;
        reset = 1;
        enable = 0;
        load = 0;
        load_data = 8'b10111001;
        #10 reset = 0;
        load = 1;
        #10 load = 0;
        #10 enable = 1;
        #150
        $finish;
    end

    always #5 clk = ~clk;
endmodule
```

[Solution 42] Bit counting

a) Below is a possible Verilog description of the module:

```
module bit_counter(
    input clk,
    input reset,
    input [7:0] load_data,
    input load,
    input enable,
    output reg [3:0] count,
    output reg done
);
    reg [7:0] val, next_val;      // Internal 8-bit register value
    reg [1:0] state, next_state; // To keep track of the state
    reg [3:0] next_count;         // To keep track of count

    // Three states are needed
    parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;

    // Logic to determine the next state, value and count
    always @(*) begin
        next_state = S1;
        next_val = 8'b0;
        next_count = 4'b0;
        case (state)
            S1: begin
                next_state = (enable == 1'b1) ? S2 : S1;
                next_val = (load == 1'b1) ? load_data : val;
                next_count = 4'b0;
            end
            S2: begin
                next_state = (val == 8'b0) ? S3 : S2;
                next_val = (enable == 1'b1) ? val >> 1 : val;
                next_count = ((val[0] == 1'b1) & (enable == 1'b1)) ?
                             count + 1 : count;
            end
            S3: begin
                next_state = (load == 1'b1) ? S1 : S3;
                next_val = val;
                next_count = count;
            end
            default: begin
                next_state = S1;
                next_val = 8'b0;
                next_count = count;
            end
        endcase
    end

    // Logic to update FFs
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            state <= S1;
            val <= 8'b0;
            count <= 4'b0;
        end else begin
            state <= next_state;
            val <= next_val;
            count <= next_count;
        end
    end

    // Logic to change the output signal done
    always@(*) begin
        done = (state == S3);
    end
endmodule
```

b) The simulation obtained with the above description:

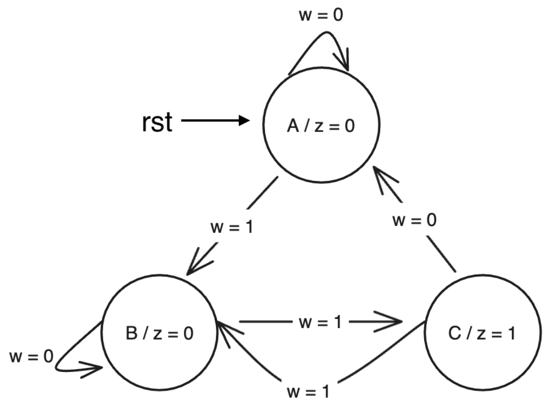


```
Time
              clk =0
            reset =0
           enable =1
             load =0
load_data[7:0] =10111001
      count[3:0] =5
             done =1
      state[1:0] =10
        val[7:0] =00000000
```

Type	Signals
wire	clk
reg	count[3:0]
reg	done
wire	enable

[Exercise 43] Moore FSM

The state diagram of a Moore machine with one bit input `w` and one bit output `z` is given below. The machine has three states, A, B, and C. The output `z` is one when the machine is in state C and 0 otherwise. The machine has a synchronous active-high reset signal `rst` and the initial state is A.



The states are encoded with two bits and their corresponding values are as follows: A = $(00)_2$, B = $(01)_2$, and C = $(11)_2$.

a) Draw the state transition table of this Moore machine.

b) Draw the circuit of this state machine using only two-input and three-input `AND` gates and two-input `OR` gates. You can use the inverted outputs of the flip-flops.

c) Find the maximum operating frequency of the implementation of the circuit in part (b) with the following timing constraints:

- $t_{\text{setup}} = 100$ ps
- $t_{\text{hold}} = 100$ ps
- $t_{\text{cQ,min}}$ (minimum clock-to-q delay)$= 150$ ps
- $t_{\text{cQ,max}}$ (maximum clock-to-q delay)$= 200$ ps
- $t_{\text{AND2}} = 300$ ps
- $t_{\text{AND3}} = 400$ ps
- $t_{\text{OR2}} = 400$ ps

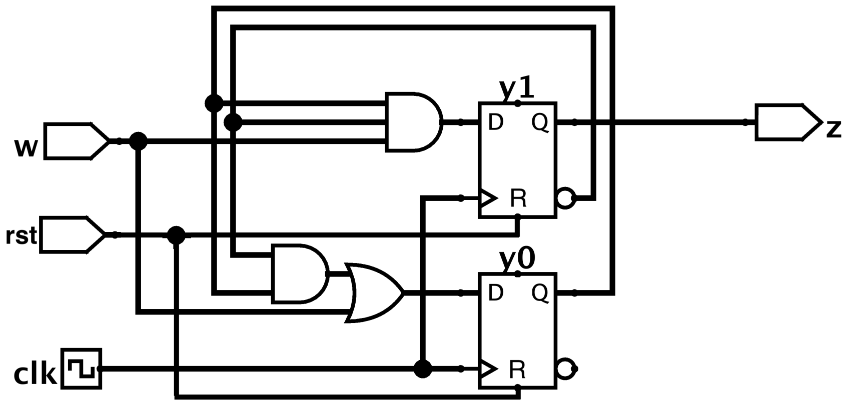
[Solution 43] Moore FSM

a) The state transition table of the Moore machine is as follows:

Current State y_1y_0	**Input** w	**Reset** rst	**Next State** $y_1'y_0'$	**Output** z
00	0	0	00	0
00	1	0	01	0
01	0	0	01	0
01	1	0	11	0
11	0	0	00	1
11	1	0	01	1
00	x	1	00	0
01	x	1	00	0
11	x	1	00	1

b) The circuit of the Moore machine is given below. Please note that the flip-flops have synchronous reset inputs. The next state and output logic is as follows:

- $y_1' = \overline{y_1}y_0w$
- $y_0' = w + \overline{y_1}y_0$
- $z = y_1$



c) The fastest clock frequency is obtained when

$$t_{cQ,\max} + t_{\text{comb, max}} + t_{\text{setup}} = T_{\text{CLK}} = 1/f_{\text{CLK}}$$

The longest combinatorial path goes through one two-input AND gate and one two-input OR gate, giving:

$$200 \text{ ps} + 300 \text{ ps} + 400 \text{ ps} + 100 \text{ ps} = T_{\text{CLK}} = 1000 \text{ ps}$$

$$f = 1/(1000 \times 10^{-12} \text{ s}) = 1 \times 10^{9} \text{ Hz} = 1 \text{ GHz}$$

Part III: RISC-V

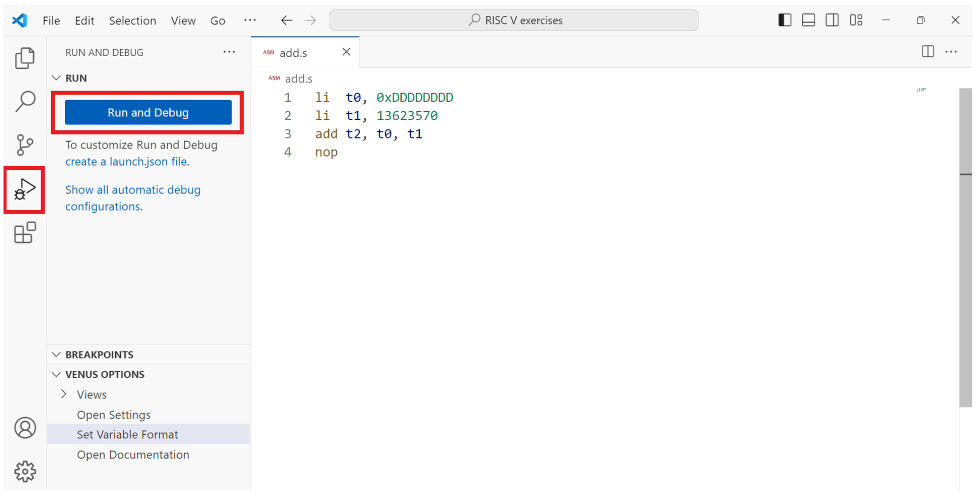
[Exercise 1] My First Program

Let us write a simple program that adds two 32-bit integer variables and stores the result in a third one. To get started, follow the steps below:

1. Open Visual Studio Code
2. Create a new file and save it as `add.s`
3. Write the following code in the file

```
li  t0, 0xDDDDDDDD
li  t1, 13623570
add t2, t0, t1
nop
```

4. From the sidebar, open the *Run and Debug* menu and click on the button of the same name (see Fig. 83).

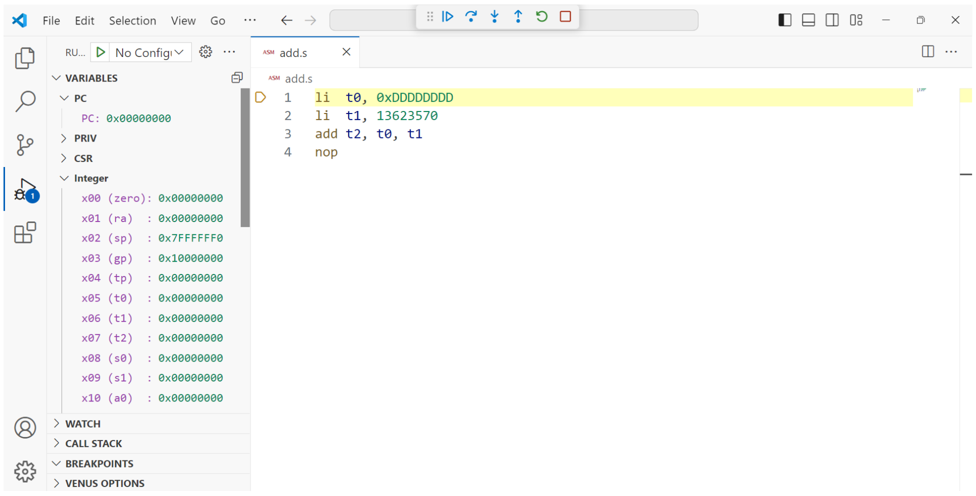


```
li  t0, 0xDDDDDDDD
li  t1, 13623570
add t2, t0, t1
nop
```

Figure 83: Run and Debug

As a result, more windows will open (see Fig. 84). The newly open left window provides multiple sections that will be useful for debugging, in particular:

- **VARIABLES / Integer**: Displays the registers and their contents. You can also modify them for debugging purposes.
- **BREAKPOINTS**: Manage breakpoints, which are used to pause the program's execution at a specific line of code.
- **VENUS OPTIONS / Views**: Toggle components (Memory, LEDs, etc.).
- **VENUS OPTIONS / Set variable format**: Change the number representations (Bin/Dec/Hex).



```
PC: 0x00000000
```

```
x00 (zero): 0x00000000
x01 (ra)  : 0x00000000
x02 (sp)  : 0x7FFFFFF0
x03 (gp)  : 0x10000000
x04 (tp)  : 0x00000000
x05 (t0)  : 0x00000000
x06 (t1)  : 0x00000000
x07 (t2)  : 0x00000000
x08 (s0)  : 0x00000000
x09 (s1)  : 0x00000000
x10 (a0)  : 0x00000000
```

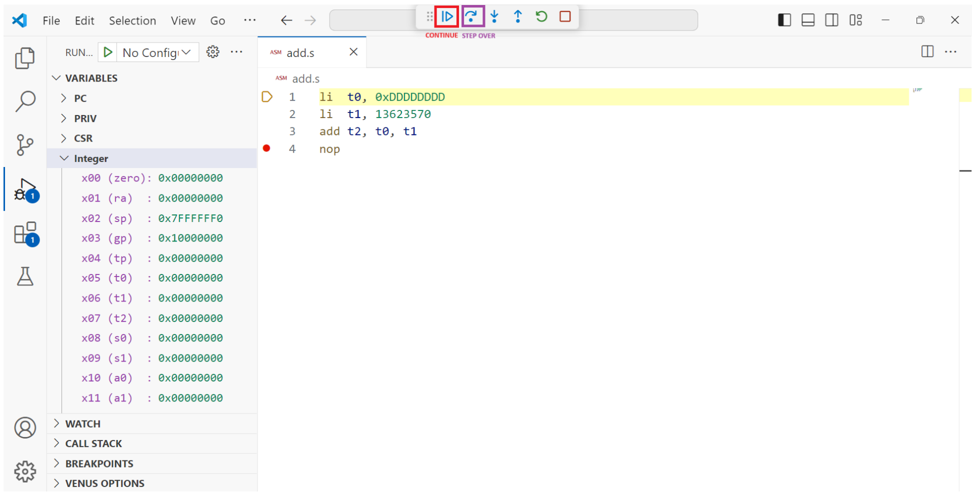
```
li  t0, 0xDDDDDDDD
li  t1, 13623570
add t2, t0, t1
nop
```

Figure 84: Debug view

This simple program has four lines of code, performing the following operations:

1. Line 1: Write the immediate value `0xDDDDDDDD` into register `t0`.
2. Line 2: Write the immediate value `13623570` into register `t1`.
3. Line 3: Add the values of registers `t0` and `t1` and save the result in register `t2`.
4. Line 4: No operation. Adding this seemingly useless line of code allows us to observe the result of the addition (i.e., the contents of register `t2`) before the program ends.

The next step is to use the debugging tools to see the result of the addition. Start by adding a breakpoint on line four by clicking left to the line number. Adding the breakpoint at this location will pause the program's execution immediately after the addition. Then, run the program and press *Continue* once (see Fig. 85).



```
1   li  t0, 0xDDDDDDDD
2   li  t1, 13623570
3   add t2, t0, t1
4   nop
```

```
x00 (zero): 0x00000000
x01 (ra)  : 0x00000000
x02 (sp)  : 0x7FFFFFF0
x03 (gp)  : 0x10000000
x04 (tp)  : 0x00000000
x05 (t0)  : 0x00000000
x06 (t1)  : 0x00000000
x07 (t2)  : 0x00000000
x08 (s0)  : 0x00000000
x09 (s1)  : 0x00000000
x10 (a0)  : 0x00000000
x11 (a1)  : 0x00000000
```

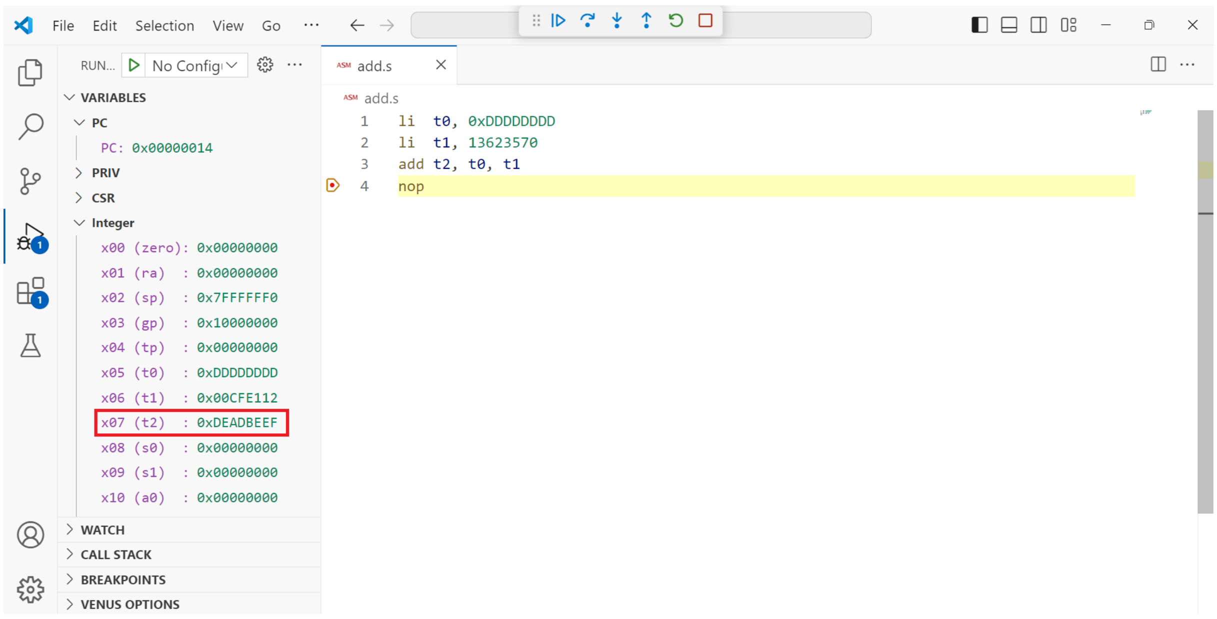
Figure 85: Debugging steps

Continue runs the program up to the next breakpoint, whereas *Step over* would execute one program line.

The program has now been paused after the addition. Look at the variables (Integer section) to find the contents of register `t2`. What do you see? Remember that to change the format of the registers, you should navigate to "Venus Options" and "Set Variable Format."

[Solution 1] My First Program

The value of the register at the end of the program is `0xDEADBEEF` (see Fig. 86).



```
VARIABLES
  PC
    PC: 0x00000014
  PRIV
  CSR
  Integer
    x00 (zero): 0x00000000
    x01 (ra)  : 0x00000000
    x02 (sp)  : 0x7FFFFFF0
    x03 (gp)  : 0x10000000
    x04 (tp)  : 0x00000000
    x05 (t0)  : 0xDDDDDDDD
    x06 (t1)  : 0x00CFE112
    x07 (t2)  : 0xDEADBEEF
    x08 (s0)  : 0x00000000
    x09 (s1)  : 0x00000000
    x10 (a0)  : 0x00000000
```

```
li  t0, 0xDDDDDDDD
li  t1, 13623570
add t2, t0, t1
nop
```

Figure 86: Register values at the end of the program

Note: This exercise is a simple introduction to the debugging tools inspired by the [RISC-V Venus Simulator embedded in VS Code page]. This page contains multiple exercises to help you get started with the debugging tools, although most of them cover more advanced topics. We encourage you to explore them to get a better understanding of what Venus offers.

[Exercise 2] Bit Busters

Alice tried to write a program to count the number of bits set to one in a register. The program is as follows:

```
    li   x1, 0x00123456
    li   x2, 0
    li   x3, 1
    li   x4, 0

loop:
    and  x4, x1, x3
    add  x2, x2, x4
    srai x1, x1, 1
    bne  x1, x0, loop
```

Note that register x0 is hardwired to constant zero (i.e., its contents cannot be changed).

a) Run the program and keep track of the value of register x1 at the beginning of each of the first five iterations of the loop. When does the program terminate? What is the final value of register x2?

b) Bob took Alice's code and ran it with the value 0x8BADF00D in register x1. Surprisingly, Bob found that the result was not what he expected.

Repeat Bob's experiment: run the program and note the value of register x1 at the beginning of each of the first five iterations of the loop. When does the program terminate? Can you find the issue with Alice's code and fix it?

[Solution 2] Bit busters

a) Set a breakpoint at the beginning of the loop (line seven) and run the program. We then press *continue* to run the program until the next breakpoint. We repeat this four more times to get the value of `x1` at the beginning of the first five iterations of the loop. The value of the register `x1` at each iteration is:

Iteration	Hexadecimal	Binary
0	`0x00123456`	`0b00000000000100100011010001010110`
1	`0x00091A2B`	`0b00000000000010010001101000101011`
2	`0x00048D15`	`0b00000000000001001000110100010101`
3	`0x0002468A`	`0b00000000000000100100011010001010`
4	`0x00012345`	`0b00000000000000010010001101000101`

One easy way to find the value of `x2` at the end of the program is to insert a dummy instruction at the end of the program, such as `nop`. We add a breakpoint on that line and disable other breakpoints. We then run the program and press *continue*. The value of `x2` at the end of the program is nine.

b) The value of register `x1` at the beginning of the first five iterations of the loop are as follows:

Iteration	Hexadecimal	Binary
0	`0x8BADF00D`	`0b10001011101011011111000000001101`
1	`0xC5D6F806`	`0b11000101110101101111100000000110`
2	`0xE2EB7C03`	`0b11100010111010110111110000000011`
3	`0xF175BE01`	`0b11110001011101011011111000000001`
4	`0xF8BADF00`	`0b11111000101110101101111100000000`

The program should terminate when `x1` equals zero. However, as the table above shows, the value of `x1` will never be zero.

The issue with Alice's program is that it uses a `srai` instruction to shift the value of `x1` to the right. This instruction fills the most significant bits with the sign bit, which is one. Therefore, the value of `x1` will never be zero, and the program will never terminate.

We should use the `srli` instruction instead of `srai` to fix this issue.

An alternative and less error-prone implementation of bit counting algorithm would be to terminate the loop after exactly 32 iterations.

[Exercise 3] Understanding and Encoding Instructions

Consider the following RISC-V instructions:

1. `add t0, t1, t2`
2. `slt t4, s0, s3`
3. `sltu t2, t2, t5`
4. `addi t0, t1, 16`
5. `slli s2, s2, 0x3`
6. `lui t6, 0x00012300`

Answer the question below for each instruction:

a) Assuming the instruction is stored at addresses `0x0`, determine the value of the data byte found at memory address `0x2`. Consider little-endian byte order.

[Solution 3] Understanding and Encoding Instructions

a) Below are the RISC-V binary and hexadecimal encoding of each instruction.

1. **`add`** `t0, t1, t2`:

funct7	rs2	rs1	funct3	rd	opcode
0000000	00111	00110	000	00101	0110011

   Hexadecimal: `0x007302B3`

   The value of the data byte at memory address `0x2` is `0x73`.

2. **`slt`** `t4, s0, s3`:

funct7	rs2	rs1	funct3	rd	opcode
0000000	10011	01000	010	11101	0110011

   Hexadecimal: `0x01342EB3`

   The value of the data byte at memory address `0x2` is `0x34`.

3. **`sltu`** `t2, t2, t5`:

funct7	rs2	rs1	funct3	rd	opcode
0000000	11110	00111	011	00111	0110011

   Hexadecimal: `0x01E3B3B3`

   The value of the data byte at memory address `0x2` is `0xE3`.

4. **`addi`** `t0, t1, 16`:

`imm[11:0]`	`rs1`	`funct3`	`rd`	`opcode`
000000010000	00110	000	00101	0010011

Hexadecimal: `0x01030293`

The value of the data byte at memory address `0x2` is `0x03`.

5. **`slli`** `s2, s2, 0b011`:

`shamt`	`rs1`	`funct3`	`rd`	`opcode`
000000000011	10010	001	10010	0010011

Hexadecimal: `0x00391913`

The value of the data byte at memory address `0x2` is `0x39`.

6. **`lui`** `t6, 0x00012300`:

`imm[31:12]`	`rd`	`opcode`
00010010001100000000	11111	0110111

Hexadecimal: `0x12300FB7`

The value of the data byte at memory address `0x2` is `0x30`.

[Exercise 4] Triple XOR

Consider the following sequence of RISC-V instructions:

```
xor t0, t0, t2
xor t2, t0, t2
xor t0, t0, t2
```

Suppose the initial values in registers `t0` and `t2` are `0x12345678` and `0xABCDEF00`, respectively.

a) Determine the values in registers `t0` and `t2` after executing each instruction in the sequence. Verify your answers with the help of the Venus RISC-V simulator.

b) What has this sequence of `xor` instructions achieved? Knowing the properties of the XOR logic operation, explain why the values in the registers change the way you observed, after each instruction.

[Solution 4] Triple XOR

a) The values in registers `t0` and `t2` after executing each instruction in the sequence are:

After execution of instruction	`t0`	`t2`
0 (Initally)	`0x12345678`	`0xABCDEF00`
1	`0xB9F9B978`	`0xABCDEF00`
2	`0xB9F9B978`	`0x12345678`
3	`0xABCDEF00`	`0x12345678`

b) The purpose of this sequence of instructions is to swap the values in registers `t0` and `t2`. This is a known technique for swapping the values of two variables without using a temporary variable. The sequence of instructions can be explained as follows:

1. **`xor`** `t0, t0, t2` — This XORs the value of `t0` with the value of `t2` and writes the result back in `t0`. So `t0` now contains the value $\texttt{t0} \oplus \texttt{t2}$.

2. **`xor`** `t2, t0, t2` — This XORs the new value in `t0` (which is $\texttt{t0} \oplus \texttt{t2}$) with the original value of `t2`, and writes the result in `t2`. This effectively assigns `t2` the initial value of `t0`, because:

$$\begin{aligned}(\texttt{t0} \oplus \texttt{t2}) \oplus \texttt{t2} &= \texttt{t0} \oplus (\texttt{t2} \oplus \texttt{t2})\\ &= \texttt{t0} \oplus 0\\ &= \texttt{t0}\end{aligned}$$

3. **`xor`** `t0, t0, t2` — This XORs the value in `t0` (which is $\texttt{t0} \oplus \texttt{t2}$) with the value in `t2` (which is now the original value of `t0`), and writes the result back in `t0`. This completes the swap, because:

$$\begin{aligned}(\texttt{t0} \oplus \texttt{t2}) \oplus \texttt{t0} &= \texttt{t0} \oplus (\texttt{t0} \oplus \texttt{t2})\\ &= (\texttt{t0} \oplus \texttt{t0}) \oplus \texttt{t2}\\ &= 0 \oplus \texttt{t2}\\ &= \texttt{t2}\end{aligned}$$

[Exercise 5] Checking for Overflows

In many programming environments, ignoring arithmetic overflow can lead to unexpected results. In particular, RISC-V relies on software to handle overflow checking. This means that developers must implement their own mechanisms to detect and handle overflow conditions in their code.

Your task in this exercise is to write RISC-V assembly code to handle overflow detection for both unsigned and signed integer addition. If an overflow occurs, you should set a flag in the temporary register `t6`, i.e. set the value of `t6` to one.

a) Write a sequence of RISC-V instructions to detect overflow in unsigned integer addition.

Hint: Recall the `sltu` instruction (Set Less Than Unsigned). You may use it to compare two registers as if they were unsigned integers and set the value of the destination register to one.

b) Write a sequence of RISC-V instructions to detect overflow in signed integer addition.

Hint: You should make a total of three comparisons to detect overflow in signed integer addition.

[Solution 5] Checking for Overflows

a) Unsigned addition overflows when the result is smaller than either of the operands. The following sequence of instructions detects overflow in unsigned integer addition:

```
add  t0, t1, t2
sltu t6, t0, t1
```

To see how this code works, let's consider the following example. Let:

```
t1 = 0xFFFFFFFF
t2 = 0x00000001
```

Then:

```
add  t0, t1, t2 # t0 = 0x00000000
                # (overflow occurred)

sltu t6, t0, t1 # (0 < 0xFFFFFFFF) = 1
```

So, `t6` is set to `1`, indicating that an unsigned overflow has occurred.

b) The idea is that the sum of the two registers should be less than one of the operands if and only if the other operand is negative. The following sequence of instructions detects overflow in signed integer addition:

```
add   t0, t1, t2        # t0 = t1 + t2
slti  t3, t2, 0         # t3 = (t2 < 0) ? 1 : 0
slt   t4, t0, t1        # t4 = (t0 < t1) ? 1 : 0
xor   t6, t3, t4        # t6 = t3 ^ t4
```

Below is the explanation of the code:

- **`add t0, t1, t2`**: Computes the sum of `t1` and `t2` and writes the result in `t0`.
- **`slti t3, t2, 0`**: Sets `t3` to one if the value in `t2` is negative, and to zero otherwise.
- **`slt t4, t0, t1`**: Sets `t4` to one if `t0`, i.e., `t1` + `t2`, is less than `t1`, and to zero otherwise.
- **`xor t6, t3, t4`**: Checks whether both `t3` and `t4` flags are set. If only one of them is set, signed overflow has occurred.

To see how this code works in practice, let's walk through examples. Note that signed overflow occurs when:

- Adding two positive numbers produces a negative result.
- Adding two negative numbers produces a positive result.

Case 1: adding two positive numbers

Let:

```
t1 = 0x7FFFFFFF
t2 = 0x00000001
```

Then:

```
add   t0, t1, t2        # t0 = 0x80000000
slti  t3, t2, 0         # t3 = 0
slt   t4, t0, t1        # t4 = 1
xor   t6, t3, t4        # t6 = 1
```

So, `t6` is set to `1`, indicating signed overflow.

Case 2: adding two negative numbers

Let:

```
t1 = 0x80000000
t2 = 0xFFFFFFFF
```

Then:

```
add   t0, t1, t2        # t0 = 0x7FFFFFFF
slti  t3, t2, 0         # t3 = 1
slt   t4, t0, t1        # t4 = 0
xor   t6, t3, t4        # t6 = 1
```

Again, `t6` is set to `1`, correctly detecting signed overflow.

[Exercise 6] Encoding Branches and Memory Instructions

Consider the following instructions:

```
beq  t0, t1, 42
bltu s1, s2, -4
lw   t2, 12(s3)
sb   t0, 4(s6)
```

For each instruction, write the binary and hexadecimal encoding of the instruction in RISC-V.

[Solution 6] Encoding Branches and Memory Instructions

Below are the RISC-V binary and hexadecimal encoding of each instruction.

1. `beq t0, t1, 42`:

`imm[12|10:5]`	`rs2`	`rs1`	`funct3`	`imm[4:1|11]`	`opcode`
0|000001	00110	00101	000	0101|0	1100011

Hexadecimal: `0x02628563`

2. `bltu s1, s2, -4`:

`imm[12|10:5]`	`rs2`	`rs1`	`funct3`	`imm[4:1|11]`	`opcode`
1|111111	10010	01001	110	1110|1	1100011

Hexadecimal: `0xFF24EEE3`

3. `lw t2, 12(s3)`:

`imm[11:0]`	`rs1`	`funct3`	`rd`	`opcode`
000000001100	10011	010	00111	0000011

Hexadecimal: `0x00C9A383`

4. `sb t0, 4(s6)`:

`imm[11:5]`	`rs2`	`rs1`	`funct3`	`imm[4:0]`	`opcode`
0000000	00101	10110	000	00100	0100011

Hexadecimal: `0x005B0223`

[Exercise 7] Understanding RISC-V (II)

Consider the following RISC-V program. Registers s0 and s1 are initialized as follows:

- s0: The starting memory address of a contiguous data array of 32-bit integers.
- s1: The number of elements in the data array.

```
mysteriouscode:
    li   t0, 0
    add  t1, zero, s0
    lw   t2, 0(s0)
    lw   t3, 0(s0)
label0:
    lw   t5, 0(t1)
    slt  t4, t2, t5
    bne  t4, zero, label1
    add  t2, zero, t5
label1:
    slt  t4, t5, t3
    bne  t4, zero, label2
    add  t3, zero, t5
label2:
    addi t0, t0, 1
    addi t1, t1, 4
    bne  s1, t0, label0
    add  t4, t2, t3
    srai s2, t4, 1
```

a) This program computes something and places the result in register s2. Describe in a sentence what the program computes. Is the result it produces always accurate, i.e., is it an approximate or the exact value?

b) Does the code assume the array contains signed or unsigned integers? Explain your answer. Knowing which of the two data types the code currently supports, propose code modifications necessary to handle the currently unsupported data type.

[Solution 7] Understanding RISC-V (II)

a) The program's purpose is to find the arithmetic mean of the array's minimum and maximum values.

The reported mean is not always accurate. Rather, it is an approximate and not an exact value because of the integer division by two in the last instruction.

A commented version of the code with an example array that can be run directly in VSCode is given below:

```
.data
    array: .word 2, -1, 10, 90, -7, 40, 96, 23

.text
    la   s0, array          # s0 points to the array
    li   s1, 8              # loading length of array in s1
mysteriouscode:
    li   t0, 0              # init element counter (t0)
    add  t1, zero, s0       # t1 points to 1st element of array
    lw   t2, 0(s0)          # init min val (t2) with 1st elem
    lw   t3, 0(s0)          # init max val (t3) with 1st elem
label0:
    lw   t5, 0(t1)          # load the current element into t5
    slt  t4, t2, t5         # (t2 (min) >= t5 (curr)) => t4 = 0
    bne  t4, zero, label1   # if min < current, skip min update
    add  t2, zero, t5       # update min value in t2
label1:
    slt  t4, t5, t3         # (t5 (curr) >= t3 (max)) => t4 = 0
    bne  t4, zero, label2   # if curr < max, skip max update
    add  t3, zero, t5       # update max value in t3
label2:
    addi t0, t0, 1          # increment element counter
    addi t1, t1, 4          # move to next element
    bne  s1, t0, label0     # repeat as long as elements remain
    add  t4, t2, t3         # t4 = min + max
    srai s2, t4, 1          # s2 = (min + max)/2
```

b) The elements of the array pointed to by `s0` must be signed because the comparisons used to find the maximum and minimum are done on signed numbers using the `slt` instruction. The mean calculations are also done using signed numbers. Thus the program must be modified as follows to handle unsigned numbers:

```
...
slt  t4, t2, t5  =>  sltu t4, t2, t5
...
slt  t4, t5, t3  =>  sltu t4, t5, t3
...
srai s2, t4, 1   =>  srli s2, t4, 1
```

[Exercise 8] Comparing Signs

Write a program in RISC-V assembly that analyzes two data arrays of the same size, each containing 32-bit signed integers. The program compares the sign of the element at index i of the first array with the sign of the element at the same index i of the second array, covering all indices i from the beginning to the end of the arrays. The program reports the number of times the comparison revealed the signs were different.

You can assume that the registers `s0`, `s1`, and `s2` are initialized as follows:

- `s0`: The memory address of the first data array.
- `s1`: The memory address of the second data array.
- `s2`: The number of elements in each of the two data arrays.

Your program should not modify the above registers. At the end, register `s3` should contain the result, i.e., the number of times the elements at the same index in both arrays had different signs.

[Solution 8] Comparing Signs

The RISC-V program is given below. Two arrays of 32 bits are written in the data section to serve as input for the program. The program compares the sign of each pair of elements from the two arrays and counts the number of pairs with different signs.

```
.data
    arr1:   .word 2, -3, 10, -5, 7, 200, -100,  23
    arr2:   .word 1,  3, 11,  6, 6, -10,   27, -57

.text
    la   s0, arr1            # Load address of 1st array into s0
    la   s1, arr2            # Load address of 2nd array into s1
    li   s2, 8               # Number of elements in the arrays

    li   s3, 0               # init counter/result
    beq  s2, zero, finish    # if no elements, finish

    add  t0, zero, s0        # t0 points at 1st array elements
    add  t1, zero, s1        # t1 points at 2nd array elements
    add  t6, zero, s2        # copy number of elements to t6
loop:
    lw   t3, 0(t0)           # load an element from 1st array
    lw   t4, 0(t1)           # load an element from 2nd array
    xor  t5, t3, t4          # t5 = t3 XOR t4
    srli t5, t5, 31          # t5 = sign bit of t3 XOR t4
    add  s3, s3, t5          # add sign bit to counter

    addi t6, t6, -1          # decrement remaining elements
    addi t0, t0, 4           # move to next element in 1st array
    addi t1, t1, 4           # move to next element in 2nd array
    bne  t6, zero, loop      # repeat if more elements
finish:
    nop                      # end of program
```

[Exercise 9] A Special Addition

As we know, RISC-V registers are 32 bits wide. Sometimes, however, it might be necessary to perform operations on numbers that are larger than 32 bits. Keeping this context in mind, consider the following RISC-V program:

```
add  t0, t2, t4
sltu t6, t0, t2
add  t7, t3, t5
add  t1, t7, t6
```

Describe in a sentence what the program does. What is the purpose of the `sltu` instruction? What is the purpose of the `add` instructions that follow it?

[Solution 9] A Special Addition

The program performs multi-word unsigned addition—a technique for adding numbers larger than the processor's word size. In the RISC-V RV32I architecture, the processor's word size is 32 bits, meaning each register can hold 32 bits of data. For numbers larger than 32 bits, the value is split and held across multiple registers. For example, a 64-bit number is divided into two 32-bit parts, each placed in a separate register.

To perform multi-word unsigned addition, i.e., adding numbers that are split into multiple words, the lower 32 bits of the 64-bit number are added first, followed by the upper 32 bits. The addition of the lower 32 bits might also generate a carry, which can be detected by checking for the overflow. The generated carry should also be added to the addition of the upper 32 bits.

Let's break down the program:

- **`add`** `t0, t2, t4`: Adds the lower 32 bits of the two 64-bit numbers.
- **`sltu`** `t6, t0, t2`: By checking for overflow in the addition of the lower 32 bits, we can determine whether a carry was generated. Register `t6` is set to one if the addition of the lower 32 bits resulted in a carry and zero otherwise.
- **`add`** `t7, t3, t5`: Adds the upper 32 bits of the two 64-bit numbers.
- **`add`** `t1, t7, t6`: Adds the carry from the lower 32-bit addition to the upper 32-bit addition.

In this program, `t2` and `t4` hold the lower 32 bits of the two 64-bit numbers, while `t3` and `t5` hold the upper 32 bits. The `sltu` instruction is used to detect if there is a carry from the lower 32-bit addition. In the case of a carry, the upper 32-bit addition needs to account for the carry by adding one to the result. The `add` instruction that follows the `sltu` instruction adds the upper 32 bits and the next `add` adds the carry to the upper 32-bit addition.

Based on the overflow detection mechanism you learned in the previous exercise, you can also implement special addition for signed 64-bit values.

[Exercise 10] Length of a String

In multiple programming languages, strings are represented as arrays of characters terminated by a null character. Assume that a single character is represented by eight bits (one byte) and that the null character is represented by the value `0x00`.

Your task is to write a RISC-V program that calculates the length of the string. Your code should take the address of the first character of a string (array of characters) and output the length of the string (excluding the null character).

Consider the following:

- The address of the first character of the string is passed in register `s0`.
- At the end of the program, register `s1` should contain the length of the string.

Assume little-endian byte ordering. You should use only `lb` and `sb` instructions to access the memory.

[Solution 10] Length of a String

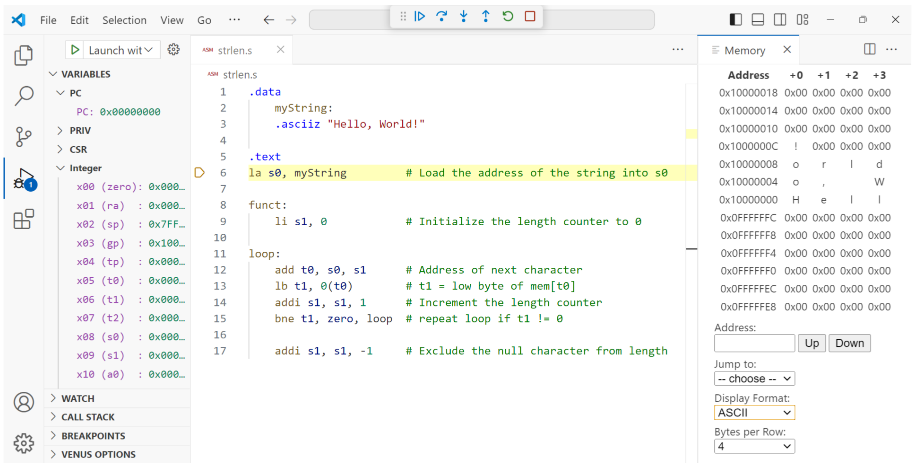
Below is the RISC-V assembly code:

```
    li   s1, 0            # Initialize the length counter to 0

loop:
    add  t0, s0, s1       # Address of next character
    lb   t1, 0(t0)        # t1 = low byte of mem[t0]
    addi s1, s1, 1        # Increment the length counter
    bne  t1, zero, loop   # repeat loop if t1 != 0

    addi s1, s1, -1       # Exclude null character from length
```

Want to try it out? Follow the instructions below: Open VSCode. Copy the code above and paste it into a .s file. Add the following at the top of the file:

```
.data
    myString:
    .asciiz "Hello, World!"
.text
    la   s0, myString     # Load address of string into s0
```

Run the program and open the memory view. In the *Jump to* menu, select *data* and in *Display format*, select *ASCII*. Your screen should look like this:



```
PC: 0x00000000
x00 (zero): 0x000…
x01 (ra)  : 0x000…
x02 (sp)  : 0x7FF…
x03 (gp)  : 0x100…
x04 (tp)  : 0x000…
x05 (t0)  : 0x000…
x06 (t1)  : 0x000…
x07 (t2)  : 0x000…
x08 (s0)  : 0x000…
x09 (s1)  : 0x000…
x10 (a0)  : 0x000…
```

```
.data
    myString:
    .asciiz "Hello, World!"

.text
la s0, myString         # Load the address of the string into s0

funct:
    li s1, 0            # Initialize the length counter to 0

loop:
    add t0, s0, s1      # Address of next character
    lb t1, 0(t0)        # t1 = low byte of mem[t0]
    addi s1, s1, 1      # Increment the length counter
    bne t1, zero, loop  # repeat loop if t1 != 0

    addi s1, s1, -1     # Exclude the null character from length
```

Address	+0	+1	+2	+3
0x10000018	0x00	0x00	0x00	0x00
0x10000014	0x00	0x00	0x00	0x00
0x10000010	0x00	0x00	0x00	0x00
0x1000000C	!	0x00	0x00	0x00
0x10000008	o	r	l	d
0x10000004	o	,		W
0x10000000	H	e	l	l
0x0FFFFFFC	0x00	0x00	0x00	0x00
0x0FFFFFF8	0x00	0x00	0x00	0x00
0x0FFFFFF4	0x00	0x00	0x00	0x00
0x0FFFFFF0	0x00	0x00	0x00	0x00
0x0FFFFFEC	0x00	0x00	0x00	0x00
0x0FFFFFE8	0x00	0x00	0x00	0x00

Address:
Jump to: -- choose --
Display Format: ASCII
Bytes per Row: 4

Figure 87: Memory view of the string "Hello, World!"

At the end, the value of register `s1` should be 13 (length of the string "Hello, World!").

[Exercise 11] Matrix Manipulations and Endianness

Consider the following RISC-V program, supposing that initially, i.e., at the beginning of the execution, registers s0, s1, s2, and s3 have the following values:

- s0: The starting memory address of a two-dimensional array (a matrix), which contains unsigned 8-bit numbers.
- s1: The number of rows in the matrix.
- s2: The number of columns in the matrix.
- s3: An arbitrary unsigned 8-bit number.

```
start:
    add  t0, zero, zero
    add  t1, zero, zero
    add  t3, zero, zero
    add  t5, s0, zero
loop:
    lbu  t2, 0(t5)
    add  t2, t2, s3
    slt  t4, t2, t3
    bne  t4, zero, skip
    add  s4, t0, zero
    add  s5, t1, zero
    add  t3, t2, zero
skip:
    sb   t2, 0(t5)
    addi t5, t5, 1
    addi t0, t0, 1
    bne  t0, s1, loop
    add  t0, zero, zero
    addi t1, t1, 1
    bne  t1, s2, loop
```

Consider that, in the memory, the matrix is stored in column-major order, i.e., the elements of the first column are stored first, the elements of the second column are stored next, and so on. The elements of a column are stored one after another in increasing order of their row indices. The rows and columns of the matrix are indexed starting from zero. Therefore, the starting memory address of the matrix (in register s0) is the address of the element at column index zero and row index zero.

Consider the matrix below, stored at the beginning of the `.data` region of memory. The top row has the index zero, and the leftmost column also has the index zero.

$$\begin{pmatrix} 12 & 34 & 56 \\ 78 & 113 & 24 \\ 35 & 46 & 57 \\ 11 & 122 & 33 \end{pmatrix}$$

a)

i) Find the values of registers `s4` and `s5`, at the end of the program execution. You may assume that `s3 = 0`.

ii) If `s3 = 0`, what is the value of the matrix element at row and column zero at the end of the program?

iii) If `s3 = 8`, what is the value of the matrix element at row and column zero at the end of the program?

iv) On code line 9, instruction **`slt`** is used. However, the matrix contains unsigned 8-bit numbers, and it would be intuitively appropriate to use instruction **`sltu`** instead. In what way does the choice between **`slt`** and **`sltu`** impact this program execution?

b) Assuming no overflows or loss of information (e.g., truncation of bits of the intermediate results), describe, in a nutshell, what this program does.

c) As mentioned, the matrix is stored at the beginning of the `.data` region in memory, i.e., starting from the address `0x1000 0000`. What are the memory contents at addresses from `0x1000 0000` to `0x1000 0008`? Assume little-endian byte ordering.

d) Assembly provides a `.byte` directive. If used in the `.data` section, this directive allows initializing memory one byte at a time. For example,

```
.data
    array: .byte 1, 2, 5, 0, 6, 3, 20, 7
```

will initialize an array of eight bytes with the values `[1 2 5 0 6 3 20 7]`.

Complete the program by adding memory initialization using the `.byte` directive, and the instructions for initializing the registers `s0`, `s1`, `s2`, and `s3`. Test the program.

e) Let us now relax the previous assumption that overflows and information loss (e.g., data truncation) cannot occur. Answer the following questions.

i) What is the maximum number of bits required to represent the result of the instruction `add t2, t2, s3` (code line 8)?

ii) In this code, there is an instruction that truncates the result of `add t2, t2, s3` and, under some circumstances, causes information loss. Find and explain it.

iii) One of the ways to handle data truncation would be to replace, when appropriate, the result of the instruction `add t2, t2, s3` with the largest value an 8-bit unsigned number can take. Modify the program to implement the proposed solution and test it in the RISC-V simulator.

[Solution 11] Matrix Manipulations and Endianness

a)

i) For the given matrix and `s3 = 0`, the maximum value in the matrix is 122, located in row three and column one. At the end of the execution, `s4` equals three and `s5` equals one.

ii) If `s3 = 0`, all elements of the matrix remain unchanged.

iii) If `s3 = 8`, all matrix elements will be incremented by eight.

iv) In the instruction `slt t4, t2, t3`, operand `t2` is a sum of `t2` and `s3`. Before this addition, `t2` is loaded by the instruction `lbu t2, 0(t5)`, which loads an unsigned byte into `t2` while setting the upper 24 bits to zero. Similarly, `s3` is also an unsigned 8-bit number with upper 24 bits equal to zero. The sum of `t2` and `s3`, therefore, would never make the MSB of `t2` equal to one, keeping the sum nonnegative.

The other operand `t3` of instruction `slt t4, t2, t3` is also an unsigned 8-bit number with upper 24 bits equal to zero. Given that both the operands of `slt` are positive, we can use `slt` without causing any incorrect behavior and even use `sltu` without any problem.

b) The program does two things:

1. For each matrix element, it adds the value in register `s3` and the matrix element and replaces the element with the resulting sum.
2. It finds the maximum value in the matrix and saves the row index and the column index of the maximum element to registers `s4` and `s5`, respectively.

A commented version of the program is given below:

```
start:
    add  t0, zero, zero     # Initialize row counter (t0)
    add  t1, zero, zero     # Initialize column counter (t1)
    add  t3, zero, zero     # Initialize max value (t3)
    add  t5, s0, zero       # Copy matrix address to t5

loop:
    lbu  t2, 0(t5)          # Load element from matrix
    add  t2, t2, s3         # Add s3 to element
    slt  t4, t2, t3         # Compare element with max value
    bne  t4, zero, skip     # Skip if element not greater
                            # than max
    add  s4, t0, zero       # Store row of max element in s4
    add  s5, t1, zero       # Store max element's column in s5
    add  t3, t2, zero       # Update max value
skip:
    sb   t2, 0(t5)          # Store updated element back in
                            # matrix
    addi t5, t5, 1          # Move to next element
    addi t0, t0, 1          # Increment row counter
    bne  t0, s1, loop       # Repeat for all rows
    add  t0, zero, zero     # Reset row counter
    addi t1, t1, 1          # Increment column counter
    bne  t1, s2, loop       # Repeat for all columns
```

c) The memory contents of addresses `0x1000 0000` to `0x1000 0008` will be:

Address	**Value**
`0x1000 0000`	12
`0x1000 0001`	78
`0x1000 0002`	35
`0x1000 0003`	11
`0x1000 0004`	34
`0x1000 0005`	113
`0x1000 0006`	46
`0x1000 0007`	122
`0x1000 0008`	56

d) To initialize the memory of the simulator, we can use the `.byte` directive and store the elements of the matrix byte by byte. The initialization of the memory and registers `s0`, `s1`, `s2`, and `s3` is shown below:

```
.data
matrix:
    .byte 12, 78, 35, 11, 34, 113, 46, 122, 56, 24, 57, 33

.text
    la   s0, matrix
    addi s1, zero, 4
    addi s2, zero, 3
    add  s3, zero, zero
start:
    ...
```

e) i) The maximum number of bits needed to represent the result of adding two unsigned 8-bit numbers is nine. The largest value of `t2` and `s3` is `0xFF` (eight bits unsigned). Therefore, the sum `t2 + s3` requires nine bits.

ii) The instruction in question is `sb t2, 0(t5)` because it stores the least significant eight bits of register `t2` to memory and ignores the remaining bits. If the ignored bits are zeros, there is no issue. However, if the bit at position nine is set (because the sum written in `t5` is too large to be represented in eight bits), the result written to memory will be inaccurate.

iii) See the code below.

```
# ... original code
    add  t2, t2, s3          # Compute the sum
# new code, for detecting and handling overflow
    srli t6, t2, 8           # get rid of all but overflow bit
    beq  t6, zero, continue  # if the overflow bit is zero,
                             # continue with the original code
                             # otherwise
    addi t2, zero, 0xFF      # overwrite the sum with 0xFF
# back to the original code
continue:
    slt  t4, t2, t3
# ... the rest of the original code
```

Here is a description of the changes made to the program:

- After adding `s3` and `t2`, the program detects if the sum can be expressed with eight bits by performing a logical right shift by eight bits. If the resulting number after the logical right shift by eight bits is not zero, it indicates that the number from the addition of `s3` and `t2` needed more than eight bits.

- If `t2` uses only eight bits, `t6` will be zero and the code will branch to the label `continue`, to continue with the original program.
- Otherwise, the value in `t2` will be overwritten with the highest unsigned number representable in eight bits, before the original program continues.

[Exercise 12] Vector Comparator

Suppose that two vectors **A** and **B**, containing 16-bit signed values, are stored in memory starting from an address contained in register `s0`. The values for A[i] and B[i] are stored in the same 32-bit word in memory (see Figure 88 below). An element of vector **A** occupies the 16 most significant bits of the word. An element of vector **B** occupies the 16 least significant bits of the word. The end of each vector is indicated by all zeros. You may assume that both vectors contain the same number of elements and that no element except the last contains all zeros.

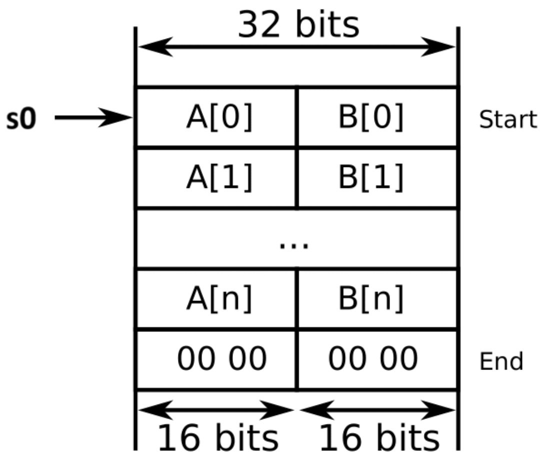


Figure 88: Memory layout for the vectors **A** and **B**.

Write a RISC-V program that compares the vectors **A** and **B** element-wise and counts the number of elements for which A[i] > B[i]. The total count should be written in register `s1`. You are allowed to access memory using only the `lw` instruction. Assume little-endian byte ordering.

[Solution 12] Vector Comparator

The RISC-V program is given below. The program compares the elements of vectors **A** and **B** element-wise and counts the number of elements for which A[i] > B[i].

```
.data
AB:
    .word 0x40006, 0x30001, 0x70006, 0xC000C, 0x5000C, 0x0
    # A = [4, 3, 7, 12, 5, 0], B = [6, 1, 6, 12, 12, 0]

.text
    la   s0, AB                # Load address of AB into s0
start:
    add  s1, zero, zero        # Initialize counter to 0
    add  t0, zero, s0          # Copy address of AB into t0
loop:
    lw   t2, 0(t0)             # Load A[i] and B[i] into t1
    beq  t2, zero, finish      # Exit loop if A[i] and B[i] are 0
    srai t3, t2, 16            # t3 = (Sign Ext) & A[i]
    slli t2, t2, 16            # t2 = B[i] & 0x0000
    srai t2, t2, 16            # t2 = (Sign Ext) & B[i]
    slt  t4, t2, t3            # if (t2 < t3) t4=1 else t4=0
    add  s1, s1, t4            # s1 = s1 + t4
    addi t0, t0, 4             # Move to the next elements
    j    loop
finish:
    nop                        # End of program
```

[Exercise 13] Array Comparator

a) Write a program in RISC-V that compares two vectors of 32-bit signed numbers. The program stores '`0`' in register `a0` if at least a pair of elements with the same index in both vectors differ in absolute value by more than 1000 (decimal). Otherwise, the program stores '`1`' in register `a0`.

You can assume that when the program starts, register `a0` contains the address of the first vector in memory, while register `a1` contains the address of the second vector and register `a2` contains the number of elements in each vector.

Note: to find the absolute value of the difference between two numbers, subtract the smaller from the greater.

b) Briefly discuss when can an overflow occur in your program.

[Solution 13] Array Comparator

a) The code of the solution is given below:

```
prog:
    li t5, 1              # initialize return value to 1
    li t6, 1000           # set threshold to 1000
loop:
    beq a2, zero, finish# if counter reach zero, go to finish
    lw  t2, 0(a0)         # load element from vector1
    lw  t3, 0(a1)         # load element from vector2
    slt t4, t3, t2        # if(t3<t2) t4=1
    bne t4, zero, next    # if(t4=1) goto next
    sub t3, t3, t2        # t3 = t3 - t2
    j   next2
next:
    sub t3, t2, t3        # t3 = t2 - t3
next2:
    slt  t4, t6, t3       # if(abs difference>threshold) t4=1
    bne  t4, zero, err    # if(t4!=0) goto err
    addi a0, a0, 4        # move to next element in vector1
    addi a1, a1, 4        # move to next element in vector2
    addi a2, a2, -1       # decrement loop counter
    j    loop
err:
    add t5, zero, zero    # set return value to 0
finish:
    mv a0, t5             # move return value to a0
    nop
```

b) An overflow can occur when a negative value is subtracted from a positive one and the difference is greater than the maximal positive value.

Example: `0000 0000` - `8000 0000`; `7FFF FFFF` - `FFFF FFFF`;

[Exercise 14] Understanding RISC-V Assembly Code

Consider the following RISC-V program:

```
    add  t0, a0, zero
    add  t1, a1, zero
    add  t2, a2, a2
    add  t2, t2, t2
    add  t3, t0, t2
loop:
    lw   t4, 0(t0)
    sw   t4, 0(t1)
    addi t0, t0, 4
    addi t1, t1, 4
    sltu t5, t0, t3
    bne  t5, zero, loop
```

Assume that initially registers `a0` and `a1` store addresses in memory and register `a2` stores an integer N. Registers `t0` to `t5` are used to store temporary values and `zero` is a register that always has the value zero.

a) Briefly comment each line of the code.

b) Describe in one sentence what this program does (its purpose).

c) Why did the instruction **`addi`** add 4 to registers `t0` and `t1`?

d) Why is the instruction **`sltu`** (set less than unsigned) used instead of the instruction **`slt`**?

[Solution 14] Understanding RISC-V (I)

a) Commented code:

```
    add t0, a0, zero     # t0 <- a0
    add t1, a1, zero     # t1 <- a1
    add t2, a2, a2       # t2 <- 2*a2
    add t2, t2, t2       # t2 <- 4*a2
    add t3, t0, t2       # t3 <- a0 + 4*a2
loop:
    lw   t4, 0(t0)       # load word from t0
    sw   t4, 0(t1)       # store word to t1
                           # (copy word from t0 to t1)
    addi t0, t0, 4       # t0 <- t0+4
                           # (points on the next word)
    addi t1, t1, 4       # t1 <- t1+4
                           # (points on the next word)
    sltu t5, t0, t3      # if (t0 < t3)
                           # t5 <- 1 else t5 <- 0

    bne  t5, zero, loop # if (t5 != 0) go to loop
```

b) This RISC-V program copies the contents of the memory area that ranges from address `a0` to address `a0+4N`, to the memory area ranging from `a1` to `a1+4N`.

c) `t0` and `t1` hold the addresses of the source and destination memory areas, respectively. These registers are referred to as *pointers* because their values point to memory addresses. Each word in memory is 32 bits (4 bytes) and must be aligned to an address that is a multiple of 4. The **`lw`** and **`sw`** instructions operate on 32-bit words. To access the next word in memory, the current address must be incremented by 4. This is achieved using the instructions `addi t0, t0, 4` and `addi t1, t1, 4`, which update pointers to refer to the subsequent word in the memory.

d) The instruction `sltu t5, t0, t3` compares the values of registers `t0` and `t3`, both of which are pointers. The system's address space ranges from `0x0000'0000` to `0xFFFF'FFFF`, meaning that pointers are always unsigned, as they refer to memory addresses. Therefore, the comparison between pointers is performed using the **`sltu`** instruction, which operates on unsigned values.

[Exercise 15] Understanding RISC-V

Consider the following RISC-V program:

```
begin:
    add  t0, a0, zero
    add  t1, a1, zero
    add  t2, a2, zero
    add  t3, zero, zero
    add  t4, zero, zero
outer:
    lbu  t5, 0(t0)
    bne  t3, a3, cont
inner:
    lbu  t6, 0(t1)
    sb   t6, 0(t2)
    addi t1, t1, 1
    addi t2, t2, 1
    addi t4, t4, 1
    bne  t6, zero, inner
    addi t2, t2, -1
    addi t4, t4, -1
cont:
    sb   t5, 0(t2)
    addi t0, t0, 1
    addi t2, t2, 1
    addi t3, t3, 1
    bne  t5, zero, outer
    addi t3, t3, -1
    add  a4, t3, t4
fin:
    nop
```

a) Describe in a sentence what this program does, knowing that when the program starts, registers `a0`, `a1`, `a2` and `a3` contain initial values, and at the end of execution the result is stored in `a4`. Registers `a0` and `a1` contain each the memory address of a string that ends with the NULL character, i.e. a zero byte (`0000’0000`).

b) Modify the program such that the **`lbu`** instruction is replaced by **`lw`** without altering the functionality (behaviour). String addresses (i.e. beginning of the string) are always multiples of four and the processor is little-endian. The **`sb`** instruction is available.

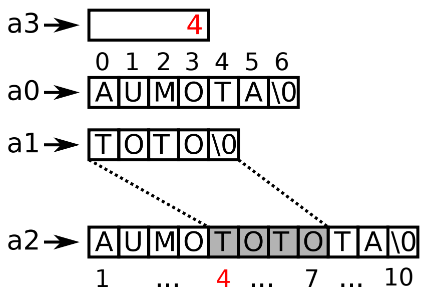
c) Should the program in the preceding point be modified if the processor were big-endian? If so, briefly describe the necessary modifications.

[Solution 15] Understanding RISC-V

a) The commented code:

```
begin:
    add  t0, a0, zero    # move address of 1st string to t0
    add  t1, a1, zero    # move address of 2nd string to t1
    add  t2, a2, zero    # move address of result string to t2
    add  t3, zero, zero  # initialize t3(index of 1st string)
    add  t4, zero, zero  # initialize t4(index of 2nd string)
outer:
    lbu  t5, 0(t0)       # load one character from 1st string
    bne  t3, a3, cont    # if index != a3, go to cont
inner:
    lbu  t6, 0(t1)       # load one character from 2nd string
    sb   t6, 0(t2)       # store character in result string
    addi t1, t1, 1       # increment address of 2nd string
    addi t2, t2, 1       # increment address of result string
    addi t4, t4, 1       # increment index of 2nd string
    bne  t6, zero, inner # if character != 0, go to inner
    # the following two instructions are necessary to avoid
    # counting the NULL character of the 2nd string
    # in the length of the result string
    addi t2, t2, -1      # decrement address of result string
    addi t4, t4, -1      # decrement index of 2nd string
cont:
    sb   t5, 0(t2)      # store character in result string
    addi t0, t0, 1      # increment address of 1st string
    addi t2, t2, 1      # increment address of result string
    addi t3, t3, 1      # increment index of 1st string
    bne  t5, zero, outer# if character != 0, go to outer
    addi t3, t3, -1     # decrement index of 1st string
    add  a4, t3, t4     # compute and return length of result string
fin:
    nop
```

At the end of execution, register a4 contains the length of a newly constructed string, and register a0 contains the memory address of this new string. This new string is created by taking the string pointed to by the initial value in a0 and inserting the string pointed by the initial value in a1 at the position specified by the initial value in a3, as shown in the figure below:



b) The following program does not use the **lbu** instruction. Instead, it uses the **lw** instruction keeping in mind that the processor is little-endian.

```
begin:
    add  t0, zero, a0      # move address of 1st string to t0
    add  t1, zero, a1      # move address of 2nd string to t1
    add  t2, zero, a2      # move address of result string to t2
    add  t3, zero, zero    # initialize t3(index of 1st string)
    add  t4, zero, zero    # initialize t4(index of 2nd string)
outer:
    andi s0, t3, 0x3       # Check if index is a multiple of 4
    bne  s0, zero, no_ld1  # if not, skip load
    lw   t5, 0(t0)         # load one word from 1st string
    addi t0, t0, 4         # increment address of 1st string
no_ld1:
    andi s0, t5, 0xff      # *** Get byte
    srli t5, t5, 8         # *** Prepare next byte
    bne  t3, a3, cont      # if index != a3, go to cont
inner:
    andi s1, t4, 0x3       # Check if index is a multiple of 4
    bne  s1, zero, no_ld2  # if not, skip load
    lw   t6, 0(t1)         # load one word from 2nd string
    addi t1, t1, 4         # increment address of 2nd string
no_ld2:
    andi s1, t6, 0xff      # +++ Get byte
    srli t6, t6, 8         # +++ Prepare next byte
    sb   s1, 0(t2)         # store character in result string
    addi t2, t2, 1         # increment address of result string
    addi t4, t4, 1         # increment index of 2nd string
    bne  s1, zero, inner   # if character != 0, go to inner
    addi t2, t2, -1        # decrement address of result string
    addi t4, t4, -1        # decrement index of 2nd string
cont:
    sb   s0, 0(t2)         # store character in result string
    addi t2, t2, 1          # increment address of result string
    addi t3, t3, 1          # increment index of 1st string
    bne  s0, zero, outer    # if character != 0, go to outer
    addi t3, t3, -1         # decrement index of 1st string
    add  a4, t3, t4         # compute and return length of result string

fin:
    nop
```

c) Yes. Instead of the sequence of instructions **andi** and **srli** (marked by $\star\star\star$ and +++) the following instruction sequence might be used instead (alternate solutions are possible):

```
    srl  s0, t5, 24       # prepare byte
    andi s0, s0, 0xff     # get byte
    sll  t5, t5, 8        # prepare next
                 ...
    srl  s1, t6, 24       # prepare byte
    andi s1, s1, 0xff     # get byte
    sll  t6, t6, 8        # prepare for next
```

[Exercise 16] Vector Difference

Consider a vector C such that $C[i] = |A[i] - B[i]|$ for all valid indices i. Each element of C is the absolute difference between the corresponding elements of vectors A and B. The vectors are stored in memory as 32-bit signed integers.

We want to write a RISC-V program to create vector C from vectors A and B. In the beginning, registers `a0`, `a1`, and `a2` point to the starting address of vectors A, B, and C, respectively. Moreover, register `a3` indicates the number of elements in these vectors.

a) Write the program described above, ignoring any possible overflows. The program should not use any subtraction or arithmetic negation instructions, but it can use logical operations and addition like **`xor`**.

b) Discuss the possible overflow cases. Which instructions in your program could potentially generate an overflow? Briefly describe (without necessarily modifying the program) how to detect such overflows.

[Solution 16] Vector Difference

a) Because neither arithmetic negation nor subtraction are available, bitwise negation and addition must be used instead to implement the requested program.

```
begin:
    add  t0, zero, a0    # t0 <- a0
    add  t1, zero, a1    # t1 <- a1
    add  t2, zero, a2    # t2 <- a2
    add  t3, zero, a3    # t3 <- a3
loop:
    beq  t3, zero, fin   # if t3 = 0 then goto finish
    lw   t4, 0(t0)       # t4 <- mem[t0]
    lw   t5, 0(t1)       # t5 <- mem[t1]
    not  t5, t5          # t5 <- not t5
    addi t5, t5, 1       # t5 <- t5 + 1 (***)
    add  t5, t4, t5      # t5 <- t4 + t5 (+++)
    slt  t4, t5, zero    # check the sign
    beq  t4, zero, skip  # if positive goto skip
    not  t5, t5          # t5 <- not t5
    addi t5, t5, 1       # t5 <- t5 + 1 (***)
skip:
    sw   t5, 0(t2)       # mem[t2] <- t5
    addi t0, t0, 4       # t0 <- t0 + 4
    addi t1, t1, 4       # t1 <- t1 + 4
    addi t2, t2, 4       # t2 <- t2 + 4
    addi t3, t3, -1      # t3 <- t3 - 1
    j    loop
fin:
    nop
```

b) The instructions marked by *** and +++ could potentially cause an overflow. We already showed how an overflow can be detected in assignment *Checking for Overflows*. Please check it for more details.

Note that the *** case is a specific case that can have a simpler custom overflow detection mechanism. In this case, an overflow occurs if register `t5` contains the smallest negative number ('1' at the MSB followed by '0's) before the **not** instruction is performed. Thus, we can detect a potential overflow by comparing `t5`'s value against the smallest negative number.

[Exercise 17] Understanding RISC-V

Study the following RISC-V program:

```
begin:
    mv   t0, a0
    mv   t1, a1
    mv   t2, zero
    addi a0, zero, -1

cont:
    lbu  t4, 0(t1)
outer:
    lbu  t3, 0(t0)
    beq  t3, zero, fin
    bne  t3, t4, skip
    mv   a0, t2
    mv   t5, t0
inner:
    addi t0, t0, 1
    addi t1, t1, 1
    lbu  t3, 0(t0)
    lbu  t4, 0(t1)
    beq  t4, zero, fin
    beq  t3, zero, fail
    beq  t3, t4, inner
    addi t0, t5, 1
    mv   t1, a1
    addi a0, zero, -1
    j    cont

skip:
    addi t0, t0, 1
    addi t2, t2, 1
    j    outer

fail:
    addi a0, zero, -1
fin:
    nop
```

a) Describe in a sentence what this program does, knowing that when the program starts, registers `a0` and `a1` each contain the memory address of a string ending with a

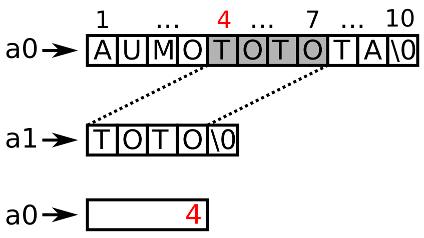
NULL character, i.e. a zero byte. At the end of execution, the result is stored in register `a0`.

b) Explain in a few words what the content of `t5` represents and the situation in which this value is needed.

c) Suppose that the **`lbu`** instruction is not available, but instead **`lw`** is used to read the memory. Write a code snippet that provides the same functionality as the instruction **`lbu`** `a1, 0(a0)`, knowing that `a0` contains the memory address of the byte to be loaded, and `a1` is the register where this byte must be stored at the end of the execution. Consider a big-endian processor. Recall that the **`lbu`** instruction does not perform a sign extension. Make sure that the addresses that are passed to **`lw`** are always aligned, i.e. multiples of 4.

[Solution 17] Understanding RISC-V

a) At the end of execution, register `a0` contains the position (index) of the first appearance of a substring in the input string. The initial value in `a1` points on the substring and the initial value in `a0` points on the input string. If the substring is not found, the negative position is stored (-1).



```
begin:
    mv   t0, a0         # t0 <- a0
    mv   t1, a1         # t1 <- a1
    mv   t2, zero       # t2 <- 0
    addi a0, zero, -1   # a0 <- -1

cont:
    lbu  t4, 0(t1)      # t4 <- mem[t1]

outer:
    lbu  t3, 0(t0)      # t3 <- mem[t0]
    beq  t3, zero, fin  # if t3 = 0 goto fin
    bne  t3, t4, skip   # if t3 <> t4 goto skip
    mv   a0, t2         # remember index
    mv   t5, t0         # wrong guess backup

inner:
    addi t0, t0, 1      # t0 <- t0 + 1
    addi t1, t1, 1      # t1 <- t1 + 1
    lbu  t3, 0(t0)      # t3 <- mem[t0]
    lbu  t4, 0(t1)      # t4 <- mem[t1]
    beq  t4, zero, fin  # return        found
    beq  t3, zero, fail # if t3 = 0 fail
    beq  t3, t4, inner  # continue inner loop
    addi t0, t5, 1      # recover wrong guess
    mv   t1, a1         # recover
    j    cont           # goto continue
```

```
skip:
    addi t0, t0, 1       # t0 <- t0 + 1
    addi t2, t2, 1       # t2 <- t2 + 1
    j    outer           # goto outer

fail:
    addi a0, zero, -1    # a0 <- -1

fin:
    nop
```

b) Register `t5` enables recovery from a "wrong guess". It holds the address where the matching started. If it does not succeed (the substring is not found), the matching is restarted from the next potential match at the address `t5` + 1.

c) The following code provides the functionality of the **`lbu`** instruction. Two possible solutions are presented. Both of them assume a big-endian processor.

```
xlbu:
    li   t1, 0xfffffffc # t1 <- 0xfffffffc
    and  t0, a0, t1     # t1 <- a0 & t1 (align)
    li   t1, 0xff000000 # t1 <- 0xff000000
    andi t2, a0, 0x3    # t2 <- a0 & 0x3 (get offset)
    lw   t3, 0(t0)      # t3 <- mem[t0]

loop:
    beq  t2, zero, done # if t2 = 0 done
    slli t3, t3, 8      # t3 <- t3 << 8 (next byte)
    addi t2, t2, -1     # t2 <- t2 - 1
    j    loop           # goto loop

done:
    and  t3, t3, t1     # t3 <- t3 & 0xff000000
    srli a1, t3, 24     # a1 <- t3 >> 24
    nop
```

Another possible solution that uses **`srli`**:

```
xlbu:
    li   t1, 0xfffffffc  # t1 <- 0xfffffffc
    and  t0, a0, t1      # t1 <- a0 & t1 (align)
    andi t2, a0, 0x3     # t2 <- a0 & 0x3 (get offset)
    lw   t3, 0(t0)       # t3 <- mem[t0]
```

```
loop:
    sltiu t1, t2, 3       # check t2 < 3
    beq   t1, zero, done # if t2 = 3 done
    srli  t3, t3, 8       # t3 <- t3 >> 8 (next byte)
    addi  t2, t2, 1       # t2 <- t2 + 1
    j     loop            # goto loop

done:
    andi a1, t3, 0xff     # a1 <- t3 & 0xff (get byte)
    nop
```

[Exercise 18] Binary Coded Decimal

Write a RISC-V program that converts from the BCD (Binary Coded Decimal) representation to the ordinary binary representation. In BCD representation, numbers are encoded directly from their decimal representation and each decimal digit is represented in binary using 4 bits. For example, the value 1992 in decimal is encoded in BCD using 16 bits as `0001'1001'1001'0010`, while its ordinary binary representation is `0000'0111'1100'1000`. Certain binary values such as `1111'1010'1100'1110` cannot represent BCD values; this happens whenever a group of 4 bits represents a value greater than 9.

The unsigned 32-bit value to be converted is located in register `a0` and the binary result at the end of the conversion must be saved in register `a0`. If the value contained in register `a0` cannot represent a BCD value, the `a0` register must contain -1 at the end of the execution.

a) Write the conversion function ignoring any possible overflows. You can assume the availability of multiplication instructions on 32-bit operands:

```
mul  rd, rs, rt  # rd = rs * rt
muli rd, rs, imm # rd = rs * imm
```

b) The **`mul`** and **`muli`** instructions do not exist in RISC-V. Modify the program such that it no longer makes use of these multiplication instructions.

c) Discuss the possible overflow cases. Modify the program, if necessary, to remedy the situation.

[Solution 18] Binary Coded Decimal

a) Note that the conversion can be decomposed into a series of multiplications by 10 followed by an addition. For example, a 4-digit BCD number like $abcd_{10}$ can be represented as:

$$abcd_{10} = a * 10^3 + b * 10^2 + c * 10^1 + d * 10^0 = ((a * 10 + b) * 10 + c) * 10 + d$$

The 32-bit value in register `a0` has eight 4-bit parts each representing a decimal digit. We use the following code to extract the value of each part and obtain the binary value using the conversion mechanism shown above.

```
begin:
    add  t0, a0, zero      # t0 <- a0
    add  t6, zero, zero    # t6 <- 0
    beq  t0, zero, fin     # terminate if input is zero
    addi t1, zero, 8       # t1 <- 8 (number of digits)
loop:
    srli  t2, t0, 28       # t2 <- the current leftmost digit
    sltiu t3, t2, 10       # check if t2 < 10
    bne   t3, zero, skip   # skip if the digit is valid
    addi  t6, zero, -1     # t6 <- -1, error
    j     fin              # finish
skip:
    muli  t6, t6, 10       # t6 <- t6 * 10
    add   t6, t6, t2       # t6 <- t6 + t2
    slli  t0, t0, 4        # bring the next digit to bits 31-28
    addi  t1, t1, -1       # t1 <- t1 - 1, decrement the counter
    bne   t1, zero, loop   # loop until t1 becomes zero
fin:
    addi a0, t6, 0         # a0 <- t6, bring the result to a0
    nop
```

b) Multiplication by 10 can be replaced by shifting and adding. The idea is X * 10 = X*8 + X*2 as shown in the following RISC-V code:

```
    ...
skip:
    slli  t3, t6, 3        # mul t6 by 8
    slli  t4, t6, 1        # mul t6 by 2
    add   t6, t3, t4       # t6 <- 10 * t6
    ...
```

c) It is not possible to have an overflow because the largest number in eight digit BCD representation with 32 bits is smaller than the largest 32-bit unsigned number (or even the largest number in two's complement representation).

[Exercise 19] Understanding RISC-V

Consider the following RISC-V program.

```
start:
    addi t1, a1, -1
outer:
    beq  t1, zero, fin
    add  t0, a0, zero
    add  t2, t1, zero
inner:
    beq  t2, zero, cont
    lw   t3, 0(t0)
    lw   t4, 4(t0)
    sltu t5, t4, t3
    beq  t5, zero, skip
    sw   t3, 4(t0)
    sw   t4, 0(t0)
skip:
    addi t0, t0, 4
    addi t2, t2, -1
    j    inner

cont:
    addi t1, t1, -1
    j    outer
fin:
    nop
```

a) Describe in a sentence what the program does, knowing that when the program starts, registers `a0` and `a1` contain the address of an array in memory and the number of its elements, respectively.

b) Explain in a few words the purpose of the two **`lw`** and the two **`sw`** instructions in this context.

c) What is the type of the processed values ? Justify your answer.

d) Explain in a few words what the contents of the three registers `t0`, `t1` and `t2` represent. Modify the program in order to correctly handle the case where `a1 = 0`.

e) The original program does not indicate any result at the end of execution. We want the program to store the following values in register `a0`:

- `0` in the case where no data in the memory has been changed.
- `-1` in the case where some data has been changed in the memory.

Modify the program such that it stores the correct result in `a0` at the end of execution.

[Solution 19] Understanding RISC-V

a) This program sorts the elements of a vector in an ascending order.

b) The two `lw` and two `sw` instructions are used to swap the locations of two elements in memory.

c) The program processes unsigned integers as the comparison is done using the `sltu` (set less than unsigned) instruction.

d) Register `t0` is used as a pointer on the elements of the vector. Registers `t1` and `t2` are used as counters to control the two loops. `t1` contains the remaining iterations of the `inner` loop, in other words, the number of iterations of the `outer` loop. Register `t2` contains the total number of iterations of the `inner` loop.

When the initial value of `a1` is 0, the program will not behave correctly (there is a risk of having the program generate wrong addresses). The following case should thus be avoided:

```
start:
    beq  a1, zero, fin
    addi t1, a1, -1
    ...
```

e) We present here possible changes.

```
start:
    addi t6, zero, 0     # t6 <- 0
    addi t1, a1, -1

outer:
    beq   t1, zero, fin

    ...                          # no changes

    sw    t3, 4(t0)
    sw    t4, 0(t0)
    addi  t6, zero, -1   # t6 <- -1

skip:
    addi  t0, t0, 4
    addi  t2, t2, -1
    j     inner

```

```
cont:
    addi t1, t1, -1
    j    outer

fin:
    mv a0, t6              # a0 <- t6
    nop
```

[Exercise 20] Understanding RISC-V

Analyze the following RISC-V program:

```
program:
    slli t0, a1, 2
    add  t0, a0, t0
    addi t0, t0, -4
loop:
    slt  t1, a0, t0
    beq  t1, zero, fin
    lw   t2, 0(a0)
    lw   t3, 0(t0)
    sw   t2, 0(t0)
    sw   t3, 0(a0)
    addi t0, t0, -4
    addi a0, a0, 4
    j    loop

fin:
    nop
```

When the program starts, `a0` contains the memory address of a vector of 32-bit numbers and `a1` contains an integer.

a) Describe in a sentence what the program does.

b) Must the numbers contained in the vector be either signed or unsigned? Or is it possible to have both signed and unsigned numbers in the vector ? Briefly explain your answer.

c) We would like to change this program so that it can process (handle) bytes. To this effect we need another program that, given four bytes in `a0`, stores the same four bytes in the reverse order in `a0`: byte B_3 (bits 32-24) is swapped with byte B_0 and B_2 with B_1. Write such a program respecting ordinary RISC-V conventions.

[Solution 20] Understanding RISC-V

a) This program inverts the order of elements in a vector in memory.

> `a0` points (starting from the beginning) on the vector's element that must be swapped next.
>
> `t0` points (starting from the end) on the vector's (other) element that must be swapped with the first.

```
program:
    slli  t0, a1, 2      # t0 = Nb elements * sizeof(elem)
    add   t0, a0, t0     # t0 points to the end of the vector
    addi  t0, t0, -4     # t0 points to the last element
loop:
    slt   t1, a0, t0
    beq   t1, zero, end # if( a0 >= t0) go to end
    lw    t2, 0(a0)      # Store in t2 mem(a0)
    lw    t3, 0(t0)      # Store in t3 mem(t0)
    sw    t2, 0(t0)      # Copy t2 to mem(t0)
    sw    t3, 0(a0)      # Copy t3 to mem(a0)
    addi  t0, t0, -4     # Update the end pointer
    addi  a0, a0, 4      # Update the start pointer
    j     loop           # Continue the loop

end:
    nop
```

b) The RISC-V program does not modify any of the vector's elements in a way that depends on their numerical value or interpretation; it only moves them in memory. There is thus no restriction on their type, they can be signed, unsigned or in any other representation.

c) The following code inverts the order of the bytes of a 32-bit input.

```
Inv_byte:
    li   t2, 0           # t2 will store the result
    addi t0, zero, 0xFF # Mask byte 7-0 in t0
    and  t1, a0, t0      # Byte 7-0 in t1
    slli t1, t1, 24      # Shift byte 7-0 to position 31-24
    add  t2, zero, t1    # Byte 31-24 of the result is ready

    slli t0, t0, 8      # Mask byte 15-8 in t0
    and  t1, a0, t0     # Byte 15-8 in t1
    slli t1, t1, 8      # Shift byte 15-8 to position 23-16
    or   t2, t2, t1     # Byte 23-16 of the result is ready

    slli t0, t0, 8      # Mask byte 23-16 in t0
    and  t1, a0, t0     # Byte 23-16 in t1
    srli t1, t1, 8      # Shift byte 23-16 to position 15-8
    or   t2, t2, t1     # Byte 15-8 of the result is ready

    slli t0, t0, 8      # Mask byte 31-24 in t0
    and  t1, a0, t0     # Byte 31-24 in t1
    srli t1, t1, 24     # Shift byte 31-24 to position 7-0
    or   t2, t2, t1     # Byte 7-0 of the result is ready

fin:
    mv a0, t2
    nop
```

[Exercise 21] CORDIC

The CORDIC algorithm (COordinate Rotation DIgital Computer) allows calculating the length of a vector V(X,Y) using the following iterative relationships:

$$x_{i+1} = x_i - d_i \cdot y_i \cdot 2^{-i} \tag{1}$$
$$y_{i+1} = y_i + d_i \cdot x_i \cdot 2^{-i} \tag{2}$$
$$z_{i+1} = z_i - d_i \cdot tan^{-1}(2^{-i}) \tag{3}$$

Where $d_i = \begin{cases} -1 & \text{if } y_i > 0 \\ 1 & otherwise \end{cases}$

x, y, and z are initialized as follows: $x_0 = X$, $y_0 = Y$, and $z_0 = 0$.

After n iterations (n large enough), x_n, y_n, and z_n can be expressed as follows:

$$x_n \approx A_n \cdot \sqrt{X^2 + Y^2} \tag{4}$$
$$y_n \approx 0 \tag{5}$$
$$z_n \approx tan^{-1}\left(\frac{Y}{X}\right) \tag{6}$$

If we neglect constant A_n, these iterative formulas allow calculating the length of a vector as well as its angle with the x axis.

a) Suppose that initially $x_0 = -24$, $y_0 = 32$, and $z_0 = 0$. Calculate by hand the value of x_2 and y_2. Do not calculate z_2.

b) Using the aforementioned iterative formulas, write a RISC-V program that computes in 31 iterations the length and angle (with the x axis) of a vector whose initial x and y coordinates are available in registers `a0` and `a1`. These values are 32-bit signed integer values encoded in two's complement. Register `a2` contains the memory address of an array of 32-bit words which contains the already computed $tan^{-1}(2^{-i})$ coefficients (you do not need to compute them). Element 0 in the array is the coefficient of iteration 0, element 1 in the array is the coefficient of iteration 1 and so on. Ignore possible overflows. At the end of execution, the length and angle should be stored in `a0` and `a1` respectively.

[Solution 21] CORDIC

a) We can find x_2 and y_2 using the iterative formulas. Initially, we have:

$$\begin{aligned} x_0 &= -24 \\ y_0 &= 32 \\ z_0 &= 0 \end{aligned}$$

Iteration 1 will be:

$$\begin{aligned} &y_0 > 0 \Rightarrow d_0 = -1 \\ &x_1 = -24 - (-1) \cdot 32 \cdot 2^0 = -24 + 32 = 8 \\ &y_1 = 32 + (-1) \cdot (-24) \cdot 2^0 = 32 + 24 = 56 \end{aligned}$$

Then, iteration 2:

$$\begin{aligned} &y_1 > 0 \Rightarrow d_1 = -1 \\ &x_2 = 8 - (-1) \cdot 56 \cdot 2^{-1} = 8 + 28 = 36 \\ &y_2 = 56 + (-1) \cdot 8 \cdot 2^{-1} = 56 - 4 = 52 \end{aligned}$$

b) The RISC-V program that calculates the length and angle of a vector using the CORDIC formulas is given below:

```
cordic:
    add   t0, zero, a0      # t0 <- x0
    add   t1, zero, a1      # t1 <- y0
    add   t2, zero, zero    # t2 <- z0 (init to 0)
    add   t3, zero, a2      # t3 <- a2 (coef table)
    addi  t4, zero, 1       # t4 <- 1 (index)
loop:
    sltiu t5, t4, 32        # if t4 = 32
    beq   t5, zero, end     #  then goto end
    slt   t5, zero, t1      # if  0 < yi , t5=1 else t5=0
    beq   t5, zero, dpos    #  then go to dpos, else
dneg:
    sra   t5, t1, t4        # t5 <- yi*2**(-i)
    sra   t6, t0, t4        # t6 <- xi*2**(-i)
    add   t0, t0, t5        # compute next xi
    sub   t1, t1, t6        # compute next yi
    lw    t5, 0(t3)         # t5 <- coeff[i]
    add   t2, t2, t5        # compute next zi
    j      cont              # goto cont
dpos:
    sra    t5, t1, t4        # t5 <- yi*2**(-i)
    sra    t6, t0, t4        # t6 <- xi*2**(-i)
    sub    t0, t0, t5        # compute next xi
    add    t1, t1, t6        # compute next yi
    lw     t5, 0(t3)         # t5 <- coeff[i]
    sub    t2, t2, t5        # compute next zi
cont:
    addi   t3, t3, 4          # t3 <- next coeff addr
    addi   t4, t4, 1          # t4 <- next index
    j      loop               # goto loop
end:
    add    a0, zero, t0      # a0 <- xn
    add    a1, zero, t2      # a1 <- zn
    nop
```

[Exercise 22] Division Rest

We want to write a program that calculates the rest of an integer division of a 32-bit unsigned integer by 15 without performing the division per se. We can use the following property: the remainder of the division of a number by 15 is equal to the sum of the digits of its hexadecimal representation, repeatedly calculated until a single hexadecimal digit is obtained.

Thus, you will start by summing all the digits (in hexadecimal representation) of the given number. If the hexadecimal representation of this sum has more than one digit, you will apply the same procedure again to the result: sum the digits of the sum. You will repeat this process until you obtain a value that is represented by a single hexadecimal digit. This value is the rest of the integer division by 15, except if this final single-digit value is equal to 15 (which is represented as '0xF' in hexadecimal and corresponds to the decimal value 15), in which case the rest is 0 (and not 15).

For example, let N = `0x32041EF2` = 839130866. We calculate the rest of its division by 15 as follows:

- We first sum the 8 digits of the hexadecimal representation of N
  `3 + 2 + 0 + 4 + 1 + E + F + 2 = 0x29 (41)`
- Since the obtained sum `0x29` has two digits, we compute the sum of its digits
  `2 + 9 = 0xB (11)`
  which has only one digit

This value `0xB` is the result of the integer division of N by 15, as it has only one digit and is different from 15 (no need to replace it with 0). It can easily be verified that $839'130'866 = 55'942'057 \cdot 15 + 11$.

a) Write a program that calculates and stores in `a0` the rest of the division of a 32-bit number N by 15 at the end of execution. The initial value of N is supplied through the `a0` register. The value of this register does not necessarily need to be preserved during the program's execution. Implement the iterative process described above to arrive at the single-digit hexadecimal sum.

[Solution 22] Division Rest

a) We make the following choices to calculate the rest of the division by 15:

- To isolate the 4 bits necessary to compute the partial sums we successively use the **sll** and **srl** instructions. Alternately we could also use a mask.
- To repeatedly process the sum of digits until a single hexadecimal digit is obtained, we use the "jump" instruction (**j**) to return to the summation logic if the result is still greater than a single hexadecimal digit.

A possible solution is given below:

```
start:
    addi t0, zero, 28
    addi t1, zero, 28
    add  t2, zero, zero  # intermediary sum
    addi t3, zero, 8     # 32-bit hex digits
sum:
    beq  t3, zero, rec   # end partial sums test
    sll  t4, a0, t0
    srl  t4, t4, t1
    add  t2, t2, t4      # partial sum
    addi t0, t0, -4      # shift left value
    addi t3, t3, -1      # loop counter update
    j    sum
rec:
    add  t5, t2, zero
    li   t6, 0xFFF0
    and  t5, t5, t6
    beq  t5, zero, fin   # check if sum is single hex digit
    add  a0, t2, zero    # update a0 with the sum for next iteration
    j    start           # jump back to the beginning for the next summat
fin:
    addi t0, zero, 15    # Edge case - check if final result is 15
    bne  t2, t0, skip
    addi t2, zero, 0
skip:
    add  a0, zero, t2
    nop
```

[Exercise 23] Run-Length Encoding

Write a RISC-V program that performs Run-Length Encoding. This encoding is efficient for representing a string containing characters that are often repeated consecutively. Each character, repeated or not, is represented using two bytes, the first being the character itself, and the second the number of times this character is repeated consecutively. For example, the string `'aaaabccc'` will be encoded into the RLE sequence `'a',4,'b',1,'c',3`.

Some specifications of the RISC-V program that performs the RLE encoding are given below:

- Characters are in ASCII format (an unsigned byte whose value ranges from 0 to 127).
- When the program starts, register `a0` contains the memory address of a string that ends with a null character, i.e., the last byte of the string is zero.
- The result of the program is a list of bytes representing the encoded string. The memory address where the program must write the encoded string is given in register `a1`. The encoded list of bytes also ends with a null byte.

Figure 89 below shows the input string and the output encoded list of bytes:

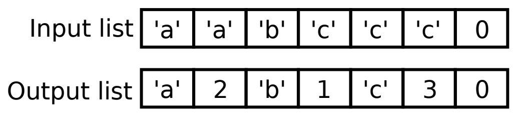


Figure 89: Example of RLE encoding

a) Write a RISC-V program that performs RLE encoding. Assume that a character is never repeated consecutively more than 255 times.

b) Give a simple way of modifying the encoding in the event of a character being repeated more than 255 times. Modify the code of the program accordingly.

c) Modify the code to implement a more efficient encoding as follows:

- If a character has a single occurrence and is not repeated, the `'1'` is omitted.

- If a character is repeated (consecutive occurrences), the encoding is essentially the same as before. However, in order to determine whether an element (byte) represents a character or a number of repetitions, the most significant bit (MSB) is set to `'1'` in the latter case (i.e., 128 is added to the value representing the number of repetitions).

The example string `'aaaabccc'` is now encoded into the sequence `'a',132` (4 + 128), `'b','c',131`. Note that since the MSB is now used to distinguish between characters and repetitions, the maximum number of repetitions that can be handled is now 127 instead of 255 (we lose the MSB).

[Solution 23] Run-Length Encoding

a) The code of the RISC-V program that performs RLE encoding is given below with appropriate comments. `t0` contains the last read character, `t1` contains the repeated character, and `t2` the number of repetitions.

```
rle:  lbu  t0, 0(a0)      # Initialisations
      beq  t0, zero, fin
      add  t1, zero, t0
      add  t2, zero, zero

loop: bne  t0, t1, diff   # Check if char is different
      addi t2, t2, 1      # If similar, increment counter
      j    meme

# Write the repeated char and the number of repetitions
# on the output list. Update the pointer of the output list
diff: sb   t1, 0(a1)
      sb   t2, 1(a1)
      addi a1, a1, 2

# If the input list is completed, terminate the program
      beq  t0, zero, fin

# Update the repeated char and the number of repetitions
      add  t1, t0, zero
      addi t2, zero, 1

meme: addi a0, a0, 1      # Go to next char
      lbu  t0, 0(a0)      # Read the next char
      j    loop

fin:  sb   zero, 0(a1)    # Ends the output list with 0
      nop
```

b) The simplest way of modifying the code is to interpret a character that is repeated more than 255 times as a new character. For example, if a character repeats 256 times, it will be encoded as `'a', 255, 'a', 1`. This only requires adding a simple control sequence after the `loop` label to implement the desired new functionality. The modification is thus:

```
loop: bne  t0, t1, diff
      addi t3, zero, 255   # Verify the 255 limit
      beq  t2, t3, diff    # If the limit is reached
```

```
                                 # consider the next repeated
                                 # char as different

       addi t2,t2,1
       j    meme
```

Note that in order to increase the performance of the code, we can move instruction `addi t3, zero, 255` within the initialization part of the program.

c) We only need to slightly modify the part that writes the encoded sequence: i.e., the first two lines after the `diff` label. If there are no repetitions, we simply increment the pointer on the output (encoded) sequence of bytes and continue, otherwise we add 128 to the number of repetitions and write it to the encoded string.

The modified code is given below.

```
diff:
    sb   t1, 0(a1)
    addi t4, zero, 1
    beq  t2, t4, ignore  # Repeated char ?
    addi t2, t2, 128     # yes, add 128
    sb   t2, 1(a1)       # write on the output encoded string
    addi a1, a1, 1       # increment pointer

ignore:
    addi a1, a1, 1       # increment pointer
    beq  t0, zero, fin
        ...
```

[Exercise 24] `srli` or `srai`, that is the question!

Consider the following RISC-V program:

```
start:
      slli t0, a0, 16
      srli t0, t0, 16
      slli t1, a1, 16
      srli t1, t1, 16
      add  a0, zero, zero
loop:
      beq  t1, zero, end
      andi t2, t1, 1
      beq  t2, zero, cont
      add  a0, a0, t0
cont:
      slli t0, t0, 1
      srli t1, t1, 1
      j    loop
end:
      nop
```

When the program starts, two initial values are present in registers `a0` and `a1`. At the end of execution, the result is stored in register `a0`.

a) Describe briefly what the program does.

b) What is the maximum effective size (in bits) of the initial values in `a0` and `a1`, and the final value in `a0`? Is there an overflow risk in the program's execution? Explain.

c) Are the initial values in `a0` and `a1` treated as signed or unsigned numbers by the program? Explain.

d) If we replaced only the first **`srli`** instruction by **`srai`**, would this program still perform a useful but different action? If so, describe the new action and the result. If not, precisely explain why modifying the program in this manner makes no sense.

e) What if we replaced both the first and second **`srli`** instructions by **`srai`**?

f) What if we replaced all three **`srli`** instructions by **`srai`**?

[Solution 24] `srli` or `srai`, that is the question!

a) The program performs the multiplication of two 16-bit unsigned numbers.

The product of the two numbers is calculated using the shift and add algorithm. The solution with comments is given below:

```
# Initialize the inputs and the output.
# Set the upper 16 bits of a0 and a1 to zero.
start:
    slli t0, a0, 16
    srli t0, t0, 16
    slli t1, a1, 16
    srli t1, t1, 16

# Initialize the product
    add  a0, zero, zero

loop:
    beq t1, zero, end    # If t1 = zero, we end the loop

# The least significant bit of t1 determines if t0 must be added to a0.
    andi t2, t1, 1        # t2 <- the least significant bit of t1
    beq  t2, zero, skip  # if t2 = zero, skip addition
    add  a0, a0, t0       # otherwise, add t0 to a0

# Prepare the two operands for the next iteration by shifting them by 1 b
# The first operand (t0) is shifted to the left.
# The second operand (t1) is shifted to the right.
skip:
    slli t0, t0, 1      # Shift t0 to the left
    srli t1, t1, 1      # Shift t1 to the right
    j    loop           # jump to loop

end:
    nop
```

b) As the upper 16 bits of both operands are initialized to zero at the beginning of the program, their maximum effective size is 16 bits. The result is 32-bit wide, therefore, there can't be any overflow because multiplying two 16-bit numbers yields a value that can be represented using 32 bits: $(2^{16}-1)(2^{16}-1) = 2^{32} - 2^{17} + 1 < 2^{32} - 1$

c) The 16 most significant bits of the operands are set to zero by the `srli` instructions and no sign extension is performed. These operands are thus interpreted as unsigned.

d) If we replace the first `srli` instruction by `srai`, we interpret the first operand as signed, while the other remains unsigned. The question is whether this has any effect on the correct execution of the program. There are two cases to consider:

- The first operand is positive. This case is not different than the original program. Thus using `srai` has no effect and the program stores a valid result in `a0` at the end of execution.
- The first operand is negative. Consider $-O_1$ represents the value of the first operand. Treating such a value as an unsigned number gives us $2^{32} - O_1$. Therefore, multiplying this value with a positive number like O_2 yields:

  $$(2^{32} - O_1) \cdot O_2 = 2^{32} \cdot O_2 - O_1 \cdot O_2 = 0 - O_1 \cdot O_2 = (-O_1) \cdot O_2$$

  The result is correct if we interpret it as a signed value. Note that $2^{32} \cdot O_2$ cannot be represented with 32 bits and becomes zero.

Thus the program still stores a valid result in `a0` if we replace `srli` with `srai` as long as the result is interpreted as signed.

e) If we replace both the first and second `srli` instructions by `srai`, the program still stores a valid result in `a0`. There are four cases to consider:

- Both operands are positive. This case is the same as the original program.
- The first operand is negative and the second operand is positive. This case is similar to the one discussed in the previous point.
- The first operand is positive and the second operand is negative. This case is similar to the previous point. Consider O_1 and $-O_2$ represent the values of the first and second operands, respectively. We have:

  $$O_1 \cdot (2^{32} - O_2) = 2^{32} \cdot O_1 - O_1 \cdot O_2 = -O_1 \cdot O_2 = O_1 \cdot (-O_2)$$

  Therefore, the result is correct if it is interpreted as signed.
- Both operands are negative. Consider $-O_1$ and $-O_2$ as the value of the first and second operands, respectively. We have:

$$\begin{aligned}
(2^{32}-O_1)(2^{32}-O_2) &= (2^{64}-2^{32}\cdot O_1-2^{32}\cdot O_2+O_1\cdot O_2) \\
&= 2^{32}(2^{32}-O_1-O_2)+O_1\cdot O_2 \\
&= 0+O_1\cdot O_2 \\
&= O_1\cdot O_2
\end{aligned}$$

This is the expected and correct value.

Hence, changing the mentioned **`srli`** instructions by **`srai`** does not alter the behaviour of the program as long as the result is interpreted as signed.

f) The program is not correct if all three **`srli`** instructions are replaced by **`srai`**. Indeed, if register `t1` contains a negative value, shifting it using **`srai`** will extend the sign bit. Repeatedly shifting this negative value to the right using **`srai`** will keep the most significant bit as 1, and the value in `t1` will never reach zero. Consequently, the program remains in an infinite loop.

[Exercise 25] Floating Point Larger-Than

Floating point formats in computer science are similar to the common scientific notation, with a mantissa and an exponent. For example $-9.062 \cdot 10^2$ or $3.87 \cdot 10^{-1}$. Usually we want to represent numbers with a mantissa bound between 10^0 and 10^1 in which case it is considered as normalized. If a given number is not normalized it is multiplied by an appropriate power of 10 to adapt the exponent so that the mantissa is within normalized bounds. For example :

$$\begin{aligned} -0.0041 \cdot 10^4 &= \quad -4.1 \cdot 10^1 \\ 9760.1 \cdot 10^{-8} &= 9.7601 \cdot 10^{-5} \end{aligned}$$

Now consider the following 32-bit binary floating point format:

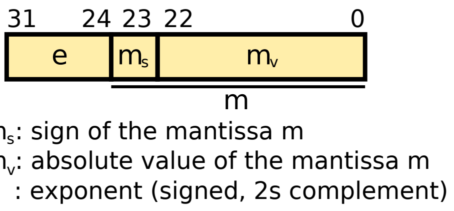

31
24
23
22
0
e
m_s
m_v
m
m_s: sign of the mantissa m
m_v: absolute value of the mantissa m
e : exponent (signed, 2s complement)

Figure 90: 32-bit floating point format

Bits 31 to 24 represent the exponent `e` in Two's Complement, while bits 23 to 0 represent the mantissa `m` in sign and magnitude, i.e. bit 23 represents the sign and bits 22 to 0 the absolute value. A value in floating point format can thus be represented as:

$$(-1)^{m_s} \cdot m_v \cdot 2^e$$

The point in the mantissa is implied after the most significant bit MSB, and thus bit 22 has a weight of 1, bit 21 a weight of $\frac{1}{2}$, bit 20 a weight of $\frac{1}{4}$ and so on. Moreover, numbers are normalized, i.e. $2^0 \leq m_v \leq 2^1$, meaning that the mantissa is aligned in a way such that bit 22 is always `'1'` and the exponent is adjusted accordingly. Of course in a real application this would be implied because it is useless, except to represent the value zero whose existence we will ignore in what follows.

As an example we represent some values in normalized floating point notation:

$$\begin{aligned} 14 &= \quad 1.75 \cdot 2^3 &= \langle 0000'0011'0111'0000'0000'0000'0000'0000 \rangle \\ -0.312 &= -1.25 \cdot 2^{-2} &= \langle 1111'1110'1101'0000'0000'0000'0000'0000 \rangle \end{aligned}$$

a) Write a RISC-V program that receives two numbers in the aforementioned floating point format in registers `a0` and `a1` when the program starts. Your program should compare the two numbers and store 1 in register `a0` at the end of execution if the first number is strictly greater than the second one. If not, the program should store 0 in register `a0`.

b) Write a RISC-V program that converts a non-normalized number into a normalized one, according to the floating point format presented earlier. When the program starts, register `a0` contains the non-normalized number. The normalized number should be stored in register `a0` at the end of execution. The zero number must be ignored as well as potential overflows of the floating point format.

[Solution 25] Floating Point Larger-Than

a) To perform the required comparison, we can use the algorithm illustrated below:

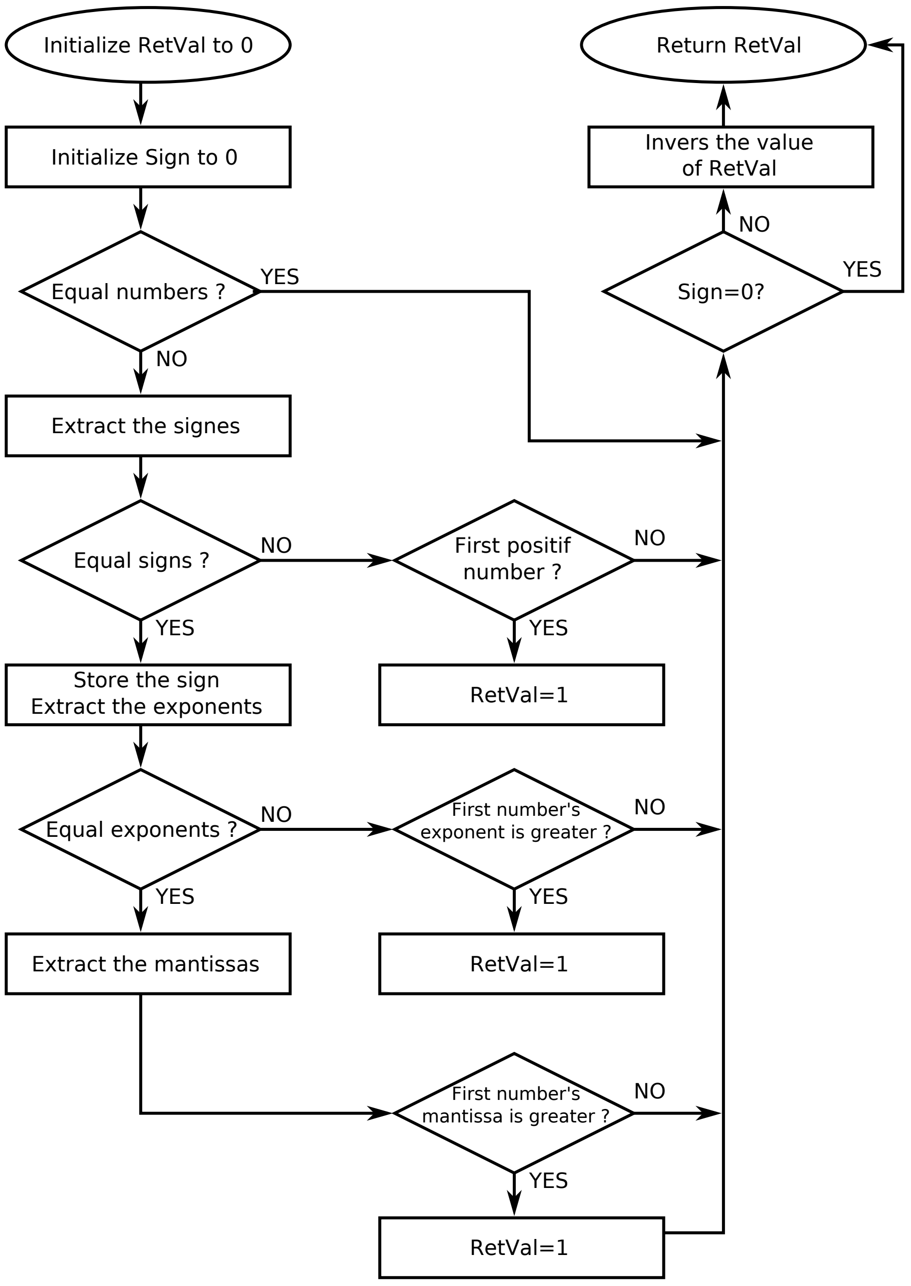


Figure 91: Algorithm for the comparison

The corresponding RISC-V program is given below with comments:

```
compare:
    add  t4, zero, zero         # Extract RetVal
    add  t0, zero, zero         # Initialize sign
    beq  a0, a1,   smaller_eq   # Equal numbers?
sign_check:
    slli t1, a0,   8            # Extract the signs
    srli t1, t1,   31
    slli t2, a1,   8
    srli t2, t2,   31
    beq  t1, t2,   set_sign     # Same signs ?
    beq  t1, zero, greater      # 1st pos number?
    j    smaller_eq
set_sign:
    add  t0, zero, t1           # Store the sign
exp_check:
    srai t1, a0,   24           # Extract the exponent
    srai t2, a1,   24
    beq  t1, t2,   mantissa_chk # same exponent?
    slt  t3, t1,   t2
    beq  t3, zero, greater
    j    smaller_eq
mantissa_chk:
    slli t1, a0,   9            # Extract the mantissa
    slli t2, a1,   9
    sltu t3, t2,   t1
    beq  t3, zero, smaller_eq
greater:
    addi t4, zero, 1            # RetVal=1
smaller_eq:
    xor  t4, t4, t0             # Adjust RetVal
    mv   a0, t4                 # Store result in a0
    nop
```

b) The idea is to shift the mantissa to the left until the MSB is `'1'` and adjust the exponent accordingly. The RISC-V code is given below:

```
normalize:
    slli t0, a0,  9          # Extract the mantissa
    srai t1, a0,  24         # Extract the exponent
test_nrm:
    slt  t2, zero,t0         # If MSB is 1
    beq  t2, zero,reformat   # end loop
    slli t0, t0,  1          # Left shift mantissa
    addi t1, t1,  -1         # Decrement exponent
    j    test_nrm            # Stay in the loop
reformat:
    srli t2, a0,  23
    slli t2, t2,  31         # Sign of MSB
    srli t0, t0,  1          # Shift mantissa by 1 bit
    add  t2, t2,  t0         # Mantissa and sign
    srli t2, t2,  8          # Shift mantissa and sign
    slli t1, t1,  24         # Shift exponent
    add  a0, t2,  t1         # The format is correct
    nop
```

[Exercise 26] Understanding RISC-V

Consider the following RISC-V program:

```
start: add  t0,   zero, a0
       add  t1,   zero, a1
       add  t2,   zero, a2
       add  a0,   zero, zero
loop:  beq  t2,   zero, fin
       lw   t3,   0(t0)
       lw   t4,   0(t1)
       addi t5,   zero, 32
       slt  t6,   t2,   t5
       beq  t6,   zero, cont1
       add  t5,   zero, t2
cont1: xor  t6,   t3,   t4
cont2: andi t3,   t6,   1
       add  a0,   a0,   t3
       srli t6,   t6,   1
       addi t2,   t2,   -1
       addi t5,   t5,   -1
       bne  zero, t5,   cont2
       addi t0,   t0,   4
       addi t1,   t1,   4
       j    loop
fin:   nop
```

When the program starts, initial values are present in registers a0, a1, and a2. Registers a0 and a1 contain each the start address of a list of bytes, and a2 contains an unsigned number. At the end of execution, the result is stored in register a0.

a) Explain in a sentence what the above RISC-V code does.

b) Is the given code written for a little-endian or big-endian processor? Clearly state the reason.

c) The above code makes use of the **xor** instruction. Suppose this instruction does not exist. Write a sequence of instructions that replace the **xor** instruction.

[Solution 26] Understanding RISC-V

a) The supplied RISC-V code counts the number of bits that are different in the two input lists. The beginning of each list is given in registers a0 and a1, while a2 contains the number of bits to compare. The result is stored in register a0 at the end of execution. A commented version of the RISC-V code is given below:

```
# Initializations
start: add  t0, zero, a0   # Addresses of the lists
       add  t1, zero, a1   # in t0 et t1
       add  t2, zero, a2   # The number of bits in t2
       add  a0, zero, zero # Initialize the result

# The loop takes 32 bits of each list and counts the number of
# different bits for each pair of 32 bits. It finds the total
# number of different bits in the lists
loop:
    beq  t2, zero, fin     # All the bits have been checked?
    lw   t3, 0(t0)         # Put the 32 following bits of
    lw   t4, 0(t1)         # each list in t3 and 4
    addi t5, zero, 32      # Init. counter at 32 (bits)
    slt  t6, t2,   t5      # If less than 32 bits remain
    beq  t6, zero, cont1   # set counter value to the number
    add  t5, zero, t2      # of remaining bits

cont1:
    xor  t6, t3, t4        # The different bits of the 32 bit
                           # pair are stored in t6

# The loop cont2 counts the number of bits set to 1 in the
# register t6 (the number of different bits in the current
# comparison)
cont2:
    andi  t3,  t6, 1       # Extract the last bit
    add   a0,  a0, t3      # Add this bit to a0
    srli  t6,  t6, 1       # Right shift t6 by 1
    addi  t2,  t2, -1      # Decrement the counter of total bits
    addi  t5,  t5, -1      # Decrement the counter of  the loop
    bne   zero, t5, cont2  # Check the end of the loop
    addi  t0,  t0, 4       # Increment pointer
    addi  t1,  t1, 4       # to read the next 32 bits
    j     loop             # Jumps to the top of the loop
fin:
    nop
```

b) The RISC-V code is written for a little-endian processor. This can be deduced from the fact that counting the bits that are different starts from the least significant bit (instruction `andi t3, t6, 1`. That is, it is assumed that the least significant byte of a word is stored at the smallest address in memory, which characterizes little-endian processors.

Figure 92 shows how the bytes are organized in the memory and the difference between little and big-endian processors.

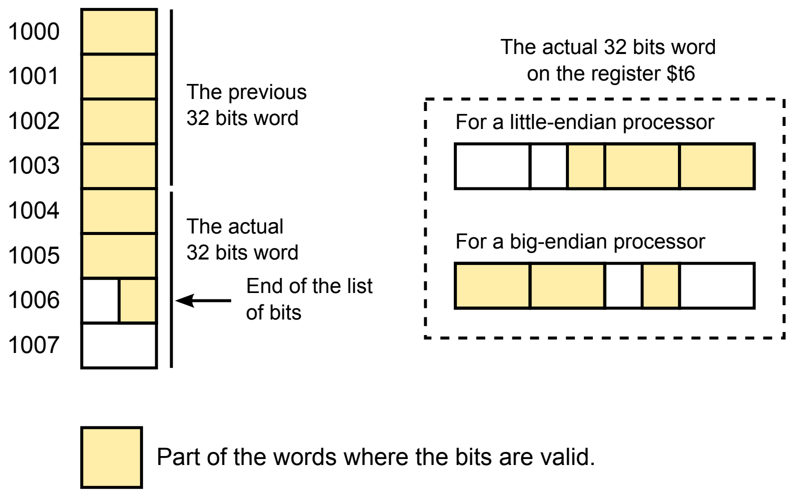

1000
1001
1002
1003
1004
1005
1006
1007
The previous 32 bits word
The actual 32 bits word
End of the list of bits
The actual 32 bits word on the register $t6
For a little-endian processor
For a big-endian processor
Part of the words where the bits are valid.

Figure 92: Words in memory for little/big5-endian processors

c) The **xor** operation can be expressed logically as follows:

$$\text{A xor B} = \text{A} \cdot \bar{\text{B}} + \bar{\text{A}} \cdot \text{B} \tag{1}$$

We can thus replace instruction `xor t6, t3, t4` with the following sequence of instructions, the result is stored in `t6` as in the original instruction.

```
    not t6, t3          # not(A) in t6
    not s0, t4          # not(B) in s0
    and s1, s0, t3      # A.not(B) in s1
    and s0, t6, t4      # B.not(A) in s0
    or  t6, s1, s0      # A.not(B) + B.not(A) in t6
```

[Exercise 27] Multiplication in Finite Fields

We would like to create a RISC-V program that implements elementary operations for finite fields modular arithmetic, in binary representation. A finite field F(2^n) is the set of integers that can be represented on n bits.

When performing an addition in a finite field, each pair of bits is added independently, i.e., the carry is not propagated to the adjacent higher weight pair as done in an ordinary addition.

Therefore, an addition corresponds to a simple bit-wise **`xor`** operation. The following example compares an ordinary addition to one carried out in a finite field F(2^4):

Ordinary Addition	**Addition in Finite Field F(2^n)**
0 1 0 1	0 1 0 1
+ 0 0 1 1	+ 0 0 1 1
1 0 0 0	0 1 1 0

Multiplication in a finite field F(2^n) is carried out in two phases. The first phase consists in multiplying the two numbers similarly to an ordinary multiplication, i.e., by generating partial products then summing them. The addition of these partial products will yield a different result because this operation is defined differently in finite fields. This is illustrated by the following example:

Ordinary Multiplication	**Multiplication in Finite Field F(2^n)**
1 0 1 1	1 0 1 1
× 1 1 1 0	× 1 1 1 0
0 0 0 0	0 0 0 0
1 0 1 1	1 0 1 1
1 0 1 1	1 0 1 1
+ 1 0 1 1	+ 1 0 1 1
1 0 0 1 1 0 1 0	1 1 0 0 0 1 0

a) Write a RISC-V program that implements the first phase of the multiplication in a finite field. When the program starts, the two 16-bit operands are in registers `a0` and `a1`. The 31-bit result is stored in register `a0` at the end of execution.

It can be observed that the result of the multiplication's first phase cannot be correct as it generally does not belong to the finite field F(2^4). This is why the second phase is necessary. The second phase of a multiplication in a finite field F(2^4) consists in finding a result on n bits. To this effect we use a property of finite fields: in each finite field, a value $m \in$ F(2^n) is associated to 2^n. For example, in the finite field F(2^4), we can have $2^4 = m = 3$. Thanks to this property, all bits whose weight is greater or equal to n can be rewritten as a function of m and added to the result.

In the example we have F(2^4) and $2^4 = 3$:

$$\begin{aligned} 2^4 &= \quad 3 \quad &&= 0011 \\ 2^5 &= 2^4 << 1 &&= 0110 \\ 2^6 &= 2^4 << 2 &&= 1100 \end{aligned}$$

Starting from a certain power that depends on m, many iterations are necessary before having a result on n bits:

$$\begin{aligned} 2^7 &= 2^4 << 3 = 11000 = 2^4 + 8 = 0011 + 1000 = 1011 \\ 2^8 &= 2^4 << 4 = 110000 = 2^5 + 2^4 = 0110 + 0011 = 0101 \\ 2^9 &= 2^4 << 5 = 1100000 = 2^6 + 2^5 = 1100 + 0110 = 1010 \\ &\ldots \end{aligned}$$

Note that all sums are done according to the definition of the addition in the finite field (which is bit-wise XOR). Given this property, we can replace each bit of the first phase's result whose weight lies between n and $2n-1$ with values obtained from m, so as to obtain a value that belongs to the finite field F(2^n). In the preceding multiplication example, if $2^4 = 3$, we obtain:

$$1100010 = 2^6 + 2^5 + 2 = 1100 + 0110 + 0010 = 1000$$

Thus, in the finite field F(2^4), with $2^4 = 3$, the result of the multiplication of 11 by 14 is 8.

b) In the finite field F(2^4), with $2^4 = 3$, multiply 5 by 13 giving all steps of the calculation.

c) In order to reduce the result to n bits, we can go through all bits whose weight lies between n and $2n - 1$ and if their value is `'1'` we set it to `'0'` and add a value depending on m. Does it seem wiser to go through the bits starting from the one with the most significant weight or from the one with the least significant weight ? Why ? How should be generated the value that you will add ?

d) Create a RISC-V program that reduces a 32-bit number in a given finite field. When the program starts, the number is supplied in register `a0` (and could be the result of the multiplication's first phase). The size n of the finite field is given as an initial value in register `a1`. The value m to be associated to 2^n is given as an initial value in `a2`. Register `a0` is used to store the result at the end of execution.

[Solution 27] Multiplication in Finite Fields

a) Multiplication's first phase:

```
phase1: add  t3, zero, zero # partial sum
        add  t0, zero, a0
        add  t1, zero, a1
loop:   beq  t1, zero, fin
        andi t2, t1, 1
        beq  t2, zero, next
        xor  t3, t3, t0
next:   slli t0, t0, 1
        srli t1, t1, 1
        j    loop
fin:    addi a0, t3, 0
        nop
```

b) The result of 5 times 13 in the finite field is given below:

$$\begin{array}{r} 1\,1\,0\,1 \\ \times \qquad 0\,1\,0\,1 \\ \hline 1\,1\,0\,1 \\ 0\,0\,0\,0 \\ 1\,1\,0\,1 \\ +\ 0\,0\,0\,0 \\ \hline 0\,1\,1\,1\,0\,0\,1 \end{array}$$

$111001 = 2^5 + 2^4 + 9 = 0110 + 0011 + 1001 = 1100$

c) By going through the bits from right to left we run the risk of reintegrating bits with an index greater than $n-1$, and thus having to test the same bits several times. By going through them from left to right this problem no longer exists, as there is no way of re-injecting new bits into already visited slots. The value to be added is initialized as follows: $r = (2^n + m) << (31 - n)$. Thus for the first iteration, if the most significant bit of the value to convert is `'1'`, performing a `xor` between this value and r will allow inverting the most significant bit and adding m (shifted to the right position). Then, r must be shifted by 1 bit to the right for the next iteration.

d) The code for the second phase is given below:

```
phase2: add  t3, zero, a0
        addi t0, zero, 1
        slli t1, t0, 31
        sll  t0, t0, a1
        add  t0, t0, a2      # t0 = 2^n + m
        addi t2, zero, 31    # t2 = 31
        sub  t2, t2, a1      # t2 = t2 - a1
        sll  t0, t0, t2
loop:   srl  t2, t3, a1
        beq  t2, zero, fin
        and  t2, t3, t1
        beq  t2, zero, next
        xor  t3, t3, t0
next:   srli t1, t1, 1
        srli t0, t0, 1
        j    loop
fin:    addi a0, t3, 0
        ret
```

`t0` is the replacement value for the most significant bits. `t1` is the mask used to go through the bits whose index (weight) is greater than $n - 1$.

[Exercise 28] Architecting a new instruction

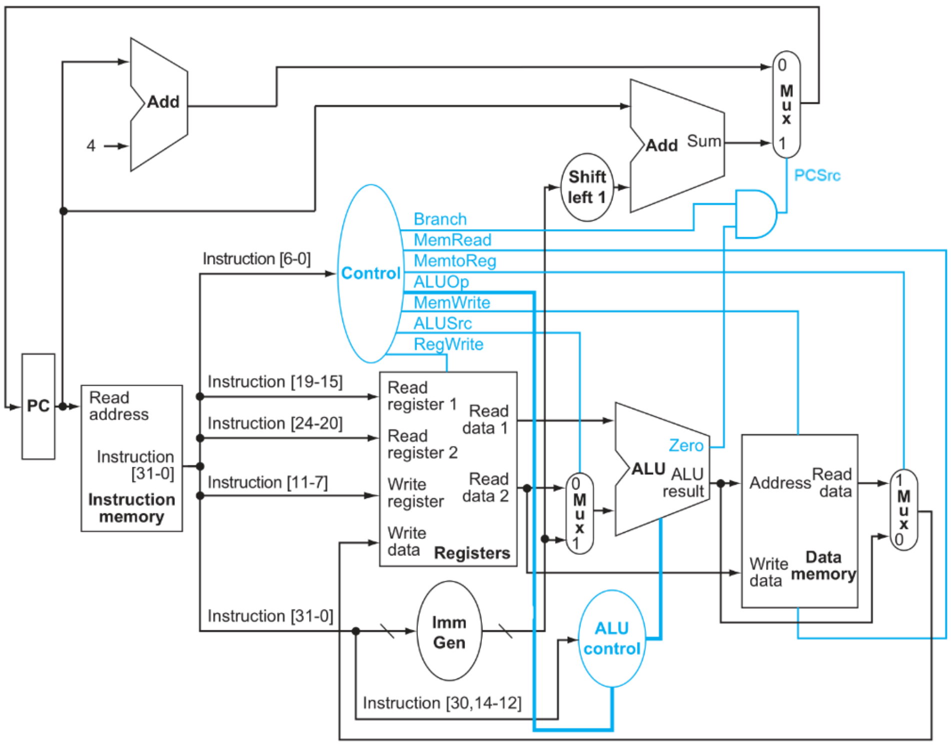


Figure 93: Single-cycle CPU schematic

Consider the single-cycle CPU as illustrated in Figure 93. We want to add a new instruction to this CPU, called `rdpc`. This instruction will copy the current value of the PC register into a destination register located inside the register file.

`rdpc` will only require a destination register; no source register operands are determined (since the value will be taken from the PC register) and no immediate value is specified. Thus, for instance, `rdpc x2` will copy the *current* value of PC into `x2`; this is the memory address correponding to this instruction.

a) Make the required modifications in the datapath in order to support this new instruction. You only need to make changes to the schematic; assume that the instruction set supports `rdpc` and the opcode exists.

Add the necessary logic so `rdpc` can be implemented; you may add one new control signal (called `PCCopy`) to the CPU. Explain the changes you have made to the schematic, and if you add a control signal, explain when the signal will be active.

[Solution 28] Architecting a new instruction

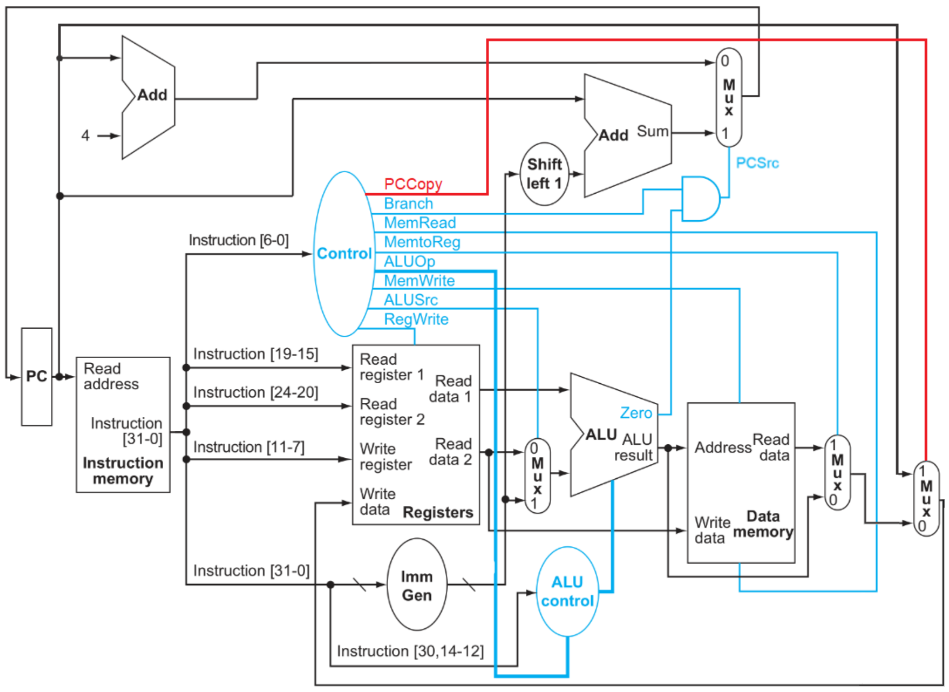


Figure 94: Modified single-cycle CPU schematic

a) Figure 94 denotes how this operation can be achieved on the CPU schematic.

The signal `PCCopy` will be active if and only if the instruction is equal to `rdpc`.

A multiplexer has been added in the path to the registers' write data, and the control signal for this multiplexer is `PCCopy`. If the instruction is `rdpc`, the value of PC will be passed over as write data for the registers. Otherwise, whatever value exists on the previous multiplexer (as was the case normally, without our changes) will be written.