МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Кафедра комп'ютеризованих систем управління

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

__________Литвиненко О.Є.
«_____»____________2022 р.

КВАЛІФІКАЦІЙНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ЗДОБУВАЧА ОСВІТНЬОГО СТУПЕНЯ "МАГІСТР"

Тема: Програмний засіб моделювання комбінаційних схем

Виконавець: ______________ Шудря В.Д.

Керівник: ______________ Марченко Н.Б.

Нормоконтролер: ______________ Тупота Є.В.

Київ 2022

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет Кібербезпеки, комп'ютерної та програмної інженерії
Кафедра комп’ютеризованих систем управління
Спеціальність 123 «Комп'ютерна інженерія»
(шифр, найменування)

Освітньо-професійна програма «Системне програмування»
Форма навчання денна

ЗАТВЕРДЖУЮ
Завідувач кафедри
__________Литвиненко О.Є.
«_____»____________2022 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи (проєкту)

Шудря Віталій Дмитрович
(прізвище, ім'я, по батькові випускника в родовому відмінку)

1. Тема кваліфікаційної роботи (проєкту): Програмний засіб моделювання комбінаційних схем

затверджена наказом ректора від «16» вересня 2022 р. №1530/ст

2. Термін виконання роботи (проєкту): з 05.09.2022 по 30.11.2022

3. Вихідні дані до роботи (проєкту): державні стандарти України, тема кваліфікаційної роботи, технічне завдання

4. Зміст пояснювальної записки:

1) аналіз предметної області та постановка задачі;

2) опис технологій для розробки та проєктування додатку;

3) опис процесу розробки та основних модулів додатку.

5. Перелік обов'язкового графічного (ілюстративного) матеріалу:

1) *UML*-діаграма бази даних;

2) структура клієнтського додатку;

3) структура *web*-серверу;

4) діаграма станів клієнтського додатку;

5) діаграма станів *web*-серверу;

6) *use-case* діаграма клієнтського додатку.

6. Календарний план-графік

№ пор.	Завдання	Термін виконання	Відмітка про виконання
1	Проаналізувати літературу по темі кваліфікаційної роботи	05.09.2022-10.09.2022	
2	Провести аналіз існуючих рішень та розробити постановку завдання	11.09.2022-16.09.2022	
3	Спроєктувати систему та обрати необхідні для розробки технології	17.09.2022-30.09.2022	
4	Розробити систему за проєктом	01.10.2022-31.10.2022	
5	Оформити пояснювальну записку	01.11.2022-27.11.2022	
6	Оформити графічний та ілюстративний матеріал	28.11.2022-30.11.2022	

7. Дата видачі завдання: «05» вересня 2022 р.

Керівник кваліфікаційної роботи (проєкту) _____________ Марченко Н.Б.
(підпис керівника) (П.І.Б.)

Завдання прийняв до виконання __________________ Шудря В.Д.
(підпис випускника) (П.І.Б.)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи “Програмний засіб моделювання комбінаційних схем”: 70 с., 26 рис., 15 літературних джерел, 2 додатки.

Ключові слова: СХЕМА, ЛОГІКА, МОДЕЛЮВАННЯ, *WEB*-ДОДАТОК, ОБЧИСЛЕННЯ, КОМБІНАТОРИКА.

Об'єкт дослідження: процес створення засобу моделювання комбінаційних схем.

Предмет дослідження: Програмний засіб моделювання комбінаційних схем.

Мета кваліфікаційної роботи: реалізація програмного забезпечення моделювання комбінаційних схем.

Методи дослідження: *HTTP*, *REST*, *WEB API*, *IDE Visual Studio* 2019, *C#* 8.0, *ASP.NET Core*, *ASP.NET Core MVC*, *Entity Framework Core*, *LINQ*, *JavaScript ES*6, *HTML* 5, *CSS* 3, *Ajax*, *JQuery*, *PostgreSQL.*

В даній кваліфікаційній роботі було проведено аналіз предметної області та існуючих засобів моделювання комбінаційних схем, на основі результату аналізу було спроєктовано і реалізовано програмний засіб моделювання комбінаційних схем. Засіб надає користувачу можливість побудови та симуляції комбінаційних схем на різних рівнях.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ 6
ВСТУП 7
РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАДАЧІ 10
1.1. Аналіз предметної області 10
1.2. Аналіз існуючих засобів створення комбінаційних схем 25
1.3. Аналіз недоліків існуючих рішень 29
1.4. Постановка задачі 30
1.5. Висновки до розділу 31
РОЗДІЛ 2 ВИКОРИСТАНІ ТЕХНОЛОГІЇ ТА ПРОЄКТУВАННЯ 32
2.1. Визначення необхідний технологій для розробки 32
2.2. Проєктування функціональної частини системи 43
2.3. Висновки до розділу 52
РОЗДІЛ 3 РОЗРОБКА ПРОГРАМНОГО ЗАСОБУ 53
3.1. Розробка *web*-серверу 53
3.2. Розробка клієнтського додатку 57
3.3. Висновки до розділу 66
ВИСНОВКИ 67
СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ 69
ДОДАТОК А 70
ДОДАТОК Б 76

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

HTTP (*Hyper Text Transfer Protocol*) – Протокол передачі гіпертексту

БД – База Даних

СУБД – Система Управління Базами Даних

WEB – Всесвітня павутина

API (*Application Programming Interface*) – Інтерфейс прикладного програмування

REST (*Representational State Transfer*) – Передача репрезентативного стану

ВСТУП

Комп'ютерна логіка охоплює логічні, математичні та технічні задачі, принципи та моделі цифрових систем. Комп'ютерна логіка включає безліч моделей і способів функціонування комп'ютерних систем та їх компонентів. Комп'ютерна логіка досліджує основні функції цифрових систем.

Комп'ютерна логіка вивчає моделі, методи аналізу та синтезу логічних комбінаційних схем, цифрових автоматів, які є базовими логічними моделями сучасних цифрових систем контролю та управління.

Фундаментальним елементом комп'ютерної системи є поняття автомата. Автомат – це математична модель технічної системи. У цьому випадку він трактується як «чорний ящик», що має кінцеве число входів і виходів, а також деяку кількість внутрішніх станів.

Є безліч сфер управління які потребують високий рівень відповідальності, такі як:

- ядерна промисловість;
- хімічна промисловість;
- комплекси виробничого призначення;
- оборонні комплекси;
- космічні комплекси.

Успіх у роботі залежить від чіткості та злагодженості виконуваних дій, від можливості однозначної обробки інформації. Різний характер фізичних процесів, складний характер взаємодії викликають труднощі розробки, алгоритмізації і програмуванні завдань управління. Виникають труднощі, пов'язані із забезпеченням досягнення чіткості та структурованості. Для вирішення цих завдань використовується розроблений математичний апарат теорії автоматів за допомогою синтезу комбінованих схем.

Синтез комбінованих схем зводиться до реалізації аналітичних виразів булевих функцій за допомогою логічних елементів. Один з методів синтезу цифрових автоматів з пам'яттю дозволяє звести завдання структурного синтезу вільного автомата з пам'яттю до синтезу комбінованих схем. Метод синтезу, в основі якого закладено цей принцип, був названий канонічним методом структурного синтезу автоматів з пам'яттю. Канонічний метод структурного синтезу оперує елементарними автоматами, які поділяються на два класи:

– перший клас складають елементарні автомати, які називають елементами пам’яті;
– другий клас складають елементарні комбінаційні автомати – логічні елементи.

Для побудови та відладки комбінаційної схеми використовуються спеціалізовані програмні додатки, з можливостю графічного редагування та часової симуляції моделі.

Мета кваліфікаційної роботи – реалізація програмного забезпечення моделювання комбінаційних схем для використання в побудові логічних схем цифрових систем.

Галузь застосування – сфера розробки цифрових систем, підприємства та виробництва, наукова діяльність, освітня сфера.

Структура та зміст теоретичної та практичної частини кваліфікаційної роботи. Кваліфікаційна робота складається зі вступу, трьох розділів та висновків до них.

У першому розділі проведено аналіз предметної області та теоретичних основ побудови систем, складові та загальні характеристики систем. Також був проведений аналіз існуючих рішень та були визначені основні недоліки.

Можна виділити наступні функціональні можливості такої системи:

– реєстрація та авторизація користувачів;
– доступ до власних репозиторіїв схем;
– створення та моделювання комбінаційних схем;

– можливість перегляду репозиторіїв інших користувачів;

– моделювання та перегляд часових діаграм.

Другий розділ представляє собою опис архітектури програмно додатку та перелік і опис використовуваних технологій. Програмний засіб моделювання комбінаційних схем є комплексною структурою, яка складається з трьох основних компонент:

– клієнтський додаток — *JavaScript*, *HTML*, *CSS*;

– *web*-сервер — *C#*, *ASP.NET Core*, *Blazor*, *Entity Framework*;

– база даних — *PostgreSQL.*

Третій розділ присвячений процесу створення програмного засобу та пояснення роботи основних його компонентів.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАДАЧІ

1.1. Аналіз предметної області

Комп’ютерна логіка, яку ми знаємо сьогодні, вимагала співпраці тисяч людей протягом тисячоліть. До того, як електронні пристрої були створені, математики та філософи неусвідомлено створювали двійкову мову та закладали основу для сучасних обчислювальних пристроїв.

Пізніше ці концепції були застосовані до подібних комунікаційних систем, таких як азбука Морзе та двійковий код, з якими ми знайомі сьогодні. Такі математики, як Джордж Буль і Чарльз Беббідж, застосували ці концепції до механічних пристроїв для їх автоматизації. Хоча обчислювальна техніка має багату історію, логічні пристрої, які дозволяють комп’ютерам функціонувати, привертають менше уваги.

У 10 столітті в стародавньому Китаї вчений на ім'я Шао Юн змінив гексаграми з І-цзин у формат, схожий на двійковий. Деякі вчені навіть припускають, що І-цзин надихнув Готфріда Лейбніца на створення двійкового коду.

У Стародавній Греції також існували власні таблиці істинності з ТАК і НІ, які функціонували як ВВІМК і ВИМК, якщо говорити про випадок транзисторів. Швидше за все, Лейбніц черпав натхнення у грецьких математиків, оскільки їхні роботи були найдоступнішими для вчених у Європі протягом 17-го та 18-го століть.

Хоча Готфрід Вільгельм Лейбніц вважається винахід двійкової системи, якою ми її знаємо сьогодні, перші логічні пристрої були створені Чарльзом Беббіджем. Його першим різницевим механізмом був механічний калькулятор, призначений для таблиці поліноміальних рівнянь і друку результатів на рулоні паперу.

Першими поширеними логічними елементами були електромагнітні реле, які в основному були перемикачами ВВІМК-ВИМК. Електромеханічне реле було винайдено Джозефом Генрі в 1835 році, але геніальність його винаходу була усвідомлена лише пізніше, коли його реле почали використовувати в телеграфі. Після винаходу поворотного циферблата в 1890 році були розроблені більш складні реле з 10 положеннями.

Перші логічні схеми цифрових програмованих обчислювальних пристроїв були розміщені у вакуумних трубках, на основі яких були побудовані такі відомі комп'ютери, як *ENIAC*, *Colossus* і *Pilot Ace* Алана Тюрінга. Ці перші цифрові комп'ютери були величезними і мали тисячі вакуумних ламп. Для того, щоб зробити обчислювальні пристрої загальнодоступними, логічну схему потрібно було зменшити фізично.

Зменшення розміру відбулося в 1960-х роках з появою резисторно-транзисторною логікою (*RTL*) і транзисторно-транзисторною логікою (*TTL*). *RTL* були першими логічними пристроями, які були включені до інтегральних схем і пізніше були використані в комп'ютері наведення *Apollo*. Перші інтегральні схеми *TTL* були розроблені в 1963 році *Sylvania* і були популяризовані серією 7400 від *TI* (*Texas Instruments Incorporated*). Відтоді сучасна логічна схема скорочує ці концепції та додає додаткові логічні елементи.

1.1.1 Алгебра логіки

Алгебра логіки широко використовується для опису роботи цифрових автоматів. Висловлювання є основним поняттям алгебри логіки і є деяким виразом, в якому можна стверджувати про його істинність або хибність. Прийнято позначати істинність цифрою - 1, а хибність - 0, тобто якщо висловлювання x при 1 - істинне, то при 0 - хибне. Вводиться поняття логічної змінної x, яка може набувати значень {1, 0}, в зв'язку з чим говорять про абсолютно істинне ($x = 1$) або абсолютно хибне ($x = 0$) висловлювання.

Над логічними змінними допускаються різні операції, тому можна говорити про логічні функції логічних змінних, які також набувають значення 0 або 1. З чотирьох функцій від однієї логічної змінної найважливішою є функція логічного заперечення (НІ), яка змінює значення логічної змінної на протилежне.

З шістнадцяти існуючих функцій від двох змінних виділимо дві основі логічне додавання або диз'юнкція (формула 1.1) та логічне множення або кон'юнкція (формула 1.2).

$$x_1 + x_2 = y, \quad (1.1)$$

$$x_1 \bullet x_2 = y. \quad (1.2)$$

Результат операції логічного додавання є істинним, якщо хоча б один з операндів дорівнює логічній одиниці, а результат операції логічного множення є хибним, якщо хоча б один із операндів дорівнює логічному нулю. Ці дві операції спільно з логічним запереченням складають булевий базис (І, АБО, НІ).

Між логічними функціями існують певні відносини, а самі вони мають ряд властивостей. Зокрема, справедливі аксіоми, що вказані у формулах 1.3 – 1.10.

$$x = !!x, \quad (1.3)$$

$$x + x = x; x \bullet x = x, \quad (1.4)$$

$$x + 0 = x, \quad (1.5)$$

$$x + 1 = 1, \quad (1.6)$$

$$x \bullet 0 = 0, \quad (1.7)$$

$$x \bullet 1 = x, \quad (1.8)$$

$$x \bullet !x = 0, \quad (1.9)$$

$$x + !x = 1. \quad (1.10)$$

За аналогією із властивостями арифметичних операцій можна говорити про властивості логічних операцій:

– асоціативність (формула 1.11);

– комутативність (формула 1.12);

– дистрибутивність – кон'юнкції щодо диз'юнкції (формула 1.13) та диз'юнкції щодо кон'юнкції (формула 1.14).

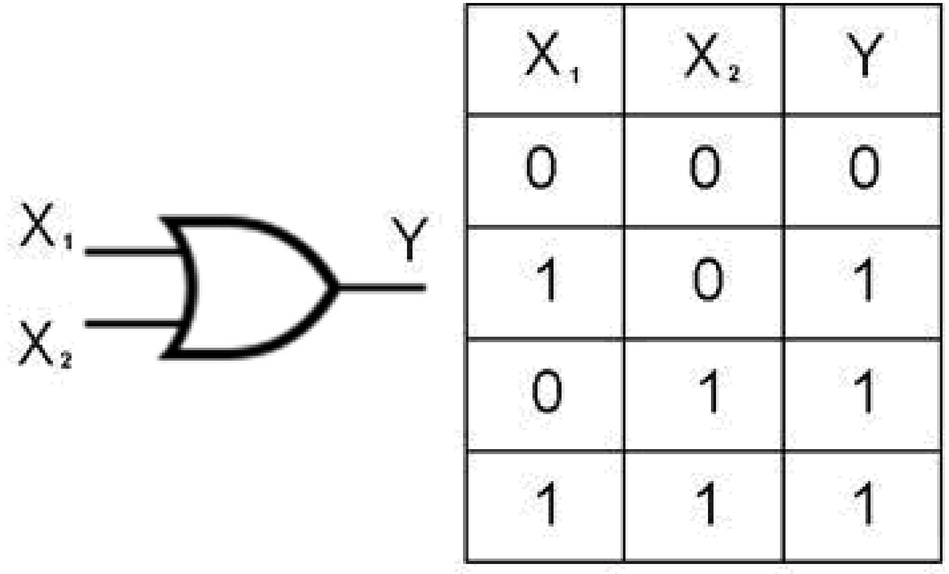
$$x_1 + (x_2 + x_3) = (x_1 + x_2) + x_3;\ x_1 \bullet (x_2 \bullet x_3) = (x_1 \bullet x_2) \bullet x_3, \quad (1.11)$$

$$x_1 + x_2 = x_2 + x_1;\ x_1 \bullet x_2 = x_2 \bullet x_1, \quad (1.12)$$

$$x_1 \bullet (x_2 + x_3) = (x_1 \bullet x_2) + (x_1 \bullet x_3), \quad (1.13)$$

$$x_1 + x_2 \bullet x_3 = (x_1 + x_2) \bullet (x_1 + x_3). \quad (1.14)$$

Існують різні способи подання логічних функцій. Дуже наочним є табличне уявлення, при якому для кожного набору значень змінних у таблиці істинності (рис 1.1) вказується значення самої логічної функції. Але такий спосіб незручний для аналізу властивостей функцій логіки алгебри.



X_1	X_2	Y
0	0	0
1	0	1
0	1	1
1	1	1

Рис. 1.1. Логічний елемент АБО

1.1.2 Логічні елементи

Логічний елемент — це невеликий цифровий електронний пристрій, який виконує булеву функцію з двома входами та одним виходом. Вхідними даними для цього елемента являються бінарні числа. Логічна 1 дорівнює істині або в

електротехнічному форматі високій напрузі, а логічний 0 відповідно хибний або низька напруга. Кожен логічний елемент відповідає таблиці істинності, яка відображає можливі комбінації вхідних даних та їх виходів. Роботу кожного окремого логічного елемента можна легко зрозуміти, і вона схожа на додавання та множення, які ми вже знаємо у звичайній математиці.

Роботу кожного окремого логічного елемента можна легко зрозуміти, і вона схожа на додавання та множення, які ми вже знаємо у звичайній математиці. Логічний елемент ідентичний вимикачу світла, так якщо він увімкнений, то на виході логічна одиниця, коли вимкнений, то вихід дорівнює логічному нулю.

1.1.3 Типи логічних елементів і таблиці істинності

Базове символічне представлення логічних елементів зображене у таблиці 1.1.

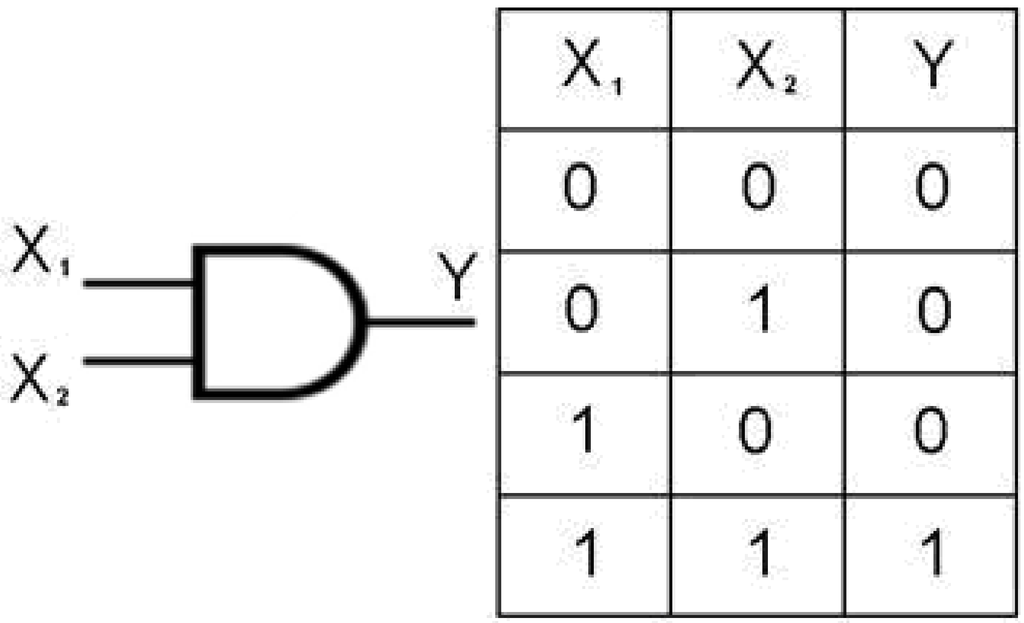
Таблиця 1.1

Логічні елементи

Логічний елемент	Логічна операція
	І (кон'юнкція)
	АБО (диз'юнкція)
	НІ (інвертор)
	І-НІ (штрих Шефера)

Логічний елемент	Логічна операція
	АБО-НІ (стрілка Пірса)
	заперечне АБО
	заперечне АБО-НІ

Логічний елемент І (кон’юнкція) (рис. 1.2). Елемент І вважається основним логічним елементом, він виконує операцію множення на логічних входах. Щоб активувати цей елемент, усі входи повинні дорівнювати логічній одиниці (увімкнено). Як тільки один електричний вхід дорівнює логічному нулю (вимкнено), тоді і тільки тоді вихід буде дорівнювати логічному нулю (вимкнено).

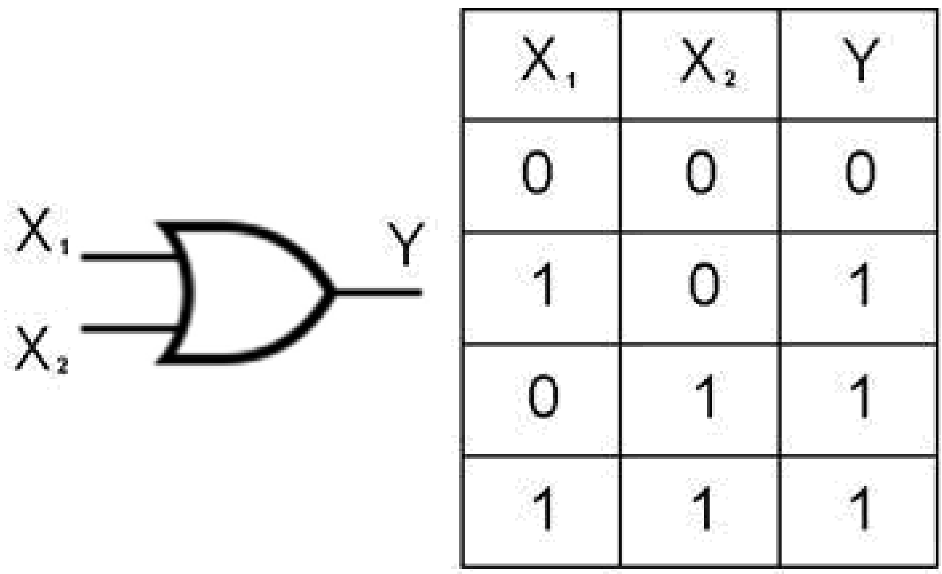


X_1	X_2	Y
0	0	0
0	1	0
1	0	0
1	1	1

Рис. 1.2. Логічний елемент І

Логічний елемент АБО (диз'юнкція) (рис. 1.3). Цей тип воріт є менш обмежувальним, ніж елемент І. Якщо будь-який із входів дорівнює логічній

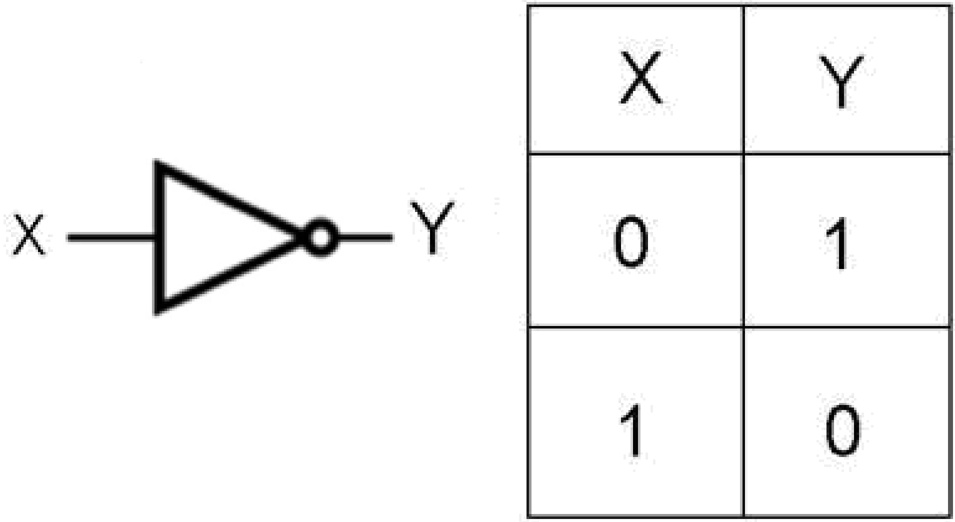
одиниці (увімкнено), тоді на виході також буде логічна одиниця, незалежно від того, чи дорівнюють інші входи логічним одиницям чи нулям.



X_1	X_2	Y
0	0	0
1	0	1
0	1	1
1	1	1

Рис. 1.3. Логічний елемент АБО

Логічний елемент НІ (інвертор) (рис. 1.4). Є найпростішими серед усіх інших логічних воріт. Він виконує операцію інверсії над одним входом. Логічний елемент НІ має єдиний вхід і вихід, які завжди протилежні один одному. Наприклад, якщо вхід, що досягає логічного елемента НІ, дорівнює логічній одиниці, тоді вихід буде завжди дорівнювати логічному нулю і навпаки.

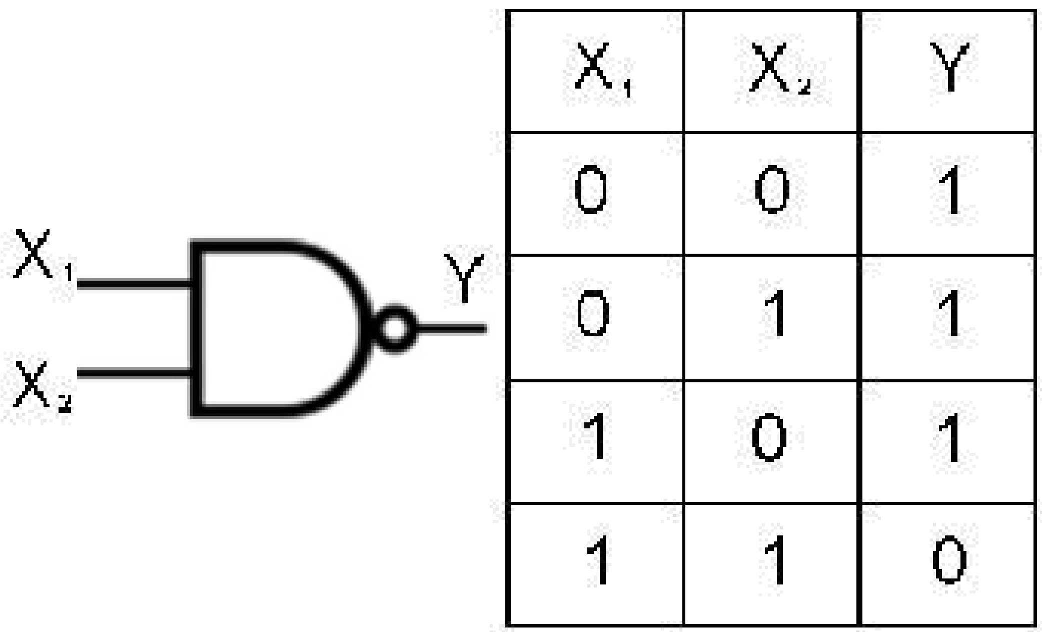


X	Y
0	1
1	0

Рис. 1.4. Логічний елемент НІ

Логічний елемент І-НІ (штрих Шефера) (рис. 1.5). Це дещо складніший елемент ніж елементи І та НІ окремо. Вихід буде хибним (логічний нуль), лише

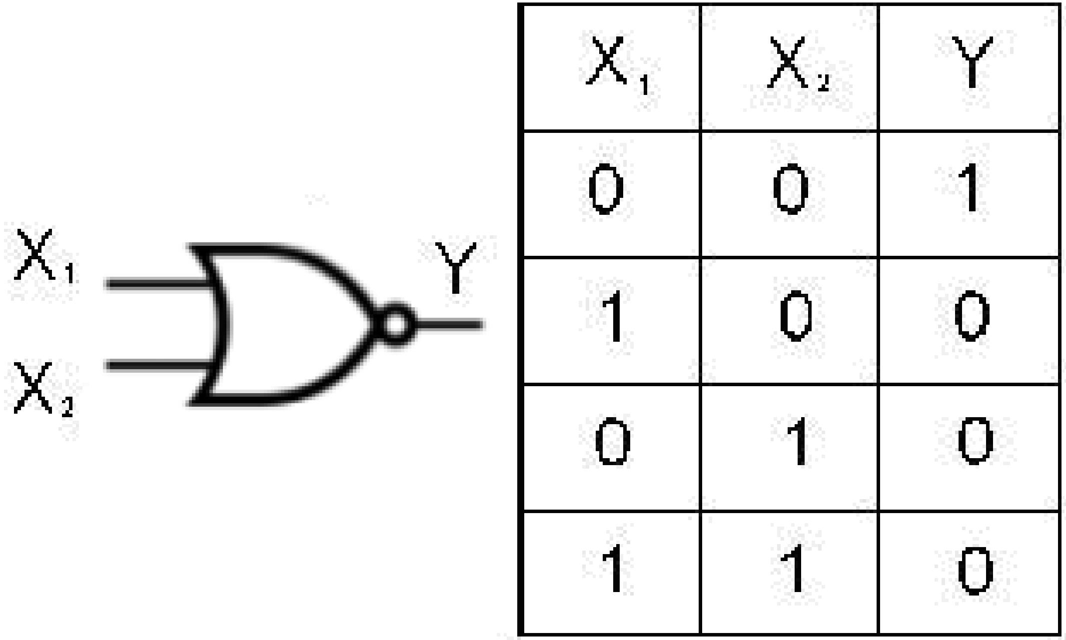
якщо обидва входу є істинними (логічна одиниця), тоді як у будь-якій іншій ситуації вихід буде істинним.



X_1	X_2	Y
0	0	1
0	1	1
1	0	1
1	1	0

Рис. 1.5. Логічний елемент І-НІ

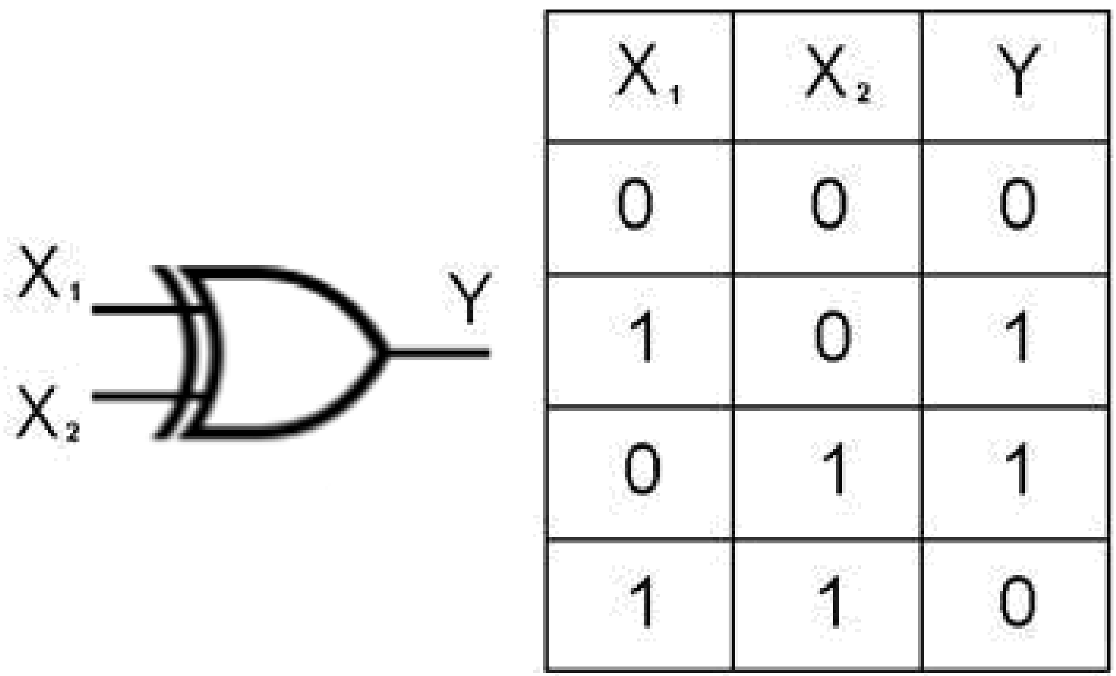
Логічний елемент АБО-НІ (стрілка Пірса) (рис. 1.6). Поєднує в собі функції елемента АБО з інвертором. Вихідні дані цього типу елемента є істинними (логічна одиниця), лише якщо обидва його входи є хибними (логічний нуль). Будь-яка інша комбінація вхідних даних призведе до хибного результату.



X_1	X_2	Y
0	0	1
1	0	0
0	1	0
1	1	0

Рис. 1.6. Логічний елемент АБО-НІ

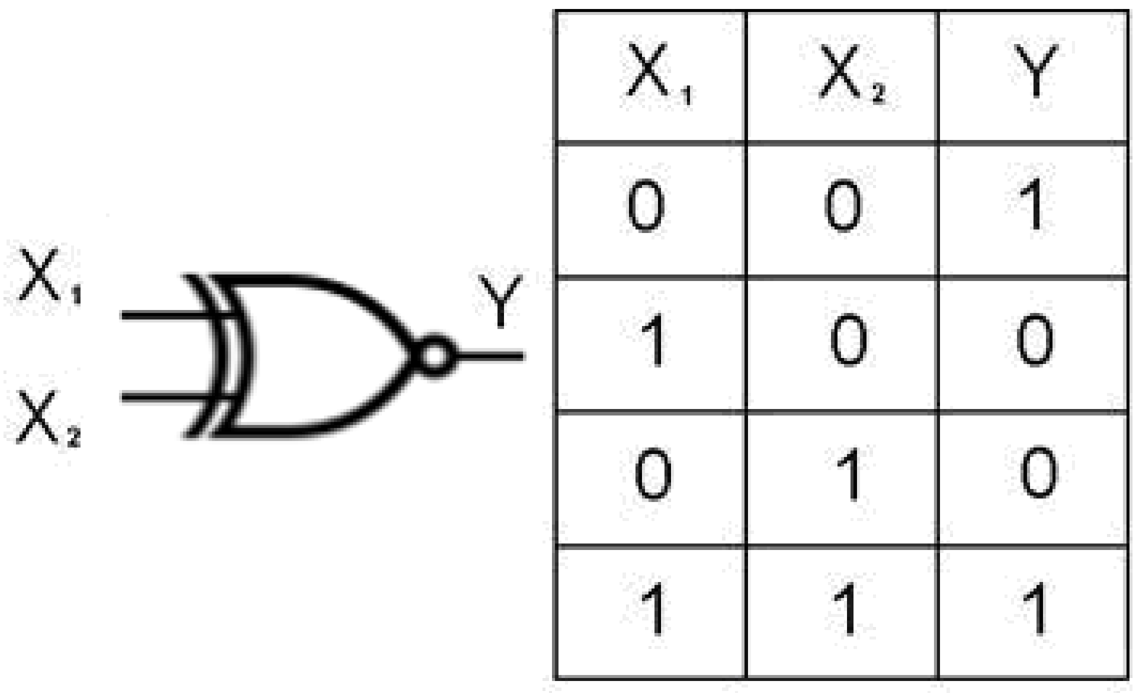
Логічний елемент заперечне АБО (рис. 1.7). Логічний елемент, який дає істинний (логічна одиниця) вихідний сигнал, коли кількість істинних вхідних сигналів непарна. Елемент заперечне АБО реалізує ексклюзивну АБО з математичної логіки; тобто вихідний результат є істинним, якщо один і тільки один із вхідних сигналів є істинним. Якщо обидва вхідні сигнали є хибними (логічний нуль) або обидва є істинними, результат буде хибний. заперечне АБО представляє функцію нерівності, тобто результат є істинним, якщо вхідні дані не однакові, інакше результат є хибним.



X_1	X_2	Y
0	0	0
1	0	1
0	1	1
1	1	0

Рис. 1.7. Логічний елемент заперечне АБО

Логічний елемент заперечне АБО-НІ (рис. 1.8). Це ще один приклад комбінованого логічного елемента з його відмінними особливостями. Елемент заперечне АБО-НІ або також включає в себе інвертор, та для отримання істинного сигналу на виході всі його двійкові входи мають бути подібними незалежно від того, чи всі вони істинні (логічна одиниця) чи хибні (логічний нуль).



X_1	X_2	Y
0	0	1
1	0	0
0	1	0
1	1	1

Рис. 1.8. Логічний елемент заперечне АБО-НІ

Випадки використання логічних елементів:

– Електронні пристрої – кожен електронний пристрій, який ми використовуємо сьогодні, містить цифрові схеми та логічні елементи. Хоча схема логічного елемента виконує лише основні логічні функції, вона все ще є найважливішою частиною будь-якої цифрової схеми. Роль логічних воріт полягає у прийнятті рішень на основі цифрових вхідних сигналів.

– Промислові підприємства – Логічні елементи зазвичай використовуються на промислових підприємствах як параметри безпеки. Елемент АБО використовується для того, щоб виявити будь-які дії в системі. Він повідомляє нам про виникнення будь-яких небажаних подій. Таким чином, логічні елементи діють як індикатор безпечного значення для будь-яких порушених або перевищених параметрів.

– Вимірювання частоти – логічні елементи також використовуються для вимірювання частоти хвилі або імпульсу. Один із логічних елементів І, також широко відомий як шлюз дозволу, пропускає хвилі лише з певною частотою та зупиняє решту.

1.1.4 Приклад використання логічних елементів.

Повний суматор. Повний суматор є одним із таких прикладів використання логічних елементів. Повний суматор працює з трьома входами та має два виходи, такі як сума та перенесення. Повний суматор широко використовується для обчислень, він виконує операцію додавання наведених даних. Цей процес відбувається за лічені секунди, оскільки час перемикання швидкий у порівнянні з аналоговими схемами. Схема повного суматора наведена на рис. 1.9.

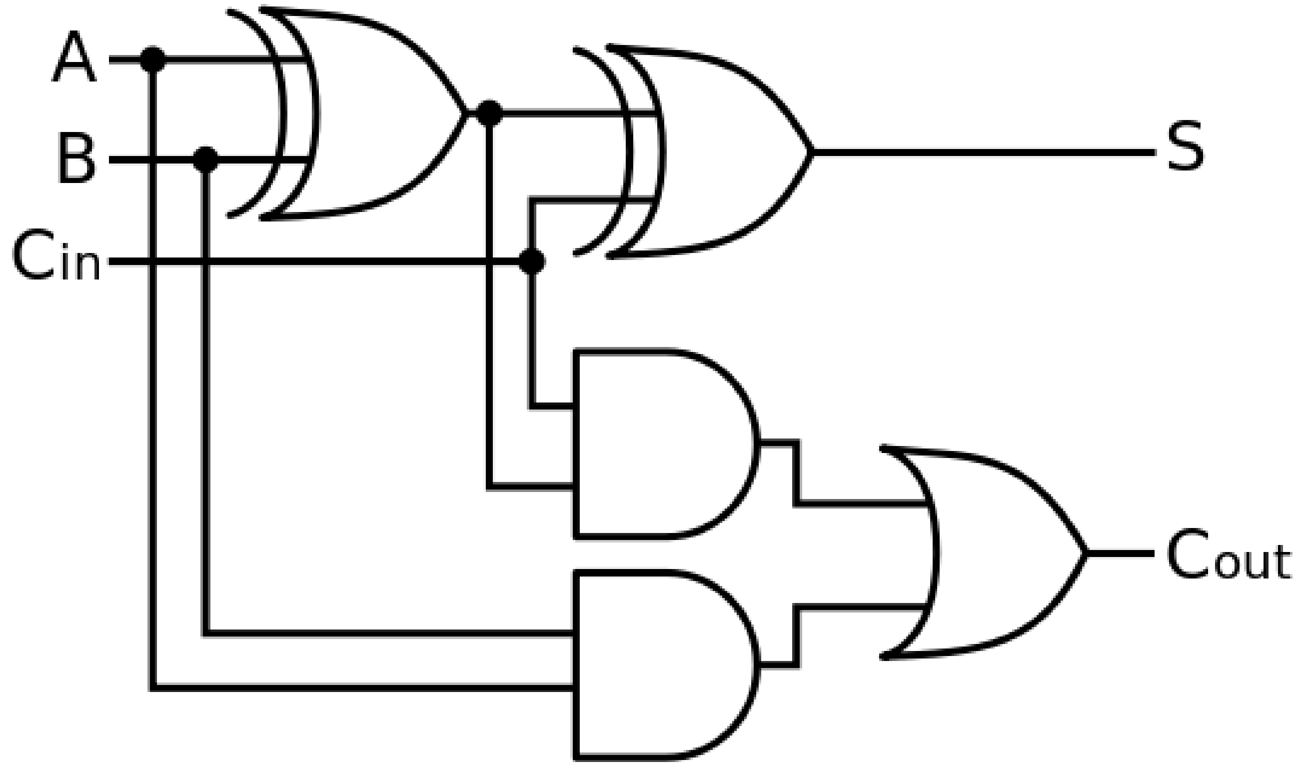


Рис. 1.9. Схема повного суматора

Повний суматор може бути реалізований багатьма різними способами, наприклад, за допомогою спеціальної схеми транзисторного рівня або складається з інших логічних елементів.

Базис повного суматора складається з логічних елементів виключне АБО, АБО та І. Нижче наведено таблицю істинності повного суматора в таблиці 1.2.

Таблиця 1.2

Таблиця істинності повного суматора

Вхідні сигнали			Вихідні сигнали	
A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Тригер – це схема, яка має два стабільних стани і може використовуватися для зберігання інформації про стан — бістабільний мультивібратор. Схема може змінювати стан за допомогою сигналів, що подаються на один або більше керуючих входів, і матиме один або два виходи. Це основний елемент зберігання в послідовній логіці. Тригери є основними елементами цифрових електронних систем, які використовуються в комп'ютерах, комунікаціях та багатьох інших типах систем.

Тригери можна розділити на загальні типи:

– *SR* («встановлення-скидання»);
– *D* («дані» або «затримка»);
– *T* («перемикач»);
– *JK*.

Поведінка певного типу може бути описана за допомогою так званого характеристичного рівняння, яке виводить «наступний» (тобто після наступного тактового імпульсу) вихідний сигнал Q_{next} в рамках вхідного сигналу та/або поточного вихідного сигналу Q.

SR тригер (рис. 1.10) – асинхронний тригер, який зберігає свій попередній стан при неактивному стані обох входів та змінює свій стан при подачі на один із його входів активного рівня. При використанні статичних логічних елементів як будівельних блоків, найбільш фундаментальним є тригер *SR*, де *S* і *R* означають «встановлення» та «скидання». Його можна сконструювати з пари перехресно пов'язаних логічних елементів АБО-НІ або І-НІ. Збережений біт присутній на виході, позначеному *Q*.

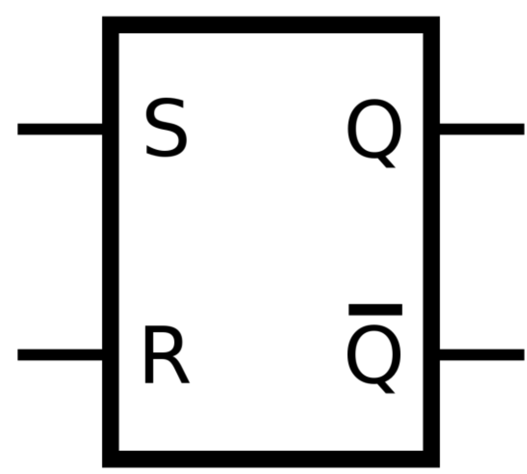


Рис. 1.10. *SR* тригер

Схему *SR* тригера на базисі АБО-НІ зображено на рис 1.11.

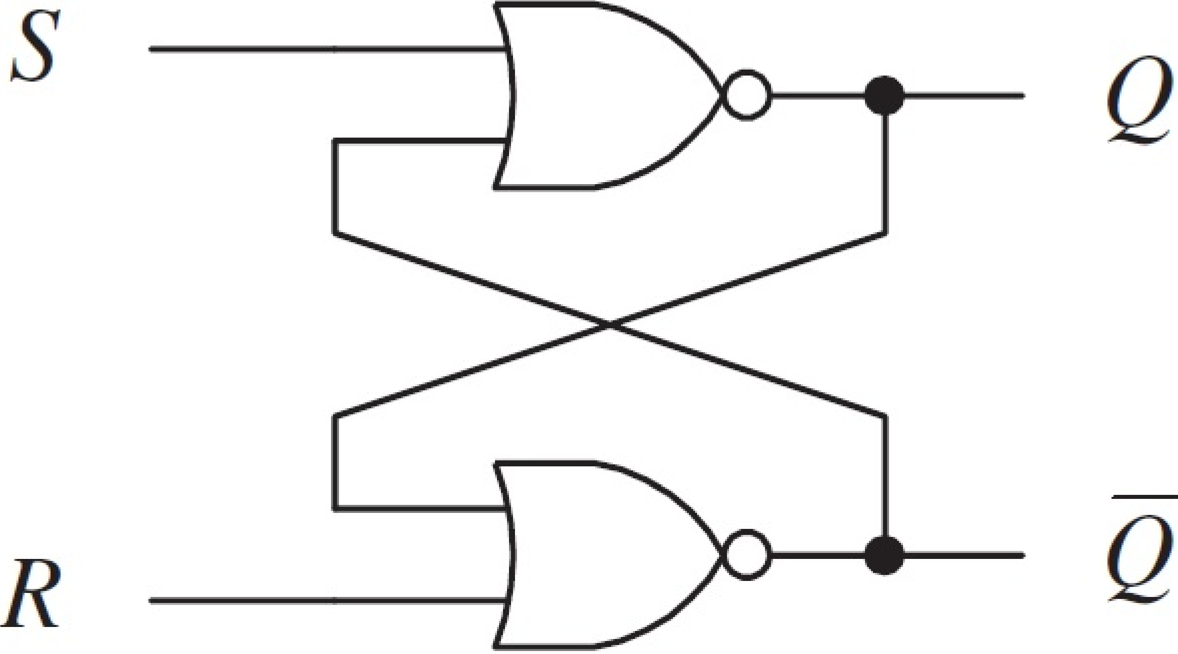


Рис. 1.11. Схема *SR* тригера на базисы АБО-НІ

В таблиці 1.3 наведено стани *SR*-тригера на елементах АБО-НІ.

Таблиця 1.3

Таблиця переходів *SR*-тригера на елементах АБО-НІ

S	R	$Q_{(t)}$	$!Q_{(t)}$
0	1	0	1
1	0	1	0
0	0	$Q_{(t-1)}$	$!Q_{(t-1)}$
1	1	-	-

D-тригер (або тригер затримки) (рис. 1.12) – запам'ятовує стан входу та видає його на вихід. *D*-тригери мають, як мінімум, два входи:

– інформаційний *D*;

– синхронізації *C*.

Збереження інформації відбувається після спаду імпульса синхронізації *C*. Оскільки інформація на виході залишається незмінною до приходу чергового імпульса синхронізації, *D*-тригер називають також тригером із запам'ятовуванням інформації.

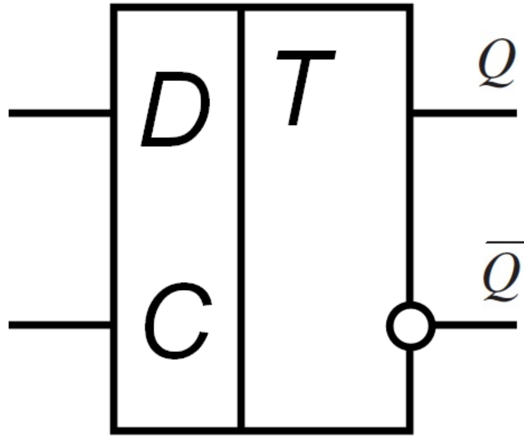


Рис. 1.12. *D*-тригер

Схема D-тригера зображено на рис 1.13.

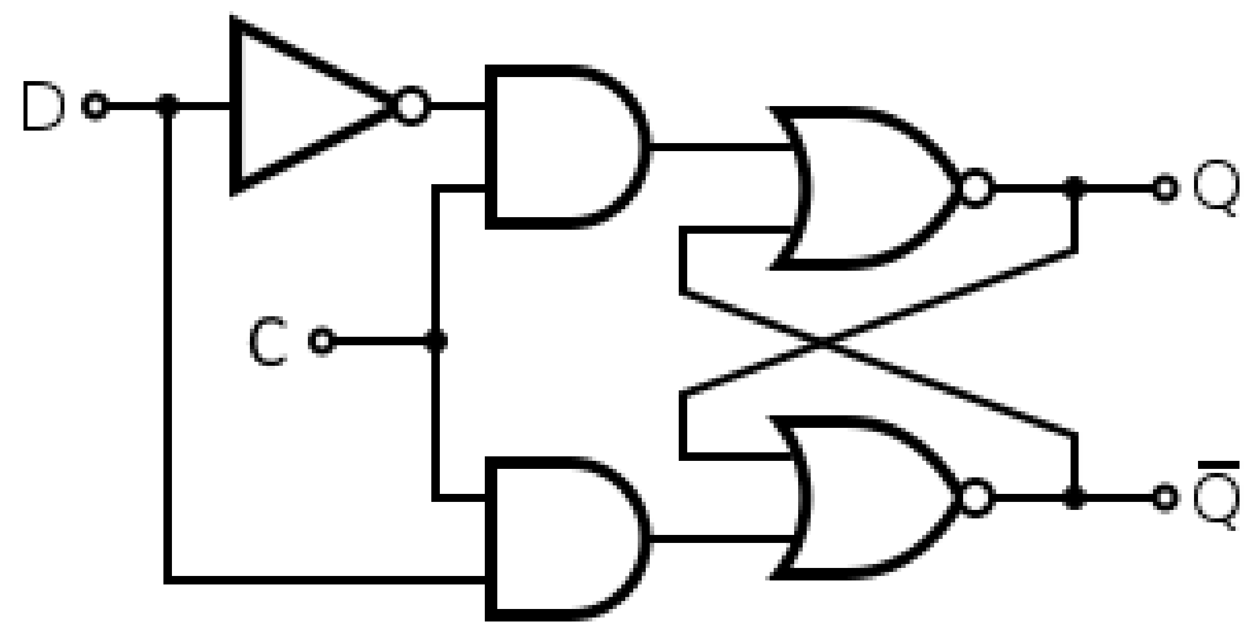


Рис. 1.13. Схема D-тригера

В таблиці 1.4 наведено стани *D*-тригера.

Таблиця 1.4

Таблиця переходів *D*-тригера.

D	*C*	*Q*	*!Q*
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

JK-тригер (рис 1.14) – працює так само як *SR*-тригер, з одним лише винятком: при подачі логічної одиниці на обидва входи *J* і *K* стан виходу тригера змінюється на протилежний, тобто виконується операція інверсії. Вхід *J* аналогічний входу *S* у *SR*-тригера. Вхід *K* аналогічний входу *R* у *SR*-тригера. При подачі одиниці на вхід *J* та нуля на вхід *K* вихідний стан тригера стає рівним логічній одиниці. А при подачі одиниці на вхід *K* та нуля на вхід *J* вихідний стан тригера стає рівним логічному нулю. *JK*-тригер на відміну від *RS*-тригера не має заборонених станів на основних входах.

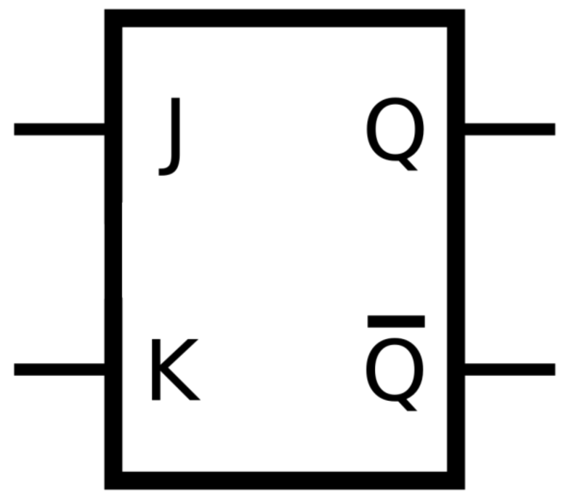


Рис. 1.14. *JK* тригер

В таблиці 1.5 наведено стани *JK*-тригера.

Таблиця 1.5

Таблиця переходів *JK*-тригера.

J	K	$Q_{(t)}$	$Q_{(t+1)}$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

1.2. Аналіз існуючих засобів створення комбінаційних схем

На сьогоднішній день існує багато систем створення та моделювання комбінаційних схем. В цьому підрозділі наведено опис та аналіз найбільш популярних програмних засобів у цій сфері.

1.2.1 *Altera Quartus II*

Altera Quartus II (рис 1.15) – це середовище компанії *Altera* для програмування та розробки під ПЛІС. *Quartus II* дозволяє аналізувати і синтезувати *HDL* конструкції, що дозволяє розробнику складати свої проекти, виконувати часовий аналіз, тестувати *RTL* діаграми, імітує реакцію дизайну на різні подразники, і налаштувати цільовий пристрій на програміста. *Quartus* включає в себе реалізацію *VHDL* та *Verilog* для опису апаратного забезпечення, візуального редагування логічних схем та моделювання векторних сигналів.

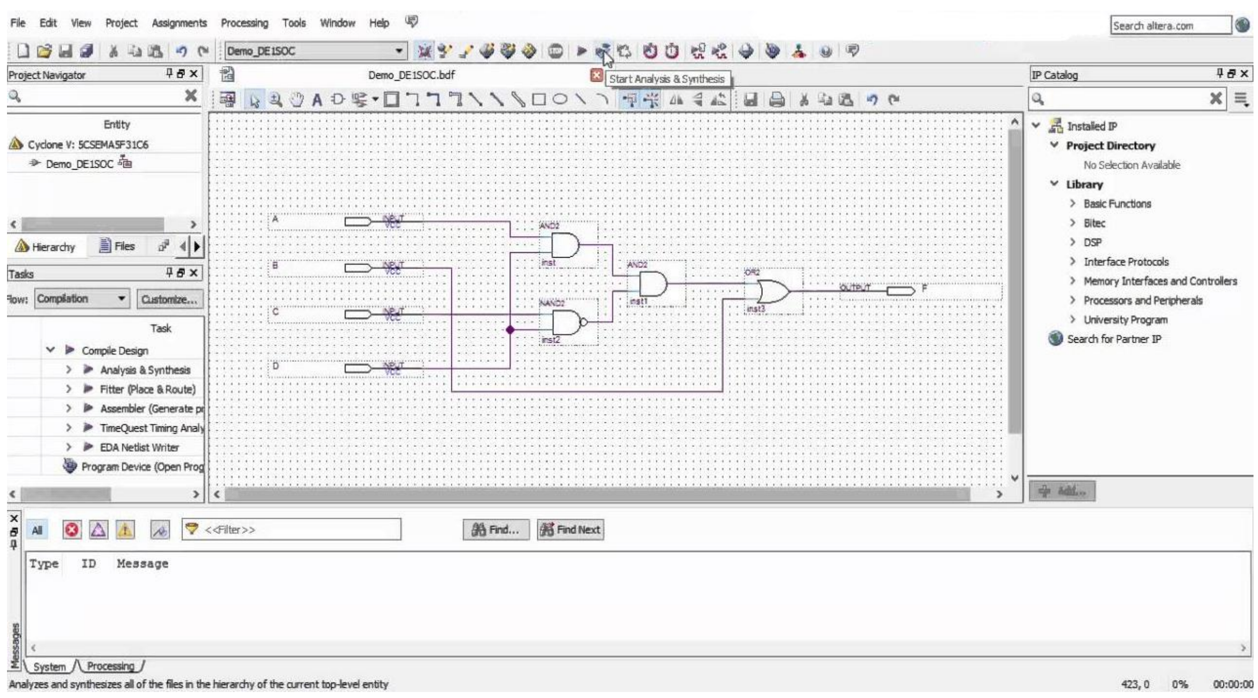


Рис. 1.15. Інтерфейс додатку *Altera Quartus II*

1.2.2. *ModelSim*

ModelSim (рис 1.16) – багатомовне середовище опису та моделювання електронного обладнання за допомогою *Mentor Graphics*, *VHDL*, *Verilog* і *SystemC*. Забезпечує розробку описів алгоритмів роботи цифрових пристроїв, може бути

підключене до систем проектування на ПЛІС. Середовище включає потужний вбудований налагоджувач для *SystemC*.

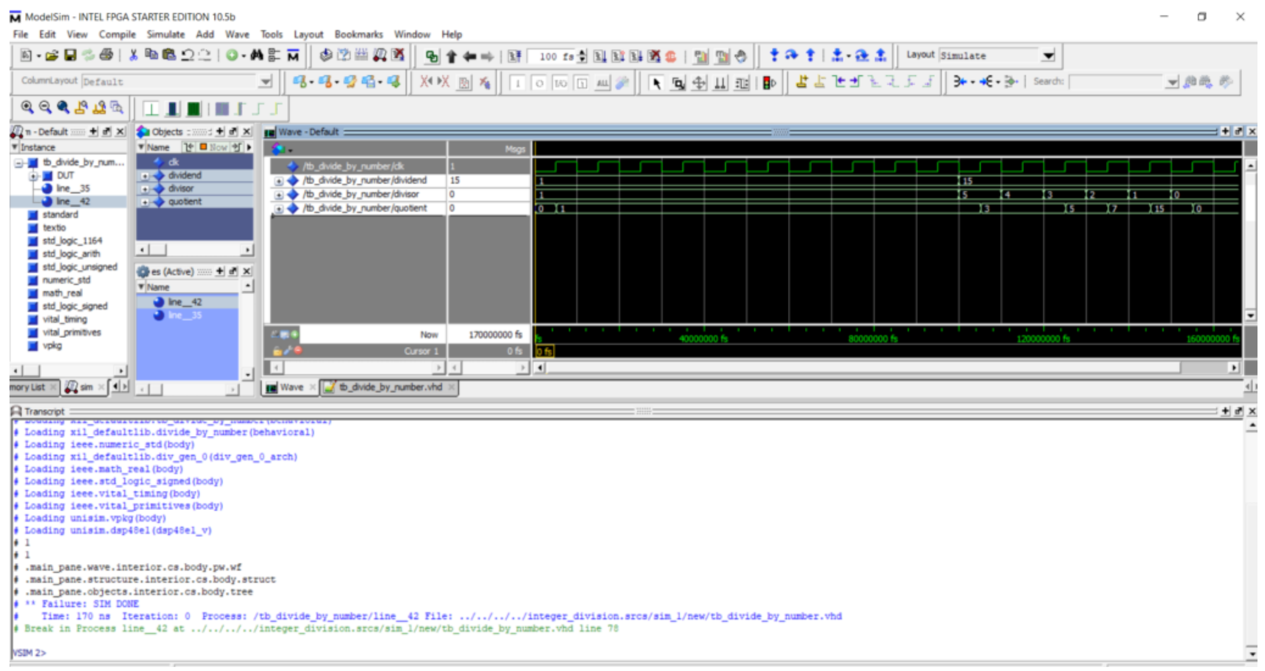


Рис. 1.16. Інтерфейс додатку *ModelSim*

1.2.3. *Xilinx ISE*

Xilinx ISE (рис 1.17)– це програмний інструмент від *Xilinx* для синтезу та аналізу проектів *HDL*, головним чином націлений на розробку вбудованого програмного забезпечення для сімейств продуктів *Xilinx FPGA* та *CPLD* інтегральних схем (ІС). Його наступником став *Xilinx Vivado*. Використання останнього випущеного видання з жовтня 2013 року продовжується для внутрішньосистемного програмування застарілих конструкцій апаратного забезпечення, що містить старіші ПЛІС і *CPLD*.

ISE дозволяє розробнику синтезувати ("компілювати") свої конструкції, виконувати аналіз часу, вивчати діаграми *RTL*, моделювати реакцію конструкції на різні вхідні значення.

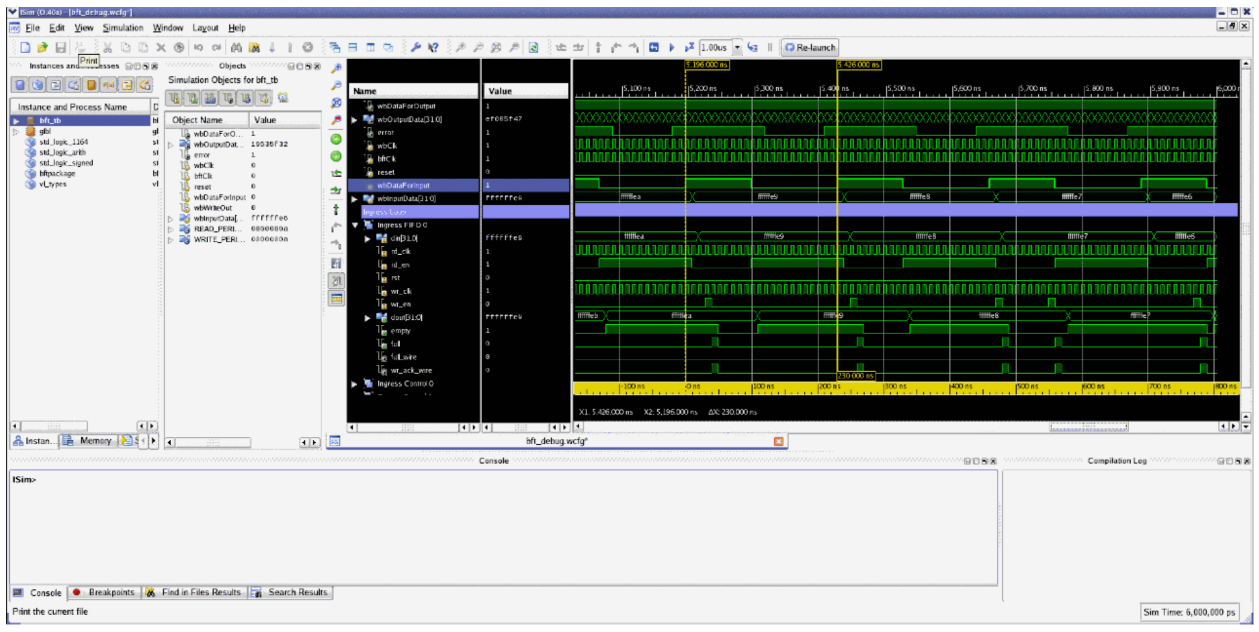


Рис. 1.17. Інтерфейс додатку *Xilinx ISE*

1.2.4. *AFDK*

AFDK (рис 1.18) – програма моделювання логічних схем. Ця програма дозволяє створювати різні об'єкти для комбінаційних схем, з урахуванням затримок та тимчасових відхилень, починаючи від простих логічних елементів та закінчуючи складними елементами пам'яті, а також макросами (макрос – це спроектована та налагоджена логічна схема, укладена в корпус). Дозволяє проектувати (малювати) та моделювати в синхронному та асинхронному режимах логічні схеми.

Він дозволяє створювати та редагувати логічні схеми, здійснювати моделювання їх роботи у синхронному (без урахування затримок сигналів у елементах схеми) та асинхронному (з урахуванням затримок) режимах, а також зберігати отримані моделі у вигляді файлів на дисках.

Комплекс включає систему підказок, що полегшує роботу в різних режимах моделювання. Для роботи з програмою використовується система ієрархічних меню.

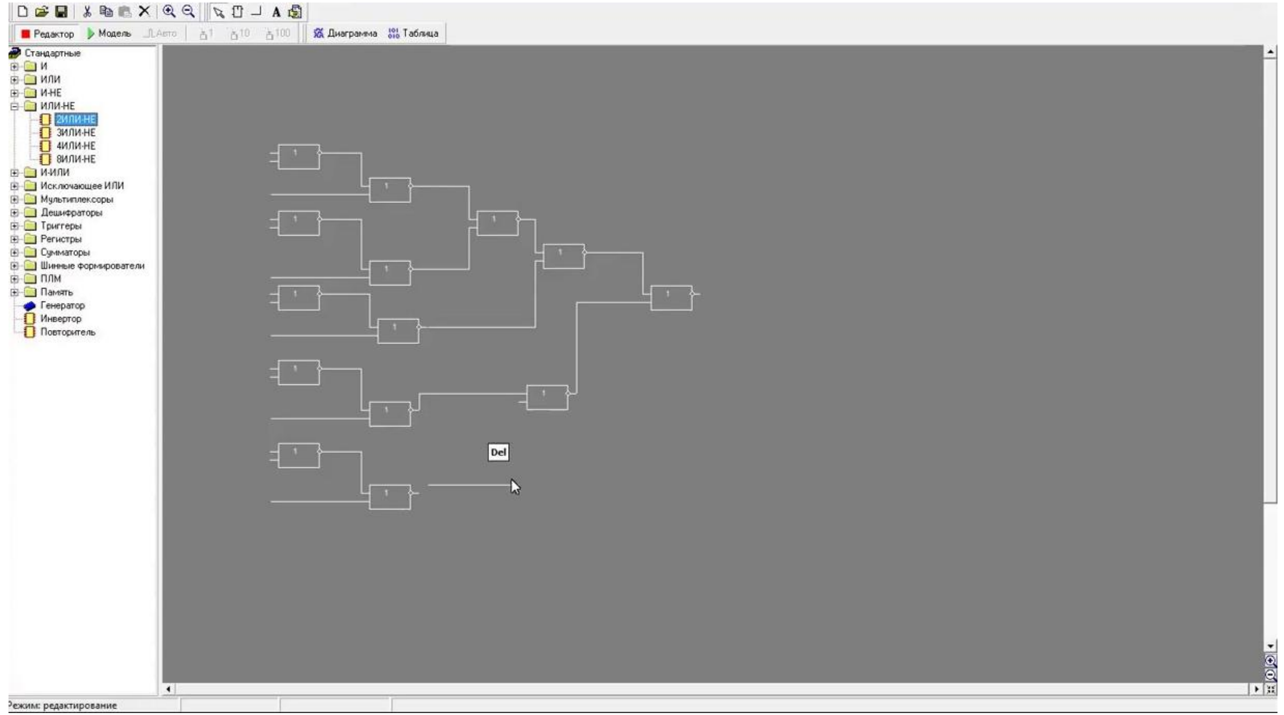


Рис. 1.18. Інтерфейс додатку *AFDK*

1.3. Аналіз недоліків існуючих рішень

Проаналізувавши реалізації існуючих рішень, можна виділити декілька загальних недоліків більшості реалізацій:

– проблема кросплатформенності;

– застарілість ПЗ та відсутність його підтримки.

Проблема кросплатформенності наявна у всіх проаналізованих застосунках та виявляється у тому, що всі вони *Windows* та *Linux* базовані та не підтримують *Web*-версії.

Застарілість ПЗ та відсутність його підтримки стосується перш за все такого додатку як *AFDK*, що робить роботу з ним досить складною адже існує багато дефектів в цій програмі, наприклад неправильне відображення елементів схема та невірне моделювання часової діаграми.

1.4. Постановка задачі

Провівши аналіз предметної області та існуючих програмних засобів створення та моделювання комбінаційних схем, було сформовано наступні вимоги до системи: програмний засіб моделювання комбінаційних схем повинен бути розроблений як *WEB*-базована система з клієнт-серверною архітектурою.

Можна виділити наступні функціональні можливості такої системи на клієнтській стороні:

- реєстрація та авторизація користувачів, доступ до власних репозиторіїв схем;
- створення та моделювання комбінаційних схем;
- можливість перегляду репозиторіїв інших користувачів;
- моделювання та перегляд часових діаграм;
- моделювання та перегляд таблиці істинності.

Також при розробці програмної системи слід приділити особливу увагу до того, щоб інтерфейс такого застосунку був максимально простим та інтуїтивним у використанні.

Серверна частина повинна відповідати наступним вимогам:

- відкритість;
- масштабованість;
- захищеність та безпечність системи.

Вимоги до бази даних:

- незалежність даних;
- безпека даних;
- висока швидкодія запитів до бази даних.

Відмовостійкість – система повинна в будь-який момент відповідати на запити користувача. Незалежно від стану системи, моделювання та графічне відображення схем повинно працювати коректно в будь-який момент часу.

Програмний засіб моделювання комбінаційних схем є комплексною структурою, яка складається з трьох основних компонент:

– клієнтський додаток — *JavaScript*, *HTML*, *CSS*;
– *web*-сервер — *C#*, *ASP.NET Core*, *Blazor*, *Entity Framework*;
– база даних — *PostgreSQL.*

1.5. Висновки до розділу

В даному розділі була проаналізована предметна область кваліфікаційної роботи, а саме існуючі методи побудови та моделювання комбінаційних схем. Були розглянуті переваги та недоліки найбільш популярних на сьогодні рішень:

– *Altera Quartus II*;
– *ModelSim*;
– *Xilinx ISE*;
– *AFDK*.

Також слід навести недоліки, що були виявлені в цих системах:

– проблема кросплатформенності;
– проблема застарілість ПЗ та відсутність його підтримки;
– дорожнеча програмного забезпечення та його обслуговування.

На основі наведених вище переваг та недоліків було сформульовано постановку задачі для розроблюваного додатку моделювання комбінаційних схем, функціональні та архітектурні вимоги.

РОЗДІЛ 2
ВИКОРИСТАНІ ТЕХНОЛОГІЇ ТА ПРОЄКТУВАННЯ

2.1. Визначення необхідний технологій для розробки

Програмний засіб моделювання комбінаційних схем є комплексною структурою, яка складається з трьох основних компонент:

– клієнтський додаток — *JavaScript*, *HTML*, *CSS*;
– *web*-сервер — *C#*, *ASP.NET Core*, *Blazor*, *Entity Framework*;
– база даних — *PostgreSQL.*

2.1.1. Протокол *HTTP*

Програмний модуль *Web*-сервера обробки даних повинен працювати по протоколу *HTTP* для спілкування з мобільним додатком.

Hyper Text Transfer Protocol – протокол для гіпермедійних систем, який знаходиться на рівні програмного забезпечення.

Версія *HTTP*-0.9 була протоколом передачі простих даних через канал зв'язку. Версія *HTTP*-1.0 вже покращила протокол, створивши формат повідомлень типу *Multipurpose Internet Mail Extension*, що містить інформацію про відправленні дані.

HTTP — це протокол для отримання ресурсів, наприклад документів *HTML*. Він є основою будь-якого обміну даними в Інтернеті та є протоколом клієнт-сервер, що означає, що запити ініціюються одержувачем, зазвичай веб-браузером. Повний документ реконструюється з різних отриманих піддокументів, наприклад, тексту, опису макета, зображень, відео, сценаріїв тощо. Клієнти та сервери спілкуються шляхом обміну окремими повідомленнями (на відміну від потоку даних).

Протокол *HTTP* здійснюється обмін ресурсами сервера та клієнта через інтернет. Клієнтські пристрої надсилають запити на сервери щодо ресурсів, необхідних для завантаження веб-сторінки після чого сервери надсилають відповіді клієнту. Запити та відповіді обмінюються піддокументами - такими як дані про зображення, текст, текстові макети тощо – які з'єднуються клієнтським веб-браузером для відображення певної веб-сторінки.

Кожен *HTTP*-запит містить закодовані дані з такою інформацією, як:

- конкретна версія *HTTP*;
- *URL*-адреса, що вказує на веб-ресурс;
- метод *HTTP* – вказує на тип дії (створення, видалення, відправка, тощо), яку сервер повинен виконати;
- заголовки *HTTP*-запитів – включають в себе тип браузера, який використовується та які дані запит очікує від сервера;
- тіло *HTTP* – необов'язкова інформація, яка потрібна серверу. Тіло може включати в себе будь-яку інформацію, наприклад логіни для входу, файли, тощо.

Відповіді *HTTP* зазвичай містять такі дані:

– код стану *HTTP* відповіді – вказує на стан запиту до клієнтського пристрою. Можуть свідчити про успіх виконання запиту або про помилки на стороні сервера чи клієнта;

– заголовки відповіді *HTTP* – включають в себе інформацію про сервер та запитувані ресурси.

2.1.2. Структура *Web*-серверу

Мова розмітки гіпертексту (*HTML*) - це основний мовний стандарт, що використовується для впорядкування та форматування веб-сторінок та інших документів у Всесвітній павутині. Вона часто використовується в поєднанні з каскадними таблицями стилів (*CSS*) та *JavaScript* для створення повністю

адаптивних веб-сторінок, які відображаються правильно на різних розмірах екранів.

HTML визначає, які частини тексту є абзацами, заголовками, гіперпосиланнями, а *CSS* визначає, як ці частини візуально виглядають. *JavaScript*, навпаки, додає на сторінку динамічні елементи, такі як спливаючі вікна, анімована графіка, прокрутка банерів, тощо.

*HTML*5 – найновіша версія *HTML*. Цей термін стосується двох речей. Одним з них є сама оновлена мова *HTML*, яка має нові елементи та атрибути. Другий – це більший набір технологій, які працюють із цією новою версією *HTML* – як із новим форматом відео – і дозволяють створювати складніші та потужніші веб-сайти та програми.

Більшість веб-сторінок сьогодні створено за допомогою *HTML*4. Незважаючи на те, що з часу написання першої версії *HTML* у 1993 році *HTML*4 значно покращився, *HTML*4 усе ще мав свої обмеження. Його найбільшим було те, що коли веб-розробники хотіли додати на свій сайт вміст або функції, які не підтримувалися в *HTML*, їм потрібно було використовувати нестандартні технології, такі як *Adobe Flash*, які вимагали від користувачів встановлення плагінів браузера.

*HTML*5 було розроблено, щоб усунути потребу в цих нестандартних технологіях. За допомогою цієї нової версії *HTML* можна створювати веб-програми, які працюють в автономному режимі, підтримують відео та анімацію високої чіткості та отримують дані о геолокації.

Каскадні таблиці стилів (*CSS*) забезпечують центральне розташування інформація про те, як різні шрифти, кольори переднього плану, кольори фону та ін. слід застосовувати до різних елементів *HTML* на веб-сторінці.

CSS розшифровується як каскадна таблиця стилів. Його головна мета – надати веб-сторінці стиль і моду. *CSS* надає властивості кольору, макета, фону, шрифту та кордону. Функції *CSS* забезпечують кращу доступність вмісту, покращену гнучкість і контроль, а також специфікацію характеристик представлення.

*CSS*3 розшифровується як *Cascading Style Sheet level* 3, яка є розширеною версією *CSS*. Він використовується для структурування, стилізації та форматування веб-сторінок. До *CSS3* додано кілька нових функцій, і він підтримується всіма сучасними веб-браузерами. Найважливішою особливістю *CSS*3 є поділ стандартів *CSS* на окремі модулі, які простіше вивчати та використовувати.

Деякі з основних модулів *CSS*3:

– коробкова модель;
– текстові ефекти;
– селектори;
– фони та рамки;
– анімації;
– інтерфейс користувача (*UI*);
– розмітка кількох стовпців;
– 2*D*/3*D* трансформації.

Javascript (*JS*) - це мова сценаріїв, яка в основному використовується в для розробки веб-додатків. Вона використовується для вдосконалення *HTML*-сторінок і зазвичай вже вбудована у *HTML*-код.

JavaScript може відображати веб-сторінки інтерактивно та динамічно незалежно від веб-сервера. Це дозволяє сторінкам реагувати на події, демонструвати спеціальні ефекти, перевіряти дані на стороні клієнта, створювати файли *cookie*, визначати браузер користувача тощо.

Початкові версії мови сценаріїв були лише для внутрішнього використання. Після того, як *Netscape* подав його в *ECMA International* як стандартну специфікацію для веб-браузерів, *JavaScript* став піонером у випуску *ECMAScript*.

Відтоді *JavaScript* продовжував розвиватися разом із новими браузерами, такими як *Mozilla Firefox* і *Google Chrome*. Останній навіть почав розробляти перший сучасний двигун *JavaScript* під назвою *V*8, який компілює байт-код у рідний машинний код.

Сьогодні *JavaScript* має багато фреймворків і бібліотек для спрощення складних проектів, таких як *AngularJS*, *jQuery* та *ReactJS*. Кожен фреймворк *JavaScript* має функції, спрямовані на спрощення процесу розробки та налагодження.

Наприклад, зовнішні фреймворки *JavaScript*, такі як *jQuery* та *ReactJS*, покращують ефективність проектування. Вони дозволяють розробникам повторно використовувати й оновлювати компоненти коду, не впливаючи один на одного.

2.1.3. Мова програмування *C#*

C# є похідною мовою від мови програмування *C* і подібна до мови *C++*. *C#* використовує ті ж самі оператори, що і *C++*, є об'єктно-орієнтованим, чутливим до регістру та має майже однаковий синтаксис як у *C++*.

Але найважливішою відмінністю *C#* від *C++* є те, що програма написана на мові *C#* працює в керованому середовищі виконання - це означає, що програма не має прямого доступу до пам'яті і їй не потрібно стежити за нею, створювати або видаляти об'єкти. Це дозволяє уникнути помилок звернень до пам'яті та своєчасної очистки від сміття.

Програми *C#* виконуються на *.NET*, віртуальній системі виконання, що називається загальномовним середовищем виконання (*CLR*), і наборі бібліотек класів. *CLR* – це реалізація Microsoft спільної мовної інфраструктури (*CLI*), міжнародного стандарту. *CLI* є основою для створення середовищ виконання та розробки, у яких мови та бібліотеки безперебійно працюють разом.

Вихідний код, написаний мовою *C#*, компілюється в проміжну мову (*IL*), яка відповідає специфікації *CLI*. Код *IL* і ресурси, такі як растрові зображення та рядки, зберігаються в збірці, зазвичай із розширенням *.dll*. Збірка містить маніфест, який надає інформацію про типи, версію та культуру збірки.

Коли програма *C#* виконується, збірка завантажується в *CLR*. *CLR* виконує компіляцію *Just-In-Time* (*JIT*) для перетворення коду *IL* у власні машинні

інструкції. *CLR* надає інші служби, пов’язані з автоматичним збиранням сміття, обробкою винятків і керуванням ресурсами. Код, який виконує *CLR*, іноді називають «керованим кодом». «Некерований код» компілюється в рідну машинну мову, націлену на певну платформу.

Сумісність мов є ключовою особливістю *.NET*. Код *IL*, створений компілятором *C#*, відповідає специфікації загального типу (*CTS*). Код *IL*, згенерований з *C#*, може взаємодіяти з кодом, який було згенеровано з версій *.NET F#*, *Visual Basic*, *C++*. Існує понад 20 інших *CTS*-сумісних мов. Одна збірка може містити декілька модулів, написаних різними мовами *.NET*. Типи можуть посилатися один на одного, як якщо б вони були написані однією мовою.

Тип визначає структуру та поведінку будь-яких даних у *C#*. Оголошення типу може включати його члени, базовий тип, інтерфейси, які він реалізує, та операції, дозволені для цього типу. Змінна — це мітка, яка посилається на екземпляр певного типу.

У *C#* існує два види типів: типи значень і типи посилань. Змінні типів значень безпосередньо містять свої дані. Змінні посилальних типів зберігають посилання на свої дані, останні відомі як об’єкти. За допомогою посилальних типів дві змінні можуть посилатися на один і той самий об’єкт, а операції над однією змінною можуть впливати на об’єкт, на який посилається інша змінна.

Ідентифікатор – це ім'я змінної. Ідентифікатор – це послідовність символів *Unicode* без пробілів. Ідентифікатор може бути зарезервованим словом *C#*, якщо воно має префікс @. Використання зарезервованого слова як ідентифікатора може бути корисним під час взаємодії з іншими мовами.

Типи значень *C#* далі поділяються на прості типи, типи перерахувань, типи структур, типи значень із можливістю обнуління та типи значень кортежу. Посилальні типи *C#* далі поділяються на типи класів, типи інтерфейсу, типи масивів і типи делегатів.

2.1.4. *ASP.NET Core* та *ASP.NET Core MVC*

ASP.NET Core – платформа з відкритим кодом, яка призначена для створення хмарних додатків, таких як веб-додатки, *IoT* додатки та мобільні серверні системи. *ASP.NET Core* призначений для роботи як у хмарних сервісах, так і в локальній мережі.

ASP.NET Core – це нова міжплатформна платформа з відкритим вихідним кодом для створення сучасних хмарних додатків, підключених до Інтернету, таких як веб-програми, програми *IoT* і мобільні серверні програми. Програми *ASP.NET Core* можуть працювати на *.NET Core* або на повній версії *.NET Framework*. Він був розроблений, щоб забезпечити оптимізовану структуру розробки для програм, які розгортаються в хмарі або виконуються локально. Він складається з модульних компонентів з мінімальними накладними витратами, тож ви зберігаєте гнучкість під час створення своїх рішень.

ASP.NET Core працює поверх протоколу *HTTP* і використовує команди та політики *HTTP* для встановлення двостороннього зв'язку між браузером і сервером. *ASP.NET Core* дозволяє виконувати і обробляти будь-які запити, завантажувати і зберігати файли, спілкуватися з базою даних, спілкуватися з іншими веб-додатками. *ASP.NET Core* є основою, яку можна розширювати різним функціоналом, в тому числі і власноручно створеним.

На рис. 2.1 детально зображено життєвий цикл запиту в *ASP.NET Core.*

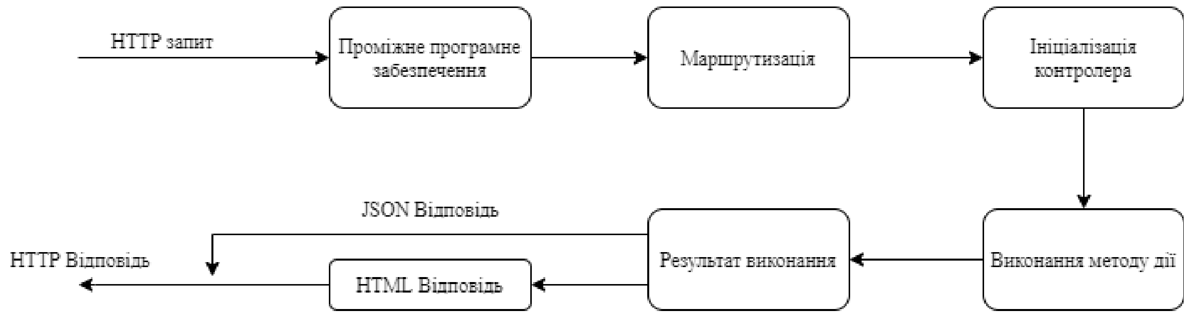


Рис. 2.1. Життєвий цикл *HTTP* запиту в *ASP.NET Core*

ASP.NET Core MVC - це фреймворк для створення веб-додатків та *API* за допомогою шаблону проєктування *Model-View-Controller*.

Архітектурний шаблон *Model-View-Controller* (*MVC*) розділяє додаток на три основні групи компонентів: моделі, представлення та контролери.

За допомогою цього шаблону запити користувачів направляються до контролера, який відповідає за роботу з моделлю даних для виконання дій користувача. Контролер обирає необхідне представлення (*HTML* файл), яке потрібно відобразити користувачеві і надає йому потрібну модель даних.

Модель – представляє стан програми та будь-яку бізнес-логіку або операції, які вона повинна виконувати.

Представлення – відповідають за представлення даних через інтерфейс користувача.

Контролери – компоненти, які обробляють запити користувача, працюють із моделлю та, зрештою, вибирають подання для візуалізації.

Служба – це компонент, призначений для загального використання в програмі. Послуги стають доступними через впровадження залежностей. *ASP.NET Core* містить простий вбудований контейнер інверсії керування (*IoC*), який за замовчуванням підтримує ін’єкцію конструктора, але його можна легко замінити на обраний контейнер *IoC*. На додаток до переваг слабкого зв’язку, *DI* робить сервіси доступними у програмі.

Модель хостингу *ASP.NET Core* не прослуховує запити безпосередньо; скоріше він покладається на реалізацію *HTTP*-сервера для пересилання запиту до програми. Пересланий запит загортається як набір інтерфейсів функцій, які програма потім компонує в *HttpContext*. *ASP.NET* Core містить керований кросплатформний веб-сервер під назвою *Kestrel*, який зазвичай запускається за робочим веб-сервером, таким як *IIS* або *nginx*.

2.1.5. Blazor

Blazor – безкоштовна веб-платформа з відкритим вихідним кодом, що дозволяє розробникам створювати веб-програми з використанням *C#* та *HTML*. Додаток *Blazor* може взаємодіяти з *JavaScript* (причому вони працюють на стороні клієнта), наприклад, викликати (повторно використовувати) функції *JavaScript* з *.NET* методів.

Blazor розділяє те, як він обчислює зміни інтерфейсу користувача (модель програми/компонента) і те, як ці зміни застосовуються (рендерер). Це відрізняє Blazor від інших фреймворків інтерфейсу користувача, таких як *Angular* або *ReactJS*/*React Native*, які можуть створювати лише інтерфейси на основі веб-технологій. Використовуючи різні засоби візуалізації, *Blazor* може створювати не лише веб-додатки, але й мобільні інтерфейси.

За своєю суттю, модель додатка/компонента *Blazor* відповідає за обчислення змін інтерфейсу користувача, також є можливість використовувати різні рендерери, щоб контролювати, як інтерфейс користувача насправді відображається та оновлюється. Ці рендерери частіше називаються моделями хостингу.

Модель хостингу *Blazor Server* на даний момент є єдиним підтримуваним варіантом для розробки *Blazor*. Він був випущений ще у вересні 2019 року з *.NET Core* 3.

Програми *Blazor* засновані на компонентах. Компонент у *Blazor* — це елемент інтерфейсу користувача, такий як сторінка, діалогове вікно або форма введення даних. Клас компонента зазвичай записується у формі сторінки розмітки *Razor* із розширенням файлу *.razor*. Компоненти *Blazor* формально називаються компонентами *Razor*, неофіційно компонентами *Blazor*.

Razor — це синтаксис для поєднання розмітки *HTML* із кодом *C#*, призначений для підвищення продуктивності розробника. Razor дозволяє перемикатися між розміткою *HTML* і *C#* в одному файлі за допомогою підтримки

програмування *IntelliSense* у *Visual Studio*. *Razor Pages* і *MVC* також використовують *Razor*.

На відміну від *Razor Pages* і *MVC*, які побудовані навколо моделі запит/відповідь, компоненти використовуються спеціально для логіки та композиції інтерфейсу користувача на стороні клієнта.

Blazor Server забезпечує підтримку розміщення компонентів Razor на сервері в програмі *ASP.NET Core*. Оновлення інтерфейсу користувача обробляються через підключення *SignalR*.

З’єднання, яке використовує *Blazor Server* для зв’язку з браузером, також використовується для обробки викликів взаємодії *JavaScript*. Серверні програми *Blazor* відтворюють вміст інакше, ніж традиційні моделі для відтворення інтерфейсу користувача в програмах *ASP.NET Core* за допомогою переглядів *Razor* або сторінок *Razor*. Обидві моделі використовують мову *Razor* для опису вмісту *HTML* для рендерингу, але вони суттєво відрізняються способом рендерингу розмітки. Під час відтворення сторінки або перегляду *Razor* кожен рядок коду *Razor* видає *HTML* у текстовій формі. Після візуалізації сервер позбавляється екземпляра сторінки або перегляду, включаючи будь-який створений стан. Коли виникає інший запит на сторінку, вся сторінка знову відтворюється в *HTML* і надсилається клієнту.

Blazor Server створює графік компонентів для відображення, схожий на *HTML* або *XML Document Object Model* (*DOM*). Графік компонентів містить стан властивостей і полів. *Blazor* оцінює графік компонентів, щоб створити двійкове представлення розмітки, яке надсилається клієнту для рендерингу.

Після встановлення з’єднання між клієнтом і сервером статичні попередньо візуалізовані елементи компонента замінюються інтерактивними елементами. Попереднє відтворення вмісту на сервері робить програму більш чуйною на клієнті.

2.1.6. СУБД та *Entity framework*

В даному проєктуванні була обрана СУБД *PostgreSQL*. *PostgreSQL* — об'єктно-реляційна система керування базами даних (СКБД). Є альтернативою як комерційним СКБД (*Oracle Database*, *Microsoft SQL Server*, *IBM DB2* та інші), так і СКБД з відкритим кодом (*MySQL*, *Firebird*, *SQLite*).

Порівняно з іншими проектами з відкритим кодом, такими як *Apache*, *FreeBSD* або *MySQL*, *PostgreSQL* не контролюється якоюсь однією компанією, її розробка можлива завдяки співпраці багатьох людей та компаній, які хочуть використовувати цю СКБД та впроваджувати у неї найновіші досягнення. Сервер *PostgreSQL* написаний на мові *C*. Зазвичай розповсюджується у вигляді набору текстових файлів із сирцевим кодом. Для інсталяції необхідно відкомпілювати файли на своєму комп'ютері і скопіювати в деякий каталог.

У реалізації веб-сервера є вимога уникнути використання і написання прямих запитів мови *SQL*, такі як оператори *SELECT*, *CREATE*, *DELETE*. Для цього потрібно використовувати технологію *Entity framework.*

Entity Framework (*EF*) - це об'єктно-реляційне відображення (*ORM*) з відкритим вихідним кодом (рис. 2.2).

Entity Framework дозволяє розробникам працювати з даними у вигляді об'єктів та властивостей, що стосуються домену, таких як клієнти та адреси клієнтів, тощо, без необхідності працювати з реальними таблицями та стовпцями в базі даних. За допомогою *Entity Framework* розробники можуть працювати на вищому рівні абстракції, коли мають справу лише з даними.

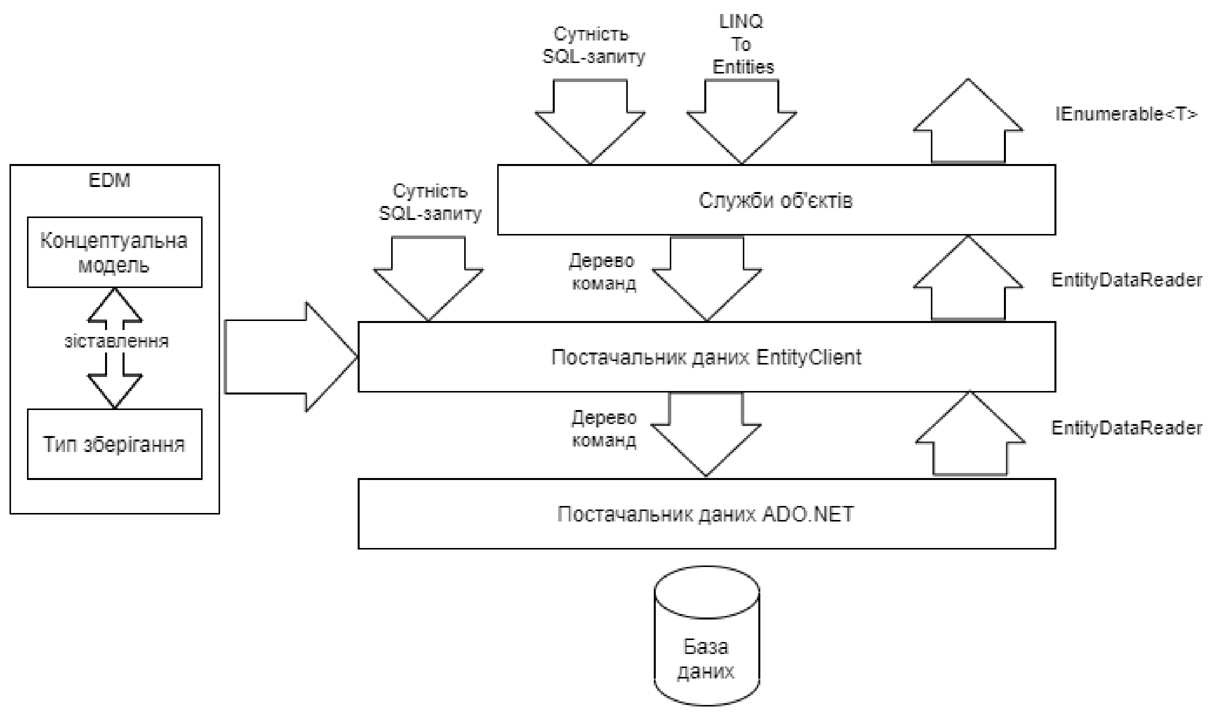


Рис. 2.2. Архітектура *Entity framework*

2.2. Проєктування функціональної частини системи

Програмний додаток моделювання комбінаційних схема безпекою складається з трьох основних компонентів:

– клієнтський додаток — *JavaScript*, *HTML*, *CSS*;
– *web*-сервер — *C#*, *ASP.NET Core*, *Blazor*, *Entity Framework*;
– база даних — *PostgreSQL.*

2.2.1 Проєктування бази даних

База даних представляє собою головний ресурс всієї необхідної інформації для розробки та функціонування програмного забезпечення. База даних для розроблюваної системи представлена у вигляді реляційної моделі, тобто основною сутністю такої моделі є відношення між таблицями.

Була сформована *UML*-діаграма, яка включає в себе всі об’єкти системи та відношення між ними (рис. 2.3).

В таблиці 2.1 наведено опис всіх сутностей бази даних.

Таблиця 2.1

Описання таблиць бази даних

Назва таблиці	Опис таблиці
users	Список користувачів системи
usersschemarepo	Список репозиторіїв користувача
userreposhare	Список репозиторіїв, які можуть бути доступні іншим користувачам
schema	Список схем та їхня логіка
schemahistory	Історія результатів моделювання схем

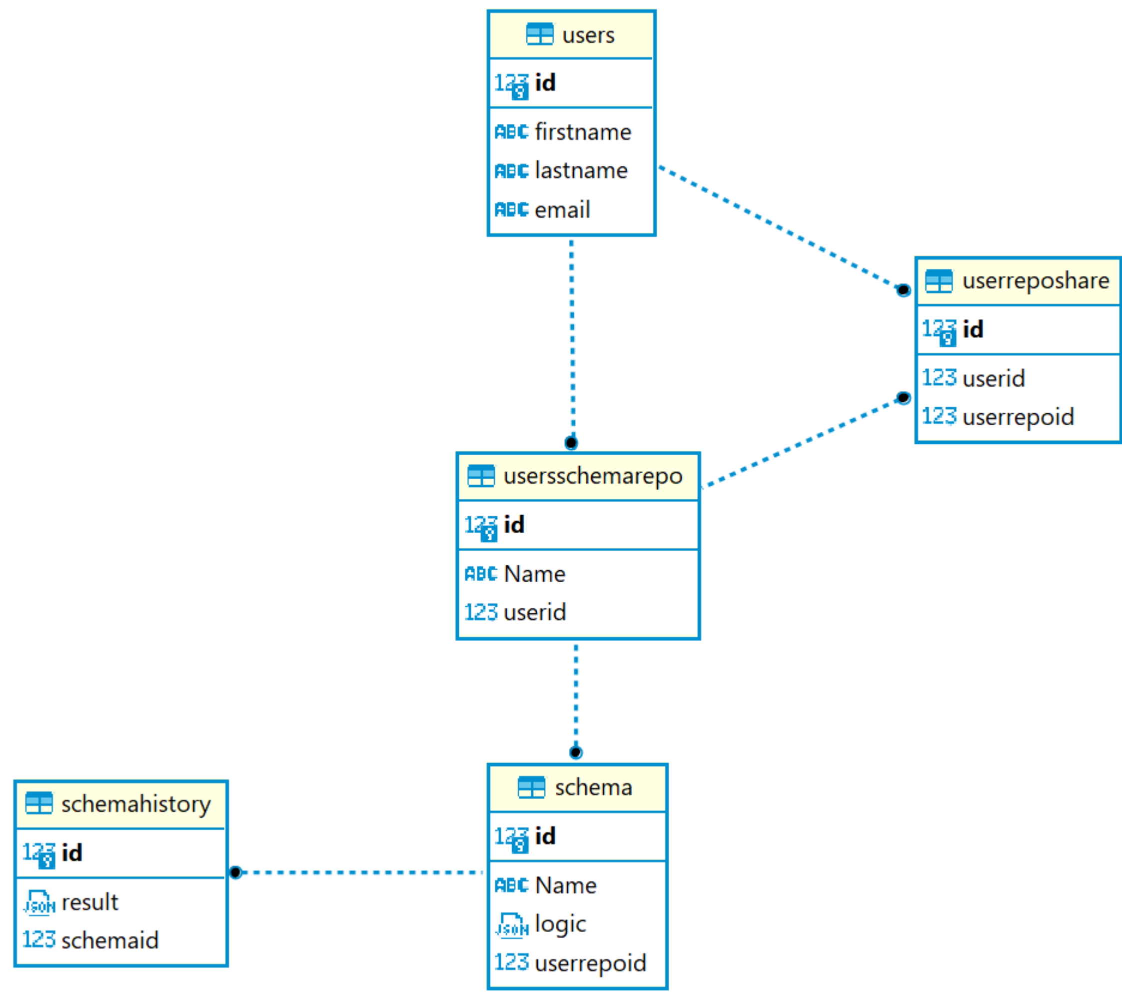


Рис. 2.3. *UML*-діаграма бази даних

2.2.2. Архітектура клієнтського додатку

Клієнтський додаток повинен в першу чергу відображати інтерфейс користувача, за допомогою технології *HTML* та *CSS*. Основною частиною клієнтського додатка є конструктор комбінаційних схем, за допомогою технології *JavaScript* повинно бути створена програма для роботи конструктора, який повинен додавати, видаляти логічні елементи, зв'язувати їх між собою, моделювати часову діаграму та таблиці істинності та зберігати схему у базу даних. Додатковими частинами клієнтського додатку також є: авторизація користувача, особистий кабінет користувача та репозиторій схем.

Всі запити від клієнтського додатку до серверу обробляє та виконує технологія *Blazor* за допомогою *SignalR*, яка надсилає асинхронні сповіщення клієнту від сервера та навпаки.

SignalR надає простий *API* для створення віддалених викликів процедур (*RPC*) між серверами, які викликають функції *JavaScript* у клієнтських браузерах (та інших клієнтських платформах) із коду *.NET* на стороні сервера. *SignalR* також містить *API* для керування з'єднаннями (наприклад, події підключення та від'єднання) і групування з'єднань.

SignalR автоматично керує з'єднаннями та дозволяє транслювати повідомлення всім підключеним клієнтам одночасно, як у чаті. Також він може надсилати повідомлення певним клієнтам. З'єднання між клієнтом і сервером є постійним, на відміну від класичного *HTTP*-з'єднання, яке повторно встановлюється для кожного зв'язку.

Повна структура клієнтського додатку зображена на рис 2.4.

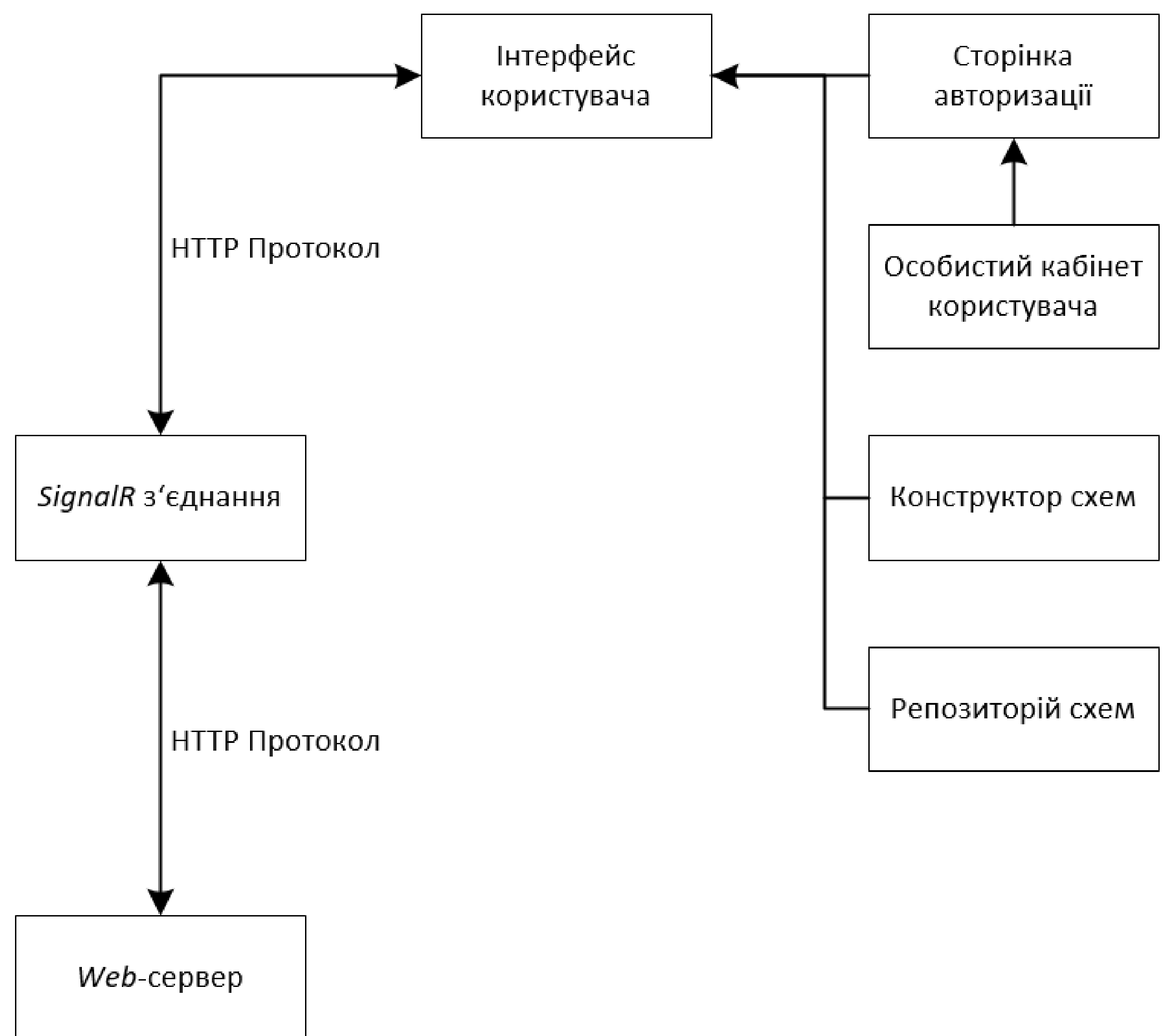


Рис. 2.4. Структура клієнтського додатку

2.2.3. Архітектура *web*-серверу

Web-сервер відповідає за маршрутизацію запитів, оброки запитів до бази даних та реагує на зміни та взаємодію користувача з інтерфейсом. Web-сервер оброблює запити всіх користувачів паралельно та незалежно один від одного, за допомогою технології *ASP .Net Core* забезпечується безпека та шифрування даних користувача для кожного запита.

Повна структура *web*-серверу зображена на рис 2.5.

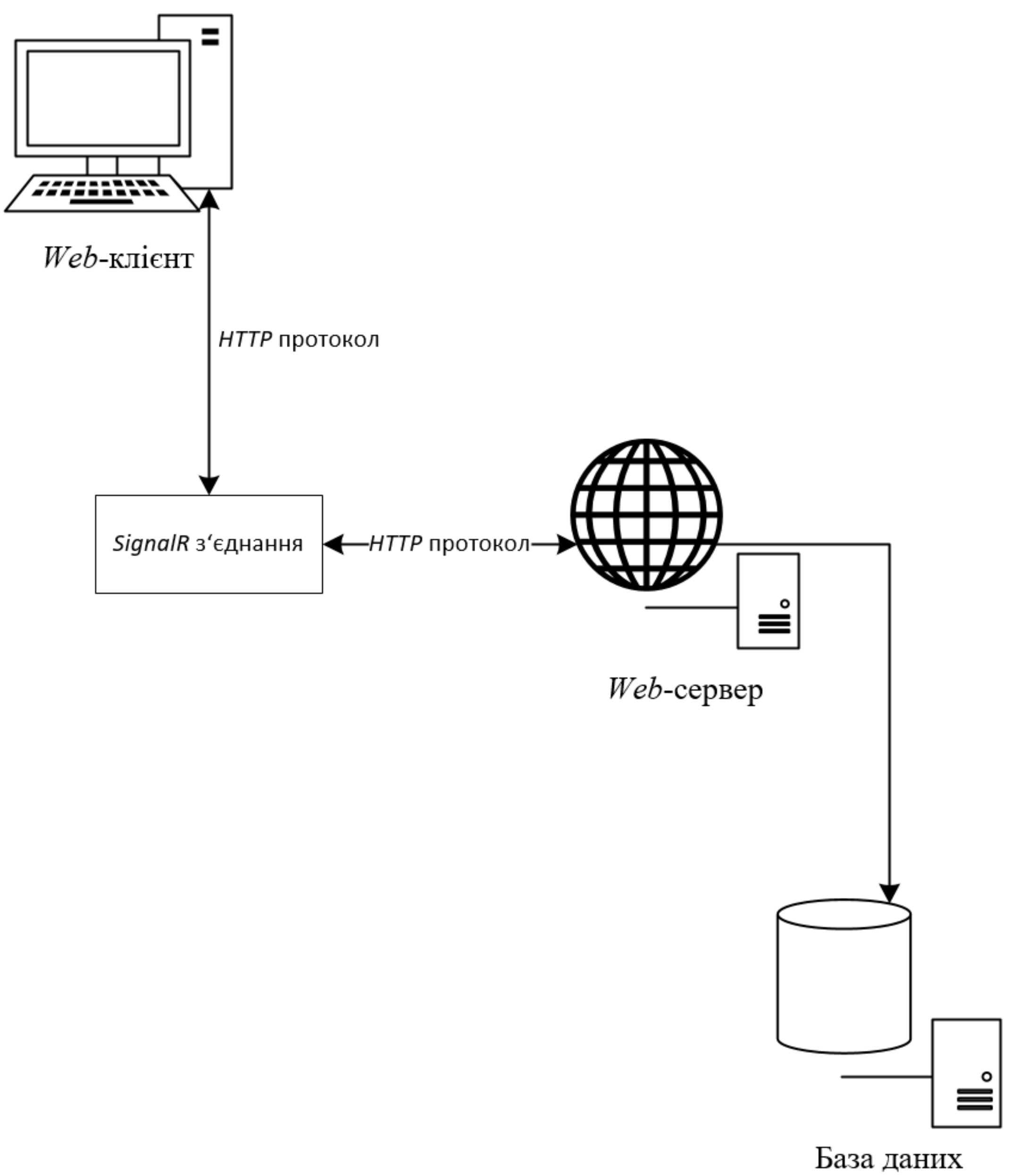


Рис. 2.5. Структура *web*-серверу

2.2.4. Діаграма станів клієнтського додатку

Клієнтський додаток працює на основі зашифрованих запитів, тому для початку роботи користувач має пройти авторизацію, якщо він вже авторизований, то немає необхідності проходити авторизацію повторно, так як користувач вже

має необхідні дані та ключі, які записуються в зашифрованому місці браузера (*Cookie*).

Як тільки користувач пройшов авторизацію, клієнт починає відправляти запити на отримання даних про вміст поточної сторінки та дані з бази. Після початкового відображення поточної сторінки, клієнт очікує на інші дії з боку клієнта, а також з боку сервера, якщо трапилась. Якщо поточний стан клієнт змінився, він відправить сигнал серверу про новий стан, якщо поточний стан сервера змінився, то сервер відправить клієнту нову інформацію для поточної сторінки.

Клієнт та сервер обмінюються даними та повідомленнями тільки за вимогою (зміна стану клієнта або серверу), це збільшує продуктивність та оптимізує роботу сервера, так як на кожного клієнта витрачається стільки ресурсів, скільки потрібно на даний момент часу. Також, це дозволяє оновлювати вміст поточної сторінки частково, а не всю сторінку повністю.

Архітектура клієнт-сервера — це обчислювальна модель, у якій сервер розміщує, надає та керує більшістю ресурсів і послуг, запитуваних клієнтом. Вона також відома як модель мережевих обчислень або мережа клієнт-сервер, оскільки всі запити та послуги доставляються через мережу. Архітектура клієнт-сервер або модель має інші системи, підключені через мережу, де ресурси розподіляються між різними комп’ютерами.

Як правило, архітектура клієнт-сервера влаштована таким чином, що клієнти часто розташовані на робочих станціях або на персональних комп’ютерах, тоді як сервери розташовані в інших місцях мережі, зазвичай на більш потужних машинах. Така модель особливо корисна, коли клієнти та сервер виконують рутинні завдання.

На рис. 2.6 приведена діаграма станів клієнтського додатку.

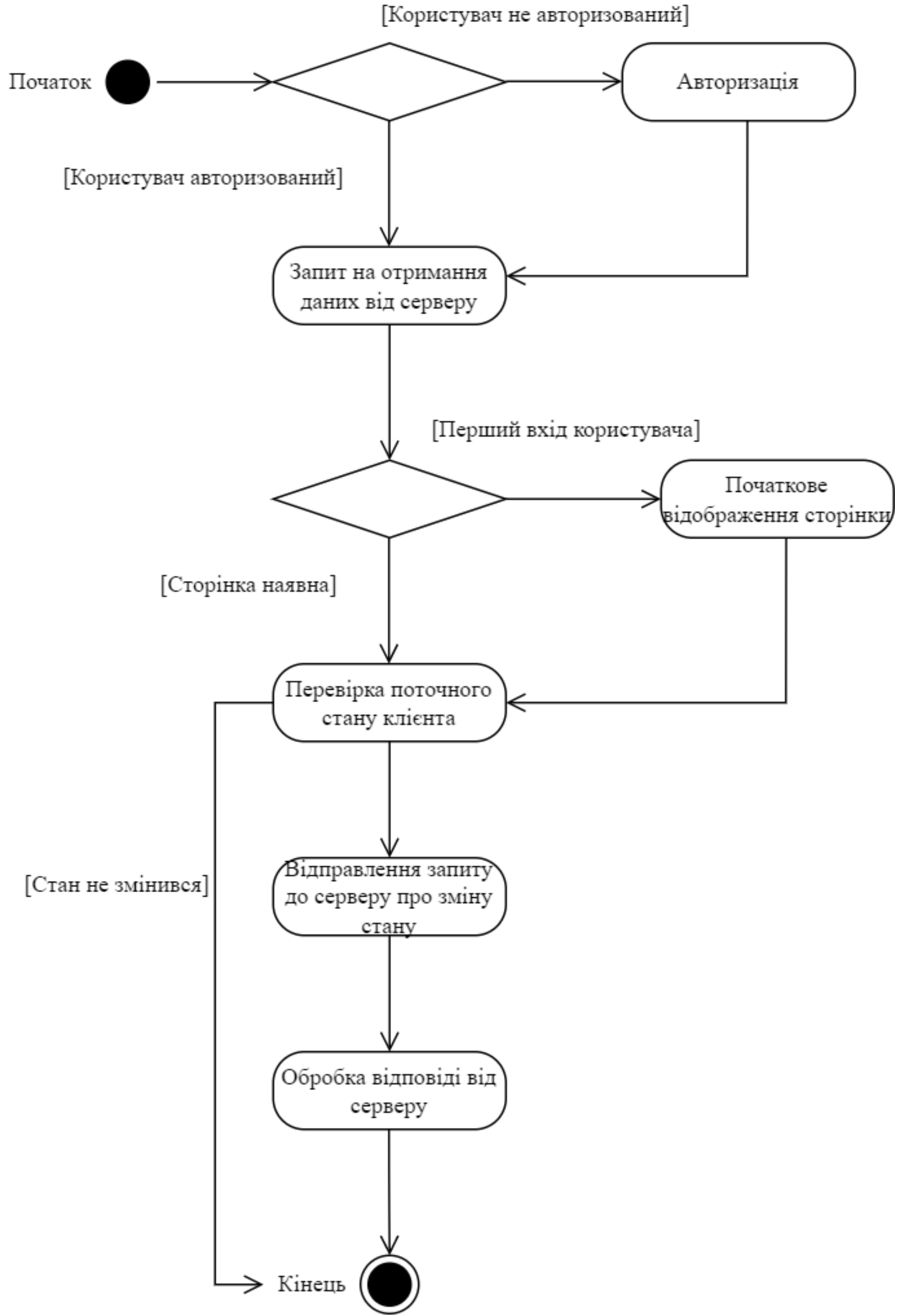


Рис. 2.5. Діаграма станів клієнтського додатку

2.2.5. Діаграма станів *web*-сервера

Web-сервер відповідає за обробку запитів від клієнтського додатку та повідомляє клієнту про зміну поточного стану зі сторони серверу. Також *web*-сервер маніпулює з базою даних, виконує запити по типу отримання та запису

даних. Клієнт та сервер обмінюються даними та повідомленнями тільки за вимогою (зміна стану клієнта або серверу), це відбувається за допомогою подій та повідомлень, які виникають на як і на стороні клієнта так і на стороні серверу, це дає можливість серверу отримати стан кожного елемента на поточній сторінці, та оновлювати лише потрібні, такий підхід оптимізує роботу сервера з великою кількістю клієнтів.

На рис. 2.6 приведена діаграма станів *web*-серверу.

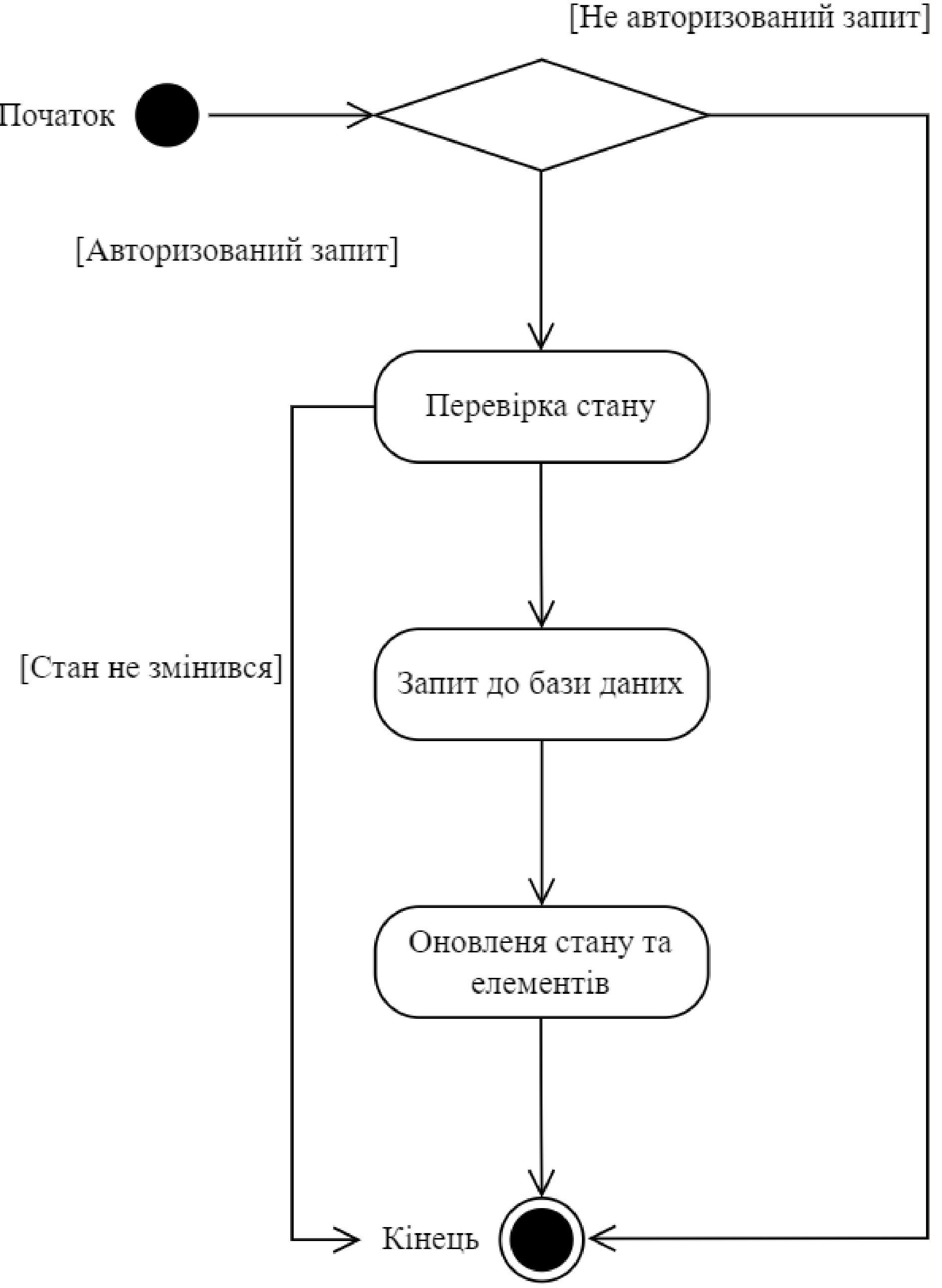


Рис. 2.6. Діаграма станів *web*-серверу

2.2.6. Діаграма взаємодії користувача з додатком

Функціонально клієнтський додаток має можливість реєстрація та авторизація користувачів, доступ до власних репозиторіїв схем, створення та моделювання комбінаційних схем, можливість перегляду репозиторіїв інших користувачів, моделювання та перегляд часових діаграм, моделювання та перегляд таблиці істинності.

Також користувач має можливість збереження створених схем та в майбутньому редагувати їх або видаляти.

На рис. 2.7 представлена *Use-Case* діаграма клієнтського додатку.

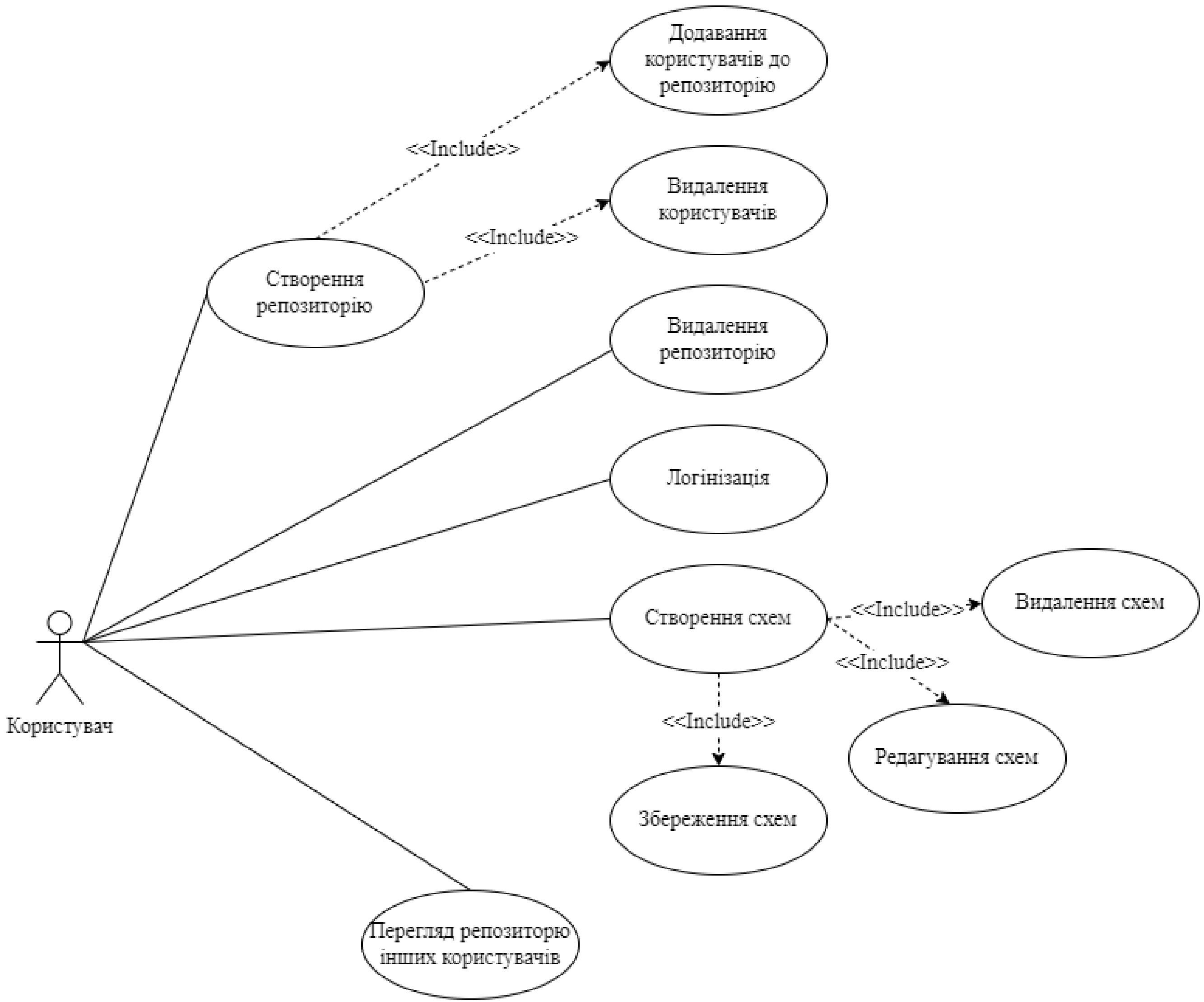


Рис. 2.7. *Use-Case* діаграма клієнтського додатку

2.3. Висновки до розділу

В даному розділі були проаналізовані та обрані всі необхідні технології для проєктування додатку.

Розроблюваний додаток складається з трьох основних елементів, для кожного з яких були визначені необхідні технології:

- клієнтський додаток — *JavaScript*, *HTML*, *CSS*;
- *web*-сервер — *C#*, *ASP.NET Core*, *Blazor*, *Entity Framework*;
- база даних — *PostgreSQL*.

Для кожного основного елемента були описані і створені діаграма станів та *use-case* діаграма.

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМНОГО ЗАСОБУ

3.1. Розробка *web*-серверу

Сервер повинен передавати дані користувачам по мережі і мати зручний інтерфейс конфігурації, для цієї мети підійшла технологія *ASP.NET Core*. *ASP.NET Core* дозволяє взаємодіяти з операційною системою, в тому числі з послідовним портом, взаємодіяти з мережею і відправляти запит по протоколу *HTTP*.

Для відображення інтерфейсу є можливість використовувати технологію *Blazor* для формування, відображення сторінок і створення бізнес-логік моделей.

3.2.1. Конфігурація *web*-сервера

ASP.NET Core за своєю суттю є звичайним консольним додатком, точка входу якого знаходиться в класі *Program* в методі *Main()*:

```
public class Program
  {
    public static void Main(string[] args);
   }
```

Щоб додаток мав можливість спілкуватися по мережі, йому треба надати конфігурацію.

Метод *WebApplication.CreateBuilder(args);* починає конфігураційний процес і створює екземпляр конструктора веб-сервера, за допомогою якого можна конфігурувати веб-сервер:

```
var builder = WebApplication.CreateBuilder(args);
```

Конструктор веб-сервера має можливість отримувати значення конфігурації за допомогою параметра *Configuration*. *ASP.NET Core* зберігає всю зовнішню

конфігурацію у форматі *JSON* файлу, та по замовчуванню конфігурація зберігається у файлі *appsettings.json* і має вигляд:

```
{
  "ConnectionString":
"host=localhost;database=postgres;password=admin;username=postgres",
  "DetailedErrors": true,
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  }
}
```

Для того, щоб встановити зв’язок *Entity Framework* з базою даних *PostgreSQL* потрібно додати контекст бази, для цього необхідно отримати дані о підключенні до бази даних (*connection string*), ці дані потрібно отримати за параметра *Configuration*:

```
var connectionString = builder.Configuration.GetSection("ConnectionString");
```

Після того як була отримана дані о підключенні, потрібно додати контекст бази даних, в даному випадку їх два:

```
builder.Services.AddDbContext<ApplicationDbContext>(options =>
options.UseNpgsql(connectionString, options =>
options.MigrationsAssembly("WebAFDK")));
builder.Services.AddDbContext<IdentityDbContext>(options =>
options.UseNpgsql(connectionString, options =>
options.MigrationsAssembly("WebAFDK")));
```

Для того, щоб *Entity Framework* мав можливість автоматично оброблювати дані про користувачів, потрібно додати сутність в якій описуються дані користувача за допомогою методу *AddDefaultIdentity*, також необхідно вказати

контекст бази даних, через який будуть оброблюватись запити, цю задачу вирішує метод *AddEntityFrameworkStores*:

builder.Services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true) .AddEntityFrameworkStores<IdentityDbContext>();

Для збереження та обробки даних користувача використовується контекст бази *IdentityDbContext*, для інших типів (схеми, репозиторії) використовується контекст бази *ApplicationDbContext*.

Для роботи компонентів інтерфейсу *ASP.NET Core* та *Blazor* потрібно додати необхідні сервіси:

builder.Services.AddRazorPages();

builder.Services.AddServerSideBlazor();

В даному випадку використовується модель *Blazor* як *Server Side*, це означає, що всі перевірки та зміни стану, отримання поточних даних та конструювання всіх компонентів інтерфейсу відбувається на стороні серверу, а клієнт отримує готовий результат, це дає змогу змістити важкі обчислення з клієнтської машини на сервер.

Для того, щоб *Blazor* мав змогу перевіряти авторизацію користувача автоматично, треба додати сервіс *AuthenticationStateProvider*, який відповідає за перевірку поточного стану авторизації користувача, ця перевірка відбувається автоматично на будь-якій сторінці додатку:

builder.Services.AddScoped<AuthenticationStateProvider, RevalidatingIdentityAuthenticationStateProvider<IdentityUser>>();

Наступним кроком буде реєстрація сервісу для підтримки локалізації, в даному додатку підтримуються дві локалізації: українська та англійська:

builder.Services.AddLocalization();

var supportedCultures = new[] { "en-US", "uk-UA" };

var localizationOptions = new RequestLocalizationOptions()

.SetDefaultCulture(supportedCultures[1])

.AddSupportedCultures(supportedCultures) .

AddSupportedUICultures(supportedCultures);

Після того як всі необхідні конфігурації застосовані та всі необхідні сервіси додані, потрібно виконати збірку додатку web-сервера за допомогою метода конструктора *Build()*:

var app = builder.Build();

В змінній *app* знаходиться екземпляр додатку web-сервера, на цьому етапі вже неможна змінювати конфігурацію та сервіси додатку, але можна додавати проміжні програми для обробки запитів до серверу.

Першою підпрограмою буде використовуватись підпрограма локалізації, це дасть серверу можливість отримати дані о поточній локалізації користувача, в подальшому ці дані буде використовувати сервіс локалізації та застосовувати необхідну локалізацію в інтерфейсі автоматично:

app.UseRequestLocalization(localizationOptions);

В технології *ASP.NET Core* підпрограми мають конвеєрну структуру, це означає, що наступні підпрограми будуть оброблювати запит, який обробила попередня підпрограма і так далі.

Наступна підпрограма дає можливість серверу використовувати протокол *HTTPS* для безпечної маршрутизації:

app.UseHttpsRedirection();

Для коректної роботи серверу, щоб сервер міг оброблювати запити, виконувати маршрутизацію та знав власні маршруті до сторінок клієнта, треба додати наступні під програми:

app.UseStaticFiles();

app.UseRouting();

app.MapControllers();

app.MapBlazorHub();

app.MapFallbackToPage("/_Host");

Щоб оброблювати та отримувати дані о авторизації користувача з запитів, потрібно додати наступні підпрограми:

app.UseAuthentication();

app.UseAuthorization();

В кінці конфігурації підпрограм за допомогою методу екземпляру додатку *Run()* виконується запуск веб-серверу:

app.Run();

На даному етапі, після всіх вище описаних кроків, веб-сервер повноцінно може виконувати свою роботу та оброблювати запити.

3.2. Розробка клієнтського додатку

Основною конструкцією та єдиним статичним файлом розмітки в технології *Blazor* є файл *_Layout.cshtml*. В даному файлі описується розмітка клієнтського додатку, основа, яка буде повторюватись на всіх інших сторінках, також ця розмітка відображається тільки один раз та не змінюється в подальшому, тому вона і називається статичною. Також, в даному файлі додаються всі посилання на файлі *JavaScripit* та стилів *CSS* та додаються необхідні метадані:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="utf-8" />
   <meta name="viewport" content="width=device-width, initial-scale=1.0" />
   <base href="~/" />
   <link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
   <link href="css/site.css" rel="stylesheet" />
   <link href="WebAFDK.styles.css" rel="stylesheet" />
        <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.min.css">
        <link rel="stylesheet" href="simulator/css/simstyle.css">
        <script src="vendor/js/jquery-3.3.1.slim.min.js"></script>
        <script src="vendor/js/popper.min.js"></script>
```

```
        <script src="vendor/js/bootstrap.min.js"></script>
        <!-- p5 stuff -->
        <script src="vendor/js/p5.min.js"></script>
        <!-- Simulator stuff -->
        <script type="module" src="simulator/js/simulator.js"></script>
    <component type="typeof(HeadOutlet)" render-mode="ServerPrerendered"
/>
</head>
<body>
    @RenderBody()
    <div id="blazor-error-ui">
        <environment include="Staging,Production">
            An error has occurred. This application may no longer respond until
reloaded.
        </environment>
        <environment include="Development">
            An unhandled exception has occurred. See browser dev tools for details.
        </environment>
        <a href="" class="reload">Reload</a>
        <a class="dismiss">🗙</a>
    </div>
    <script src="_framework/blazor.server.js"></script>
    <script
src="_content/Blazor.Extensions.Canvas/blazor.extensions.canvas.js"></script>
</body>
</html>
```

В файлі *_Host.cshtml* додається компонент *Blazor*, який отримує тип компонента додатку та модель роботи *Blazor*:

```
<component type="typeof(App)" render-mode="ServerPrerendered" />
```

В компоненті *App* описується конструкція, яку буде використовувати кожний наступний компонент інтерфейсу:

```
<CascadingAuthenticationState>
   <Router AppAssembly="@typeof(App).Assembly">
      <Found Context="routeData">
         <AuthorizeRouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
         <FocusOnNavigate RouteData="@routeData" Selector="h1" />
      </Found>
      <NotFound>
         <PageTitle>Not found</PageTitle>
         <LayoutView Layout="@typeof(MainLayout)">
            <p role="alert">Sorry, there's nothing at this address.</p>
         </LayoutView>
      </NotFound>
   </Router>
</CascadingAuthenticationState>
```

Всі сторінки *Blazor* використовують макет. Макет – це розмітка, яка повинна повторюватись на кожній сторінці, щоб утилізувати велику частку коду, використовуються макети.

В даному випадку перший та основний макет є *MainLayout*:

```
<PageTitle>WebAFDK</PageTitle>
<div class="page">
   <div class="sidebar">
      <NavMenu />
   </div>
   <main>
      <div class="top-row px-4 auth">
         <LoginDisplay />
```

```
<a href="https://docs.microsoft.com/aspnet/" target="_blank">About</a>
        </div>
        <article class="content px-4">
            @Body
        </article>
    </main>
</div>
```

Макети також можуть змінюватись динамічно в залежності від поточного стану інтерфейсу.

Основною та першою сторінкою за маршрутом «/» є *Index.razor*, в ній знаходиться лише один компонент конструктора схем:

```
@page "/"
@using WebAFDK.Components
@using WebAFDK.Pages.BaseClasses
@layout MainLayout
@inherits BasePage
<PageTitle>Index</PageTitle>
<DesignerComponent/>
```

Конструктор, який знаходиться в компоненті *DesignerComponent* складається з декількох частин.

Перша частина – це верхнє меню конструктора, в якому знаходяться такі функції як:

- «Редагувати»;
- «Змістити»;
- «Видалити»;
- «Зберегти»;
- «Завантажити».

Верхнє меню описана цією розміткою:

```
<div class="collapse navbar-collapse" id="navbarCollapse">
```

```
        <ul class="navbar-nav mr-auto">
            <li class="nav-item">
                <div class="navGroupTools">
                    <button type="button" class="btn btn-outline-light Edit active"
tool="Edit"
                            onclick="activeTool(this)">
                        <i class="fa fa-edit"></i>
                        Edit
                    </button>
                    <button type="button" class="btn btn-outline-light" tool="Move"
onclick="activeTool(this)">
                        <i class="fa fa-arrows"></i>
                        Move
                    </button>
                    <button type="button" class="btn btn-outline-light" tool="Delete"
onclick="activeTool(this)">
                        <i class=" fa fa-trash-o"></i>
                        Delete
                    </button>
                </div>
                <div class="navGroupTools">
                    <label type="button" class="btn btn-outline-light"
style="margin:0px">
                        <input id="projectFile" type="file" />
                        <i class="fa fa-upload"></i>
                        Load
                    </label>
                    <label type="button" class="btn btn-outline-light">
                        <input id="saveProjectFile" type="file" />
                        <i class="fa fa-save"></i>
                        Save
                    </label>
                </div>
            </li>
        </ul>
</div>
```

Друга частина конструктора – це бокове меню, яке складається з набору логічних елементів, кожен елемент якого являється активною кнопку, при

активації якої в графічний області конструктора створюється необхідний елемент. Наприклад елемент входу описується такою розміткою:

```
<button type="button" tool="LogicInput" title="Logic Input" onclick="activeTool(this)"
    class="list-group-item list-group-item-action pl-1">
    <img src="simulator/img/LogicInput.svg" class="ml-2" width="32">
</button>
```

Наступна частина розмітки бокового меню описує логічні елементи:

```
<button type="button" tool="NOT" isGate="true" onclick="activeTool(this)"
    class="list-group-item list-group-item-action pl-1">
  <img src="simulator/img/NOT.svg" width="50">
</button>
<button type="button" tool="AND" isGate="true" onclick="activeTool(this)"
    class="list-group-item list-group-item-action pl-1">
  <img src="simulator/img/AND.svg" width="50">
</button>
<button type="button" tool="NAND" isGate="true" onclick="activeTool(this)"
    class="list-group-item list-group-item-action pl-1">
  <img src="simulator/img/NAND.svg" width="50">
</button>
<button type="button" tool="OR" isGate="true" onclick="activeTool(this)"
    class="list-group-item list-group-item-action pl-1">
  <img src="simulator/img/OR.svg" width="50">
</button>
<button type="button" tool="NOR" isGate="true" onclick="activeTool(this)"
    class="list-group-item list-group-item-action pl-1">
  <img src="simulator/img/NOR.svg" width="50">
</button>
<button type="button" tool="XOR" isGate="true" onclick="activeTool(this)"
    class="list-group-item list-group-item-action pl-1">
```

```
    <img src="simulator/img/XOR.svg" width="50">
</button>
<button type="button" tool="XNOR" isGate="true" onclick="activeTool(this)"
        class="list-group-item list-group-item-action pl-1">
    <img src="simulator/img/XNOR.svg" width="50">
</button>
```

В конструкторі також є можливість додавати тригери різних типів. Кожен тригер сам по собі має різну конфігурацію (синхронні/асинхронні та різні базиси реалізації), для вибору такої конфігурації тригера, під час його розміщення відображається модальне вікно. Нижче наведено приклад модального вікна конфігурації *JK*-тригеру:

```
<div class="modal fade" id="FF_JK-TypeSettings" tabindex="-1" role="dialog" aria-hidden="true">
    <div class="modal-dialog modal-dialog-centered" role="document">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title">JK-Type Flip-Flop Settings</h5>
            </div>
            <div class="modal-body">
                <h6>Choose the type</h6>
                <select class="custom-select FF_JK-Setting">
                    <option selected>Positive Edge-Triggered</option>
                    <option>Negative Edge-Triggered</option>
                </select>
            </div>
            <div class="modal-footer">
                <button type="button" class="btn btn-secondary" data-dismiss="modal">Cancel</button>
                <button type="button" class="btn btn-primary" tool="FF_JK" onclick="activeTool(this)"
```

```
                data-dismiss="modal">
              Create
            </button>
          </div>
        </div>
      </div>
    </div>
```

Логічні елементи були поділені на окремі структури, кожна з яких має індивідуальну логіку та написана мовою *JavaScript*. Нижче приведено список всіх структур в окремих файлах:

– *Gate.js* – логіка логічних елементів, описує поведінку базових елементів;
– *Wire.js* – описує логіку зв’язків між елементами;
– *FF_D.js* – описує логіку *D* тригера;
– *FF_T* – описує логіку *T* тригера;
– *FF_JK* – описує логіку *JK* тригера;
– *FF_SR* – описує логіку *SR* тригера;
– *Clock.js* – логіка генератора тактових сигналів;
– *LogicInput.js* – логіка логічного входу;
– *LogicOutput.js* – логіка логічного виходу;
– *Integrated.js* – базова логіка тригерів;
– *Node.js* – базова логіка для всіх елементів конструктора.

Нижче приведено приклад логіки генератора тактових сигналів на мові *JavaScript*:

```
import { LogicInput } from "./LogicInput.js";
/**
 * Digital Clock
 * @classdesc Digital Clock
 * @extends LogicInput
 */
export class Clock extends LogicInput {
```

```
  /**
   * @param {*} period TODO
   * @param {*} dutycycle TODO
   */
  constructor(period, dutycycle) {
    super();
    this.truePeriod = period * dutycycle / 100;
    this.falsePeriod = period * (100 - dutycycle) / 100;
    this.lastTick = new Date().getTime();
    this.strInfo = "CLOCK \nT = " + period + " ms\nD% = " + dutycycle;
  }
  /**
   * Function to call for drawing the Clock
   */
draw() {
    const currTick = new Date().getTime();
    const period = (this.value) ? this.truePeriod : this.falsePeriod;
    if (currTick - this.lastTick > period) {
      this.toggle();
      this.lastTick = currTick;
    }
    super.draw();
  }
  /**
   * Function to display Clock Info
   */
  printInfo() {
    noStroke();
    fill(0);
    textSize(12);
```

```
        textStyle(NORMAL);
        text(this.strInfo, this.posX - 20, this.posY + 25);
    }
};
```

Вся основна логіка конструктора комбінаційних схем та ініціалізація графічного поля була написана за допомогою мови *JavaScript*, вихідний код якого наведено в додатку А.

Приклад логіки логічного елемент, написаного за допомогою мови *JavaScript*, наведено в додатку Б.

3.3. Висновки до розділу

В даному розділі були описані процеси розробки веб-серверу та клієнтського додатку. Були описані все методи конфігурації веб-серверу та додавання підпрограм.

Використані технології *ASP.NET Core* та *Blazor* були застосовані для побудови клієнтського додатку, було наведено приклад структури та основні конструкції інтерфейсу додатку.

За допомогою мови *JavaScript* було побудовано конструктор комбінаційних схем, який дозволяю створювати та моделювати схеми різних типів.

ВИСНОВКИ

В ході виконання кваліфікаційної роботи був розроблений програмний додаток моделювання комбінаційних схем.

Архітектурно система складається з таких рівнів:

– веб-сервер;

– клієнтський додаток;

– база даних.

Щодо використаних для розробки технологій, то було застосовано наступний інструментарій:

– веб-сервер – *C#*, *ASP.NET Core*, *ASP.NET Core MVC*, *Entity Framework Core*;

– клієнтський додаток – *HTML* 5, *CSS* 3, *JavaSciprt*, *Blazor*;

– база даних – *PostgreSQL.*

Створена система відповідає таким критеріям:

– кросплатформність;

– використання сучасних технологій;

– відкритий код;

– безкоштовна.

В першому розділі було проведено аналіз предметної області кваліфікаційної роботи, тобто існуючі реалізації програмних додатків моделювання комбінаційних схем

На основі аналізу вже існуючих рішень, їх преваг та недоліків, було сформульовано постановку задачі для розроблюваного програмного додатку моделювання комбінаційних схем.

В другому розділі були проаналізовані та обрані всі необхідні технології для проєктування програмного додатку моделювання комбінаційних схем.

Для кожного основного елемента були описані і створені діаграма станів та *use-case* діаграма.

В третьому розділі були описані процеси розробки основних модулів та частин системи та методи, що були застосовані для цього.

Результати виконання кваліфікаційної роботи слід використовувати при проектуванні та реалізації комбінаційних схем, послідовних схем та цифрових автоматів.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бойченко С.В., Іванченко О.В. Положення про дипломні роботи (проєкти) випускників Національного авіаційного університету. – К.: НАУ, 2017. – 63 с.

2. ДСТУ 3008–95 “Документація. Звіти у сфері науки і техніки. Структура і правила оформлення”.

3. Комп’ютерна логіка. Підручник. Вид. 2-ге перероб. та доп. – Київ : Видавництво Ліра-К, 2017. – 324 с.

4. *Richter J. CLR via C#. – Microsoft Press*, 2010. – 873 с.

5. *Flanagan D. JavaScript: the definitive guide. – O'Reilly, Beijing*, 2011. – 1086 с.

6. *Freeman A. Pro ASP.NET Core 3: develop cloud-ready web applications using MVC, Blazor, and Razor Pages. – Apress L.P., Berkeley, CA*, 2020. - 736 с.

7. *Troelsen A., Japikse P. Pro C# 10 with .NET 6: foundational principles and practices in programming. – Apress, New York*, 2022. – 1705 с.

8. *Nagel C. Professional C# 7 and .NET Core 2.0. – John Wiley & Sons, Incorporated, Newark*, 2018. – 1441 с.

9. *McFarland D. CSS3: the Missing Manual, 3rd Edition. O'Reilly Media, Inc.*, 2012. – 648 с.

10. Буров Є. Комп'ютерні мережі. Магнолія 2010. – 262 с.

11. *Freeman A. Pro jQuery 2.0. – Apress, New York*, 2013 – 1073 с.

12. *Spurlock J. Bootstrap. Responsive Web-Development. – O'Reilly*, 2013. – 128 с.

13. *Himschoot P. Microsoft Blazor: Building Web Applications in .NET – Apress; 2nd ed. Edition*, 2020. – 303 с.

14. *Himschoot P. Microsoft Blazor: Building Web Applications in .NET 6 and Beyond – Apress; 3nd ed. Edition*, 2021. – 672 с.

15. *Himschoot P. Blazor Revealed: Building Web Applications in .NET– Apress; 1st ed. Edition*, 2019. – 285 с.

ДОДАТОК А

ВИХІДНИЙ КОД КОНСТРУКТОРА СХЕМ

```
import { activeTool, currMouseAction } from "./menutools.js"
import { MouseAction } from "./circuit_components/Enums.js"
import { WireManager } from "./circuit_components/Wire.js";
export let gateIMG = [];
export let IC_IMG = [];
export let gate = [];
export let logicInput = [];
export let logicOutput = [];
export let logicClock = [];
export let srLatch = [];
export let flipflop = [];
export let wireMng;
export let colorMouseOver = [0 ,0x7B, 0xFF];
export function preload() {
    gateIMG.push(loadImage('simulator/img/LogicInput.svg'));
    gateIMG.push(loadImage('simulator/img/NOT.svg'));
    gateIMG.push(loadImage('simulator/img/AND.svg'));
    gateIMG.push(loadImage('simulator/img/NAND.svg'));
    gateIMG.push(loadImage('simulator/img/OR.svg'));
    gateIMG.push(loadImage('simulator/img/NOR.svg'));
    gateIMG.push(loadImage('simulator/img/XOR.svg'));
    gateIMG.push(loadImage('simulator/img/XNOR.svg'));
    IC_IMG.push(loadImage('simulator/img/SR_Latch.svg'));
    IC_IMG.push(loadImage('simulator/img/SR_Latch.svg'));
    IC_IMG.push(loadImage('simulator/img/SR_Latch_Sync.svg'));
    IC_IMG.push(loadImage('simulator/img/FF_D.svg'));
```

```
    IC_IMG.push(loadImage('simulator/img/FF_D_MS.svg'));
    IC_IMG.push(loadImage('simulator/img/FF_T.svg'));
    IC_IMG.push(loadImage('simulator/img/FF_JK.svg'));
}
export function setup() {
    const canvHeight = windowHeight - 90;
    let canvas = createCanvas(windowWidth - 115, canvHeight, P2D);
    canvas.parent('canvas-sim');
    document.getElementsByClassName("tools")[0].style.height = canvHeight;
    wireMng = new WireManager();
}
export function windowResized() {
    const canvHeight = windowHeight - 90;
    resizeCanvas(windowWidth - 115, canvHeight);
    document.getElementsByClassName("tools")[0].style.height = canvHeight;
}
export function draw() {
    background(0xFF);
    stroke(0);
    strokeWeight(4);
    fill(0xFF)
    rect(0, 0, width, height);
    wireMng.draw();
    for (let i = 0; i < gate.length; i++)
        gate[i].draw();
    for (let i = 0; i < logicInput.length; i++)
        logicInput[i].draw();
    for (let i = 0; i < logicOutput.length; i++)
        logicOutput[i].draw();
```

```
    for (let i = 0; i < logicClock.length; i++)
        logicClock[i].draw();
    for (let i = 0; i < srLatch.length; i++)
        srLatch[i].draw();
    for (let i = 0; i < flipflop.length; i++)
        flipflop[i].draw();
}
export function mousePressed() {
    for (let i = 0; i < gate.length; i++)
        gate[i].mousePressed();
    for (let i = 0; i < logicInput.length; i++)
        logicInput[i].mousePressed();
    for (let i = 0; i < logicOutput.length; i++)
        logicOutput[i].mousePressed();
    for (let i = 0; i < logicClock.length; i++)
        logicClock[i].mousePressed();
    for (let i = 0; i < srLatch.length; i++)
        srLatch[i].mousePressed();
    for (let i = 0; i < flipflop.length; i++)
        flipflop[i].mousePressed();
}
export function mouseReleased() {
    for (let i = 0; i < gate.length; i++)
        gate[i].mouseReleased();
    for (let i = 0; i < logicInput.length; i++)
        logicInput[i].mouseReleased();
    for (let i = 0; i < logicOutput.length; i++)
        logicOutput[i].mouseReleased();
    for (let i = 0; i < logicClock.length; i++)
        logicClock[i].mouseReleased();
```

```
    for (let i = 0; i < srLatch.length; i++)
        srLatch[i].mouseReleased();
    for (let i = 0; i < flipflop.length; i++)
        flipflop[i].mouseReleased();
}
export function doubleClicked() {
    for (let i = 0; i < logicInput.length; i++)
        logicInput[i].doubleClicked();
}
export function mouseClicked() {
    if (currMouseAction == MouseAction.EDIT) {
        for (let i = 0; i < gate.length; i++)
            gate[i].mouseClicked();
        for (let i = 0; i < logicInput.length; i++)
            logicInput[i].mouseClicked();
        for (let i = 0; i < logicOutput.length; i++)
            logicOutput[i].mouseClicked();
        for (let i = 0; i < logicClock.length; i++)
            logicClock[i].mouseClicked();
        for (let i = 0; i < srLatch.length; i++)
            srLatch[i].mouseClicked();
        for (let i = 0; i < flipflop.length; i++)
            flipflop[i].mouseClicked();
    } else if (currMouseAction == MouseAction.DELETE) {
        for (let i = 0; i < gate.length; i++) {
            if (gate[i].mouseClicked()) {
                gate[i].destroy();
                delete gate[i];
                gate.splice(i, 1);
```

```
    }
}
for (let i = 0; i < logicInput.length; i++) {
    if (logicInput[i].mouseClicked()) {
        logicInput[i].destroy();
        delete logicInput[i];
        logicInput.splice(i, 1);
    }
}
for (let i = 0; i < logicOutput.length; i++) {
    if (logicOutput[i].mouseClicked()) {
        logicOutput[i].destroy();
        delete logicOutput[i];
        logicOutput.splice(i, 1);
    }
}
for (let i = 0; i < logicClock.length; i++) {
    if (logicClock[i].mouseClicked()) {
        logicClock[i].destroy();
        delete logicClock[i];
        logicClock.splice(i, 1);
    }
}
for (let i = 0; i < srLatch.length; i++) {
    if (srLatch[i].mouseClicked()) {
        srLatch[i].destroy();
        delete srLatch[i];
        srLatch.splice(i, 1);
    }
}
```

```
      for (let i = 0; i < flipflop.length; i++) {
        if (flipflop[i].mouseClicked()) {
          flipflop[i].destroy();
          delete flipflop[i];
          flipflop.splice(i, 1);
        }
      }
    }
    wireMng.mouseClicked();
}
window.preload = preload;
window.setup = setup;
window.draw = draw;
window.windowResized = windowResized;
window.mousePressed = mousePressed;
window.mouseReleased = mouseReleased;
window.doubleClicked = doubleClicked;
window.mouseClicked = mouseClicked;
window.activeTool = activeTool;
```

ДОДАТОК Б

ВИХІДНИЙ КОД КЛАСУ ЛОГІЧНОГО ЕЛЕМЕНТА

```
import { currMouseAction, backToEdit } from "../menutools.js"
import { gateIMG } from "../simulator.js";
import { gateType, MouseAction } from "./Enums.js"
import { Node } from "./Node.js";
import { colorMouseOver } from "../simulator.js";
export class Gate {
   constructor(strType) {
      this.strType = strType;
      this.type = this.convertToType(strType);
      this.width = gateIMG[this.type].width;
      this.height = gateIMG[this.type].height;
      this.posX = mouseX - (this.width / 2);
      this.posY = mouseY - (this.height / 2);
      this.isSpawned = false;
      this.offsetMouseX = 0;
      this.offsetMouseY = 0;
      this.isMoving = false;
      this.isSaved = false;
      this.input = [];
      this.input.push(new Node(this.posX, this.posY + 15));
      if (this.type != gateType.NOT) {
         this.input.push(new Node(this.posX, this.posY + this.height - 15));
         this.input[0].setBrother(this.input[1]);
         this.input[1].setBrother(this.input[0]);
      }
```

```
        this.output = new Node(this.posX + this.width, this.posY + this.height / 2,
true);
        this.nodeStartID = this.input[0].id;
    }
    destroy() {
        for (let i = 0; i < this.input.length; i++) {
            this.input[i].destroy();
            delete this.input[i];
        }
        this.output.destroy();
        delete this.output;
    }
    draw() {
        if (!this.isSpawned) {
            this.posX = mouseX - (this.width / 2);
            this.posY = mouseY - (this.height / 2);
        }else if(!this.isSaved)
        {
            this.isSaved = true;
        }
        if (this.isMoving) {
            this.posX = mouseX + this.offsetMouseX;
            this.posY = mouseY + this.offsetMouseY;
        }

        if (this.type == gateType.NOT) {
            this.input[0].updatePosition(this.posX, this.posY + this.height / 2);
        } else {
            this.input[0].updatePosition(this.posX, this.posY + 15);
            this.input[1].updatePosition(this.posX, this.posY + this.height - 15);
```

```
        }
        this.output.updatePosition(this.posX + this.width, this.posY + this.height /
2);
        if (this.isMouseOver()) {
            noFill();
            strokeWeight(2);
            stroke(colorMouseOver[0], colorMouseOver[1], colorMouseOver[2]);
            rect(this.posX, this.posY, this.width, this.height);
        }
        image(gateIMG[this.type], this.posX, this.posY);
        for (let i = 0; i < this.input.length; i++)
            this.input[i].draw();
        this.generateOutput();
        this.output.draw();
    }
    refreshNodes()
    {
        let currentID = this.nodeStartID;
        this.input[0].setID(currentID);
        currentID++;
        if (this.type != gateType.NOT)
        {
            this.input[1].setID(currentID);
            currentID++;
        }
        this.output.setID(currentID);
    }
    generateOutput() {
        this.output.setValue(this.calculateValue());
    }
```

```
calculateValue() {
   switch (this.type) {
      case gateType.NOT:
         return !this.input[0].getValue();
      case gateType.AND:
         return this.input[0].getValue() && this.input[1].getValue();
      case gateType.NAND:
         return !(this.input[0].getValue() && this.input[1].getValue());
      case gateType.OR:
         return this.input[0].getValue() || this.input[1].getValue();
      case gateType.NOR:
         return !(this.input[0].getValue() || this.input[1].getValue());
      case gateType.XOR:
         return this.input[0].getValue() ^ this.input[1].getValue();
      case gateType.XNOR:
         return !(this.input[0].getValue() ^ this.input[1].getValue());
   }
}
convertToType(str) {
   switch (str) {
      case "NOT":
         return gateType.NOT;
      case "AND":
         return gateType.AND;
      case "NAND":
         return gateType.NAND;
      case "OR":
         return gateType.OR;
      case "NOR":
         return gateType.NOR;
```

```
        case "XOR":
            return gateType.XOR;
        case "XNOR":
            return gateType.XNOR;
    }
}

isMouseOver() {
    if (mouseX > this.posX && mouseX < (this.posX + this.width)
        && mouseY > this.posY && mouseY < (this.posY + this.height))
        return true;
    return false;
}
mousePressed() {
    if (!this.isSpawned) {
        this.posX = mouseX - (this.width / 2);
        this.posY = mouseY - (this.height / 2);
        this.isSpawned = true;
        backToEdit();
        return;
    }
    if (this.isMouseOver() || currMouseAction == MouseAction.MOVE) {
        this.isMoving = true;
        this.offsetMouseX = this.posX - mouseX;
        this.offsetMouseY = this.posY - mouseY;
    }
}
mouseReleased() {
    this.isMoving = false;
}
```

```
    mouseClicked() {
        let result = this.isMouseOver();

        for (let i = 0; i < this.input.length; i++)
            result |= this.input[i].mouseClicked();
        result |= this.output.mouseClicked();
        return result;
    }
};
```