

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Є.О. Батрак

АРХІТЕКТУРА КОМП'ЮТЕРНИХ СИСТЕМ

ЛАБОРАТОРНИЙ ПРАКТИКУМ

*Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра,
за освітньою програмою «Інформаційне забезпечення робототехнічних
систем» спеціальністю 126 «Інформаційні системи та технології»*

Київ
КПІ ім. Ігоря Сікорського
2020

Рецензенти: *ТОЛЮПА С.В., д.т.н., професор, професор кафедри кібербезпеки та захисту інформації Київського національного університету ім. Тараса Шевченко*

ДРУЖИНИН В.А., д.т.н., професор, професор кафедри радіотехніки та радіо-електронних систем Київського національного університету ім. Тараса Шевченко

Відповідальний редактор *ЦЬОПА Н.В., канд. техн. наук, доц.*

*Гриф надано Методичною радою КПП ім. Ігоря Сікорського (протокол № 4 від 10 . 12 .2020р .)
за поданням Вченої ради факультету (протокол № 4 від 23 . 11 .2020р .)*

Електронне мережне навчальне видання

Батрак Євгеній Олександрович, канд. техн. наук, доц.

АРХІТЕКТУРА КОМП'ЮТЕРНИХ СИСТЕМ ЛАБОРАТОРНИЙ ПРАКТИКУМ

Архітектура комп'ютерних систем: лабораторний практикум [Електронний ресурс] : навчальний посібник для студ. спеціальності 126 «Інформаційні системи та технології» / Є. О. Батрак ; КПП ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 12,3 Мбайт). – Київ : КПП ім. Ігоря Сікорського, 2020. – 110 с.

Посібник призначений для опанування теоретичних та практичних навичок, які необхідні майбутнім фахівцям для вивчення дисципліни «Архітектура комп'ютерних систем». Даний посібник дозволяє ознайомитися з основними розробками в створенні сучасних методів генерування інтегральних схем, роботи в Quartus з мовою SystemVerilog HDL. Архітектура комп'ютерних систем — це дисципліна, яка описує функціональність, організацію та реалізацію комп'ютерних систем. Опис комп'ютерної архітектури містить систему команд, логічні побудови та реалізації. Для кожної теми наведено перелік питань для самоконтролю, задачі для роботи в аудиторії різного рівня складності. Посібник призначений для студентів спеціальності 126 «Інформаційні системи та технології» всіх форм навчання.

©Є.О. Батрак 2020
© КПП ім. Ігоря Сікорського, 2020

Зміст

Вступ.....	4
Розділ 1. Теоретичні відомості.....	7
1.1. Мова опису апаратури SystemVerilog.....	7
1.2. Модулі.....	7
1.3. Типи даних.....	7
1.4. Оператори.....	8
1.5. Блоки always.....	9
1.6. Твердження if...else.....	10
1.7. Твердження case.....	10
1.8. Створення проекту в середовищі розробки Quartus.....	10
Розділ 2. Приклади реалізації цифрових пристроїв.....	19
2.1. Приклади реалізації цифрових пристроїв на ПЛІС Cyclone II.....	19
2.2. Реалізація схеми чотирирозрядного лічильника.....	19
2.3. Реалізація схеми дешифратора.....	23
2.4. Реалізація схеми розподільника імпульсу заданої форми.....	25
Розділ 3. Лабораторні роботи	19
3.1. Лабораторна робота №1 «Операція додавання».....	32
3.2. Лабораторна робота №2 «Операція віднімання».....	47
3.3. Лабораторна робота №3 «Операція множення».....	59
3.4. Лабораторна робота №4 «Операція ділення».....	73
3.5. Лабораторна робота №5 «Алгоритмічно-логічний пристрій (АЛП)».....	82
3.6. Лабораторна робота №6 «Тригер, регістр, JK-тригер, JK-регістр».....	91
3.7. Лабораторна робота №7 «JK-засувка, ROM, RAM».....	104
Список використаних джерел.....	110

ВСТУП

Мета даного посібника допомогти виконати лабораторний практикум з курсу «Архітектура комп'ютерних систем» за допомогою сучасних методів генерування інтегральних схем. Далі будуть описані основи HDL (hardware description language) роботи в Quartus з мовою SystemVerilog HDL.

Незважаючи на те, що в даний момент інформація про базові конструкції мови Verilog HDL широко представлена і в російськомовних книгах, і в інтернет-документах, вона по більшій мірі є загальнотеоретичної і не містить прикладів конкретних практично реалізованих цифрових схем. Ситуація з освоєнням мови також ускладнена тим, що досить велика кількість операторів мови Verilog HDL не беруть участі безпосередньо в синтезі цифрових схем і призначені лише для тестування створюваних схем. Який починає працювати на мові Verilog досить важко відчуту цю різницю. Інший труднощами освоєння мови є використання в ньому тих же за назвою конструкцій (ключових слів), що і в поширених мовах для програмування процесорів (Сі, Паскаль і т.д.), що штовхають новачків на невірну інтерпретацію «знайомих» ключових слів. З урахуванням того, що нині мови програмування процесорів увійшли практично в усі програми навчання технічних вузів, освоєння мови Verilog супроводжується боротьбою швидше з непотрібними стереотипами, почерпнутими з класичних мов, ніж з розумінням мотивів, покладених в основу синтезу цифрових схем по текстовим конструкціям.

На цьому, однак, всі труднощі початківця не закінчуються, ще одну трудність становить освоєння середовища програмування, в якій виконуються всі роботи по розробці цифрової схеми. Через те, що такі середовища крім синтезу «абстрактної» цифрової схеми (тобто не прив'язаної до конкретної мікросхеми ПЛІС – інтегральної схемою з прогамованими логічними характеристиками) виконують досить багато завдань, зокрема, «підгонку» (вибір конкретних елементів в конкретній ПЛІС і ліній їх сполуки) отриманої при синтезі «абстрактної» схеми до архітектури (Топології) конкретної ПЛІС, виконують тимчасовий аналіз схеми, реалізованої в конкретній ПЛІС, допомагають в налагодженні схеми в ПЛІС і т.д. така багатозадачність середовища програмування ПЛІС призводить до того, що її інтерфейс рясніє безліччю інструментів і не завжди є інтуїтивно зрозумілим, через що початківець просто втрачається.

Внаслідок такого становища освоєння мови Verilog краще починати не з опису ключових слів мови, як це прийнято в класичних мовах програмування процесорів, а з готових прикладів цифрових схем, що спираються на базові конструкції синтезується частини мови Verilog. І тут найважливішим є розуміння

основних принципів програми, що синтезує загальну логічну схему за текстом Verilog коду. Таку програму називають або «схемогенератор», або «Схемосінтезатор» (форма найбільш близька до англійської назви). Для розуміння логіки роботи схемосінтезатора обов'язковою умовою є супровід Verilog коду «абстрактної» схемою (не прив'язаної до апаратної платформи конкретної ПЛІС), створеної за нього і зображеної графічно на так званому рівні реєстрових передач (RTL - register-transfer level). Рівень RTL – це рівень графічного зображення схеми, досить віддалений від низового рівня транзисторів, коли схема зображується, в основному, за допомогою реєстрів, тригерів, мультиплексорів, дешифраторів, цифрових компараторів, суматорів і буферних елементів з мінімальним залученням найпростіших схем І, АБО, що виключає до малювання RTL-схем. Програмне забезпечення сучасних САПР для великих структурованих (ієрархічних) проектів поповнює свої RTL-схеми додатковими блоками, приховують RTL-схему молодшого за проектною ієрархією модуля і зображуваними в вигляді прямокутників. У такому ключі зображуються цифрові автомати (або в англійській традиції state-machines – машини станів) без доступу до їх внутрішньої RTL-схеми.

Початківцю HDL-програмісту (або FPGA-програмісту – за назвою технології найпопулярніших зараз програмованих ПЛІС) буває важко зорієнтуватися, як виробити «хороший стиль» програмування, що дозволяє отримувати надійну і компактну схему. Одним з простих способів для формування такого стилю є перегляд RTL-схеми, створюваної схемогенератором. Зокрема, під час написання HDL-програм небажано «бездумно» нарощувати число вкладених або численних гілок в умовних операторах, так це призводить до формування численних каскадів мультиплексорів, що знижують компактність схеми.

Синтезовані RTL-схеми у різних виробників програм-схемосінтезаторів можуть трохи відрізнятися. В даному посібнику RTL-схеми отримані за допомогою САПР Quartus II (від фірми Altera). Такий вибір пов'язаний, в основному, з тим, що дана САПР дозволяє не тільки синтезувати схеми, але також програмувати (конфігурувати) ПЛІС, що використовуються в практичних роботах, і немає необхідності у вивченні програм сторонніх виробників. Розробники САПР приділяють велику увагу виразності RTL-схем, і не тільки в сенсі зручності компоновки всіх елементів і сполучних проводів, але і в сенсі барвистості їх подання. щоб читач в повній мірі міг оцінити цю роботу розробників САПР, частина RTL-схем, представлених в посібнику, отримана в 9-й версії Quartus II (з функціональними вузлами без колірної заливки), а частина – в 13-ої версії (точніше в версії 13.1 з функціональними вузлами з синьо-зеленої заливкою), що вийшла

п'ять років після 9-ї версії.

Мета цього посібника полягає в тому, щоб продемонструвати особливості та можливості Verilog для опису та синтезу цифрових схем з використанням тільки тих конструкцій, які корисні для логічних схем. Однак ці, або дуже схожі конструкції, підтримуються іншими засобами розробки. При бажанні читач може порівняти проекти Verilog з подібними проектами, використовуючи мови VHDL і AHDL, і побачити як сильні, так і слабкі сторони мови, порівнюючи його з іншими.

SystemVerilog — мова опису апаратури (HDL), що використовується для опису та моделювання електронних систем. Слід зазначити, що опис апаратури, написаний мовою SystemVerilog (як і іншими HDL-мовами) прийнято називати програмами, але, на відміну від загальноприйнятого поняття програми, як послідовності інструкцій, тут програма представляє множину операторів, які виконуються паралельно і циклічно під керуванням об'єктів, названих сигналами. Кожен такий оператор є моделлю певного елемента реальної функціональної схеми апаратури, а сигнал — аналогом реального логічного сигналу. Так само для мови SystemVerilog не застосовується термін «виконання програми». Фактично, виконання Verilog- програми є моделюванням функціональної схеми, яку вона описує, що виконується спеціальною програмою — Verilog-симулятором. Синтаксис мови дещо схожий з С. Бітові оператори абсолютно ідентичні (крім них нам майже нічого не будемо використовувати). Основний принцип написання програм — це опис модулів, які в свою чергу будуть слугувати частинами інших модулів і так далі до опису кінцевого бажаного модуля. Модуль — це блок цифрової апаратури, який має входи та виходи. Наприклад логічний елемент “І”, мультиплексор або АЛП є прикладами модулів. Існує два загальноприйнятих типи опису функціональності модуля — поведінковий та структурний. Поведінкова модель описує, що робить модуль. Структурна модель описує те, як побудовано модуль з простих елементів.

Розділ 1. Теоретичні відомості

1.1. Мова Опису Апаратури Systemverilog

SystemVerilog – мова опису і верифікації апаратури, що є розширенням мови Verilog. Verilog був розроблений компанією Gateway Design Automation в 1984. У 1989 році, після придбання останніх фірмою Cadence Design Systems, Verilog став відкритим стандартом під керуванням спільноти Open Verilog International. У 2005 році мова була розширена. Ці розширення об'єднані в єдиний стандарт, який зараз називається SystemVerilog. SystemVerilog поєднує можливості верифікації HVL (апаратний мову верифікації) з простотою Verilog для забезпечення єдиної платформи як для проектування, так і для верифікації. Розробники Verilog зробили його синтаксис дуже схожим на синтаксис мови C, що спрощує його освоєння. Verilog має препроцесор, дуже схожий на препроцесор мови C, і основні керуючі конструкції «if», «while» також подібні однойменним конструкціям мови C. Слід зазначити, що опис апаратури, написаний на мові Verilog (як і на інших HDL-мовах) прийнято називати програмами, але на відміну від загальноприйнятого поняття програми як послідовності інструкцій, тут програма задає структуру системи. Так само для мови Verilog застосовуємо термін "виконання програми".

1.2. Модулі

Verilog дозволяє групувати логіку в блоки. Кожен блок логіки називається «модулем». Модулі мають входи і виходи, які ведуть себе як сигнали wire. Модуль оголошується за допомогою слова module. При описі модуля записують його ім'я, перераховують порти (входи і виходи) і типи портів. В кінці визначення модуля повинно бути написано endmodule.

```
module <modulename> (module_terminal_list); endmodule
```

1.3. Типи Даних

SystemVerilog додав багато нових типів даних і поліпшив існуючі типи даних, щоб поліпшити використання пам'яті під час виконання симуляторів.

Цілочисельні типи даних можуть бути класифіковані на два типу: з двома станами і з чотирма станами. Типи з двома станами можуть набувати значень тільки 0, 1. Тоді як типи з чотирма станами можуть приймати 0, 1, X (невідоме значення), Z (стан високого опору). Типи з двома станами споживають менше пам'яті і прискорюють симуляцію.

Цілочисельні типи даних з двома станами:

- shortint: 16-розрядний ціле число зі знаком
- int: 32-розрядний ціле число зі знаком;
- longint: 64-розрядний ціле число зі знаком;
- байт: 8-розрядний ціле число зі знаком;
- біт: тип векторів, який визначається користувачем

Цілочисельні типи даних з чотирма станами:

- logic: тип векторів, який визначається користувачем;
- reg: тип векторів, який визначається користувачем;
- wire: тип векторів, який визначається користувачем;
- integer: 32-розрядний ціле число зі знаком;
- time: 64-розрядний ціле число без знака

Цілочисельні типи можуть бути зі знаком або без знаку. Таким чином, вони можуть змінити результат арифметичної операції. За замовчуванням byte, shortint, int, integer і longint є типами даних зі знаком, а bit, reg, logic і wire - без знака.

1.4. Оператори

Оператори в SystemVerilog такі ж, як і в інших мовах програмування. Вони можуть бути унітарними, бінарними і тернарного.

```
a = ~ b;           // ~ унітарний оператор
a = b && c;         // && бінарний оператор
a = b ? c : d;      // ? : Тернарний оператор
```

У таблиці 1.1 наведені оператори застосовуються в SystemVerilog.

Таблиця 1.1.

Тип оператора	Символ оператора	Виконувана операція
Арифметичні	*	множення
	/	розподіл
	+	додавання
	-	віднімання
	%	модуль
	+	унарний плюс
	-	унарний мінус

Логічні	!	логічне НЕ
	&&	логічне І
	 	логічне АБО
Порівняння	>	більше
	<	менше
	>=	більше або дорівнює
	<=	менше або дорівнює
Рівність	==	рівність
	!=	нерівність
Редукція	~	інверсія
	~&	NAND
	 	OR
	~ 	NOR
	^	XOR
	^~	XNOR
	~^	XNOR
Зрушення	>>	зрушення вправо
	<<	зрушення вліво
Об'єднання	{}	об'єднання
Умови	?:	умовний оператор

1.5. Блоки always

Always виконується кожен раз, як тільки змінить своє значення хоча б одна з змінних, що знаходиться в списку чутливості цього твердження. Синтаксис блоку always:

```
always @ (event-expression)
begin
... statements ...
End
```

Для того, щоб описати тип апаратури в явному вигляді, SystemVerilog додає 3 нових процес блоку: always_comb, always_ff, і always_latch.

Затвердження always_comb використовується для моделювання комбінаційної логіки. Блок always_ff дозволяє описувати послідовних логіку. За допомогою блоків always_latch описуються засувки.

1.6. Твердження if ... else

Оператори if ... else перевіряють умову, щоб вирішити, виконувати частину коду чи ні. Якщо вираз після if вірно, то виконується перший блок тверджень. Якщо вираз брехня, виконується блок після else. Синтаксис оператора if ... else:

```
if (expr)
begin
... statements ...
end

else
begin
... statements ...
End
```

1.7. Твердження case

Оператори case використовуються, коли у нас є одна змінна, яку необхідно перевірити на наявність декількох значень. Замість використання декількох вкладених операторів if-else, по одному для кожного шуканого значення, використовується один оператор case. Синтаксис оператора case:

```
case (expr)
case_item1: begin ... end
case_item2: begin ... end

default: begin ... end
endcase
```

1.8. Створення проекту в середовищі розробки Quartus

Для проектування схем цифрових пристроїв необхідно створити директорію, де будуть міститися файли проекту. Через робочий стіл запускаємо програму Quartus II. Натискаємо Create a New Project (New Project Wizard).

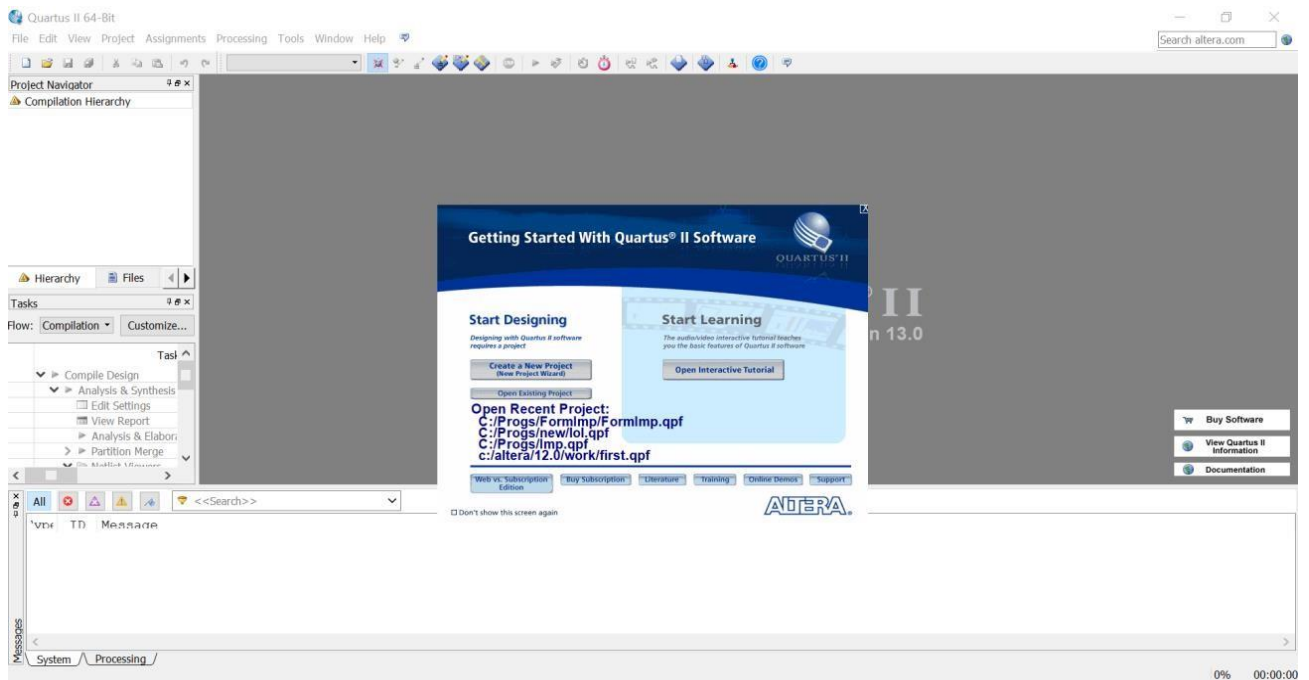


Рис. 1.1. Початкове вікно Quartus II

У вікні New Project Wizard встановлюємо шлях до директорії, де буде збережений наш проект, ім'я проекту і ім'я об'єкта проекту верхнього рівня. Зазвичай ім'я проекту і проекту верхнього рівня збігаються. Натискаємо Next для переходячи в наступне вікно.

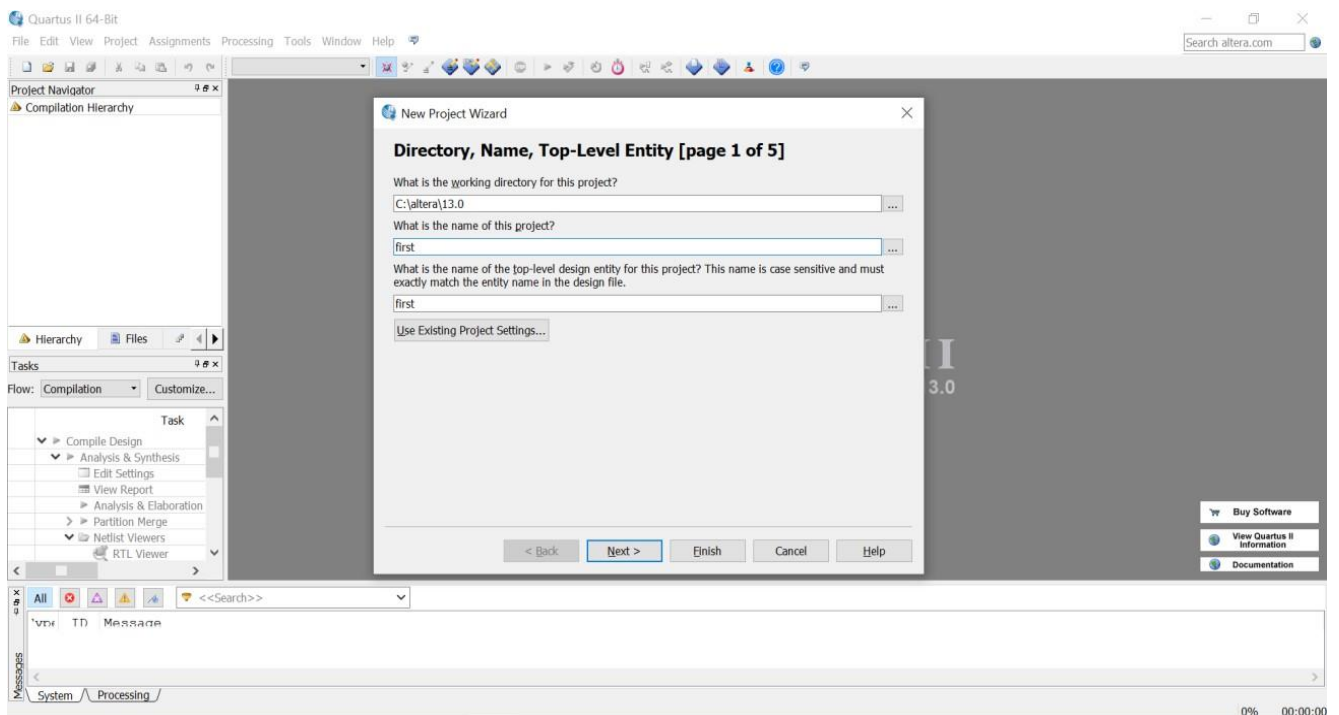


Рис. 1.2. New Project Wizard установка директорії і вибір імені проекту

У наступному вікні можна додати в проект інші файли, модулі з інших проектів. Так як у нас немає раніше створених файлів, пропускаємо цей крок і натискаємо Next.

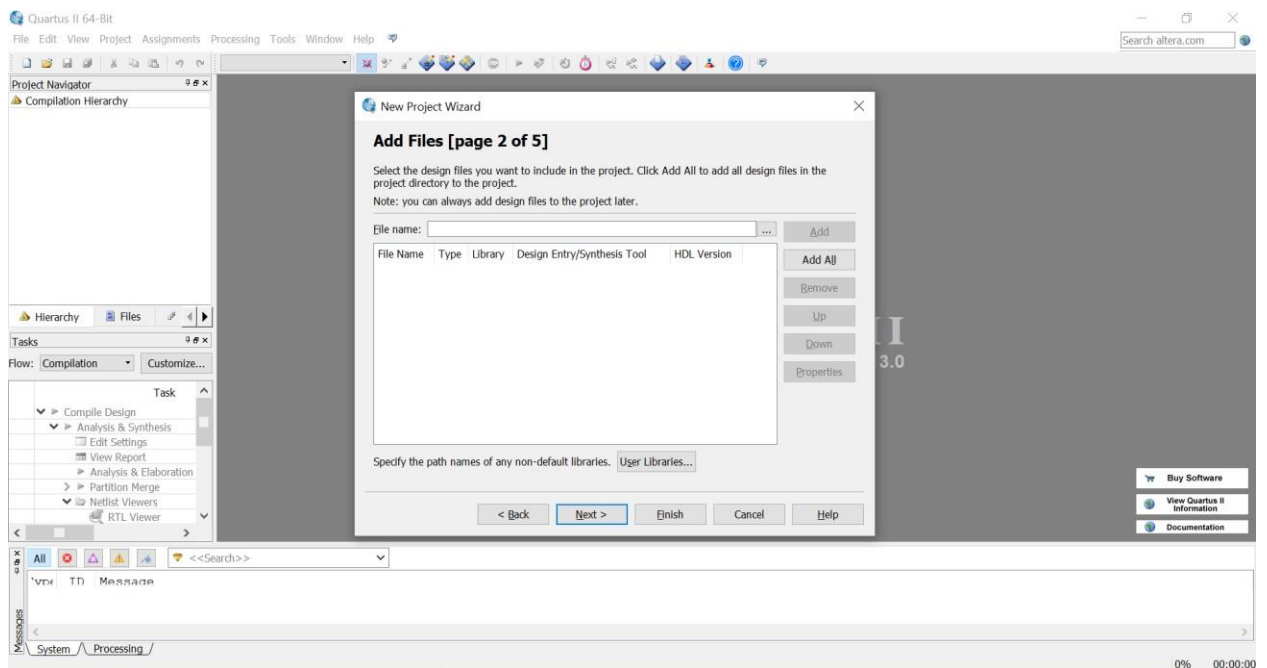


Рис. 1.3. New Project Wizard додавання файлів

Далі необхідно визначити тип нашого пристрою. Отлагодная плата DE1 відноситься до сімейства Cyclone IV. У рядку Family вибираємо Cyclone IV (E)(GX). Зі списку Available Devices вибираємо ім'я нашого пристрою – будь-яку мікросхему і натискаємо Next.

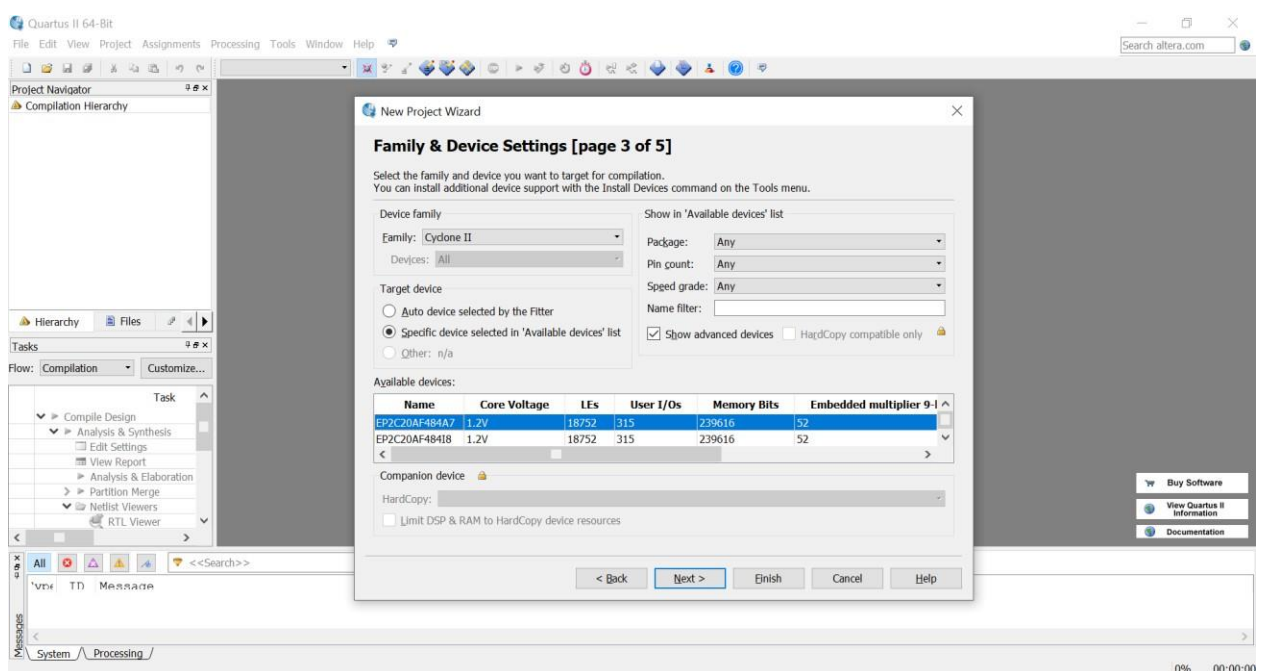


Рис. 1.4. New Project Wizard вибір типу пристрою

Користувачеві пропонують визначити додаткові програмні інструменти, які будуть використовуватися в проєкті. Ми пропускаємо цей крок. Натискаємо Finish.

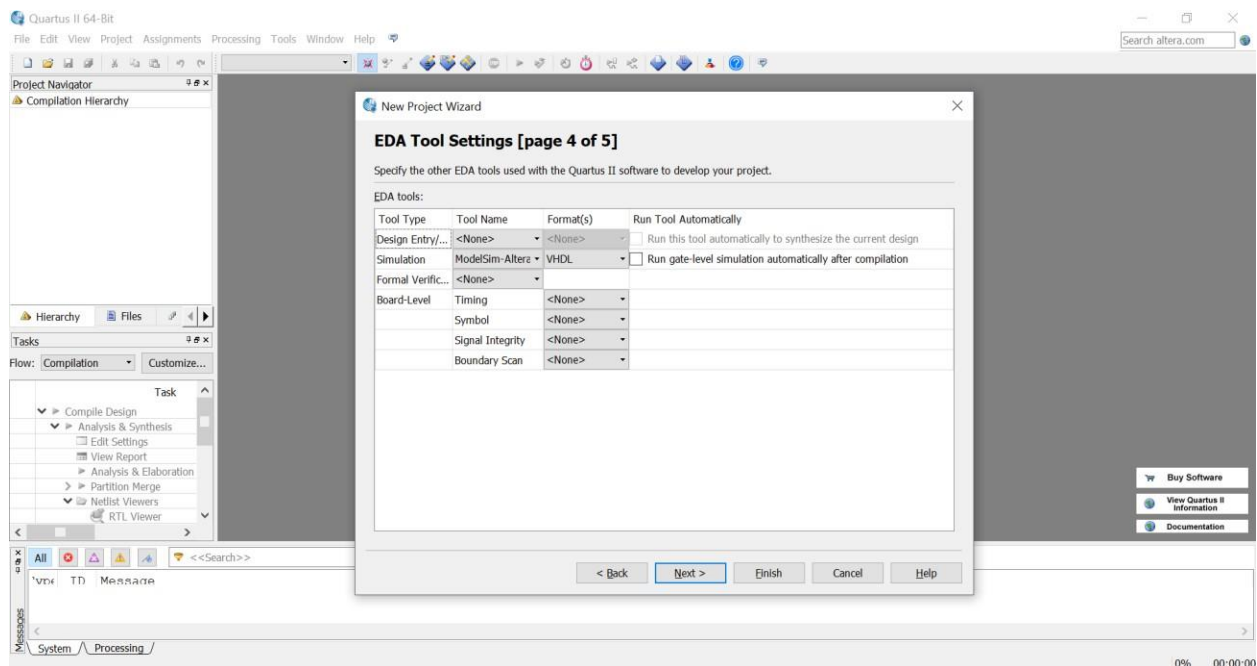


Рис. 1.5 New Project Wizard вибір додаткових пристроїв

Проект створений. Тепер потрібно додати файли, що описують логіку проєкту. Вибираємо File / New

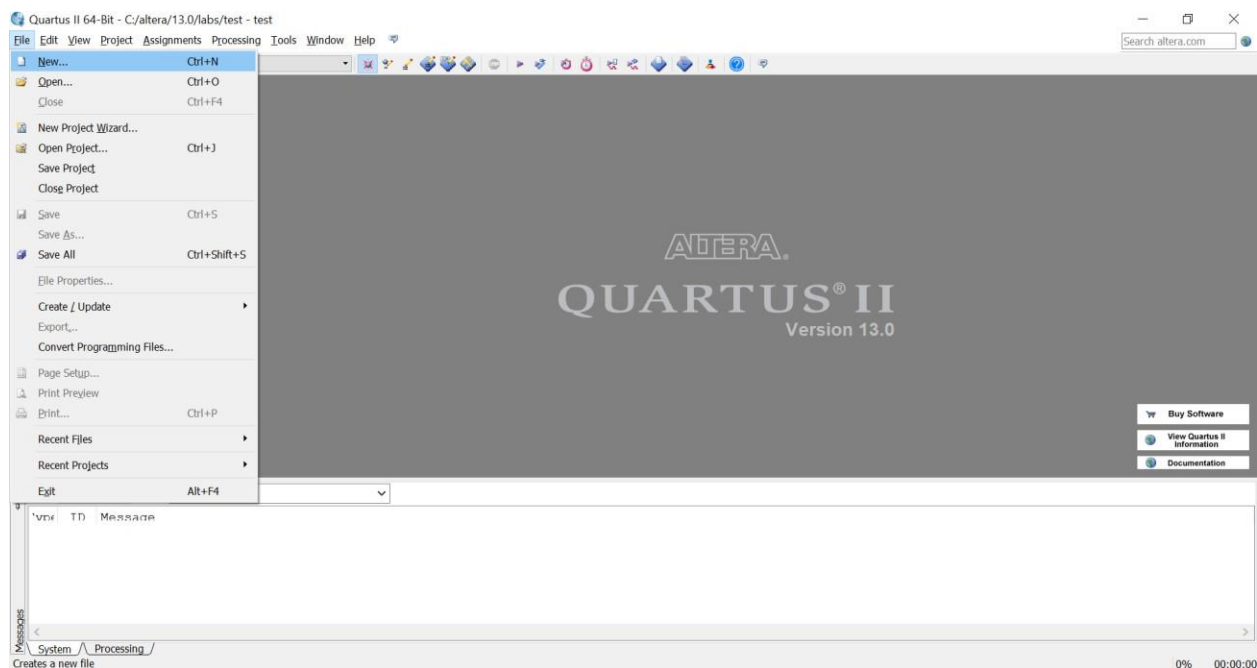


Рис. 1.6. Створення нового файлу

З'явився діалог, де треба вибрати якого типу файл необхідно створити. Вибираємо SystemVerilog HDL File і натискаємо OK.

У створеному файлі і буде написаний наш код. Загальний вид файлу виглядає наступним чином:

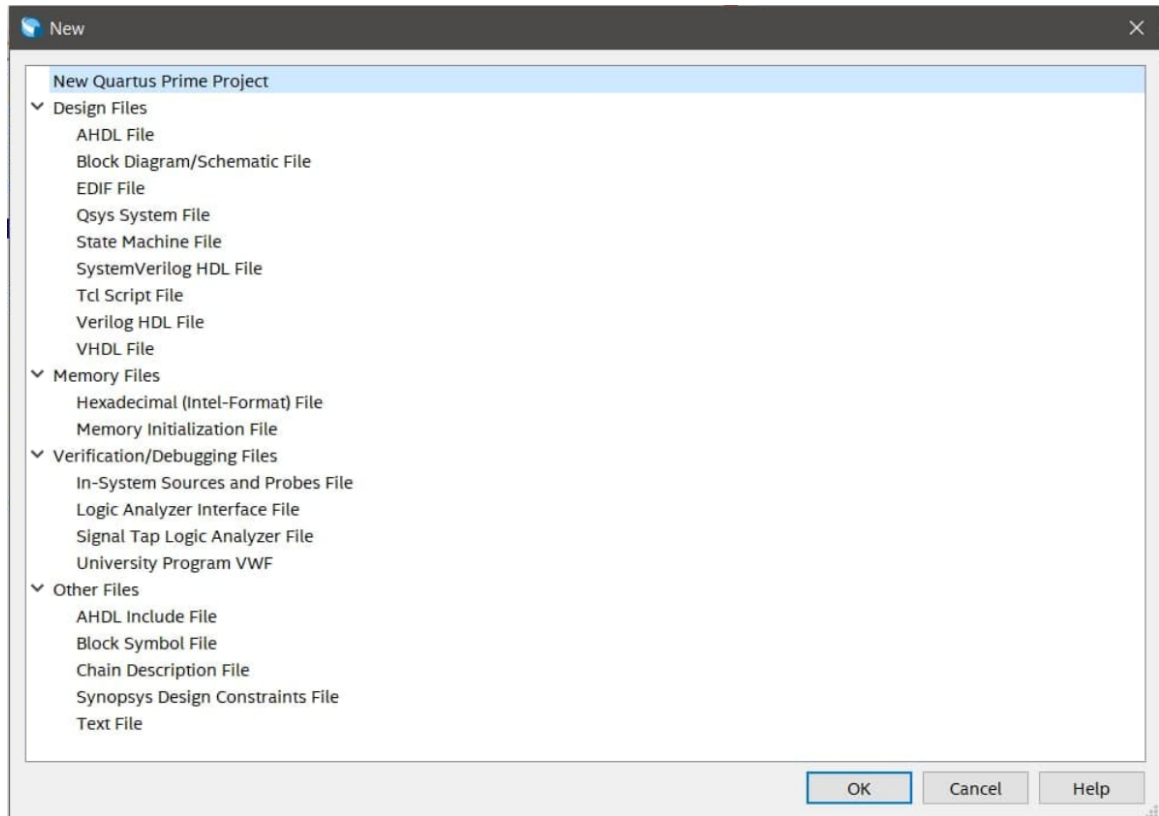


Рис. 1.7. Вид вікна System Verilog HDL

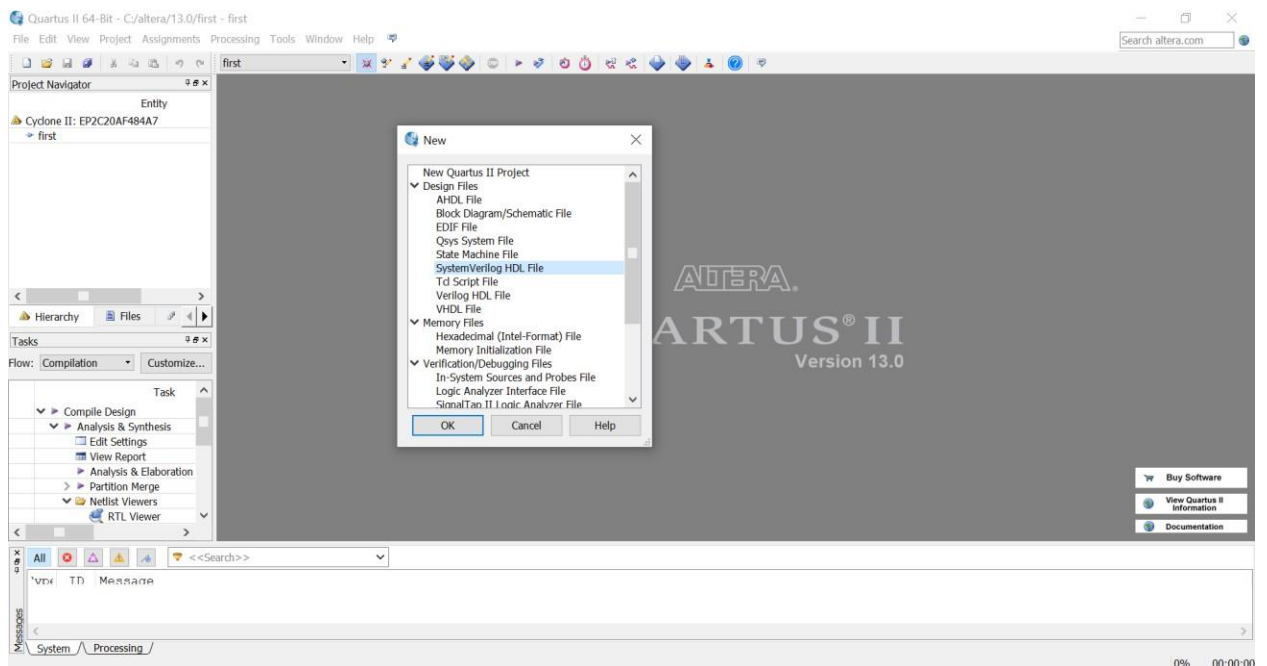


Рис. 1.8. Вибір типу файлу System Verilog HDL

Після написання коду потрібно його зберегти. Через меню File вибираємо Save as.

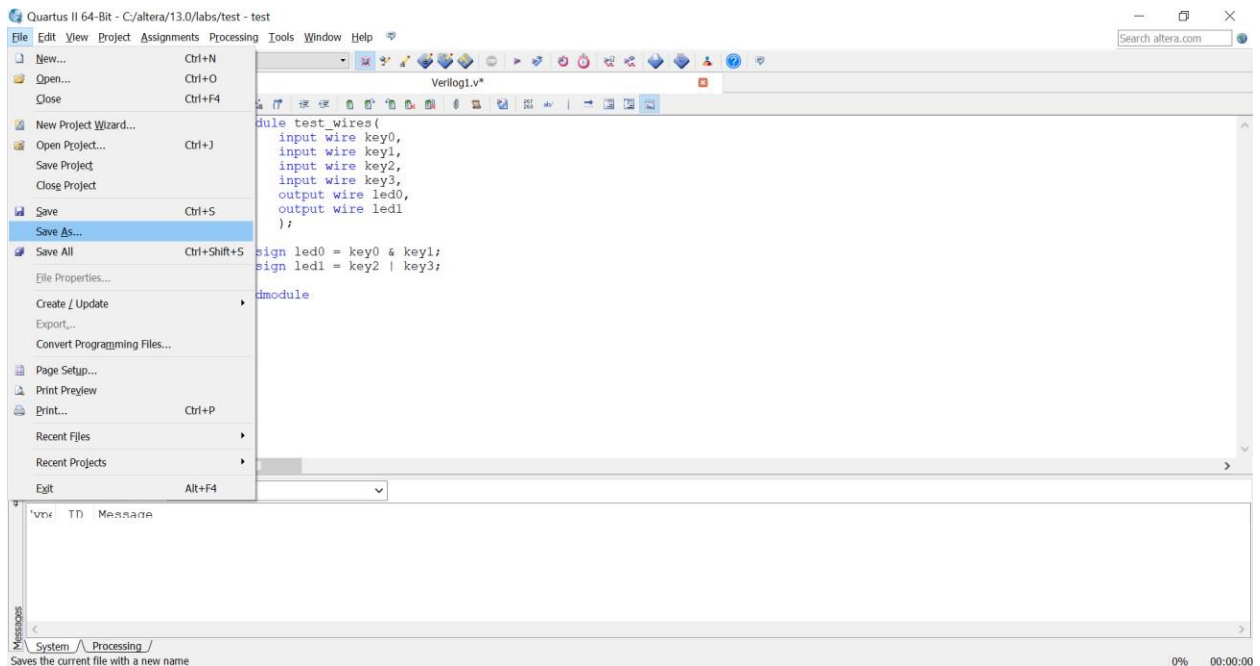


Рис. 1.9. Збереження проекту

Зберігаємо файл під назвою нашого модуля.

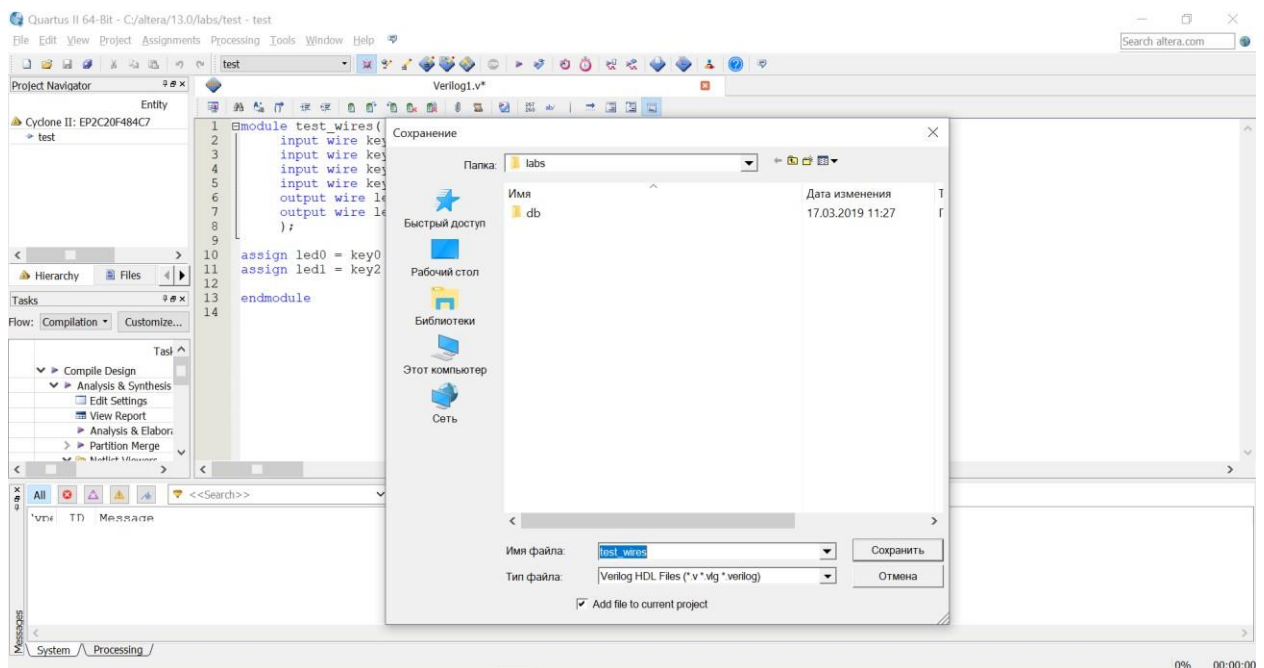


Рис. 1.10. Вибір папки і імені файлу

Проект може складатися з декількох модулів. Головний модуль може включати в себе інші модулі з інших файлів. Тому необхідно вибрати в проекті головний модуль. Через меню Project вибираємо Set As Top Level Entity.

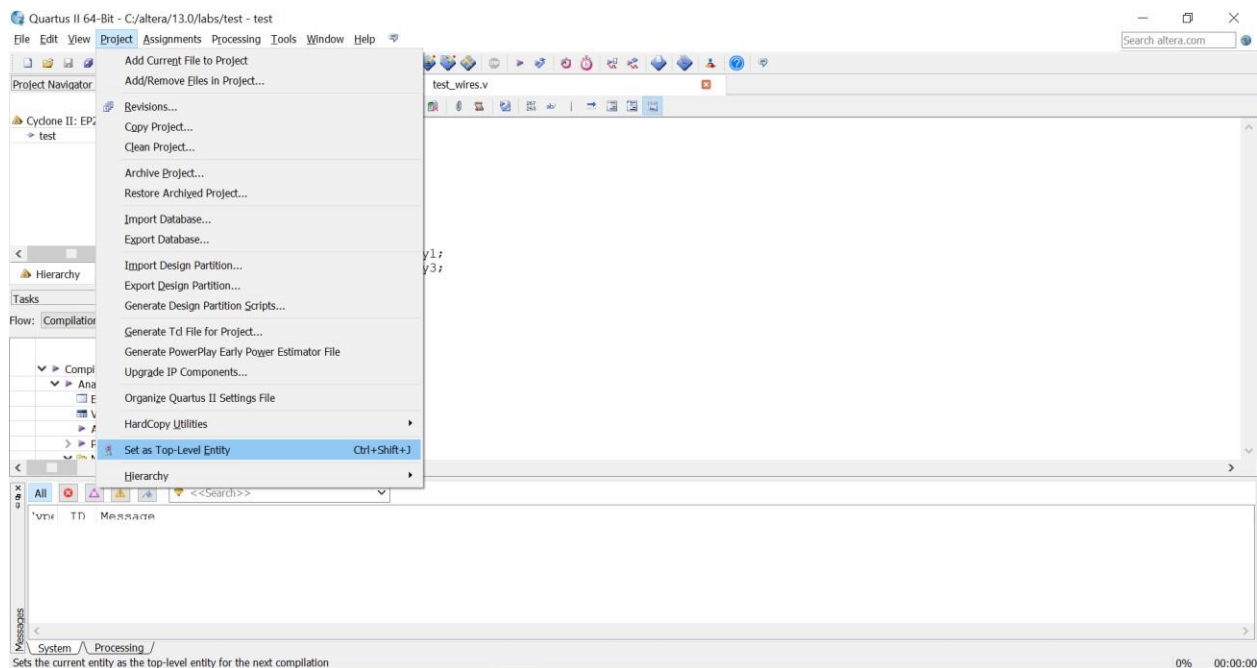


Рис. 1.11. Вибір головного модуля проекту

Аналіз схеми виконується при компіляції проекту. Запустити компілятор можна через меню Processing / Start Compilation як показано на рис. 1.12.

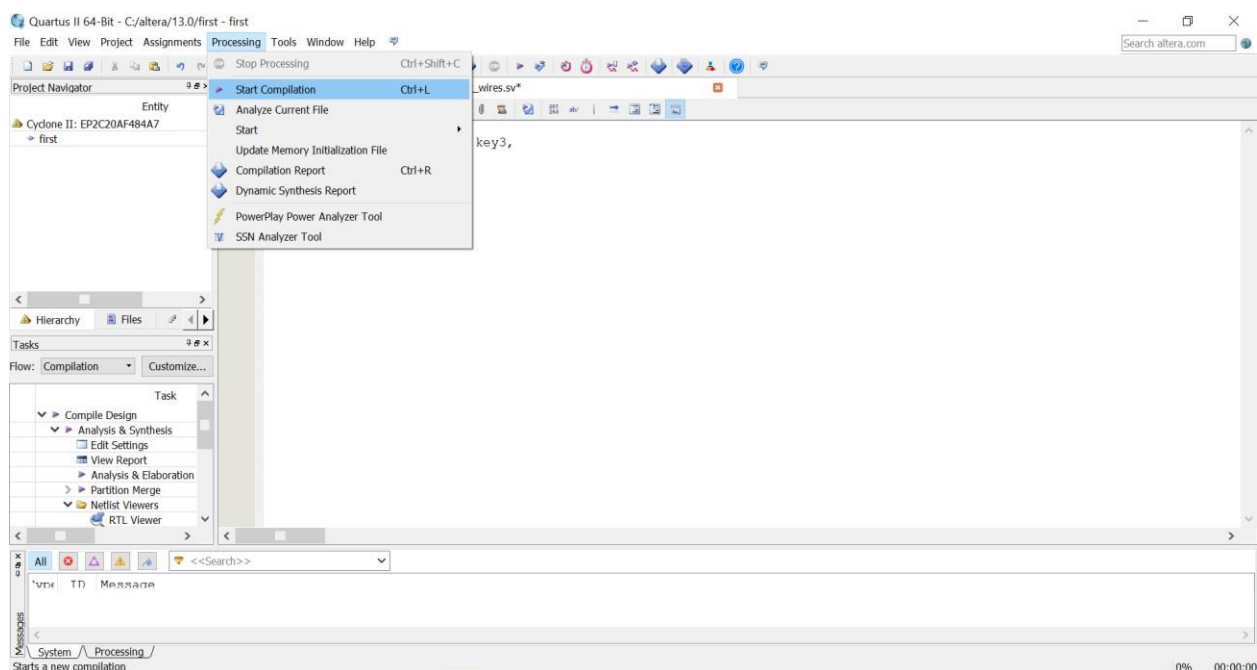


Рис. 1.12. Запуск компілятора

Прогрес компіляції буде відображатися в нижньому лівому вікні Quartus II. Після закінчення процесу компіляції виводиться вікно, де відобразиться успішність або неуспішність виконання компіляції. У нижній частині програми Quartus II є вікно з різними повідомленнями про помилки та попередження.

Узгодження висновків створюється через редактор погоджень Assignments / Assignment Editor:

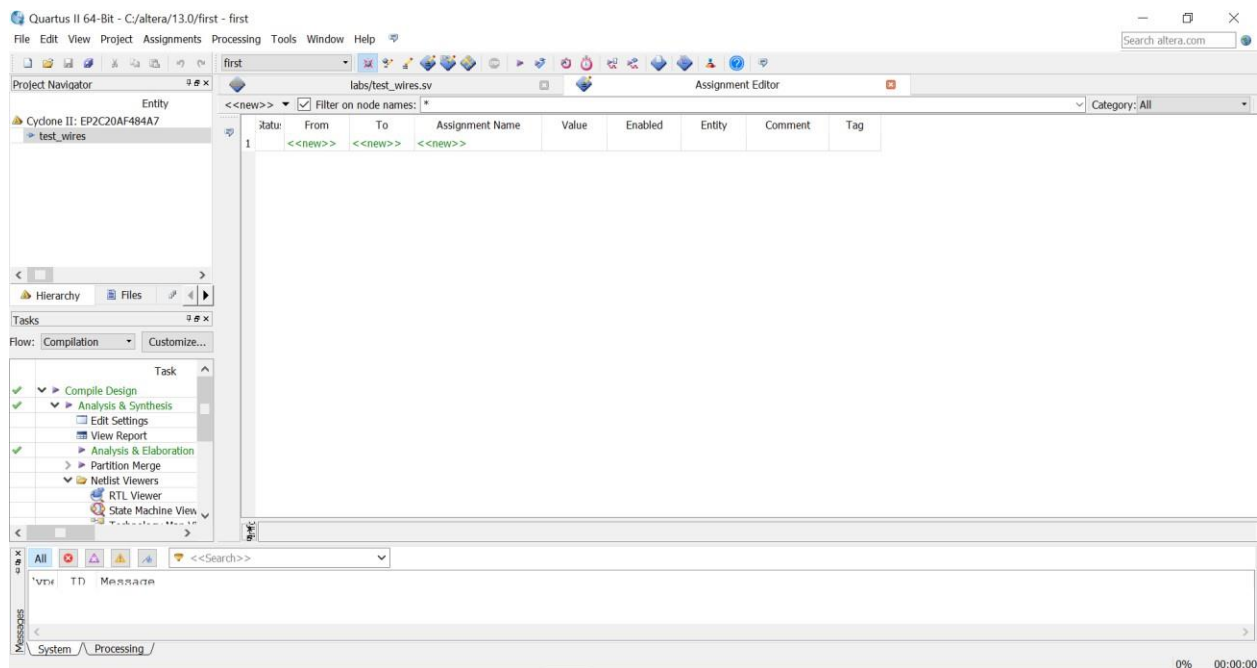


Рис. 1.13. Вікно редактора погоджень введів / висновків

У колонці «To» записується ім'я входів і виходів схеми. У колонці «Assignment Name» вибираємо зі списку настройку Location. У колонці «Value» вписуємо пін контакту мікросхеми. Їх можна дізнатися в документації User Manual, яка поставляється разом з налагоджування платою DE1.

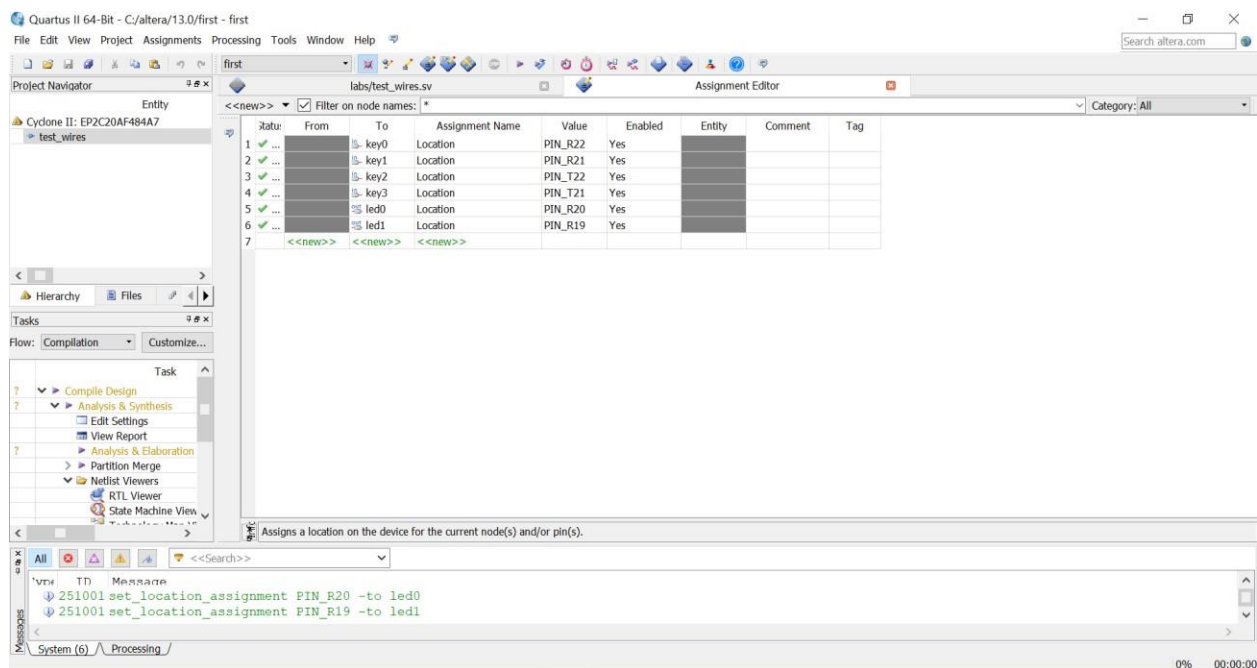


Рис. 1.14. Вікно редактора погоджень введів / висновків з встановленими номерами входів і виходів

Ще раз компілюємо проект. Після успішної компіляції проекту його можна зашити в мікросхему. Для цього запускаємо програматор. Вибираємо пункт меню Tools / Programmer.

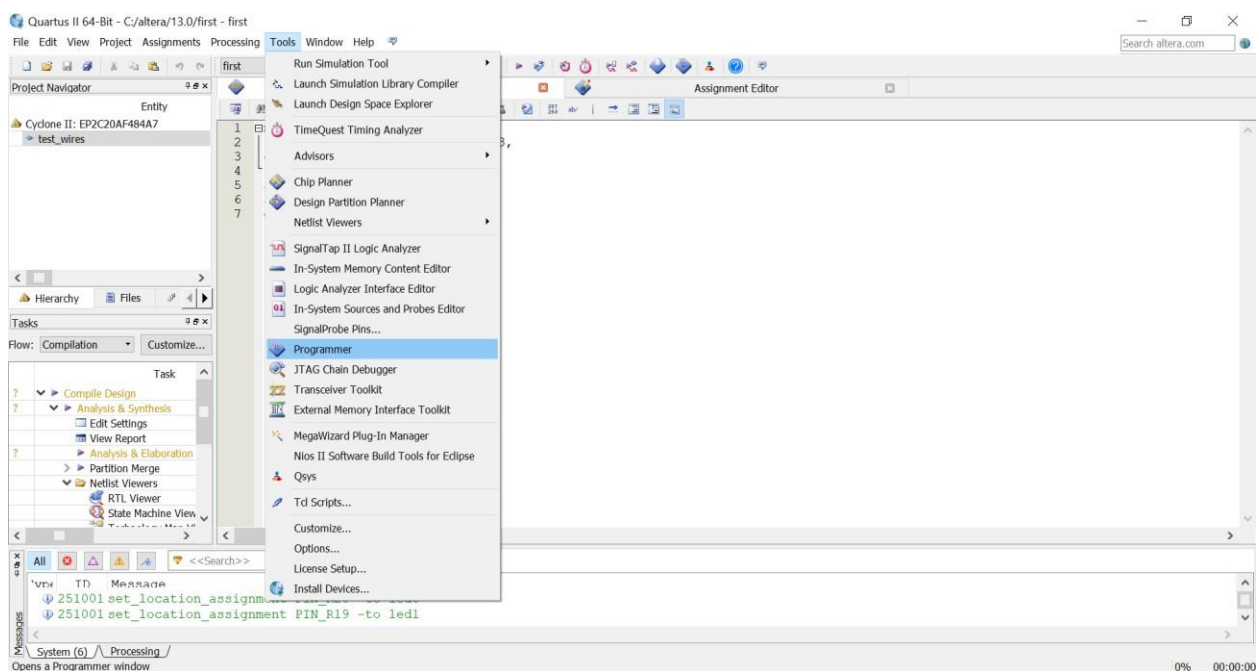


Рис. 1.15. Запуск програматора

Нижче зображено вікно вибору програм. Тут вибирається спосіб зв'язку середовища розробки та налагоджування плати; початок і зупинка процесу зашивання проекту, додавання пристроїв.

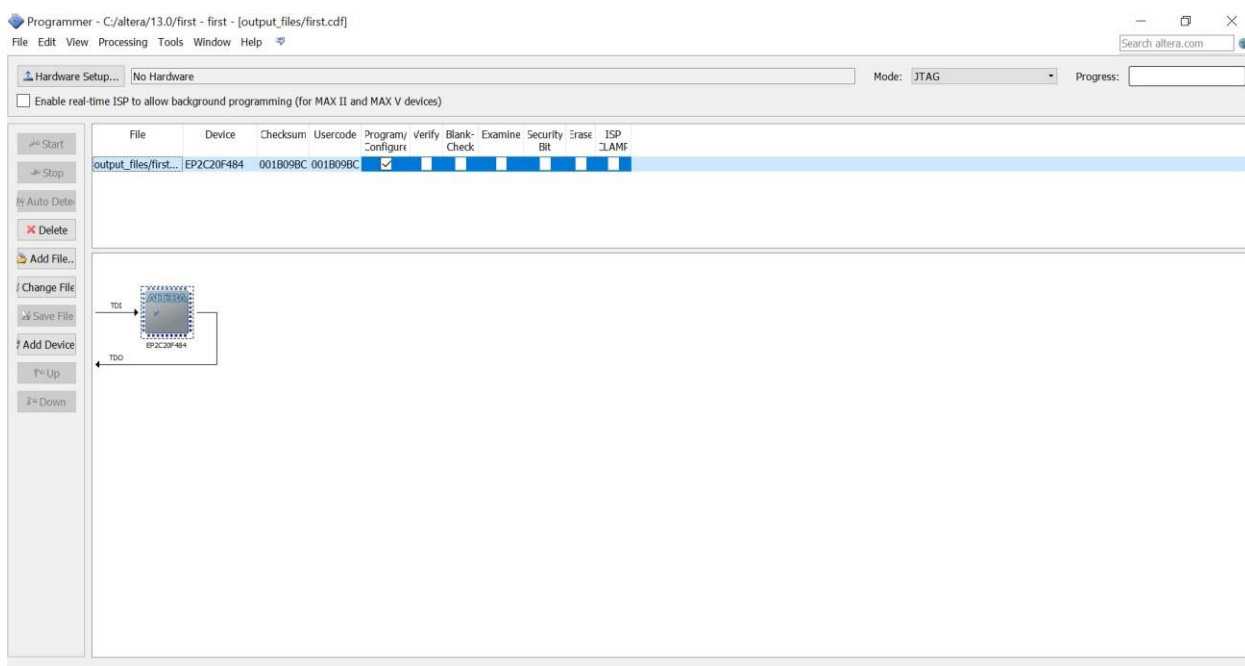


Рис. 1.16. Вікно програматора

Вибираєте проект, який збираєтеся зашити в плату і натискаєте Start. Перед стартом перевірте, чи підключена плата до комп'ютера. Для включення DE1 підключіть в мережу і натисніть кнопку Power Switch.

Розділ 2. Приклади реалізації цифрових пристроїв

2.1. Приклади реалізації цифрових пристроїв на ПЛІС Cyclone II

У цьому розділі будуть представлені приклади програм цифрових пристроїв для програмованої логічної інтегральної схеми Cyclone II фірми Altera. Програми написані на мові опису апаратури SystemVerilog в середовищі розробки апаратури Quartus II.

2.2. Реалізація схеми чотирирозрядного лічильника

У нашому проекті в якості вхідного тактового сигналу використовується кнопка KEY0, вхід скидання – перемикач SW0, вихідний сигнал – 4 червоних світлодіода LEDR0, LEDR1, LEDR2, LEDR3. При натисканні кнопки KEY0 інкрементується лічильник і у вигляді двійкового коду візуалізується на червоних світлодіодах LEDR.

Створюємо проект як це було показано в теоретичній частині. Створюємо файл SystemVerilog і в робочій частині середовища розробки пишемо код опису схеми. Приклад коду представлений нижче.

```
module counter (                                // створення модуля з назвою counter
input logic clk, reset,                        // задаємо входи
output logic [3: 0] q);                       // і виходи
always_ff @ (posedge clk,                    // передньому фронту clk і reset
posedge reset)
begin
if (reset) q = 4'b0000;                       // скидання
else q = q + 4'b0001;                         // рахунок імпульсів end
endmodule                                     // кінець модуля
```

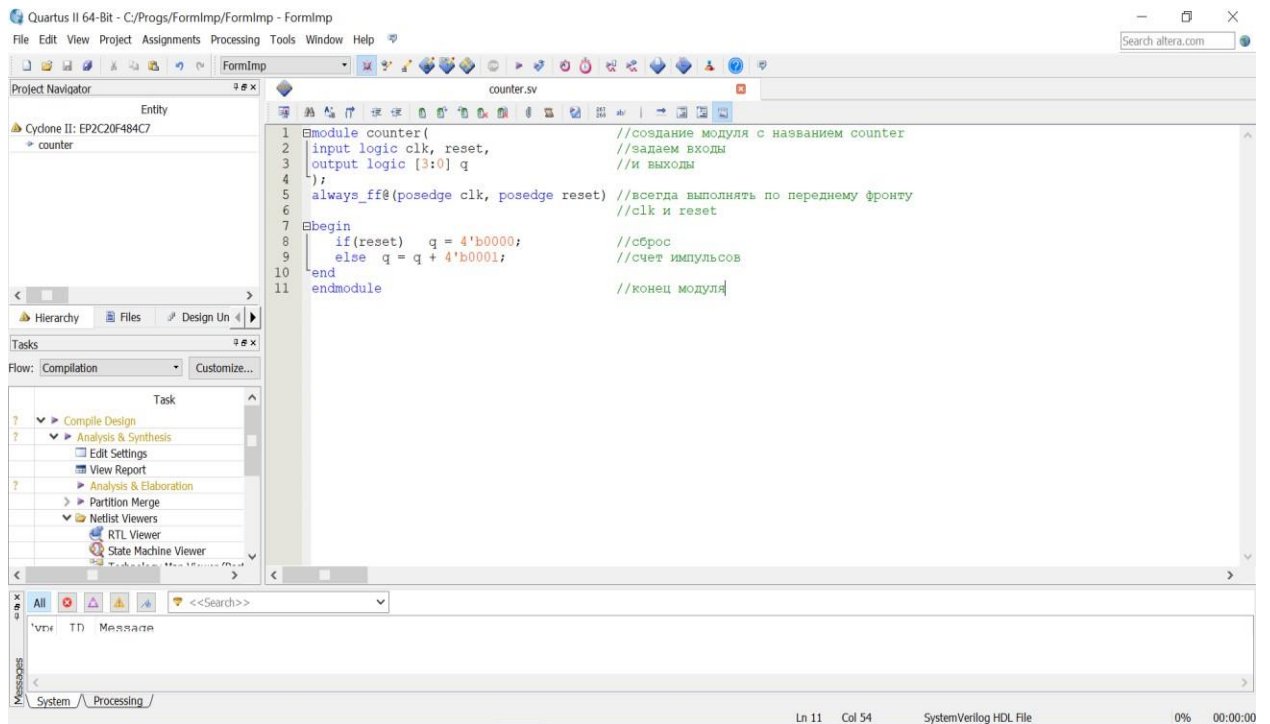


Рис. 2.1. Головне вікно Quartus II з написаним кодом

Пов'язуємо наші входи і виходи з висновками ПЛІС. У редакторі узгодження (Assignments / Assignment Editor) вводів / висновків записуємо Піни відповідних входів і виходів, як показано на рис. 2.2.

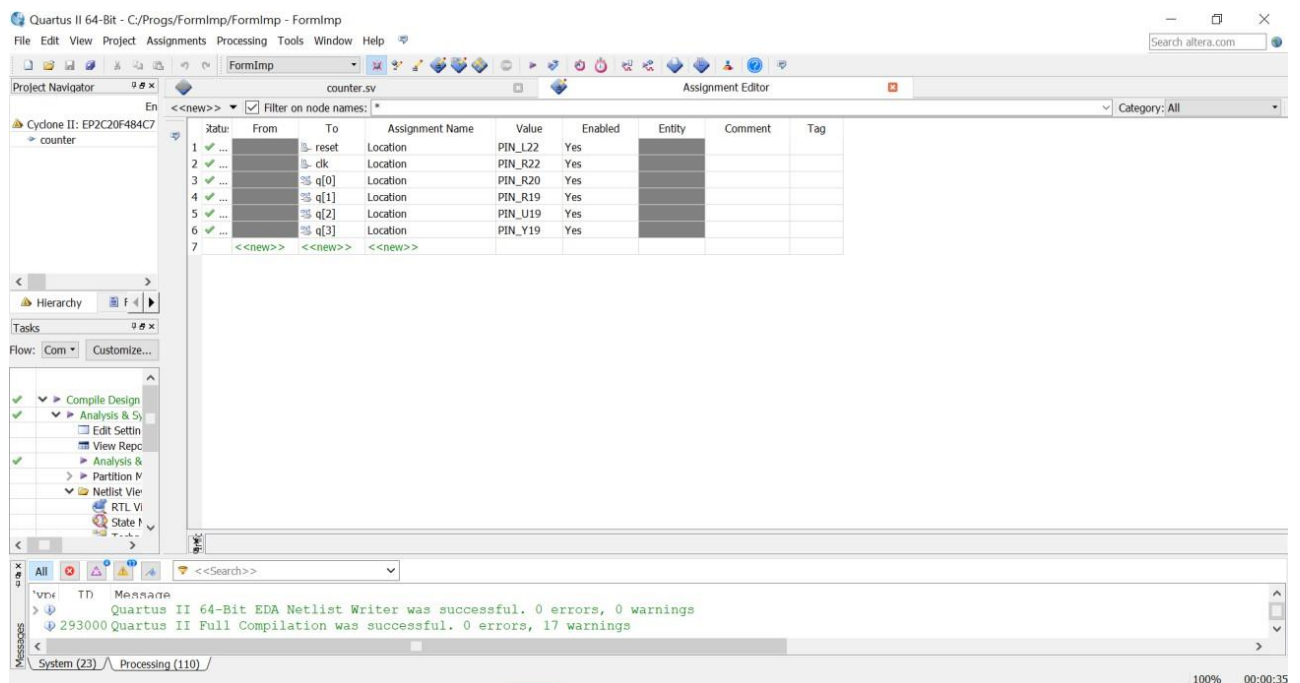


Рис. 2.2. Вікно узгодження входів / висновків

Щоб подивитися схемотехнічне уявлення 4-розрядного лічильника переходимо в меню Tools / Netlist Viewers / RTL Viewer.

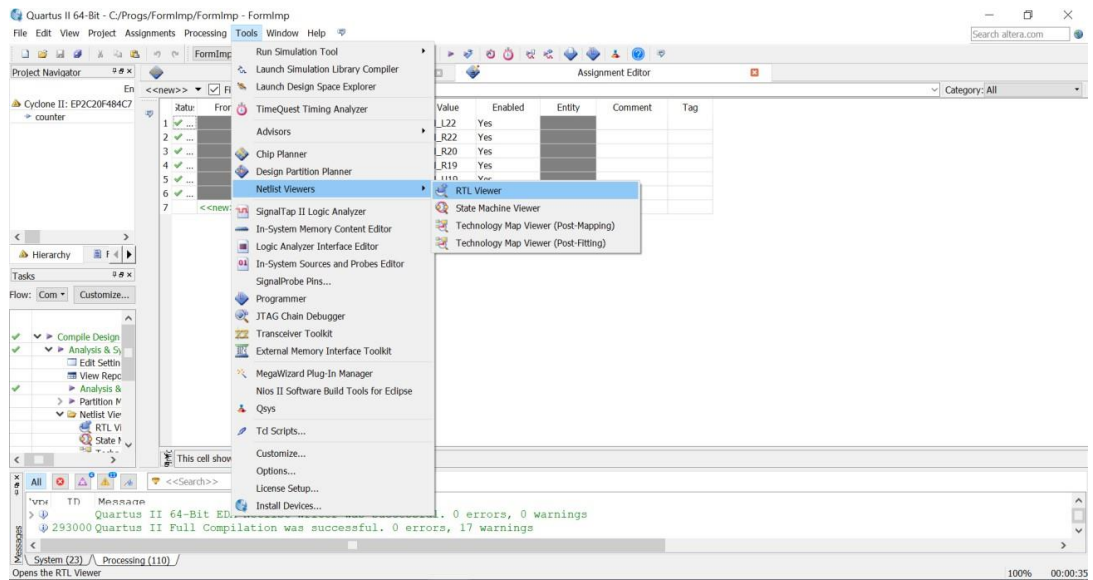


Рис. 2.3. Меню для переходу в RTL Viewer

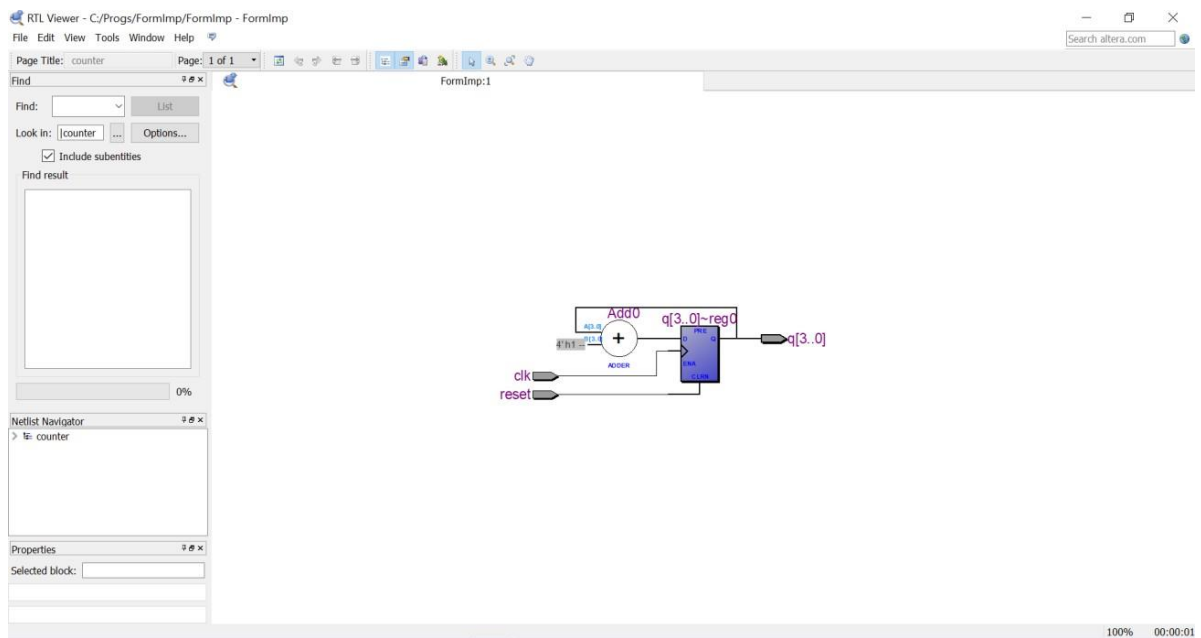


Рис. 2.4. Схемотехнічне уявлення лічильника в RTL Viewer

Зашивка проекту в отладочную плату здійснюється через програматор (Tools / Programmer).

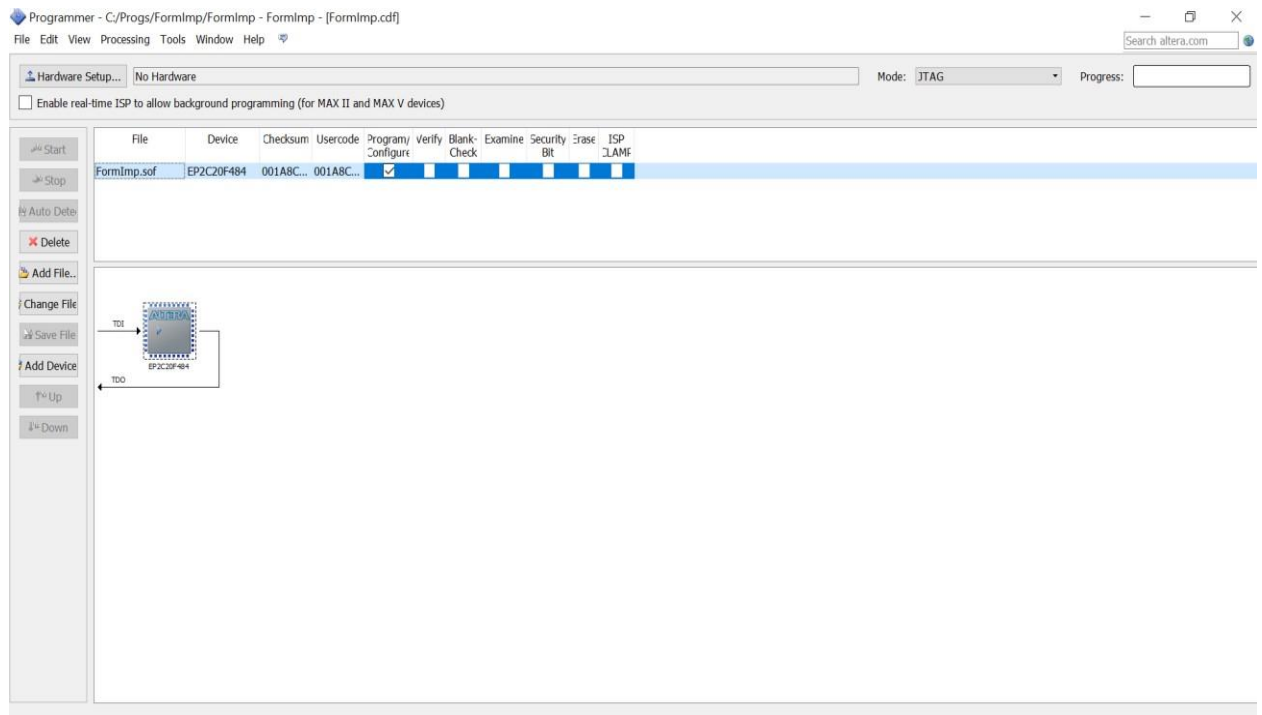


Рис. 2.5. вікно програматора

Нижче представлені зображення роботи лічильника.

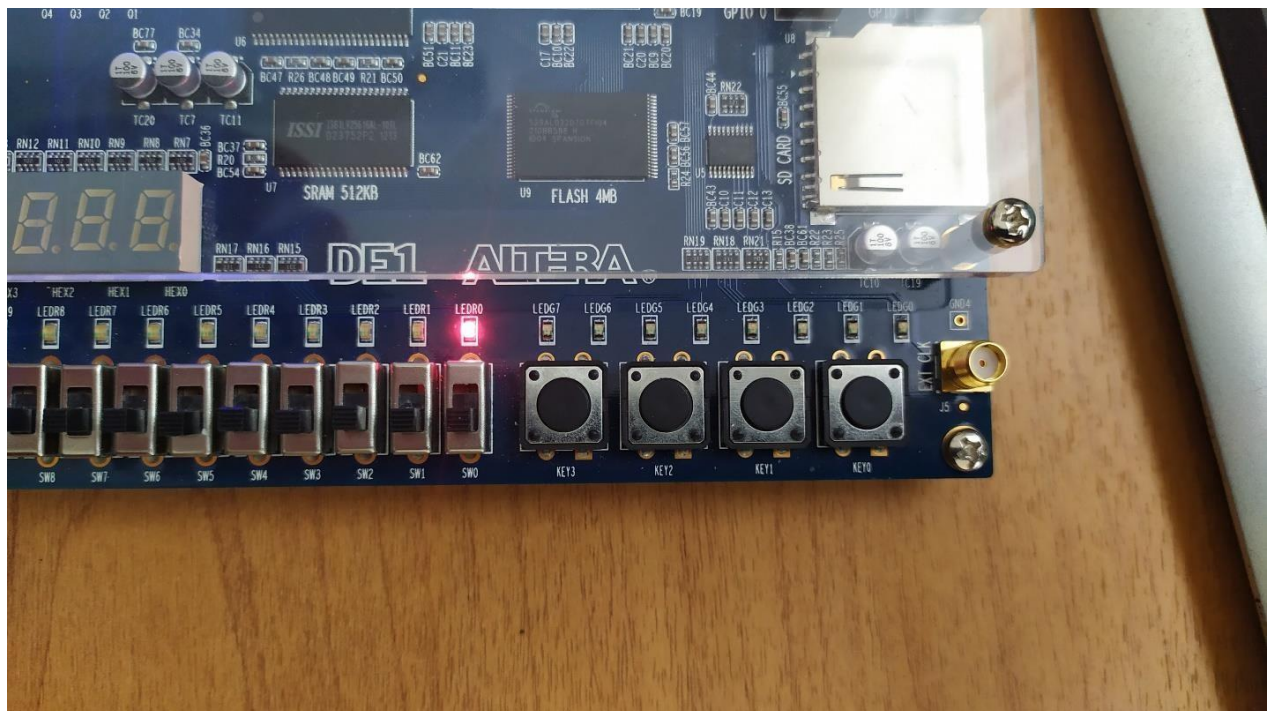


Рис. 2.6. робота лічильника після першого тактового імпульсу

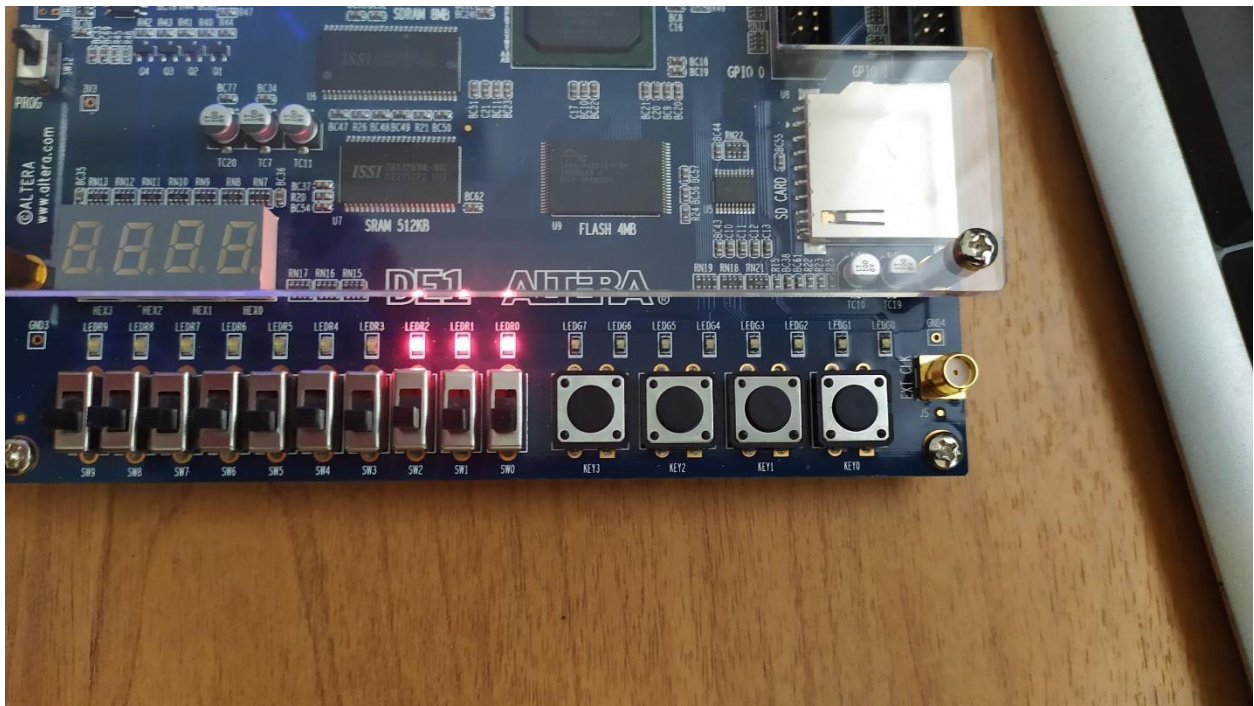


Рис. 2.7. Робота лічильника після сьомого тактового імпульсу

2.3 Реалізація схеми дешифратора

В якості вхідного сигналу використовуються перемикачі SW0, SW1, SW2; вихідний сигнал - 8 червоних світлодіода LEDR0 - LEDR7. Перемикачами задається число в двійковому поданні. Залежно від заданого числа загоряється певний світлодіод.

Створюємо новий проект. Створюємо файл SystemVerilog і в робочій частині середовища розробки пишемо код опису схеми. Приклад коду:

```
module desh (
input logic [2: 0] SW,
output logic [7: 0] LEDR);
always_comb begin case (SW)
3'b000: LEDR = 8'b00000001;
3'b001: LEDR = 8'b00000010;
3'b010: LEDR = 8'b00000100;
3'b011: LEDR = 8'b00001000;
3'b100: LEDR = 8'b00010000;
3'b101: LEDR = 8'b00100000;
3'b110: LEDR = 8'b01000000;
3'b111: LEDR = 8'b10000000;
endcase
end
endmodule
```

// створення модуля з ім'ям desh
// вказуємо входи
// і виходи
// кожному вхідному сигналу
// вказуємо на якому з виходів
// горітиме світлодіод
// кінець модуля

У редакторі узгодження вводів / висновків записуємо Піни відповідних входів і виходів, як показано на рис. 2.8.

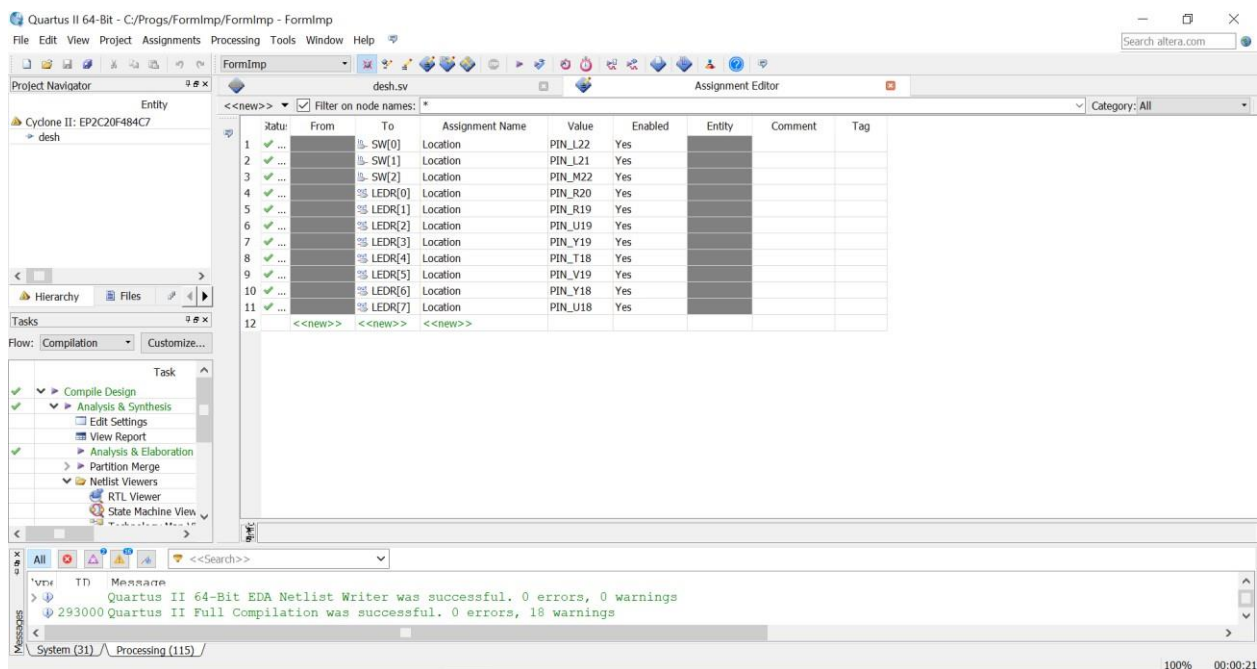


Рис. 2.8. Вікно узгодження вводів / висновків

Схемотехнічне уявлення дешифратора 3 на 8 в RTLViewer показано на рис. 2.9

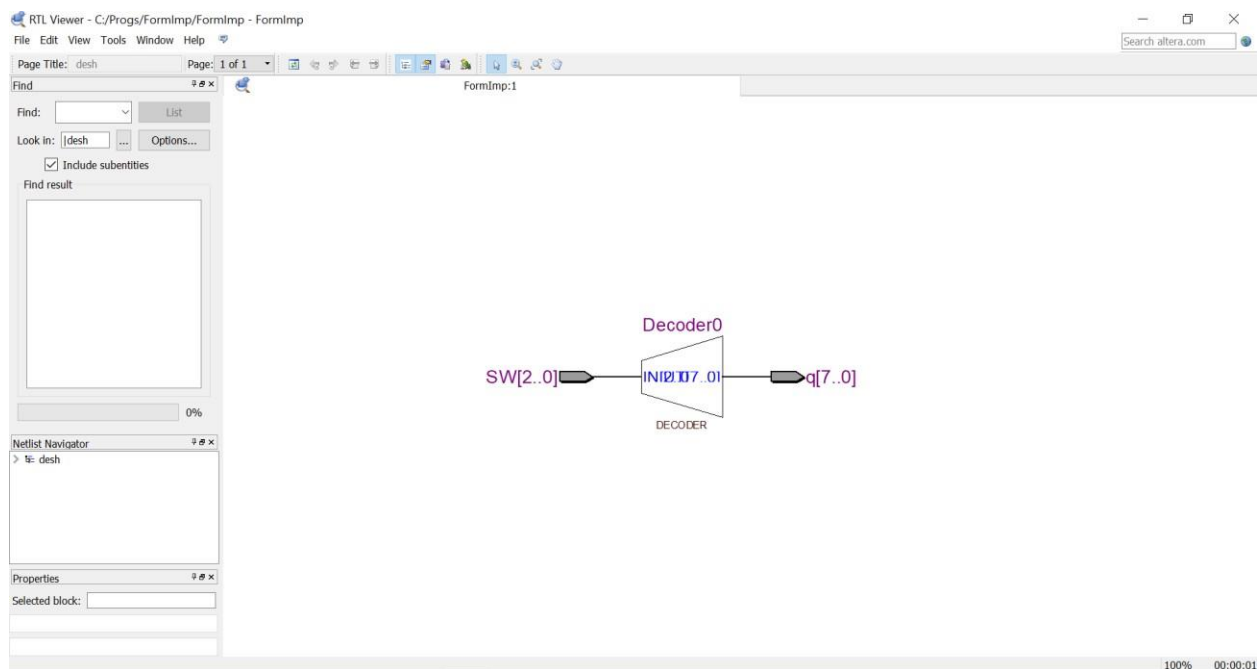


Рис. 2.9. Схемотехнічне уявлення дешифратора в RTL Viewer

На рис. 2.10 і рис. 2.11 представлені результати роботи дешифратора на отладоочной платі DE1.

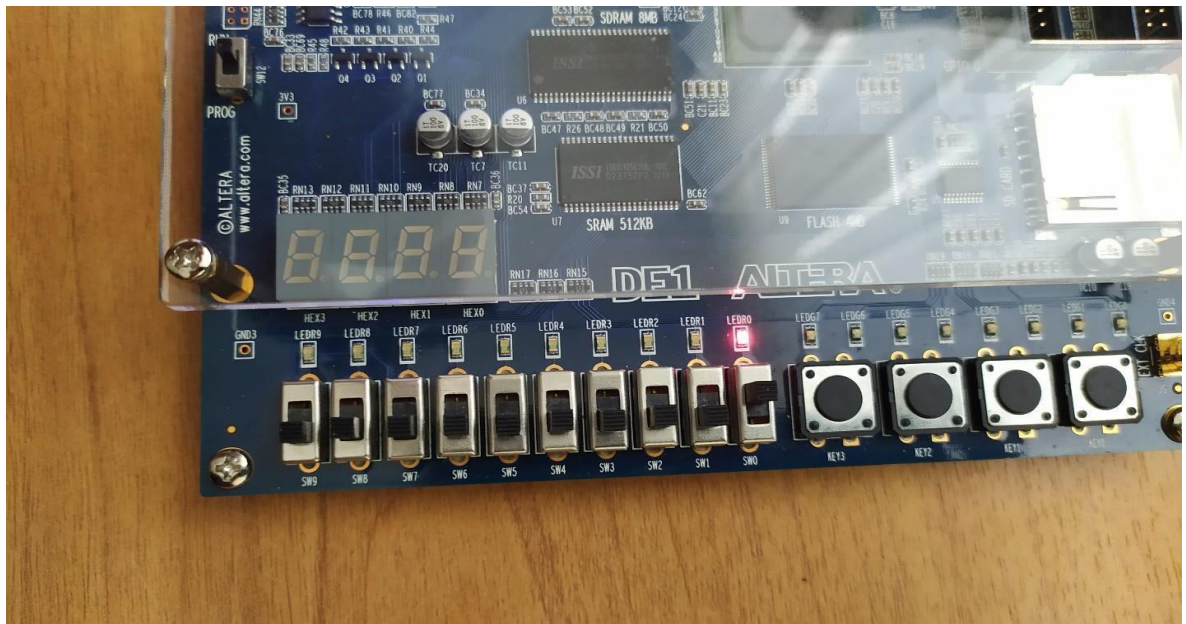


Рис. 2.10. Візуалізація роботи дешифратора при подачі на вхід схеми одиниці в двійковому поданні

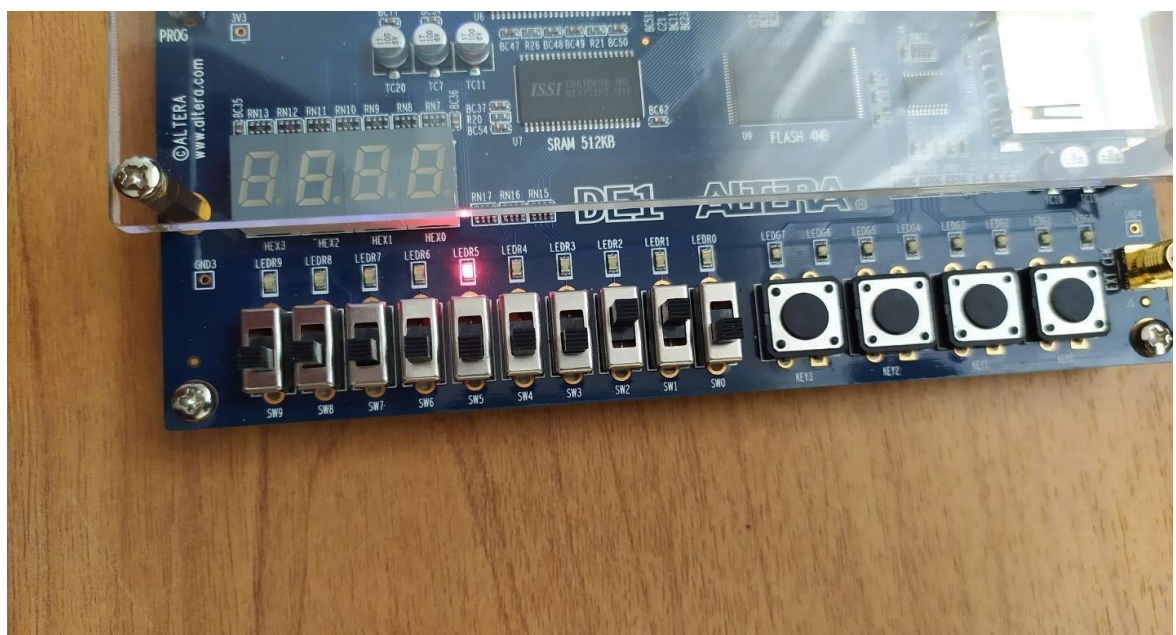


Рис. 2.11. Візуалізація роботи дешифратора при подачі на вхід схеми числа 6 в двійковому поданні

2.4 Реалізація схеми розподільника імпульсу заданої форми

Як генератор імпульсів використовується вбудований в плату осцилятор з частотою 50 МГц. За допомогою осцилографа спостерігаємо за вихідними сигналами. Осцилограф підключається до роз'ємів GPIO0.

Створюємо новий проект. Створюємо файл SystemVerilog і в робочій частині середовища розробки пишемо код опису схеми. Приклад коду:

module FormImp (// створюємо модуль з ім'ямFormImp
input logic clk, reset,	// встановлюємо входи
output logic [7: 0] GPIO);	// і виходи
logic [3: 0] i;	// створюємо реєстр 4 біт для зберігання
logic [15: 0] X;	// стану лічильника
	// створюємо реєстр 16 біт для зберігання
always_ff @ (posedge clk,	// стану дешифратора
posedge reset)	// створюємо лічильник
begin	
if (reset) i = 4'b0000; else	
i = i + 4'b0001;	
end	
always_comb	// створюємо дешифратор
begin	
case (i)	
4'b0000: X =	
16'b00000000000000001;	
4'b0001: X =	
16'b00000000000000010;	
4'b0010: X =	
16'b00000000000000100;	
4'b0011: X =	
16'b00000000000001000;	
4'b0100: X =	
16'b00000000000010000;	
4'b0101: X =	
16'b00000000000100000;	
4'b0110: X =	
16'b00000000010000000;	
4'b0111: X =	
16'b00000000100000000;	
4'b1000: X =	
16'b00000001000000000;	
4'b1001: X =	
16'b00000010000000000;	
4'b1010: X =	

```

16'b000000100000000000;
4'b1011: X =
16'b000010000000000000;
4'b1100: X =
16'b000100000000000000;
4'b1101: X =
16'b001000000000000000;
4'b1110: X =
16'b010000000000000000;
4'b1111: X =
16'b100000000000000000;
endcase
end
always_ff @ (posedge X [0], // входи тригера підключаємо
posedge X [8]) // до 0-му
begin // і 8-му виходу дешифратора
// установка
if (X [0]) GPIO [0] <= 1;
else if (X [8]) GPIO [0] <=
0;
// скидання
end
always_ff @ (posedge X [1], // входи тригера підключаємо
posedge X [9]) // до 1-му
begin // і 9-му виходу дешифратора
// установка
if (X [1]) GPIO [1] <= 1;
else if (X [9]) GPIO [1] <=
0;
// скидання
end
always_ff @ (posedge X [2], // входи тригера підключаємо
posedge X [10]) // до 2-му і 10-му
begin // виходу дешифратора
// установка
if (X [2]) GPIO [2] <= 1;
else if (X [10]) GPIO [2] <=
0;
// скидання
end
always_ff @ (posedge X [3], // входи тригера підключаємо
posedge X [11]) // до 3-му і 11-му
begin // виходу дешифратора
// установка
if (X [3]) GPIO [3] <= 1;

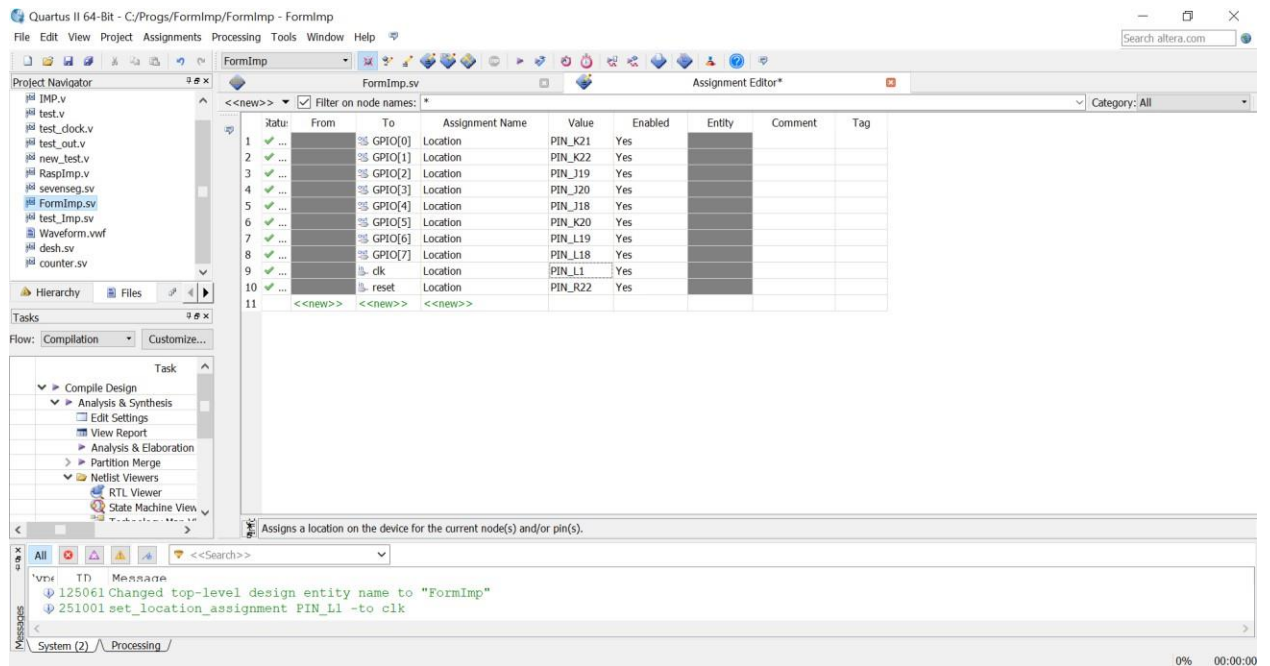
```

```

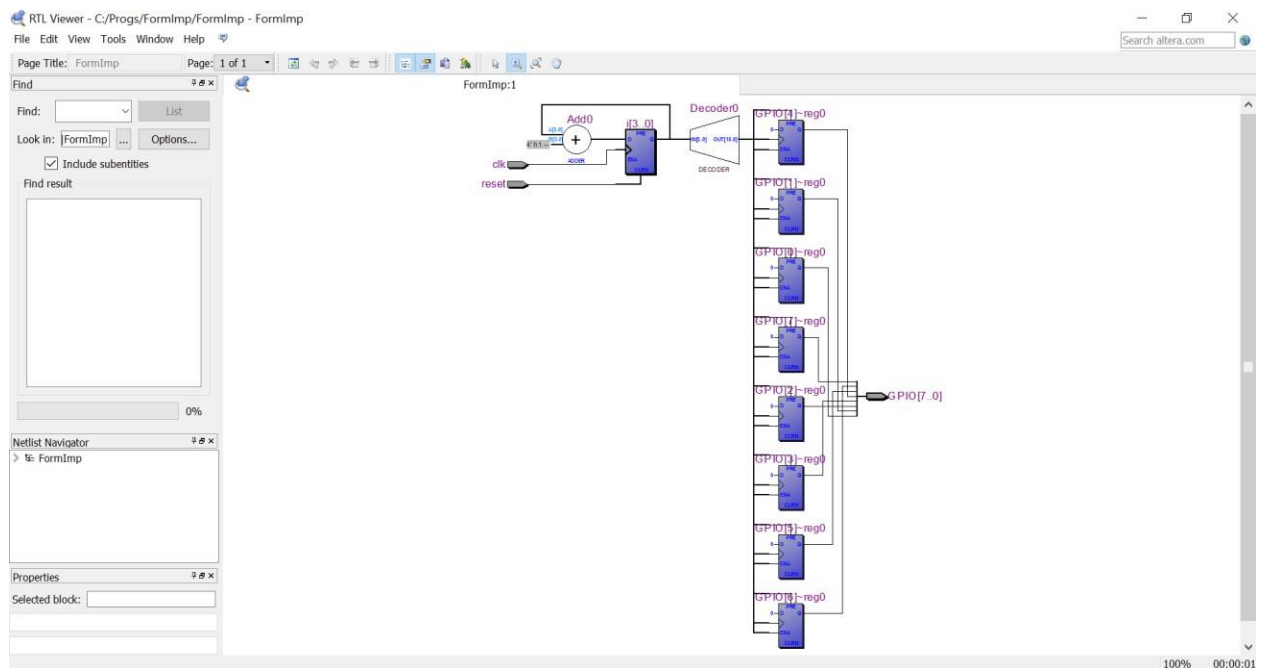
else if (X [11]) GPIO [3] <= // скидання
0;
end
always_ff @ (posedge X [4], // входи тригера підключаємо
posedge X [12]) // до 4-му і 12-му
begin // виходу дешифратора
if (X [4]) GPIO [4] <= 1; // установка
else if (X [12]) GPIO [4] <= // скидання
0;
end
always_ff @ (posedge X [5], // входи тригера підключаємо
posedge X [13]) // до 5-му і 13-му
begin // виходу дешифратора
if (X [5]) GPIO [5] <= 1; // установка
else if (X [13]) GPIO [5] <= // скидання
0;
end
always_ff @ (posedge X [6], // входи тригера підключаємо
posedge X [14]) // до 6-му і 14-му
begin // виходу дешифратора
if (X [6]) GPIO [6] <= 1; // установка
else if (X [14]) GPIO [6] <= // скидання
0;
end
always_ff @ (posedge X [7], // входи тригера підключаємо
posedge X [15]) // до 7-му і 15-му
begin // виходу дешифратора
if (X [7]) GPIO [7] <= 1; // установка
else if (X [15]) GPIO [7] <= // скидання
0;
end
// кінець модуля
endmodule

```

У редакторі узгодження вводів / висновків записуємо Піни відповідних входів і виходів, як показано на рис. 2.12.



Схемотехнічне уявлення розподільника імпульсів в RTLViewer показано на рис. 2.13.



Кабель осцилографа з'єднуємо так - плюс кабелю з GPIO_0, а мінус з клемою GRD. Плюс іншого кабелю осцилографа з'єднуємо, наприклад, з GPIO_0, а мінус також з клемою GRD.

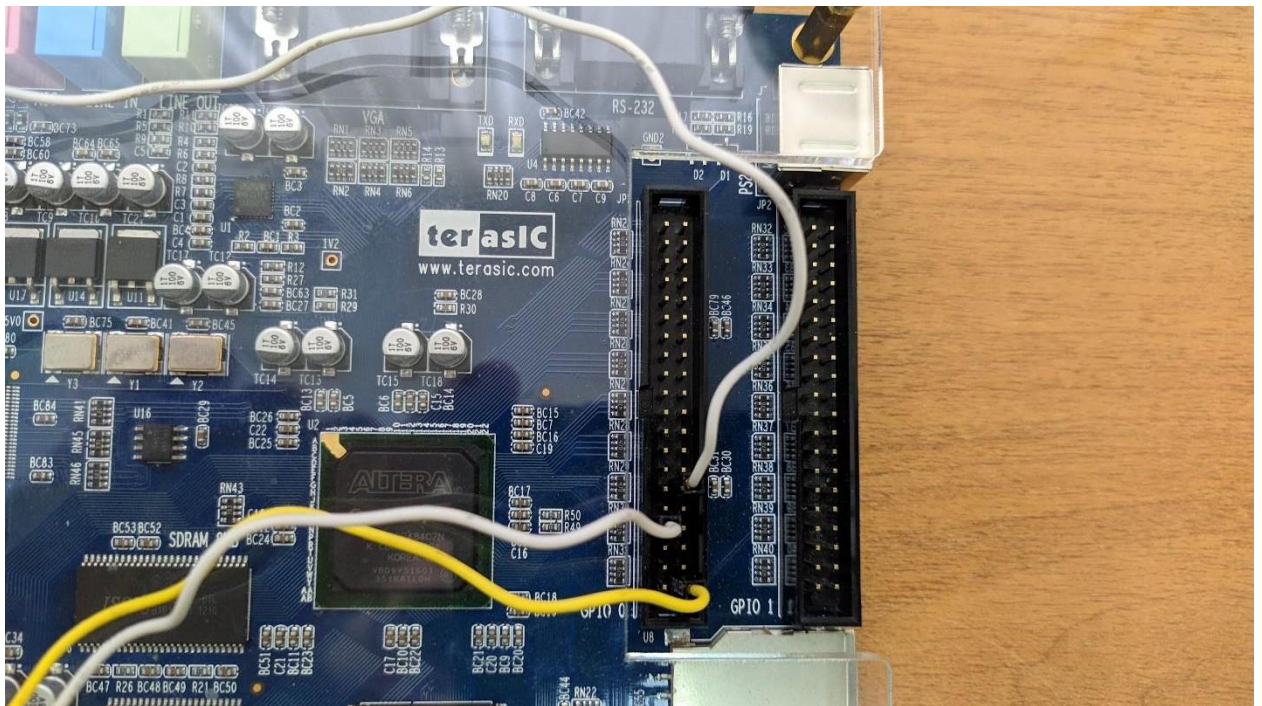


Рис. 2.14. Підключення проводів до контактів GPIO

Осцилограми вихідних сигналів представлені на рис. 2.15 і 2.16.

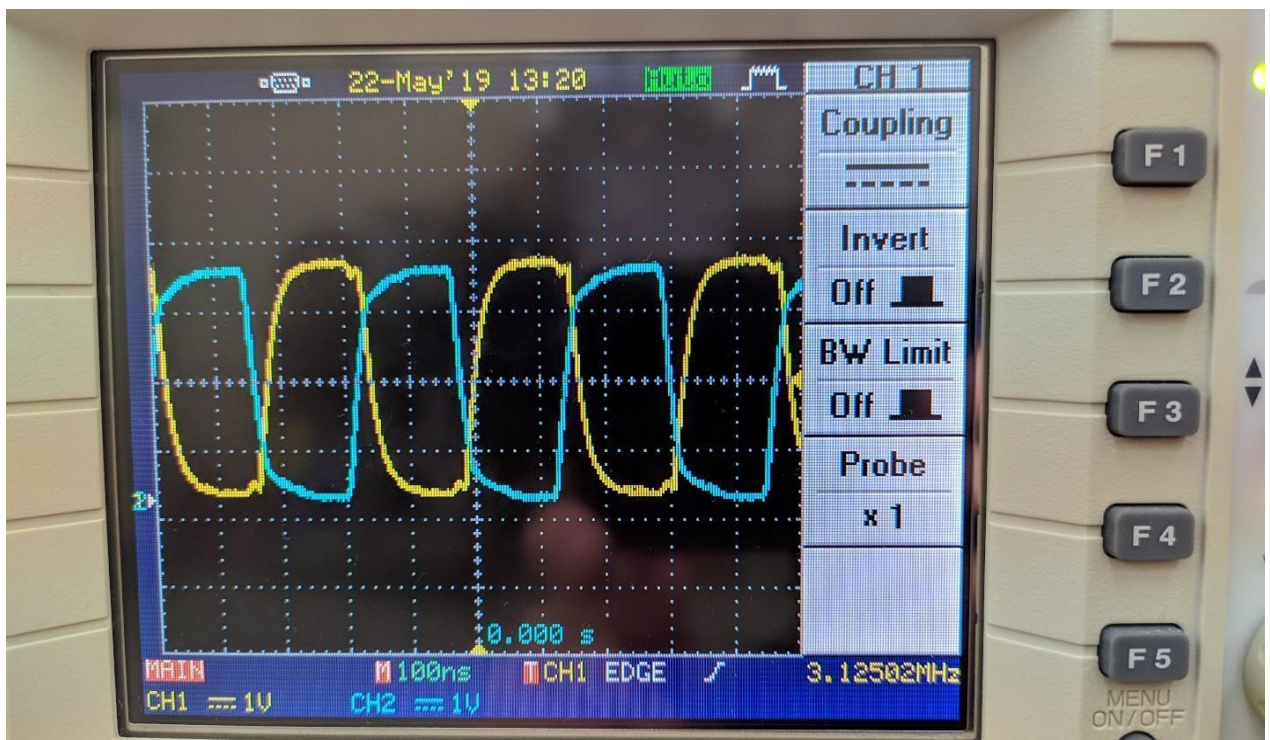


Рис. 2.15. Осцилограми імпульсів з виходів GPIO [0] і GPIO [7]

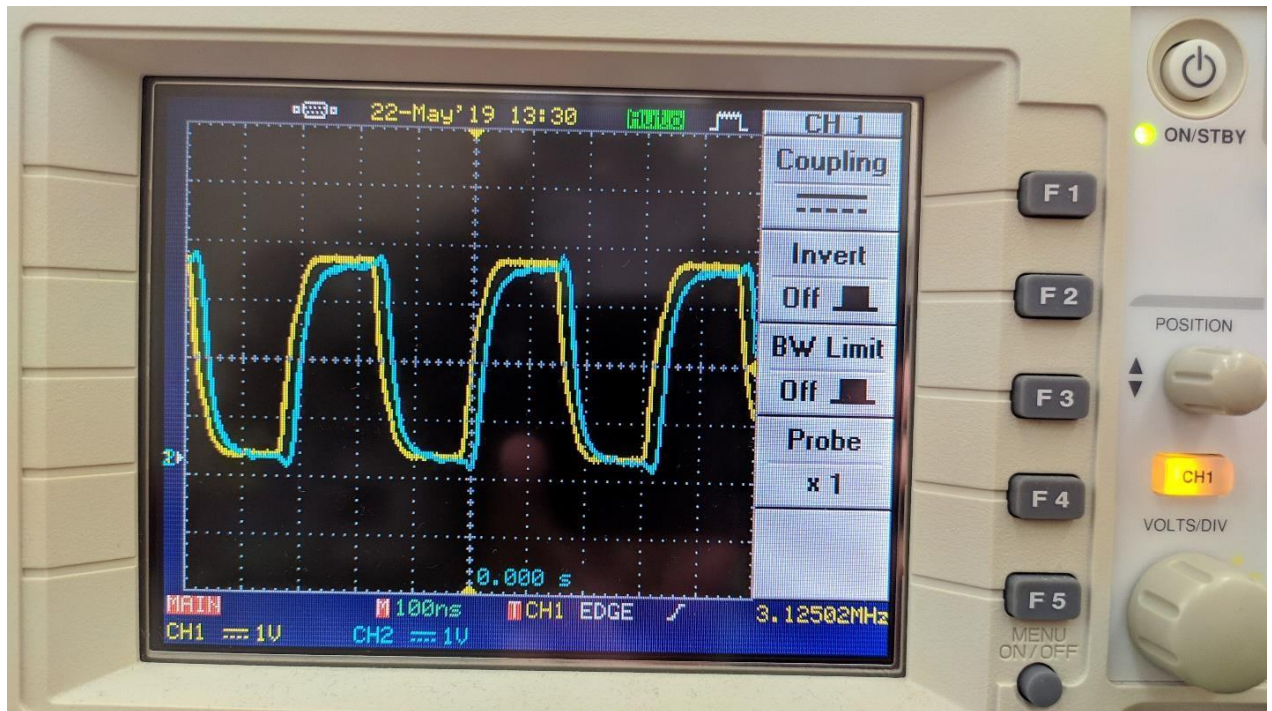


Рис. 2.16. Осцилограми імпульсів з виходів GPIO [3] і GPIO [4]

Висновки

В ході виконання даної роботи було зроблено наступне:

1. Вивчено отладочная плата DE1 на ПЛІС Cyclone IV.
2. Освоєно середовище розробки Altera Quartus II і мова апаратного програмування SystemVerilog HDL.
3. Реалізовано на мові SystemVerilog:
 - Лічильник;
 - Дешифратор;
 - Розподільник імпульсів.

Таким чином, створений лабораторний практикум на Cyclone IV. Ця робота буде допомагати студентам опановувати мову опису апаратури SystemVerilog HDL і середовище розробки Quartus II.

Розділ 3. Лабораторні роботи

3.1. Лабораторна робота № 1

Тема: Операція додавання

Постановка задачі:

Об'єкт дослідження: арифметико-логічний пристрій.

Мета роботи: спроектувати комбінаційні схеми для роботи алгоритмічно-логічного пристрою.

Методи дослідження та апаратура: формалізація АЛП за допомогою середовища проектування Quartus та мови опису апаратури SystemVerilog.

Арифметико-логічний пристрій (англ. Arithmetic Logic Unit, ALU) — блок процесора, що служить для виконання арифметичних та логічних перетворень над даними, що іменуються операндами. Цей пристрій є фундаментальною частиною будь-якого обчислювача, навіть найпростіші мікроконтролери мають його в складі свого ядра. Центральний процесор та відеопроцесор можуть мати кілька АЛП, що відрізняються своїм функціональним призначенням або типом оброблюваних даних.

Результати: реалізовано арифметико-логічний пристрій для виконання базових математичних операцій двох 8-бітних цілочисельних операндів.

Завданням даної лабораторної роботи є розробка архітектурного рішення для виконання операції додавання в середовищі Quartus Prime 16.1., а саме побудова багаторозрядного (8-бітного) суматора.

Вхідними даними є два восьми бітні числа(що задані у бінарній системі числення та є цілими).

Результатом повинно буде одне восьми-бітне бінарне число та один біт переповнення.

Результатом також має бути вихідний код Verilog HDL та вентильна схема, які якраз і будуть забезпечувати правильне виконання поставленого завдання.

Завдання на лабораторну роботу:

Розробити архітектурне рішення для виконання операції додавання. Вхідними аргументами операції мають бути два восьми-бітні числа, що задані у бінарній системі числення, є цілими і не мають знаку. Результатом є одне восьми-бітне бінарне число та один біт переповнення.

Розробка має бути виконана у вигляді вентильної схеми і програмного коду Verilog HDL. Вхідні аргументи повинні бути подані у вигляді логічних ліній a[7:0],

b[7:0]. Вихідне значення операції повинне бути подано у вигляді y[7:0], of[0].

Працездатність архітектури і розробки повинні бути показані у вигляді процесу роботи логічної вентиляційної схеми та у вигляді результатів її роботи, що мають відповідати відповідним булевим законам.

Опис результатів, а також методів, підходів і засобів розробки повинні бути подані у вигляді звіту. Оформлення і вмісту звіту мають відповідати ДСТУ3008-95.

Розробка може проводитися у таких середовищах: MatLab SImulink; Multisim (EWB); ISIS Proteus; Quartus II.

Склад звіту лабораторної роботи

В звіті повинно бути відображено:

1. Тема роботи.
2. Мета роботи.
3. Постановка задачі.
4. Виконання завдання з відповідними скріншотами.
5. Відповіді на контрольні запитання.
6. Висновок

Суматор. Основні компоненти

Суматор – логічна схема, що виконує операцію побітового додавання з переповненням.

У паралельних n-розрядних суматорах значення всіх розрядів операндів поступають одночасно на відповідні входи одно розрядних підсумовуючих схем.

У послідовних суматорах значення розрядів операндів та перенесення, які запам'ятовувалися в минулому такті, поступають послідовно в напрямку від молодших розрядів до старших на входи одного одно розрядного суматора.

В паралельно-послідовних суматорах числа розбиваються на частини, наприклад, байти, розряди байтів поступають на входи восьми розрядного суматора паралельно (одночасно), а самі байти — послідовно, в напрямку від молодших до старших байтів з врахуванням запам'ятованого перенесення.

Суматор складається з окремих схем, які називаються однорозрядними суматорами; вони виконують усі дії з додавання значень однойменних розрядів двох чисел (операндів).

Отже, як виконується операція додавання?

Операція додавання є базовою у персональному комп'ютері. Продуктивність операції додавання і особливо обмеження, пов'язані з механізмом перенесення, впливають на загальну продуктивність комп'ютера.

Додавання чисел у бінарній формі реалізується за допомогою суматорів, що зазвичай виконують цілочисельне додавання в електронних цифрових обчислювальних машинах.

$$\begin{array}{r}
 7 + 2 = 9 \\
 \text{1 1} \\
 0111 \\
 + 0010 \\
 \hline
 1001
 \end{array}$$

Рис 3.1. Приклад додавання чисел у бінарній формі

Також було розглянуто роботу напівсуматора, для того щоб зрозуміти роботу повного суматора. Напівсуматор — це модуль, який має 2 входи (A та B) та 2 виходи (S та Cout). S — це сума A та B. Якщо A та B дорівнюють 1, то вихід S має стати рівним 2. Таке число не може бути представлене у вигляді одного двійкового розряду. У цьому випадку результат вказується разом з переносом Cout в наступний розряд.

На рисунку 3.2 зображено таблицю істинності для напівсуматора, та вентиляну схему самого напівсуматора.

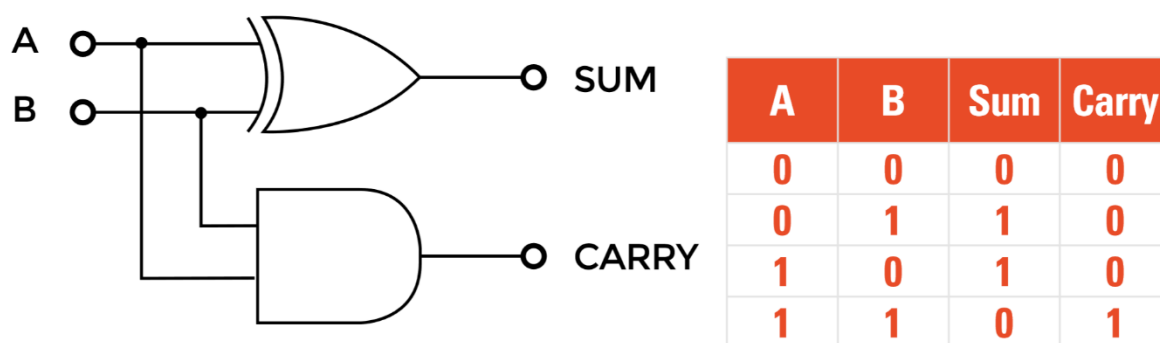


Рис.3.2. Таблиця істинності для напівсуматора

Повний суматор подібний до напівсуматора (все ж вони суматори), але так як суматор ще називають повним,, то він має додатковий вхід Cin. Тобто можна зробити висновок що тепер буде додаватися 3 однобітових числа і їх результат також буде записуватись у 2 біти Cout(Carry) та S(Sum).

Full Adder Truth Table

Inputs			Outputs	
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

Рис.3.3. Таблиця істинності повного суматора

Реалізація у Quartus Prime 1

Для початку ми створили пустий проект і налаштування встановили відповідно до вимог:

Summary	
When you click Finish, the project will be created with the following settings:	
Project directory:	C:\intelFPGA_lite\16.1
Project name:	lab1
Top-level design entity:	lab1
Number of files added:	0
Number of user libraries added:	0
Device assignments:	
Design template:	n/a
Family name:	Cyclone IV GX
Device:	EP4CGX15BF14A7
Board:	n/a
EDA tools:	
Design entry/synthesis:	<None> (<None>)
Simulation:	ModelSim-Altera (Verilog HDL)
Timing analysis:	()
Operating conditions:	
VCCINT voltage:	1.2V

Рис. 3.4. Налаштування проекту

Наступним кроком ми створили файл у якому будемо створювати вентиляну схему нашого суматора

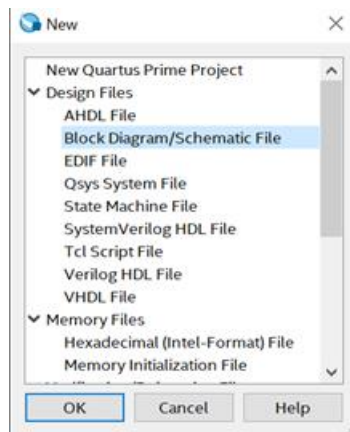


Рис. 3.5. Створення файлу

Отримали наступне вікно

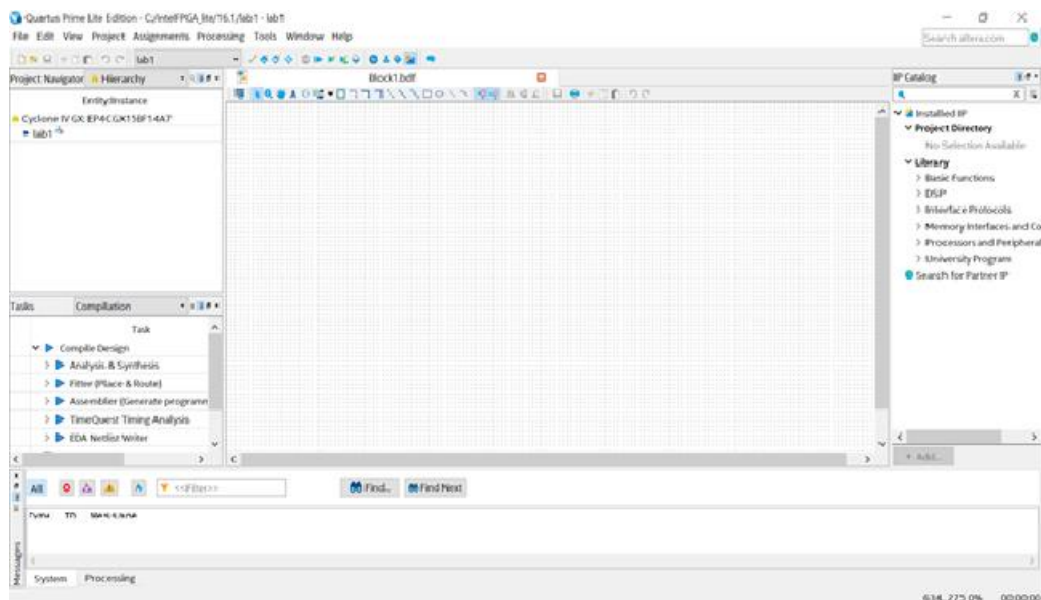


Рис.3.6. Вікно для побудови схеми суматора

Якщо сітка відсутня можна перейти до вкладки (tools > options) та вставити сітку. Двічі натиснувши по вікну, отримали змогу додати моделі, обрали primitives > logic > and2 та or2.

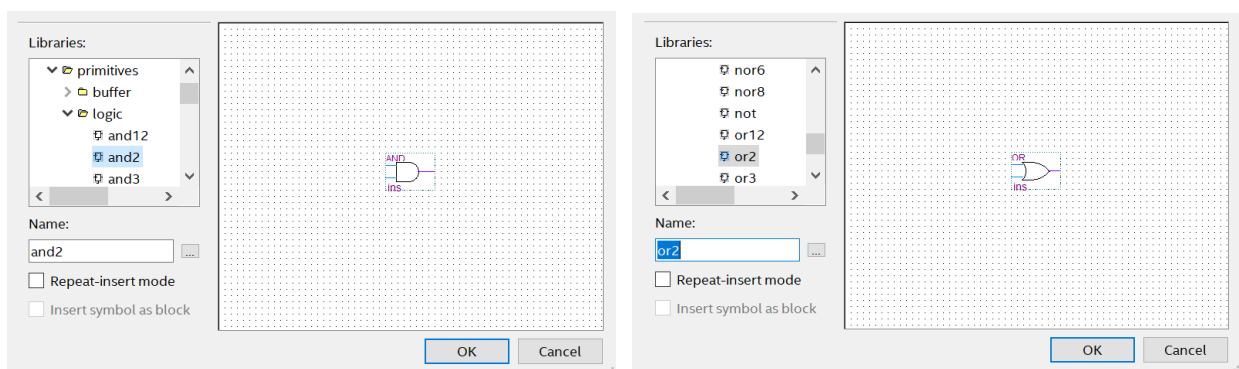


Рис.3.7. Додавання моделей у проект

Далі встановлюємо входи та виходи (3 та 2 відповідно). Отримали наступний набір моделей

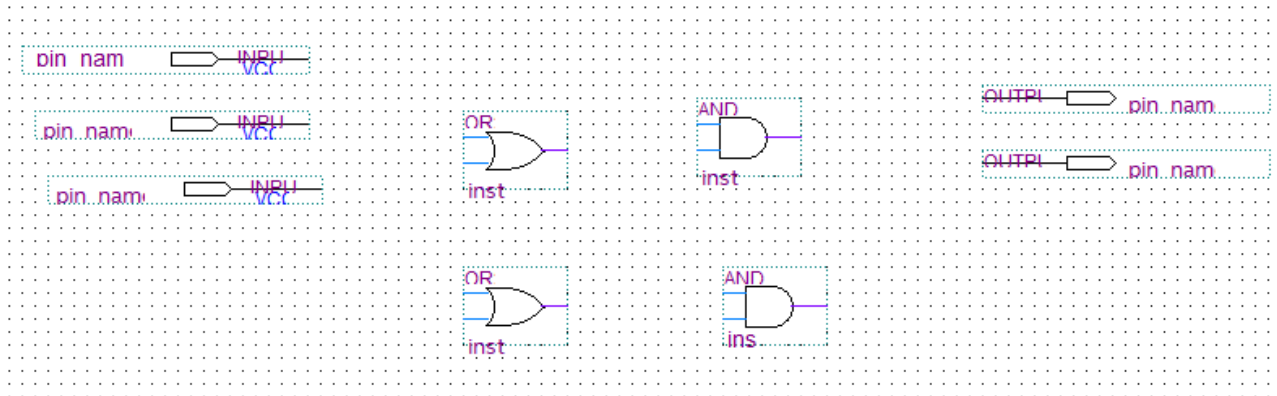


Рис.3.8. Необхідні для роботи моделі

Перейменували назви входів та виходів. Остаточний вигляд схеми буде таким:

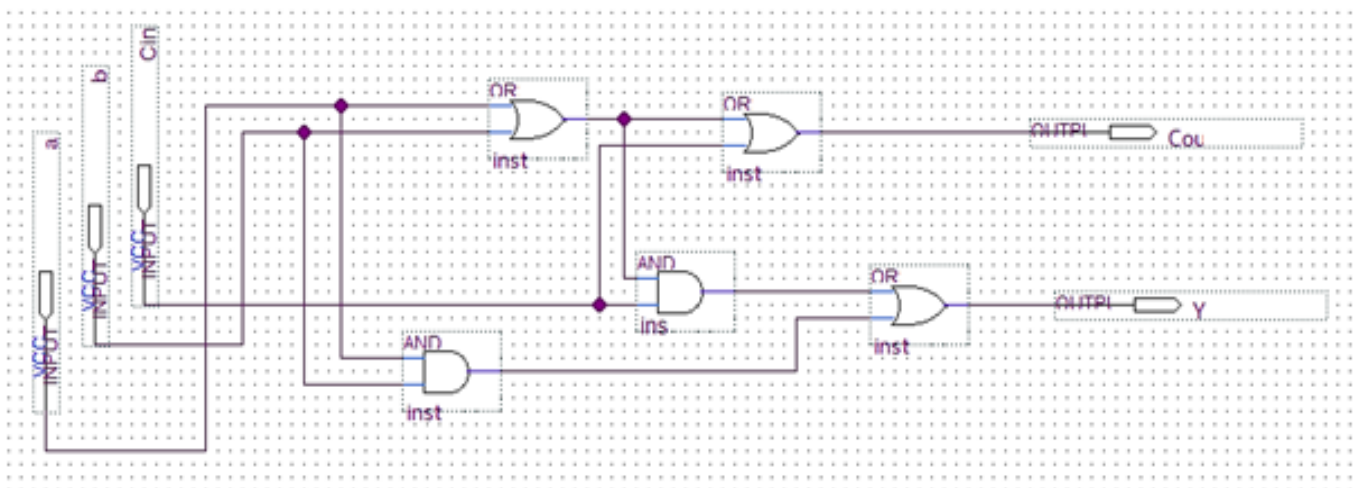


Рис.3.9. Схема суматора

Далі запустили компіляцію для перевірки роботи схеми, як можемо бачити на рисунку нижче, компіляція була успішною

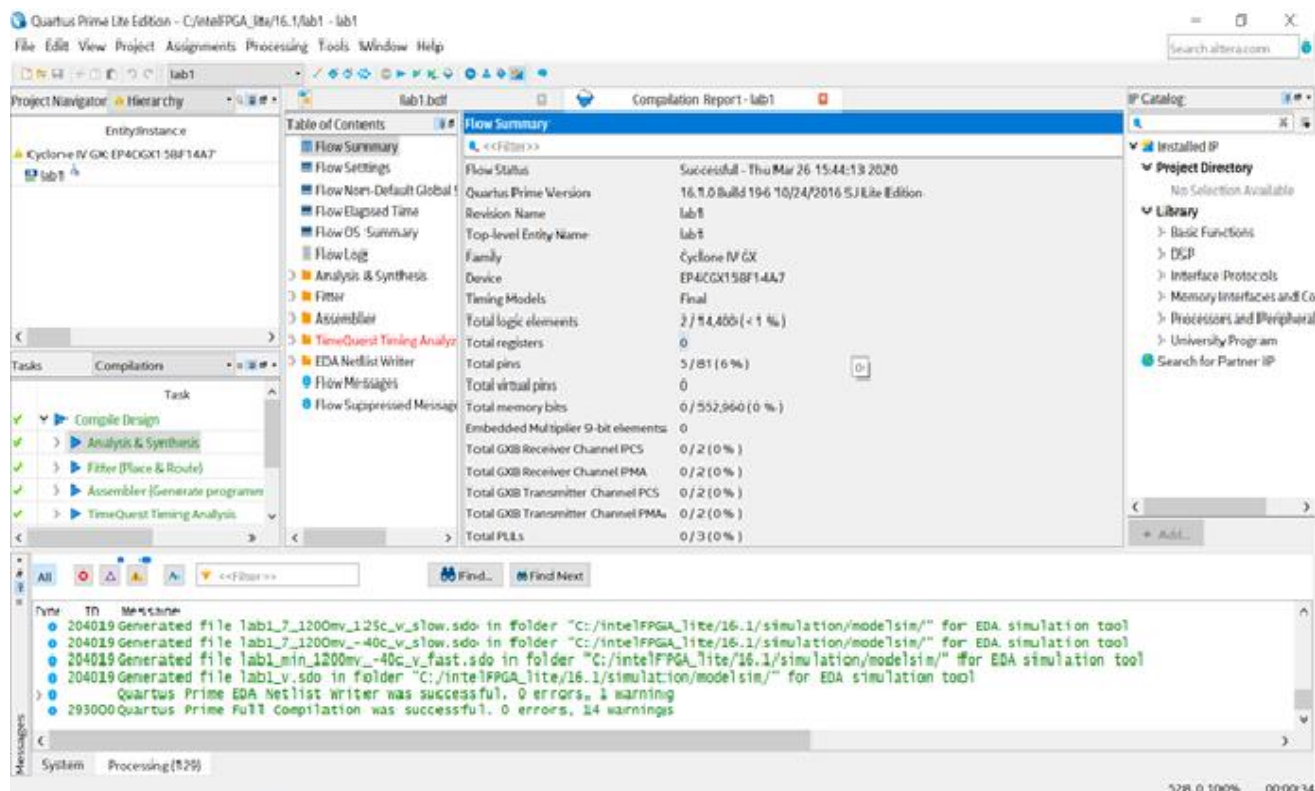


Рис.3.10. Компіляція схеми

Наступним кроком ми створили файл Verilog та написали відповідний код для коректної роботи суматора:

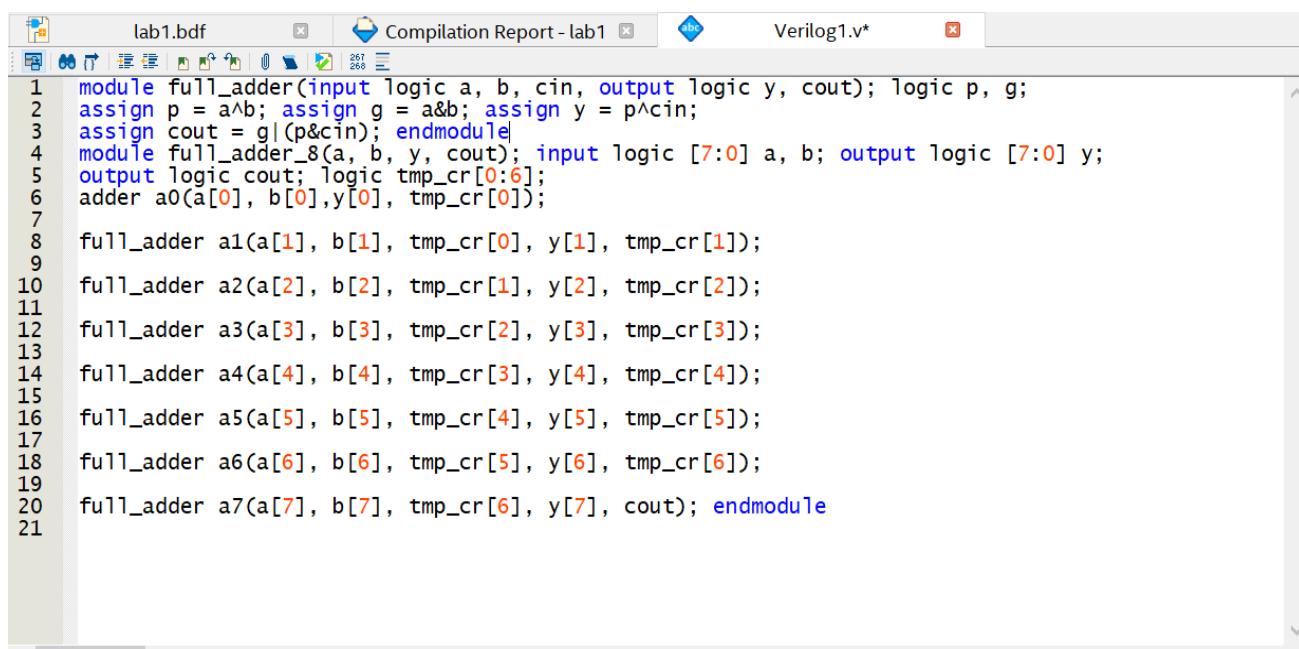


Рис.3.11. Вікно HDL Verilog

Для того щоб постійно не вставляти зображення, нижче наведемо повний лістинг коду:

Файл full_adder.sv

```
module full_adder(input logic a, b, cin, output logic y, cout);
logic p, g;
assign p = a^b; assign g = a&b; assign y = p^cin;
assign cout = g|(p&cin);
endmodule
```

Файл full_adder_8.sv

```
module full_adder_8(a, b, y, cout); input logic [7:0] a, b; output
logic [7:0] y;
output logic cout; logic tmp_cr[0:6];
adder a0(a[0], b[0], y[0], tmp_cr[0]);
full_adder a1(a[1], b[1], tmp_cr[0], y[1], tmp_cr[1]);
full_adder a2(a[2], b[2], tmp_cr[1], y[2], tmp_cr[2]);
full_adder a3(a[3], b[3], tmp_cr[2], y[3], tmp_cr[3]);
full_adder a4(a[4], b[4], tmp_cr[3], y[4], tmp_cr[4]);
full_adder a5(a[5], b[5], tmp_cr[4], y[5], tmp_cr[5]);
full_adder a6(a[6], b[6], tmp_cr[5], y[6], tmp_cr[6]);
full_adder a7(a[7], b[7], tmp_cr[6], y[7], cout);
endmodule
```

Файл adder.sv

```
module full_adder(input logic a, b, output logic y, cout); logic p,
g;
assign p = a^b; assign g = a&b; assign y = p; assign cout = g
endmodule
```

Цей пристрій слугує для виконання операцій додавання. Для реалізації восьмибітного суматора, потрібно побудувати однобітний повний суматор. Як бачимо, модуль починається з ключового слова `module` після нього йде ім'я модуля та список вхідних і вихідних сигналів. Оператор `assign` описує комбінаційну логіку. Сигнали типу `logic` — логічні змінні, що можуть приймати значення 0 або 1. Модуль закінчується ключовим словом `endmodule`. Функціональна схема складається з логічних елементів XOR, OR та AND.

Опис роботи програми:

Спочатку ми строїли модуль `full_adder`, який приймає логічні сигнали `a, b, cin` дає на виході логічні сигнали `S, cout`. Потім створюємо логічні змінні `p, g`. Спрощуємо розрахунки, і розраховуємо значення логічних змінних. $G=AB$

Перепишуємо `S, Cout` з урахуванням розрахунків. У другому файлі створюємо модуль `full_adder_8`, який приймає на вхід `a, b`, які передають восьмибітні логічні

сигнали. На виході модуль дає 8-бітне число і cout, що відповідає за перенесення. Так як при реалізації нам треба буде 7 разів переносити біт переповнення то створили шини `scr_cg[0:6]`. Після цього у модулі послідовно створили напівсуматор `adder` та 7 екземплярів модуля `full_adder` у кожен з яких передаються відповідні сигнали.

Отримали схему:

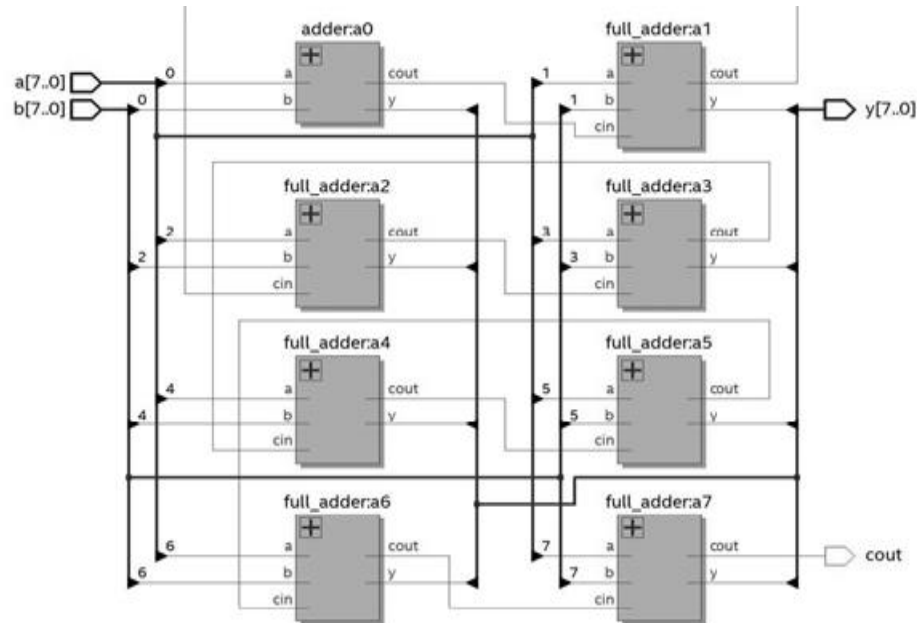


Рис.3.12. Схема з урахуванням усіх редакцій

На схемі ми можемо побачити 8 з'єднаних модулів. Можемо побачити, що сигнал `cout` відправляється на вхід `cin` наступного повного суматора.

Також натиснувши на `+` у правому верхньому кутку екземплярів, можемо побачити їх внутрішню схему.

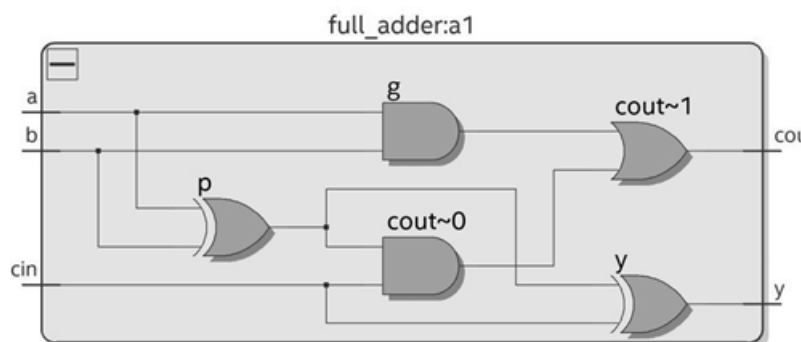


Рис.3.13. Внутрішні схема повного суматора

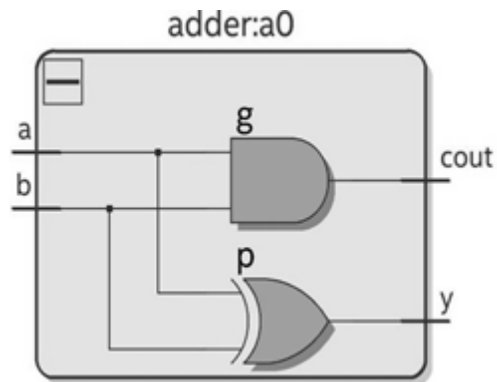


Рис.3.14. Внутрішня схема

Верифікацію проводимо за допомогою Quartus ModelSim Altera. Проведемо декілька тестів для перевірки правильності роботи схеми.

> a	B01101110	01101110	10100110	10101011	10011010
> b	B00010101	00010101	01100001	00011101	01011101
cout	B0				
> y	B10000011	10000011	00000111	11001000	11110111

Рис.3.15. Результати роботи схеми

$$\begin{aligned}
 01101110_2 (110_{10}) + 00010101_2 (21_{10}) &= 10000011_2 (131_{10}) \\
 10100110_2 (166_{10}) + 01100001_2 (97_{10}) &= 100000111_2 (263_{10}) \\
 10101011_2 (171_{10}) + 00011101_2 (29_{10}) &= 11001000_2 (200_{10}) \\
 10011010_2 (154_{10}) + 1011101_2 (93_{10}) &= 11110111_2 (247_{10})
 \end{aligned}$$

Рис.3.16. Перевірка на правильність вручну

Щоб перевірити правильність побудови функціональної схеми, побудуємо таблицю істинності. Виконаємо перевірку, виконавши самостійно операції, а також зробимо симуляцію для повного одно-бітного суматора

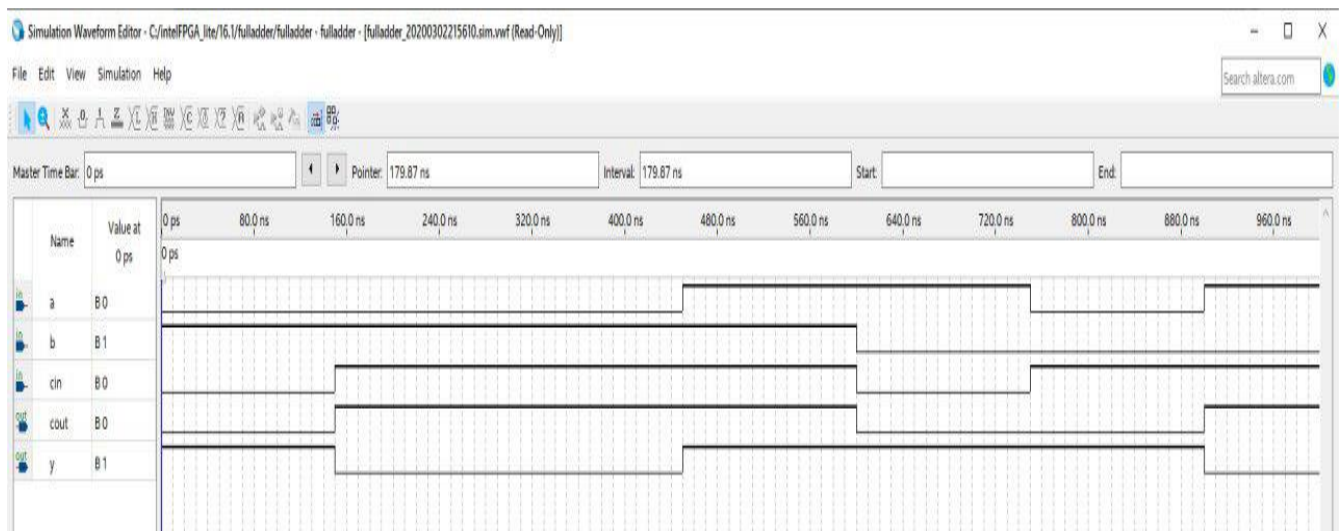


Рис. 3.17. Симуляція повного суматора у середовищі Quartus

a	b	cin	cout	Y
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

Рис. 3.18. Таблиця істинності повного суматора у середовищі Quartus

Результатом порівняння є таблиця і графічне зображення у середовищі програмування Quartus. Як ми бачимо, результати співпали.

Діаграма станів

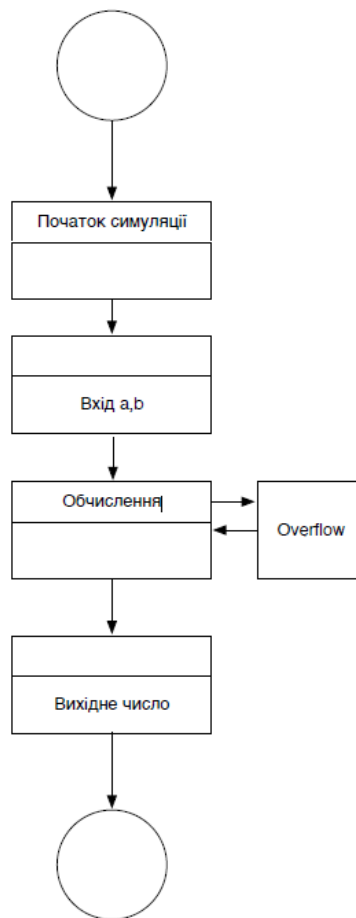


Рис.3.19. Діаграма станів

Реалізація у Quartus Prime 2

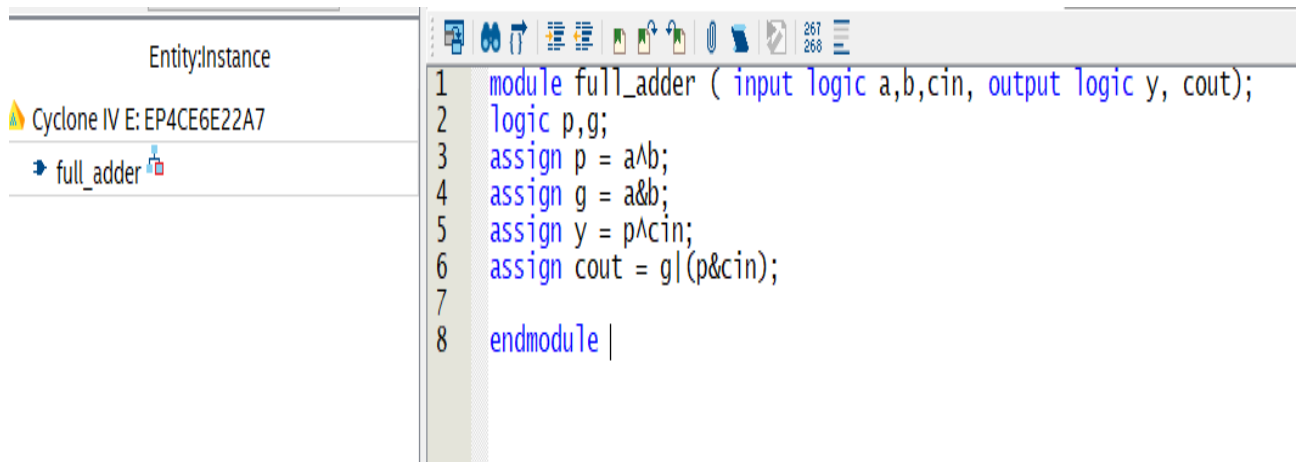
Мета роботи: розробити архітектурне рішення для виконання операції додавання. Створити восьми-бітний суматор.

Для вирішення поставленої задачі використовувалось середовище програмування «Quartus», а також апаратна частина Cyclone IV GL. Вхідними даними є 2 8-бітні шини a і b та біт cin , на виході модуля буде 8-бітний результат u , та біт $cout$. Для реалізації суматора необхідні сигнали, що будуть переносити біт переповнення, їх необхідно 7 штук. Для перевірки правдивості результатів виконано перевірку за допомогою симуляції в середовищі «Quartus». У побудові такого суматора допомагає мова програмування System Verilog HDL, яка є моделюванням функціональної схеми, яку вона описує.

Її синтаксис дещо схожий з C. Бітові оператори абсолютно ідентичні (крім них майже нічого не будемо використовувати). Основний принцип написання

програм — це опис модулів, які в свою чергу будуть слугувати частинами інших модулів і так далі до опису кінцевого бажаного модуля.

Щоб побудувати восьми-бітний суматор, скористаємося звичайними суматорами, які об'єднаємо в одне ціле.



The screenshot shows the Quartus II IDE. On the left, the 'Entity:Instance' pane shows 'Cyclone IV E: EP4CE6E22A7' and an instance of 'full_adder'. The main window displays the VHDL code for the 'full_adder' module:

```
1 module full_adder ( input logic a,b,cin, output logic y, cout);  
2 logic p,g;  
3 assign p = a^b;  
4 assign g = a&b;  
5 assign y = p^cin;  
6 assign cout = g|(p&cin);  
7  
8 endmodule |
```

Рис.3.20. Код програми full_adder

Цей код показує входні та вихідні сигнали модуля, а також описує комбінаційну логіку. Він включає внутрішні змінні P та G, і вони функціонують як вхід та вихід, які з'єднують різні частини модуля.

Можемо після компіляції переглянути цифрову RTL схему, яку ми синтезуємо. Код, призначений для побудови повного суматора, має таку функціональна схему:

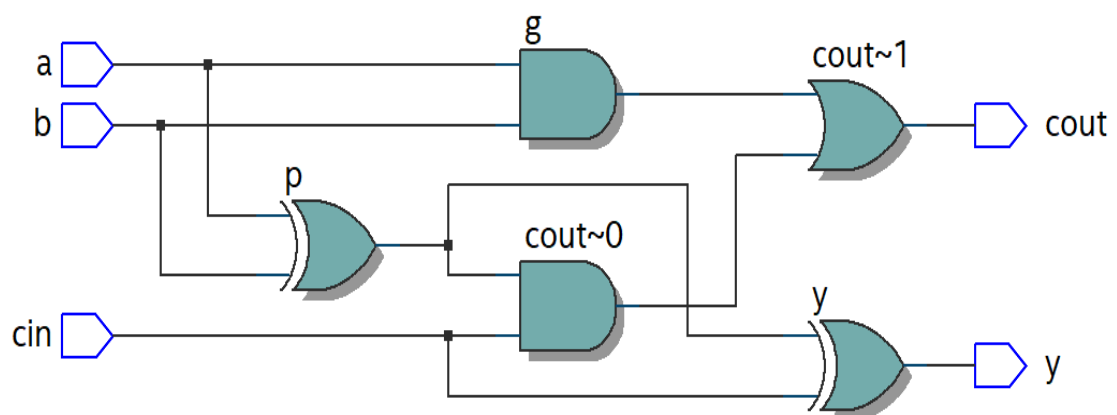


Рис.3.21. Функціональна схема повного суматора

```

1 module full_adder_8(a,b, cin, y, cout);
2   input logic [7:0] a,b;
3   input logic cin;
4   output logic [7:0] y;
5   output logic cout;
6   logic tmp_cr [0:6];
7
8   full_adder a0(a[0], b[0], cin, y[0], tmp_cr[0]);
9   full_adder a1(a[1], b[1], tmp_cr[0], y[1], tmp_cr[1]);
10  full_adder a2(a[2], b[2], tmp_cr[1], y[2], tmp_cr[2]);
11  full_adder a3(a[3], b[3], tmp_cr[2], y[3], tmp_cr[3]);
12  full_adder a4(a[4], b[4], tmp_cr[3], y[4], tmp_cr[4]);
13  full_adder a5(a[5], b[5], tmp_cr[4], y[5], tmp_cr[5]);
14  full_adder a6(a[6], b[6], tmp_cr[5], y[6], tmp_cr[6]);
15  full_adder a7(a[7], b[7], tmp_cr[6], y[7], cout);
16
17 endmodule
18

```

Рис.3.22. Код програми 8-бітного суматора

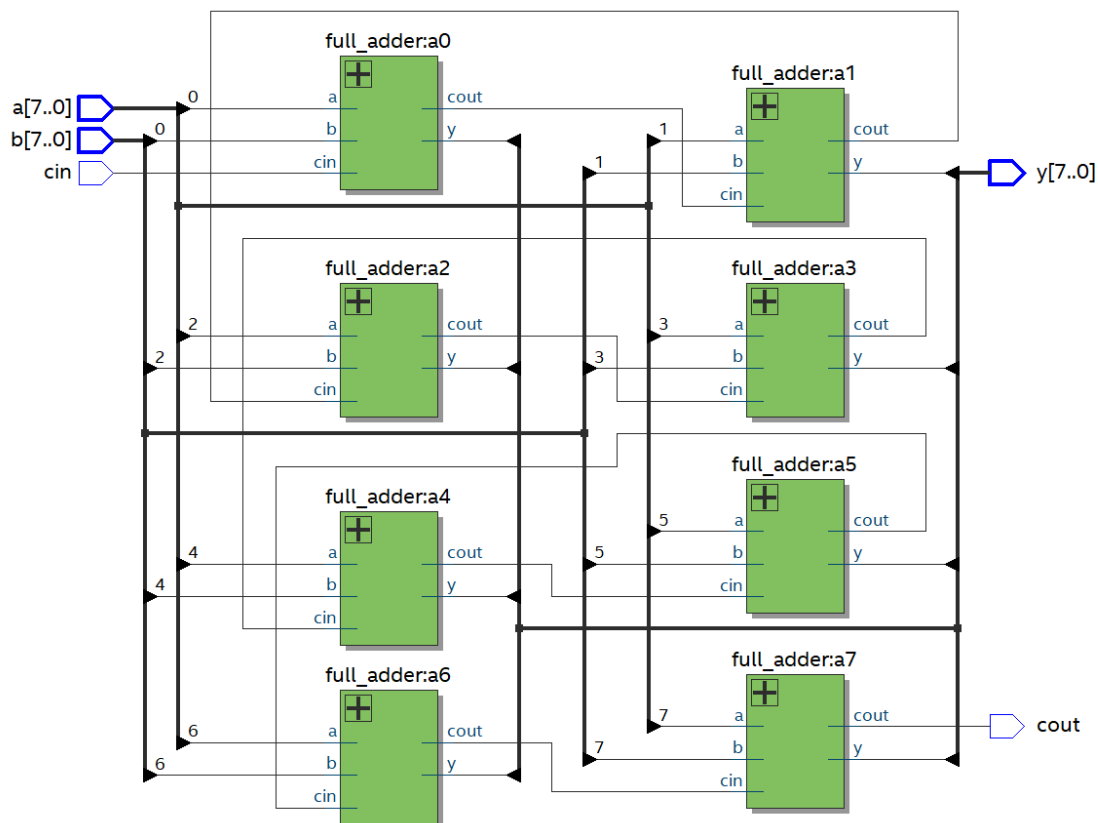


Рис.3.23. Функціональна схема 8-бітного суматора

Об'єднання у ланцюг N-повних суматорів є найпростішим способом реалізації N-розрядних суматорів. Вихід деякого розряду буде поступати на вхід наступного розряду. Після дослідження схеми реалізується 8-розрядний суматор, який написаний на мові SystemVerilog.

Для перевірки правильності побудови функціональної схеми, побудуємо таблицю істинності. Виконаємо перевірку, виконавши самостійно операції, а також зробимо симуляцію для повного одно-бітного суматора.

Результатом порівняння є таблиця 3.1 і графічне зображення у середовищі програмування:

Таблиця 3.1

a	b	cin	cout	y
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

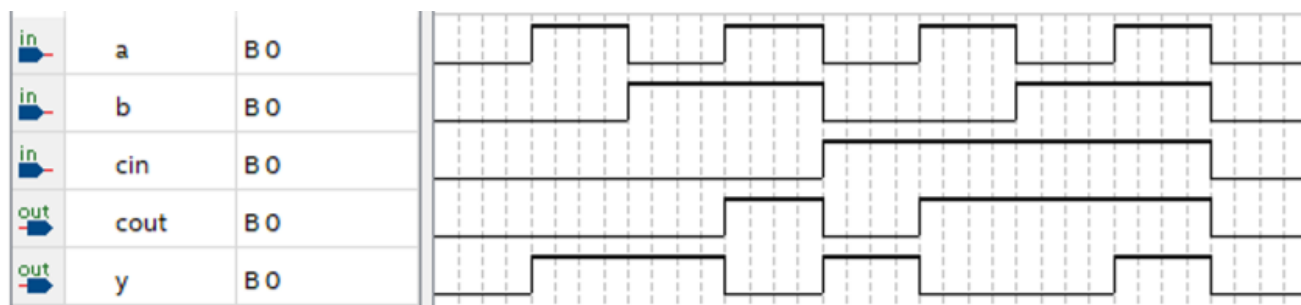


Рис.3.24. Симуляція повного суматора

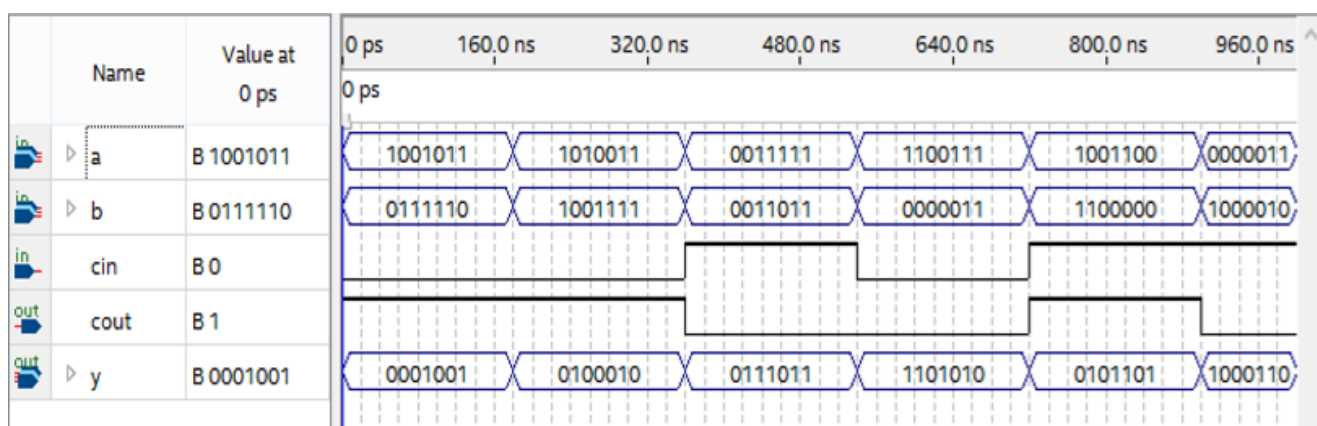


Рис.3.25. Тестування 8-бітного суматора

Висновки

У ході виконання лабораторної роботи, було створено 8-розрядний суматор, програмний код було написано використовуючи мову програмування HDL SystemVerilog в середовищі Quartus Prime 16.1, який також використовували для опису та моделювання мікросхем. Створена схема була перевірена на правильність за допомогою пакету верифікації Quartus ModelSim Altera.

Контрольні запитання

1. Поясніть і опишіть структуру коду повного суматора в даній роботі.
2. Поясніть і опишіть структуру коду 8-бітного суматора в даній роботі.
3. Що собою представляє схема повного суматора RTL-viewer?
4. Що собою представляє схема 8-бітного суматора RTL-viewer?
5. Поясніть логічні елементи які присутні на схемі повного суматора.
6. Поясніть роботу симуляції повного суматора.
7. Яким чином перевірити правильність результатів роботи 8-бітного суматора.

3.2. Лабораторна робота № 2

Тема: Операція віднімання

Постановка задачі:

Об'єкт дослідження: арифметико-логічний пристрій.

Мета роботи: спроектувати комбінаційні схеми для роботи алгоритмічно-логічного пристрою.

Методи дослідження та апаратура: формалізація АЛП за допомогою середовища проектування Quartus та мови опису апаратури SystemVerilog.

Арифметико-логічний пристрій (англ. Arithmetic Logic Unit, ALU) — блок процесора, що служить для виконання арифметичних та логічних перетворень над даними, що іменуються операндами. Цей пристрій є фундаментальною частиною будь-якого обчислювача, навіть найпростіші мікроконтролери мають його в складі свого ядра. Центральний процесор та відеопроцесор можуть мати кілька АЛП, що відрізняються своїм функціональним призначенням або типом оброблюваних даних.

Результати: реалізовано арифметико-логічний пристрій для виконання базових математичних операцій двох 8-бітних цілочисельних операндів.

Завданням даної лабораторної роботи є розроблення рішення для виконання операції віднімання, тобто побудова повного субтрактора.

Вхідними аргументами операції мають бути два числа (8-бітні, у бінарній системі).

Результатом має бути одне 8-бітне число та один біт помилки арифметики.

Розробка має бути виконана у вигляді вентильної схеми і програмного коду Verilog HDL.

Завдання на лабораторну роботу:

Розробити архітектурне рішення для виконання операції віднімання. Вхідними аргументами операції мають бути два восьми-бітні числа, що задані у бінарній системі числення, є цілими і не мають знаку. Результатом є одне восьми-бітне бінарне число та один біт помилки арифметики.

Розробка має бути виконана у вигляді вентильної схеми і програмного коду Verilog HDL. Вхідні аргументи повинні бути подані у вигляді логічних ліній $a[7:0]$, $b[7:0]$. Вихідне значення операції повинне бути подано у вигляді $y[7:0]$, $err[0]$.

Працездатність архітектури і розробки повинні бути показані у вигляді процесу роботи логічної вентильної схеми та у вигляді результатів її роботи, що мають відповідати відповідним булевим законам.

Опис результатів, а також методів, підходів і засобів розробки повинні бути подані у вигляді звіту. Оформлення і вмісту звіту мають відповідати ДСТУ3008-95.

Розробка може проводитися у таких середовищах: MatLab Simulink; Multisim (EWB); ISIS Proteus; Quartus II.

Склад звіту лабораторної роботи

В звіті повинно бути відображено:

1. Тема роботи.
2. Мета роботи.
3. Постановка задачі.
4. Виконання завдання з відповідними скріншотами.
5. Відповіді на контрольні запитання.
6. Висновок.

Субтрактор. Основні компоненти

Субтрактор – логічна схема, що призначена для виконання операції віднімання - операції, протилежної тій, яку виконує суматор.

Головною ціллю є збереження правил додавання для від'ємних та додатних чисел. Ця проблема була вирішена за допомогою введення поняття додаткового коду. Числа, записані з використанням додаткового коду та беззнакові числа однакові, тільки значення старшого біту N-бітного числа дорівнює -2^{N-1} замість 2^{N-1} . Такий запис гарантує збереження алгоритму додавання.

Оскільки віднімання, по суті, це додавання протилежного за знаком числа, то ми можемо реалізувати віднімання за допомогою вже створеного суматора та блоку, що змінює знак числа на протилежний. Для того щоб інвертувати число, нам необхідно змінити кожен біт числа на протилежний і до результату додати 1.

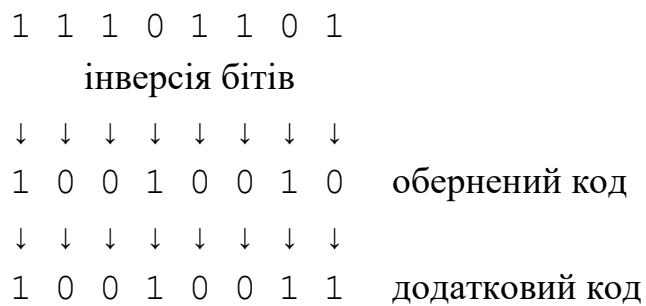


Рис.3.26. Приклад переведення від'ємного числа у обернений та додатковий код

Отже, для реалізації операції віднімання двійковим суматором, потрібно додати до зменшуваного протилежний за знаком з від'ємник. Тоді отримана сума буде різницею даних чисел: $x-y=x+(-y)$. Інверсія знаку записаного в двійковому вигляді числа відбувається таким же чином, як і в додатковому коді. Дане число потрібно інвертувати та додати до нього одиницю: $-y=(-y)+1$.

$$\begin{aligned} -19 &\rightarrow 01101 \\ \text{Бо} \\ 19_{10} &= 10011_2 \\ (-10011) + 1 &= 01100 + 1 = 01101 \end{aligned}$$

Рис.3.27. Приклад запису числа 19

Для того щоб зрозуміти загальну задачу треба зрозуміти меншу задачу, а отже спочатку розберемо принцип роботи напівсубтрактора. Таблиця істинності буде мати наступний вигляд:

Вхід		Вихід	
X	Y	D	Bout
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Напісубтрактор – це модуль, який має 2 входи (X,Y) та два виходи (D,Bout).
Вентильна схема такого напісубтрактора:

Half Subtractor

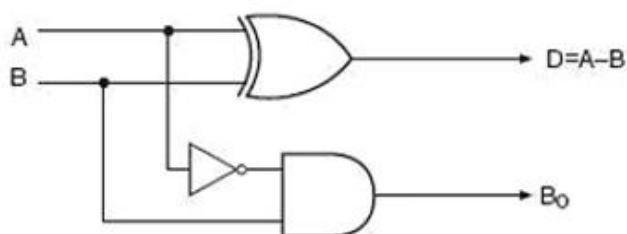


Рис.3.28. Вентильна схема напісубтрактора

Як можемо бачити , що

$$D = X \oplus Y$$

$$B_{out} = -X * Y$$

а з таких напісубтракторів можна побудувати повний субтрактор.

Повний субтрактор дуже схожий на напісубтрактора, але він має додатковий вхід на Bm (дуже подібно до лабораторної роботи з субтрактором). Таблиця істинності для повного субтрактора:

Вхід			Вихід	
X	Y	Bm	D	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1

0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

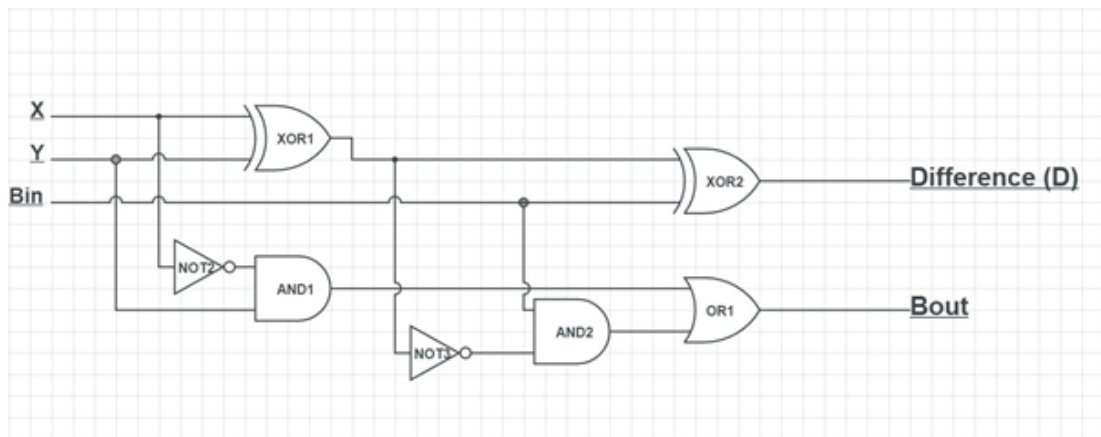


Рис.3.29. Повний субтрактор

Реалізація у Quartus Prime 1

Для початку ми створили пустий проект і налаштування встановили відповідно до вимог:

Summary

When you click Finish, the project will be created with the following settings:

Project directory:	C:\intelFPGA_lite\16.1
Project name:	lab2
Top-level design entity:	lab2
Number of files added:	0
Number of user libraries added:	0
Device assignments:	
Design template:	n/a
Family name:	Cyclone IV GX
Device:	EP4CGX15BF14A7
Board:	n/a
EDA tools:	
Design entry/synthesis:	<None> (<None>)
Simulation:	ModelSim-Altera (Verilog HDL)
Timing analysis:	()
Operating conditions:	
VCCINT voltage:	1.2V
Junction temperature range:	-40-125 °C

Рис.3.30. Налаштування проекту

Далі створюємо новий файл, додаємо моделі, усе як в попередній лабораторній, відрізняться лише будуть певні моделі та їх кількість.

Далі створюємо файл для коду та пишемо відповідно код:



Рис.3.31. Вікно для написання коду Verilog HDL

Лістинг коду:

**Файл verilog1(на рис.3.31), пізніше перейменували на
full_subtractor_8.sv**

```
module full_subtractor_8(a, b, sin, y, sout);
input logic [7:0] a, b;
input logic sin;
output logic [7:0] y;
output logic sout;
logic temp[0:6];
full_subtractor a1(a[0], b[0], sin, y[0], temp[0]);
full_subtractor a2(a[1], b[1], temp[0], y[1], temp[1]);
full_subtractor a3(a[2], b[2], temp[1], y[2], temp[2]);
full_subtractor a4(a[3], b[3], temp[2], y[3], temp[3]);
full_subtractor a5(a[4], b[4], temp[3], y[4], temp[4]);
full_subtractor a6(a[5], b[5], temp[4], y[5], temp[5]);
full_subtractor a7(a[6], b[6], temp[5], y[6], temp[6]);
full_subtractor a8(a[7], b[7], temp[6], y[7], sout);
endmodule
```

Файл full_subtractor.sv

```
module full_subtractor(a, b, sin, y, sout);  
input logic a, b, sin;  
output logic y, sout;  
logic temp[0:2];  
half_subtractor first(a, b, temp[0], temp[1]);  
half_subtractor second(temp[0], sin, y, temp[2]);  
assign sout = temp[1]|temp[2];  
endmodule
```

Файл half_subtractor.sv

```
module half_subtractor(a, b, y, s);  
input logic a, b;  
output logic y, s;  
assign y = a^b;  
assign s = ~a&b;  
endmodule
```

Спочатку створюємо напівсубтрактор (half_subtractor.sv), оскільки логіка повного субтрактора повністю базується на ньому. Далі створюємо повний субтрактор (full_subtractor.sv), який буде приймати на вхід три сигнали – біти А, В, біт sin («займу»). Далі створюємо файл full_subtractor_8 для того, щоб мати змогу виконувати операцію віднімання над 8-бітними числами. Це означає, що потрібно провести операцію віднімання для кожної пари бітів, враховуючи при цьому біти sin та sout. Отримали схему наведену на рис. 3.7.

Субтрактор призначений для арифметичного віднімання двох чисел. При відніманні двох багато розрядних двійкових чисел по модулю два (біти і перенос, який поступив з молодшого розряду) формується біт різниці та біт переносу в старший розряд або кінцевий біт. Суматори та субтрактори схожі один на одного, окрім того напівсубтрактори та повні субтрактори використовуються аналогічно напівсуматорам та повним суматорам.

Принцип віднімання багато розрядних двійкових чисел полягає у тому, що у кожному з розрядів виконуються однотипні дії: визначається цифра різниці шляхом додавання по модулю 2 цифр доданків, з яких один є інвертованим, і переносу, який надходить до наступного розряду. Ці дії реалізуються двійковим одно розрядним субтрактором. Даний випадок, коли дані з попереднього блоку надходять в наступний називається послідовним субтрактором.

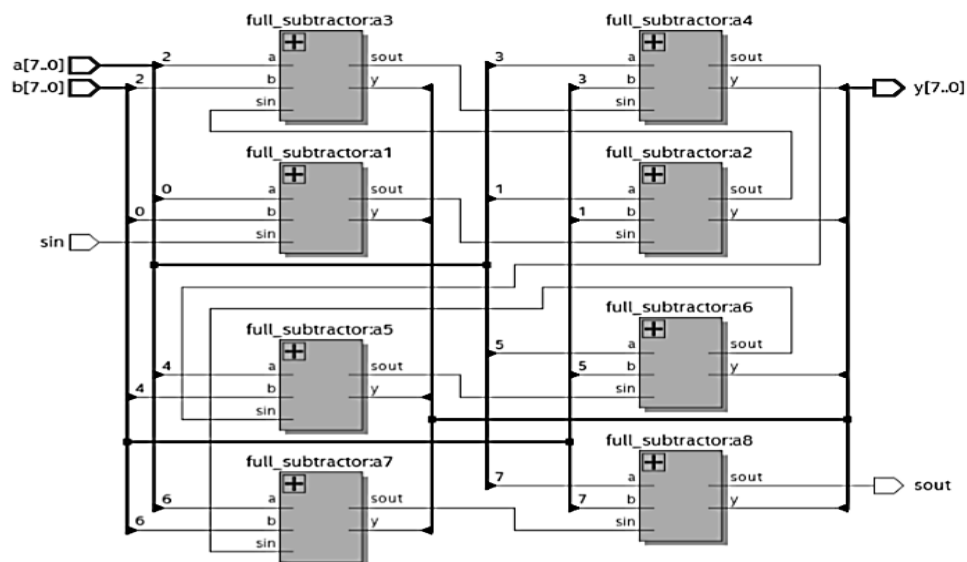


Рис.3.32. Отримана схема з використанням коду

Зі схеми можемо побачити 8 з'єднаних модуля. Сигнал Sout відправляється на вхід sin наступного повного субтрактора. Також якщо натиснути на + у верхньому кутку екземплярів, можна побачити їх внутрішню схему (як і в попередній лабораторній)

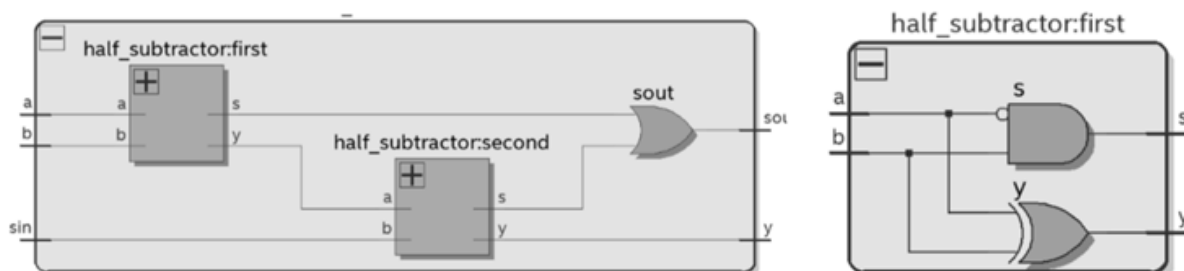


Рис.3.33. Внутрішні схеми модулів

Так як і впершій роботі, верифікацію проводимо за допомогою вбудованого пакету Quartus ModelSim Altera. Проведемо декілька тестів для перевірки схеми на коректність роботи.

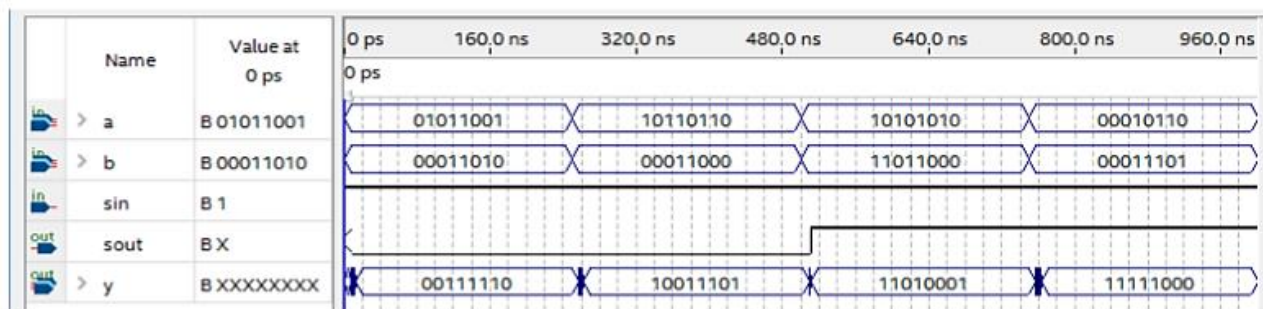


Рис.3.34. Результати перевірки

Перевірка на правильність «вручну».

$$\begin{aligned}01011001_2(89_{10}) - 00011010_2(26_{10}) &= 00011111_2(63_{10}) \\10110110_2(182_{10}) - 00011000_2(24_{10}) &= 10011101_2(158_{10}) \\10101010_2(170_{10}) - 11011000_2(216_{10}) &= 11010001_2(\text{об.код}) = 00101110_2(-46_{10}) \\00010110_2(22_{10}) - 00011101_2(29_{10}) &= 11111000_2(\text{об.код}) = 00000111_2(-7_{10})\end{aligned}$$

Рис.3.35. Перевірка на коректність

Реалізація у Quartus Prime 2

Мета роботи: розробити архітектурне рішення для виконання операції віднімання. Створити восьми-бітний субтрактор.

Для вирішення поставленої задачі використовувалось середовище програмування «Quartus», а також апаратна частина Cyclone IV GL. Вхідними даними є 2 8-бітні шини a і b та біт cin , на виході модуля буде 8-бітний результат y , та біт cout . Для реалізації субтрактора необхідні сигнали, що будуть переносити біт переповнення, їх необхідно 7 штук. Для перевірки правдивості результатів виконано перевірку за допомогою симуляції в середовищі «Quartus». У побудові такого субтрактора допомагає мова програмування System Verilog HDL, яка є моделюванням функціональної схеми, яку вона описує.

Її синтаксис дещо схожий з C. Бітові оператори абсолютно ідентичні (крім них нам майже нічого не будемо використовувати). Основний принцип написання програм — це опис модулів, які в свою чергу будуть слугувати частинами інших модулів і так далі до опису кінцевого бажаного модуля.

Щоб побудувати восьми-бітний субтрактор, скористаємося звичайними субтракторами, які об'єднаємо в одне ціле. Код для побудови функціональної схеми:

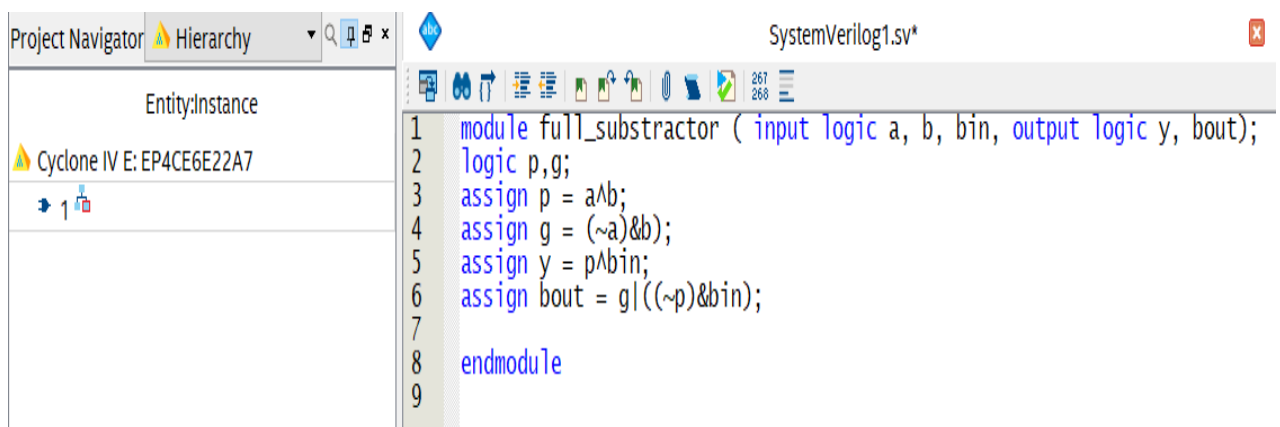


Рис.3.36. Код програми full_subtractor

Цей код показує вхідні та вихідні сигнали модуля, а також описує комбінаційну логіку. Воно включає внутрішні змінні P та G, і вони функціонують як вхід та вихід, які з'єднують різні частини модуля.

Код, призначений для побудови повного субтрактора, має таку функціональну схему:

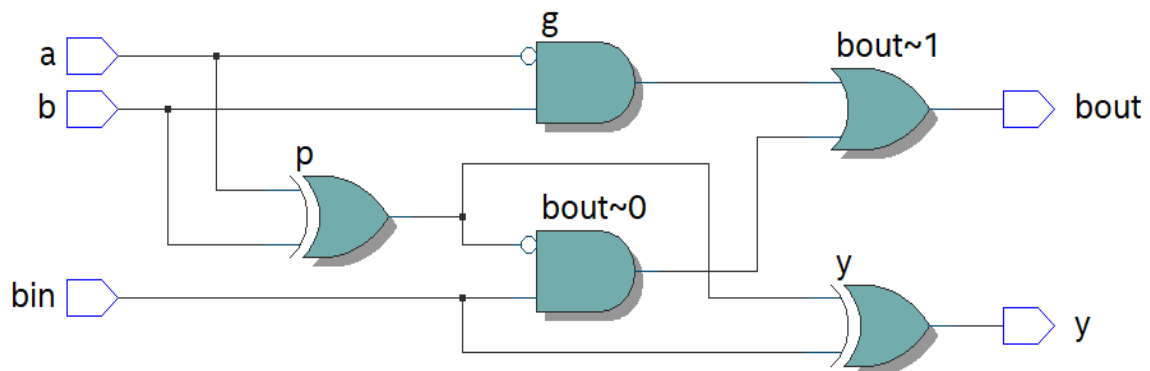


Рис.3.37. Функціональна схема повного субтрактора

```

1 module full_subtractor_8 (a,b, bin, y, bout);
2   input logic [7:0] a,b;
3   input logic bin;
4   output logic [7:0] y;
5   output logic bout;
6   logic tmp_cr[0:6];
7   full_subtractor a0(a[0], b[0], bin, y[0], tmp_cr[0]);
8   full_subtractor a1(a[1], b[1], tmp_cr[0], y[1], tmp_cr[1]);
9   full_subtractor a2(a[2], b[2], tmp_cr[1], y[2], tmp_cr[2]);
10  full_subtractor a3(a[3], b[3], tmp_cr[2], y[3], tmp_cr[3]);
11  full_subtractor a4(a[4], b[4], tmp_cr[3], y[4], tmp_cr[4]);
12  full_subtractor a5(a[5], b[5], tmp_cr[4], y[5], tmp_cr[5]);
13  full_subtractor a6(a[6], b[6], tmp_cr[5], y[6], tmp_cr[6]);
14  full_subtractor a7(a[7], b[7], tmp_cr[6], y[7], bout);
15
16 endmodule
17

```

Рис.3.38. Код програми 8-бітного субтрактора

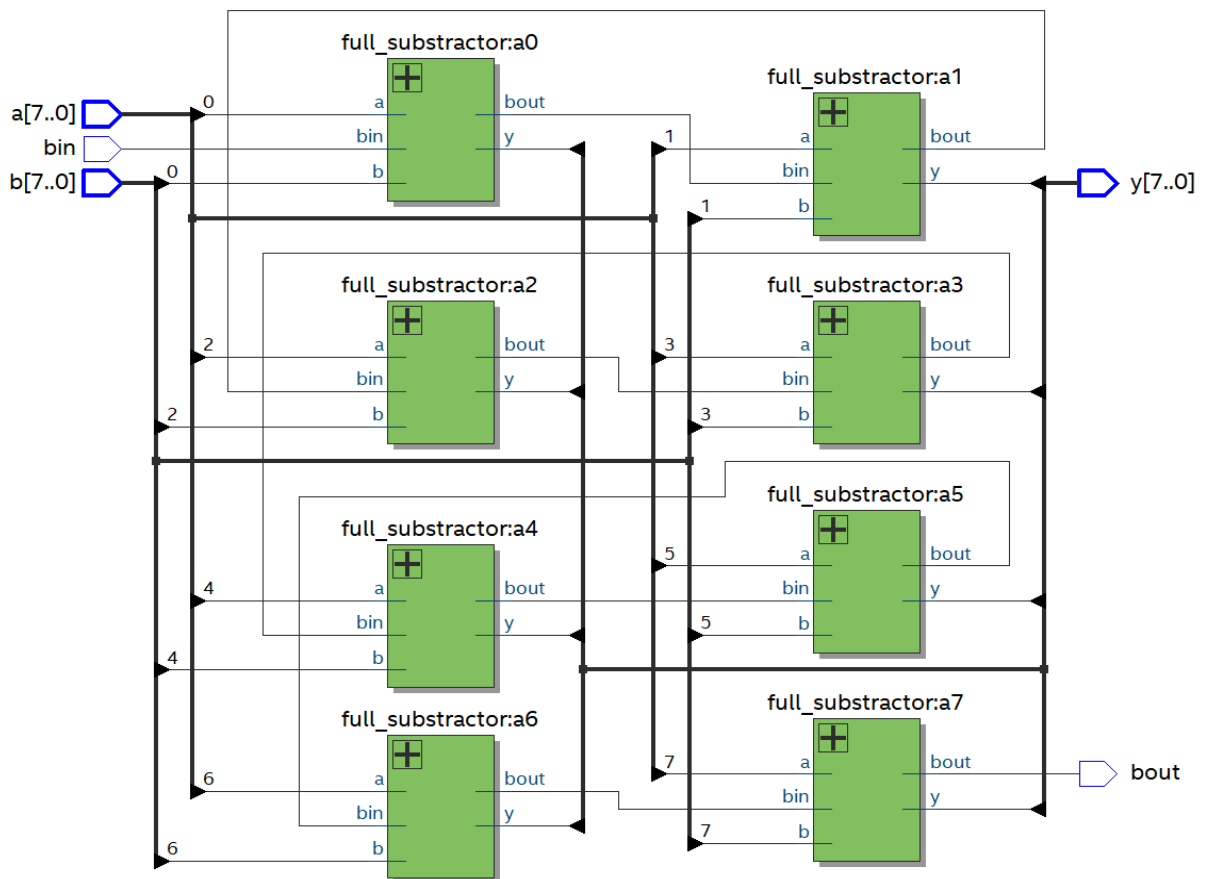


Рис.3.39. Функціональна схема субтрактора

Об'єднання у ланцюг N повних субтракторів є найпростішим способом реалізації N-розрядних субтракторів. Вихід деякого розряду буде поступати на вхід наступного розряду. Після дослідження схеми реалізується 8-розрядний субтрактор, який написаний мовою SystemVerilog.

Щоб перевірити правильність побудови функціональної схеми, побудуємо таблицю істинності. Виконаємо перевірку, виконавши самостійно операції, а також зробимо симуляцію для повного одно-бітного субтрактора. Результатом порівняння є таблиці 3.2 і графічне зображення у середовищі програмування:

Таблиця 3.2

a	b	cin	cout	y
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

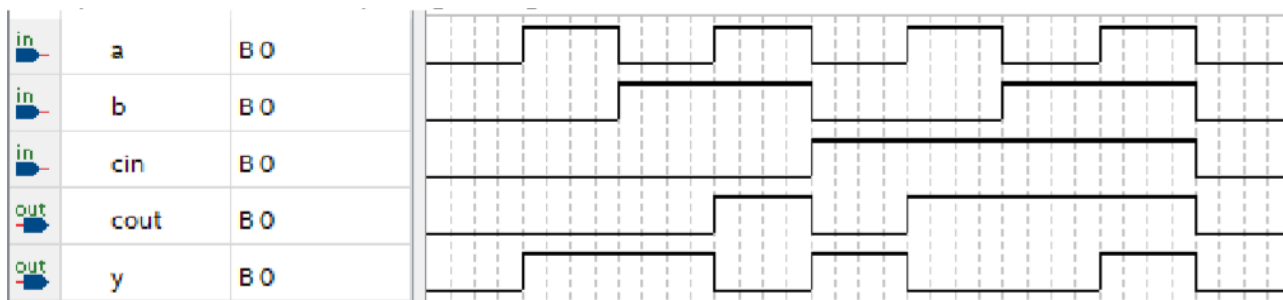


Рис.3.40. Симуляція повного субтрактора

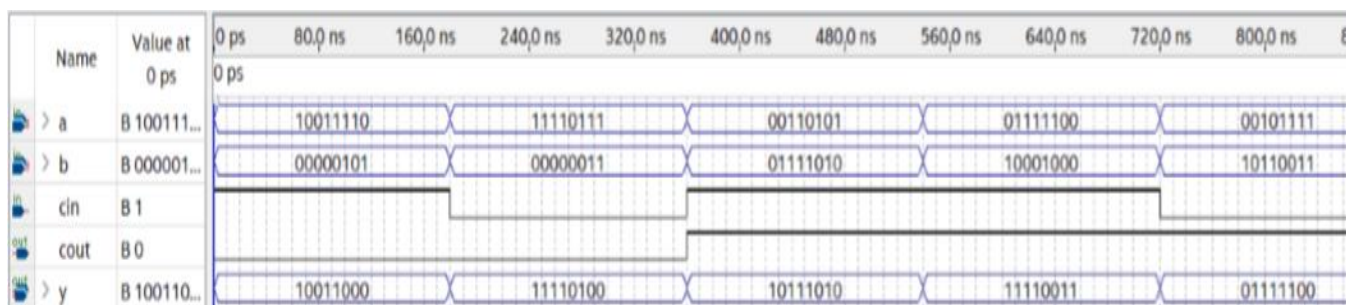


Рис.3.41. Симуляція 8-бітного субтрактора

Висновки

У ході виконання даної лабораторної роботи було побудовано субтрактор, який працює коректно. Код був написаний в HDL System Verilog в середовищі Quartus Prime. Створена система була перевірена за допомогою пакету верифікації Quartus ModelSim Altera.

Контрольні запитання

1. Поясніть і опишіть структуру коду повного субтрактора в даній роботі.
2. Поясніть і опишіть структуру коду 8-бітного субтрактора в даній роботі.
3. Що собою представляє схема повного субтрактора RTL-viewer?
4. Що собою представляє схема 8-бітного субтрактора RTL-viewer?
5. Поясніть логічні елементи які присутні на схемі повного субтрактора.
6. Поясніть роботу симуляції повного субтрактор.
7. Яким чином перевірити правильність результатів роботи 8-ми бітного субтрактора.

3.3. Лабораторна робота № 3

Тема: Операція множення

Постановка задачі:

Об'єкт дослідження: арифметико-логічний пристрій.

Мета роботи: спроектувати комбінаційні схеми для роботи алгоритмічно-логічного пристрою. Методи дослідження та апаратура: формалізація АЛП за допомогою середовища проектування Quartus та мови опису апаратури SystemVerilog.

Арифметико-логічний пристрій (англ. Arithmetic Logic Unit, ALU) — блок процесора, що служить для виконання арифметичних та логічних перетворень над даними, що іменуються операндами. Цей пристрій є фундаментальною частиною будь-якого обчислювача, навіть найпростіші мікроконтролери мають його в складі свого ядра. Центральний процесор та відеопроцесор можуть мати кілька АЛП, що відрізняються своїм функціональним призначенням або типом оброблюваних даних.

Результати: реалізовано арифметико-логічний пристрій для виконання базових математичних операцій двох 8-бітних цілочисельних операндів.

Вхідними даними мають бути два 8-бітні числа, що задані у бінарній системі числення є цілими.

Результатом є одне 16-бітне бінарне число та один біт переповнення.

Розробка має бути виконана у вигляді вентильної схеми і програмного коду Verilog HDL.

Завдання на лабораторну роботу:

Розробити архітектурне рішення для виконання операції множення. Вхідними аргументами операції мають бути два восьми-бітні числа, що задані у бінарній системі числення, є цілими і не мають знаку. Результатом є одне шістнадцяти-бітне бінарне число та один біт переповнення.

Розробка має бути виконана у вигляді вентильної схеми і програмного коду Verilog HDL. Вхідні аргументи повинні бути подані у вигляді логічних ліній $a[7:0]$, $b[7:0]$. Вихідне значення операції повинне бути подано у вигляді $yH[7:0]$, $yL[7:0]$, $of[0]$.

Працездатність архітектури і розробки повинні бути показані у вигляді процесу роботи логічної вентильної схеми та у вигляді результатів її роботи, що мають відповідати відповідним булевим законам.

Опис результатів, а також методів, підходів і засобів розробки повинні бути подані у вигляді звіту. Оформлення і вмісту звіту мають відповідати ДСТУ3008-95.

Розробка може проводитися у таких середовищах: MatLab SImulink; Multisim (EWB); ISIS Proteus; Quartus II.

Склад звіту лабораторної роботи

В звіті повинно бути відображено:

1. Тема роботи.
2. Мета роботи.
3. Постановка задачі.
4. Виконання завдання з відповідними скріншотами.
5. Відповіді на контрольні запитання.
6. Висновок.

Мультиплексор. Основні компоненти

Мультиплексор – логічна схема, що виконує операцію множення.

Буде розглянуто мультиплексор, що приймає на вхід два беззнакових 8-бітних числа, а вихідне значення операції множення буде подано у вигляді добутку та біту переповнення.

Як відбувається операція множення?

Відомо, що числа, які перемножуються, називаються множниками, а результат добутком. Множення двійкових чисел дуже схоже на десяткове множення, але тут фігурують лише одиниці та нулі.

$$\begin{array}{r}
 0 \quad 0 \quad 1 \quad 1 \\
 \times 0 \quad \times 1 \quad \times 0 \quad \times 1 \\
 \hline
 0 \quad 0 \quad 0 \quad 1
 \end{array}$$

Рис.3.42. Правила множення двійкових чисел

Двійкові числа

$$\begin{array}{r}
 111 \\
 \times 101 \\
 \hline
 111 \\
 000 \\
 111 \\
 \hline
 10001
 \end{array}$$

Десяткові числа

$$\begin{array}{r}
 7 \\
 \times 5 \\
 \hline
 35
 \end{array}$$

Рис.3.43. Приклад операції множення двійкових та десяткових чисел

Ми можемо множити числа шляхом багатократного додавання, але такий тип помножувачів не знайшов широкого поширення, адже іноді з використанням великих чисел цей процес може тривати досить довго.

Більш поширений спосіб множення в цифрових електронних пристроях – спосіб додавання зі здвигом.(відомий як спосіб здвигу та додавання).

$$\begin{array}{r}
 111 \\
 \times 101 \\
 \hline
 111 \\
 000 \\
 0111 \\
 111 \\
 \hline
 100011
 \end{array}$$

Рис.3.44. Двійкове множення зі здвигом

В даному прикладі двійкове число 111 множиться на 101 (7*5). Множення відбувається стандартним методом: вводиться лише додатковий рядок (рядок проміжного добутку). Це зроблено для полегшення розуміння процесу множення в цифрових пристроях. При аналізі даного способу множення можна виділити 3 важливих особливості:

- 1– Частковий добуток завжди дорівнює 000, якщо множник дорівнює 0, і дорівнює першому множнику, якщо другий множник – 1;
- 2– Кількість розрядів в регістрі добутку повинна бути в 2 рази більша за кількість розрядів в регістрах множників;
- 3 – При додаванні перший частковий добуток здвигається на один розряд вправо по відношенню до другого часткового добутку.

Реалізація у Quartus Prime 1

Відповідно до попередніх лабораторних робіт, створюємо новий проект, з урахуванням усіх особливостей проекту. Створюємо файл для побудови схем, та вставляємо відповідні моделі, далі створюємо файл з HDL Verilog для написання коду.

Лістинг коду для мультиплексора:

Файл multiplexor.sv

```
module multiplexor(a,b,y,cout);
input logic a[7:0];
input logic b[7:0];
output logic y[15:0];
output logic cout;
logic a2[15:0];
logic [15:0] y1,y2,y3,y4,y5,y6,y7,y8;
    конвертуємо 8-бітне число в 16-бітне
convert(a,a2);
    множимо окремо кожен біт b на 16-бітне
mult(a2,b[0],y1);
mult(a2,b[1],y2);
mult(a2,b[2],y3);
mult(a2,b[3],y4);
mult(a2,b[4],y5);
mult(a2,b[5],y6);
mult(a2,b[6],y7);
mult(a2,b[7],y8);
    здвигаємо кожен проміжний добуток на потрібну кількість розрядів
logic c2[15:0];
sdv(y2,c2);
logic [15:0] c31,c32;
sdv(y3,c31);
sdv(c31,c32);
logic [15:0] c41,c42,c43;
sdv(y4,c41);
sdv(c41,c42);
sdv(c42,c43);
logic [15:0] c51,c52,c53,c54;
sdv(y5,c51);
sdv(c51,c52);
sdv(c52,c53);
sdv(c53,c54);
logic [15:0] c61,c62,c63,c64,c65;
sdv(y6,c61);
sdv(c61,c62);
sdv(c62,c63);
```

```

sdv(c63,c64);
sdv(c64,c65);
logic [15:0] c71,c72,c73,c74,c75,c76;
sdv(y7,c71);
sdv(c71,c72);
sdv(c72,c73);
sdv(c73,c74);
sdv(c74,c75);
sdv(c75,c76);
logic [15:0] c81,c82,c83,c84,c85,c86,c87;
sdv(y8,c81);
sdv(c81,c82);
sdv(c82,c83);
sdv(c83,c84);
sdv(c84,c85);
sdv(c85,c86);
sdv(c86,c87);
logic [15:0] re1,re2,re3,re4,re5,re6;
logic temp[7:0];

```

додаємо послідовно проміжні добутки:

```

full_adder_16 r0(y1,c2,0,re1,temp[0]);
full_adder_16 r1(c32,re1,temp[0],re2,temp[1]);
full_adder_16 r2(c43,re2,temp[1],re3,temp[2]);
full_adder_16 r3(c54,re3,temp[2],re4,temp[3]);
full_adder_16 r4(c65,re4,temp[3],re5,temp[4]);
full_adder_16 r5(c76,re5,temp[4],re6,temp[5]);
full_adder_16 r6(c87,re6,temp[5],y,cout);
endmodule

```

Файл convert.sv

```

module convert(a,y);
input logic a[7:0];
output logic y[15:0];
assign y[0] = a[0];
assign y[1] = a[1];
assign y[2] = a[2];
assign y[3] = a[3];
assign y[4] = a[4];

```

```

assign y[5] = a[5];
assign y[6] = a[6];
assign y[7] = a[7];
assign y[8] = 1'b0;
assign y[9] = 1'b0;
assign y[10] = 1'b0;
assign y[11] = 1'b0;
assign y[12] = 1'b0;
assign y[13] = 1'b0;
assign y[14] = 1'b0;
assign y[15] = 1'b0;
endmodule

```

Файл mult.sv

```

input logic a[15:0];
input logic b;
output logic y[15:0];
assign y[15] = 1'b0;
assign y[14] = 1'b0;
assign y[13] = 1'b0;
assign y[12] = 1'b0;
assign y[11] = 1'b0;
assign y[10] = 1'b0;
assign y[9] = 1'b0;
assign y[8] = 1'b0;
assign y[7] = a[7]&b;
assign y[6] = a[6]&b;
assign y[5] = a[5]&b;
assign y[4] = a[4]&b;
assign y[3] = a[3]&b;
assign y[2] = a[2]&b;
assign y[1] = a[1]&b;
assign y[0] = a[0]&b;
endmodule

```

Файл sdv.sv

```
input logic a[15:0];
output logic b[15:0];
assign b[0]=1'b0;
assign b[1]=a[0];
assign b[2]=a[1];
assign b[3]=a[2];
assign b[4]=a[3];
assign b[5]=a[4];
assign b[6]=a[5];
assign b[7]=a[6];
assign b[8]=a[7];
assign b[9]=a[8];
assign b[10]=a[9];
assign b[11]=a[10];
assign b[12]=a[11];
assign b[13]=a[12];
assign b[14]=a[13];
assign b[15]=a[14];
endmodule
```

Файл full_adder.sv

```
module full_adder(input logic a, b, cin, output logic y,
cout);
logic p, g;
assign p = a^b;
assign g = a&b;
assign y = p^cin;
assign cout = g|(p&cin);
endmodule
```

Файл full_adder_16.sv

```
module full_adder_16(a, b, cin , y,cout);
input logic a[15:0];
input logic b[15:0];
input logic cin;
output logic y[15:0];
```

```

output logic cout;
logic tmp_cr[0:14];
full_adder(a[0], b[0], cin, y[0], tmp_cr[0]);
full_adder(a[1], b[1], tmp_cr[0], y[1], tmp_cr[1]);
full_adder(a[2], b[2], tmp_cr[1], y[2], tmp_cr[2]);
full_adder(a[3], b[3], tmp_cr[2], y[3], tmp_cr[3]);
full_adder(a[4], b[4], tmp_cr[3], y[4], tmp_cr[4]);
full_adder(a[5], b[5], tmp_cr[4], y[5], tmp_cr[5]);
full_adder(a[6], b[6], tmp_cr[5], y[6], tmp_cr[6]);
full_adder(a[7], b[7], tmp_cr[6], y[7], tmp_cr[7]);
full_adder(a[8], b[8], tmp_cr[7], y[8], tmp_cr[8]);
full_adder(a[9], b[9], tmp_cr[8], y[9], tmp_cr[9]);
full_adder(a[10], b[10], tmp_cr[9], y[10], tmp_cr[10]);
full_adder(a[11], b[11], tmp_cr[10], y[11], tmp_cr[11]);
full_adder(a[12], b[12], tmp_cr[11], y[12], tmp_cr[12]);
full_adder(a[13], b[13], tmp_cr[12], y[13], tmp_cr[13]);
full_adder(a[14], b[14], tmp_cr[13], y[14], tmp_cr[14]);
full_adder(a[15], b[15], tmp_cr[14], y[15], cout);
endmodule

```

Multiplier – апаратна частина, що представляє собою схему, яка виконує операцію множення над двома восьми-бітними числами без знаку. Множення беззнакових двійкових чисел подібне десятковому множенню, проте воно оперує тільки з одиницями та нулями. В обох випадках частинні додатки формуються шляхом множення окремих розрядів множника на все множене. Зсунуті частинні додатки складаються, і ми отримуємо результат.

В загальному випадку, помножувач $N \times N$ перемножує два N -розрядних числа та породжує $2N$ -розрядний результат. Частинні додатки при множенні дорівнюють або множнику, або нулю. Множення одного розряду двійкових чисел рівносильне операції “І”, тому для формування частинних додатків використовують логічні елементи “І”.

Вхідними даними є два 8-бітних числа, що задані у бінарній системі числення, є цілими і не мають знаку. Щоб реалізувати мультиплікатор необхідні, створені раніше модулі однобітного та восьмибітного суматора. Результатом буде одне 16-бітне число та біт переповнення.

Скомпілювавши програму, отримали логічну схему (рис 3.45). На схемі видно з’єднані між собою модулі. Також натиснувши +, побачили внутрішню схему.

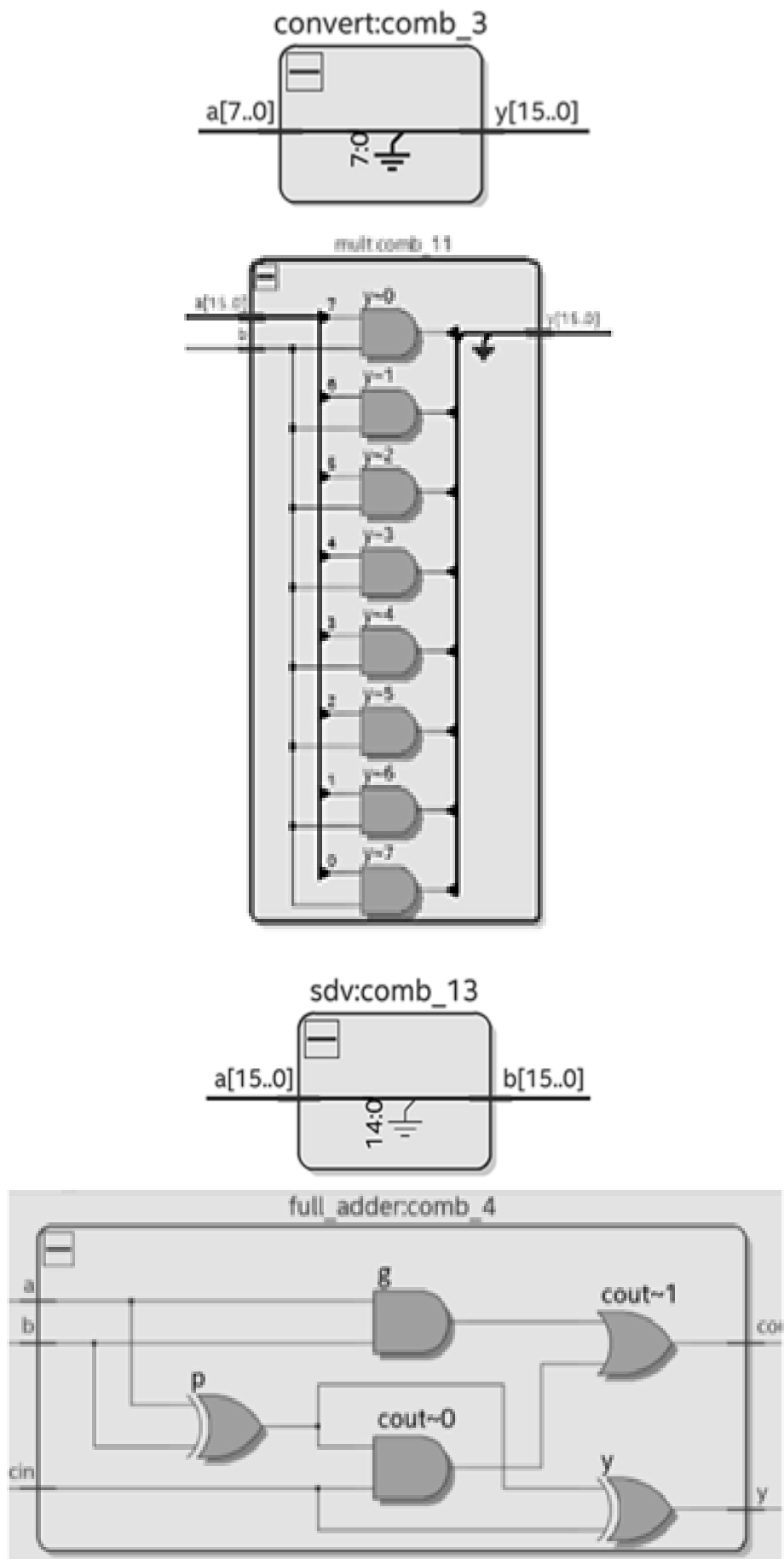


Рис.3.46. Внутрішні схеми модулів

Проведемо за допомогою вбудованого пакету Quartus ModelSim Altera декілька тестів для перевірки правильності роботи схеми.

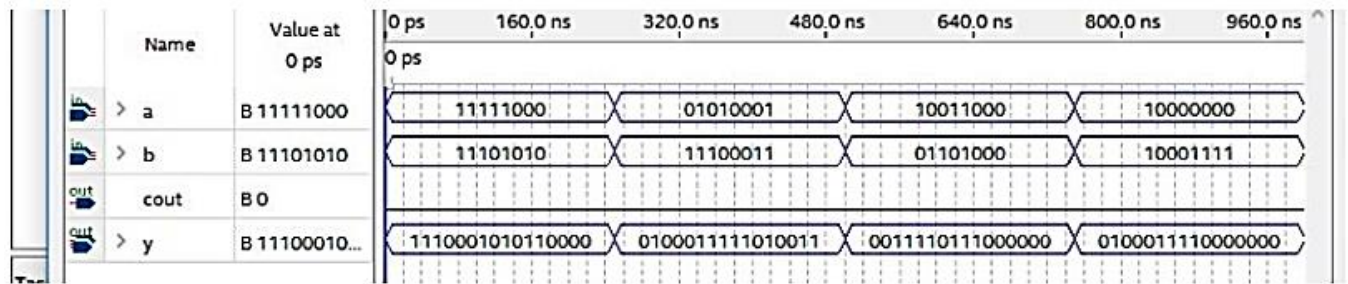


Рис.3.47. Результати роботи

Перевіримо «вручну» правильність роботи :

$$\begin{aligned}
 11111000_2 (248_{10}) * 11101010_2 (234_{10}) &= 1110001010110000_2 (5832_{10}) \\
 1010001_2 (81_{10}) * 11100011_2 (227_{10}) &= 10001111010011_2 (18387_{10}) \\
 10011000_2 (152_{10}) * 1101000_2 (104_{10}) &= 11110111000000_2 (15808_{10}) \\
 10000000_2 (128_{10}) * 1000111_2 (143_{10}) &= 100011110000000_2 (18304_{10})
 \end{aligned}$$

Рис. 3.48. Перевірка на правильність

Реалізація у Quartus Prime 2

Мета роботи: розробити архітектурне рішення для виконання операції множення. Створити восьми-бітний мультиплікатор.

Для вирішення поставленої задачі використовувалось середовище програмування «Quartus», а також апаратна частина Cyclone IV. Вхідними даними є 2 8-бітні шини a і b, на виході модуля буде 16-бітний результат y. Для реалізації мультиплікатора необхідні 16-бітні суматори, що додають проміжні добутки, їх необхідно 7 штук. Для перевірки правдивості результатів виконано перевірку за допомогою симуляції в середовищі «Quartus». У побудові такого мультиплікатора допомагає мова програмування System Verilog HDL, яка є моделюванням функціональної схеми, яку вона описує.

Основний принцип написання програм — це опис модулів, які в свою чергу будуть слугувати частинами інших модулів і так далі до опису кінцевого бажаного модуля.

Щоб побудувати восьми-бітний мультиплікатор, скористаємося 16-бітним суматором, який ми отримаємо об'єднавши два 8-бітних суматори, описані раніше.

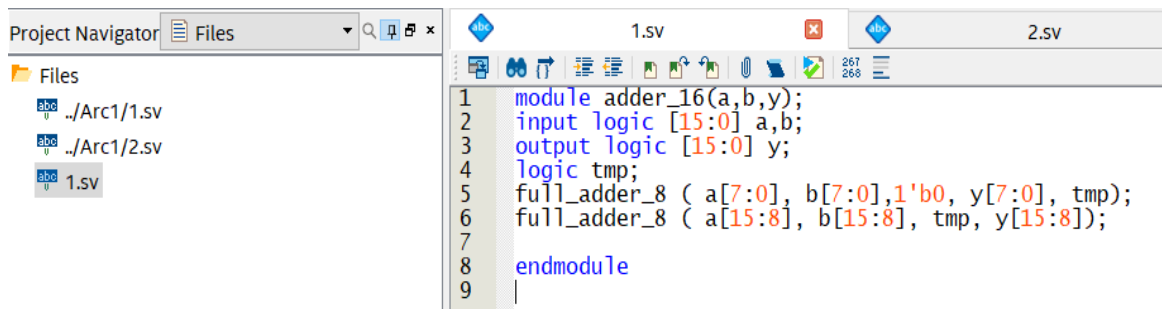


Рис. 3.49. Код для побудови 16-бітного суматора

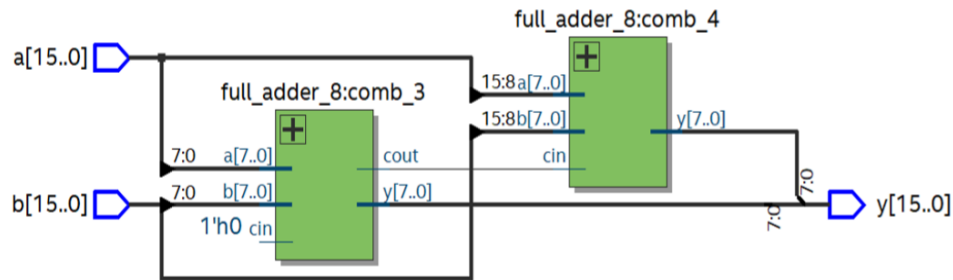


Рис. 3.50. Функціональна схема 16-бітного суматора

Для утворення 8-бітної шини потрібно включити біти від старшого до молодшого. У модуль входять дві 8-бітні шини a , b , cin , та на виході - y , $cout$. Повні суматори підключаються, та їх сигнали переносять біт переповнення, з кількістю 7 штук, що утворюють 7-бітну шину. Об'єднання у ланцюг N -повних суматорів є найпростішим способом реалізації N -розрядних суматорів. Вихід деякого розряду буде поступати на вхід наступного розряду. Після дослідження схеми реалізується 8-розрядний суматор, який написаний мовою SystemVerilog.

Щоб побудувати восьми-бітний мультиплікатор, потрібно використовувати процес-блок, який описує тип апаратури, як наприклад `always_comb`, що моделює комбінаційну логіку.

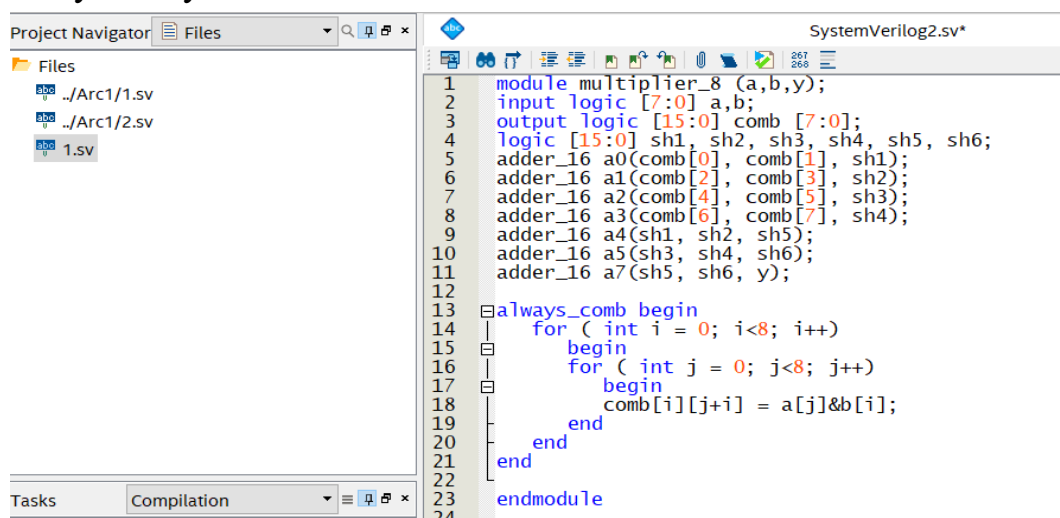


Рис. 3.51. Код програми мультиплікатора

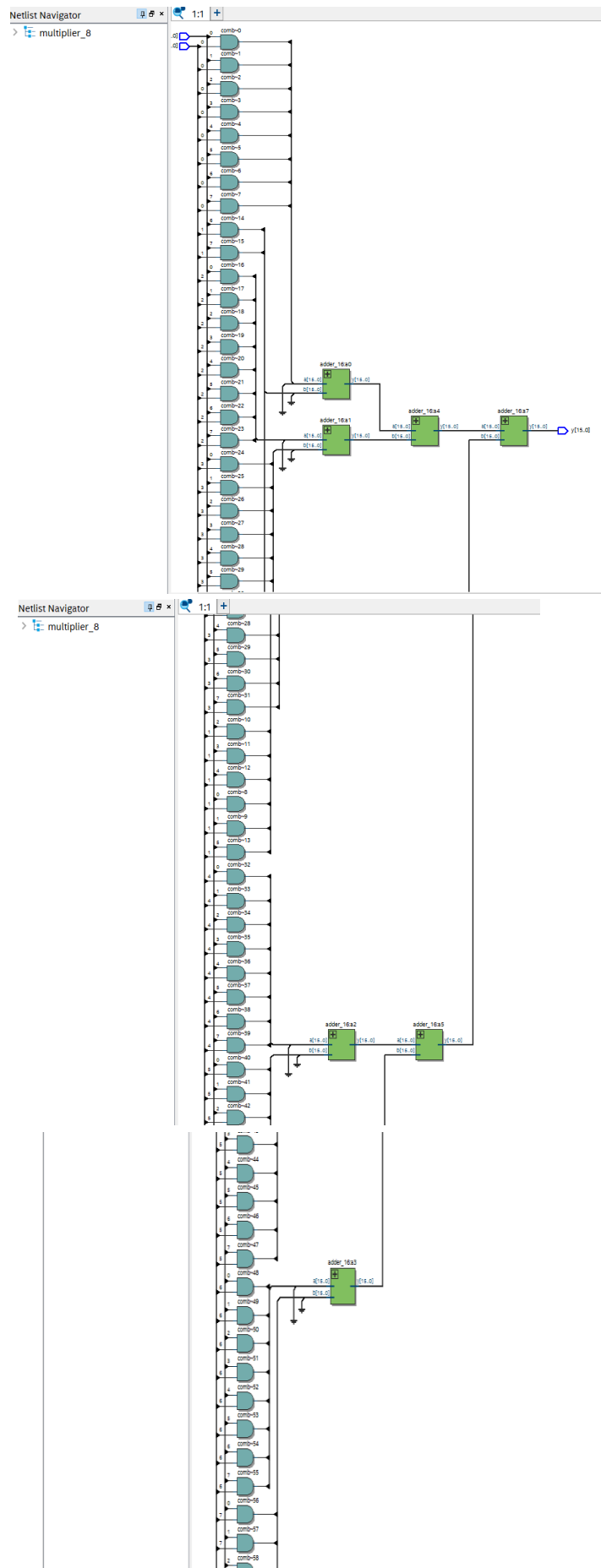


Рис.3.52. Функціональна схема 8-бітного мультиплікатора

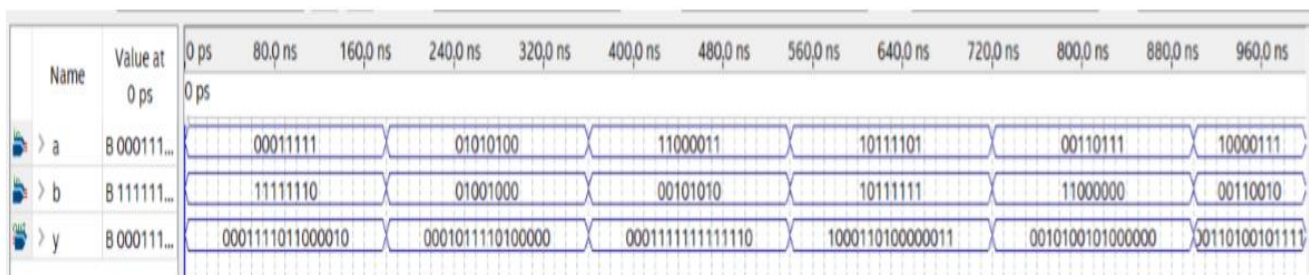


Рис. 3.53. Результати тестування 8-бітного мультиплікатора

Як бачимо, результати збігаються.

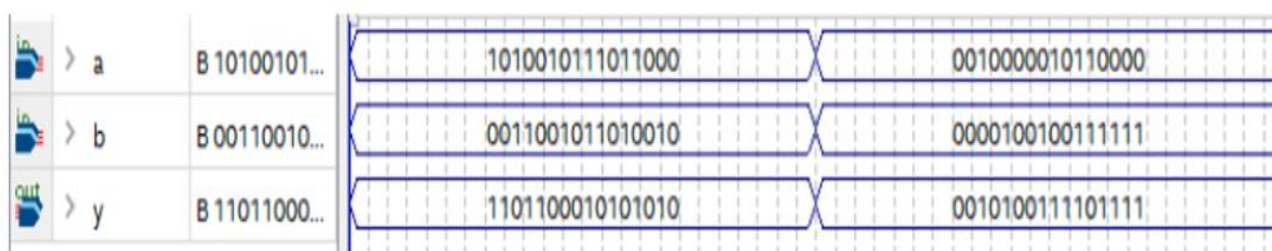


Рис.3.54. Результати тестування 16-бітного суматора

Висновки

У процесі виконання даної лабораторної роботи, було реалізовано мультиплікатор, використовуючи мову програмування HDL Verilog. Середовище виконання роботи Quartus Prime 16.1. Верифікація була перевірена за допомогою пакету QuartusModelSimAltera.

Контрольні запитання

1. Поясніть і опишіть структуру коду мультиплесора в даній роботі.
2. Поясніть і опишіть структуру коду 8-бітного мультиплесора в даній роботі.
3. Поясніть і опишіть структуру коду 16-бітного суматора в даній роботі.
4. Що собою представляє схема мультиплесора RTL-viewer?
5. Що собою представляє схема 16-бітного суматора RTL-viewer?
6. Поясніть роботу симуляції 16-бітного суматора.
7. Поясніть роботу симуляції мультиплесора.
8. Яким чином перевірити правильність результатів роботи мультиплесора та 16- бітного суматора.

3.4. Лабораторна робота № 4

Тема: Операція ділення

Постановка задачі:

Об'єкт дослідження: арифметико-логічний пристрій.

Мета роботи: спроектувати комбінаційні схеми для роботи алгоритмічно-логічного пристрою.

Методи дослідження та апаратура: формалізація АЛП за допомогою середовища проектування Quartus та мови опису апаратури SystemVerilog.

Арифметико-логічний пристрій (англ. Arithmetic Logic Unit, ALU) — блок процесора, що служить для виконання арифметичних та логічних перетворень над даними, що іменуються операндами. Цей пристрій є фундаментальною частиною будь-якого обчислювача, навіть найпростіші мікроконтролери мають його в складі свого ядра. Центральний процесор та відеопроцесор можуть мати кілька АЛП, що відрізняються своїм функціональним призначенням або типом оброблюваних даних.

Результати: реалізовано арифметико-логічний пристрій для виконання базових математичних операцій двох 8-бітних цілочисельних операндів.

Завданням даної лабораторної роботи є розроблення рішення для виконання операції ділення.

Вхідними даними мають бути два 8-бітні числа, що задані у бінарній системі числення є цілими.

Результатом має бути одне 8-бітне число.

Розробка має бути виконана у вигляді вентильної схеми і програмного коду Verilog HDL.

Завдання на лабораторну роботу:

Розробити архітектурне рішення для виконання операції ділення. Вхідними аргументами операції мають бути два восьми-бітні числа, що задані у бінарній системі числення, є цілими і не мають знаку. Результатом є одне восьми-бітне бінарне число та один біт помилки арифметики.

Розробка має бути виконана у вигляді вентильної схеми і програмного коду Verilog HDL. Вхідні аргументи повинні бути подані у вигляді логічних ліній $a[7:0]$, $b[7:0]$. Вихідне значення операції повинне бути подано у вигляді $y[7:0]$, $err[0]$.

Працездатність архітектури і розробки повинні бути показані у вигляді процесу роботи логічної вентильної схеми та у вигляді результатів її роботи, що

мають відповідати відповідним булевим законам.

Опис результатів, а також методів, підходів і засобів розробки повинні бути подані у вигляді звіту. Оформлення і вмісту звіту мають відповідати ДСТУ3008-95.

Розробка може проводитися у таких середовищах: MatLab SImulink; Multisim (EWB); ISIS Proteus; Quartus II.

Склад звіту лабораторної роботи

В звіті повинно бути відображено:

1. Тема роботи.
2. Мета роботи.
3. Постановка задачі.
4. Виконання завдання з відповідними скріншотами.
5. Відповіді на контрольні запитання.
6. Висновок.

Ділення. Основні компоненти Реалізація у Quartus Prime 1

Побудова 8-бітного дівіатора.

Схема ділення була розроблена на основі методу ділення двох бінарних чисел без відновлення остатку. Суть алгоритму полягає в наступному:

1. Віднімаємо від діленого А дільник В;
2. Аналізуємо знак отриманого часткового залишку У реєстр результату записуємо "0" якщо залишок від'ємний і одиницю в іншому випадку. Пам'ятаємо, що негативного числа відповідає наявність одиниці в 15-м розряді і навпаки;
3. Здвигаємо частковий залишок на один розряд вліво;
4. Додаємо до часткового залишку дільник В якщо залишок від'ємний або віднімаємо дільник в іншому випадку;
5. Аналізуємо знак отриманого часткового залишку. У реєстр результату записуємо "0" якщо залишок від'ємний і одиницю в іншому випадку;
6. Дії, описані в пунктах 3-5 виконуємо 8 разів.

Створимо дівіатор на мові SystemVerilog, для цього спочатку створимо 16-бітний мультиплексор, отримаємо такий код:

```

1  module multiplexor(input[16:0]a, input[16:0]b, output logic[16:0]y, output logic num);
2      reg [16:0] change;
3      always@(a,b)
4      begin
5          change = a-b;
6          if (change[16])
7              begin
8                  y <=a<<1;
9                  num = 0;
10             end
11         else
12             begin
13                 y<=change<<1;
14                 num = 1;
15             end
16         end
17     endmodule

```

Рис.3.55. Код програми мультиплексора

Після компіляції коду отримаємо таку схему:

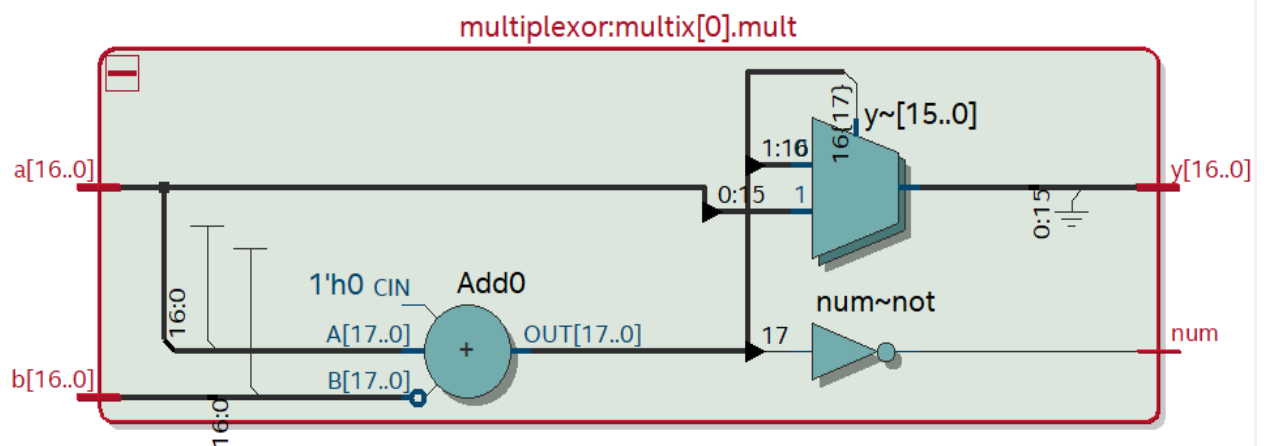


Рис.3.56. Функціональна схема мультиплексора

Тоді код 8-бітного діватора буде мати наступний вигляд:

```

1  module div_8(input [7:0]a, input[7:0]b,output logic[7:0]y, output logic err);
2
3      logic[16:0] change [8:0];
4      logic[15:0] c;
5      genvar i;
6      generate
7          for(i=0;i<8;i=i+1)
8              begin: multix
9                  multiplexor mult(change[i],c,change[i+1],y[7-i]);
10             end
11         endgenerate
12
13     always@(a,b)
14     begin
15         c = b<<8;
16         change[0]=a<<1;
17         if(!(b[7:0]))
18             begin
19                 err<=1'b1;
20             end
21         else
22             begin
23                 err<=1'b0;
24             end
25         end
26     end
27 endmodule

```

Рис.3.57. Код програми 8-бітного діватора

Після компіляції коду отримаємо таку схему:

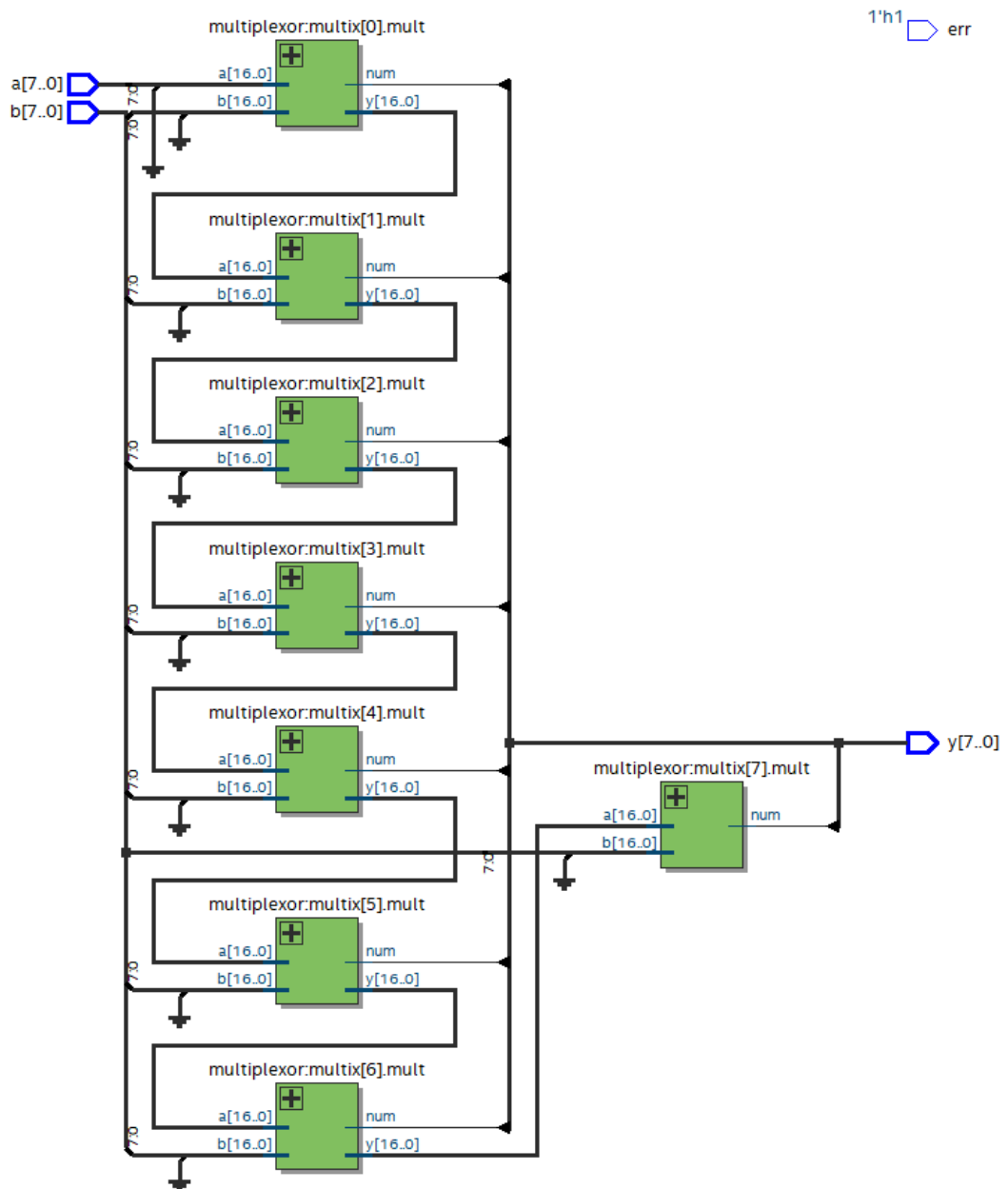


Рис.3.58. Функціональна схема дівіатора

Протестуємо наш дівіатор на випадкових вхідних даних та переконаємось у його коректній роботі:

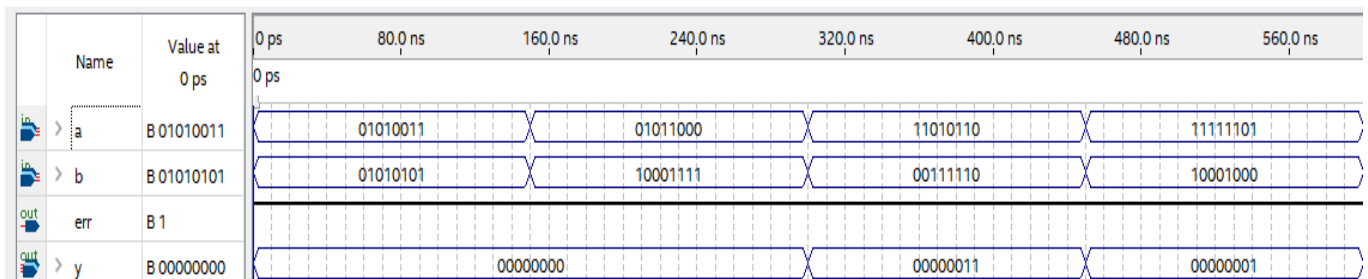


Рис.3.59. Код перевірки дівіатора

Реалізація у Proteus

Постановка задачі

Необхідно розробити архітектурне рішення, задля виконання операції ділення. Вхідними аргументами операції мають бути два 8-бітні числа(задані у бінарній системі і цілі). Результатом є одне 8-бітне число (теж бінарне) та один біт помилки арифметики. Розробка має бути виконана у вигляді вентильної схеми і програмного коду HDL Verilog. Працездатність архітектури і розробки повинні бути показані у вигляді процесу роботи вентильної схеми та у вигляді результатів її роботи, що мають відповідати булевим законам.

Аналіз реалізації блоку ділення

Блок ділення – блок, призначений для утворення частки двох операндів. Він складається з окремих схем, які виконують роль віднімання та перевірку на помилку операції.

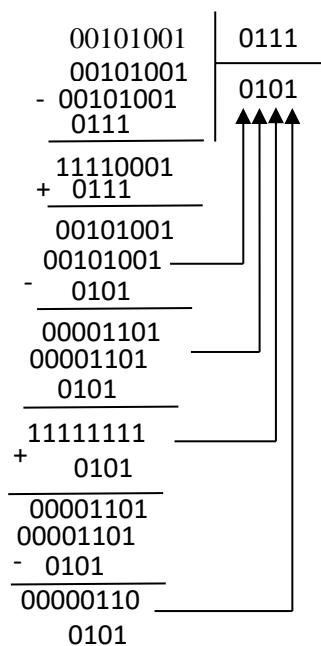


Рис.3.60. Приклад ділення з відновленням остачі

Моделювання блоку ділення

Частковий залишок R ініціалізується 0. Найбільший значущий розряд діленого A стає найменш значущим розрядом R . Дільник V багаторазово віднімається від часткового залишку та визначається знак різниці D . Якщо вона від'ємна, то розряд частки Q_i дорівнює 0 та різниця частки відкидається. В протилежному випадку Q_i дорівнює 1 і частковий залишок оновлюється, він стає рівним різниці D . Далі частковий залишок здвигається ліворуч на 1 розряд, і процес повторюється. Результат задовольняє умові $A/B=Q+R/V$.

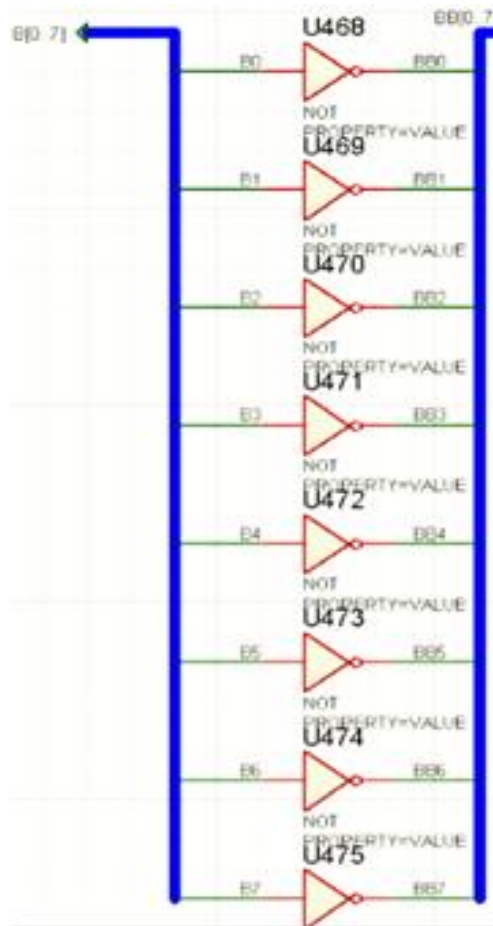


Рис.3.61. Інверсія дільника

Як ми можемо побачити на рисунку 2: $B[0..7]$ – вхід дільника, $BV[0..7]$ – $DIVRAP1, DIVPART2, CHECK_1$.

На наступному етапі (рис.5.8) відбувається перевірка на перший біт переповнення (U480), який буде дорівнювати 1, якщо код шини ($C0, C1$) буде «1» та «1». Також відбувається перевірка на ділення на нуль ($CHECK_1$).

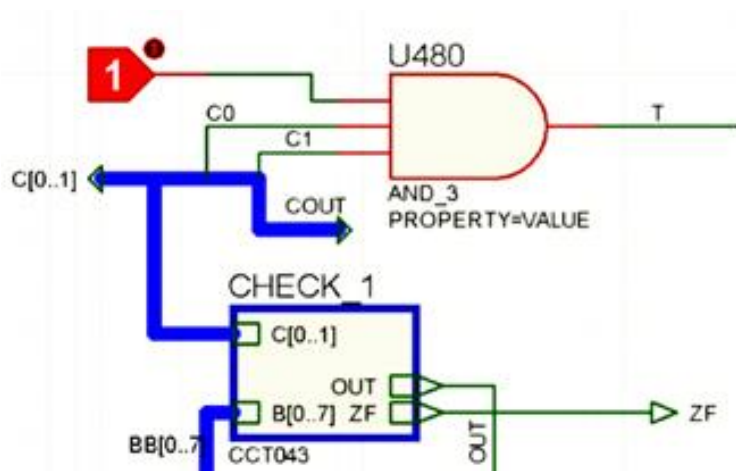


Рис.3.62. Перевірка на нуль

«Розшифруємо» позначення на рисунку: C[0..1] – вхід шини команди; T-DIVPAT1, DIVPART2, COUT – вихід шини команди, BB[0..7] – інверсія дільника, OUT – оператори «і», ZF – вихід прапорця ділення на нуль.

На наступному рисунку показана схема CHECK_1, яка перевіряє, чи дільник дорівнює 0. B[0..7] – вхід дільника, ZF- вихід прапорця ділення на нуль, OUT – вихід прапорця переповнення.

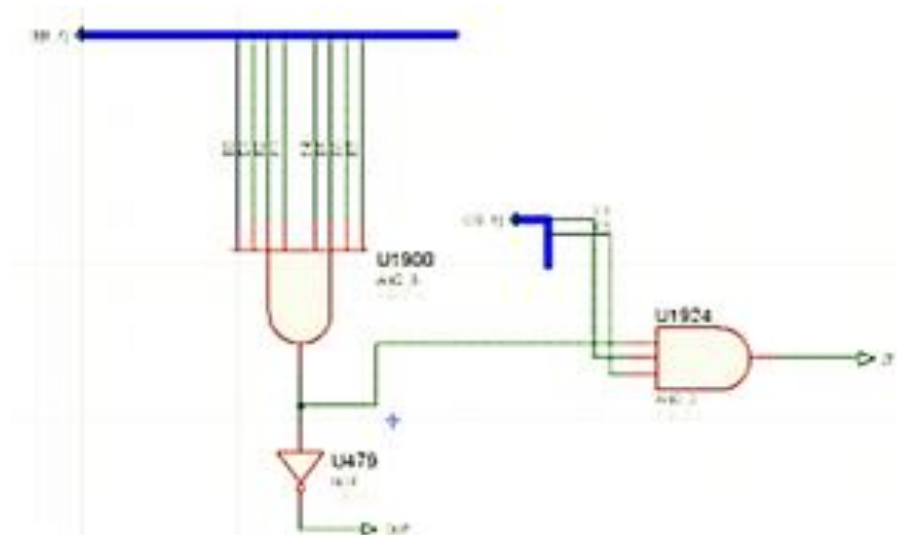


Рис.3.63. Перевірка на нуль

На рисунку 5 показана перевірка вхідних даних в залежності від значення прапорця ділення на нуль (ZF) за допомогою вентилів «і». $OUT = CHECK_1$, $ZF[0..3] \text{ DIVPART2}$, $ZFF[0..3] \text{ -DIVPART1}$, A1..A0 – вхід діленого, B[0..7]- шина дільника.

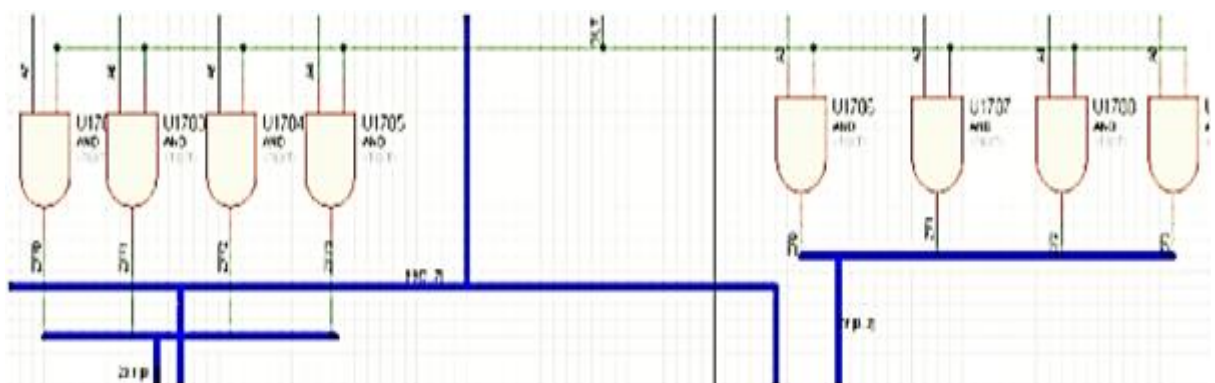


Рис.3.64. Перевірка вхідних даних

Далі було реалізовано сам процес ділення з використанням двох схем DIVPART2 та DIVPART1. На вхід поступають нульові значення переповнення. Остача передається у другий блок ділення. З останнього блоку ділення остача

переходить у відповідь через перевірку ділення на нуль. T-U480, BV[0..7] – інверсія дільника QQ[0..3], Q[0..3], R[0..7] – перевірка ZF при виході, ZFF[0..3] ZF[0..3] – перевірка вхідних даних.

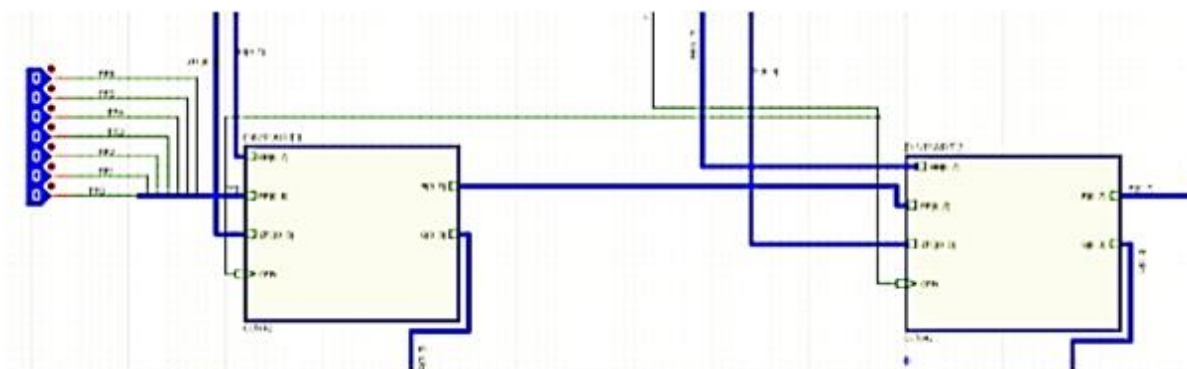


Рис.3.65. Ділення

Схема DIVPART1 та DIVPART2, вони однакові.

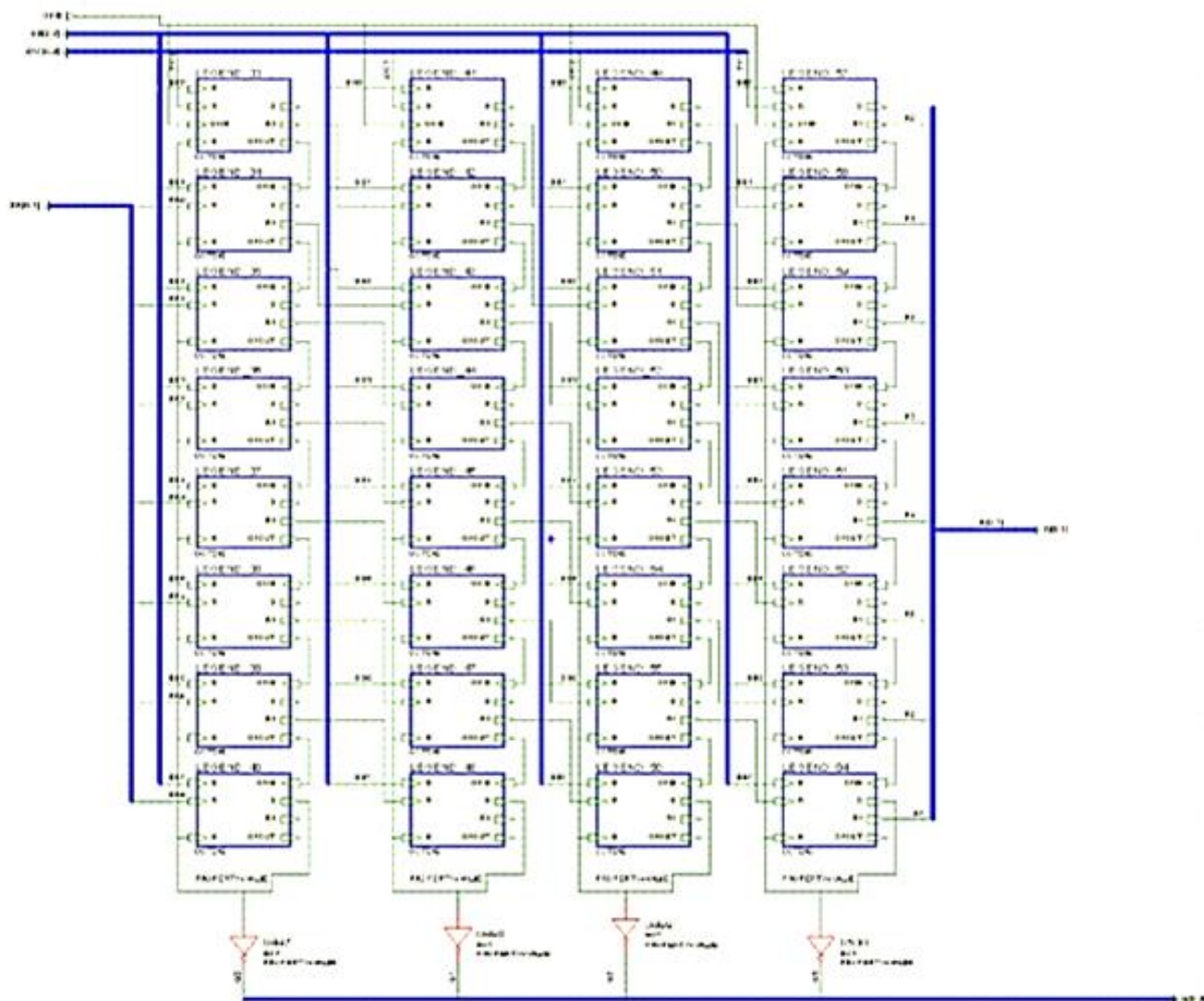


Рис.3.66. Блок ділення

Відтворена перевірка на коректний вивід відповіді в залежності від значення ZF.

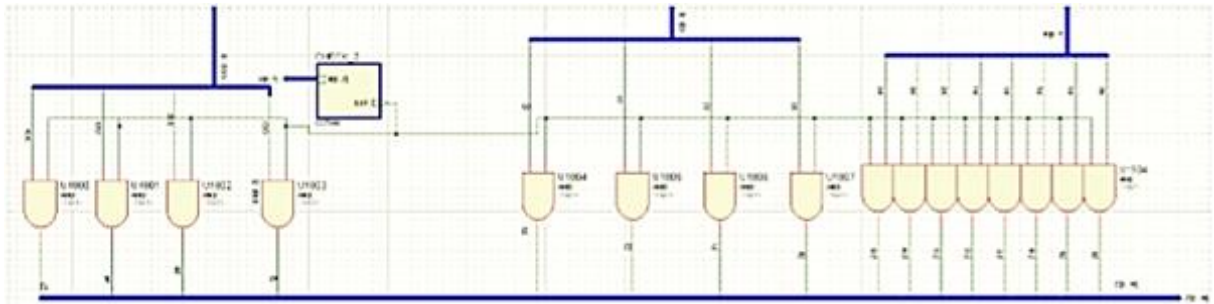


Рис.3.67. Перевірка відповіді

CHECK_2 робить те саме, що й CHECK_1 – перевіряє дільник на значення «0» і виходячи з нього, фільтрує відповідь. Друга перевірка була зроблена для коректного виводу значень при діленні на нуль. Протестуємо 8-розрядний блок ділення

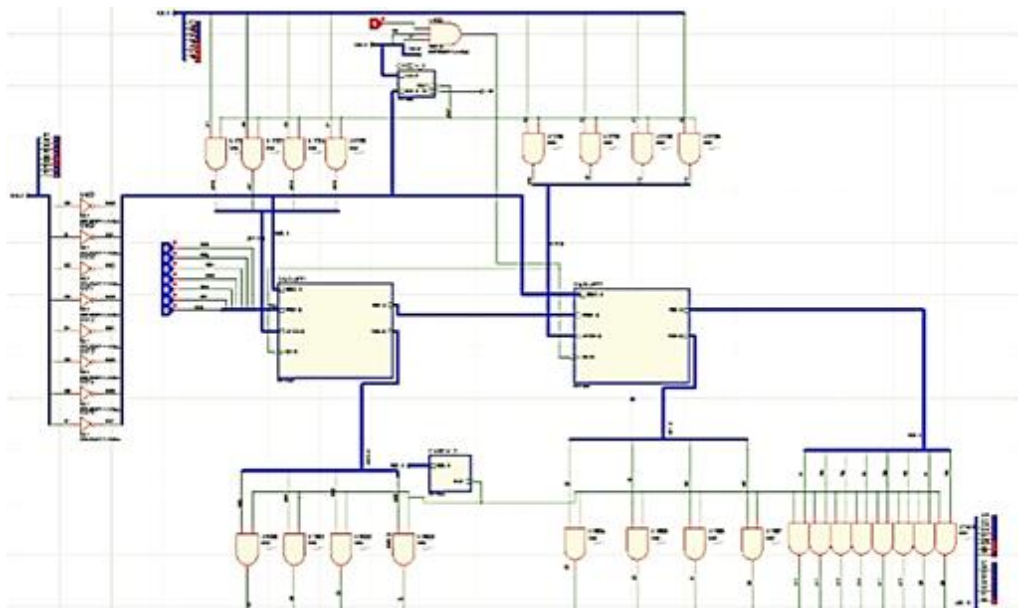


Рис.3.68. Функціонуючий 8-розрядний блок ділення

На даному рисунку видно, що блок ділення правильно провів операцію ділення над двома 8-розрядними числами як було показано у прикладі.

Висновки

У даній роботі нашою бригадою було успішно реалізовано 8-ми розрядний функціонуючий блок ділення з використанням програмного забезпечення PROTEUS. Він приймає 2 8-ми розрядні значення та видає одне 8-ми розрядне значення (цілу частину) та друге (остача). Перевірку він пройшов успішно.

У цій роботі було розроблено восьми-бітну операцію ділення на основі мультиплексора та операції віднімання. Апаратна частина будована на основі зсуву операції віднімання і циклу.

Контрольні запитання

1. Поясніть і опишіть структуру коду мультиплесора в даній роботі.
2. Поясніть і опишіть структуру коду 8-бітного дівіатора в даній роботі.
3. Що собою представляє схема мультиплесора RTL-viewer?
4. Що собою представляє схема дівіатора RTL-viewer?
5. Поясніть роботу симуляції мультиплесора.
6. Яким чином перевірити правильність результатів роботи дівіатора .

3.5. Лабораторна робота № 5

Тема: Алгоритмічно-логічний пристрій (АЛП)

Постановка задачі:

Об'єкт дослідження: арифметико-логічний пристрій.

Мета роботи: спроектувати комбінаційні схеми для роботи алгоритмічно-логічного пристрою.

Методи дослідження та апаратура: формалізація АЛП за допомогою середовища проектування Quartus та мови опису апаратури SystemVerilog.

Арифметико-логічний пристрій (англ. Arithmetic Logic Unit, ALU) — блок процесора, що служить для виконання арифметичних та логічних перетворень над даними, що іменуються операндами. Цей пристрій є фундаментальною частиною будь-якого обчислювача, навіть найпростіші мікроконтролери мають його в складі свого ядра. Центральний процесор та відеопроцесор можуть мати кілька АЛП, що відрізняються своїм функціональним призначенням або типом оброблюваних даних.

Результати: реалізовано арифметико-логічний пристрій для виконання базових математичних операцій двох 8-бітних цілочисельних операндів.

Завданням даної лабораторної роботи є розроблення рішення для виконання АЛП.

Вхідними даними мають бути два 8-бітні числа, що задані у бінарній системі числення є цілими.

Результатом має бути одне 8-бітне число.

Розробка має бути виконана у вигляді вентиляльної схеми і програмного коду Verilog HDL.

Завдання на лабораторну роботу:

Розробити архітектурне рішення для виконання вибору коду операції (додавання, віднімання, множення чи ділення). Вибір коду операції повинен ґрунтуватися на вхідному значенні $c[3:0]$, що визначає шифр операції (додавання: 00; віднімання: 01; множення: 10; ділення: 11). Архітектурне рішення повинне перемикає шини вхідних параметрів (a,b) між блоками, що були отримані у ході роботи над попередніми завданнями. Також має бути розроблене рішення для формування єдиного виходу, що враховуватиме восьми та шістнадцяти бітні варіанти операції (див. операцію ділення). Результатом виконання завдання має бути графічний блок, що має три входи $a[7:0]$, $b[7:0]$, $c[3:0]$ та вихід $y[uH[7:0], yL[7:0]]$ із урахуванням знакових бітів $err[0]$ і $of[0]$.

Розробка має бути виконана у вигляді вентиляльної схеми і програмного коду Verilog HDL. Вхідні аргументи повинні бути подані у вигляді логічних ліній $a[7:0]$, $b[7:0]$. Вихідне значення операції повинне бути подано у вигляді $y[7:0]$, $err[0]$.

Працездатність архітектури і розробки повинні бути показані у вигляді процесу роботи логічної вентиляльної схеми та у вигляді результатів її роботи, що мають відповідати відповідним булевим законам.

Опис результатів, а також методів, підходів і засобів розробки повинні бути подані у вигляді звіту. Оформлення і вмісту звіту мають відповідати ДСТУ 3008-95.

Розробка може проводитися у таких середовищах: MatLab SImulink; Multisim (EWB); ISIS Proteus; Quartus II.

Склад звіту лабораторної роботи

В звіті повинно бути відображено:

1. Тема роботи.
2. Мета роботи.
3. Постановка задачі.
4. Виконання завдання з відповідними скріншотами.
5. Відповіді на контрольні запитання.
6. Висновок.

АЛП. Основні компоненти Реалізація у Quartus Prime 1

Побудова арифметично-логічного пристрою

Для створення схеми арифметично–логічного пристрою потрібно об'єднати пристрої, які ми розробили попередньо, а саме суматор, субтрактор, пристрій множення та ділення (далі арифметичні пристрої). Принцип роботи такий: на вхід схеми подається два 8-бітні числа (А та В) та 2-бітне число коду операції (С). Далі вхідні числа та код операції подаються на вхід демультимплексору. Демультимплексор складається з 8 демультимплексорів, які на вхід приймають два однобітні числа аргументів та код операції.

Створимо АЛП на мові SystemVerilog і отримаємо такий код:

```
1 module alus(input logic [7:0]a, b,  
2             input logic c[1:0],  
3             output logic [15:0]y,  
4             output logic err);  
5     logic err1, err2, err4;  
6     logic [8:0] y1, y2, y4;  
7     logic [15:0] y3;  
8     fulladder_8 a1(a, b, 1'b0, y1, err1);  
9     subtractor_8bit a2(a, b, 1'b0, y2, err2);  
10    multiplexor a3(a, b, y3);  
11    div_8 a4(a, b, y4, err4);  
12    assign y = c[0] ? (c[1] ? y4 : y3) : (c[1] ? y2 : y1);  
13    assign err = c[0] ? (c[1] ? err4 : 1'b0) : (c[1] ? err2 : err1);  
14  
15    endmodule  
16
```

Рис.3.69. Код програми АЛП

Після компіляції коду отримаємо таку функціональну схему:

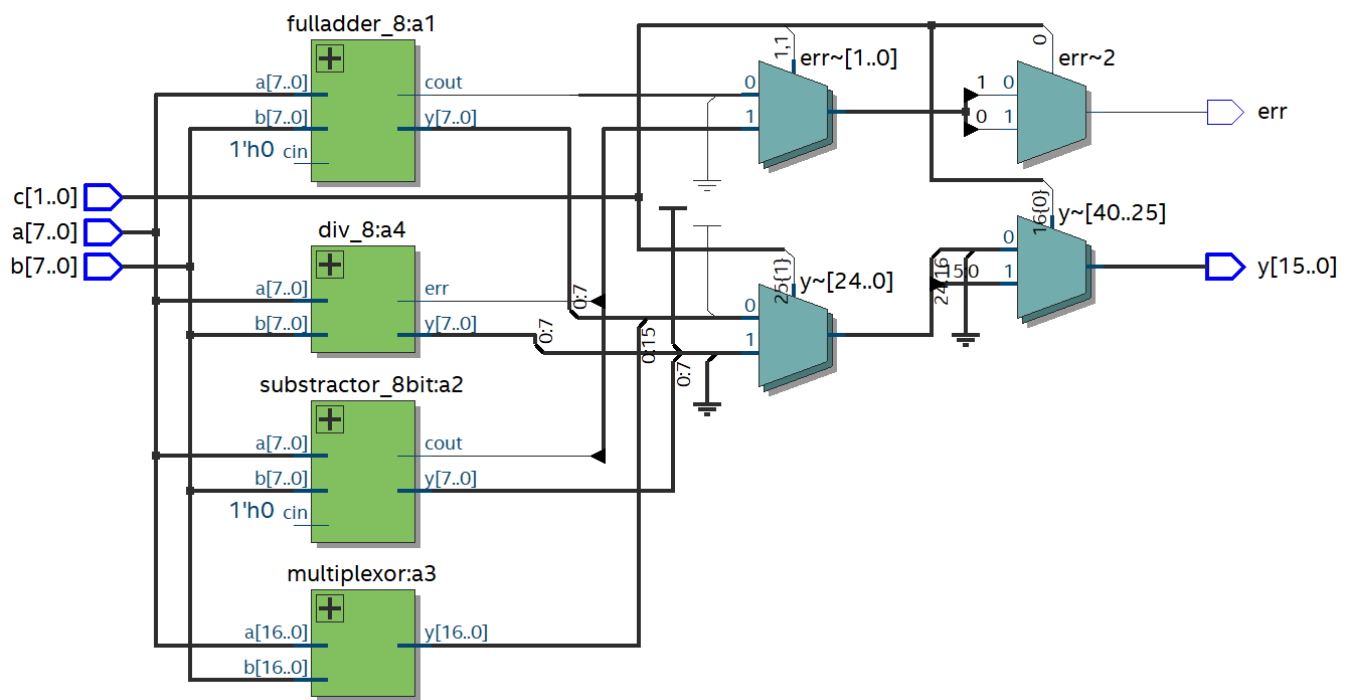


Рис.3.70. Функціональна схема АЛП

Протестуємо наш АЛП на випадкових вхідних даних та переконаємось у його коректній роботі:

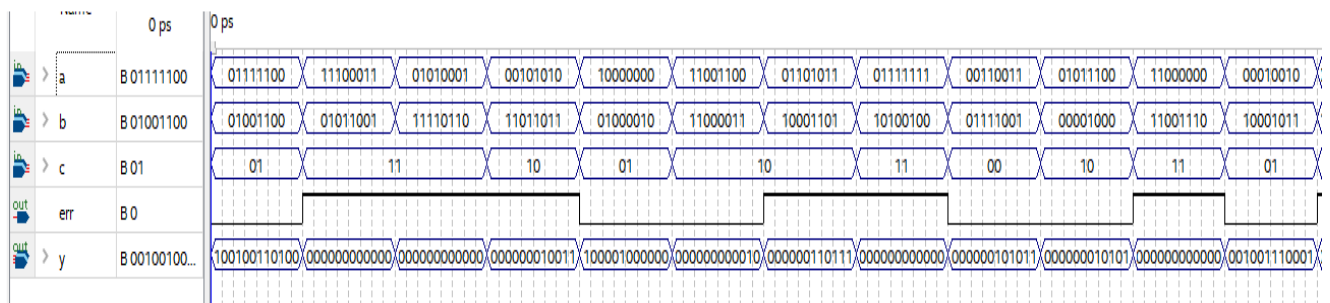


Рис.3.71. Результати перевірки АЛП

Реалізація у Proteus

Постановка задачі

Розробити архітектурне рішення для виконання вибору коду операції (додавання, віднімання, множення чи ділення). Вибір коду операції повинен ґрунтуватися на вхідному значенні $c[3:0]$, що визначає шифр операції (додавання: 00; віднімання: 01; множення: 10; ділення: 11). Архітектурне рішення повинне перемикає шини вхідних параметрів (a,b) між блоками, що були отримані у ході роботи над попередніми завданнями. Також має бути розроблене рішення для формування єдиного виходу, що враховуватиме восьми та шістнадцяти бітні

варіанти операції (див. операцію ділення). Результатом виконання завдання має бути графічний блок, що має три входи $a[7:0]$, $b[7:0]$, $c[3:0]$ та вихід $y[yH[7:0], yL[7:0]]$ із урахуванням знакових бітів $err[0]$ і $of[0]$.

Аналіз реалізації АЛП

АЛП (алгебраїчний логічний пристрій) — призначений для реалізації однієї з операцій (сума, різниця, множення, ділення) в залежності від станів бітів у шині команди (що було описано вище).

Необхідно мінімізувати витрати енергії на «непотрібні» дії та мінімізувати кількість вентилів у схемі.

Слід зазначити, що ці команди ніяк не впливатимуть на роботу з пам'яттю, і нам не доведеться змінювати саме АЛП для підключення пам'яті. Отже, на АЛП поступає деяка команда сигнал проходить по всім модулям, але вихідний мультиплексор обирає, з якого саме модуля він повинен зчитати бажаний результат. Якщо обрана дія додавання або віднімання результат потрібно зчитати з суматора, якщо — множення, то з помножувача, якщо ж — ділення, то з дільнику.

Моделювання АЛП

Для початку створимо головний блок мультиплексора (який буде «перевіряти» код операції):

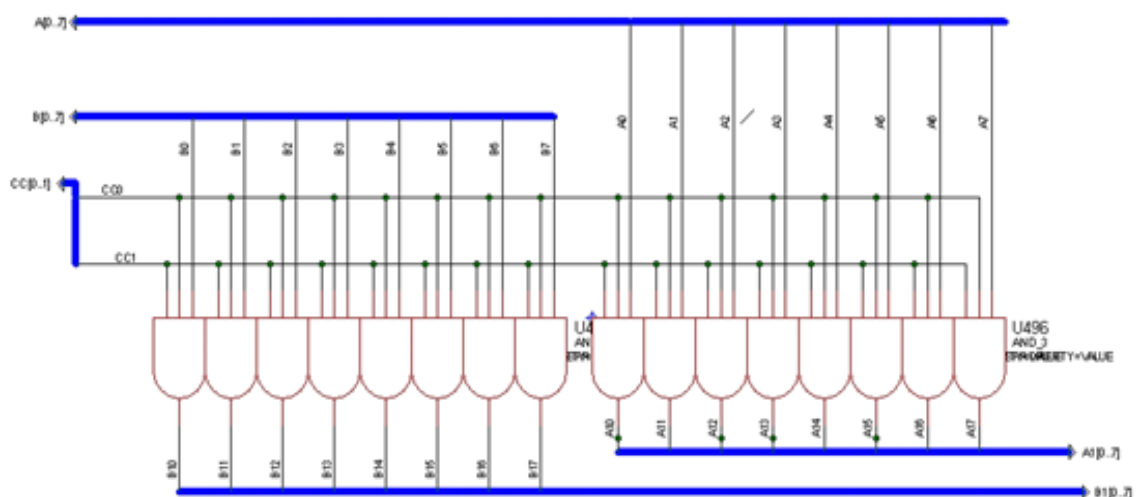


Рис.3.72. Перевірка коду операції

$A[0..7]$ — перший операнд, $B[0..7]$ -другий операнд, $CC[0..1]$ -шина коду, $A1[0..7]$, $B1[0..7]$ - перевірений операнд.

З даного рисунка видно, що ми робимо перевірку вхідних даних із кодом операції за допомогою використання 16 вентилів «і».

Перерахуємо можливі інверсії коду команди для відповідних операцій:

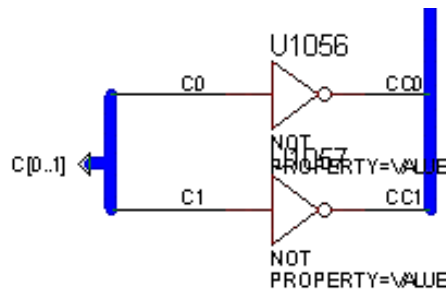


Рис.3.73. Інверсія для операції додавання

C[0..1] -шина коду, CC06 CC1 – перевірена шина коду.

З рисунка бачимо, що перевірка пройде вірно, лише якщо на вхід поступить 2 біти із «0» та інвертуються у «1».

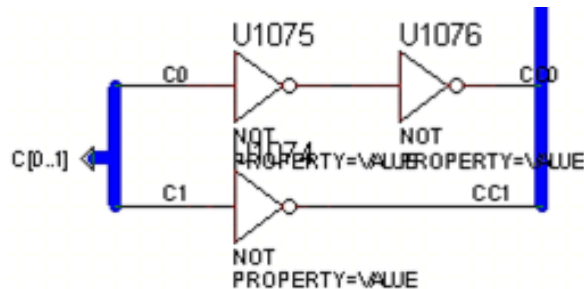


Рис.3.74. Інверсія для операції віднімання

За аналогією із операцією додавання, необхідно, щоб на вхід потрапило значення коду операції «1» та «0», а потім інвертувалось для коректної роботи.

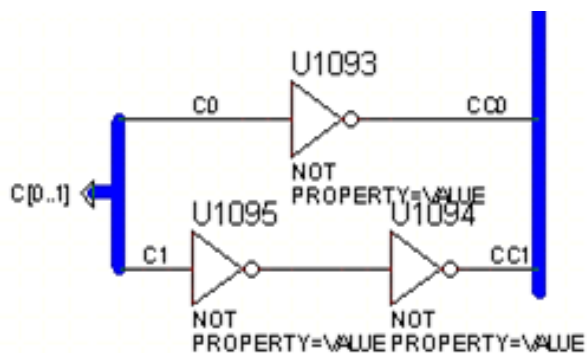


Рис. 3.75. Інверсія для операції множення

Тут необхідно, щоб на вхід потрапили значення «0» та «1», а потім інвертувались, для того щоб усе працювало успішно для множення. А для блоку ділення інвертація не потрібна, оскільки код операції [1,1], а це якраз і є необхідні біти для коректного виконання перевірки. На виході із операції робимо

демультиплексор, які приймає всі 5 значень, а на виході видає одне значення. Отримали схему:

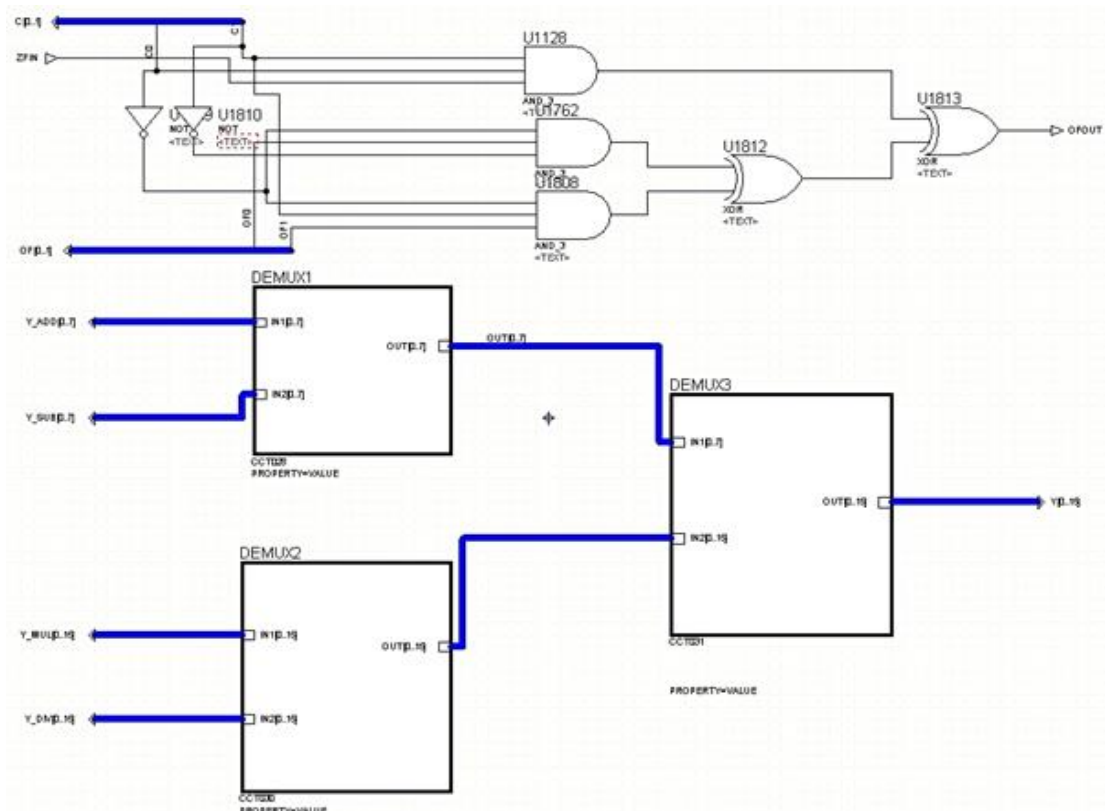


Рис. 3.76. Демультіплексор

C[0..1] – шина коду операції, ZFIN – прапорець ділення на нуль, OF[0..1] – прапор переповнення, Y_ADD[0..7], Y_MUL[0..7], Y_DIV[0..7], Y[0..15] – відповідь з відповідного блоку.

Кожен з цих блоків складається із вентилів «XOR», у кількості 8, або 16 штук, в залежності від операндів входу.

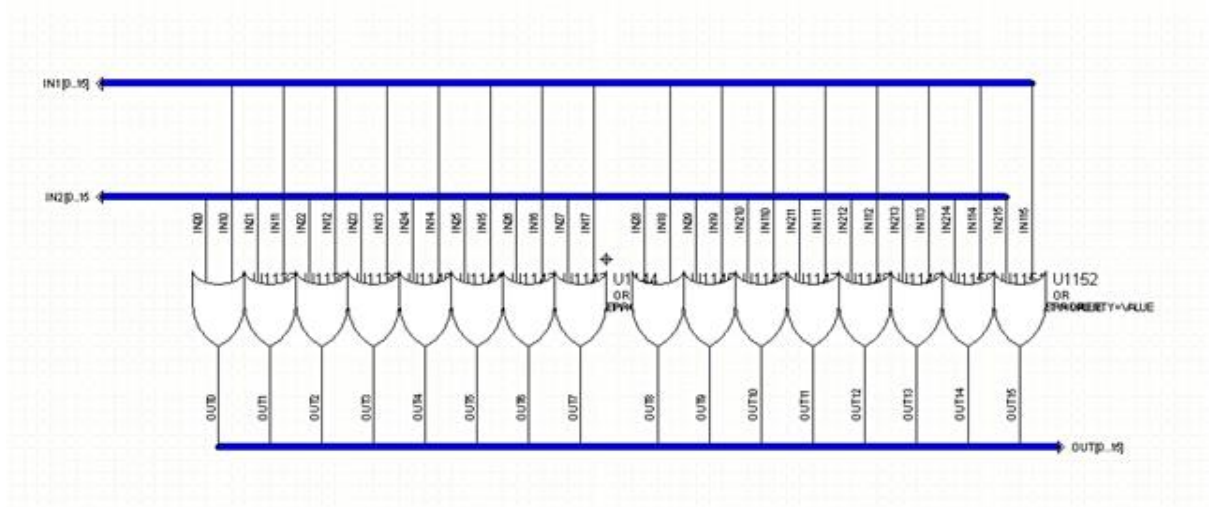


Рис.3.77. Етап демультіплексора

На рис. 6.10 реалізовано вивід одного із операндів (множення або ділення), та подальшу подачу результату у DEMUX3. У блоці DEMUX1 схема вдвічі менша (8 біт), оскільки операнди мають по 8 біт результату. DEMUX3 має наступний вигляд:

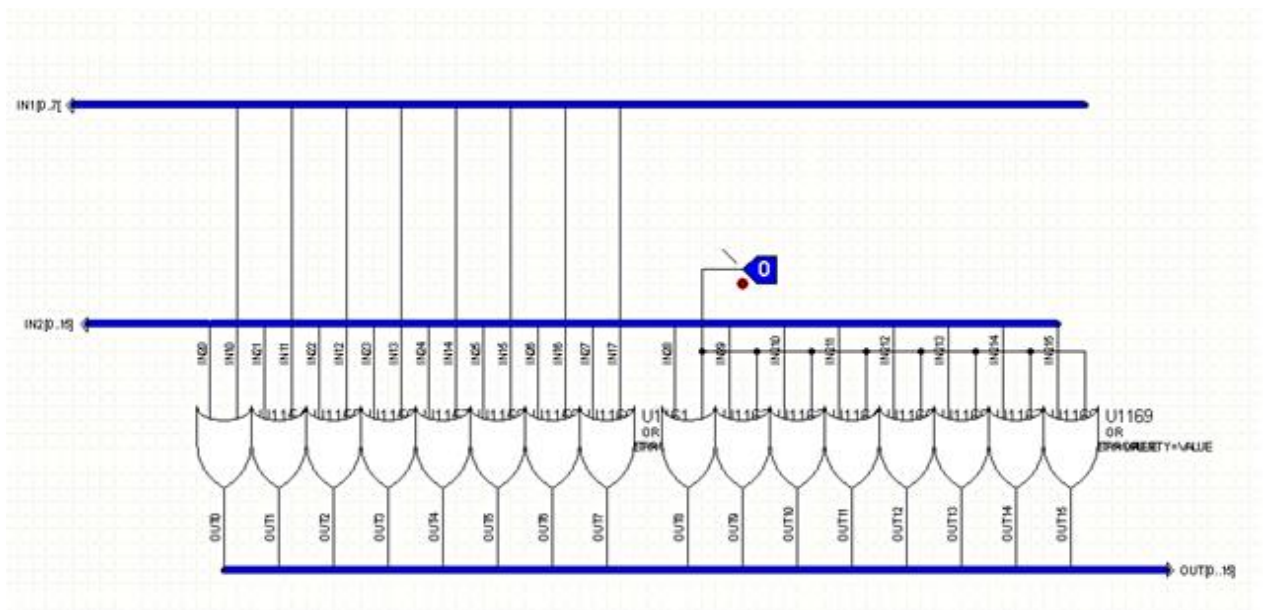


Рис.3.78. Останній етап демультиплексора

Єдина відмінність DEMUX3 від попередніх в тому, що тут присутні «0» біти для старших 8-и бітів, це пов'язано з тим, що тут поєднуються 8-и та 16-и бітні шини. На виході отримуємо готовий результат.

Після компіляції всіх етапів було отримано наступну схему:

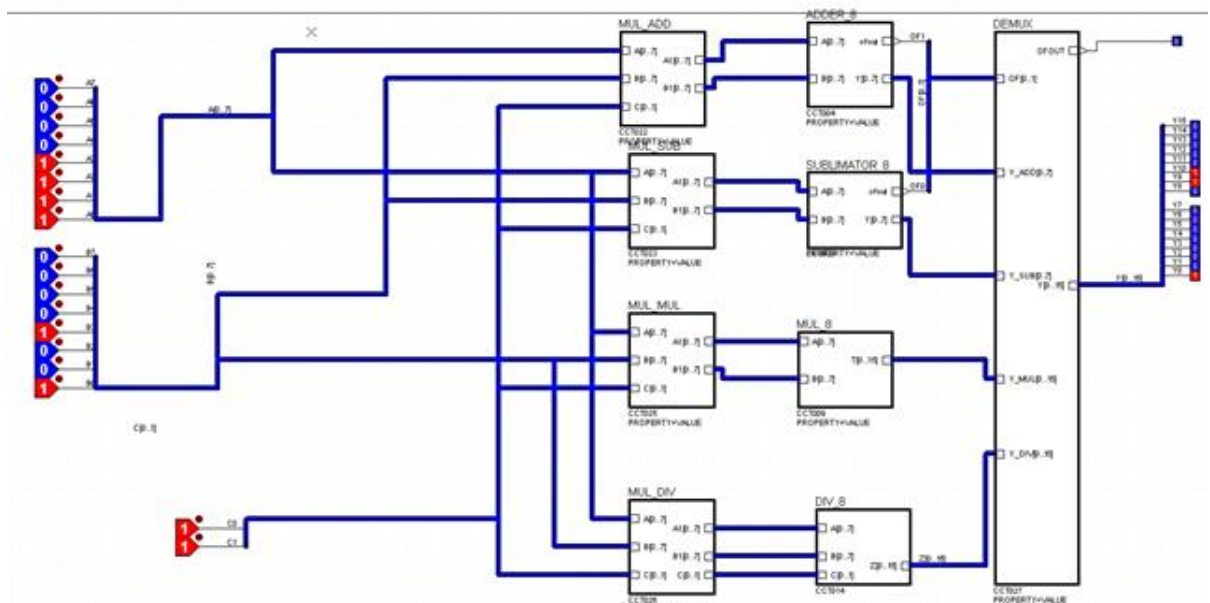


Рис.3.79. АЛП перший

Тестуємо АЛП

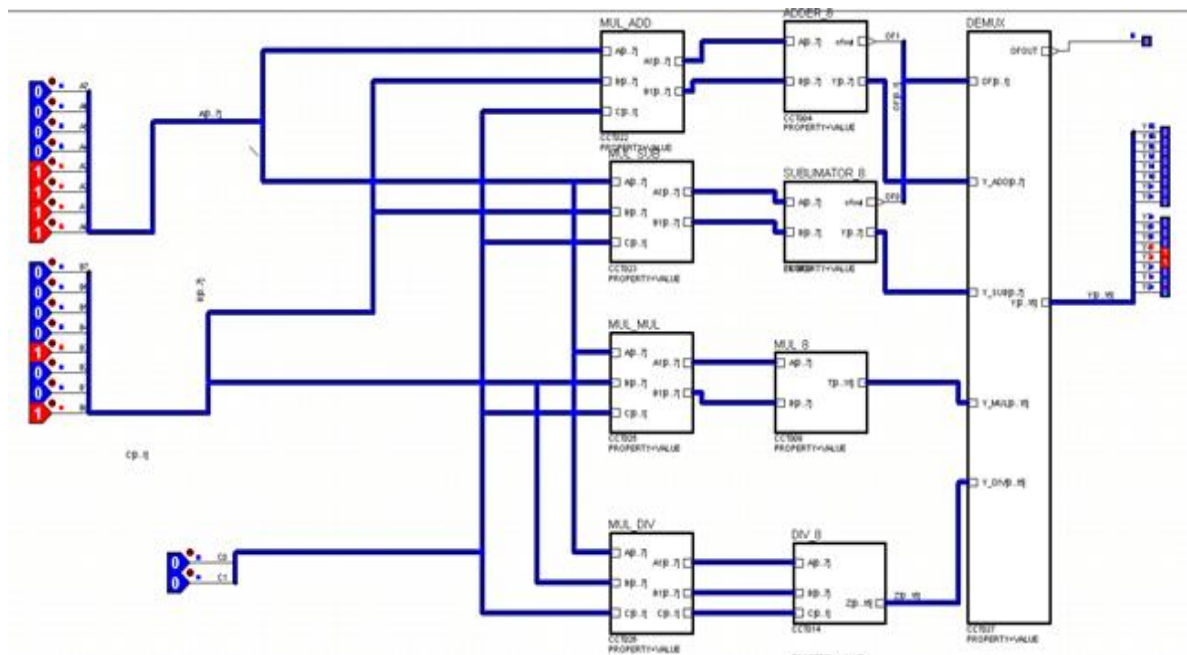


Рис.3.80. АЛП другий

Порівнюючи із рисунком 8 можна помітити коректність роботи приладу, а саме $15/9=1+6/9$ при $[1,1]$ (діленні) та $15+9=12$ ($[0,0]$ при додаванні).

Висновки

У ході виконання даної роботи було реалізовано 16-ти розрядний АЛП. Він коректно обирає команду операцій та оброблює її, та виводить відповідь. Перевірку пройшов успішно, і результати відповідають очікуванням. Виконуючи лабораторну роботу, було реалізовано арифметико-логічний пристрій, використовуючи суматор, субстрактор, мультиплексор і дівайдер. Окрім цього, використано шістнадцятибітний суматор, що використовується в реалізації восьмибітний дівайдер. Під час виконання роботи було використано базові можливості середовища Quartus та мови для побудови функціональних схем System Verilog HDL.

Контрольні запитання

1. Поясніть і опишіть структуру коду АЛП в даній роботі.
2. Що собою представляє схема АЛП RTL-viewer?
3. Поясніть роботу симуляції АЛП.
4. Яким чином перевірити правильність результатів роботи АЛП.

3.6. Лабораторна робота № 6

Тема: Тригер, регістр, JK-тригер, JK-регістр

Постановка задачі:

Об'єкт дослідження: створення тригер, регістр, JK-тригера, JK-регістру.

Мета роботи: спроектувати комбінаційні схеми для роботи JK-тригера та регістру.

Методи дослідження та апаратура: формалізація JK-тригера та регістру за допомогою середовища проектування Quartus та мови опису апаратури SystemVerilog.

Результати: реалізовано JK-тригер та регістр для виконання базових математичних операцій.

Завданням даної лабораторної роботи є розроблення рішення для виконання JK-тригера та регістру.

Вхідними даними мають бути тригер з сигналами j, k, c, що задані у бінарній системі числення.

Розробка має бути виконана у вигляді вентильної схеми і програмного коду Verilog HDL.

Завдання на лабораторну роботу:

Розробити архітектурне рішення тригеру, регістру, JK-тригеру JK-регістра. Також розробити схему для збереження результату виконання операції. Засоби збереження аргументів та результату (далі - “буфер”) мають бути виконані у вигляді восьмибітних регістрів. Буфери повинні бути оснащені каналом управління Reset.

Розробка має бути виконана у вигляді вентильної схеми і програмного коду Verilog HDL.

Працездатність архітектури і розробки повинні бути показані у вигляді процесу роботи логічної вентильної схеми.

Опис результатів, а також методів, підходів і засобів розробки повинні бути подані у вигляді звіту. Оформлення і вмісту звіту мають відповідати ДСТУ3008-95.

Розробка може проводитися у таких середовищах: MatLab SImulink; Multisim (EWB); ISIS Proteus; Quartus II.

Склад звіту лабораторної роботи

В звіті повинно бути відображено:

1. Тема роботи.
2. Мета роботи.
3. Постановка задачі.
4. Виконання завдання з відповідними скріншотами.
5. Відповіді на контрольні запитання.
6. Висновок

Тригер, регістр, JK-тригер, JK-регістр. Основні компоненти Реалізація у Quartus Prime 1

Завдання

Побудова тригеру

Опишемо тригер, чутливий до рівня вхідних сигналів а і b, пов'язаних з його входом. У всіх цих випадках, сигнал "у" зберігається, якщо поточне значення сигналу enable відмінно від '1'.

Створимо тригер на мові SystemVerilog і отримаємо такий код:

```
1  module latch1
2  □ ( input a, b, input enable,
3    □ output reg y);
4    always @(enable or a or b)
5
6  □ begin
7    □ if (enable)
8    □   y = a & b;
9    □ end
10   endmodule
11
12
```

Рис.3.81. Код програми latch

Після компіляції коду отримаємо таку функціональну схему:

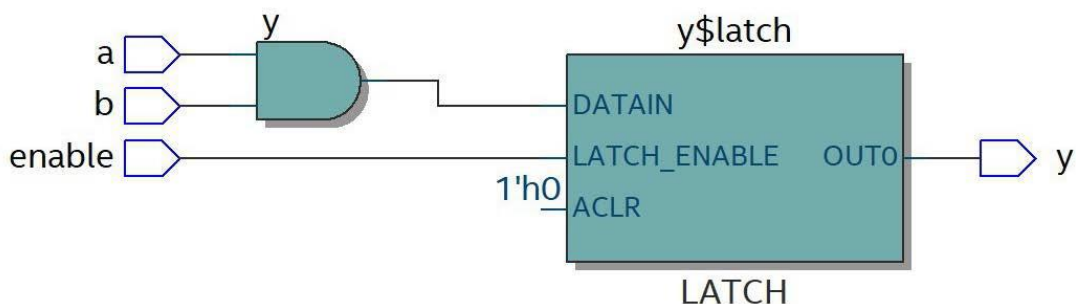


Рис.3.82. Функціональна схема latch

Протестуємо наш тригер на випадкових вхідних даних та переконаємось у його коректній роботі:

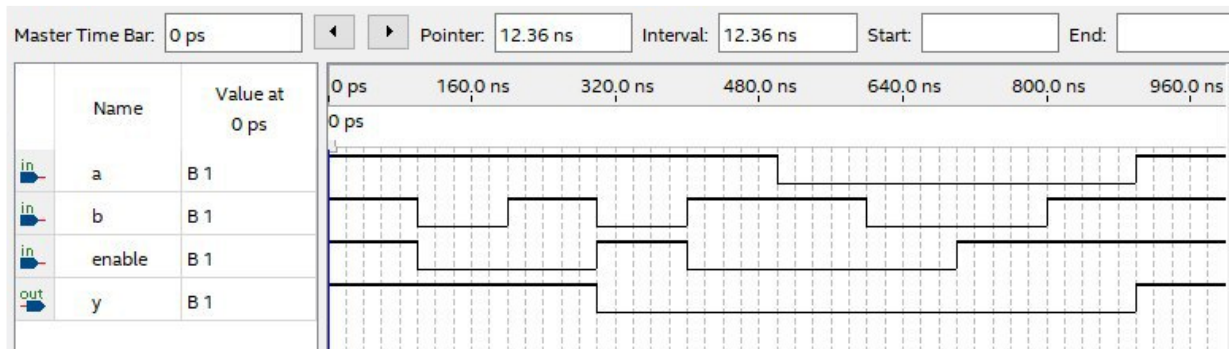


Рис.3.83. Симуляція роботи

Побудова 8-бітного регістру

Переважає більшість сучасних комерційних систем побудовано на регістрах, що використовують D-тригери які спрацьовують по передньому фронту тактового імпульсу. Сигнали, значення яким присвоєно в операторах `always` мови SystemVerilog, зберігають свій стан, поки не відбудеться подія зі списку чутливості оператора, що приводить до зміни їх значення. Тому код, використовує ці оператори з відповідними списками чутливості, може описувати послідовні схеми з пам'яттю. Наприклад, у тригера в списку чутливості є тільки сигнал `clk`, і тому тригер зберігає старе значення `q` до наступного переднього фронту `clk`, навіть якщо вхідний сигнал `d` змінився раніше. Оператор виконується, тільки коли трапляється подія, задана в списку чутливості. Оператором є `q = d`. Також тригер копіює `d` в `q` по передньому фронту тактового сигналу тільки коли вхідний сигнал `ena == 1`, а в решту часу значення `q` залишається незмінним.

Створимо 8-бітний регістр на мові SystemVerilog і отримаємо такий код:

```

1  module reg8
2  (output reg [7:0]q,input [7:0] d, input ena, clk);
3      always @(posedge clk)
4          if (ena)
5              q = d;
6  endmodule

```

Рис.3.84 Код програми reg

Після компіляції коду отримаємо таку функціональну схему:

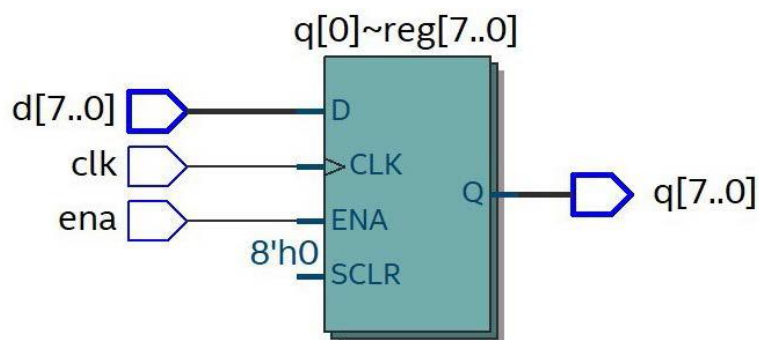


Рис.3.85. Функціональна схема reg

Протестуємо наш регістр на випадкових вхідних даних та переконаємось у його коректній роботі:

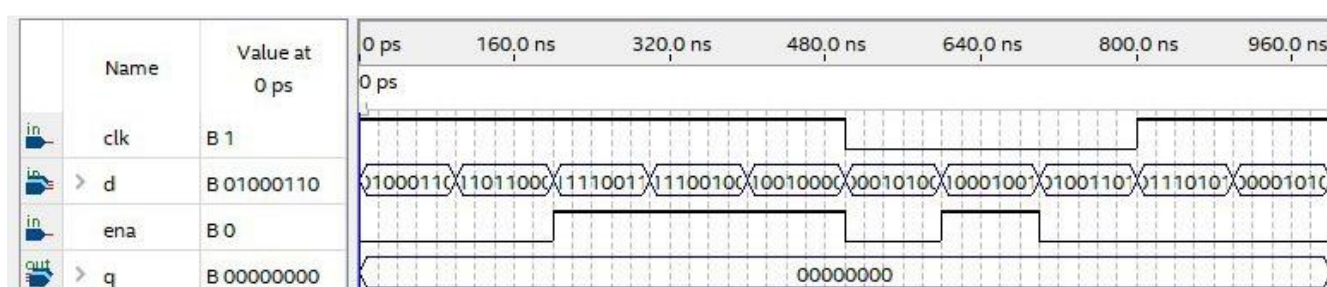


Рис.3.86. Перевірка симуляції reg

Мета роботи: розробити архітектурне рішення для створення JK-тригера та регістра на його основі.

Для вирішення поставленої задачі використовувалось середовище програмування «Quartus», а також апаратну частину Cyclone IV GL. Вхідними даними тригера є сигнали j, k, c, де c – це синхронізуючий сигнал. Вихідними даними тригера є q, q1. Регістр являє собою вісім тригерів з спільним синхронізуючим сигналом. Для перевірки правдивості результатів виконано перевірку за допомогою симуляції в середовищі «Quartus». У роботі використана мова опису апаратури System Verilog HDL.

У роботі показано код, таблиця істинності для тригера, RTL діаграми тригера та регістра, перевірка роботи цифрової схеми.

У звіті використовуються наступні опорні слова: тригер, jk-тригер, регістр, rtl-діаграма, таблиця істинності.

Напишемо код для побудови функціональної схеми JK-тригера:

```

module jk_trigger(input logic j, k ,c, output logic q, r);
always @(posedge c)
begin
    case({j, k})
        {1'b0,1'b0}:
        begin
            q <= q; r <= r;
        end
        {1'b0,1'b1}:
        begin
            q <= 1'b0; r <= 1'b1;
        end
        {1'b1,1'b0}:
        begin
            q <= 1'b1; r <= 1'b0;
        end
        {1'b1,1'b1}:
        begin
            q <= ~q; r <= ~r;
        end
    endcase
end
endmodule

```

Рис.3.87. Код програми jk_trigger

```

1  module trigger(input logic c, j, k, output logic q, inv_q);
2  always @(posedge c)
3  begin
4      case({j,k})
5          2'b00:
6              begin
7                  q = q;
8                  inv_q = inv_q;
9              end
10         2'b01:
11             begin
12                 q = 1'b0;
13                 inv_q = 1'b1;
14             end
15         2'b10:
16             begin
17                 q = 1'b1;
18                 inv_q = 1'b0;
19             end
20         2'b11:
21             begin
22                 q = ~q;
23                 inv_q = ~inv_q;
24             end
25         endcase
26     end
27 endmodule

```

Рис.3.88. Код програми trigger

Отримаємо функціональну схему JK-тригера

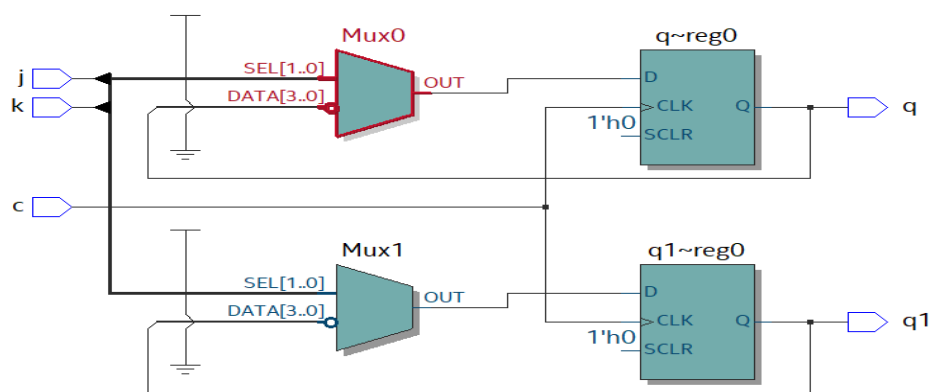


Рис.3.89. Схема JK-тригера

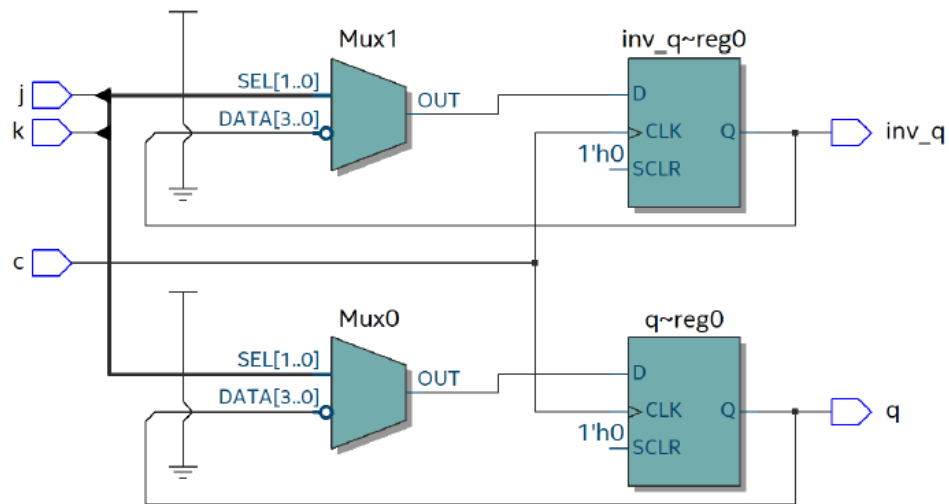


Рис.3.90. Схема JK-тригера

Вхідні дані: сигнали j, k, c – це синхронізуючий сигнал. Вихідні дані тригера: q, q1 – інверсний вихід.

Виконаємо перевірку роботи тригера

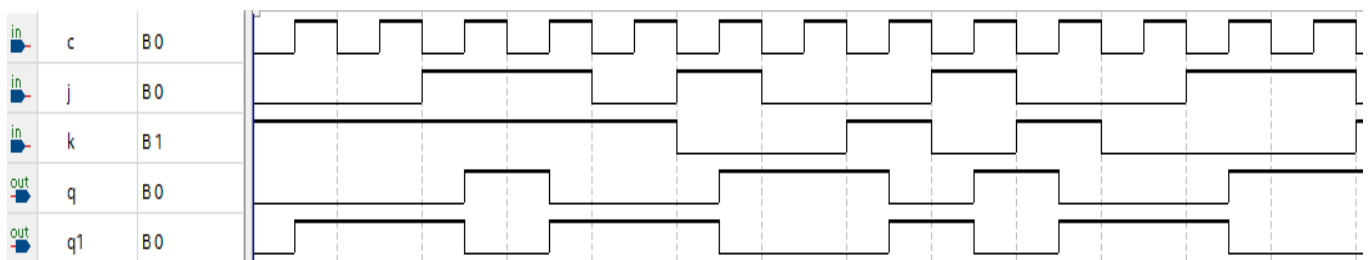


Рис.3.91. Симуляція JK-тригера

Напишемо код для побудови функціональної схеми JK-регістру:

```

module jk_reg(input logic [7:0] j, k, input logic c,
output logic [7:0] q);
  genvar i;
  generate
    for(i=0;i<8;i=i+1)
      begin: register
        jk_trigger trigger(j[i], k[i], c, q[i], 1'b0);
      end
    endgenerate
endmodule

```

Рис.3.92. Код програми jk_reg

```

module register( input logic[7:0] j, k, input logic c, output logic [7:0] state);
  generate
    genvar i;
    for( i = 0; i<8; i=i+1)
      begin: register
        trigger tr(c, j[i], k[i], state[i], 1'b0);
      end
    endgenerate
  endmodule

```

Рис.3.93. Код програми register

Отримаємо функціональну схему JK-регістру

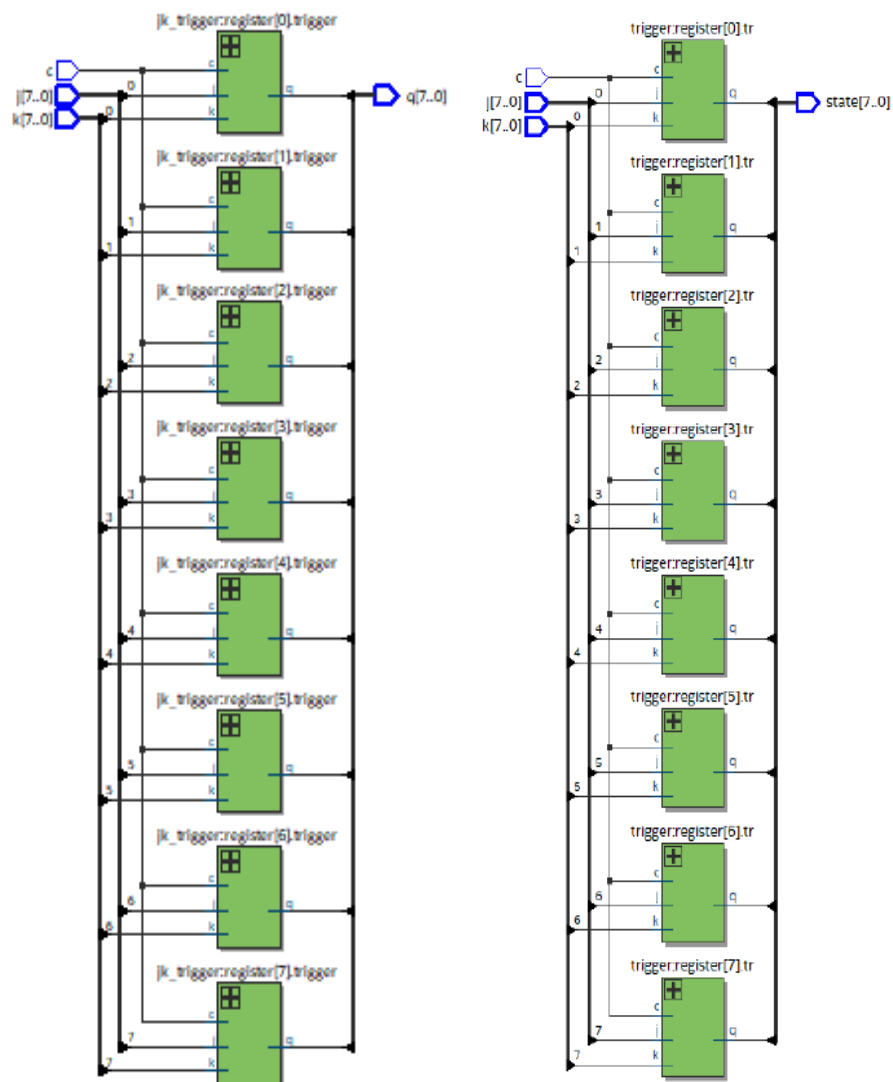


Рис.3.94. Функціональна схема регістру

Регістр – вісім тригерів з спільним синхронізуючим сигналом. Вхідні дані: восьмибітні шини [7:0] j, k, тактова частота c. Вихідні дані тригера: output logic [7:0] q.

Виконаємо перевірку роботи регістру

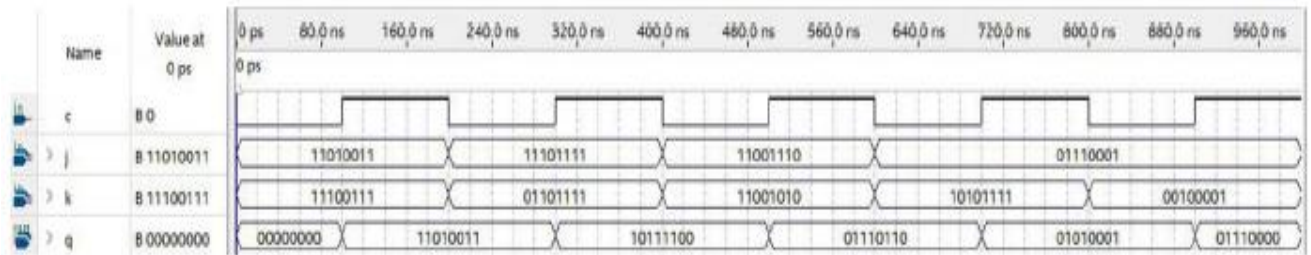


Рис.3.95. Симуляція JK-регістру

Реалізація у Quartus Prime 2

Постановка задачі та теоретичні відомості

Радіoeлектроніці існує безліч механізмів і деталей, які з допомогою найпростіших операцій дозволяють створювати складні машини. Правда, для такої мети їх потрібно дуже багато. І одним з найважливіших механізмів подібного призначення є JK-тригери. Вони дозволяють забезпечити машинну логіку для виконання найпростіших логічних операцій. Як це здійснюється? Як необхідно підключати JK-тригер? Як виглядає таблиця істинності? На ці та інші питання можна буде знайти відповіді в рамках статті.

JK тригер – це тригер, який у разі отримання на свої обидва входи логічної одиниці змінює стан свого виходу на протилежне значення. Одна з відмінностей від інших подібних приладів – відсутність заборонених станів, які можуть бути на основних входах. Як виглядає JK-тригер? Схема зображення може бути представлена з різною деталізацією, а також залежно від доповнень, які були додані людиною. Як бачите, в статті присутні різні зображення пристрою. Також, використовуючи базу JK-тригера, можна створити D-або T-модель. Як ви зможете переконатися, переглянувши таблицю істинності, цей механізм в інверсний стан переходить завжди, коли на обидва входи здійснюється подача логічної одиниці.

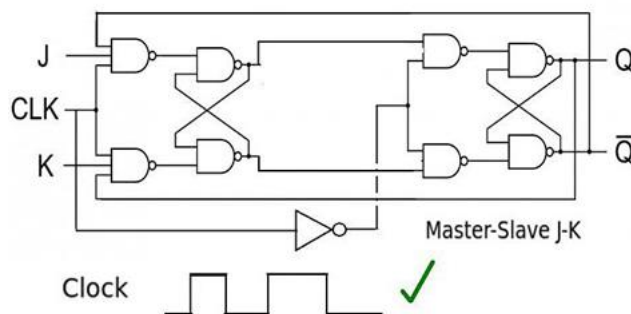


Рис.3.96. Функціональна схема

Таблиця істинності для JK trigger:

Таблиця 3.3

J	K	C	Q(t)	Q(t+1)	Пояснення
нуль	x	нуль	нуль	нуль	Зберігається інформація
нуль	x	нуль	одиниця	одиниця	
нуль	нуль	одиниця	нуль	нуль	Зберігається інформація
нуль	нуль	одиниця	одиниця	одиниця	
одиниця	нуль	одиниця	нуль	одиниця	Встановлена логічна одиниця, вхід J дорівнює одиниці
одиниця	нуль	одиниця	одиниця	одиниця	
нуль	одиниця	одиниця	нуль	нуль	Встановлюється логічний нуль, при цьому K дорівнює одиниці
нуль	одиниця	одиниця	одиниця	нуль	
одиниця	одиниця	одиниця	нуль	одиниця	лічильний режим тригера K=J=1

В таблиці 3.3 наведені дані роботи тригеру.

Реалізація JK Trigger

Спочатку створимо пустий проект та пустий файл Verilog. В ньому написали наступний код (написання модулів для тригерів, одразу для більшості типів):


```

1  module JK_ff ( input wire J,
2                input wire K,
3                input wire clk,
4                input wire rst,
5                output reg Q);
6
7      localparam HOLD = 2'b0,
8                  SET = 2'b10,
9                  RESET = 2'b01,
10                 TOGGLE = 2'b11;
11
12  always @(posedge clk) begin
13      if(rst) begin
14          Q <= 1'b0;
15      end
16      else begin
17          case({J,K})
18              HOLD : begin
19                  Q <= Q;
20              end
21              RESET : begin
22                  Q <= 1'b0;
23              end
24              SET : begin
25                  Q <= 1'b1;
26              end
27              TOGGLE : begin
28                  Q <= ~Q;
29              end
30          endcase
31      end
32  end

```

```

35 module SR_ff ( input wire S,
36                input wire R,
37                input wire clk,
38                input wire rst,
39                output reg Q);
40
41     localparam HOLD = 2'b0,
42                 SET = 2'b10,
43                 RESET = 2'b01;
44
45  always @(posedge clk) begin
46      if(rst) begin
47          Q <= 1'b0;
48      end
49      else begin
50          case({J,K})
51              HOLD : begin
52                  Q <= Q;
53              end
54              RESET : begin
55                  Q <= 1'b0;
56              end
57              SET : begin
58                  Q <= 1'b1;
59              end
60              default: begin
61                  Q <= 1'bx;
62              end
63          endcase
64      end

```

```

68 module D_ff ( input wire D,
69               input clk,
70               input rst,
71               output reg Q);
72
73  always @(posedge clk) begin
74      if(rst) begin
75          Q <= 1'b0;
76      end
77      else begin
78          Q <= D;
79      end
80  end
81  endmodule

```

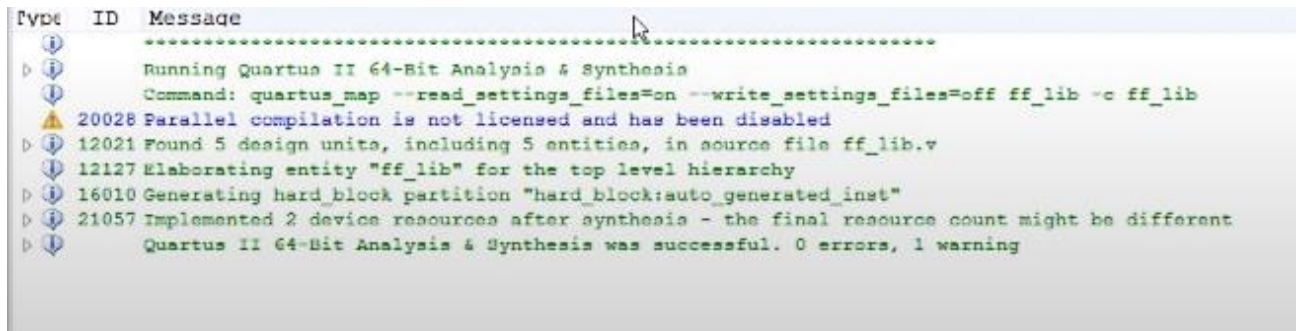
```

83 module T_ff ( input wire T,
84               input wire clk,
85               input wire rst,
86               output reg Q);
87
88  always @(posedge clk) begin
89      if(rst) begin
90          Q <= 0;
91      end
92      else begin
93          case(T)
94              1'b0: begin
95                  Q <= Q;
96              end
97              1'b1: begin
98                  Q <= ~Q;
99              end
100          endcase
101      end
102  end
103  endmodule

```

Рис.3.97. Код програми

Компілюємо:



Тепер напишемо TestVectors для нашого тригера, щоб перевірити на правильність, для цього знову написали наступний код:

```
1 module JK_ff_tb();
2
3     reg J_in, K_in, CLK, rst_in;
4     wire Q_out;
5
6     reg [2:0] testvectors [10:0];
7
8     JK_ff JK_ff_dut( J_in,
9                     K_in,
10                    CLK,
11                    rst_in,
12                    Q_out);
13
14     initial begin
15         $readmemb("", testvectors);
16     end
17
18
19 endmodule
```

Рис. 3.98. Код програми тест

Впишемо наш файл для початку опрацювання:

```
reg [2:0] testvectors [10:0];
integer i;

JK_ff JK_ff_dut( J_in,
                 K_in,
                 CLK,
                 rst_in,
                 Q_out);

initial begin
    $readmemb("C:/Users/Erdem/Desktop/all-projects-ws/verilog-quartus-tutorials/ff_lib/JK_ff_testvectors.txt", testvectors);
    i = 0;
end

always begin
    CLK = 1'b0;
    #5;
    CLK = 1'b1;
    #5;
end

always @(posedge CLK) begin
    {J_in, K_in, rst_in} = testvectors[i];
end
```

Рис.3.99. Код програми перевірка

Для перевірки на правильність напишемо послідовність бітів для опрацювання

```
0_0_1
0_0_0
1_0_0
0_1_0
1_1_0
```

*Не будемо знову вставляти зображення але у рядку «reg[2;0] testvectors [4;0] замість 10 встановили значення 4. Та до блоку always додали i+1.

Виконали перевірку на правильність компіляції:

```
.....
Running Quartus II 64-Bit Analysis & Synthesis
Command: quartus_map --read_settings_files=on --write_settings_files=off ff_lib -c ff_lib
20028 Parallel compilation is not licensed and has been disabled
12021 Found 5 design units, including 5 entities, in source file ff_lib.v
12021 Found 1 design units, including 1 entities, in source file jk_ff_tb.v
12127 Elaborating entity "ff_lib" for the top level hierarchy
16010 Generating hard block partition "hard_block:auto_generated_inst"
21057 Implemented 2 device resources after synthesis - the final resource count might be different
Quartus II 64-Bit Analysis & Synthesis was successful. 0 errors, 1 warning
```

Все вірно тому тепер перейдемо до ModelSim. Відклики збереженні нами файли, та запустили програму. Встановили наші сигнали та запустили програму. Отримали наступне:

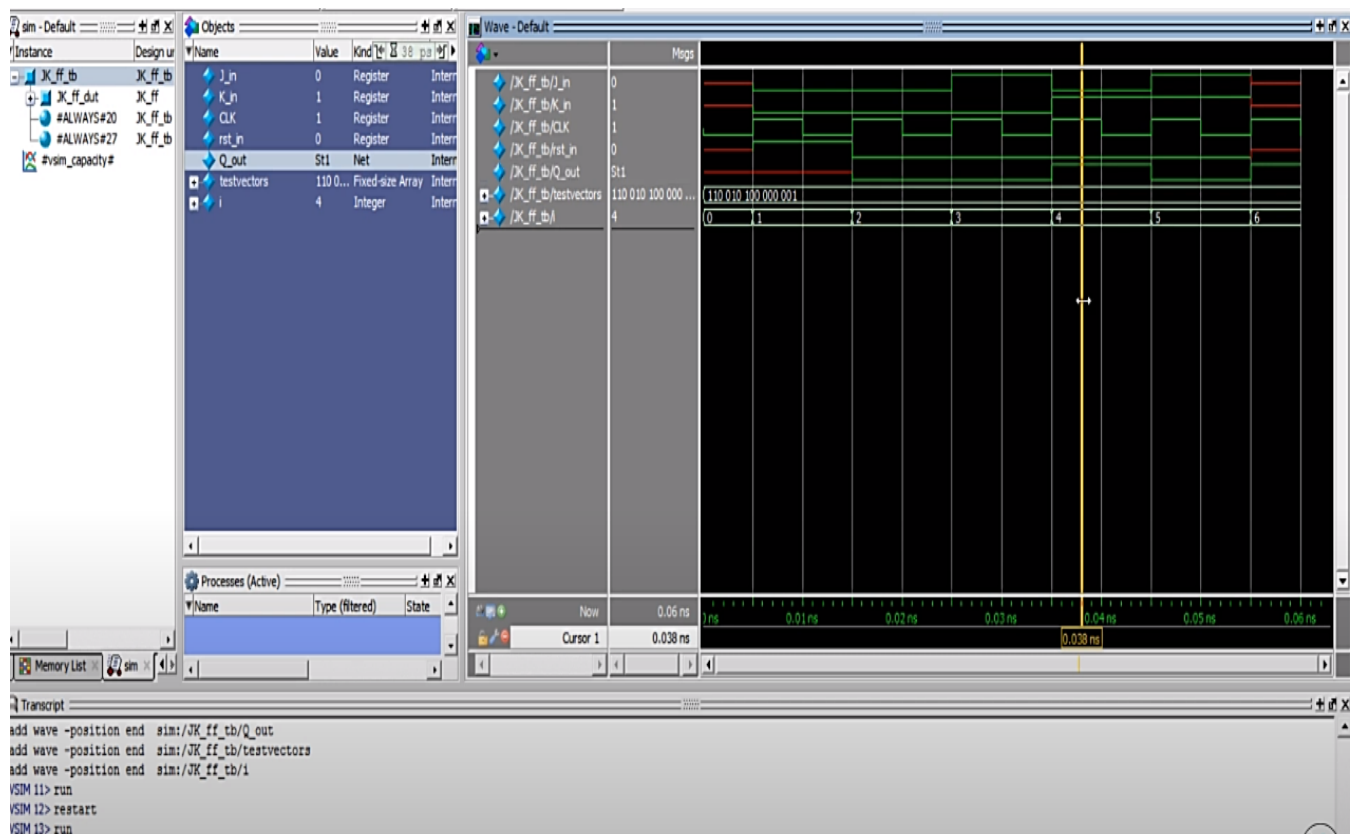


Рис.3.100. Симуляція коду

Як ми можемо бачити, програма виконалась успішно, адже послідовність відтворилась при руху лінії часу (жовтим кольором) побачили зміну бітів відповідно до поставлених умов у лівому сірому вікні.

Висновки

Виконуючи лабораторну роботу, було реалізовано тригер, JK-тригер та регістр на його основі. Даний тригер працює як звичайний RS-тригер, але із запобіганням помилок при поданні двох одиниць J і K. Під час виконання роботи було використано базові можливості середовища Quartus та мови для побудови функціональних схем System Verilog HDL.

Контрольні запитання

1. Поясніть і опишіть структуру коду тригера в даній роботі.
2. Поясніть і опишіть структуру коду JK-тригера в даній роботі.
3. Поясніть і опишіть структуру коду регістра в даній роботі.
4. Що собою представляє схема тригера RTL-viewer?
5. Що собою представляє схема JK-тригера RTL-viewer?
6. Що собою представляє схема регістра RTL-viewer?
7. Поясніть роботу симуляції тригера.
8. Поясніть роботу симуляції JK-тригера.
9. Поясніть роботу симуляції регістра.
10. Яким чином перевірити правильність результатів роботи тригера, JK-тригера, регістра.

3.7. Лабораторна робота № 7

Тема: JK-засувка, ROM, RAM

Постановка задачі:

Об'єкт дослідження: створення JK-засувка, ROM, RAM.

Мета роботи: спроектувати комбінаційні схеми для роботи JK-засувка, ROM, RAM.

Методи дослідження та апаратура: формалізація JK-засувка, ROM, RAM за допомогою середовища проектування Quartus та мови опису апаратури SystemVerilog.

Результати: реалізовано JK-засувка, ROM, RAM для виконання базових математичних операцій.

Завданням даної лабораторної роботи є розроблення рішення для виконання JK-засувка, ROM, RAM.

Розробка має бути виконана у вигляді вентильної схеми і програмного коду Verilog HDL.

Завдання на лабораторну роботу:

Розробити архітектурне рішення JK-засувка, ROM, RAM. Також розробити схему для збереження результату виконання операції. Реалізувати можливість доступу до пам'яті за відповідною адресою. Варіанти управління потоками даних повинні задаватися у вигляді коду операції. Таким чином, код операції складатиметься з двох біт вибору арифметичної операції та трьох біт вибору режиму доступу до пам'яті.

Розробка має бути виконана у вигляді вентильної схеми і програмного коду Verilog HDL.

Працездатність архітектури і розробки повинні бути показані у вигляді процесу роботи логічної вентильної схеми.

Опис результатів, а також методів, підходів і засобів розробки повинні бути подані у вигляді звіту. Оформлення і вмісту звіту мають відповідати ДСТУ3008-95.

Розробка може проводитися у таких середовищах: MatLab SImulink; Multisim (EWB); ISIS Proteus; Quartus II.

Склад звіту лабораторної роботи

В звіті повинно бути відображено:

1. Тема роботи.
2. Мета роботи.
3. Постановка задачі.
4. Виконання завдання з відповідними скріншотами.
5. Відповіді на контрольні запитання.
6. Висновок.

JK-засувка, ROM, RAM. Основні компоненти

Реалізація у Quartus Prime 1

Мета роботи: розробити архітектурне рішення для створення JK-засувки.

Для вирішення поставленої задачі використовувалось середовище програмування «Quartus», а також апаратну частину Cyclone IV GL. Вхідними даними засувки є сигнали j, k. Вихідними даними засувки є q, q1. Для перевірки правдивості результатів виконано перевірку за допомогою симуляції в середовищі «Quartus». У роботі використана мова опису апаратури System Verilog HDL.

Напишемо код для побудови функціональної схеми JK-засувки:

```
module jk_latch(input logic j, k, output logic q, q1);  
  always @(j, k)  
  begin  
    case({j, k})  
      2'b00: begin q = q; q1 = q1; end  
      2'b01: begin q = 1'b0; q1 = 1'b1; end  
      2'b10: begin q = 1'b1; q1 = 1'b0; end  
      2'b11: begin q = ~q; q1 = ~q1; end  
    endcase  
  end  
endmodule
```

Рис.3.101. Код програми jk_latch

Після компіляції коду отримаємо таку функціональну схему:

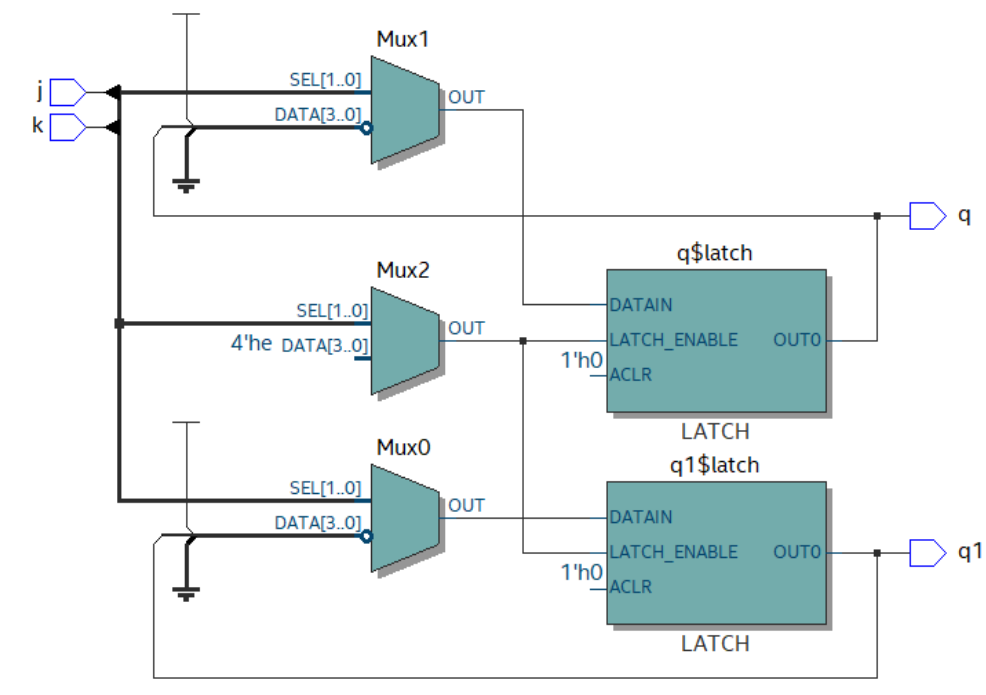


Рис.3.102. Функціональна схема JK-засувки

Вхідні дані: сигнали j, k. Вихідні дані тригера: q, q1 – інверсний вихід. Засувка – асинхронний тригер. Для перевірки правильності побудови функціональної схеми побудуємо таблицю істинності і порівняємо її з отриманою симуляцією.

JK		00	01	11	10
Q	0	0	0	1	1
	1	1	0	0	1

Рис.3.103. Таблиця істинності

Виконаємо перевірку роботи JK-засувки:

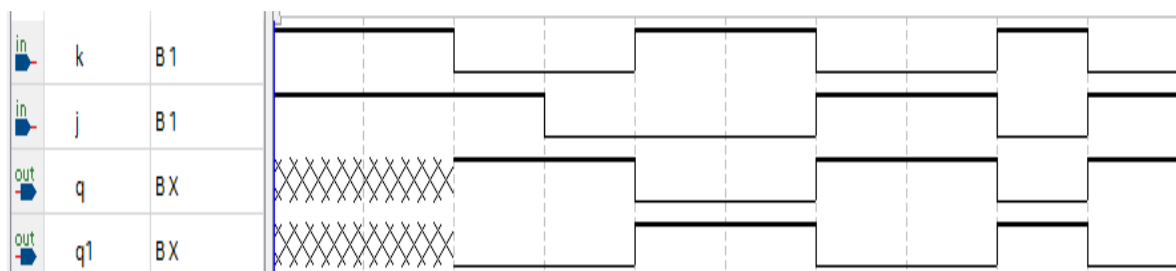


Рис.3.104. Симуляція JK-засувки

Було реалізовано JK-засувку. Ця засувка працює як звичайна RS-засувка, але зі запобіганням помилок при поданні двох одиниць J і K. Під час виконання роботи ми використали базові можливості середовища Quartus та мови для побудови функціональних схем System Verilog HDL.

Побудова постійного запам'ятовувального пристрою (ROM)

Опишемо ПЗП розмірністю 4-слова \times 3-біт. Вміст ПЗП задається в операторі case. ПЗП настільки малий, що може бути синтезований у вигляді набору логічних елементів, а не матриці.

Створимо ROM на мові SystemVerilog і отримаємо такий код:

```

1  module rom(input logic [1:0] adr, output logic [2:0] dout);
2      always_comb case(adr)
3          2'b00: dout <= 3'b011;
4          2'b01: dout <= 3'b110;
5          2'b10: dout <= 3'b100;
6          2'b11: dout <= 3'b010;
7      endcase
8  endmodule
9

```

Рис.3.105. Код програми ROM

Після компіляції коду отримаємо таку функціональну схему:

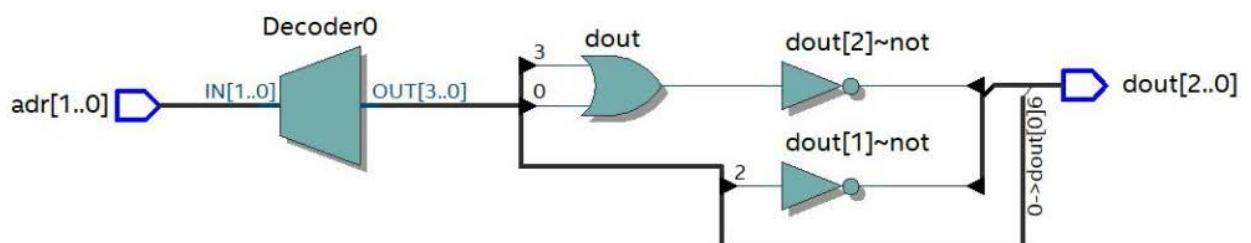


Рис.3.106. Функціональна схема ROM

Протестуємо наш ROM на випадкових вхідних даних та переконаємось у його коректній роботі:

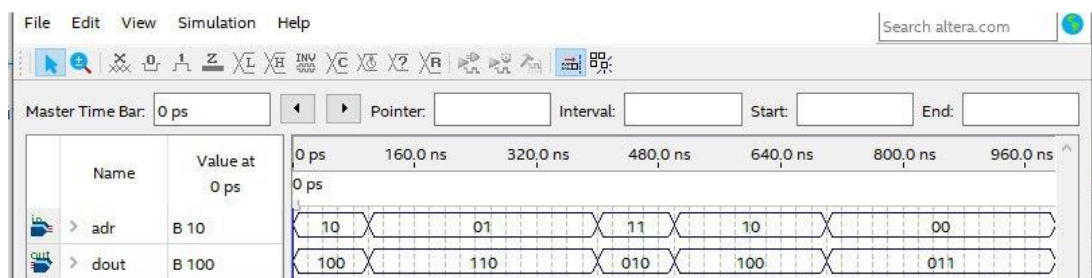


Рис.3.107. Симуляція ROM

Побудова оперативного запам'ятовувального пристрою (RAM)

Опишемо ОЗП розмірністю $2N$ -слів \times M -біт. У цього ОЗП є синхронний вхід дозволу запису. Іншими словами, запис в пам'ять відбувається по передньому фронту тактового імпульсу, якщо сигнал дозволу запису (write enable) we знаходиться в активному стані. Читання відбувається негайно. Безпосередньо після включення живлення вміст ОЗП непередбачуваний.

Створимо RAM на мові SystemVerilog і отримаємо такий код:

```
1 module ram #(parameter N=6, M=32)
2   (input logic clk, input logic we, input logic [N-1:0] adr, input logic [M-1:0] din,
3    output logic [M-1:0] dout);
4
5   logic [M-1:0] mem[2**N-1:0];
6   always_ff @(posedge clk)
7     if (we) mem [adr] <= din;
8   assign dout = mem[adr];
9 endmodule
10
```

Рис.3.108. Код програми RAM

Після компіляції коду отримаємо таку функціональну схему:

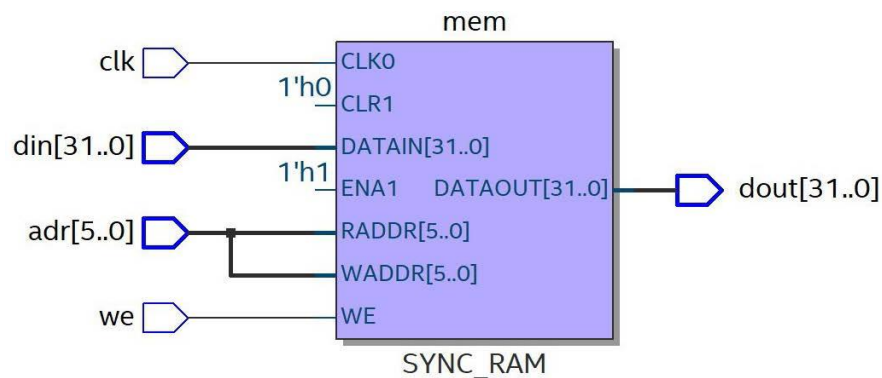


Рис.3.109. Функціональна схема RAM

Протестуємо наш RAM на випадкових вхідних даних та переконаємось у його коректній роботі:

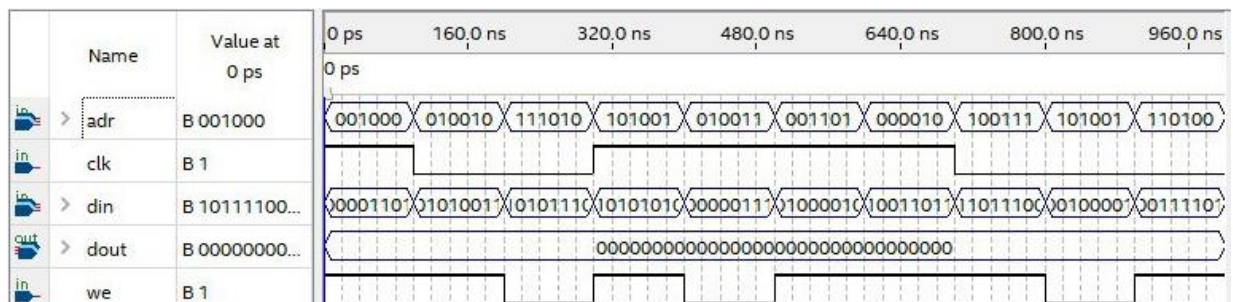


Рис.3.110. Симуляція RAM

Таким чином було розроблено архітектурне рішення для реалізації пам'яті (створено постійний запам'ятовувальний пристрій та оперативний запам'ятовувальний пристрій).

Висновки

Метою даної роботи було створення пам'яті та JK-засувки. В процесі виконання було реалізовано просту пам'ять(1біт) або JK-засувку. JK засувка використовується для зберігання двійкових чисел, та має сигнали управління J і K, що дозволяють встановлювати значення бітів, залишати їх без зміни чи інвертувати їх. На відміну від JK-триггеру, JK-засувка є асинхронною. Під час виконання роботи ми використали базові можливості середовища Quartus та мови для побудови функціональних схем SystemVerilogHDL, а також покращили знання з секвенціальної логіки.

Контрольні запитання

1. Поясніть і опишіть структуру коду JK-засувки в даній роботі.
2. Поясніть і опишіть структуру коду ROM в даній роботі.
3. Поясніть і опишіть структуру коду RAM в даній роботі.
4. Що собою представляє схема JK-засувки RTL-viewer?
5. Що собою представляє схема ROM RTL-viewer?
6. Що собою представляє схема RAM RTL-viewer?
7. Поясніть роботу симуляції JK-засувки.
8. Поясніть роботу симуляції ROM.
9. Поясніть роботу симуляції RAM.
10. Яким чином перевірити правильність результатів роботи JK-засувки, ROM, RAM.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. System Verilog Tutorial [Електронний ресурс]. – Режим доступу: <http://www.asic-world.com/systemverilog/tutorial.html>.
2. Акчурин А.Д., Юсупов К.М. Программирование на языке Verilog: Учебное пособие. — Казань: Казанский федеральный университет, 2016. — 90 с.
3. SystemVerilog [Електронний ресурс]. – Режим доступу: <https://ru.wikipedia.org/wiki/SystemVerilog>.
4. Пошаговая инструкция: создаем проект Quartus II [Електронний ресурс]. – Режим доступу: <https://marsohod.org/11-blog/78-newproject>.
5. Харрис, Д. М. Цифровая схемотехника и архитектура компьютера / Дэвид М. Харрис, Сара Л. Харрис. - Нью-Йорк : Elsevier. inc : Изд-во Morgan Kaufman, 2013. - 1662 с.
6. Основы радиоэлектроники : учебное пособие / Е. И. Манаев. - 2-е изд., перераб. и доп. - М. : Радио и связь, 1985. - 504 с.
7. Язык описания аппаратуры Verilog HDL [Електронний ресурс]. – Режим доступу: <https://marsohod.org/verilog>.
8. Насыров И.А. Конспекты лекций по цифровой электронике. Учебное пособие. - Казань: КГУ, 2006. - 98 с.
9. Хамахер, К. Организация ЭВМ [Текст] : учебное пособие / К. Хамахер, З. Вранешич, С. Заки ; перевод О. Здир. - 5-е изд. - СПб. : Питер ; Киев : BHV, 2003. - 848 с. : ил. - ("Классика Computer Science"). - Алф. указ.: с. 833-845. - 4000 экз.. - ISBN 5-8046-0162-8 (в пер.). - ISBN 966-552-122-5.
10. Дэвид М. Харрис, Сара Л. Харрис. Цифровая схемотехника и архитектура компьютера. / пер. с англ. Imagination Technologies. – М.: ДМК Пресс, 2017. – 792 с.: ил.