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

Мова опису апаратури Verilog

КОМП'ЮТЕРНИЙ ПРАКТИКУМ

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для студентів,
які навчаються за спеціальністю 172 «Телекомунікації та радіотехніка»,
спеціалізацією «Інформаційно-обчислювальні засоби електронних систем»

Київ
КПІ ім. Ігоря Сікорського
2018

Мова опису апаратури Verilog: [Електронний ресурс]: навч. посіб. для студ. спеціальності 172 «Телекомунікації та радіотехніка», спеціалізації «Інформаційно-обчислювальні засоби електронних систем» / КПІ ім. Ігоря Сікорського ; уклад.: О. І. Антонюк, Д. Ю. Лебедев. – Електронні текстові данні (1 файл, 2,657Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2018. – 59 с.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 05.2018 від 21.05.2018 р.)
за поданням Вченої ради факультету електроніки (протокол № 4 від 23.04.2018 р.)

Мова опису апаратури Verilog
КОМП'ЮТЕРНИЙ ПРАКТИКУМ

Укладачі: *Антонюк Олександр Ігорович*
Лебедев Денис Юрійович, канд. техн. наук, доц.

Відповідальний редактор *Мірошніченко А.П.*, к.т.н., доц.

Рецензенти: *Трапезон К. О.*, доцент, доцент кафедри Звукотехніки і реєстрації інформації факультету електроніки "КПІ ім. І.Сікорського"

Зміст

Вступ

Видання містить цикл лабораторних робіт з моделювання цифрових пристроїв стосовно галузі радіоелектроніки.

Матеріали надано відповідно до програми навчальної дисципліни «Системи автоматизованого проектування радіоелектронної апаратури - 1». В посібнику розглядаються наступні питання: принципи роботи САПР, призначеної для моделювання цифрових пристроїв, базові конструкції мови опису апаратури Verilog та способи опису різноманітних цифрових пристроїв. Кожна лабораторна робота містить теоретичні відомості, приклад реалізації цифрового пристрою та перелік завдань для самостійної роботи.

Перша робота дає можливість познайомитися зі спеціалізованим програмним середовищем для моделювання поведінки цифрових пристроїв за допомогою мови опису апаратури. Надаються необхідні відомості стосовно порядку створення та тестування проектів.

Наступні чотири роботи дають можливість познайомитися з базовими конструкціями мови Verilog, та способами опису простих цифрових пристроїв – комбінаційного та послідовного типів.

Останні три роботи навчають принципам опису складних пристроїв та способам їх перевірки з використанням спеціальних файлів для тестування (test-bench file).

Лабораторна робота № 1
«Знайомство з середовищем моделювання ModelSim»

1. Теоретичні відомості

Середовище моделювання ModelSim призначено для перевірки працездатності проекту, описаного однією з мов опису апаратури (HDL). Воно поєднує в собі засоби створення проекту, запису і редагування похідних файлів проекту, компілятор, що моделює програму та засоби візуалізації результатів моделювання (графічний редактор та ін.). ModelSim підтримує роботу з тестовими файлами на HDL (test-bench) або на мові Tcl (командна мова для створення керуючих файлів).

2. Порядок виконання роботи

1. Запустіть програму ModelSim за допомогою команди Пуск ⇒ Усі програми ⇒ Modelsim SE 6.4 ⇒ Modelsim.

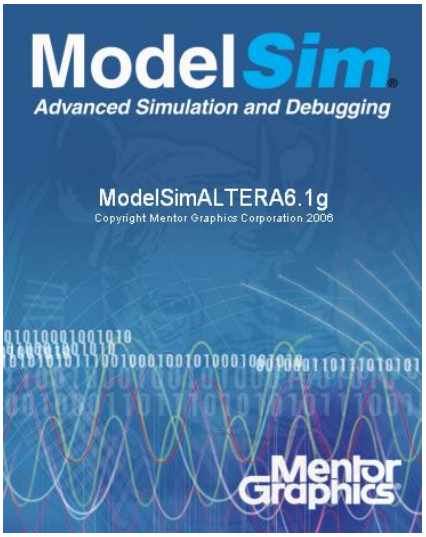


Рис. 1. Початок роботи програми Modelsim.

2. Закрийте вікно IMPORTANT Information за допомогою кнопки Close. Відкриється основне робоче вікно середовища моделювання:

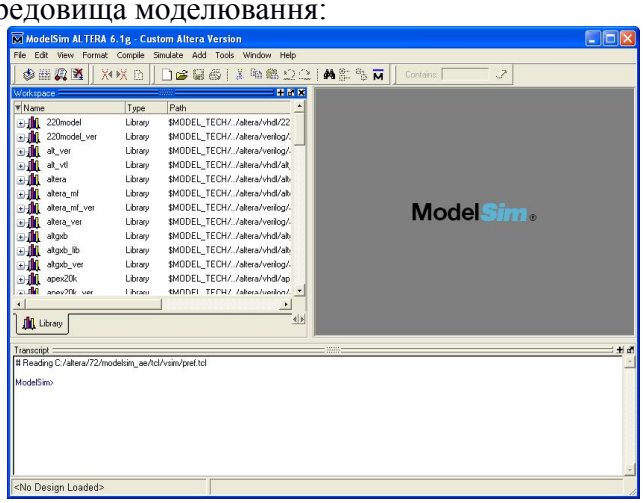


Рис. 2. Основне робоче вікно

3. Створіть новий проект. Для цього в меню File виберіть команду New⇒ **Project ..** Відкриється діалогове вікно Create Project:

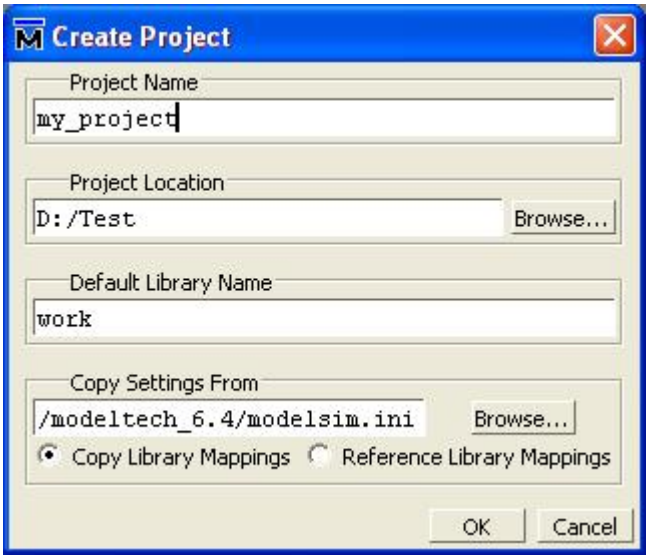


Рис. 3. Діалогове вікно Create Project

Вкажіть в ньому ім'я Вашого проекту (наприклад - my_project), робочу директорію проекту. Ім'я поточної бібліотеки проекту залиште за замовченням (work). Натисніть OK.

4. У наступному діалоговому вікні:

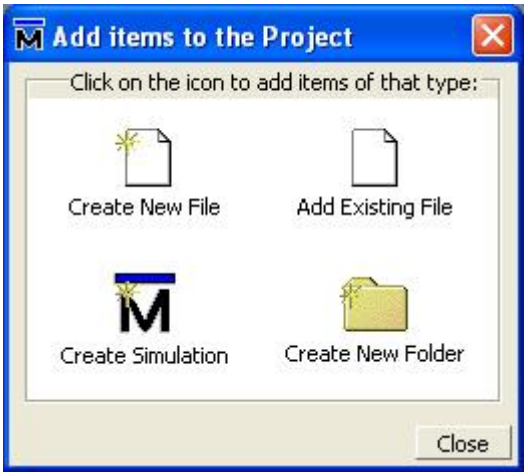


Рис. 4. Вікно налаштування.

виберіть команду додавання існуючих похідних файлів до проекту - Add Existing File.

5. У діалоговому вікні Add file to Project вкажіть файл (mult_acc.v) з дерикторії похідних даних до лабораторної роботи. Натисніть кнопку Відкрити. Підтвердіть вибір командою OK. Закрийте вікно Add items to the Project.

6. У вікні робочої області проекту (Workspace, закладка Project) з'явиться обраний файл. Подвійне натискання лівої кнопки миші на цьому файлі дозволяє відкрити текстовий редактор для перегляду і редагування похідного файлу:

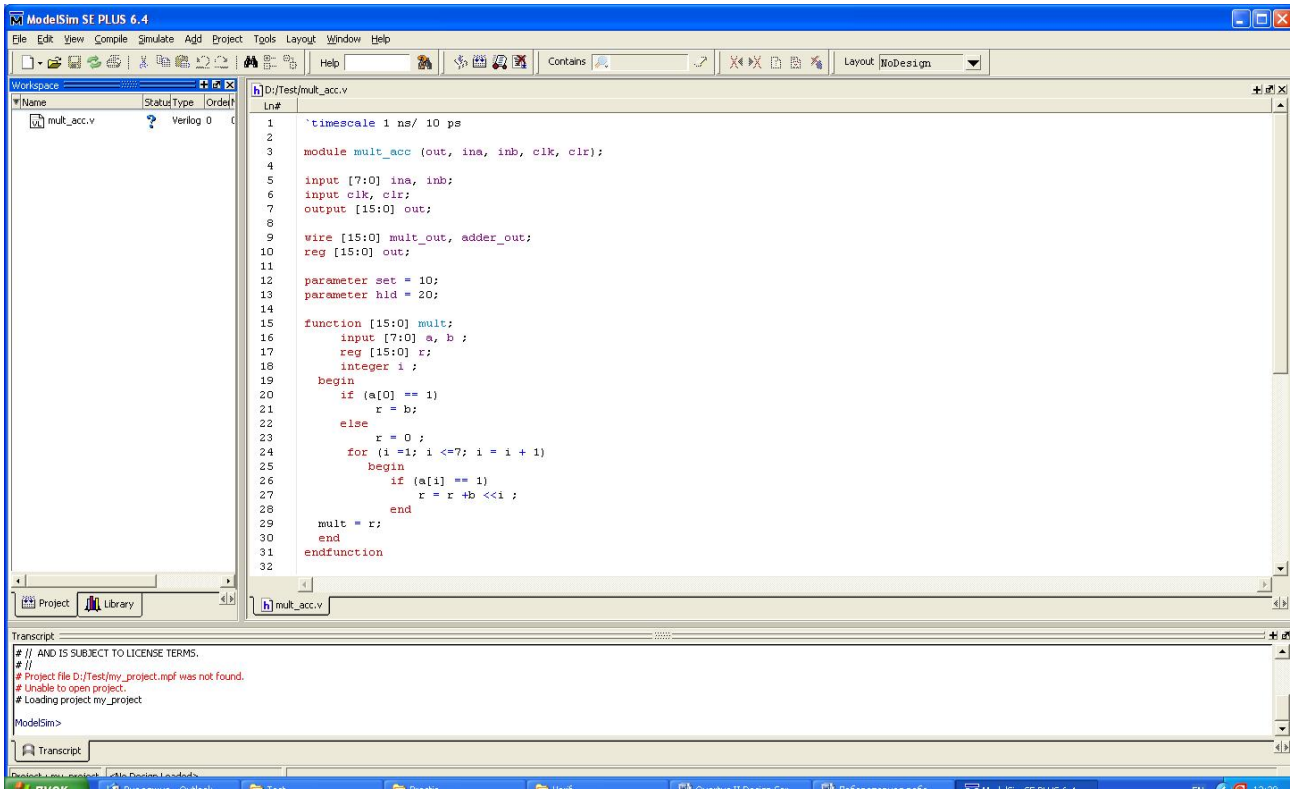


Рис. 5. Текстовий редактор

Файл mult_acc.v описує роботу пристрою MAC - перемножувач двох восьмирозрядних чисел з подальшим накопиченням результату в акумуляторі (Постарайтеся розібратися з програмою). Зверніть увагу на значок статусу біля імені файлу.

7. Скомпілюйте проект. Для цього виберіть в меню Compile команду Compile All. Інформація про результат компіляції з'явитися у вікні повідомлень (Transcript). Якщо компіляція завершилася успішно, зміниться вид значка статусу біля імені файлу.

8. Тепер перейдіть в режим моделювання. Для цього в меню Simulate виберіть команду Start Simulation. У діалоговому вікні Start Simulation вкажіть файл верхнього рівня ієрархії для моделювання (файл mult_acc з робочої бібліотеки work).

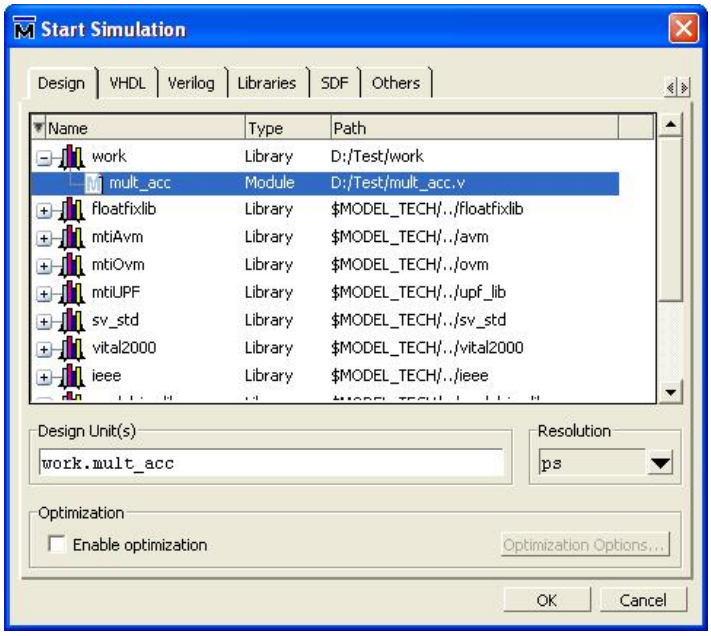


Рис. 6. Діалогове вікно Start Simulation

Вимкніть опцію Enable optimization. Вкажіть крок моделювання (Resolution), рівний ps (одна пікосекунда). Натисніть ОК.

9. Зовнішній вигляд середовища ModelSim змінився. Воно переключилося в режим моделювання. У вікні робочої області проекту (Workspace) з'явилася закладка sim (список доступних для моделювання об'єктів - похідний файл, та функції і процеси, що входять до його складу). Відкрилося вікно Objects - доступні для перегляду об'єкти (вхідні, вихідні та внутрішні сигнали).

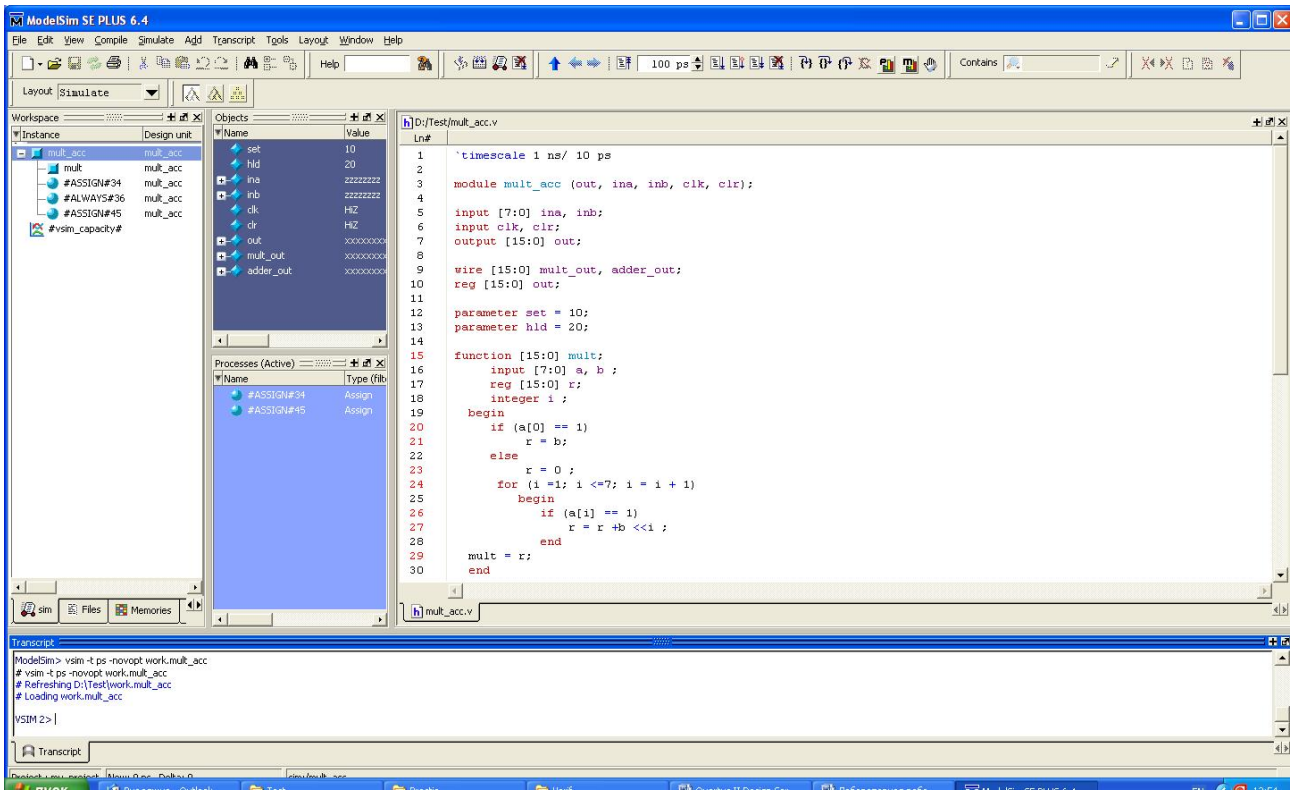


Рис. 7. Режим моделювання.

10. Зробіть вікно Object активним (натисніть лівою кнопкою миші на заголовку вікна). В меню Add виберіть команду To Wave⇒ **All items in region**. Дана команда дозволяє відкрити графічне вікно і додати в нього для перегляду всі сигнали, присутні в даному модулі.

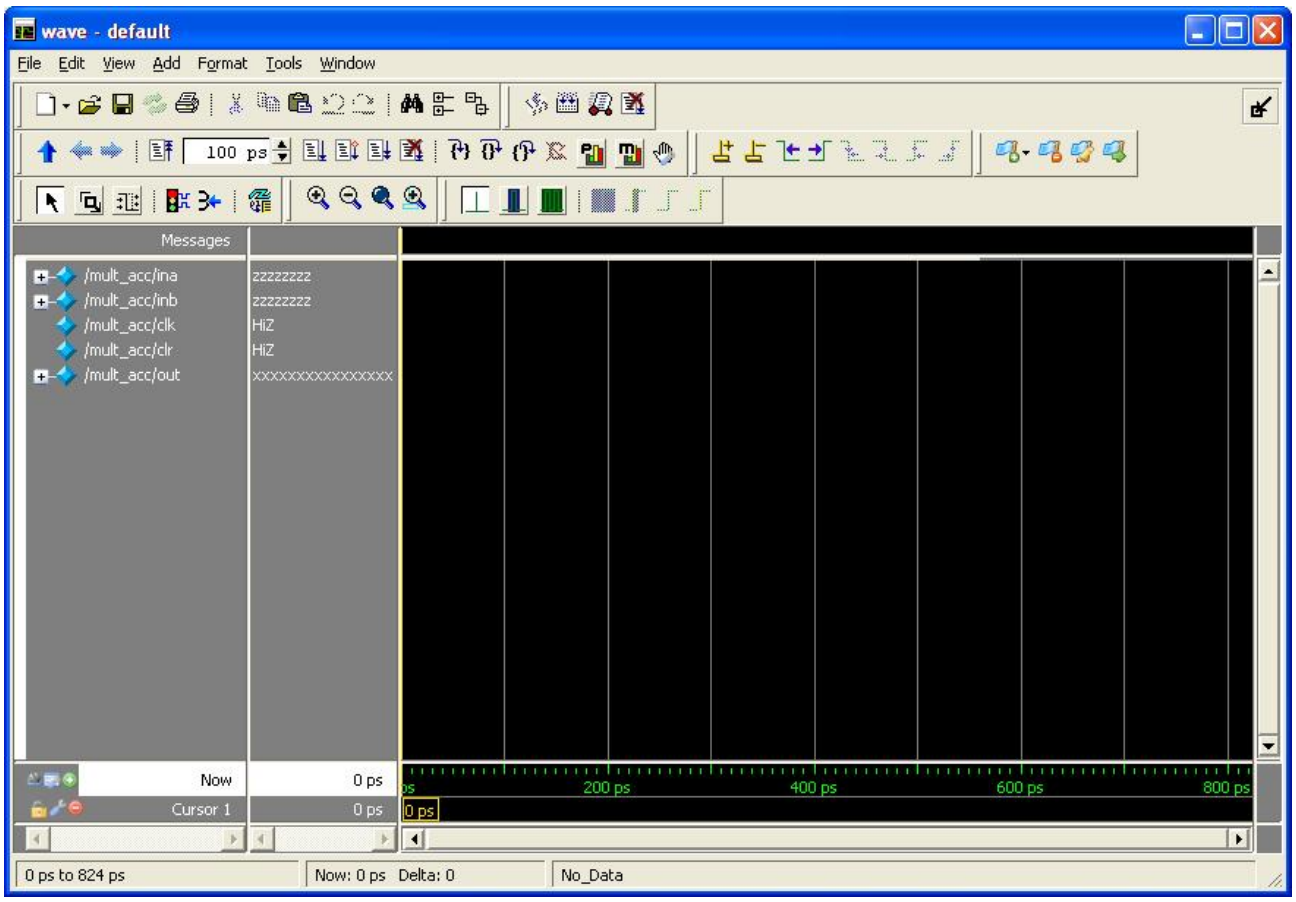


Рис. 8. Графічне вікно

11. Для того, щоб приступити до моделювання, необхідно створити файл з вхідними тестовими сигналами (test-bench). Для даної лабораторної роботи він уже створений - це файл Stim.do (мовою Tcl). Щоб подивитися його, перейдіть у вікні робочої області проекту (Workspace) до закладки Project. У меню File виберіть команду Open . У вікні вибору файлу вкажіть тип файлу - Macro Files (* .do, * .tcl) і вкажіть файл Stim.do. У вікні текстового редактора відкриється даний файл.

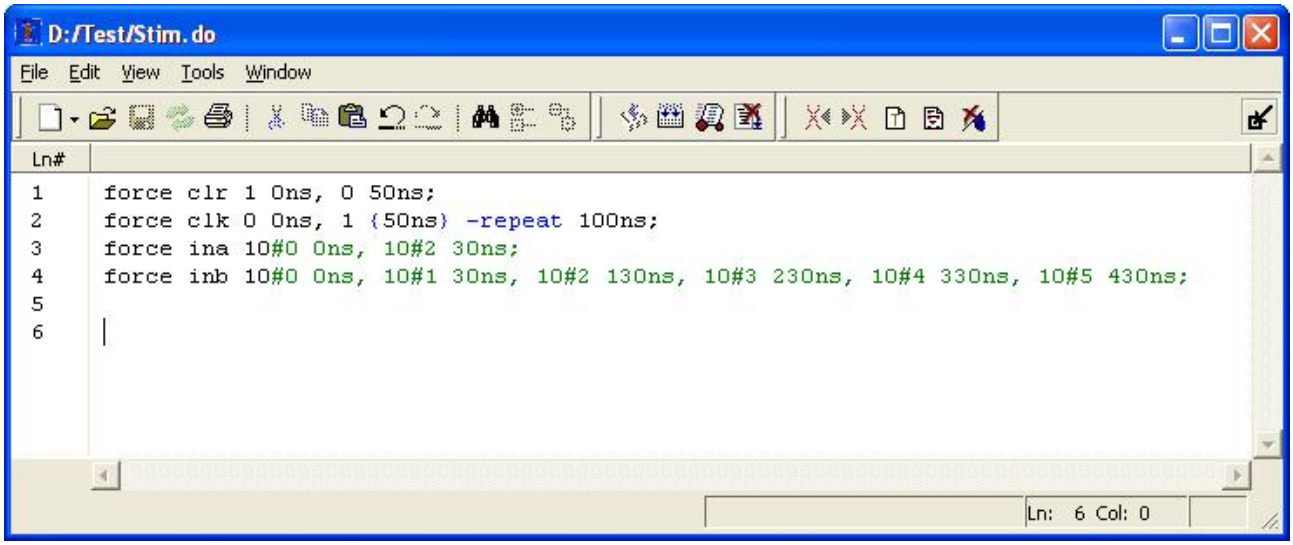


Рис. 9. Файл Stim.do

ПРИМІТКА Розглянемо докладніше формат команди формування вхідних впливів: **force clr 1 0ns, 0 50ns;** - дана команда говорить про те, що сигналу clr необхідно присвоїти

значення логічної «1» на 0 нанасекунді, а потім - логічного «0» на 50 наносекунді. Останнє значення буде зберігатися до кінця часу моделювання.
force clk 0 0ns, 1 {50ns} -repeat 100ns; - дана команда говорить про те, що сигналу clk необхідно присвоїти значення логічного «0» на 0 нанасекунді, а потім - логічної «1» на 50 наносекунді. Дана послідовність буде періодично повторюватися з періодом 100 наносекунд до кінця часу моделювання. Зверніть увагу - значення часу перед опцією - repeat обов'язково вказується в фігурних дужках.
force ina 10 # 0 0ns, 10 # 2 30ns; - дана команда присвоює наступні значення вхідному сигналу ina: на 0 наносекунді - 0 (у десятковому форматі), на 30 наносекунді - 2 (в десятковому форматі). Останнє значення буде зберігатися до кінця часу моделювання. Перша цифра перед знаком # вказує на формат числа (2 - двійковий формат, 10 - десятковий формат, 16 - шістнадцятковий формат), а цифра за знаком - його значення.

12. Для підключення тестового файлу до проекту, в меню Tools виберіть команду TCL ⇒ **Execute Macro ...** і вкажіть файл Stim.do.

13. Запуск програми на моделювання здійснюється з меню Simulate командою Run ⇒ **Run -All**. Зупинити моделювання потрібно командою Breake з меню Simulate. Результат моделювання буде відображатися в графічному вікні наступним чином:

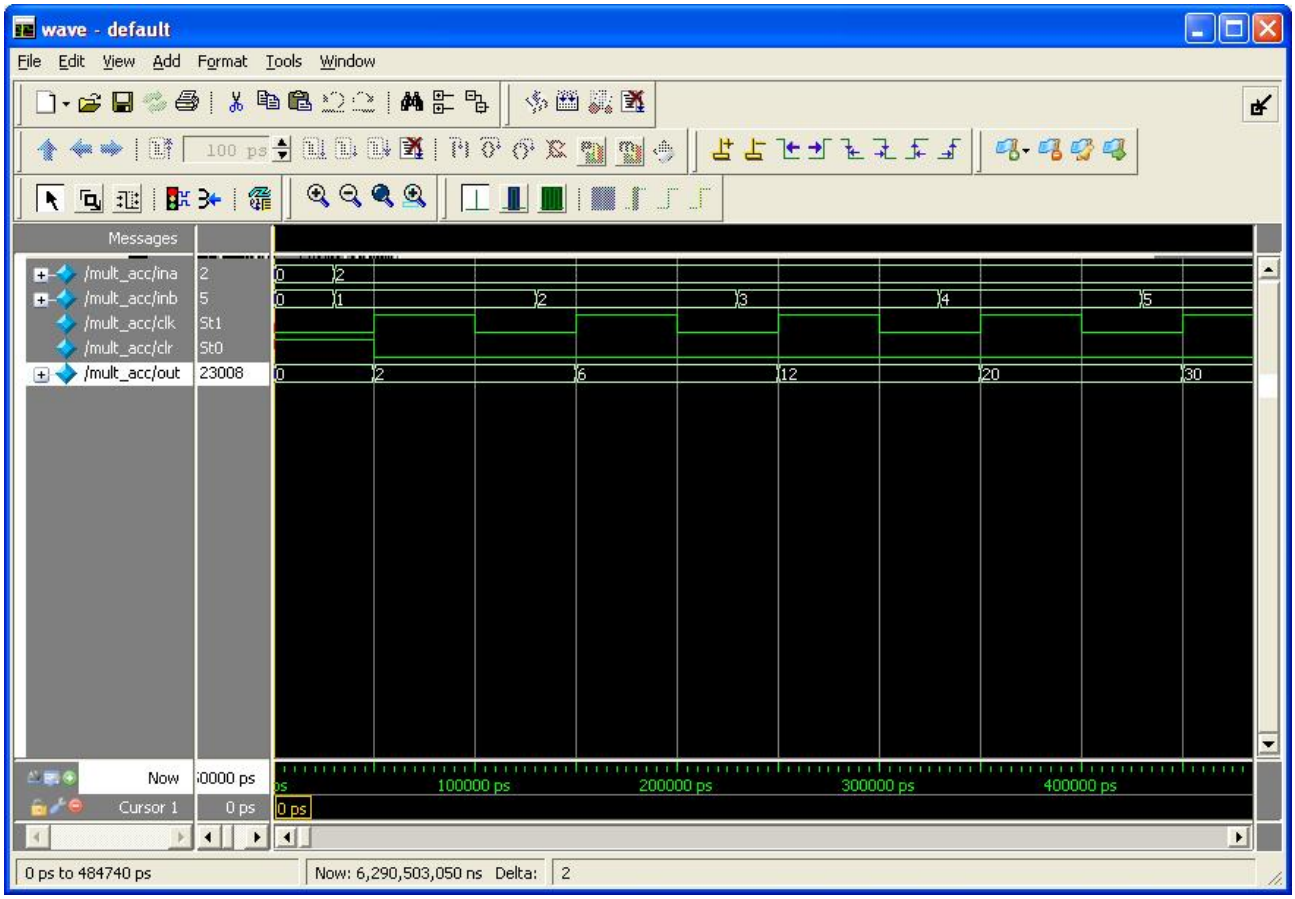


Рис. 10. Результат моделювання.

Перевірте дійсність роботи пристрою за результатами моделювання.

ПРИМІТКА Запуск моделювання може здійснюватися за допомогою команди **run** з вікна повідомлень. Формат команди: **run 100** - виконання 100 кроків моделювання; **run 1000ns** - виконання моделювання до вказаної позначки часу, що віддалена на 1000 наносекунд від поточної позиції; **run @ 300** - виконання моделювання до позначки часу,

віддаленої на 300 кроків моделювання від нульової позиції; **run @ 1500ns** - виконання моделювання до позначки часу, віддаленої на 1500 наносекунд від нульової позиції; **restart -force** - щоб скинути результати моделювання.

Зверніть увагу - для повторного моделювання, після скидання результатів, необхідно знову підключити до проекту файл з вхідними тестовими сигналами.

14. Для виходу з режиму моделювання в меню Simulate оберіть команду End Simulation.

3. Самостійна робота

Змінюючи значення вхідних сигналів і часові параметри у файлі Stim.do за завданням викладача, перевірте працездатність тестованого пристрою.

Лабораторна робота № 2
«Дослідження комбінаційних пристроїв»

1. Теоретичні відомості

Мови опису апаратури (HDL, Hardware Descripion Language) можуть бути використані на всіх етапах розробки цифрових електронних систем. Вони застосовуються на етапах проектування, верифікації, синтезу і тестування апаратури, а також для передачі даних про проект, його модифікації та необхідну інформацію до його супроводу.

Мови опису апаратури, в основному, використовуються для проектування програмованих логічних пристроїв (PLD - Programmable Logic Devices) різного рівня складності, вентильних програмованих матриць (FPGA - Field Programmable Gate Array). На сьогодняшній день знайшли застосування кілька таких мов. Найбільш популярні з них - Абель, Palasm і Cupl (використовуються для пристроїв малої ступені складності), Verilog і VHDL (для складних PLD та FPGA).

Переваги HDL:

1) За наявного HDL опису можна синтезувати принципову схему пристрою - генерація RTL опису (RTL - Register Transfers Level).
2) Схему, описану за допомогою HDL, простіше документувати.
3) HDL дозволяє реалізувати як структурний опис вузлів та ланцюгів, так і поведінковий опис. Вони мають можливості викликати функції, описані іншими мовами програмування (наприклад "C").
4) HDL дозволяє описувати такі специфічні для цифрових схем поняття, як часові затримки та паралельне виконання операцій.
5) HDL дозволяє створювати принципові схеми, ієрархічні блоки, алгоритми функціонування пристроїв, а також тестові файли (test-bench) для перевірки створених модулів.

Мова Verilog

Verilog- це мова опису апаратури, що використовується для розробки і моделювання електронних систем. Ця мова (також відома як Verilog HDL) дозволяє здійснити проектування, верифікацію і реалізацію (наприклад, у вигляді НВІС) аналогових, цифрових та змішаних електронних систем на різних рівнях абстракції.

Verilog був розроблений фірмою Gateway Design Automaton для використання всередині компанії. Потім, в 1989 р, Verilog був відкритий для загального використання. Стандарт даної мови був прийнятий в 1995 році (IEEE1364-1995).

Verilog має простий синтаксис, схожий з мовою програмування "C". Мала кількість службових слів і простота головних конструкцій спрощують вивчення і дозволяють створювати ефективні програми. На опис однієї і тієї ж конструкції в Verilog потрібно в 3-4 рази менше символів, ніж у VHDL.

Цікавою особливістю мови Verilog є наявність стандарту PLI (Program Language Interface), який дозволяє додавати функції, написані користувачем (наприклад, на C), до коду симулятора.

Короткі теоретичні відомості з мови Verilog

Типи даних, що підтримуються:

- **integer** - 32-х розрядне ціле число зі знаком;
- **real**- 64-х розрядне число з плаваючою крапкою. Синтезуючі САПР цей тип не підтримують;

• **time** - 64-х розрядне ціле беззнаковое число, що застосовується вбудованими функціями для опрацювання часу моделювання;

Створювати свої типи даних в Verilog не можна.

Крім основних типів даних, які притаманні всім мовам програмування, в HDL додається нове поняття - сигнал.

Сигнали бувають двох головних типів:

wire - ланцюги;

reg - регістри.

Відмінність сигналів **wire** від сигналів **reg** полягає в тому, що **reg** здатний зберігати присвоєне значення (працює як змінна в мовах програмування, або як пристрій послідовного типу). До сигналів типу **wire** потрібно безперервно докладати вплив (як до пристрою комбінаційного типу). Тобто, **wire** моделює зв'язок (або пристрій), який переходить в невизначений стан при відключенні вхідного впливу. Існують також типи **wand**, **wor**, **tri0**, **tri1**, **triand**, **trior, trireg** для моделювання різних типів зв'язків (**wand** - wired and або «монтажне І», **tri0** – зв’язок з можливістю переходу у високоімпедансний стан, який керується сигналом з активним нульовим рівнем, **trireg** - накопичувальна ємність, і т.п.), але такі ланцюги зустрічаються рідко і використовуються тільки для моделювання.

Ідентифікатори:

Ідентифікатори в Verilog (власні імена) є чутливими до прописних і рядкових символів і підкоряються звичайним правилам: не можуть починатися з цифри або знака $, можуть містити літери, цифри, $, та символ підкреслення.

Коментарі:

Коментарі в мові Verilog бувають двох типів і повністю відповідають коментарям мови С ++:

• // - це коментар (однорядковий)

• / *
і це коментар (груповий)
* /

Приклад:

```
integer i, j, k;          // оголошення змінних i, j і k типу integer
time start, duration;

/ * Далі йдуть оголошення
однобітових сигналів * /

wire a, b, c;
reg store, ff, A;  // reg A не збігається з wire a
```

Опис шин (груп сигналів):

Для опису шин або регістрів неодиничної розрядності використовується вказівник діапазони виду
[n: m], де m і n цілі числа.

```
wire [7: 0] data_bus;
reg [3: 0] high_nibble, low_nibble; // два 4-х розрядних регістра
```

Масиви (багатовимірні) в Verilog не підтримуються, але існує поняття "пам'яті". Пам'ять відповідає двовимірному масиву.

reg [8: 0] Fifo [31: 0]; // пам'ять, що складається з 9 32-х розрядних осередків

Припустимі значення сигналів:

Всього існує чотири типи значень, які можуть приймати сигнали (wire та reg): 0, 1, z, x. Перші три відповідають двом логічним рівням і стану з високим опором. Четвертий (x) - означає невизначений стан і використовується при моделюванні неініціалізованих сигналів, при виникненні конфліктів (два виходи з протилежними станами, які пов'язані один з одним), визначення нестабільних станів тригерів (при порушенні часових співвідношень між входами даних і тактовим входом) і т.п. Іншими словами - у всіх випадках, коли моделююча програма не може визначити значення для даного сигналу.

Для визначення значення багаторозрядних сигналів (констант, змінних) використовуються наступні конструкції: 1) **1'bz** - однорозрядний високоімпедансний сигнал; 2) **10'd1_000** - десятирозрядне число 1000, записане в десятковому вигляді (символ підкреслення ігнорується); 3) **4'bx01z** - чотирьохрозрядне двійкове число з невизначеним старшим бітом, високоімпедансним молодшим бітом, другим і третім бітами в стані логічних «1» та «0», відповідно.

У **загальному вигляді** - спочатку вказується розрядність сигналу (кількість двійкових бітів), потім одинарна лапка '(не плутати з апострофом `), далі підстава системи числення (b, o, d, h) і цифри, що використовуються в даній системі числення, які задають значення сигналу.

Для двійкової системи дозволено використання символів z і x.

Символ підкреслення застосовується для поліпшення сприйняття запису та ігнорується при синтезі і моделюванні. Використання констант без вказівки розрядності не бажано, тому що за замовчуванням константа сприймається з розрядністю 32 біта.

Дані типу **integer** можуть бути присвоєні змінній, що має тип **reg**.

Файли на мові Verilog

Мають розширення ".v". Наприклад: "module1.v", "NCO.v", "oscillator.v"

Структурний опис:

Основною структурною одиницею програми на мові Verilog є **module**. Модуль декларується ключовими словами **module - endmodule**. В одному програмному файлі можуть бути присутніми декілька модулів. Модулі не можуть бути вкладеними. Інші модулі можуть підключатися до вхідних і вихідних портів модуля, утворюючи ієрархічну структуру. При запуску компілятора мови Verilog, він формує ієрархічне дерево проекту з усіх підключених модулів і знаходить модуль верхнього рівня ієрархії. Важливо пам'ятати, що ім'я модуля верхнього рівня ієрархії має збігатися з ім'ям файлу, в якому цей модуль описаний!

Загальна структура модуля:

module SomeModule (Param1, Param2, Param3, ..., ParamN);
input Param1; // вхідний порт з ім'ям Param1
output Param2; // вихідний порт з ім'ям Param2
inout Param3; // двонаправлений порт (вхід / вихід) з ім'ям Param3
.....
`Timescale 1ns / 10ps
.....
`Include" somefile.v "
`Define nc @ (negedge clk)
/ *
Тут може бути ваш код !!!

**/*
endmodule

Отже, відразу ж після директиви **module**, вказують ім'я модуля, а за ним - в круглих дужках, послідовно імена портів модуля (інтерфейс модуля з зовнішнім світом). Порти можуть бути вхідними - **input**, вихідними - **output** або двонаправленими - **inout**. Після опису портів, можуть йти директиви компілятора **`include**,**` define**, і ін. Вони цілком збігаються з директивами компілятора мови С (*один важливий момент* - вищезгадані директиви починаються з символу апострофа, а не одинарних лапок).

Ще один нюанс, пов'язаний з використанням директиви **`include**. Як відомо, ця директива застосовується для включення тексту одного файлу в інший. Однак в Verilog модулі не можуть оголошуватися всередині іншого модуля (не можуть бути вкладеними). Тому, директиву **`include** можна використовувати або поза модулем, або включати в модуль код, який не містить опису модулів.

Директиву **`timescale** ми розглянемо в наступній лабораторній роботі.

Ключове слово assign

Ключове слово **assign** використовується для присвоєння значення сигналу типу **wire**. Даний оператор не використовується в процедурних блоках. Його синтаксис наступний:

assign <wire_name> = <expression>;

Параметр <expression> може бути представлений логічним виразом. Параметр <wire_name> представляє собою ім'я сигналу типу **wire**. Як тільки значення <expression> змінюється, отримане нове значення <expression> присвоюється сигналу з ім'ям <wire_name>. Детальніше, оператор **assign** буде розглянуто в лабораторних роботах.

2. Порядок виконання роботи

Наведемо мовою Verilog опис схеми виключного «АБО» (XOR).

Таблиця дійсності для XOR:

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

Схема, що реалізує дану функцію:

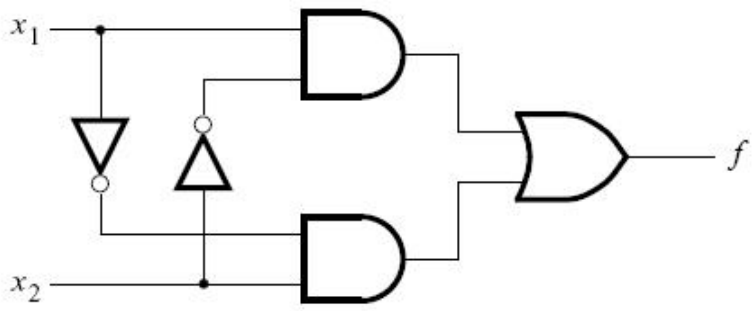


Рис. 11. Схема виключного «АБО» (XOR).

1. Створіть новий проект в середовищі ModelSim.

2. Створіть новий файл на мові Verilog з описом роботи пристрою (команда Create New File діалогового вікна Add items to the Project). Вкажіть ім'я файлу «ExclusiveOR» і мову опису файлу Verilog в діалоговому вікні Create Project File. Закрийте діалогове вікно Add items to the Project

3. Відкрийте текстовий редактор (подвійне натискання лівої кнопки миші на імені файлу). Наберіть наступний текст:

```
module ExclusiveOR (x1, x2, f);
    input x1, x2;
    output f;
    assign f = (x1 & ~ x2) | (~ x1 & x2);
endmodule
```

Збережіть файл.

4. Скомпілюйте проект.

5. Створіть файл з вхідними тестовими сигналами. У меню File виберіть команду New => Source => Do. У новому вікні наберіть наступний код:

```
force x1 2 # 0 0ns, 2 # 1 100ns;
force x2 2 # 0 0ns, 2 # 1 50ns, 2 # 0 100ns, 2 # 1 150ns;
```

і збережіть його під ім'ям Stimul.do.

6. Перейдіть в режим моделювання. Відкрийте графічне вікно і додайте до нього сигнали, що перевіряються. Підключіть до проекту файл з тестовими сигналами. Запустіть проект на моделювання. Перевірте отримані результати:

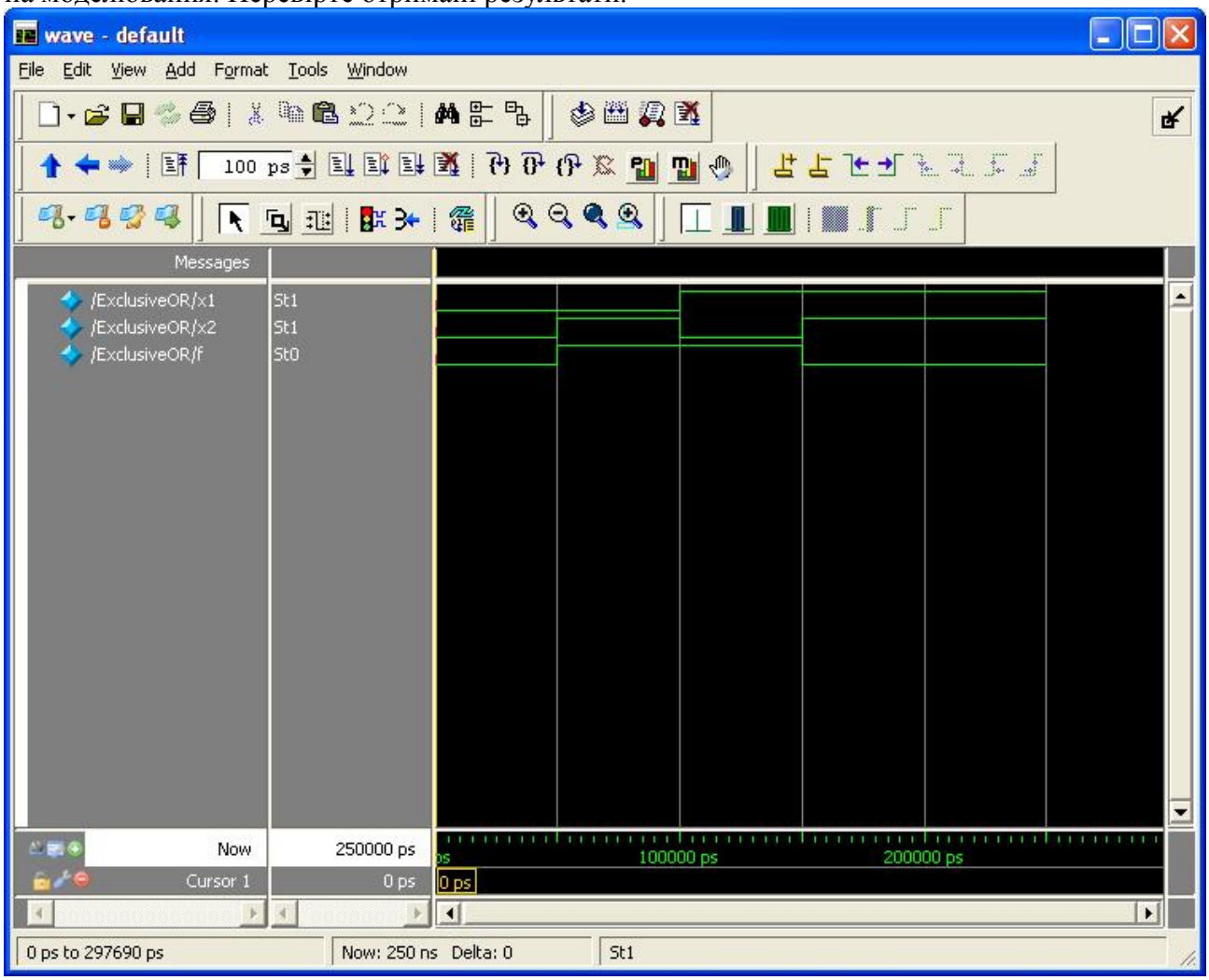


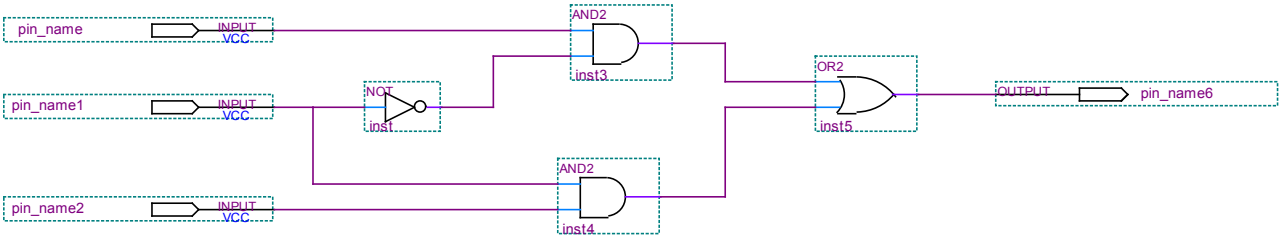
Рис. 12. Отримані результати.

7. Вийдіть з режиму моделювання.

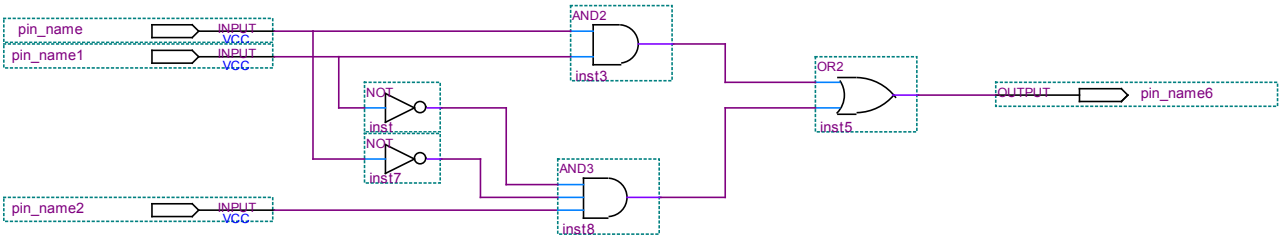
3. Самостійна робота

Завдання 1. Створіть опис і промоделюйте роботу наступних пристроїв.

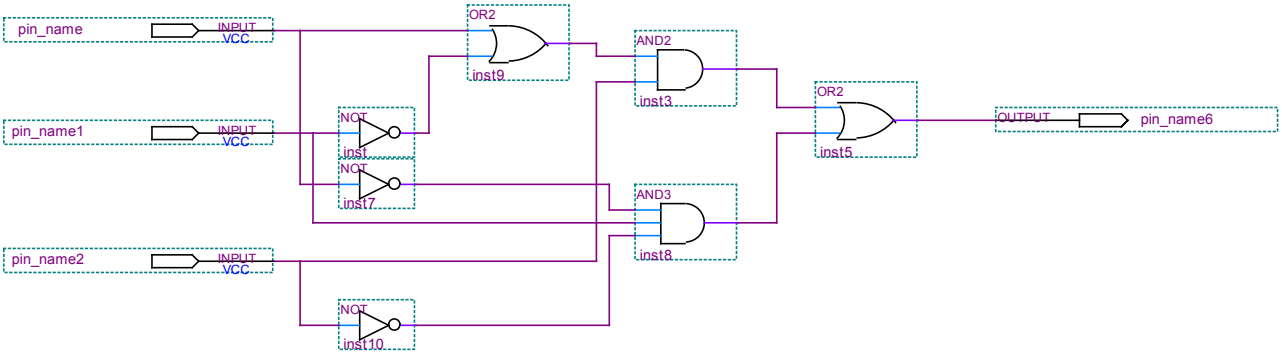
Варіант 1.



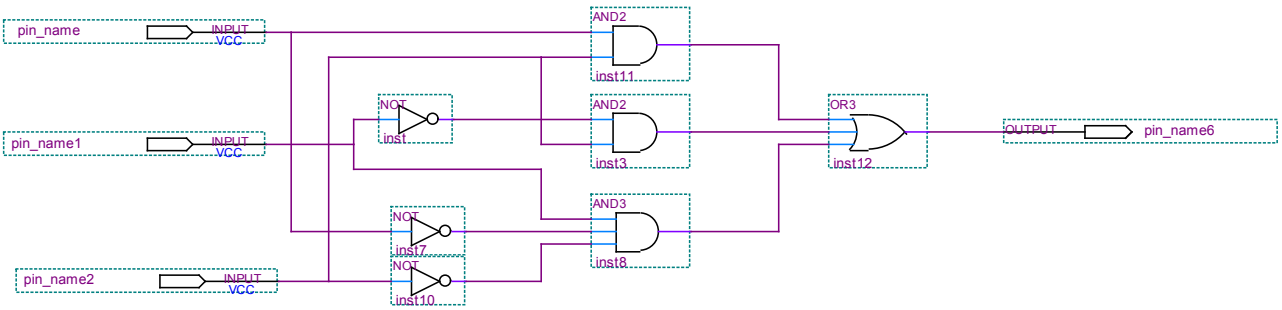
Варіант 2



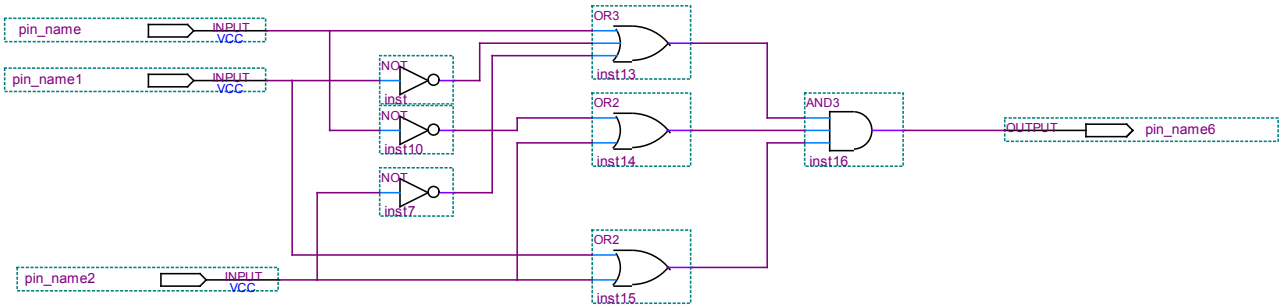
Варіант 3



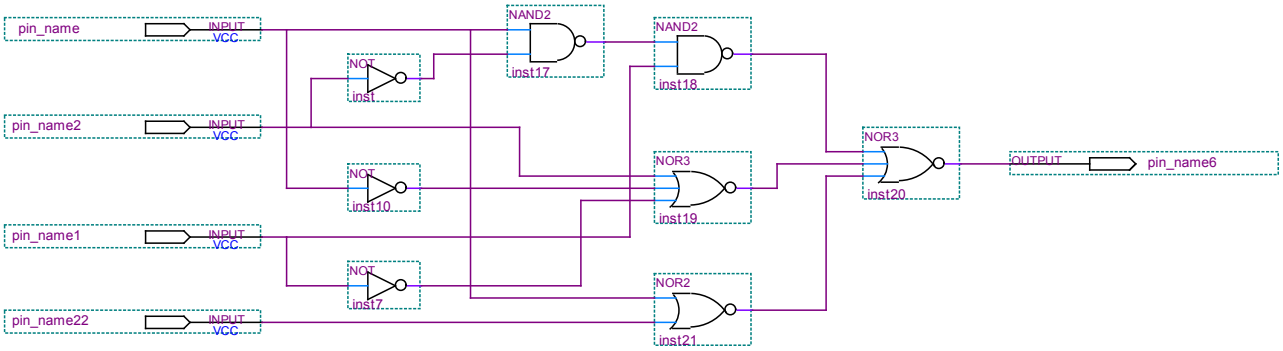
Варіант 4



Варіант 5



Варіант 6



Завдання 2. Створіть опис (якщо можливо - мінімізуйте функцію) і промоделюйте роботу пристроїв, заданих наступними логічними функціями:

1) **F = X '* Y' * Z '+ X * Y * Z + X * Y' * Z;**
2) **F = A * B + A * B '* C' + A '* B * C;**
3) **F = A '* B * (C * B * A' + B * C ');**
4) **F = X * Y * (X '* Y * Z + X * Y' * Z + X * Y * Z '+ X' * Y '* Z);**
5) **F = (A + A ') * B + B * A * C' + C * (A + B ') * (A' + B);**
6) **F = X * Y '+ Y * Z + Z' * X;**

Лабораторна робота № 3
«Комбінаційні пристрої»

1. Теоретичні відомості

Комбінаційним пристроєм називається такий пристрій, вихідні сигнали якого залежать лише від поточних значень вхідних сигналів.

До комбінаційних пристроїв відносяться мультиплексори, демультиплексори, шифратори, дешифратори та ін.

Мультиплексором називається комбінаційний логічний пристрій, призначений для управління передачею даних від декількох джерел одному вихідному каналу. Відповідно до визначення, мультиплексор повинен мати один вихід і щонайменше дві групи вхідних контактів: інформаційні та адресні. Код, що надходить на адресні входи, визначає, який з інформаційних входів в цей момент підключено до вихідного каналу. Якщо кількість адресних входів мультиплексора дорівнює n, то максимально можлива кількість його інформаційних входів буде 2^n.

Логічне рівняння, що описує роботу чотирьохканального мультиплексора (має 4 інформаційних сигнали D0 ... D3 і 2 адресних сигнали A0 ... A1) показано нижче:
Q = D0 * A1 '* A0' + D1 * A1 '* A0 + D2 * A1 * A0' + D3 * A1 * A0

Схема найпростішого однорозрядного мультиплексора 2-в-1 показана на малюнку 13:

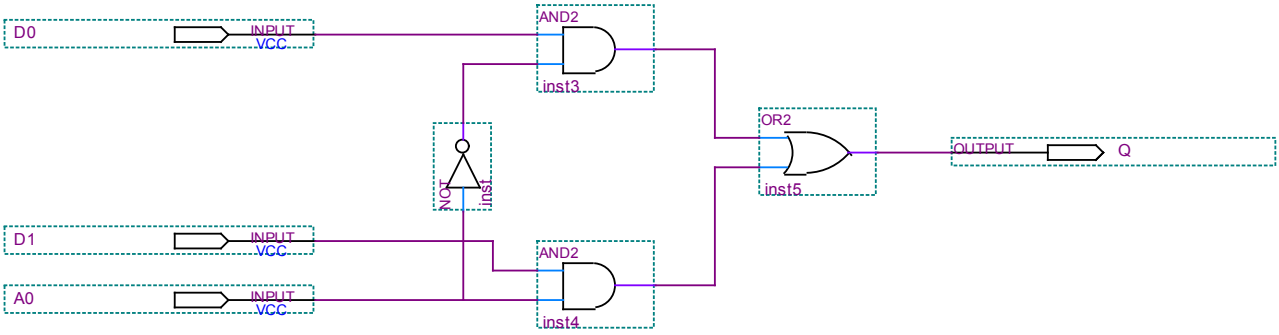


Рис. 13. Однорозрядний мультиплексор 2-в-1.

Таблиця дійсності такого мультиплексора наведена нижче:

A0	Q
0	D0
1	D1

Цей мультиплексор реалізується мовою Verilog за допомогою наступного оператора:

Assign Q = (~ A0 & D0) | (A0 & D1);

Демультиплексором називається комбінаційний логічний пристрій, призначений для управління передачею даних від одного вхідного каналу на декілька вихідних. Відповідно до визначення, демультиплексор в загальному вигляді має один інформаційний вхід, n адресних входів і 2^n виходів.

Набір логічних рівнянь, що описують роботу чотирьохканального демультиплексора (має 1 вхідний інформаційний сигнал D, 2 адресних сигнали A0 ... A1 і 4 вихідних сигнали Q0 ... Q3) показано нижче:

Q0 = D * A1 '* A0'
Q1 = D * A1 '* A0

Q2 = D * A1 * A0 '
Q3 = D * A1 * A0

Схема найпростішого однорозрядного демультиплексора 1-в-2 показана на малюнку 14:

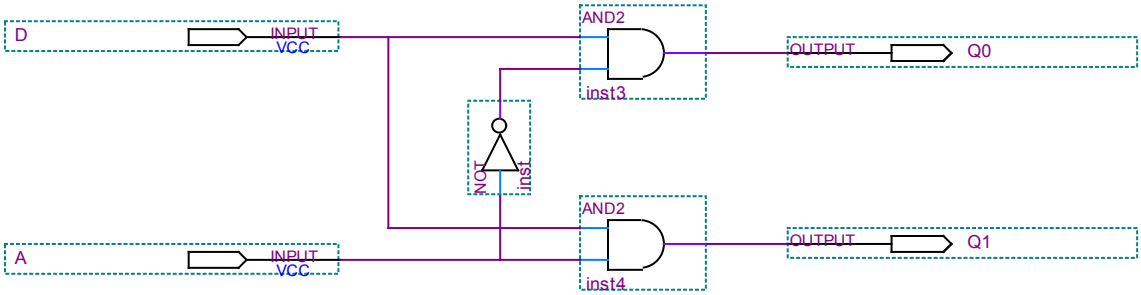


Рис. 14. Однорозрядний демультиплексор 1-в-2.

Його таблиця дійсності наведена нижче:

A	Q0	Q1
0	D	0
1	0	D

Такий демультиплексор реалізується мовою Verilog за допомогою набору наступних операторів:

Assign Q0 = ~ A0 & D0;
Assign Q1 = A0 & D1;

Шифратором або кодером називається комбінаційний логічний пристрій, що перетворює вхідний код з десяткової системи числення в двійкову. Входам шифратора послідовно присвоюються значення десяткових чисел, а активний логічний рівень сигналу на одному з входів сприймається шифратором як подача відповідного десяткового числа. На виході шифратора формується відповідний двійковий код. Відповідно, якщо шифратор має n виходів, то кількість вхідних сигналів не повинна перевищувати 2^n. Шифратор, який має 2^n входів і n виходів називається повним. Якщо кількість входів шифратора менше 2^n, він називається неповним.

Набір логічних рівнянь, що описують роботу шифратора чисел від 0 до 7 в двійковий код (**x0 ... x7** - вхідні сигнали, Q0 ... Q2 - вихідні сигнали) показано нижче:

Q0 = x1 + x2 + x3 + x5 + x7
Q_1 = x2 + x3 + x6 + x7
Q_2 = x4 + x5 + x6 + x7

Дешифратором або декодером називається комбінаційний логічний пристрій для перетворення чисел з двійкової системи числення в десяткову. Це означає, що кожному вхідному двійковому числу ставиться у відповідність сигнал, що формується на певному виході дешифратора. Дешифратор, який має n входів і 2^n виходів називається повним. Якщо кількість виходів дешифратора менше 2^n, він називається неповним.

Набір логічних рівнянь, що описують роботу двухразрядного дешифратора (**Q0 ... Q1** - вхідні сигнали, x0 ... x3 - вихідні сигнали) показано нижче:

x0 = Q0 '* Q1'
x_1 = Q0 * Q1 '
x_2 = Q0 '* Q1
x_3 = Q0 * Q1

3. Самостійна робота

Завдання 1.

Реалізуйте мультиплексор (демультиплексор) і перевірте його роботу в середовищі ModelSim відповідно до заданого варіанту:

1) Восьмизарядний мультиплексор 4-в-1 (кількість інформаційних сигналів - 4, розрядність - 8 біт).
2) Чотирьохрозрядний мультиплексор 5-в-1 (кількість інформаційних сигналів - 5, розрядність - 4 біт).
3) Однорозрядний мультиплексор 8-в-1 з входом дозволу роботи EN. Активний рівень сигналу EN - логічний «0». При подачі на вхід EN сигналу з рівнем логічної «1», вихідний сигнал мультиплексора повинен встановлюватися в високоімпедансний стан.
4) Восьмизарядний демультиплексор 1-в-4 (кількість вихідних сигналів - 4, розрядність - 8 біт).
5) Чотирьохрозрядний демультиплексор 1-в-5 (кількість вихідних сигналів - 5, розрядність - 4 біт).
6) Однорозрядний демультиплексор 1-в-8 зі входом дозволу роботи EN. Активний рівень сигналу EN - логічний «0». При подачі на вхід EN сигналу з рівнем логічної «1», вихідні сигнали демультиплексора повинні встановлюватися в високоімпедансний стан.

Завдання 2.

Реалізуйте шифратор (дешифратор) і перевірте його роботу в середовищі ModelSim відповідно до заданого варіанту:

1) Шифратор, що перетворює цифри від 0 до 9 в двійково-десятковий код.
2) Шифратор, що перетворює цифри від 0 до 15 в двійковий код.
3) Шифратор, що перетворює цифри від -5 до 4 в двійковий додатковий код.
4) Повний чотирьохрозрядний дешифратор.
5) Двійково-десятковий дешифратор.
6) Дешифратор для семисегментного індикатору. Перетворює вхідний 4-розрядний двійковий код в шістнадцяткові цифри від 0 до F, які відображаються на індикаторі. Розташування сегментів індикатору показано на малюнку 15:

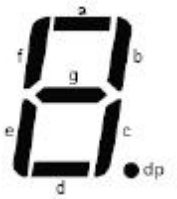


Рис. 15. Сегменти індикатору.

Активний рівень сигналу для включення сегменту - логічний «0».

Лабораторна робота № 4
«Дослідження послідовних логічних пристроїв».

1. Теоретичні відомості.

Короткі теоретичні відомості з мови Verilog (продовження).

При роботі з мовами опису апаратури необхідно завжди пам'ятати одну особливість. Програма, написана цією мовою (в тому числі - Verilog) виконується **паралельно**! У цьому головна відмінність мови Verilog від таких процедурних мов програмування, як C і Pascal. Зрозуміти це досить просто. Адже за допомогою мов HDL описується схема електрична принципова (тобто - власне пристрій, в якому електричні сигнали завжди поширюються **одночасно)**.

Для цього, в синтаксисі мови Verilog, передбачені наступні оператори, що виконуються паралельно:

• **assign**;

• **always** (цей важливий оператор ми розглянемо пізніше).

Таким чином, якщо в програмі опису модуля використовується декілька операторів **assign** і **always**, всі вони будуть виконуватися паралельно (одночасно).

Поняття моделювання та синтезу.

Роботу логічного пристрою, описаного за допомогою мови HDL, можна перевірити двома способами: або на персональному комп'ютері (**моделювання**), або безпосередньо реалізацією в мікросхемі (наприклад - ПЛІС) (**синтез**). Процес синтезу набагато складніше моделювання, оскільки передбачає, крім компіляції коду, ще й синтез логічної схеми (тобто, реалізацію пристрою за допомогою доступних компонентів ПЛІС і трасування зв’язків в обраній мікросхемі). Тому результат застосування деяких конструкцій мови Verilog при моделюванні і синтезі **відрізняється**! Не всі оператори мови Verilog є синтезованими!

Зазвичай, для моделювання використовуються спеціальні тестові програми - тестбенчі (**testbench**), які призначені для перевірки програми опису модулів (їх верифікації). Для виконання моделювання використовується спеціалізоване програмне забезпечення, наприклад - **ModelSim**.

Час моделювання.

Отже, очевидними є три факти. По-перше, моделювання логічного пристрою можна виконувати на ПК. По-друге, багато операторів мови Verilog виконуються паралельно. По-третє, ПК виконує інструкції строго послідовно. Виникає питання - як за допомогою комп'ютера виконати моделювання подій, які відбуваються одночасно? Для відповіді на це питання необхідно спочатку розібратися з термінами **реального** і **модельного** часу. Природно, що реальний час відрізняється від модельного. У реальному часі описуються всі зміни, що відбуваються в пристрої, який моделюється. У модельному часі описуються всі зміни, що відбуваються в моделюючій програмі. Такий поділ дозволяє виконати на комп'ютері спочатку послідовно всі паралельні інструкції, а потім надати результат в такому вигляді, як ніби-то вони виконувалися одночасно.

Для завдання модельного часу в мові Verilog служить конструкція **`timescale param1 / param2**. **Param1** задає одиницю модельного часу (за замовчуванням - 1 нс), а **Param2** - точність модельного часу. Наприклад **`timescale 1ns / 10ps** означає, що одиниця модельно часу -1 наносекунда. Крок в 10 одиниць модельного часу складає 10 нс. Значення 10 пікосекунд (інший параметр директиви **timescale**) вказує на точність, з якою будуть виконуватися обчислення всіх змін за один крок моделювання. Природно, що при

обчисленні затримок в пристрої, час буде округлятися з точністю до 10 пікосекунд. За допомогою цієї конструкції можна регулювати точність моделювання.

Часові затримки.

Розрізняють часові затримки, пов'язані з моделюванням і синтезом. **Часові затримки при моделюванні** задаються користувачем за допомогою оператора **#time** (параметр **time** визначає інтервал затримки в одиницях модельного часу). Зазвичай, такі затримки імітують кінцевий час поширення сигналів в реально працюючих пристроях. Наприклад, щоб сформувати затримку в 5 наносекунд, необхідно записати наступний код: **# 5;**

Часову затримку можна вводити перед оператором присвоєння значення сигналу, моделюючи час поширення сигналу в реальному пристрої:

assign # 10 c = a ^ b;

Дана конструкція описує елемент «виключне АБО» (XOR) з затримкою поширення, що дорівнює 10 (10 одиниць модельного часу - першого параметру директиви **`timescale**, який за замовченням дорівнює 1 нс). При цьому, всі затримки в командах безперервного присвоєння є інерційними. Це означає, що зміни сигналу A у часі, менші за 10 нс, не приведуть до змін сигналу C. Для того, щоб відбулася зміна сигналу C потрібно, щоб сигнал A був зафіксований в новому стані на час, більший ніж 10 нс.

А що ж буде при синтезі? Якщо використовувати затримки такого виду при синтезі, то нічого не станеться! Даний оператор є несинтезованим, тобто він буде ігноруватися. Для формування затримок в реальному пристрої використовуються інші методи.

Оператори мови Verilog

Оператор **assign**

Використовується для безперервного присвоєння значення сигналу для змінної типу **wire**. Синтаксис:

assign *var = expression;*

Дія оператора: При зміні виразу *expression* (наприклад, змінилося значення змінної, що входить в зазначений вираз), обчислюється нове значення виразу і результат присвоюється змінній *var*. У лівій частині виразу змінна може бути лише типу **wire**, а в правій частині - можлива комбінація змінних **wire**, **reg**, **integer**. Всі оператори **assign** в модулі виконуються паралельно.

Оператор **always**

Один з основних, ефективних операторів мови Verilog. Дозволяє задати постійне виконання послідовності команд. Зазначена послідовність може виконуватися або циклічно (в нескінченному циклі), або тільки після появи певної події.

Синтаксис:

always @ (... events ...)
begin
...
//послідовність операторів (виконується послідовно, один за іншим)
...
end

Опис: Даний оператор застосовується тоді, коли виникає необхідність послідовного виконання команд. Для цього, після ключового слова **always**, слід вказати необхідну послідовність операторів, укладених в блок **begin / end**. Всі команди, розташовані всередині такого блоку, виконуються послідовно, а самі оператори **always** - паралельно. Коли блок **begin / end** відсутній, то дія оператора **always** поширюється лише на один

наступний за ним оператор. Якщо в операторі **always** відсутня конструкція **@ (... events ...)**, то набір команд, розташованих між **begin** та **end**, виконується в нескінченному циклі.

Для завдання умови, необхідної для запуску блоку **always**, використовується конструкція **@ (events)**. В даному випадку, всередині круглих дужок вказується список умов (подій), що ініціюють виконання оператора **always**. Умовою може бути поява зростаючого або спадаючого фронту сигналу, зміна його значення та ін. Зростаючий фронт вказується ключовим словом **posedge**, спадаючий фронт - **negedge**. Після ключових слів **posedge** або **negedge** слід вказати ім'я сигналу. Якщо необхідною умовою є будь-яка зміна сигналу - в дужках просто вказується його ім'я.

Приклади:

always @ (posedge CLK) // умова запуску оператора **always** – зростаючий фронт сигналу *CLK;*

always @ (negedge CLK) // умова запуску оператора **always** – спадаючий фронт сигналу *CLK;*

always @ (CLK) // умова запуску оператора **always** - будь-яка зміна рівня сигналу *CLK;*

Якщо необхідно вказати кілька умов, що призводять до запуску оператора **always**, всередині дужок їх об'єднують операторами **or** або **and**. Або записують через кому (це є еквівалент оператора **or**). Наприклад, наступні оператори описують умови запуску RS тригера:

always @ (posedge CLK or R or S) // обидві ці записи призводять
always @ (posedge CLK, R, S) // до однакового результату

Можна використовувати неявну вказівку подій запуску. В цьому випадку, всередині дужок ставиться зірочка **(*)**. Спрацьовування оператора **always** відбудеться при будь-якій зміні значень змінних, що стоять в правій частині оператора присвоєння, зміні умов і виразів в операторах **if** та **case** (коли вони використовуються в операторі **always**). Приклад:

always @ (*) // те ж саме, що записати always @ (a or b or c or d or f)
y = (a & b) | (c & d) | myfunction (f);

Важливе зауваження! Всередині оператора **always** не можна використовувати змінні типу **wire** для присвоєння їм нового значення (тобто всі змінні, яким присвоюється значення всередині даного оператора, повинні бути типу **reg**). Інакше, це призведе до помилок компіляції. Всередині оператора **always** неможливо викликати інший модуль. Всі оператори **always** виконуються паралельно.

Оператор **initial**

Даний оператор використовується для завдання послідовності команд, яку необхідно виконати лише один раз - при запуску проекту на моделювання. Зверніть увагу, оператор **initial** не підтримується при синтезі, а використовується лише при моделюванні.

Синтаксис:

initial
begin
...
// операторі (виконуються послідовно)
...
end

Опис: Оператори, розташовані між конструкцією **begin** та **end** виконуються лише одного разу - при запуску програми на моделювання. Вони дозволяють задати початковий стан пристрою.

Оператор конкатенації **{}**

Фігурні дужки (оператор конкатенації) використовується для об'єднання («склеювання» або злиття) декількох різних сигналів в один. Розрядність отриманого сигналу дорівнює сумі розрядностей всіх сигналів, що поєднуються.

Синтаксис:

{arg1, arg2, ..., argN}

Опис: Змінні різної розрядності arg1 ... argN об'єднуються в один сигнал.

Оператор перевірки умови**?:**

Оператор перевірки умови (знак питання з двокрапкою для поділу дій) на мові Verilog працює схоже з оператором мови С.

Синтаксис:

assign *var1 = (condition)? var2: var3;*

Опис: Спочатку перевіряється умова **condition**, якщо вона істинна - виконується перша команда, якщо ні – команда, наступна за розділовою двокрапкою. Приклад реалізації простішого мультиплексора 2-в-1 за допомогою умовного оператора:

assign Y = (SEL)? A: B;

Оператор **parameter**. Оголошення символічного імені.

Цей оператор використовується тоді, коли необхідно призначити константам символічні імена.

Синтаксис:

parameter *symbolic_name = dig_number;*

Опис: цей оператор оголошується відразу після опису портів модуля і внутрішніх змінних. Наприклад, при створенні кінцевого автомату, станам автомату зручно привласнити символьні імена:

parameter *[1: 0] state_A = 0, state_B = 1, state_C = 2, state_D = 3;*

Оператор вибору **case**

Застосовується для вибору одного значення з декількох варіантів.

Синтаксис:

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

end
default:
begin
....
end
endcase

Опис: Оператор вибору **case** перевіряє вираз *expr* і, в залежності від його значення, виконується відповідна команда. Даний оператор гарантує виконання однієї гілки. У разі, якщо жодне з наданих значень не збігається з поточним, виконується гілка **default**. Приклад використання оператору вибору - реалізація двійковій-десяткового дешифратора (аналог м \ с К155ІД3):

case *(rega)*
4'd0: result = 10'b0111111111;
4'd1: result = 10'b1011111111;
4'd2: result = 10'b1101111111;
4'd3: result = 10'b1110111111;
4'd4: result = 10'b1111011111;
4'd5: result = 10'b1111101111;
4'd6: result = 10'b1111110111;
4'd7: result = 10'b1111111011;
4'd8: result = 10'b1111111101;
4'd9: result = 10'b1111111110;
default*: result = 10'bx;*
endcase

Оператор перевірки умови **if ... else**
Синтаксис:

if *(<Expression>)*
<Statement1>
else
<Statement2>

Опис: оператор перевірки умови перевіряє істинність виразу *Expression* (будь-який вираз мовою Verilog). Якщо умова істинна - виконується команда *Statement1* (оператор або група операторів між **begin** та **end**). Якщо умова помилкова - виконується команда *Statement2*, розташована в гілці **else**. Гілка **else** може бути відсутня, але якщо оператор містить додаткові умовні оператори **if** (як в прикладі нижче), то **else** відноситься до найближчого до неї оператору **if**. Для виконання кількох команд, при збігу умови, слід користуватися конструкцією **begin** та **end**. Вираз *Expression* вважається дійсним, якщо його значення не дорівнює 0 і не є невизначеним (x або z). Слід пам'ятати, що так само як і в мові С, операція порівняння записується як == (два поспіль знака =), на відміну від операції присвоювання = (один знак). Операція порівняння, при невизначених операндах, повертає невизначене значення (*x*). Вираз *Expression* не обов'язково має бути виразом типу *boolean*, а може бути будь-яким виразом, який може бути приведено до типу **integer**. Тут відслідковується аналогія з мовою С, єдина відмінність полягає в тому, що тип **integer** в мові Verilog, на відміну від типу **int** мови С, може приймати невизначені значення (*x* або *z*). У разі, коли вираз приймає ці значення, виконується гілка **else**. Приклад використання умовного оператору - реалізація синхронного однорозрядного регістра-мультиплексора з сигналом дозволу виходу OE:

```
  always @(Negedge Clk)
begin
        if (OE)
                out = data;
        if (LE)
                data = inA;
        else
                data = inB;
  end
```

Оператор циклу **for**

Даний оператор застосовується для організації циклічного повторення команди або групи команд при відомомій кількості ітерацій.

Синтаксис:

```
for (_index = __low_range; __index <__high_range; __index = __index + __step)
begin: (Name)
...
// some code (група циклічно виконуваних команд)
...
end
```

Опис: принцип роботи оператору циклу **for** нагадує роботу оператору циклу мови С. Єдина відмінність полягає в тому, що при збільшенні змінної *__index* виникає спокуса скористатися оператором + =, якого немає в мові Verilog. В даному випадку використовується конструкція *__index = __index + __step*. Для дострокового виходу з циклу (блоки команд повинні бути іменовані - унікальне ім'я, після ключового слова ***begin:,*** використовується оператор **disable**. Для порівняння з мовою С: дія оператора **disable** збігається з дією оператора **break** мови С

Приклад:

```
initial
begin: break
for (I = 0; i <n; i = i + 1)
always @CLK if (a == 0)
disable break;
end
```

Оператор циклу **while**

Синтаксис:

```
while (condition)
begin
...
// some code (виконуваний фрагмент програми)
...
end
```

Опис: оператор циклу **while** працює так само, як в мові С. Якщо умова, що перевіряється (*condition*) істинна, то виконується послідовність команд, розташованих між операторами ***begin*** та **end**. Якщо умова помилкова - оператор циклу не виконується. Він використовується лише в процедурних блоках, наприклад - в секції **initial**. Оператор циклу **while** не синтезується (застосовується лише для моделювання).

Оператор циклу **repeat**

Даний оператор дозволяє організувати цикл з кінцевою (фіксованою) кількістю повторень.

Синтаксис:

repeat*(N)*
begin

// some code (виконуваний фрагмент програми)

end

Опис: послідовність команд, розташованих між операторами **begin** та **end** буде виконана *N* разів. Цей оператор циклу використовується лише в процедурних блоках, наприклад - в **initial** або **always**.Оператор циклу **repeat** не синтезується (застосовується лише для моделювання).

Приклад застосування даного оператору - генерація спадаючої послідовності від 127 до 0:

```
integer count;
initial
begin
count = 128;
repeat (count)
begin
count = count - 1;
end
end
```

Оператор нескінченного циклу **forever**

Для реалізації нескінченного циклу в мові Verilog (аналогічного оператору **for (;;)** мови С) використовується оператор **forever**.

Синтаксис:

forever
begin

// some code (виконуваний фрагмент програми)

end

Опис: фрагмент програми, розташований між операторами **begin** та **end** буде повторюватися в нескінченному циклі. Тому, необхідно передбачити умови завершення циклу **forever**. Це можна зробити, наприклад, за допомогою системної функції **$finish**. Даний оператор використовується лише в процедурних блоках.Оператор нескінченного циклу **forever** не синтезується (застосовується лише для моделювання).

Приклад - формування тактового сигналу з періодом повторення, рівним 10 крокам модельного часу на інтервалі 2000 кроків модельного часу:

```
reg clock;
initial
begin
clock = 1'b0;
forever # 5 clock = ~ clock; // the clock flips every 5 time units.
end
initial # 2000 $finish;
```

Арифметичні і логічні оператори

Арифметичні оператори: **+ , - , * , / , %;**

Логічні оператори (для згрупованих значень, використовуються з однією змінною):

! - логічне заперечення;

&& - логічне "І";

|| - логічне "АБО";

Логічні оператори побітові (використовуються з декількома змінними):

~ - побітовое заперечення;

& - побітовое "І";

| - побітовое "АБО";

^ - побітовое "виключне АБО";

Оператори порівняння:

== - логічна рівність;

! = - логічна нерівність;

=== - побітова рівність (повний збіг);

! == - побітова нерівність (повне неспівпадання);

<= - логічне порівняння «менше - чи - дорівнює»;

> = - логічне порівняння «більше - або - дорівнює»;

> - логічне порівняння «більше»;

< - логічне порівняння «менше».

2. Порядок виконання роботи

Особливістю послідовних логічних пристроїв є залежність вихідного сигналу не лише від діючих в даний момент вхідних логічних сигналів, а й від тих, які діяли в попередній момент. Для виконання цих умов, послідовний пристрій повинен володіти пам'яттю. Функцію запам'ятовування значень логічних змінних в цифрових схемах виконують тригери. Таким чином, тригер є невід'ємною частиною будь-якого послідовного пристрою.

Наведемо мовою Verilog опис RS-тригеру, схема якого представлена на малюнку 16:

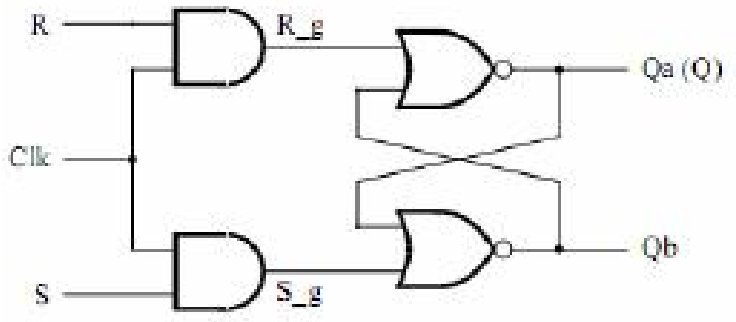


Рис. 16. RS-тригер.

Для цього:

1) Створимо новий проект в середовищі ModelSim.

2) Створимо новий похідний файл з описом RS-тригеру (похідний текст програми наводиться нижче):

```
`timescale 1 ns / 10 ps

module rstrig (Clk, R, S, Q);
 input Clk, R, S;
 output Q;

 wire R_g, S_g, Qa, Qb;
 reg Clk_r, R_r, S_r;
 parameter period = 50;

 initial
  begin
   Clk_r = 1'b0;
   forever # (Period / 2) Clk_r = ~ Clk_r;
  end
 initial
  begin
   R_r = 1'b0;
   # 40 R_r = 1'b1;
   # 80 R_r = 1'b0;
   # 130 R_r = 1'b1;
  end
 initial
    begin
   S_r = 1'b0;
   # 80 S_r = 1'b1;
  end
  initial
  # 180 $finish;

 and(R_g, R, Clk);
 and(S_g, S, Clk);
 nor(Qa, R_g, Qb);
 nor(Qb, S_g, Qa);

 assign Q = Qa;
 assign Clk = Clk_r;
 assign R = R_r;
 assign S = S_r;

endmodule
```

3) Скомпілюємо програму. Після успішної компіляції перейдемо в режим моделювання.
4) Відриємо графічне вікно і додамо в ньому сигнали, що перевіряються. Запустимо проект на моделювання. Вкажемо відповідь НІ на запитання, чи хочете Ви закінчити роботу з симулятором. Перевіремо отримані результати:

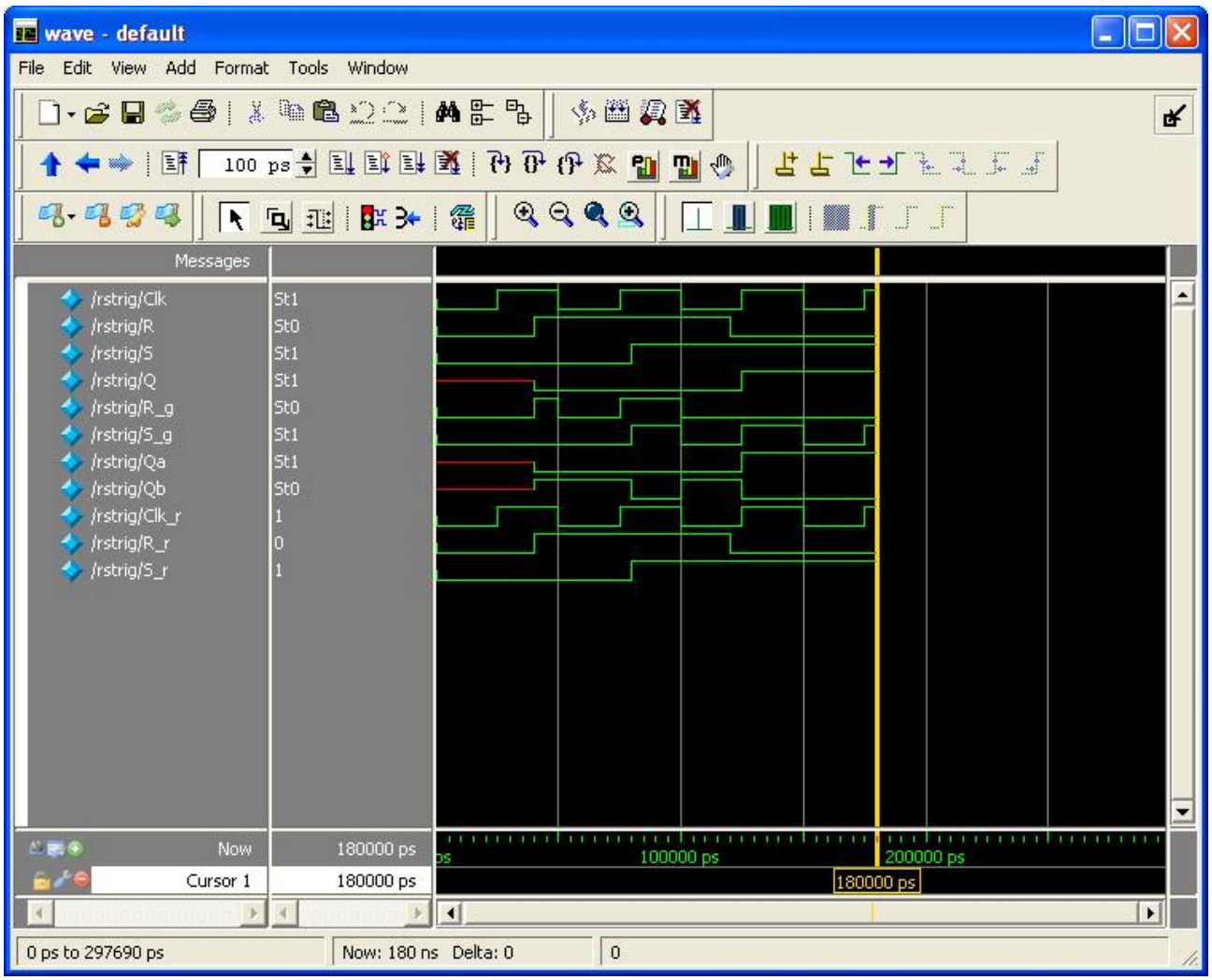


Рис. 17. Отримані результати

5) **Зверніть увагу** - в цій роботі не використовується файл з описом тестових векторів. Всі зміни вхідних сигналів задаються за допомогою оператора **initial**. Сам опис тригера виконано не на поведінковому рівні, а на структурному. Описано елементи **and** і **nor** з підключеними до них сигналами.
6) Закінчимо режим моделювання.

3. Самостійна робота

Завдання 1.

Реалізуйте послідовний пристрій і перевірте його роботу в середовищі ModelSim відповідно до заданого варіанту:

Варіант 1. 8-розрядний синхронний паралельний регістр з входами скидання і установки.

Варіант 2. 16-розрядний зсувний регістр з паралельним завантаженням. Напрямок зсуву задається логічним рівнем на відповідному вході.

Варіант 3. Перетворювач послідовного коду інтерфейсу RS-232 в паралельний код.

Варіант 4. Кільцевій 16-розрядний регістр зсуву. Вхідні дані - 8-розрядні. Напрямок та величина зсуву задаються зовнішніми сигналами.

Варіант 5. Блок пам'яті типу FIFO - вісім 8-розрядних слів.

Варіант 6. Блок ОЗП - шістнадцять 8-розрядних слів.

Завдання 2.

Реалізуйте лічильник і перевірте його роботу в середовищі ModelSim відповідно до заданого варіанту:

Варіант 1. Двійково-десятковий лічильник з паралельним завантаженням.

Варіант 2. Двійковий лічильник по модулю 16 з можливістю реверсивного рахунку. Напрямок рахунку задається логічним рівнем на відповідному вході.

Варіант 3. 4-розрядний лічильник Джонсона. Послідовність зміни станів даного лічильника наступна: 0h - 1h - 3h - 7h - Fh - Eh - Ch - 8h -0h.

Варіант 4. 3-розрядний лічильник-формувач коду Грея. Послідовність станів даного лічильника наступна: 0 - 1 - 3 - 2 - 6 - 7 - 5 - 4 - 0.

Варіант 5. 3-х канальний генератор імпульсних послідовностей із змінною фазою.

Варіант 6. Генератор ШІМ-сигналу. Коефіцієнт заповнення вихідного сигналу задається вхідним 4-розрядних кодом.

Лабораторна робота № 5
«Дослідження арифметичних пристроїв».

1. Теоретичні відомості.

Короткі теоретичні відомості з мови Verilog (продовження).

Підключення модулів у файлі верхнього рівня ієрархії.

Основною проектною одиницею на мові Verilog є модулі. При створенні складних пристроїв зручно використовувати ієрархічний підхід до їх побудови, тобто до складу одного пристрою може входити кілька модулів. Прийшов час навчитися підключати готові модулі до спроектованої схеми і пов'язувати їх між собою. Зазвичай, вихід одного модуля з'єднується з входом іншого провідником, тому зміни вихідного сигналу джерела будуть негайно передаватися на вхід приймача. Можна з'єднувати порти вводу/виводу модуля з входами пристрою. Зв'язки між модулями можуть бути типів **wire** та **reg**.

Синтаксис підключення модуля має наступний вигляд:

module_name instance_name (... io ports ...);

де ***module_name*** - ім'я викликаємого модуля, який необхідно підключити. Можна, звичайно, підключити в одному проекті одночасно декілька однакових модулів, просто вказавши для них різні імена - ***instance_name***. Тут доречно згадати механізм оголошення змінних в мовах програмування. Спочатку записується тип змінної, а потім - ім'я змінної. Так і при підключенні модуля - спочатку записується ім'я модуля, який необхідно підключити, а потім ім'я конкретного екземпляра цього модуля в файлі, де він використовується. Далі, в круглих дужках, зазначається підключення сигналів (типу **reg** або **wire**) до портів вводу/виводу модуля.

Розглянемо наступний приклад:

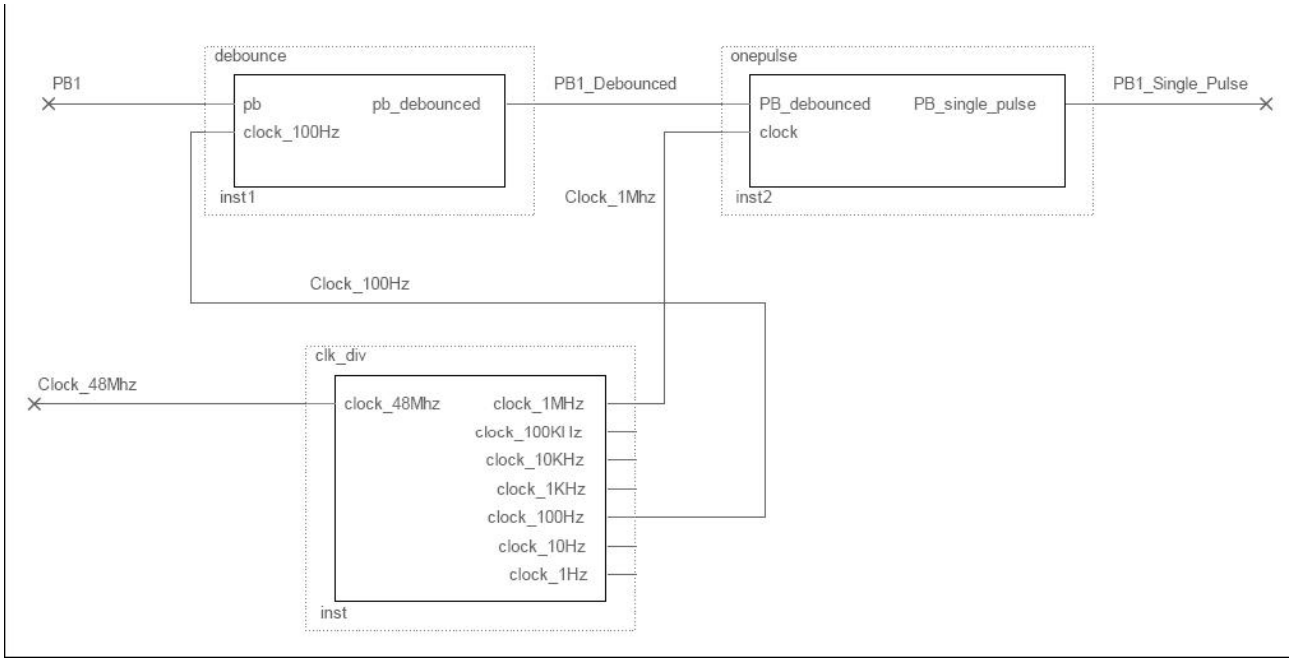


Рис. 18. Ієрархічне підключення модулів

Припустимо, спроектований пристрій містить в собі три модулі (мал.18.): **debounce**, **onepulse**, **clk_div**, кожен з яких описаний у відповідному файлі на мові Verilog (назва файлу має збігатися з ім'ям модуля). Відповідні вхідні і вихідні порти цих модулів зображені на малюнку. Вхід модуля зображується з лівого боку блоку, вихід - з правого. У

пристрої необхідно об'єднати між собою ці модулі в файлі верхнього рівня ієрархії. З'єднання модулів здійснюються за допомогою сигналів (типу **reg**) **-Clock_100Hz**, **Clock_1MHz**, **PB1_Debounced**. Файл опису пристрою буде виглядати наступним чином:

```
module hierarch (Clock_48MHz, PB1, PB1_Single_Pulse);
input Clock_48MHz, PB1;
output PB1_Single_Pulse;


reg Clock_100Hz, Clock_1MHz, PB1_Debounced;

debounce debounce1 (PB1, Clock_100Hz, PB1_Debounced);
clk_div clk_div1 (Clock_48MHz, Clock_1MHz, Clock_100Hz);
onepulse onepulse1 (PB1_Debounced, Clock_1MHz, PB1_Single_Pulse);
endmodule
```

Оператори блокуючого і неблокуючого присвоювання

У мові Verilog існують два типи операторів присвоювання: блокуючий **blocking** (=), і неблокуючий **nonblocking** (<=).

Для того щоб зрозуміти різницю між даними типами операторів присвоювання, необхідно розглянути принцип роботи Verilog симулятора. У реальному пристрої (наприклад - цифрової схеми), яка моделюється за допомогою мови Verilog, події можуть відбуватися одночасно - при зміні вхідного сигналу у всіх елементах, пов'язаних з ним, починаються відповідні процеси. Вони протікають одночасно і призводять до відповідних змін вихідних сигналів. Моделююча програма не може виконувати події одночасно, вона створює списки подій, які будуть виконуватися послідовно. Коли всі події зі списку виконані, симулятор переходить до обробки наступного часового кроку - збільшує поточний час моделювання на часовий інтервал (другий параметр в директиві **`timescale**) і виконує обробку списку подій, які повинні відбутися на даному етапі. Розглянуті події відбуваються «одночасно» - тобто на одному часовому кроці моделювання.
Припустимо, є наступний набір команд:

```
always @ (Posedge CLK) a = b;
always @ (Posedge CLK) b = a;
```

В даному прикладі змінні a і b - однорозрядні регістри, які до моменту появи зростаючого фронту тактового сигналу CLK зберігали наступне значення: a == 0 і b == 1. Яке ж значення матимуть ці змінні після виконання операції присвоєння? Це залежить від того, в якій послідовності операції присвоювання потраплять в список обробки. Тобто, поведінка такої конструкції залежить від порядку проходження операторів в програмі. Це означає, що або обидві змінні будуть дорівнювати 0, або обидві дорівнюватимуть 1 (в нашому прикладі - 1). Операція блокуючого присвоювання (=) блокує виконання інших послідовних операцій до тих пір, поки вона не буде виконана. Використання операції блокуючого присвоєння в блоках , що мають виконуватись паралельно, небажано. Але, якщо в блоці необхідно забезпечити послідовне виконання операторів, слід використовувати даний тип присвоєння.

Наступний фрагмент програми гарантує обнулення змінних a і b по передньому фронту сигналу CLK:

```
always @ (Posedge CLK)
begin
```

a = 0;
b = a;
end

Якщо в попередньому прикладі використовувати оператор неблокуючого присвоювання (<=), то поведінка пристрою зміниться:

always *@ (Posedge CLK) a <= b;*
always *@ (Posedge CLK) b <= a;*

В даному випадку, в списку подій, що виконуються на поточному часовому кроці моделювання після зміни сигналу CLK, обидві операції будуть розташовані як ті, що виконуються паралельно. Тобто, змінні a і b обміняються своїми значеннями. Після проходження зростаючого фронту сигналу CLK значення змінних будуть наступними: a == 1 b == 0. Послідовність запису a <= b; b <= a; або b <= a; a <= b; в даному випадку не має значення, тому що події моделюються одночасно.

Системні функції мови Verilog

Мова Verilog надає програмісту більше доступних можливостей для управління та аналізу результатів моделювання. Ці можливості реалізовані у вигляді системних функцій симулятора. Слід пам'ятати, що при синтезі системні функції ігноруються.

Завдяки наявності механізму PLI, що забезпечує підключення додаткової програми (написаної або користувачем, або іншою стороною) до тестових файлів, число системних функцій і завдань, які можуть виконуватися за їх допомогою, суттєво зростає. Основне призначення системних функцій - збір і аналіз інформації, взаємодія з операційною системою. Ознакою системної функції є знак **$**. Наведемо найбільш популярні системні функції:

$finish - завершення процесу моделювання;

$stop – призупинка моделювання та перехід в інтерактивний режим;

$display, **$write** - пересилка даних за допомогою функції **stdout** (дані дублюються в файлі протоколу). Поведінка такої функції відповідає функції **printf** мови С (видача форматованого рядку з підтримкою додаткових форматів, наприклад,% b -бінарний), або процедурі **write** мови Паскаль з розділеними **«,»** аргументами. Функція **$display** завершує видачу рядка командою «новий рядок»;

$monitor - відстежує зміни аргументів і в кінці кожного часового кроку моделювання відображає поточні результати (якщо були виявлені зміни значень сигналів). Формат даної функції - як у **$display**;

$readmemb, **$readmemh** - забезпечують зчитування даних (в двійковому або шістнадцятковому форматі) з файлу в пам'ять пристрою. Формат файлу дуже простий - в кожному рядку вказується слово заданої розрядності, або покажчик адреси (конструкція **@ <адреса завантаження>**). Дану функцію зручно застосовувати для моделювання ПЗУ;

$system - виконує команду операційної системи (виклик функції мови С **system ()**).

Для виконання файлових операцій використовуються функції **$fopen**, **$fclose**, **$fwrite**, **$fmonitor.** Вони дозволяють зберігати дані, що передаються, в файлах. Функції **$dumpfile**, **$dumpvars** дозволяють записувати зміни сигналів модуля, що тестується, для всього проекту або його складових частин, у файлі спеціального формату для подальшого аналізу. Це дуже корисні й ефективні функції для обробки даних.

Функція **$time** - повертає значення поточного часу моделювання.

Це невеликий перелік стандартних функцій. Їх повний список вказано в документації до моделюючої програми.

Арифметичні пристрої.

До арифметичним пристроїв відносяться перетворювачі, що виконують арифметичні дії (додавання, віднімання, множення) над вхідними даними.

На малюнку 19 представлена схема повного однорозрядного суматора і його графічне зображення.

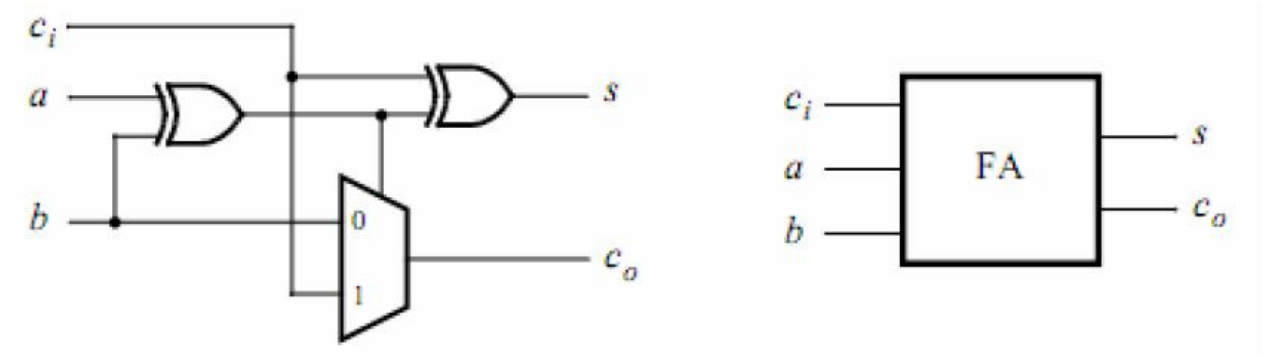


Рис. 19**.** Схема повного однорозрядного суматора і його графічне зображення.

Призначення сигналів сумматора наступне: a і b - вхідні сигнали (складові), Ci - вхід переносу з попереднього розряду, s - вихідний сигнал (сума), Co - вихід переносу до наступного розряду. Таблиця дійсності однорозрядного сумматора виглядає наступним чином:

Таблиця істинності

a	**b**	**Ci**	**Co**	**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

Для створення сумматоров більшої розрядності використовується каскадне з'єднання однорозрядних суматорів. Наприклад, на малюнку 20 показано повний чотирьохрозрядний суматор з послідовним переносом, що складається з чотирьох однорозрядних суматорів.

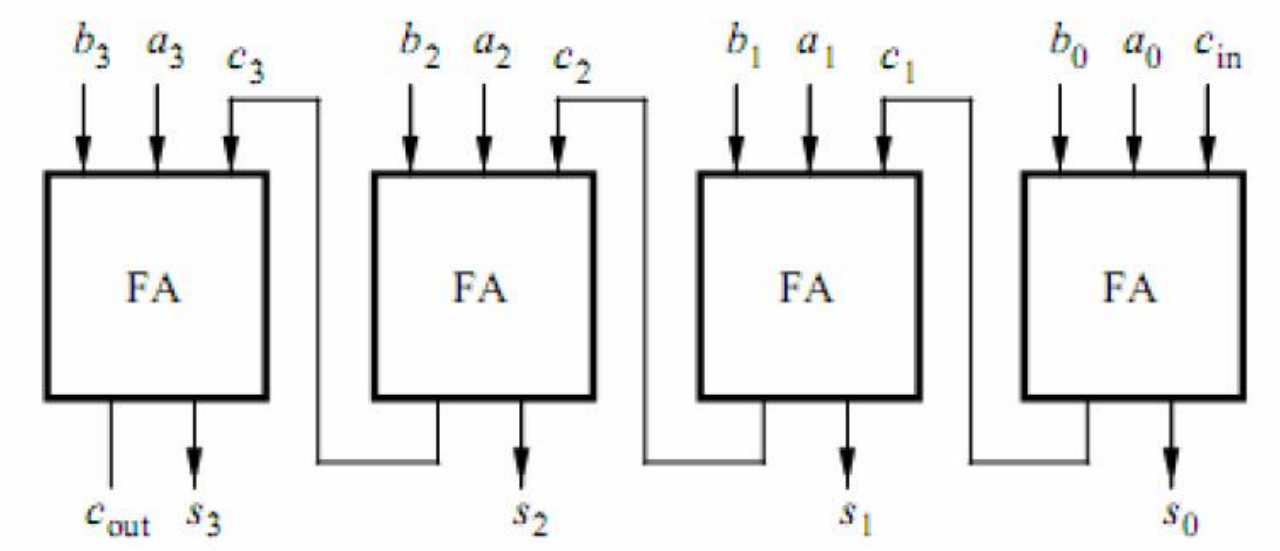


Рис. 20. Повний чотирьохрозрядний суматор з послідовним переносом.

Приклад виконання операції множення двох 4-розрядних чисел в двійковому вигляді показано на малюнку 21. На малюнку 22 показана структурна схема реалізації такого перемножувача. Він складається з логічних елементів «І» і повних однорозрядних суматорів (FA).

				a_3	a_2	a_1	a_0
				x b_3	b_2	b_1	b_0
				a_3b_0	a_2b_0	a_1b_0	a_0b_0
			a_3b_1	a_2b_1	a_1b_1	a_0b_1	
		a_3b_2	a_2b_2	a_1b_2	a_0b_2		
	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Рис. 21. Операція множення двох 4-розрядних чисел.

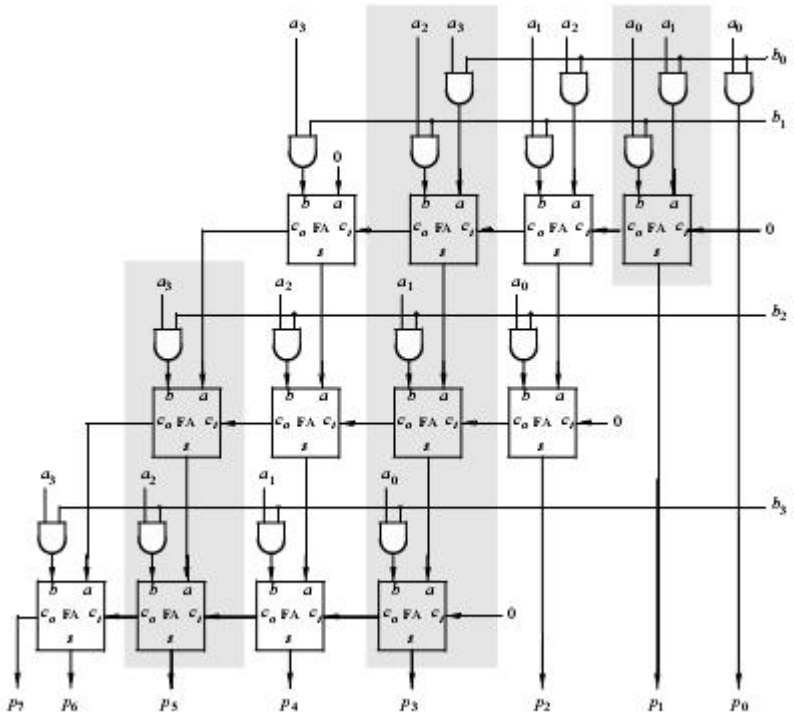


Рис. 22. Структурна схема перемножувача двох 4-розрядних чисел.

Згадані два компонента використовуються як базові для створення різних арифметичних або арифметико-логічних пристроїв (АЛП).

2. Порядок виконання роботи.

У лабораторній роботі ми створимо проект, що описує роботу повного чотирирозрядного суматора, зображеного на малюнку 3. Для головного модуля використовується структурний опис. Для перевірки сумматора створимо еталонну модель, яка використовує поведінковий опис, і тестовий файл (test-bench). Для цього:

1. Створіть новий проект в середовищі ModelSim.
2. Створіть похідний файл суматора на структурному рівні:

```
module my_sum (Ain, Bin, Ci, Sout, Co);
  input Ain, Bin, Ci;
  output Sout, Co;

  wire [3: 0] Ain, Bin, Sout, C;
  wire Ci, Co;

bitsum sum1 (Ain [0], Bin [0], Sout [0], Ci, C [0]);
bitsum sum2 (Ain [1], Bin [1], Sout [1], C [0], C [1]);
bitsum sum3 (Ain [2], Bin [2], Sout [2], C [1], C [2]);
bitsum sum4 (Ain [3], Bin [3], Sout [3], C [2], C [3]);

assign Co = C [3];

endmodule
```

```
module bitsum (A, B, S, Cin, Cout);
  input A, B, Cin;
  output S, Cout;

  wire A, B, S, Res;
  wire c1, c2, Cin, Cout;

  xor (Res, A, B);
  and(C1, A, B);
  xor(S, Cin, Res);
  and(C2, Cin, Res);
  or(Cout, c1, c2);

endmodule
```

Зверніть увагу на підключення готових модулів в даному файлі.

3. Створіть похідний файл еталонного суматора на поведінковому рівні:

```
module ref_sum (Ain, Bin, Ci, Sout, Co);
  input Ain, Bin, Ci;
  output Sout, Co;

  wire [3: 0] Sout, Ain, Bin;
  reg [4: 0] S;

  always @ (Ain, Bin, Ci)
  S = Ain + Bin + Ci;

  assign Sout = S [3: 0];
  assign Co = S [4];

endmodule
```

3. Створіть тестовий файл для подачі вхідних сигналів і порівняння роботи двох модулів:

```
module test_sum; // Top Level Testbench

wire Ci, cm, cr;
wire [3: 0] Ain, Bin;
reg [3: 0] Ain_r, Bin_r;
reg Ci_r;
wire [3: 0] res_my, res_ref;

my_sum my_block (Ain, Bin, Ci, res_my, cm);
ref_sum  ref_block (Ain, Bin, Ci, res_ref, cr);

initial
begin
   $display( "\ T \ t Time Ain Bin Ci res_my cm res_ref cr");
   $monitor ($ time ,,,,, Ain ,,,,, Bin ,,,,, Ci ,,,,, res_my ,,,,,,,, cm ,,,,,,, res_ref ,,,, ,,, cr);
   # 400 $finish;
```

```
end

initial
begin
Ain_r = 1;
# 50 Ain_r = 5;
# 50 Ain_r = 1;
# 50 Ain_r = 5;
# 50 Ain_r = 1;
# 50 Ain_r = 5;
# 50 Ain_r = 1;
# 50 Ain_r = 5;
end

initial
begin
Bin_r = 2;
# 100 Bin_r = 10;
# 100 Bin_r = 2;
# 100 Bin_r = 10;
end

initial
begin
Ci_r = 1'b0;
# 200 Ci_r = 1'b1;
end

assign Ain = Ain_r;
assign Bin = Bin_r;
assign Ci = Ci_r;

endmodule
```

Зверніть увагу на підключення модулів в даному файлі і використання системних функцій.

4. Скомпілюйте всі програми. Після успішної компіляції перейдіть в режим моделювання. В якості основного файлу для моделювання вкажіть тестовий файл.

Відрийте графічне вікно і додайте до нього сигнали, що перевіряються. Запустіть проект на моделювання. Дайте відповідь НІ на запитання, чи хочете Ви закінчити роботу з симулятором. Перевірьте отримані результати:

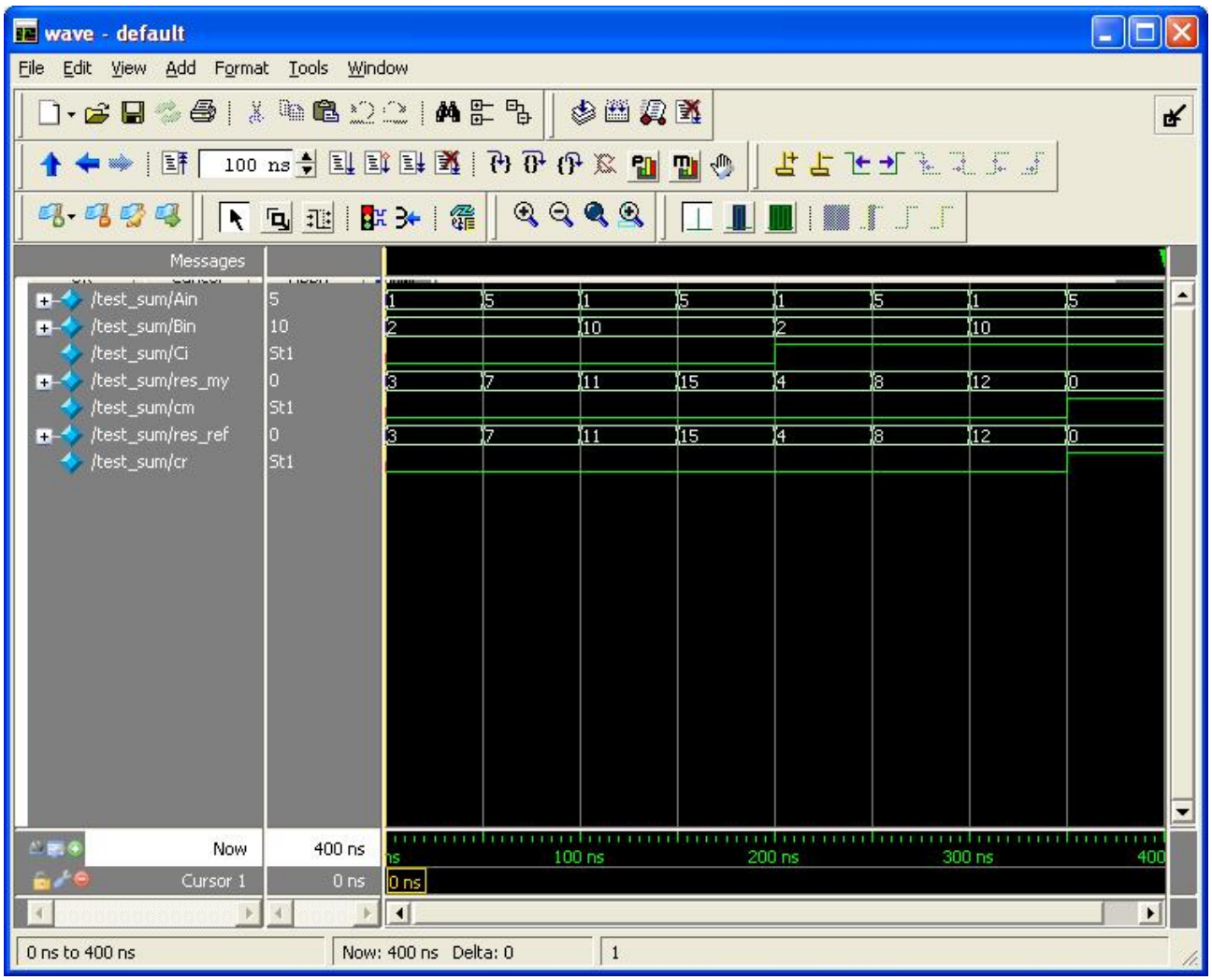


Рис. 23. Отримані результати.

Поясніть отриманий результат. Вийдіть з режиму моделювання.

3. Самостійна робота.

Спроектувати такі арифметико-логічні пристрої (вхідні дані - восьмирозрядні):
Варіант 1. АЛУ з функціями: A + B, AxorB, A-1, A-B.
Варіант 2. АЛУ з функціями: A + B, A * B, A-1, B.
Варіант 3. АЛУ з функціями: A + (A + B), A, A + 1, A-B-1.
Варіант 4. АЛУ з функціями: not (A + B), not (A * B), A + B + 1, (A + notB) +1.
Варіант 5. АЛУ з функціями: AxorB, A * B + (A + notB), A, B.
Варіант 6: Спроектувати помножувач з накопиченням (операція MAC).

Для перевірки роботи пристрою створити тестовий файл.

Лабораторна робота № 6
«Дослідження кінцевих автоматів».

1. Теоретичні відомості.

Короткі теоретичні відомості з теорії кінцевих автоматів.

У даній лабораторній роботі ми познайомимся з принципом роботи і способом реалізації тактуемих синхронних кінцевих автоматів. «Кінцевий автомат» - це загальна назва послідовних схем. Слово «тактуемий» вказує на той факт, що елементи пам'яті в кінцевому автоматі (тригери) мають тактовий вхід. Слово «синхронний» означає, що всі тригери використовують один і той же тактовий сигнал. Стан такого кінцевого автомата змінюється тільки в момент часу, коли в тактовому сигналі відбувається перехід з одного логічного рівня на інший, або, як кажуть, на черговому «такті».

На малюнку 24 наведена загальна структура тактуемого синхронного кінцевого автомата.

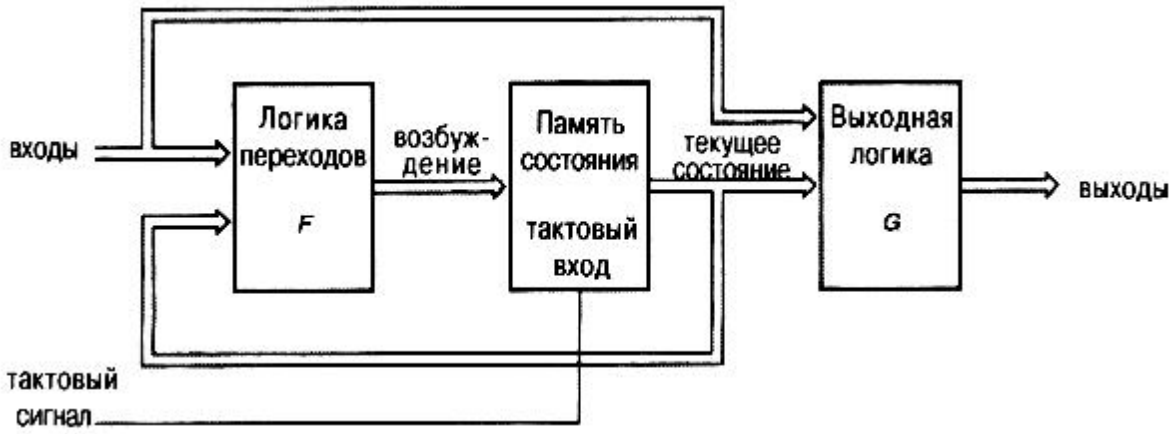


Рис. 24**.** Структура тактуемого синхронного кінцевого автомата Мілі.

Пам'ять станів являє собою набір з n тригерів, в яких зберігається поточний стан автомата. Всього є 2^n різних станів. Всі тригери підключені до загального джерела тактового сигналу, який дозволяє їм змінювати стан на кожному такті тактового сигналу.

Наступний стан кінцевого автомата визначається логікою переходів F і є функцією поточного стану і вхідного впливу. Вихідні сигнали визначаються вихідний логікою G і також залежать від поточного стану і вхідного впливу. Обидва блоки F і G є строго комбінаційними схемами. Послідовна схема, вихід якої залежить як від стану, так і від входу, називається автоматом Мілі. У деяких пристроях вихід залежить тільки від стану. Така схема називається автоматом Мура. Її загальна структура приведена на малюнку 25.

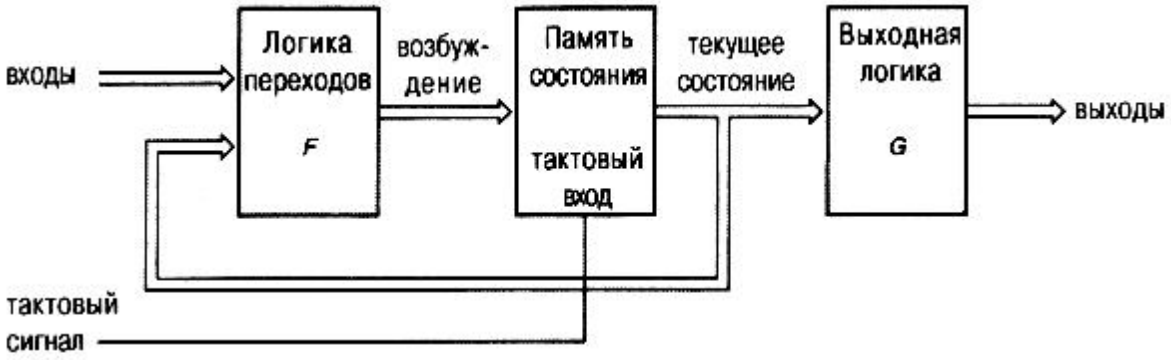


Рис. 25. Структура тактуемого синхронного кінцевого автомата Мура.

Вочевидь, що єдина відмінність між цима двома моделями кінцевих автоматів полягає в тому, як формуються вихідні сигнали. На практиці, багато кінцевих автоматів можуть мати виходи типу Мілі і виходи типу Мура, тобто мати змішану структуру.

2. Порядок виконання роботи.

У лабораторній роботі ми створимо проект, що описує роботу кінцевого автомата, представленого на малюнку 26.

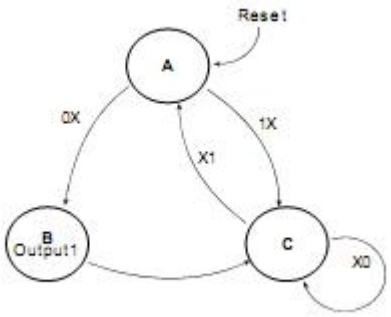


Рис. 26. Структура реалізованого кінцевого автомата.

Автомат має три стани - A, B і C. Умовою переходу з одного стану в інший є значення, яке приймається вхідними сигналами Input1 і Input2. Вихідний сигнал Output1 встановлюється в стан логічної «1» тільки тоді, коли кінцевий автомат знаходиться в стані B. Опис похідного модуля проекту виконано на поведінковому рівні.

1. Створіть новий проект в середовищі ModelSim.
2. Створіть похідний файл кінцевого автомату на поведінковому рівні:

```
module state_mach (clk, reset, Input1, Input2, output1);
 input clk, reset, Input1, Input2;
 output output1;
 reg output1;
 reg [1: 0] state;
  / * Кодування станів автомата * /
  parameter [1: 0] state_A = 0, state_B = 1, state_C = 2;

  always @ (posedge clk or posedge reset)
  begin
   if (reset)
     state = state_A;
```

```
else
 / * Визначення наступного стану автомата * /
 case (state)
  state_A:  if (Input1 == 0)
    state = state_B;
  else
    state = state_C;
  state_B: state = state_C;
  state_C: if (Input2) state = state_A;
 default: state = state_A;
     endcase
   end
   / * Стан виходу кінцевого автомата * /
   always @ (state)
   begin
     case (state)
       state_A: output1 = 0;
       state_B: output1 = 1;
       state_C: output1 = 0;
       default: output1 = 0;
     endcase
   end
 endmodule
```

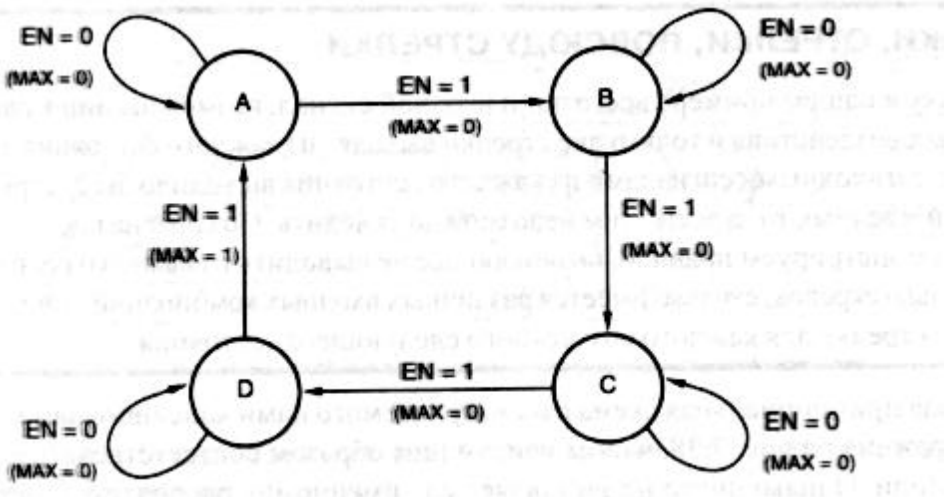
3. Створіть тестовий файл для подачі вхідних сигналів.
4. Скомпілюйте всі програми. Після успішної компіляції перейдіть в режим моделювання. В якості основного файлу для моделювання вкажіть тестовий файл.
5. Відрийте графічне вікно і додайте в нього сигнали, що перевіряються. Запустіть проект на моделювання. Дайте відповідь НІ на запитання, чи хочете Ви закінчити роботу з симулятором. Перевірьте отримані результати.

3. Самостійна робота.

Завдання 1.

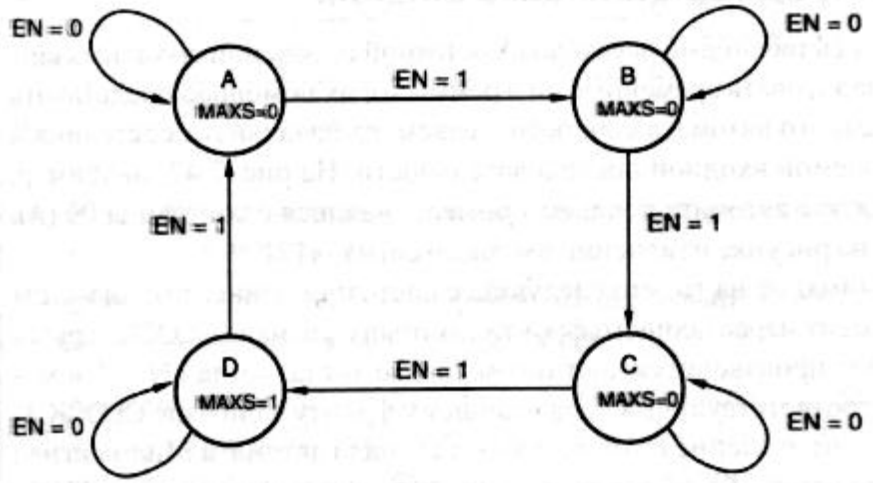
Спроектувати наступні кінцеві автомати (представлені у вигляді діаграм переходів):

Варіант 1. Схема кінцевого автомата наведена на малюнку.



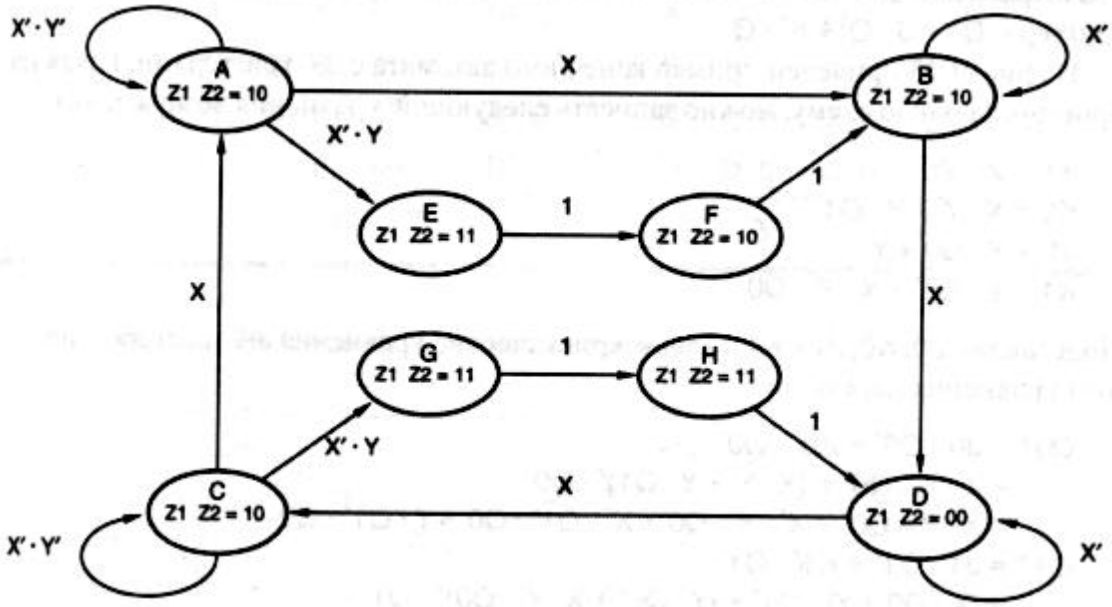
Він має 4 стани - A, B, C і D. Вхідний сигнал EN і вихідний сигнал MAX. Вихідний сигнал залежить від поточного стану кінцевого автомата і вхідного сигналу.

Варіант 2. Схема кінцевого автомата наведена на малюнку.



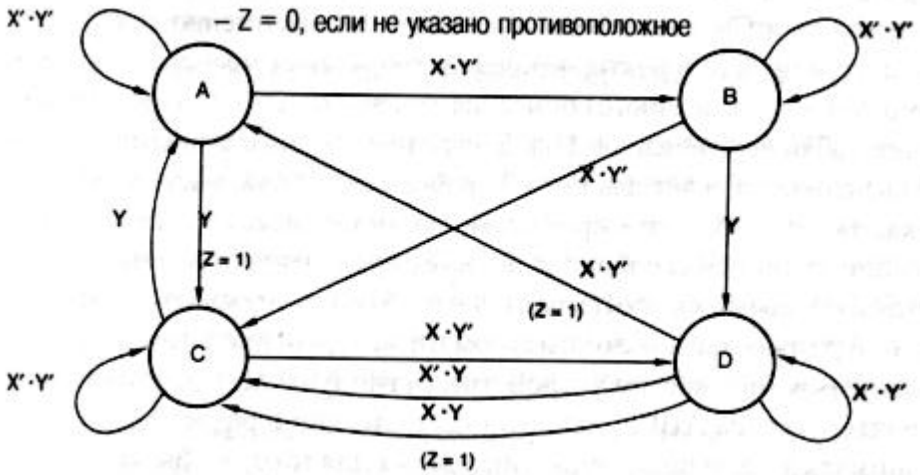
Він має 4 стани - A, B, C і D. Вхідний сигнал EN і вихідний сигнал MAX. Вихідний сигнал залежить тільки від поточного стану кінцевого автомата.

Варіант 3. Схема кінцевого автомата наведена на малюнку.



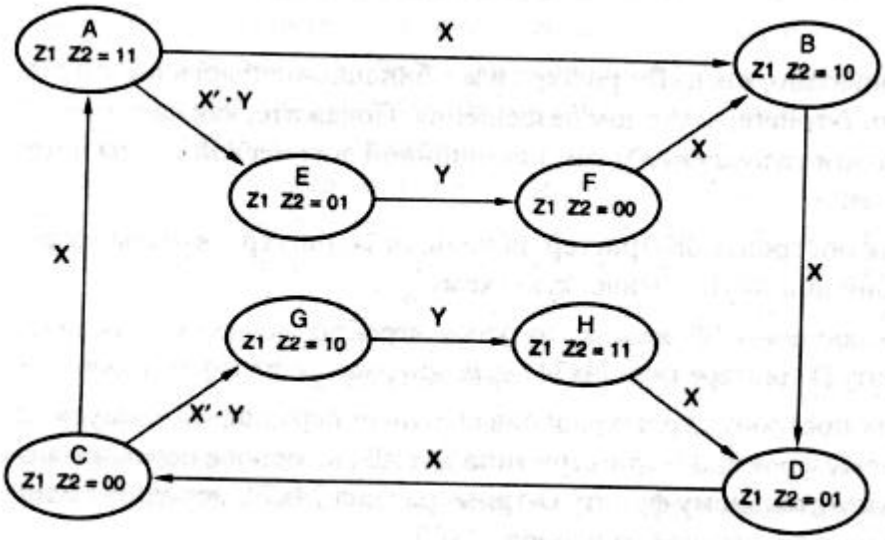
Він має 8 станів - A, B, C, D, E, F, G і H. Вхідні сигнали X і Y, вихідні сигнали Z1 і Z2. Вихідні сигнали залежать тільки від поточного стану кінцевого автомата.

Варіант 4. Схема кінцевого автомата наведена на малюнку.



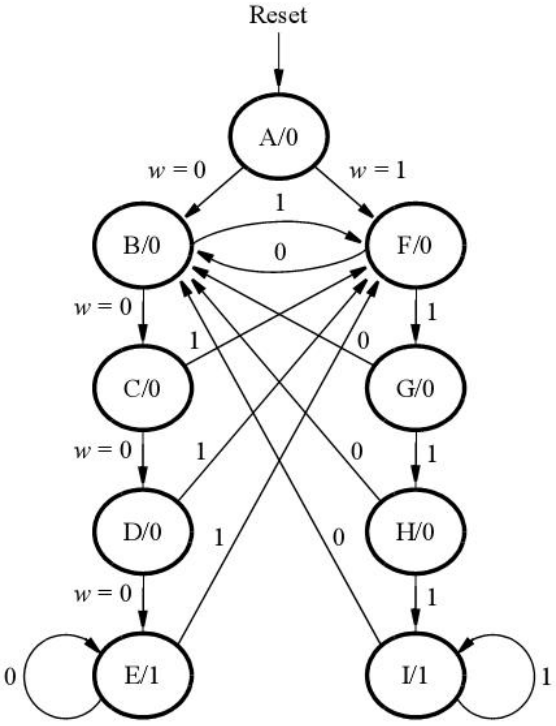
Він має 4 стани - A, B, C і D. Вхідні сигнали X і Y, вихідний сигнал Z. Вихідний сигнал залежить від поточного стану кінцевого автомата і вхідних сигналів.

Варіант 5. Схема кінцевого автомата наведена на малюнку.



Він має 8 станів - A, B, C, D, E, F, G і H. Вхідні сигнали X і Y, вихідні сигнали Z1 і Z2. Вихідні сигнали залежать тільки від поточного стану кінцевого автомата.

Варіант 6: Схема кінцевого автомата наведена на малюнку.



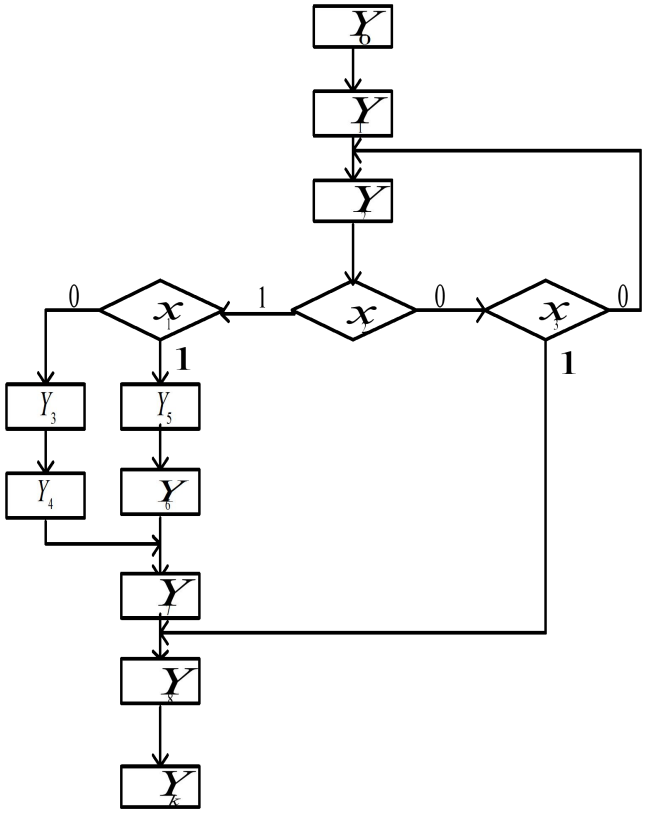
Він має 9 станів - A, B, C, D, E, F, G, H і I. Вхідні сигнали Reset і w, вихідний сигнал OUT1. Вихідний сигнал залежить тільки від поточного стану кінцевого автомата.

Для перевірки роботи пристрою необхідно створити тестовий файл.

Завдання 2.

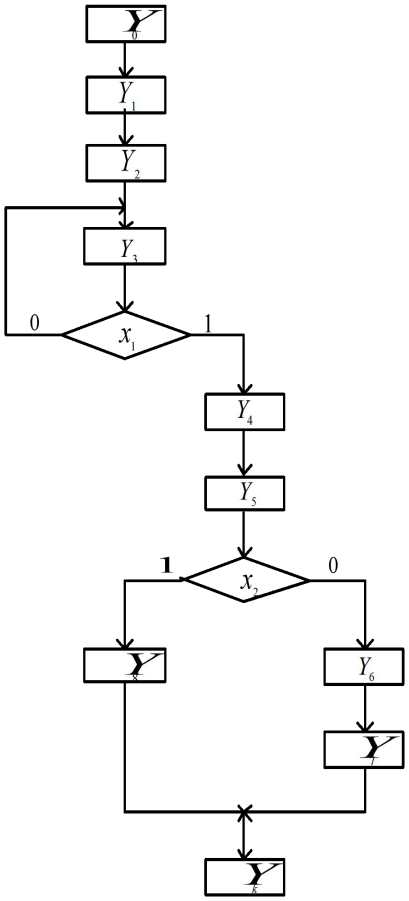
Спроектувати мікропрограмний автомат, використовуючи одну з моделей кінцевих автоматів. Мікропрограмний автомат представлений у вигляді блок-схеми.

Варіант 1. Блок-схема мікропрограмного автомата наведена нижче:



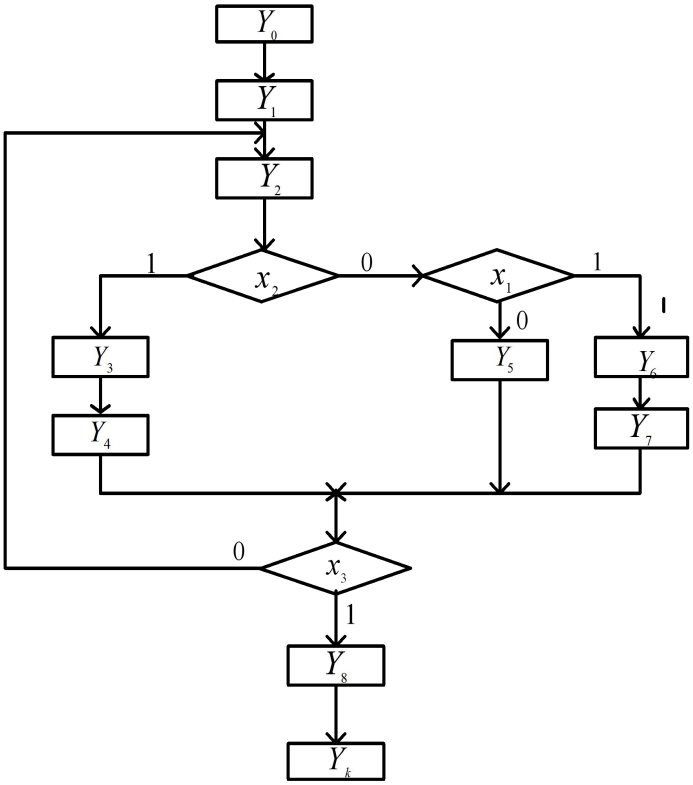
Y - стани автомату, x - вхідні сигнали. Вихідні сигнали і їх значення задайте самостійно.

Варіант 2. Блок-схема мікропрограмного автомата наведена нижче:



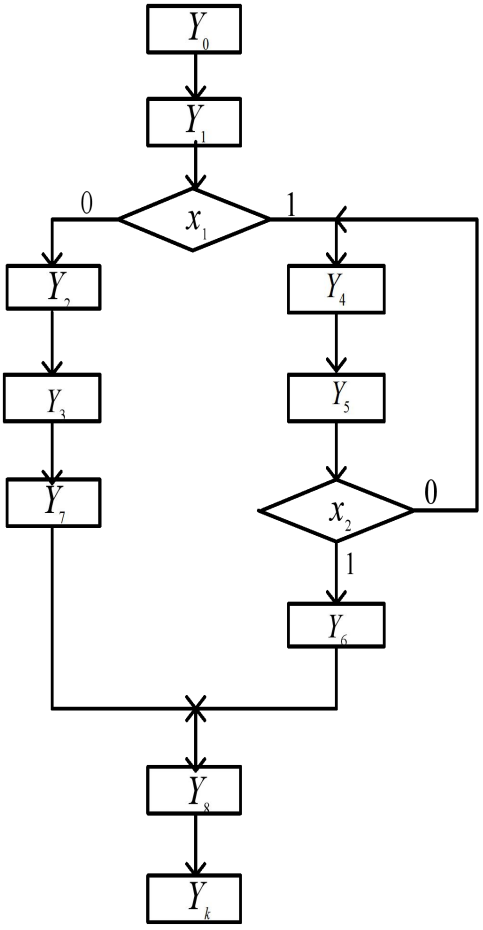
Y - стани автомату, x - вхідні сигнали. Вихідні сигнали і їх значення задайте самостійно.

Варіант 3. Блок-схема мікропрограмного автомата наведена нижче:



Y - стани автомату, x - вхідні сигнали. Вихідні сигнали і їх значення задайте самостійно.

Варіант 4. Блок-схема мікропрограмного автомата наведена нижче:



Y - стани автомату, x - вхідні сигнали. Вихідні сигнали і їх значення задайте самостійно.

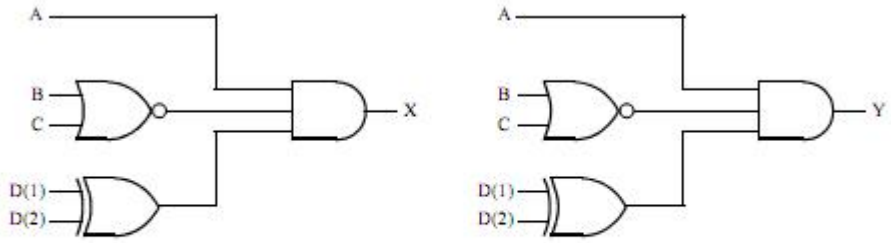
Для перевірки роботи пристрою необхідно створити тестовий файл.

Лабораторна робота № 7
«Дослідження багатофункціональних пристроїв».

1. Теоретичні відомості.

Приклади реалізації деяких пристроїв мовою Verilog.

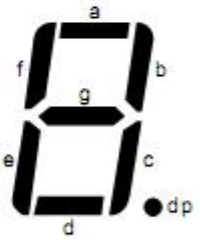
Комбінаційні схеми:




```
module gatenetwork(A, B, C, D, X, Y);
   input A;
   input B;
   input C;
   input [2:1] D;
   output X, Y;
   reg Y;
        // concurrent assignment statement
   wire X = A & ~(B|C) & (D[1] ^ D[2]);
        /* Always concurrent statement- sequential execution inside */
   always @( A or B or C or D)
        Y = A & ~(B|C) & (D[1] ^ D[2]);

endmodule
```

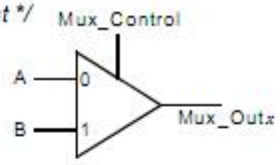
Дешифратор семисегментного індикатору:

```
module DEC_7SEG(Hex_digit, segment_a, segment_b, segment_c,
                segment_d, segment_e, segment_f, segment_g);
    input [3:0] Hex_digit;
    output segment_a, segment_b, segment_c, segment_d;
    output segment_e, segment_f, segment_g;
    reg [6:0] segment_data;

    always @(Hex_digit)
                /* Case statement implements a logic truth table using gates*/
     case (Hex_digit)
            4'b 0000:  segment_data = 7'b 1111110;
            4'b 0001:  segment_data = 7'b 0110000;
            4'b 0010:  segment_data = 7'b 1101101;
            4'b 0011:  segment_data = 7'b 1111001;
            4'b 0100:  segment_data = 7'b 0110011;
            4'b 0101:  segment_data = 7'b 1011011;
            4'b 0110:  segment_data = 7'b 1011111;
            4'b 0111:  segment_data = 7'b 1110000;
            4'b 1000:  segment_data = 7'b 1111111;
            4'b 1001:  segment_data = 7'b 1111011;
            4'b 1010:  segment_data = 7'b 1110111;
            4'b 1011:  segment_data = 7'b 0011111;
            4'b 1100:  segment_data = 7'b 1001110;
            4'b 1101:  segment_data = 7'b 0111101;
            4'b 1110:  segment_data = 7'b 1001111;
            4'b 1111:  segment_data = 7'b 1000111;
            default:   segment_data = 7'b 0111110;
     endcase
```



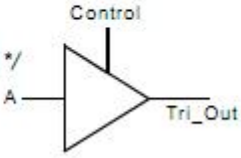
Три способи опису мультиплексору:

```
        /* Multiplexer example shows three ways to model a 2 to 1 mux */
module multiplexer(A, B, mux_control, mux_out1, mux_out2, mux_out3);
  input A;                  /* Input Signals and Mux Control  */
  input B;
  input mux_control;
  output mux_out1,mux_out2, mux_out3;
  reg mux_out2, mux_out3;
                            /* Conditional Continuous Assignment Statement */
                            /*  works like an IF - ELSE */
  wire mux_out1 = (mux_control)? B:A;
                            /* If statement inside always statement */
    always @(A or B or mux_control)
        if (mux_control)
            mux_out2 = B;
        else
            mux_out2 = A;
                            /* Case statement inside always statement */
    always @(A or B or mux_control)
        case (mux_control)
            0: mux_out3 = A;
            1: mux_out3 = B;
            default: mux_out3 = A;
        endcase
endmodule
```



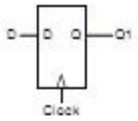
Вихідний буферний елемент з трьома станами:

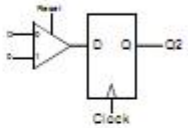
```
module tristate (a, control, tri_out);
   input a, control;
   output tri_out;
   reg tri_out;
   always @(control or a)
        if (control)
             /* Assignment of Z value generates a tri-state output */
             tri_out = 1'bZ;
        else
             tri_out = a;
endmodule
```

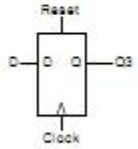


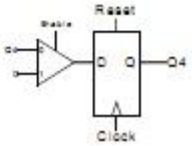
Різні варіанти реалізації D-тригеру:

```
module DFFs(D, clock, reset, enable, Q1, Q2, Q3, Q4);
input D;
input clock;
input reset;
input enable;
output Q1, Q2, Q3, Q4;
reg Q1, Q2, Q3, Q4;
                /* Positive edge triggered D flip-flop */
     always @(posedge clock)
          Q1 = D;
                /* Positive edge triggered D flip-flop */
                /*     with synchronous reset */
     always @(posedge clock)
          if (reset)
             Q2 = 0;
          else
             Q2 = D;
                /* Positive edge triggered D flip-flop */
                /*     with asynchronous reset */
     always @(posedge clock or posedge reset)
          if (reset)
             Q3 = 0;
          else
             Q3 = D;
                /* Positive edge triggered D flip-flop */
                /* with asynchronous reset and enable */
     always @(posedge clock or posedge reset)
          if (reset)
             Q4 = 0;
          else if (enable)
             Q4 = D;
endmodule
```









Восьмизарядний лічильник:

```
module counter(clock, reset, max_count, count);
     input clock;
     input reset;
     input [7:0] max_count;
     output [7:0] count;
     reg [7:0] count;
               /* use positive clock edge for counter */
     always @(posedge clock or posedge reset)
          begin
               if (reset)
                    count = 0;                  /* Reset  Counter */
               else if (count < max_count)      /* Check for maximum count */
                    count = count + 1;          /* Increment Counter */
               else
                    count = 0;                  /*  Counter set back to 0*/
          end
     endmodule
```

Приклад реалізації кінцевого автомату і його діаграма станів:

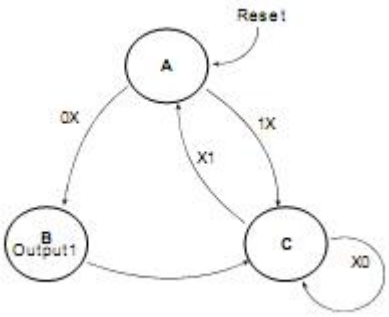
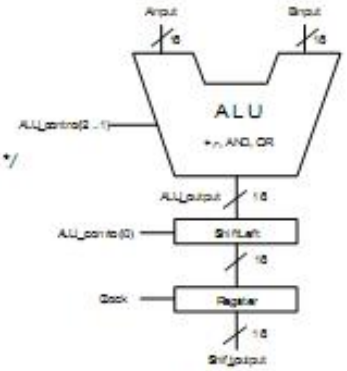


Рис. 27. Кінцевий автомат.

```
module state_mach (clk, reset, input1, input2 ,output1);
    input clk, reset, input1, input2;
    output output1;
    reg output1;
    reg [1:0] state;


            /* Make State Assignments */
    parameter [1:0] state_A = 0, state_B = 1, state_C = 2;

always@(posedge clk or posedge reset)
    begin
        if (reset)
            state = state_A;
        else
            /* Define Next State Transitions using a Case */
            /* Statement based on the Current State */
            case (state)
                state_A:
                    if (input1==0)
                        state = state_B;
                    else
                        state = state_C;
                state_B:
                    state = state_C;
                state_C:
                    if (input2) state = state_A;
                default:  state = state_A;
            endcase
    end
            /* Define State Machine Outputs */
always @(state)
    begin
        case (state)
            state_A: output1 = 0;
            state_B: output1 = 1;
            state_C: output1 = 0;
            default:  output1 = 0;
        endcase
    end
endmodule
```

АЛУ, яке виконує операції додавання, віднімання, кон'юнкції та диз'юнкції, з вихідним регістром зсуву:

```
module ALU ( ALU_control, Ainput, Binput, Clock, Shift_output);
    input [2:0] ALU_control;
    input [15:0] Ainput;
    input [15:0] Binput;
    input Clock;
    output[15:0] Shift_output;
    reg [15:0] Shift_output;
    reg [15:0] ALU_output;

        /* Select ALU Arithmetic/Logical Operation */
always @(ALU_control or Ainput or Binput)
    case (ALU_control[2:1])
        0: ALU_output = Ainput + Binput;
        1: ALU_output = Ainput - Binput;
        2: ALU_output = Ainput & Binput;
        3: ALU_output = Ainput | Binput;
        default: ALU_output = 0;
    endcase

        /* Shift bits left using shift left operator if required and load register */
always @(posedge Clock)
    if (ALU_control[0]==1)
        Shift_output = ALU_output << 1;
    else
        Shift_output = ALU_output;
endmodule
```



Приклад виклику раніше створених блоків у файлі верхнього рівня ієрархії:

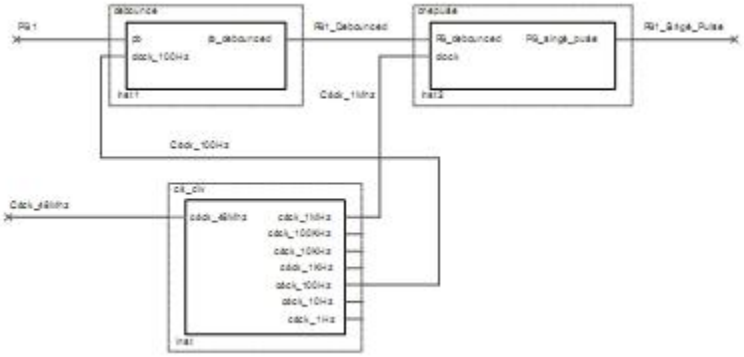


Рис. 28. Файл верхнього рівня ієрархії.

```
module hierarch(Clock_48MHz, PB1, PB1_Single_Pulse);
 input Clock_48MHz, PB1;
 output PB1_Single_Pulse;
            /* Declare internal interconnect signals */
 reg Clock_100Hz, Clock_1MHz, PB1_Debounced;

            /* declare and connect all  three modules in the hierarchy */
  debounce debounce1( PB1, Clock_100Hz, PB1_Debounced);

  clk_div clk_div1( Clock_48MHz, Clock_1MHz, Clock_100Hz);

  onepulse onepulse1( PB1_Debounced, Clock_100Hz, PB1_Single_Pulse);

endmodule
```

2. Самостійна робота.

Метою даної лабораторної роботи є розробка універсального обчислювального пристрою. Пристрій повинен складатися з двох основних блоків - арифметичного пристрою (АП) і математичного співпроцесора. АП призначене для виконання арифметичних операцій і операцій зсуву. Математичний співпроцесор - для виконання

операцій множення і ділення. Вхідними даними пристрою є чотири 16-розрядних числа. Пристрій повинен працювати в двох режимах - проста математика (одночасна робота з двома парами чисел), і комплексна математика - робота з двома комплексними числами (дійсна і уявна частини числа - 16-розрядні).

Нагадаємо правила роботи з комплексними числами.

1. Операція додавання (C = A + B):

$$C = (\mathrm{Re}[A] + \mathrm{Re}[B]) + (\mathrm{Im}[A] + \mathrm{Im}[B])$$

2. Операція віднімання (C = A - B):

$$C = (\mathrm{Re}[A] - \mathrm{Re}[B]) + (\mathrm{Im}[A] - \mathrm{Im}[B])$$

3. Операція множення комплексних чисел (C = A * B):

$$\mathrm{Re}[C] = \mathrm{Re}[A] * \mathrm{Re}[B] - \mathrm{Im}[A] * \mathrm{Im}[B]$$
$$\mathrm{Im}[C] = \mathrm{Im}[A] * \mathrm{Re}[B] + \mathrm{Re}[A] * \mathrm{Im}[B]$$

4. Операція ділення комплексних чисел (C = A / B):

$$\mathrm{Re}[C] = \frac{\mathrm{Re}[A] * \mathrm{Re}[B] + \mathrm{Im}[A] * \mathrm{Im}[B]}{\mathrm{Re}[B]^2 + \mathrm{Im}[B]^2}$$

$$\mathrm{Im}[C] = \frac{\mathrm{Im}[A] * \mathrm{Re}[B] - \mathrm{Im}[B] * \mathrm{Re}[A]}{\mathrm{Re}[B]^2 + \mathrm{Im}[B]^2}$$

Самостійно визначити всі необхідні для роботи пристрою додаткові блоки. Перевірку роботи пристрою виконайте за допомогою тестового файлу.

Лабораторна робота № 8

«Простий процесорний модуль»

1. Теоретичні відомості

На малюнку 29 зображено цифровий пристрій, що складається з декількох 16-розрядних регістрів, мультиплексора, пристрою додавання / віднімання, лічильника і пристрою керування. Дані надходять в систему через 16-розрядну шину DIN.

Вхідні дані, через 16-розрядний мультиплексор, надходять в різні регістри: R0,. . . , R7 та A. Мультиплексор дозволяє передавати дані з одного регістра в інший. Вихід мультиплексора називається шиною (bus), тому що цей термін часто вживається для каналів передачі даних від однієї системи в іншу.

Для виконання операції додавання і віднімання спочатку перший доданок, через мультиплексор, має бути завантажено в регістр A. Після цього, інше 16-розрядне число подається на шину, потім пристрій додавання / віднімання виконує необхідну операцію, результат якої зберігається в регістрі G. Згодом, дані з регістра G можуть бути передані в будь-який інший регістр.

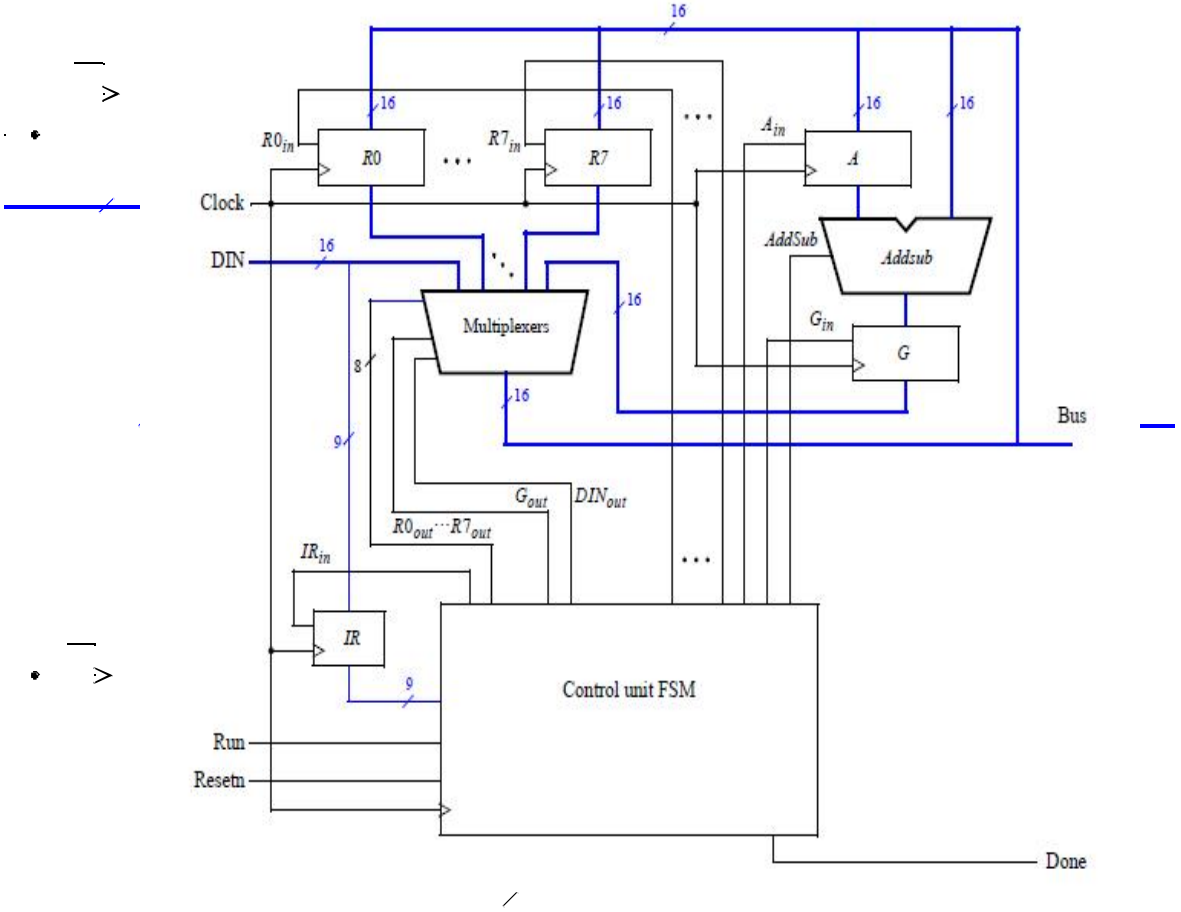


Рис. 29. Цифровий пристрій.

Система може виконувати різні операції на кожному такті, які задаються пристроєм керування. Цей пристрій визначає місце розташування вхідних даних, які повинні бути подані на шину, і в який з регістрів вони повинні бути завантажені. Наприклад, якщо пристрій керування отримав сигнали R0 out та Ain, це означає, що мультиплексор повинен передати вміст регістру R0 на шину даних, а з наступним активним фронтом сигналу тактової частоти ці дані повинні бути записані в регістр A.

Подібна система називається процесором. Він виконує дії, описані у вигляді команд. У таблиці 1 представлений перелік команд, які підтримуються процесором, що розробляється. У лівому стовпчику вказані імена команд і їх операнди. Застосований синтаксис RX ← [RY] говорить про те, що вміст регістра RY має бути завантажено в регістр RX. Команда **mv** (пересилання) дозволяє копіювати дані з одного регістра в інший. Команда mvi (безпосереднє завантаження), представлена виразом RX ← D, говорить про те, що 16-розрядна константа D повинна бути завантажена в регістр RX.

Operation	Function performed
mv Rx,Ry	$Rx \leftarrow [Ry]$
mvi Rx,#D	$Rx \leftarrow D$
add Rx, Ry	$Rx \leftarrow [Rx] + [Ry]$
sub Rx, Ry	$Rx \leftarrow [Rx] - [Ry]$

Таблиця 1. Система команд процесора.

Кожна команда кодується і зберігається в регістрі IR, використовуючи 9-розрядний формат IIIXXXYYY, де поле III вказує на команду, поле XXX вказує номер регістру RX, а поле YYY вказує номер регістру RY. Звичайно, для кодування наведеної системи команд достатньо двох біт, але три біта обрані для можливості подальшого розширення. Як видно з малюнка1, регістр IR безпосередньо підключений до дев'яти розрядів 16-розрядного вхідного порту DIN. Для команди mvi полеYYY не використовується, і константа #D подається на вхідний 16-розрядний порт DIN відразу ж після запису команди mvi в регістр IR.

Для виконання таких команд, як додавання та віднімання знадобиться більше одного такту, через велику кількість необхідних пересилань даних. Для цього, пристрій керування використовує двухразрядний лічильник, керуючий порядком проходження команд. Процесор виконує команду, яка прийшла на вхід DIN, за сигналом Run і відповідає сигналом Done, коли команда виконана. У таблиці 2 наведені керуючі сигнали, які видаються під час виконання команд з таблиці 1, і відповідні їм тактові моменти часу. Зверніть увагу, в регістрі IRin під час нульового такту зберігаються тільки керуючі команди, тому він і не показаний в наведеній таблиці.

	T_1	T_2	T_3
(**mv**): I_0	$RY_{out}, RX_{in},$ *Done*		
(**mvi**): I_1	$DIN_{out}, RX_{in},$ *Done*		
(**add**): I_2	RX_{out}, A_{in}	RY_{out}, G_{in}	$G_{out}, RX_{in},$ *Done*
(**sub**): I_3	RX_{out}, A_{in}	$RY_{out}, G_{in},$ *AddSub*	$G_{out}, RX_{in},$ *Done*

Таблиця 2. Керуючі сигнали, що встановлюються при виконанні команд

2. Самостійна робота.

Завдання 1

Створіть проект, який реалізує процесорний модуль.
Перевірте працездатність пристрою за допомогою моделюючої програми (з використанням тестового файлу).

Завдання 2

Доопрацювати процесорний модуль, створений в завданні 1. Додайте до нього блок пам'яті, як показано на малюнку 30, для зберігання програми, що виконується.

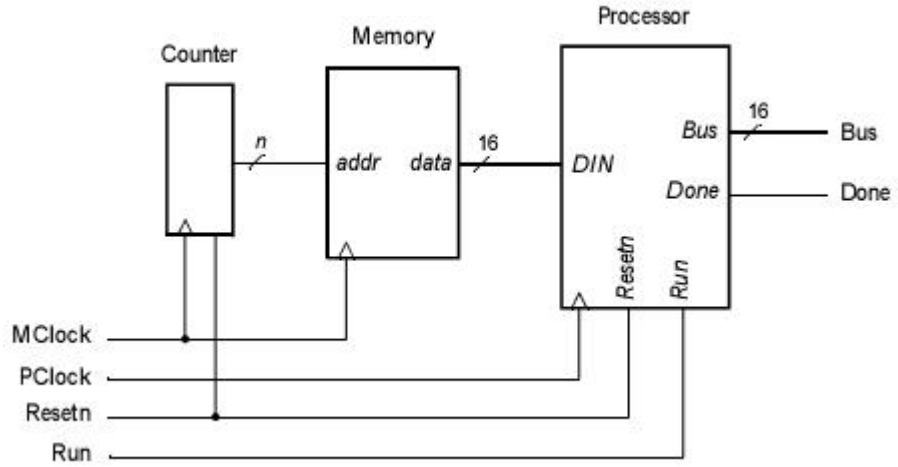


Рис. 30. Підключення блоку пам'яті до процесорного модулю.

На цьому малюнку лічильник Counter використовується для формування адреси блоку пам'яті Memory. Процесорний модуль і блок пам'яті працюють кожен зі своєю тактовою частотою - Pclock і MClock, відповідно.
Для запису програми в блок пам'яті створіть текстовий файл ініціалізації вмісту блоку пам'яті.
Перевірте працездатність пристрою за допомогою моделюючої програми.

ЛІТЕРАТУРА

1. Клайв Максфилд. Проектирование на ПЛИС. Архитектура, средства и методы. – М.; Додэка-XXI, 2007. – 408 с.
2. Рябенький В.М., Ушкаренко О.О. MAX+plusII. Основи проектування цифрових пристроїв на ПЛІС.- К.: “Корнійчук”, 2004.
3. В. Немудров, Г. Мартин. Системы-на-кристалле. Проектирование и развитие. – М.; Техносфера, 2004. – 216 с.
4. О.Н. Партала. Цифровая электроника. Издание 2-е, дополненное – СПб: Наука и Техника, 2001. - 411 с.
5. Угрюмов Е.П. Цифровая схемотехника: Учеб. пособие для вузов. – 2-е изд., перераб. и доп. СПб.: БХВ-Петербург, 2004. – 528 с.
6. В.В.Соловьев. Проектирование цифровых систем на основе программируемых логических интегральных схем. - М.; Горячая линия – Телеком, 2001.- 312 с.
7. М.О.Кузелин, Д.А.Кнышев, В.Ю.Зотов. Современные семейства ПЛИС фирмы XILINX. Справочное пособие. - М.; Горячая линия – Телеком, 2004. – 440 с.
8. В.Б.Стешенко. ПЛИС фирмы ALTERA: элементная база, система проектирования и языки описания аппаратуры. - М.; Додэка-XXI, 2002. – 530 с.
9. Р.Грушвицкий, А.Мурсаев, Е.Угрюмов. Проектирование систем на микросхемах программируемой логики. - СПб.; БХВ-Петербург, 2002. – 590 с.