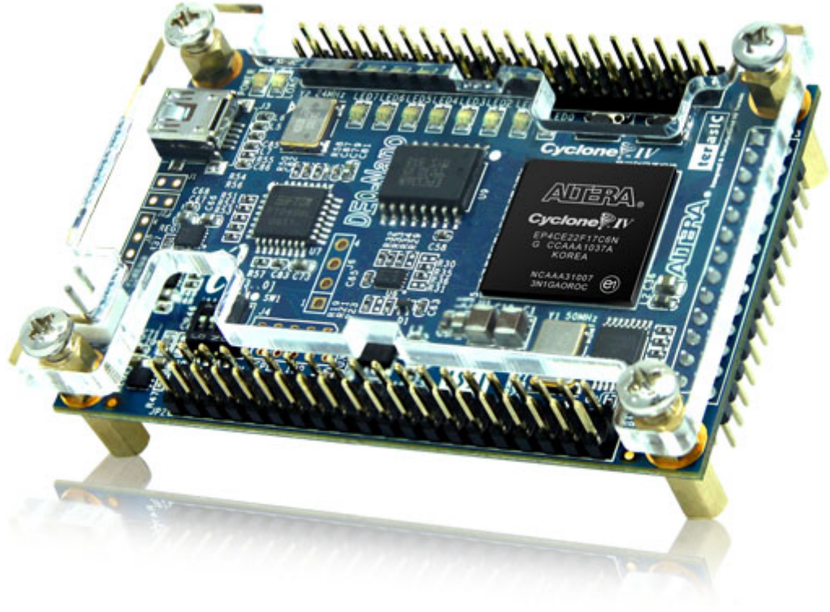
П. М. Ратушний, О. М. Жагловська, К. В. Огородник

ПЛІС ТА ЇХ ПРОГРАМУВАННЯ



Навчальне видання

Ратушний Павло Миколайович
Жагловська Олена Миколаївна
Огородник Костянтин Володимирович

ПЛІС та їх програмування

Лабораторний практикум

Рукопис оформлено: О. Жагловська

Редактор: О. Ткачук

Оригінал-макет виготовлено: О. Ткачук

Підписано до друку 17.05.2018.
Формат 29,7×42¼. Папір офсетний.
Гарнітура Times New Roman.
Друк різографічний. Ум. друк. арк. 3,29.
Наклад 50 (1-й запуск 1–20) пр. Зам. № 2018-094.

Видавець та виготовлювач
Вінницький національний технічний університет,
інформаційний редакційно-видавничий центр.
ВНТУ, ГНК, к. 114. Хмельницьке шосе, 95,
м. Вінниця, 21021.
Тел. (0432) 65-18-06.
press.vntu.edu.ua;
email: kivc.vntu@gmail.com.
Свідоцтво суб'єкта видавничої справи
серія ДК № 3516 від 01.07.2009 р.

Міністерство освіти і науки України
Вінницький національний технічний університет

Ратушний П. М., Жагловська О. М., Огородник К. В.

ПЛІС ТА ЇХ ПРОГРАМУВАННЯ

Лабораторний практикум

Вінниця
ВНТУ
2018

УДК [621.3.049.77+004.31] (076)
Р25

Рекомендовано до друку Вченою радою Вінницького національного технічного університету Міністерства освіти і науки України (протокол № 9 від 26.04.2018 р.)

Рецензенти:
І. В. Троцишин, доктор технічних наук, професор
В. Ю. Кучерук, доктор технічних наук, професор
В. Г. Петрук, доктор технічних наук, професор

Ратушний, П. М.

Р25 ПЛІС та їх програмування : лабораторний практикум / Ратушний П. М., Жагловська О. М., Огородник К. В. – Вінниця : ВНТУ, 2018. – 57 с.

У практикумі подано теоретичні й фізичні принципи концепції будови запрограмованих схем, методи проектування програмованих схем у вигляді лабораторних робіт. Лабораторний практикум підготовлено відповідно до плану кафедри та програми дисципліни «Програмовані логічні інтегральні схеми».

Лабораторний практикум призначено для студентів спеціальності 153 – «Мікро- та наносистемна техніка» з курсу «Програмовані логічні інтегральні схеми».

УДК [621.3.049.77+004.31] (076)

ЗМІСТ

ВСТУП…….. 4
ЛАБОРАТОРНА РОБОТА № 1
Знайомство із структурою навчального стенду Altera, підключення, встановлення драйверів пристрою, знайомство з програмним середовищем Quartus II ……...…...…… 5
ЛАБОРАТОРНА РОБОТА № 2
Написання перших програм мовою Verilog, компіляція програм та програмування мікросхеми ПЛІС……...……..... 20
ЛАБОРАТОРНА РОБОТА № 3
Створення складного проекту з декількох модулів з використанням графічних оболонок ……..…….. 27
ЛАБОРАТОРНА РОБОТА № 4
Арифметичні та логічні функції в мові Verilog ……...…….. 38
ЛАБОРАТОРНА РОБОТА № 5
Процедурні блоки в мові Verilog.……..…. 42
ЛАБОРАТОРНА РОБОТА № 6
Робота з VGA-адаптером, виведення зображень на монітор..... 47
СПИСОК ЛІТЕРАТУРИ……...…… 56

ВСТУП

Мова Verilog призначена для опису обладнання, а відповідно, конкретна программа на мові Verilog являє собою модель відповідного електронного цифрового пристрою. Основний принцип поведінкового проектування електронних схем можна сформулювати таким чином: якщо модель повністю визначає функціональність пристрою і його реакцію на будь-які допустимі зовнішні впливи, то на основі такої моделі можна автоматично синтезувати цифрову логічну схему, яка реалізовує пристрій, який моделюється. Засоби автоматичного синтезу цифрових схем вимагають використання для опису моделей деякої формальної мови. На сьогодні розроблено низку мов, які отримали в науково-технічній літературі назву HDL (Hardware Description Language – мова опису обладнання), які використовуються в межах компаній-розробників, залишаючись їхніми внутрішніми інтелектуально-програмними інструментами. Крім того, деякі компанії-виробники інтегральних схем програмованої логіки (наприклад, компанія «Альтера») створюють власні мови HDL і можуть включати їх у склад лише власних інтегрованих середовищ розробки і синтезу логічних ланцюгів. По-справжньому широке розповсюдження отримали VHDL і Verilog – мови міжнаціонального спілкування в сфері розробки електронних цифрових пристроїв.

Лабораторний практикум складається з опису шести лабораторних робіт, які включають знайомство із структурою навчального стенду Altera, написання перших програм мовою Verilog, створення складного проекту з декількох модулів.

Зміст лабораторних робіт складається з мети роботи, теоретичних відомостей, робочого завдання, методичних вказівок стосовно виконання роботи, складу звіту, контрольних питань.

ЛАБОРАТОРНА РОБОТА № 1

Знайомство із структурою навчального стенду Altera, підключення, встановлення драйверів пристрою, знайомство з програмним середовищем Quartus II

Мета роботи: ознайомитись з основними компонентами навчального стенду ALTERA DE0, навчитись підключати, встановлювати драйвери. Переглянути основні компоненти програмного середовища Quartus, перевірити в програмному середовищі чи встановлений пристрій. За допомогою тестової програми перевірити роботу основних елементів навчальної плати.

Теоретичні відомості

Концепція будови запрограмованих схем

Кожна програмована інтегральна схема містить певний набір логічних схем, а також структуру шляхів, що уможливлює реалізацію сполучень між окремими логічними схемами, відповідно до потреб впроваджуваного проекту. Наочно це показано на рис. 1.1.

На рис. 1.1, (а) показано приклад набору елементів і шляхів: два логічних елементи A і B, а також шляхи a, b, c, d, e, f. Якщо захочемо сполучити вихід елемента A з входом елемента B, то потрібно виконати таке сполучення як показано, наприклад, на рис. 1.1, (б).

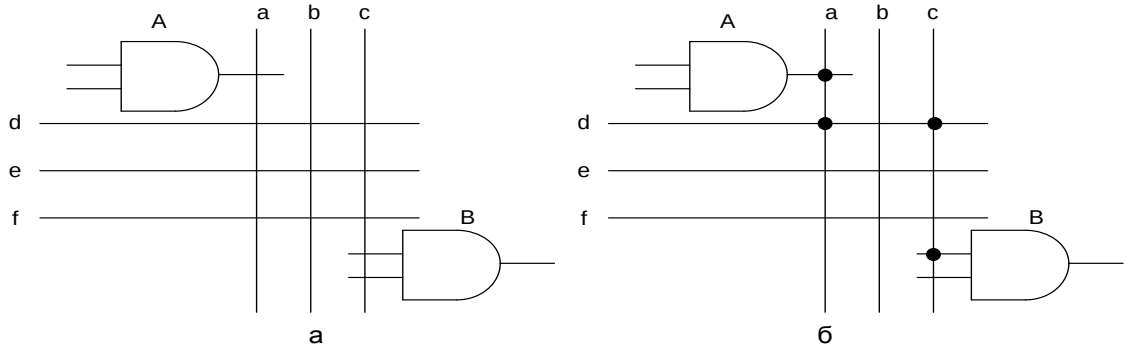


Рисунок 1.1 –
Приклад набору елементів і шляхів в програмованій схемі (а);
приклад реалізації сполучень між елементами A і B (б)

Реалізація сполучень між доступними схемами за допомогою доступних шляхів визначається як програмування схеми або як конфігурування схеми.

Тепер з'ясуємо, яким способом, використовуючи готову (замкнену) інтегральну схему, можна програмувати (виконувати) сполучення між відповідними шляхами в інтегральній структурі. Відомих є декілька методів реалізації таких сполучень. Один із способів полягає у використанні структур запобіжників. Якщо між двома шляхами помістимо

запобіжник (так як на рис. 1.2, (а), то, поки цей запобіжник не перепалиться, буде існувати сполучення між шляхами a і b. Після перепалення запобіжника сполучення перерветься (рис. 1.2, б).

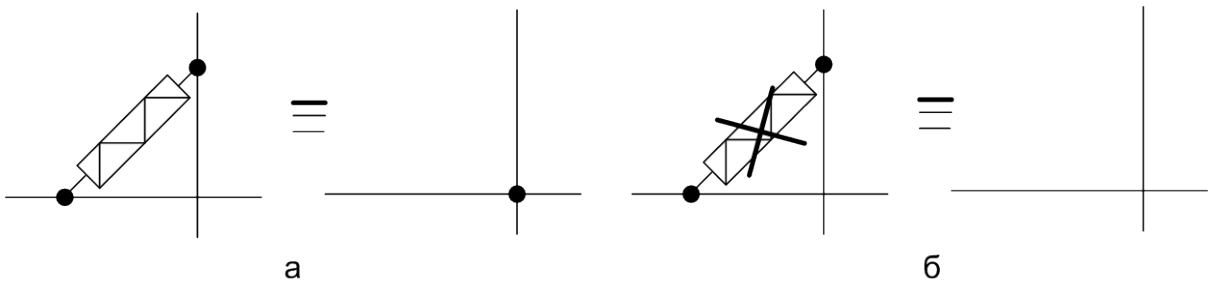


Рисунок 1.2 – Програмування сполучення з використанням запобіжника: а) робочий запобіжник; б) спалений запобіжник

При використанні запобіжників в програмованій схемі спочатку виконуються усі можливі сполучення і програмування полягає в усуненні непотрібних зв'язків, тобто у спаленні відповідних запобіжників. Програмування структури сполучень відбувається в спеціальних програматорах. У такому разі процес програмування є необоротним. Погано запрограмована схема придатна лише для викидання.

Застосовуються також рішення, у яких замість запобіжників використовуються так звані антизапобіжники, тобто елементи, що на початку не передають, а лише після програмування стають перешкодою. У такому випадку спочатку в структурі немає жодних сполучень між шляхами і тільки процедура програмування активізує відповідні зв'язки.

В іншому разі, замість запобіжників використовуються транзистори з подвійним (плаваючим) елементом. Зображення такого транзистора показано на рис. 1.3. Поки на цей додатковий елемент Gf, що ізольований від середовища, не буде подано навантаження, транзистор поводиться як звичайний транзистор типу MOS – при відповідній полярності транзистор проводить струм між стоком D і джерелом S. Якщо, натомість, на додатковий елемент буде подано навантаження, то при такій самій поляризації, як перед цим, транзистор буде відімкнутий і не проводитиме струм між стоком D і джерелом S. Оскільки плаваючий елемент є ізольованим від середовища, то навантаження, подане на цей елемент, може втриматися довгий час (мінімум 10 років) і стан сполучення, реалізованого транзистором, можна вважати сталим.

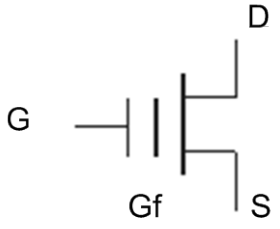


Рисунок 1.3 – Транзистор з подвійним елементом

Програмування схем з транзисторами з подвійним елементом реалізовується в спеціальних програматорах. Однак цього разу, у разі помилки можна змінити стан запрограмованих сполучень (усіх одночасно), використовуючи для цього спеціальний пристрій, у якому схема може бути просвітлена ультрафіолетовим промінням, що спричиняє усунення навантажень з плаваючих елементів. Далі схему можна повторно запрограмувати.

В іншому випадку для реалізації сполучень використовуються транзистори MOS, що керуються сигналами із співпрацюючих тригерів FF; приклад сполучення показано на рис. 1.4. Якщо в тригері буде запам'ятований логічний нуль, то транзистор MOS не передає сигнал. Якщо, натомість, у тригері буде запам'ятована логічна одиниця, то транзистор буде передавати сигнал.

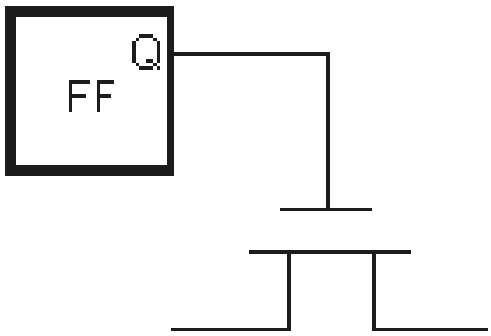


Рисунок 1.4 – Реалізація програмованих сполучень з використанням транзисторів MOS і співпрацюючих тригерів (комірок пам'яті)

У цьому випадку програмування структур полягає на записуванні інформації до тригерів, співпрацюючих з відповідними транзисторами MOS. Цю інформацію можна ввести на початку роботи з пристроєм, безпосередньо після вмикання живлення. Інформація про конфігурацію сполучень у програмованій схемі буде зберігатися до часу вимикання живлення і при повторному вмиканні живлення має бути повторно введена. У попередньо розглянутих рішеннях запрограмована конфігурація сполучень утримувалась незалежно від того, чи живлення увімкнене, чи ні.

Ще інша можливість програмування сполучень полягає у використанні схем мультиплексорів (рис. 1.5). Це рішення вигідне тоді, коли на такий шлях ми хочемо передати сигнал від одного з декількох шляхів. Вибір відповідного вхідного шляху виконується за допомогою адресних входів, керованих допоміжними тригерами. Програмування сполучень полягає, як і дотепер, у записуванні відповідної інформації до відповідних тригерів. Додамо, що на практиці це реалізовується без виймання програмованої схеми з пристрою.

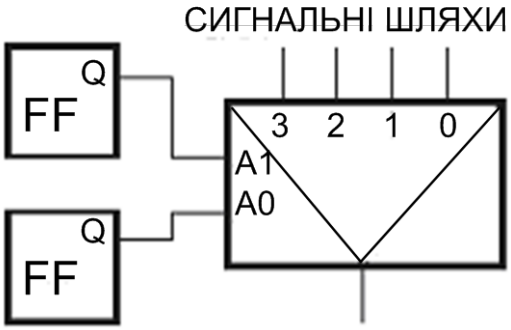


Рисунок 1.5 – Сполучення сигнальних шляхів
з використанням мультиплексора

Перед тим як перейдемо до вказаних прикладів програмованих схем, потрібно вивчити умови рисування схем, прийняті в програмованих схемах. На рис. 1.6, (а) показано класичні правила рисування зображення логічних елементів (на прикладі елемента AND), а на рис. 1.6, (б) показано правила, що застосовуються в програмованих схемах. Зазначимо, що в класичних правилах шляхи вхідних сигналів зменшуються безпосередньо до символу елемента. У нових правилах до символу елемента підходить одна лінія, яку перетинають перпендикулярно лінії сигнальних шляхів. Якщо хочемо зазначити, що певний сигнальний шлях має бути під'єднаний до елемента, то на перетині цього шляху з лінією, що доходить до елемента, вставляється символ сполучення.

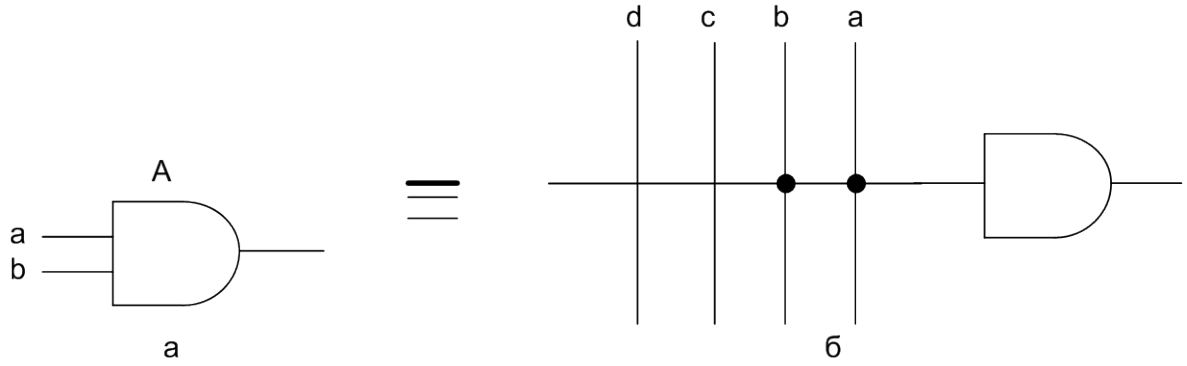


Рисунок 1.6 – Два правила позначення сполучень, що підходять до входів елементів: а) класичне; б) застосовуване в програмованих схемах

В обох випадках, показаних на рисунку 1.6, на входи елемента подаються сигнали a і b. Сигнали c і d не подаються на входи елемента.

Прості програмовані схеми

Посеред виготовлюваних програмованих схем є схеми різної складності. Застосовуються також різні назви. Прості схеми невеликої складності найчастіше містять набори елементів добутків, сум і тригерів. Зустрічаються назви PAL (Programmable Array Logic), PLA (Programmable Logic Array), PLD (Programmable Logic Devices), GAL (Generic Array

Logic). Ці назви пов’язані або з певною внутрішньою архітектурою програмованої схеми, або це фірмові назви. Далі розглянемо приклад схеми з цієї групи схем, а саме: схему GAL16V8.

На рис. 1.7 показано логічне позначення схеми GAL16V8 (Зазначено тільки інформаційні сигнали). Позначення, що починаються з однієї літери I, належать до входів схеми; позначення, що починаються з літери O, – до виходів схеми; позначення, що починаються з літер IO, – до виводів схеми, які можуть виконувати функцію входу або виходу.

Складова в назві схеми 16V8 інформує про те, що, використовуючи схему, можна мати в розпорядженні 16 входів і до 8 виходів. Беручи до уваги доступний набір кінців схеми, показаний на рис. 7, можна зауважити, що загальна кількість одночасно використовуваних входів і виходів не може перевищувати 18. Літера V інформує, що активний рівень вихідного сигналу може бути змінним (низький або високий). Схема GAL16V8 може бути запрограмована на роботу в різних конфігураціях. Найчастіше використовуються такі конфігурації: комбінаційна, позначена як 16V8C, а також регістрова, позначена як 16V8R.

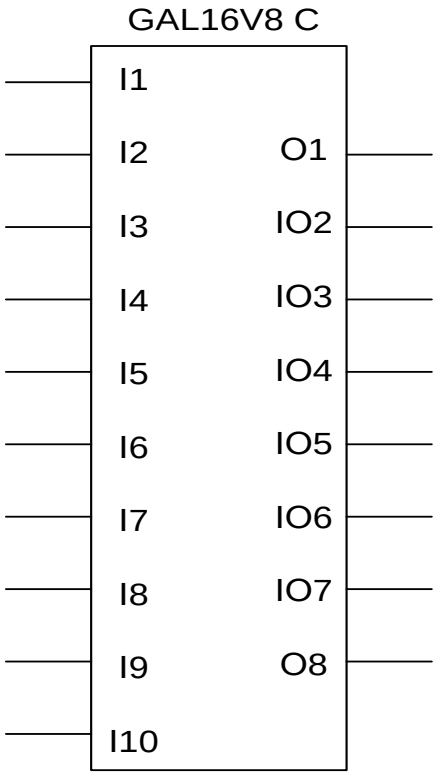


Рисунок 1.7 – Логічне позначення схеми GAL16V8C

Внутрішня структура схеми GAL16V8C містить 8 ідентичних сегментів. На рис. 1.8 показано один з цих сегментів.

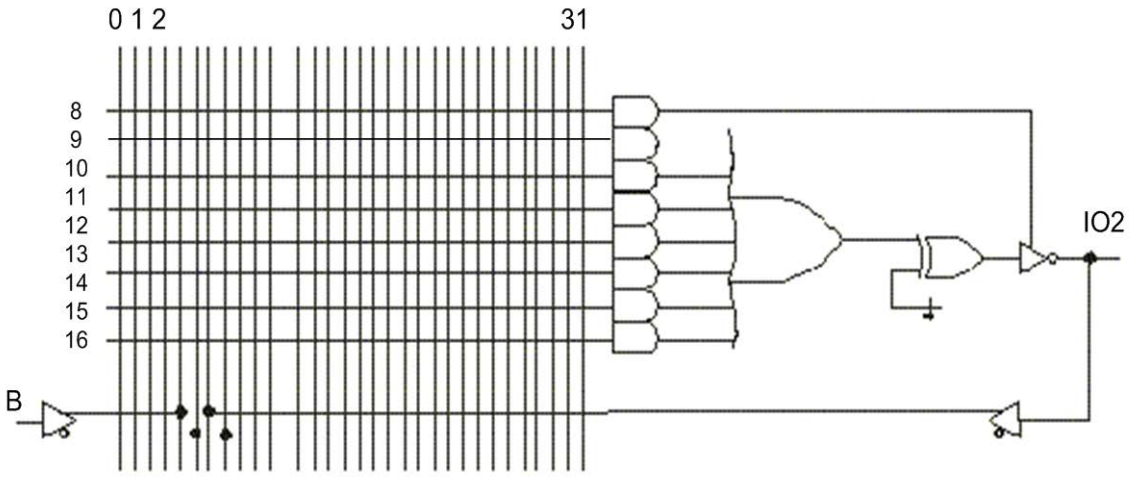


Рисунок 1.8 – Один сегмент схеми GAL16V8C

Сегмент уможливлює реалізацію функції суми добутків. Сполучення між виходами семи добутків і входами суми є сталі і не потребують програмування. Можна, натомість, програмувати сполучення з входами відповідних добутків. До кожного добутку можна додати до 16 сигналів або їх заперечення (розрядні лінії 8–15 на рис. 1.8). Вхідні сигнали та їх заперечення доступні на 32 шляхах (перпендикулярні шляхи 0–31 на рис. 1.8). Зазначимо, що кожен сигнал, який приходить ззовні (наприклад, I3), одразу заперечується. Це забезпечують схеми, пов’язані з відповідними входами, які передають усередину структури вихідні сигнали як у звичайному вигляді, так і в запереченому.

Вихід зі схеми суми може бути заперечений або ні, залежно від того, чи програмовані входи елемента XOR сполучені з логічною одиницею, чи з логічним нулем. (Пам’ятаємо, що при логічному нулі на одному з входів елемента XOR на виході елемента з’явиться те саме логічне значення, що є на другому вході, а при логічній одиниці на одному з входів елемента XOR на виході з’явиться заперечене логічне значення відносно того, що є на другому вході елемента). Вихідний сигнал з елемента XOR передається на вихід (IO2) через тристабільний елемент. Тристабільний елемент керується сигналом з восьмого добутку в сегменті. При закритому тристабільному елементі вихід IO2 виконує роль входу, а при відкритому тристабільному елементі – роль виходу, причому вихідний сигнал є одночасно вхідним сигналом для схеми GAL 16V8R, завдяки чому можливо реалізовувати послідовні схеми.

У регістровій конфігурації схеми GAL 16V8R у сегменті додатково є доступний тригер. Схема логічної частини сегмента показано на рис. 1.9. У цій конфігурації для записування інформації до тригера використовується загальний сигнал CLK, поданий ззовні і доведений до часових входів тригерів у всіх сегментах схеми. Вихідний тристабільний елемент

керується загальним сигналом OE, поданим ззовні. Тому «звільнений» восьмий елемент використовується як додатковий елемент, під’єднаний до входу елемента суми.

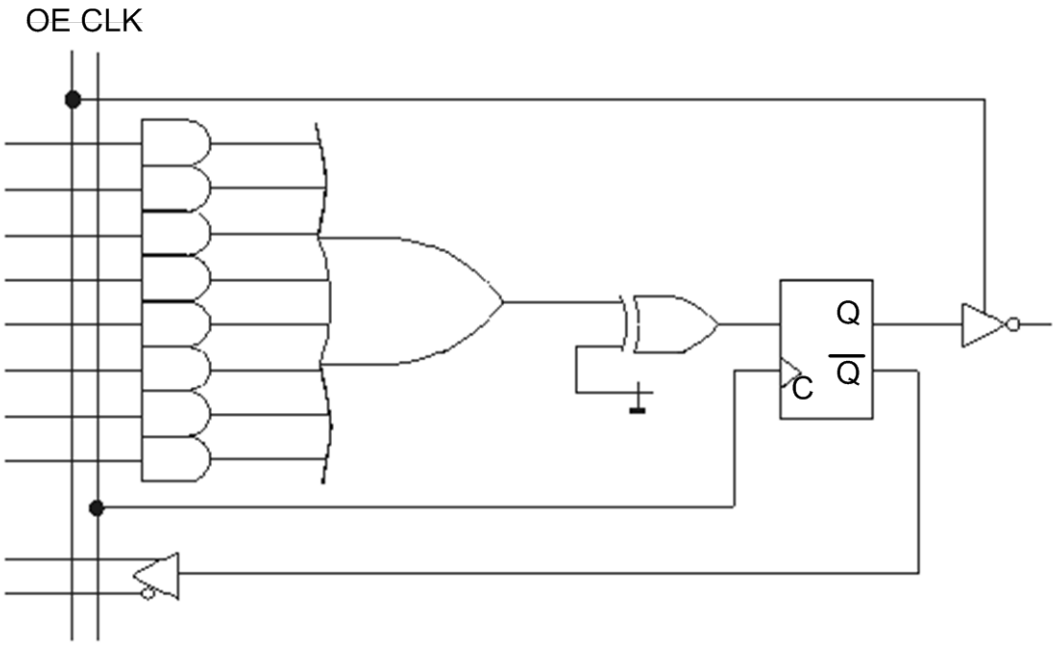


Рисунок 1.9 – Логічна частина схеми GAL16V8R

Зазначимо, що цього разу всередину схеми зворотно подається сигнал із запереченого виходу тригера, а також, що вихід сегмента не може виконувати функцію входу. Тому в регістровій конфігурації схема GAL16V8R має 8 доступних зовнішніх входів (плюс 8 входів з виходів тригерів) і 8 зовнішніх виходів (плюс входи CLK і OE). На рис. 1.10 показано графічне зображення схеми GAL16V8R.

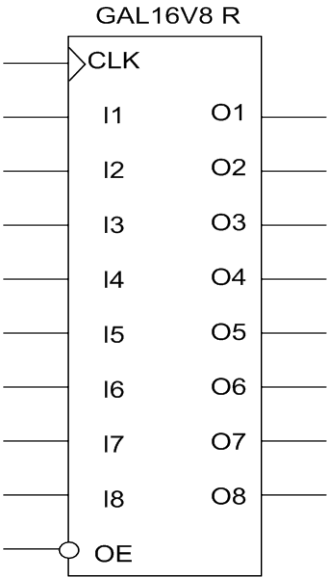


Рисунок 1.10 – Графічне зображення схеми GAL16V8R

Поряд із схемами GAL16V8 виготовлюють схеми з більшою кількістю входів і виходів (GAL20V8, GAL22V10). Подібні схеми випускаються різними виробниками.

Програмовані схеми CPLD і FPGA

Розглянуті вище схеми типу GAL дозволяють реалізовувати досить складні логічні проекти. Однак для багатьох практичних застосувань ця складність є недостатньою і через це розроблено багато інших схем із значно збільшеною складністю (Складність програмованої схеми часто оцінюється як кількість елементів NAND, котрі потрібно використати, щоб реалізувати функції за допомогою розглянутої програмованої схеми). Історично розрізняють дві концепції будови складних програмованих схем: схеми CPLD, а також схеми FPGA.

Концепція будови схем CPLD полягає на тому, щоб в одній програмованій схемі інтегрувати певну кількість простих програмованих схем і забезпечити відповідну структуру шляхів, що уможливить програмування сполучень. У концепції схем FPGA припускають, що в інтегральній структурі програмованої схеми буде міститися певна кількість так званих макрокомірок або логічних блоків (кожна макрокомірка чи логічний блок – це набір вибраних логічних схем), а також відповідна структура шляхів для сполучень. Зараз різниці між схемами, реалізованими за допомогою цих концепцій, стають більш заплутаними і найчастіше для складних програмованих схем використовуються назви FPGA.

Складні програмовані схеми, як правило, містять три складових елементи (рис. 1.11): набір макрокомірок, мережа сполучень, а також схеми входу/виходу. Макрокомірки становлять матрицю блоків, що займає середню частину сполучень. Мережа програмованих сполучень займає вільні місця між макрокомірками. Натомість схеми входу/виходу розміщені на краях структури і забезпечують співпрацю програмованої схеми із середовищем.

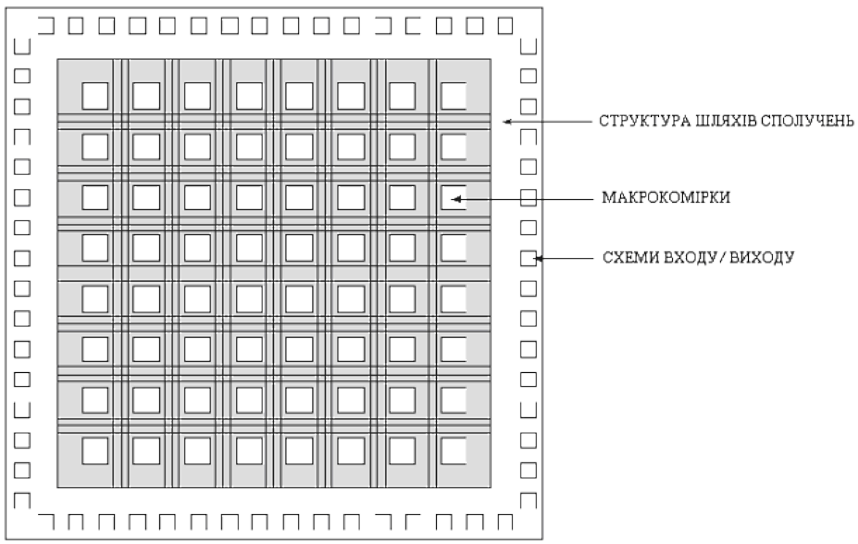


Рисунок 1.11 – Загальна архітектура схем FPGA

Кожна комірка входу/виходу, як правило, уможливлює співпрацю з різними інтегральними схемами (TTL, CMOS) з різними значеннями напруги, зі тристабільними схемами або із схемами з відкритим колектором та ін. Комірки входу/виходу програмуються так само, як макрокомірки.

Макрокомірки містять різні набори логічних схем, що уможливлюють реалізацію складних логічних функцій. Відповідні макрокомірки можуть самостійно реалізовувати певні функції або можуть співпрацювати з іншими комірками, що знаходяться в структурі програмованої схеми.

Мережа сполучень забезпечує можливість програмування сполучень як всередині відповідних макрокомірок, так і між ними, а також сполучень з комірками входу/виходу. Структури мереж сполучень мають різні варіанти рішень – окремі виробники використовують свої рішення, намагаючись забезпечити еластичність проведення сполучень, а також швидкість передачі сигналів. Як правило, виділяють спеціальні шляхи, що слугують для поширення часових сигналів.

Проектування з програмованими схемами

Програмовані схеми уможливлюють реалізацію складних, навіть дуже складних, проектів. На практиці процес проектування підтриманий спеціальним програмуванням, забезпеченим зазвичай фірмами, що виготовлюють програмовані схеми.

У процесі проектування потрібно реалізувати декілька таких завдань. Увесь процес проектування схематично показано на рис. 1.12.

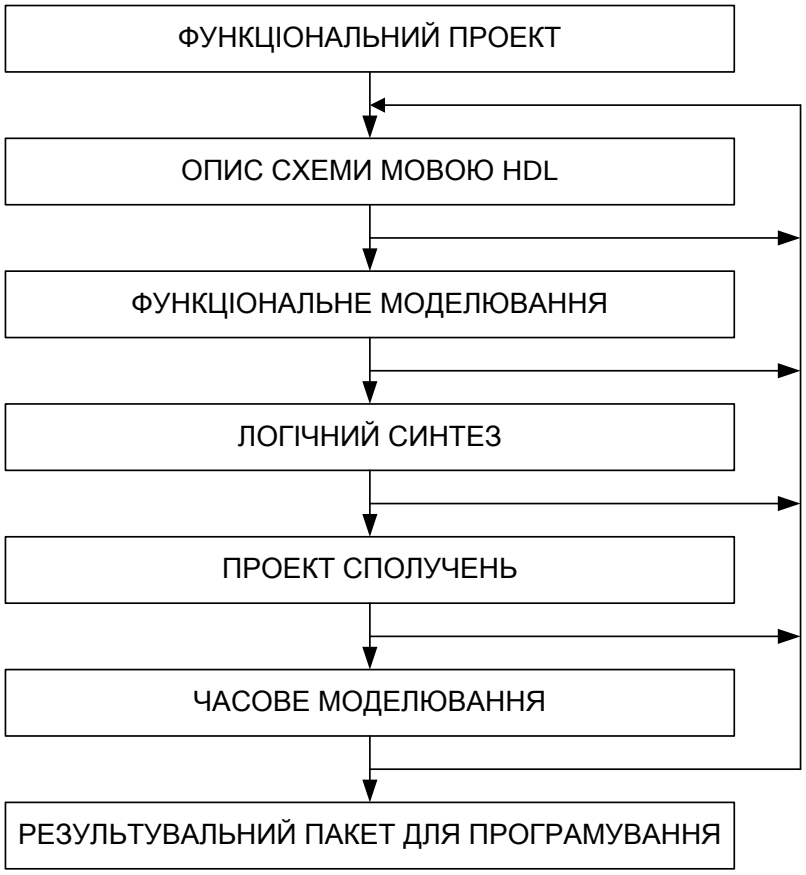


Рисунок 1.12 – Приклад процесу проектування програмованих схем

На початку процесу проектування потрібно мати функціональний проект. Мають бути визначені всі вимоги, що висуваються до проектованої схеми (пристрою).

Після формулювання функціональних припущень можливо описати проект вибраною спеціальною мовою опису пристроїв HDL. На практиці проектування найчастіше використовують мови ABEL, VHDL і VERILOG. Ці мови уможливлюють опис функцій, що має здійснювати проектований пристрій. Мова ABEL (її фірмові версії, наприклад, AHDL) є відносно проста і її використовують у випадку простих програмованих схем. У разі великих проектів і користування схемами FPGA використовуються мови VHDL і VERILOG. Можливості обох цих мов близькі між собою.

Після опису проекту однією з мов HDL можливе функціональне моделювання з метою перевірки правильності опису і функціонування проектованої схеми.

Почергово виконується етап логічного синтезування проекту. У результаті з'являється логічна схема проектованої схеми.

Тепер можливий перехід до відображення логічної схеми на структуру використовуваної програмованої схеми. Найперше виконується декомпозиція схеми на такі фрагменти, які можуть бути реалізовані доступним сегментом програмованої схеми (у простих схемах) або макрокомірки (у складних схемах). Далі визначається структура сполучень, яку потрібно буде потім фізично реалізувати – запрограмувати відповідні сполучення.

Почергово можна виконати повне моделювання проекту з урахуванням часових залежностей.

У процесі виконання відповідних етапів проекту може виявитися, що такий етап не можна реалізувати або що були виявлені суперечності з функціональними припущеннями (етапи моделювання). Тоді потрібно повернутися до одного з попередніх етапів і ввести відповідні зміни.

Якщо процес проектування буде успішно доведений до кінця, як результат отримуємо кінцевий пакет з інформацією, необхідною для програмування програмованої схеми або з використанням програматора, або безпосередньо в системі.

Навчальна плата Altera DE0 є чудовою платформою для реалізації проектів в програмованій логіці. Однією з переваг програмування ПЛІС є модульність проектів. Тобто є можливість створювати окремі модулі, які можна використовувати для різних проектів. З окремих модулів можна створювати графічні елементи та створювати проект у графічному вигляді. Платформа DE0 обладнана програмованою логічною схемою FPGA Cyclone III 3C16. Плата містить 346 контактів вводу/виводу, зокрема порти для зовнішніх пристроїв (монітора, клавіатури, карти пам'яті, та інші).

Програмне забезпечення Quartus II є основним інструментом, що використовується для розробки, тестування, компіляції проектів цифрових пристроїв та прошивки їх в кристал ПЛІС. Установити цю програму можна безкоштовно з диску, що є в комплекті до навчальної платформи, або скачавши з сайту *www.altera.com/download.*

Платформа DE0 містить інтегрований USB Blaster для програмування (прошивки) FPGA. Для зв’язку персонально комп’ютера з навчальною платформою потрібно встановити драйвер для USB Blaster.

Хід роботи

1. Перевірте, чи встановлено на ПК програмне забезпечення Quartus II. Зазвичай воно встановлюється в папку C:\altera. Якщо не встановлено, то встановіть.

2. Підключіть стенд DE0 до ПК через кабель USB. Якщо драйвера на пристрій відсутні на такому ПК (можна перевірити в «Диспетчері пристроїв»), то операційна система запропонує встановити драйвер. Виберіть пункт «Встановити із визначеного місця» (рис. 1.13).

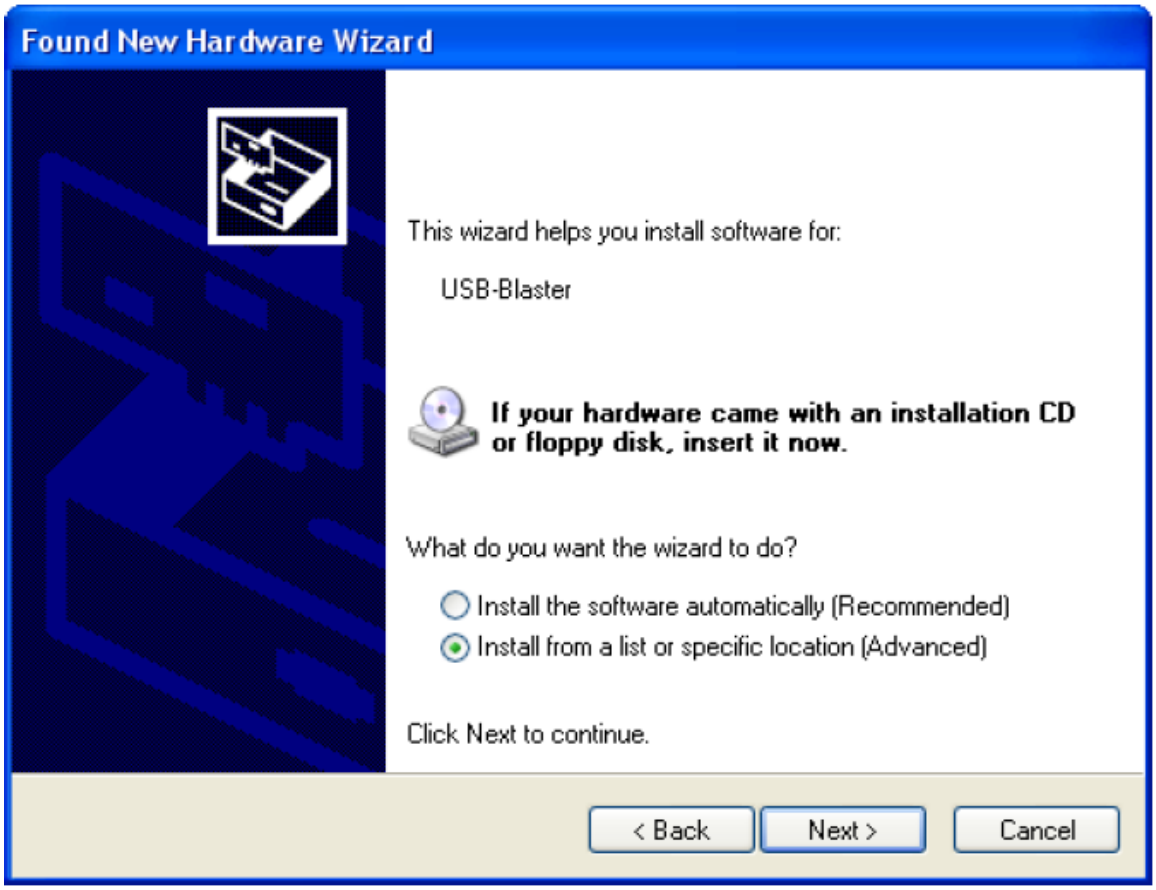


Рисунок 1.13 – Діалогове вікно для вибору місця пошуку драйвера

Оберіть шлях до файлів драйвера (зазвичай вони в папці C:\altera\91sp1\quartus\drivers\usb-blaster\x32). Якщо розрядність операційної системи – 32, то обирайте папку x32, якщо ж 64, то відповідно x64 (рис. 1.14, 1.15).

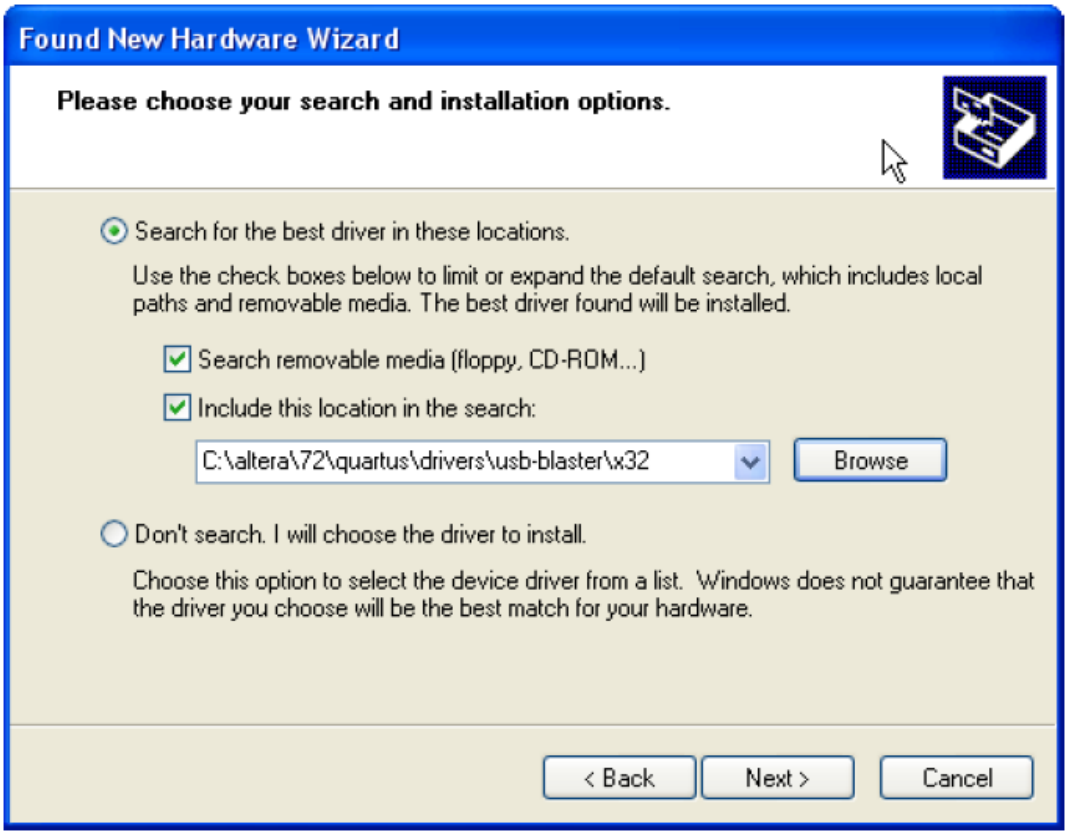


Рисунок 1.14 – Діалогове вікно встановлення драйвера

Рисунок 1.15 – Діалогове вікно вибору папки з драйвером

Якщо операційна система покаже попереджувальне вікно (рис. 1.16), то натисніть кнопку «Continue Anyway».

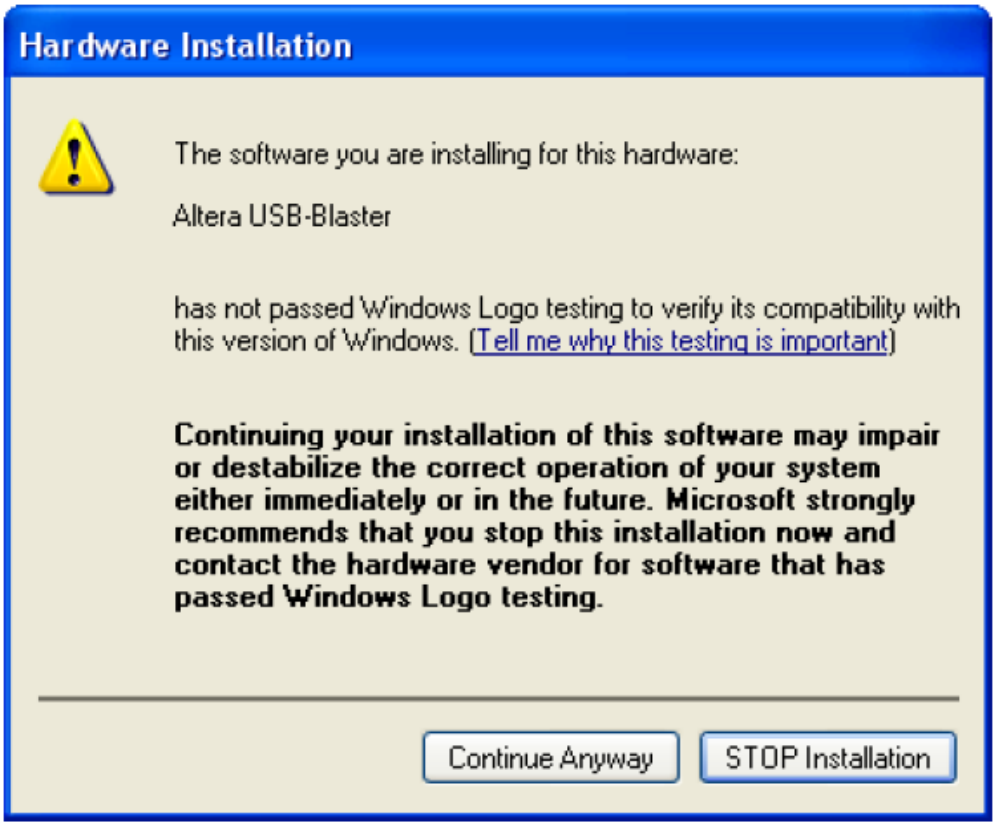


Рисунок 1.16 – Попереджувальне діалогове вікно

Якщо драйвер установився, то з’явиться діалогове вікно (рис. 1.17)

Рисунок 1.17 – Діалогове вікно про успішне завершення встановлення

3. Тепер навчальний стенд готовий до роботи. Запустіть програму Quartus. У розділі «Tools» виберіть пункт «Programmer», натисніть кнопку «Hardware setup… ». Якщо в стрічці з поточним активним пристроєм напис «No Hardware», то оберіть з випадаючого списку «USB-Blaster».

4. Увімкніть навчальну платформу червоною кнопкою.

5. Ознайомтесь з компонентами навчальної платформи (рис. 1.18) з використанням файлу «DE0_User_manual.pdf».

6. Запустіть тестову програму DE0_ControlPanel.exe та протестуйте основні компоненти плати, використовуючи інструкції, наведені у файлі «DE0_User_manual.pdf».

7. Сформулюйте висновки про здобуті навички.

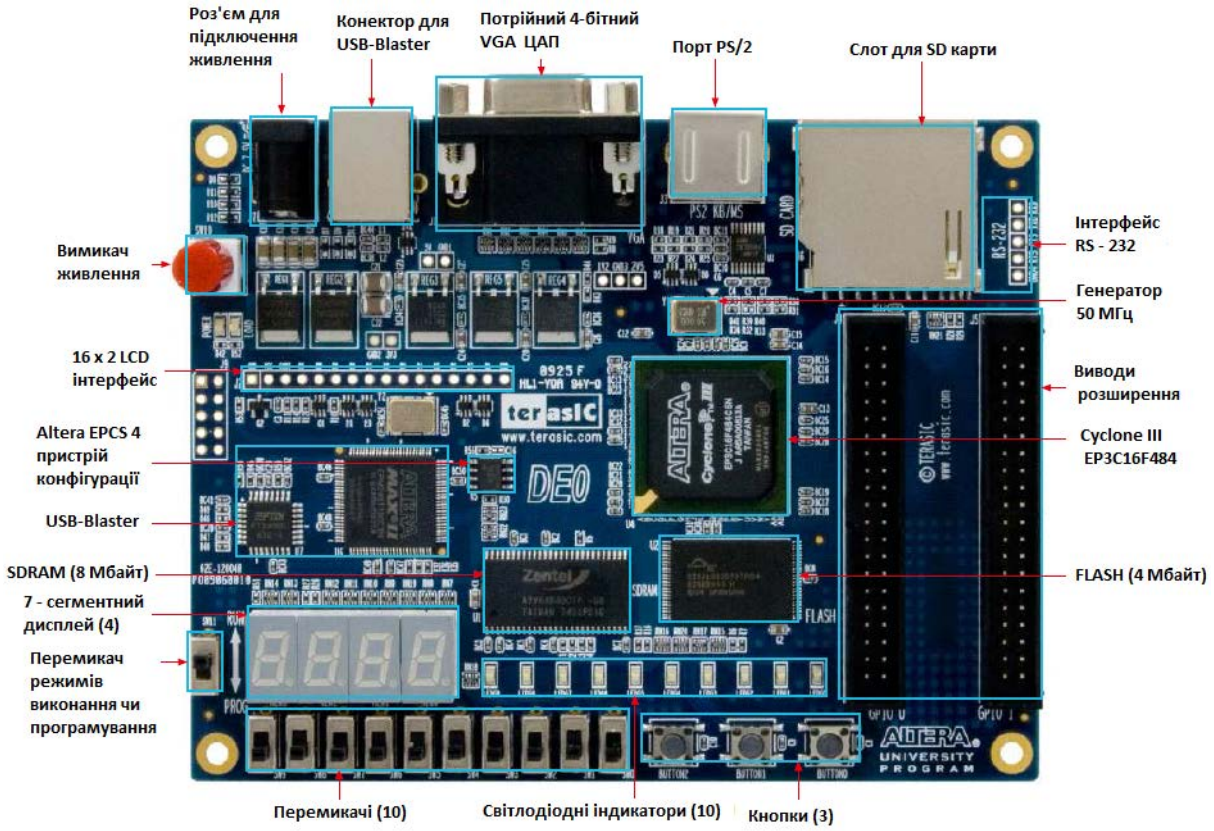


Рисунок 1.18 – Вигляд навчальної платформи DE0

Перелік контрольних запитань

1. Яка процедура встановлення драйверів навчального стенду?
2. Як перевірити в програмному середовищі, чи встановлений пристрій?
3. Опишіть основні компоненти навчального стенду та їхнє призначення.
4. Опишіть порти входу/виходу на платі та їхнє призначення.
5. З чого складаються програмовані логічні інтегральні схеми?

6. Опишіть архітектуру внутрішньої будови FPGA.

7. Що є конфігуруванням схеми?

8. Якими способами можна програмувати сполучення в інтегральній схемі?

9. У якому випадку доцільно використовувати програмування схем з використанням мультиплексорів?

10. Набори яких основних елементів містять прості програмовані схеми?

11. У чому полягає концепція будови схем CPL?

12. У чому полягає концепція будови схем FPGA?

13. З яких елементів складаються складні програмовані логічні схеми?

14. Наведіть послідовність процедур процесу проектування логічних схем.

ЛАБОРАТОРНА РОБОТА № 2

Написання перших програм мовою Verilog, компіляція програм та програмування мікросхеми ПЛІС

Мета роботи: навчитись створювати проекти в середовищі Quartus, отримати навики написання простих програм мовою Verilog, скомпілювати їх, проаналізувати і виправити помилки, назначити робочі виводи, запрограмувати мікросхему.

Теоретичні відомості

Базові типи джерел сигналів

Verilog – мова опису цифрових схем з відповідними типами даних.

Сигнали – це електричні імпульси, які передаються по провідниках (wires) між логічними елементами схеми. Провідники переносять інформацію не проводячи над нею жодних обчислень. У цифровій схемі сигнали важливі для передачі двійкових даних. Один з базових типів джерела сигналу в мові Verilog – це ланцюг або провідник ***wire***. Таким чином, якщо у вас є арифметичний або логічний вираз, ви можете асоціювати результат виразу з іменованим провідником і пізніше використовувати його в інших виразах. Це дещо схоже на змінні. Значення провідника (***wire***) – це функція того, що приєднано до нього.

Ось приклад декларації однобітового провідника в програмі, написаній мовою Verilog:

```
wire a;
```

Ви можете йому призначити інший сигнал таким чином:

```
wire b;
assign a = b;
```

Або ви можете визначити сигнал і зробити призначення йому одночасно в одному виразі:

```
wire a = b;
```

У вас можуть бути провідники, що передають кілька бітів:

```
wire [3:0] c; // це чотири проводи
```

Кілька проводів, які передають кілька бітів інформації називаються «шина», іноді «вектор».

Призначення до них робляться так само:

```
wire [3:0] d;
assign c = d;  // «підключення» шини d до шини c
```

Кількість провідників у шині визначається будь-якими двома цілими числами розділеними двокрапкою всередині квадратних дужок.

```
wire [11:4] e; // восьмибітова шина
wire [0:255] f; // 256-ти бітова шина
```

З шини можна вибрати деякі потрібні біти і призначити іншому проводу:

```
wire g;
assign g = f[2]; // назначити проводу g біт № 2  шини f
```

Крім того, обирати з шини біт можна змінною:

```
wire [7:0] f;
wire i = f[h]; // назначити сигналу i біт номер h  з шини f
```

Ви можете вибрати з сигнальної шини деякий діапазон бітів і призначити інший шині з тією ж кількістю бітів:

```
wire [3:0] j = e[7:4];
```

Також, у більшості діалектів Verilog, ви можете визначити масиви сигнальних шин:

```
wire [7:0] k [0:19]; // масив з двадцяти 8-ми бітових шин
```

Існує інший тип джерела сигналу, який називається регістр – ***reg***. Регістр ***reg*** у мові Verilog позначає змінну, яка може зберігати значення, а не апаратний регістр. Тип ***reg*** використовують при поведінковому і процедурному описі цифрової схеми. Якщо регістру постійно присвоюється значення комбінаторної (логічного) функції, то він поводиться так, як і провідник ***wire***. Якщо ж регістру присвоюється значення в синхронній логіці, наприклад по фронту сигналу тактової частоти, то йому, в кінцевому рахунку, буде відповідати фізичний D-тригер або група D-тригерів.

D-тригер – це логічний елемент, здатний запам'ятовувати один біт інформації. Регістри описуються так само, як і провідники:

```
reg [3:0] m;
reg [0:100] n;
```

Вони можуть використовуватися, як і провідники, у правій частині виразів як операнди:

```
wire [1:0] p = m[2:1];
```

Ви можете визначити масив регістрів, які зазвичай називають «пам'ять»:

```
reg [7:0] q [0:15]; //память з 16 слів, кожне по 8 біт
```

Ще один тип джерела сигналу – це ***integer***. Він схожий на регістр ***reg***, але завжди є 32-бітовим, цілочисельним, знаковим типом даних. Наприклад, оголосимо змінну типу ***integer***:

```
integer loop_count;
```

Verilog дозволяє групувати логіку в блоки. Кожен блок логіки називається «модулем» (***module***). Модулі мають входи і виходи, які поводять себе як сигнали ***wire***. При описі модуля спершу перераховують його порти (входи і виходи):

```
module my_module_name (port_a, port_b, w, y, z);
```

А потім описують напрямок сигналів:

```
input port_a;
output [6:0] port_b;
input [0:4] w;
inout y; // двонаправлений сигнал
```

Вихід модуля може бути одразу задекларовано як регістр ***reg***, а не як провід ***wire***.

Ще простіше можна одразу в описі модуля вказати тип і напрямок сигналів:

```
module my_module
(
    input wire port_a,
    output wire [6:0]port_b,
    input wire [0:4]w,
    inout wire y,
    output reg [3:0]z
);
```

Тепер можна використовувати вхідні сигнали, як провода ***wire***:

```
wire r = w[1];
```

Тепер можна робити постійні призначення виходам, як функції від входів:

```
assign port_b = h[6:0];
```

Наприкінці опису логіки кожного модуля пишемо слово ***endmodule***.

```
module my_module_name (input wire a, input wire b, output
wire c);
assign c = a & b;
endmodule
```

Ще один тип джерела сигналу, який розглянемо, – це постійні сигнали або просто числа:

```
wire [12:0] s = 12; /* 32-бітове десяткове число, яке буде
“обрізане” до 13 біт */
wire [12:0] z = 13’d12; //13-бітове десяткове число
wire [3:0] t = 4'b0101; //4-бітове двійкове число
wire [7:0] q = 8'hA5; // 8-бітове шістнадцяткове число A5
wire [63:0] u = 64'hd79dbe1aca8ebabe; /*64-бітове
шістнадцяткове число */
```

Якщо точно не визначити розмір числа, то воно приймається за замовчуванням 32-розрядним. Це може бути проблемою при присвоєнні сигналів з більшою або меншою розрядністю.

Числа можуть використовуватися у різних арифметичних і логічних виразах. Наприклад, можна додати 1 до вектора *aa*:

```
wire [3:0] aa;
wire [3:0] bb;
assign bb = aa + 1;
```

Хід роботи

1. Запустіть Quartus і створіть новий проект:

а) пункт меню «File → New Project Wizard…»;

б) створіть папку, у якій буде зберігатись проект, та вкажіть шлях до неї. Введіть назву файлу проекту (англійським текстом), ім’я файлу верхнього рівня вводиться автоматично, таке ж як і назва файлу проекту;

в) наступним кроком майстер запропонує додати файли до проекту, пропустіть цей крок («Next»);

г) оберіть мікросхему (рис. 2.1), яку будете програмувати (назва на корпусі мікросхеми);

д) наступним кроком запропоновано обрати додаткові інструменти моделювання, пропустіть цей крок («Next»). Перегляньте введені дані та натисніть «Finish».

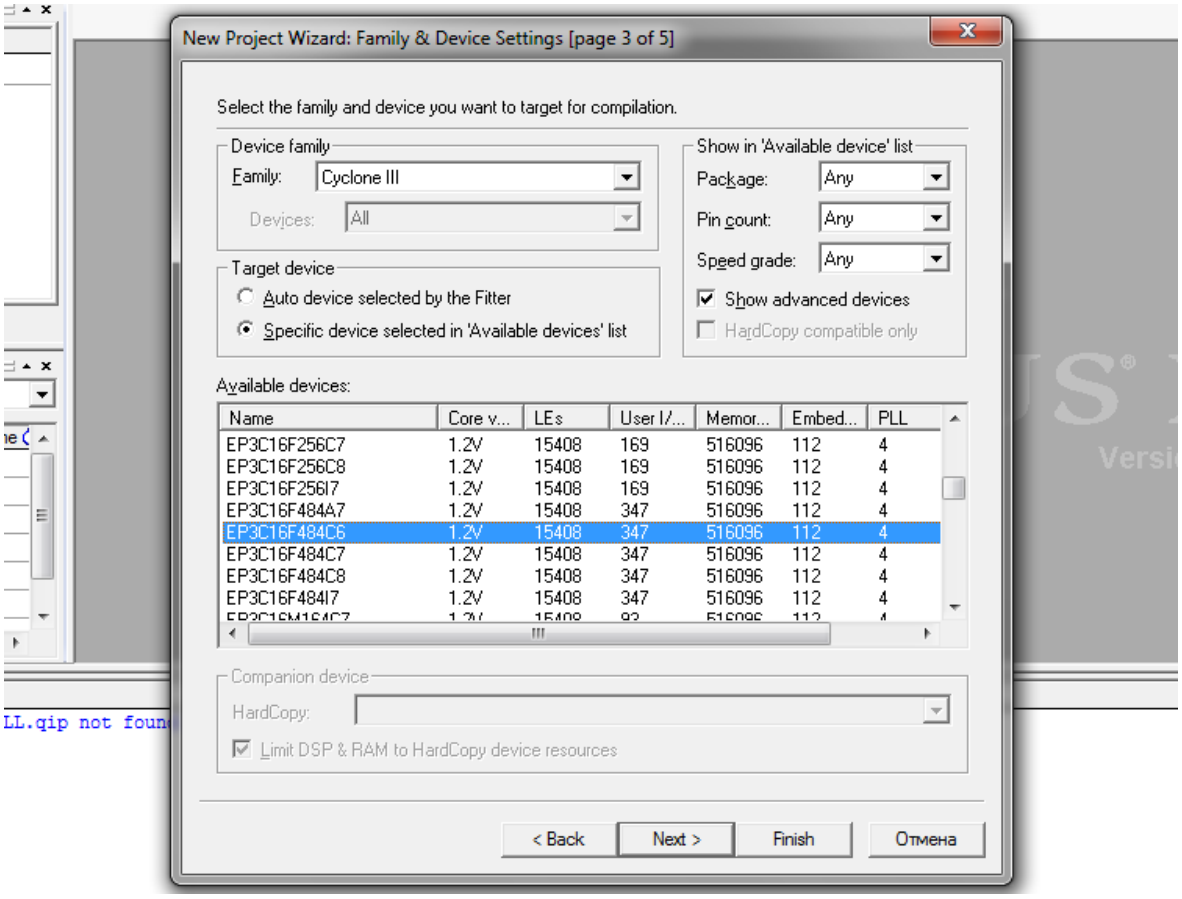


Рисунок 2.1 – Вибір сімейства мікросхеми та її назви

2. Ознайомтесь з інтерфейсом програми. У вікні «Project Navigator» наведено ієрархію проекту та складові файли проекту. У вікні «Tasks» наведено перелік задач, які може виконати Quartus. У вікні «Messages» показуються інформаційні повідомлення, попередження та помилки проекту.

3. Створіть новий файл проекту: «New → Verilog HDL File».

4. Напишіть просту програму: нехай при увімкненні перемикача з'явиться високий рівень напруги на виході (логічна «1»), яка увімкне світлодіод чи інший пристрій (рис. 2.2).

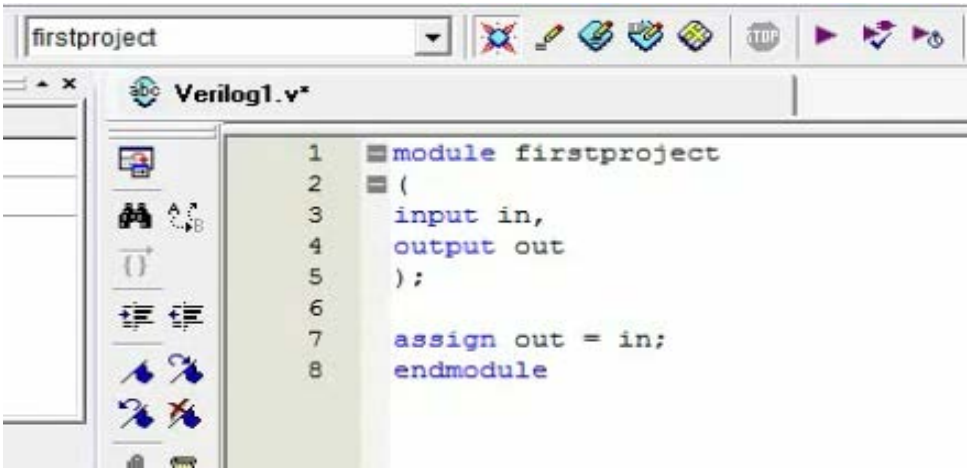
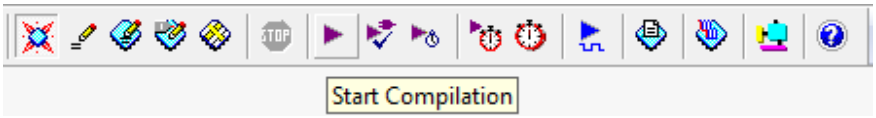


Рисунок 2.2 – Текст програми

Опис мовою Verilog починається із завдання назви модуля. Модуль – це елементарний будівельний блок мови Verilog. Прості проекти можна описати в одному модулі, а складні потребують відповідної ієрархічної структури. Далі описуються входи і виходи схеми. У нас буде один вхід з назвою «in» і один вихід з назвою «out». Вони описуються в дужках після назви модуля, розділяються комою, а після дужок – крапка з комою. Далі – опис роботи схеми (назначити виходу «out» значення входу «in»). Завершення опису відбувається ключовим словом «endmodule». Зверніть увагу, що в програмі зарезервовані ключові слова виділені синім. Не називайте змінні словами, які є зарезервовані в мові Verilog.

5. Збережіть файл *з такою ж назвою, як і назва модуля*.

6. Скомпілюйте проект, натиснувши кнопку «Start Compilation».



7. Якщо в тексті програми були помилки, то компілятор вкаже на них. Виправте помилки, якщо вони були, і знову скомпілюйте проект. Якщо помилок у тексті немає, компілятор видасть декілька попереджень, з яких лише одне суттєве: не назначені виводи схеми (Pins).

8. Назначте входи і виходи схеми реальним ніжкам ПЛІС. В розділі «Assignments» оберіть пункт «Pins» (рис. 2.3).

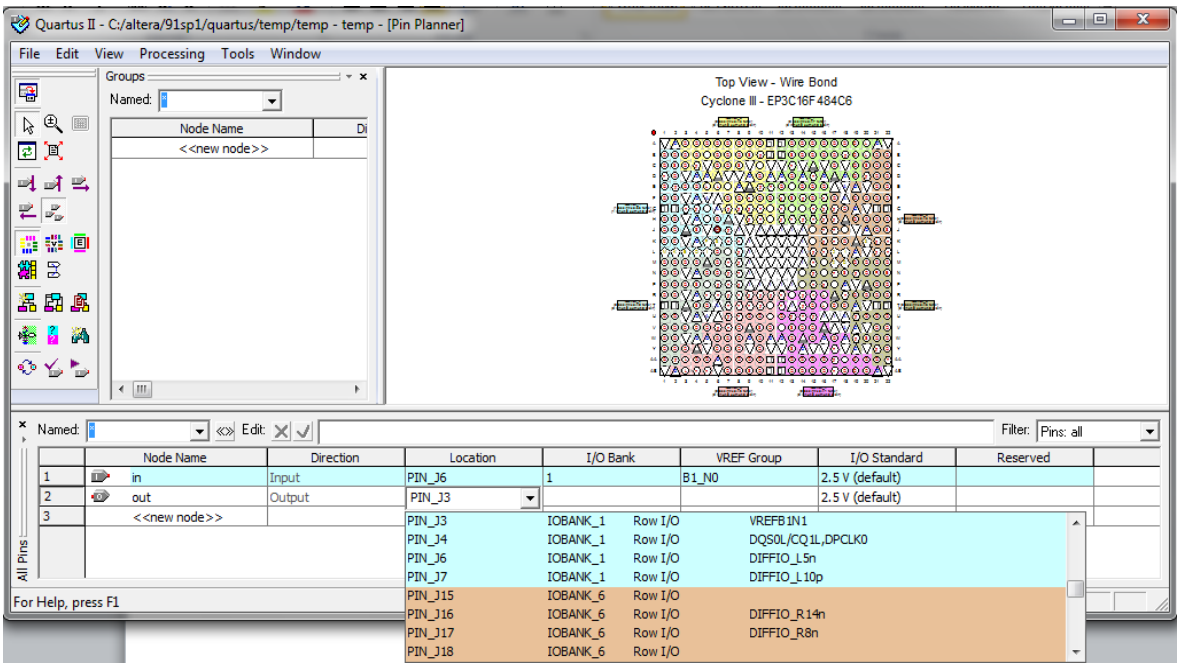


Рисунок 2.3 – Інтерфейс інструменту назначення виводів

Для входу «in» можна, наприклад, обрати один з перемикачів на платі (SW0 ... SW9). Назви всіх елементів плати та відповідні їм виводи мікросхеми ПЛІС наведено в файлі «DE0_User_manual.pdf». Для виходу «out» можна обрати один зі світлодіодів (LEDG0 .. LEDG9).

9. Після назначення всіх виводів скомпілюйте проект знову. Якщо критичних помилок немає, то проект можна завантажити в мікросхему ПЛІС. У розділі «Tools» оберіть пункт «Programmer». Підключіть плату до ПК USB кабелем, увімкніть її, перемикач «RUN/PROG» має бути встановлений у «RUN». Оберіть файл прошивки та натисніть кнопку «Start». Якщо вона не активна, то натисніть «Hardware Setup…» і перевірте чи вибраний пристрій.

10. Якщо програмування відбулось, то перевірте роботу пристрою перемикаючи відповідний перемикач і спостерігаючи зміну стану відповідного світлодіода.

11. Створіть програму для періодичного мигання світлодіода. Для цього можна використати генератор тактових імпульсів (50 МГц), який є на платі DE0, та двійковий лічильник імпульсів. Молодший розряд лічильника буде змінювати значення з частотою 50 МГц, а кожен наступний – з частотою у 2 рази менше. Розрахуйте, який розряд лічильника потрібно підключити до світло діода, щоб він мигав з частотою менше 10 Гц. За вищенаведеним алгоритмом створіть новий проект та напишіть таку програму (рис. 2.4).

У 3 і 4 рядках описані входи і виходи модуля. У 7 рядку створено 23-розрядний двійковий лічильник, якому одразу присвоєно початкове значення 0. Рядок 9 вказує на те, що завжди при позитивному фронті змінної «clk» буде виконуватись код між подальшими «begin» та «end». Тобто щоразу при зміні значення змінної «clk» з 0 в 1 до лічильника буде додаватись 1. У рядку 14 відбувається присвоєння змінній «out» значення старшого розряду лічильника.

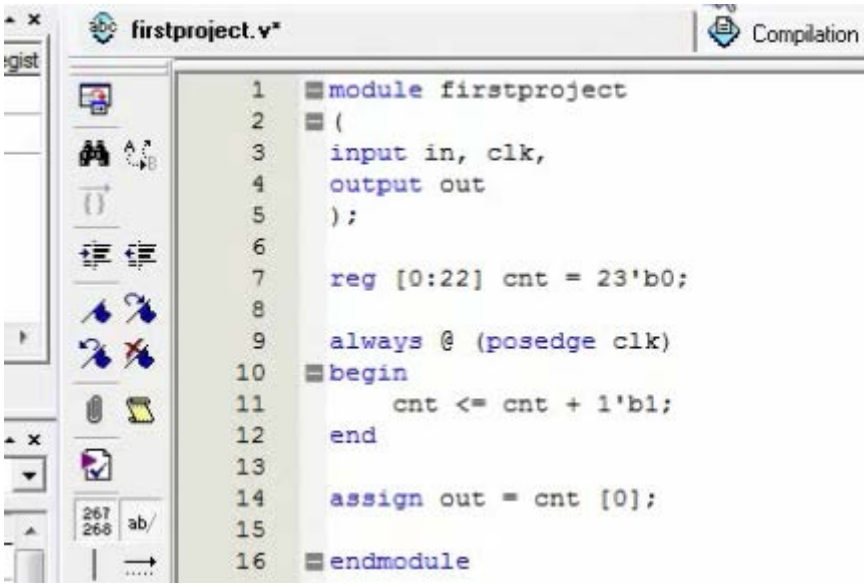


Рисунок 2.4 – Текст програми мигання світлодіода

12. Скомпілюйте проект. Якщо є помилки – виправте, і знову скомпілюйте. При назначенні виводів як вхід «clk» використайте генератор тактових імпульсів (50 МГц), який підключений до одного з виводів ПЛІС. Цю інформацію можна отримати з файлу «DE0_User_manual.pdf». Для виходу «out» використайте один зі світлодіодів плати. Після назначення виводів знову скомпілюйте та запрограмуйте ПЛІС.

13. За завданням викладача змініть частоту мигання світлодіода.

14. Зробіть висновки про здобуті знання та навички.

Перелік контрольних запитань

1. Опишіть базові типи джерел сигналів у мові Verilog.
2. Яким чином у мові Verilog відбувається декларація проводів?
3. Яким чином в мові Verilog відбувається присвоєння значень проводам?
4. Опишіть способи декларації шин та присвоєння їм значень.
5. Поясніть поняття регістру в мові Verilog. Що є багатобітовим регістр?
6. Опишіть порядок опису модуля. Яким чином описуються входи та виходи модуля?
7. Опишіть спосіб запису чисел у мові Verilog.
8. Напишіть декілька чисел в різних системах числення відповідно до правил їхнього запису в мові Verilog.
9. Опишіть етапи створення проекту в Quartus.
10. Опишіть етапи перевірки, компіляції, призначення виводів та прошивки мікросхеми.
11. Як змінити частоту мигання світлодіода в другій програмі?

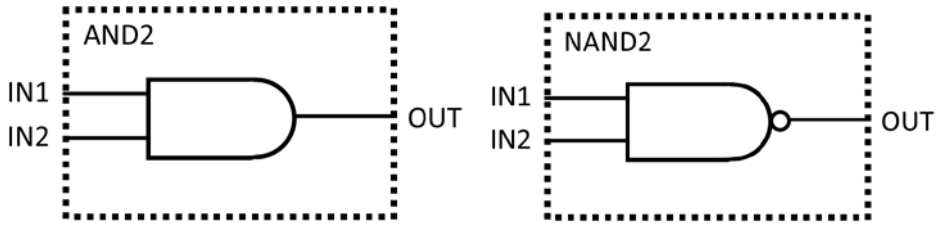
ЛАБОРАТОРНА РОБОТА № 3

Створення складного проекту з декількох модулів з використанням графічних оболонок

Мета роботи: навчитись створювати складні проекти, що складаються з багатьох модулів. Навчитись створювати графічні елементи для написаних модулів та з'єднувати їх у графічній оболонці.

Теоретичні відомості

Ми вже знаємо, що таке модуль. У проекті, особливо складному, буває багато модулів, з'єднаних між собою. Насамперед, потрібно зауважити, що зазвичай у проекті завжди є один модуль найвищого рівня (top level). Він складається з декількох інших модулів, які можуть містити ще модулі і так далі. Не обов'язково, щоб всі модулі були написані однією мовою опису апаратури. Зовсім навпаки. Досить зручно і наочно мати модуль найвищого рівня виконаним у вигляді схеми, що складається з модулів більш низького рівня. Ці модулі можуть бути написані різними людьми, різними мовами (Verilog, VHDL, AHDL) і навіть виконані у вигляді схеми. Насправді, це все справа смаку і можливостей компілятора (синтезатора), а також вимог замовника. Отже, усередині тіла будь-якого модуля, можна оголошувати екземпляри інших модулів і потім з'єднувати їх один з одним проводами. Нехай у нас є такі примітивні модулі. Ви бачите графічне представлення найпростіших логічних елементів і нижче їх таблиця істинності – значення логічної функції на виході при заданих значеннях на входах (рис. 3.1). Зліва зображено двовходовий логічний елемент «І». На його виході одиниця, якщо на першому і на другому вході одиниця. Справа зображений двовходовий логічний елемент «І-НЕ». На його виході – нуль, якщо на першому і другому входах – одиниці.



AND2	0	1	x	z
0	0	0	0	0
1	0	1	X	X
x	0	X	X	X
z	0	X	X	X

NAND2	0	1	x	z
0	1	1	1	1
1	1	0	X	X
x	1	X	X	X
z	1	X	X	X

Рисунок 3.1 – Логічні елементи та їхні таблиці істинності

Зверніть увагу, що на вході може бути не тільки логічний нуль або одиниця, але також невизначене значення «X», або вхід може бути підключений до двонаправленого сигналу (inout), що знаходиться в високоомному стані «Z». Ці значення так само вказані в таблиці істинності для повноти картини.

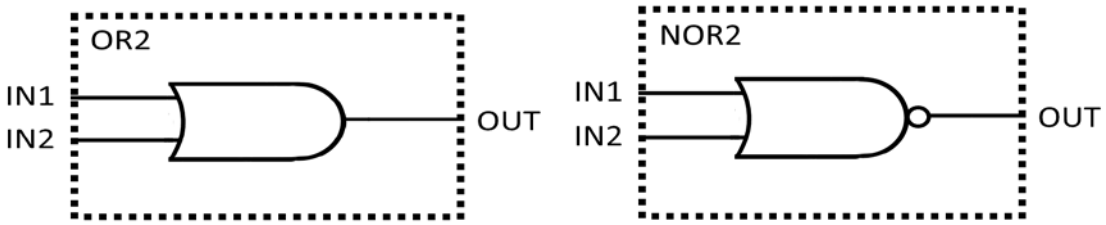
Ці логічні модулі можна було б описати мовою Verilog таким чином:

```
module AND2 (output OUT, input IN1, input IN2);
    assign OUT = IN1 & IN2;
endmodule

module NAND2 (output OUT, input IN1, input IN2);
    assign OUT = ~(IN1 & IN2);
endmodule
```

Проте такі описи робити не потрібно. Це базові елементи (gates) і вони напевно мають бути в стандартних бібліотеках синтезатора.

Далі ще пара важливих логічних елементів (рис. 3.2):



OR2	0	1	x	z
0	0	1	X	X
1	1	1	1	1
x	X	1	X	X
z	X	1	X	X

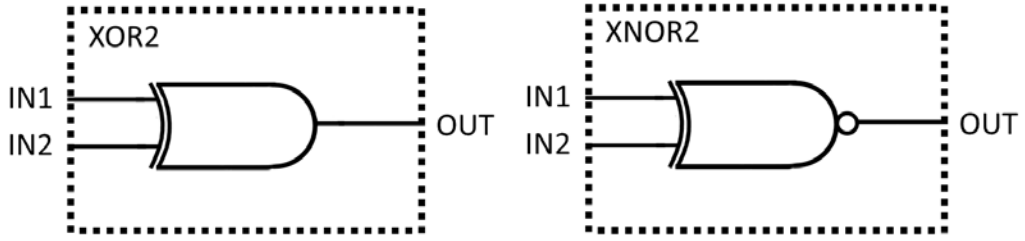
NOR2	0	1	x	z
0	1	0	X	X
1	0	0	0	0
x	X	0	X	X
z	X	0	X	X

Рисунок 3.2 – Логічні елементи та їхні таблиці істинності

Зліва логічний елемент «АБО». На виході одиниця, якщо на першому або другому вході одиниця. Праворуч – логічний елемент «АБО-НЕ». На виході нуль, якщо на першому або другому вході – одиниця.

Наступні важливі примітиви пов’язані з «запереченням рівнозначності» (рис. 3.3). Елемент зліва (XOR) має на виході нуль якщо на обох входах однакове значення (або обидва входи нуль, або обидва входи одиниця).

Елемент праворуч (XNOR) – теж саме, тільки з інверсією. На виході одиниця, якщо або обидва входи нуль, або обидва входи одиниця.

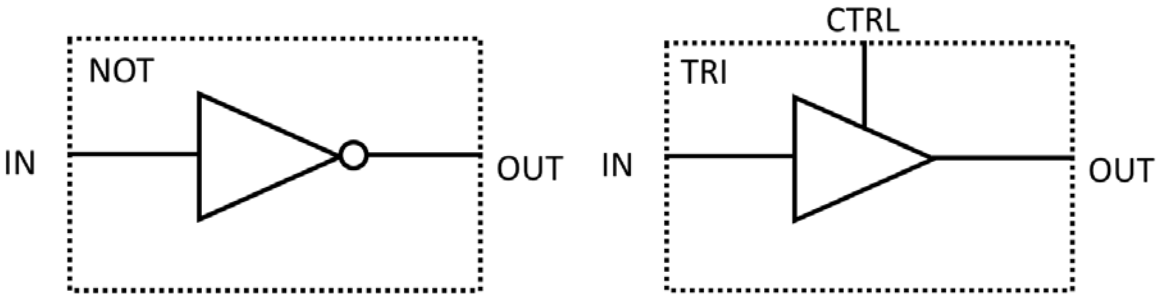


XOR2	0	1	x	z
0	0	1	X	X
1	1	0	X	X
x	X	X	X	X
z	X	X	X	X

XNOR2	0	1	x	z
0	1	0	X	X
1	0	1	X	X
x	X	X	X	X
z	X	X	X	X

Рисунок 3.3 – Логічні елементи та їхні таблиці істинності

Далі ще пара важливих базових елементів:



NOT	OUT
0	1
1	0
X	X
Z	X

TRI	0	1	x	z
0	Z	0	L	L
1	Z	1	H	H
x	Z	X	X	X
z	Z	X	X	X

Рисунок 3.4 – Логічні елементи та їхні таблиці істинності

Зліва елемент «НЕ». На його виході значення протилежне початкового значення. Якщо на вході одиниця, то на виході нуль, і навпаки. Праворуч – буферний елемент, що використовується для двонаправлених сигналів

(inout). Цей елемент «пропускає» через себе вхідний сигнал, тільки якщо на вході CTRL є керуюча одиниця. Якщо на вході CTRL нуль, то елемент «Відключається» від вихідного проводу переходячи у високоомний стан. Такі елементи використовуються тільки для виводів цифрових мікросхем. Іншими словами, використовувати двонаправлені сигнали (inout) доцільно тільки в модулі найвищого рівня. Отже, ми знаємо про основні базові логічні елементи – і це теж модулі. Використовуємо їх в модулі вищого рівня. Зробимо однобітовий суматор за схемою (рис. 3.5).

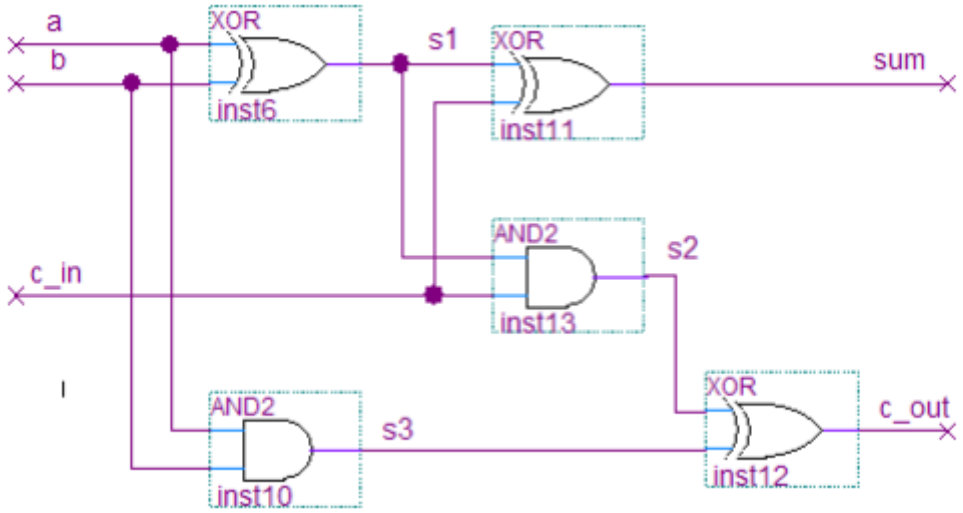


Рисунок 3.5 – Схема однобітного суматора

Цей суматор додає два однобітових числа a і b. При виконанні додавання однобітових чисел може трапитися «переповнення», тобто результат вже буде двобітовим (1 + 1 = 2 або у двійковому вигляді 1'b1 + 1'b1 = 2'b10). Тому у нас є вихідний сигнал перенесення c_out. Додатковий вхідний сигнал c_in слугує для прийому сигналу переносу від суматорів молодших розрядів, при побудові багатобітових суматорів.

Цю схему можна описати мовою Verilog, встановлюючи в тілі модуля екземпляри інших модулів. Цей опис на рівні елементів. Ми використаємо в нашому модулі три модуля XOR та два модуля AND2.

```
module adder1 (output sum, otpput c_out, input a, input b,
input c_in);
    wire s1, s2, s3;
    XOR my_1_xor(.OUT(s1), .IN1(a), .IN2(b));
    AND2 my_1_and2(.OUT(s3), .IN1(a), .IN2(b));
    XOR my_2_xor(.OUT(sum), .IN1(s1), .IN2(c_in));
    AND2 my_2_and2(.OUT(s2), .IN1(s1), .IN2(c_in));
    XOR my_3_xor(.OUT(c_out), .IN1(s2), .IN2(s3));
endmodule
```

Порядок опису модуля такий:

1. Пишемо назву модуля, тип якого нам потрібен.

2. Пишемо назву конкретно цього екземпляра модуля.

3. Описуємо підключення сигналів: точка і потім ім’я сигналу модуля, потім в дужках ім’я провідника, який до нього підключений.

Звичайно, зовсім необов’язково описувати модуль так, як наведено вище. Набагато простіше написати такий код однобітового суматора:

```
module adder1 (output sum, otpput c_out, input a, input b,
input c_in);
    assign sum = (a^b)^c_in;
    assign c_out = ((a^b)&c_in) ^(a&b);
endmodule
```

Просто важливо розуміти, що існують різні методи опису, і потрібно вміти ними усіма користуватися. Тепер у нас є однобітовий суматор і ми можемо зробити, наприклад, чотирибітовий (з послідовним переносом).

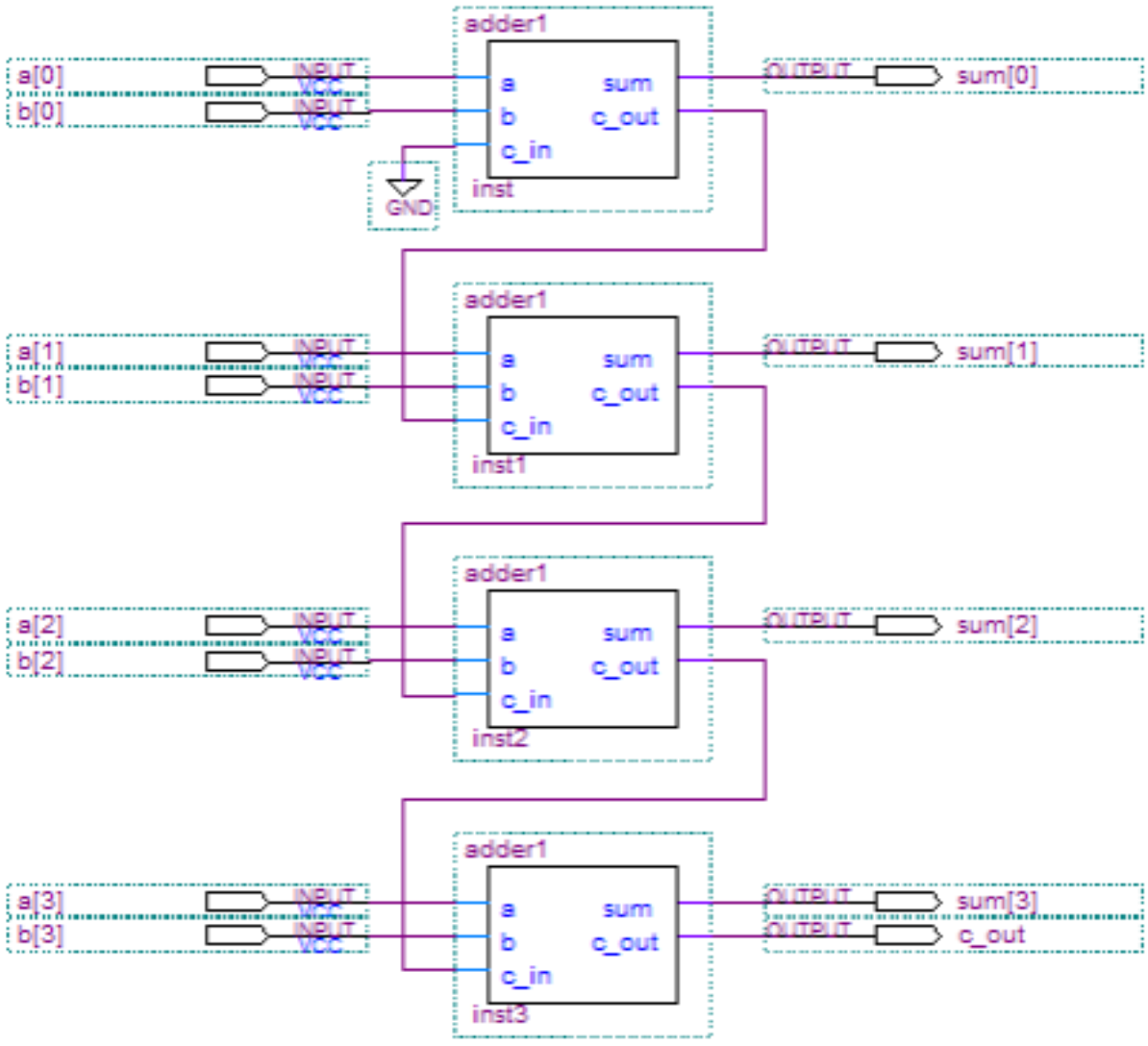


Рисунок 3.6 – Схема чотирибітового суматора
з послідовним переносом

Мовою Verilog це ж буде виглядати таким чином:

```
module adder4 (output [3:0] sum, otpput c_out, input [3:0]a,
input [3:0]b);
    wire c0, c1, c2;
    adder1 my0_adder1 (.sum(sum[0]), .c_out(c0), .a(a[0]),
b(b[0]), .c_in(1'b0));
    adder1 my1_adder1 (.sum(sum[1]), .c_out(c1), .a(a[1]),
b(b[1]), .c_in(c0));
    adder1 my2_adder1 (.sum(sum[2]), .c_out(c2), .a(a[2]),
b(b[2]), .c_in(c1));
    adder1 my3_adder1 (.sum(sum[3]), .c_out(c_out),
.a(a[3]), b(b[3]), .c_in(c2));
endmodule
```

Таким чином, ми реалізували чотирибітовий суматор. Ми отримали його як модуль верхнього рівня adder4, що складається з модулів adder1, які, насамперед, складаються з модулів-примітивів AND2 і XOR.

Хід роботи

1. Створіть складний проект з кількох модулів, об'єднаних за допомогою графічної оболонки програмного середовища Quartus. Для прикладу розглянемо проект «генератора затриманих імпульсів», загальна схема якого подана на рис. 3.7, а алгоритм роботи – на рис. 3.8.

Схема містить два однакових модулі формування імпульсу (PULSE), модуль затримки (DELAY), і модуль подільника частоти (CLK DIVIDER). Тривалість імпульсів та тривалість затримки між ними може бути змінена користувачем у деяких межах.

Алгоритм роботи. Після увімкнення схема переходить у режим очікування. По фронту сигналу запуску (TRIGGER) формується перший імпульс і починається розрахунок затримки. При досягненні потрібного значення формується другий імпульс і схема повертається в початковий стан. Можливе також скидання схеми вручну сигналом RESET.

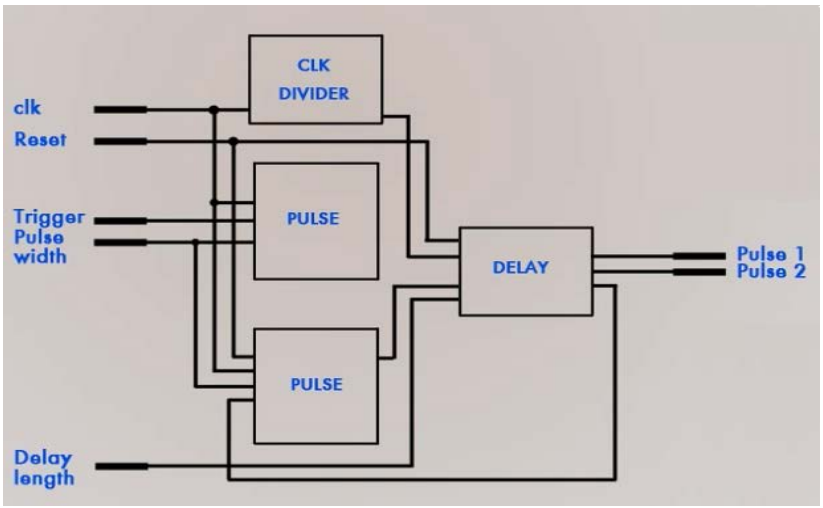


Рисунок 3.7 – Структурна схема генератора затриманих імпульсів

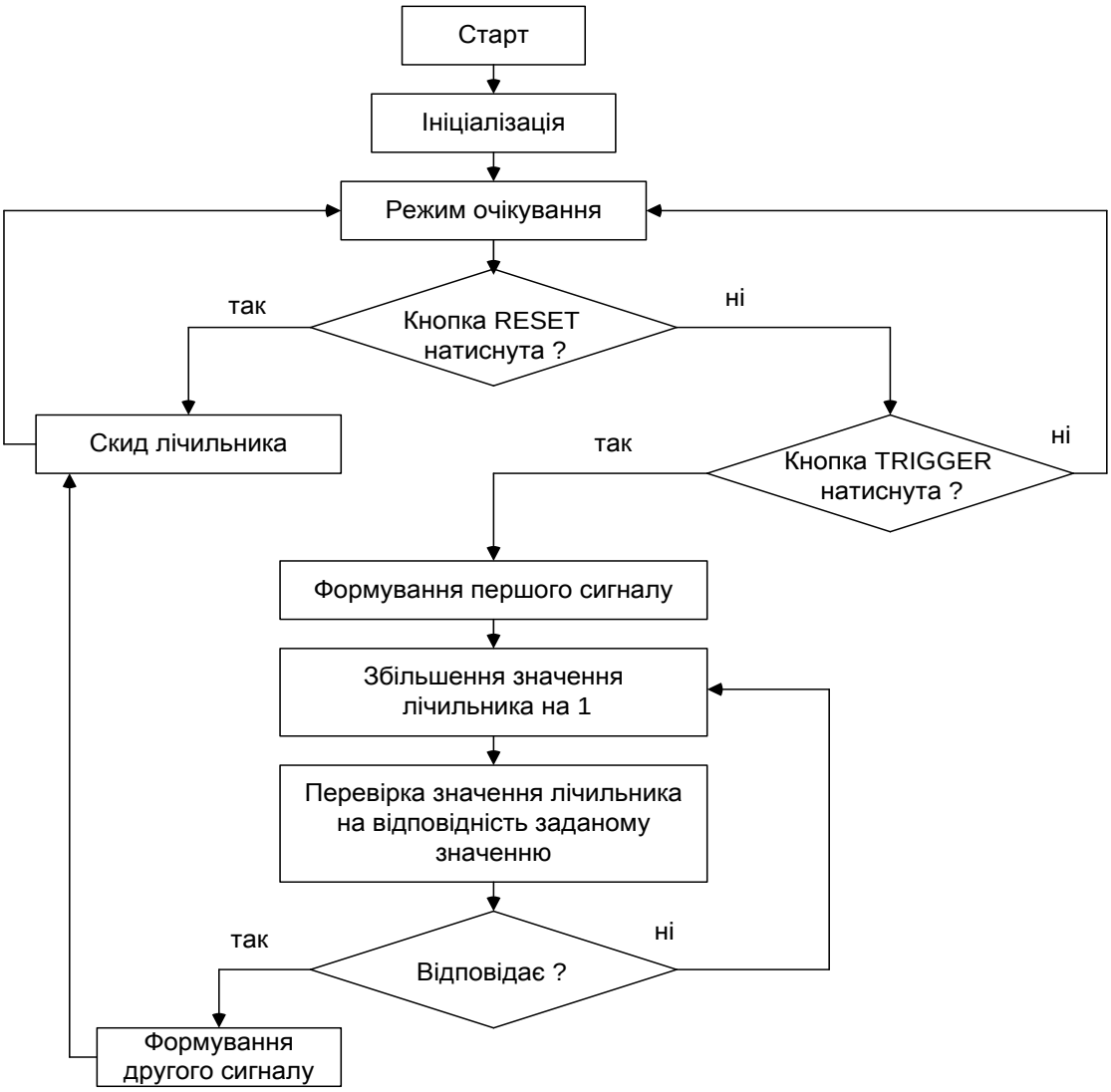


Рисунок 3.8 – Алгоритм роботи генератора затриманих імпульсів

2. Створіть новий проект в Quartus, введіть назву проекту, директорію зберігання, оберіть серію ПЛІС, яку будете програмувати.

3. Створіть файл опису мовою Verilog, введіть текст модуля, наведений нижче. Тут CLK – вхід тактової частоти, RESET – сигнал скидання, TRIGGER – сигнал запуску, шина DATA використовується для визначення тривалості імпульсу, OUT – вихід, на якому формується імпульс.

Збережіть файл з назвою «pulse». Виконайте аналіз тексту модуля. Якщо є помилки, то виправте їх і знову виконайте аналіз. Якщо помилок немає, то знову збережіть файл.

```
module pulse
    (
    input CLK, RESET, TRIGGER,
    input [3:0] DATA,
    output OUT
    );
```

```
    reg [3:0] cnt  = 4'b0;
    reg cnt_en = 1'b0;
    wire rst_int = RESET || (cnt == DATA);

    always @ (posedge TRIGGER or posedge rst_int)
    if (rst_int) begin
        cnt_en <= 1'b0;
    end else begin
        cnt_en <= 1'b1;
    end

    always @ (posedge CLK or posedge rst_int)
    if (rst_int)
        cnt <= 0;
    else
    if (cnt_en)
        cnt <= cnt + 1'b1;

    assign OUT = cnt_en;
endmodule
```

4. Створіть наступний модуль (формування затримки між імпульсами). Створіть файл опису мовою Verilog, введіть текст модуля, наведений нижче. Тут два входи IN для першого та другого імпульсу, CLK – вхід тактової частоти, RESET – сигнал скидання, шина DATA – для визначення тривалості затримки, вихід TRIG використовується для формування сигналу запуску для другого модуля формування імпульсу, виходи OUT – для виводу готових сформованих імпульсів. Скомпілюйте, виправте помилки та збережіть модуль.

```
module delay
    ( input [1:2] IN, input CLK, RESET, input [3:0] DATA,
output TRIG, output [1:2] OUT );

    reg [3:0] cnt  = 4'b0;
    reg cnt_en = 1'b0;
    wire rst_int = RESET || (cnt == DATA);

    always @ (posedge IN [1] or posedge rst_int)
    begin
        if (rst_int)
            cnt_en <= 1'b0;
        else
```

```
        cnt_en <= 1'b1;
    end

    always @ (posedge CLK or posedge rst_int)
    begin
      if (rst_int)
        cnt <= 4'b0;
      else if (cnt_en & (!IN [2]))
        cnt <= cnt + 1'b1;
      else
        cnt <= 4'b0;
     end

    assign TRIG = cnt == DATA;
    assign OUT = IN;
endmodule
```

5. Створіть наступний модуль для ділення частоти сигналів з тактового генератора, щоб отримати час тривалості імпульсів і затримки порядку кількох секунд. Скомпілюйте, виправте помилки та збережіть модуль.

```
module clkdivider
    (
    input CLK_IN,
    output CLK_OUT
    );

    reg [23:0] cnt  = 24'b0;

    always @ (posedge CLK_IN)
    cnt <= cnt + 1'b1;

    assign CLK_OUT = cnt [23];

endmodule
```

6. Далі потрібно з'єднати створені модулі в одну схему. Це можна зробити кількома способами: за допомогою однієї з мов опису, або за допомогою графічного вводу. Використаємо графічну оболонку для поєднання модулів в готову схему. Натисніть NEW та створіть новий файл Block Diagram/Schematic File. Quartus створить новий файл формату .bdf.

7. Перед тим як розмістити створені модулі в графічному файлі, необхідно створити для кожного з них символ-файл (рис. 3.9). У вкладці Files навігатора проекту в контекстному меню кожного файлу виберіть «Create Symbol Files for Current File».

8. Для розміщення графічних символів файлів двічі клацніть у полі графічного редактора та в папці Project оберіть створені символи файлів, натисніть OK та розмістіть в графічному редакторі. У проекті потрібно два блоки pulse, тому продублюйте його за допомогою клавіші Ctrl.

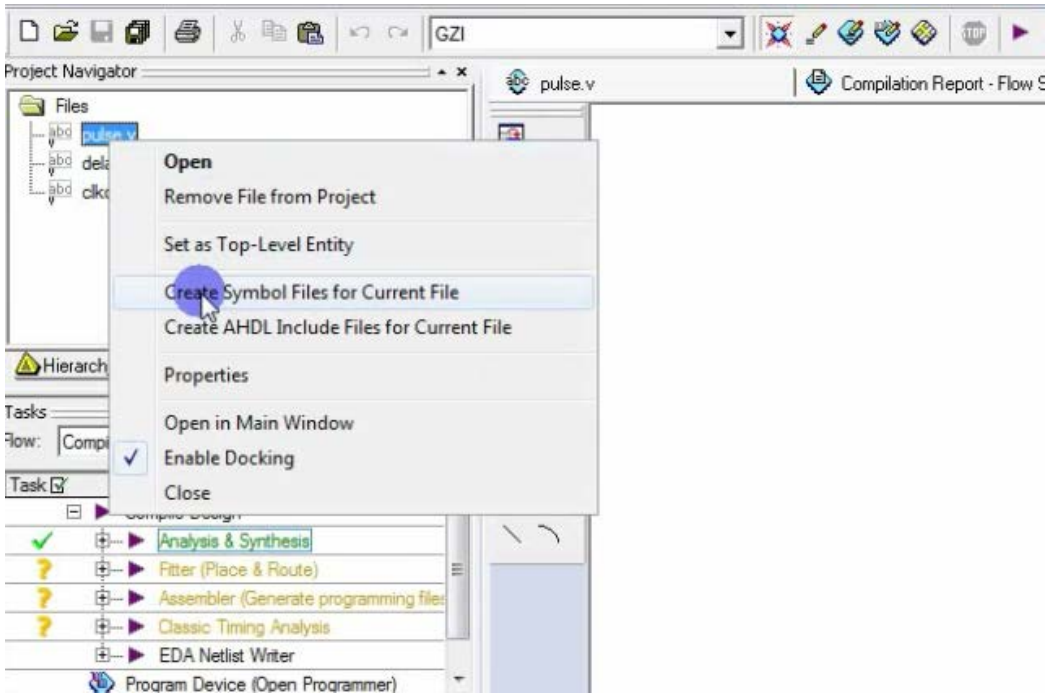


Рисунок 3.9 – Створення символ-файлів для модулів

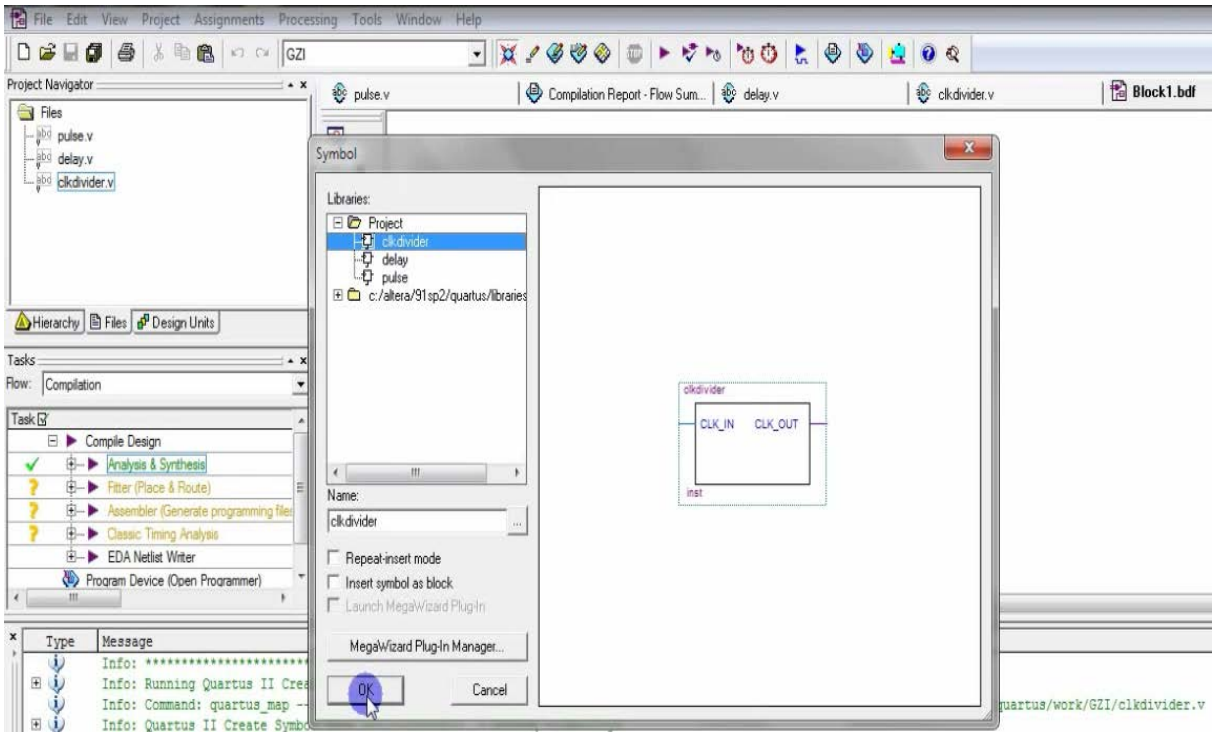


Рисунок 3.10 – Розміщення символів файлів

9. Зберіть схему за рис. 3.11 використовуючи створені символи модулів та стандартні елементи (інвертори NOT та виводи INPUT і OUTPUT). Задайте назву входів та виходів схеми. Зверніть увагу, що назви шин мають містити в квадратних дужках розрядність, (наприклад PULSE_N[3..0]). Якщо шина названа правильним чином, то контакт з неї позначено більш товстою лінією. Виходи з модулів pulse приєднуються до вхідної шини модуля delay. Для того щоб контакт відбувся, потрібно дати відповідні назви виводам та вхідній шині (наприклад, in[1] та in[2] для виходів й in[1..2] для вхідної шини). Щоб дати назву провідникам та шині виділіть їх та правою кнопкою миші оберіть Properties.

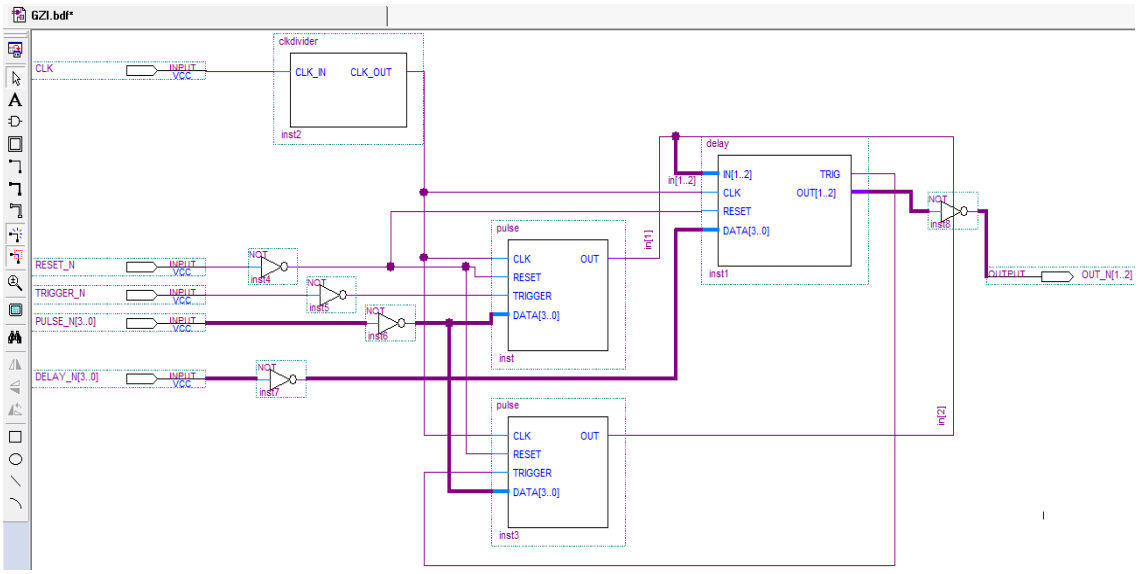


Рисунок 3.11 – Схема пристрою

10. Збережіть графічний файл, призначте його файлом верхнього рівня (Set as top level), скомпілюйте. Якщо є помилки, то виправте і знову збережіть і скомпілюйте.

11. Призначте виводи схеми. Вхід CLK – до тактового генератора 50 МГц, входи RESET_N і TRIGGER_N – до кнопок, шини PULSE_N[3..0] і DELAY_N[3..0] – до перемикачів, якими будете задавати тривалість імпульсу та затримки, вихід OUT_N[1..2] – до двох світлодіодів.

12. Скомпілюйте проект знову та запрограмуйте ПЛІС. Перевірте роботу проекту. Зробіть висновки про здобуті знання та навички.

Перелік контрольних запитань

1. Опишіть ієрархію складного проекту, поняття файлу верхнього рівня.
2. Опишіть поняття модулів в текстовому вигляді і в графічному вигляді.
3. Опишіть текстом мови Verilog модулі, що відповідають базовим логічним елементам.
4. Яким чином створюються графічні аналоги текстових модулів?
5. Яким чином текстом описати зв'язок виходів одного модуля із входами іншого?
6. Яким чином створити шину в графічному файлі?

ЛАБОРАТОРНА РОБОТА № 4
Арифметичні та логічні функції в мові Verilog

Мета роботи: освоїти лексику мови Verilog для виконання логічних та арифметичних операцій. Закріпити практичні навички зі створення складних проектів. Створити проект простого калькулятора.

Теоретичні відомості

Ознайомимося з основними арифметичними і логічними операторами мови Verilog.

Додавання і віднімання.

Ось приклад модуля, який одночасно і складає і віднімає два числа. Тут вхідні операнди у нас 8-бітові, а результат 9-бітовий. Verilog коректно згенерує біт переносу (carry bit) і помістить його в дев'ятий біт вихідного результату. З погляду Verilog, вхідні операнди беззнакові. Якщо потрібна знакова арифметика, то про це потрібно окремо подбати.

```
module simple_add_sub ( operandA, operandB,
out_sum, out_dif);
// два вхідних 8-бітових операнди
    input [7:0] operandA, operandB;
 /* виходи для арифметичних операцій мають
додатковий  9-й біт переповнення */
    output [8:0] out_sum, out_dif;
    assign out_sum = operandA + operandB; // додавання
    assign out_dif = operandA - operandB; // віднімання
endmodule
```

Бітові логічні операції.

Бітові операції в Verilog виглядають так само, як і в мові С. Кожен біт результату обчислюється окремо відповідно бітам операндів.

Нижче наведено приклад бітових логічних операцій мовою Verilog.

```
module simple_bit_logic ( operandA, operandB, out_bit_and,
out_bit_or, out_bit_xor, out_bit_not);
    input [7:0] operandA, operandB;
    output [7:0] out_bit_and, out_bit_or, out_bit_xor,
out_bit_not;
    assign out_bit_and = operandA & operandB; // І
    assign out_bit_or = operandA | operandB; // АБО
    assign out_bit_xor = operandA ^ operandB; // виключне
АБО
```

```
    assign out_bit_not = ~operandA; // НЕ
endmodule
```

Булеві логічні операції.

Булеві логічні оператори відрізняються від бітових операцій. Так само, як і в мові С, тут значення всієї шини розглядається як TRUE («1») якщо хоча б один біт в шині одиниця або FALSE («0»), якщо всі біти шини – нуль. Результат виходить завжди однобітовий незалежно від розрядності операндів.

```
module simple_bit_logic ( operandA, operandB,
out_bool_and, out_bool_or, out_bool_not);
    input [7:0] operandA, operandB;
    output out_bool_and, out_bool_or, out_bool_not;
    assign out_bool_and = operandA && operandB; // І
    assign out_bit_or = operandA || operandB; // АБО
    assign out_bit_not = !operandA; // НЕ
endmodule
```

Оператори редукції.

Verilog має оператори редукції. Ці оператори дозволяють виконувати операції між бітами всередині однієї шини. Так, можна визначити, чи всі біти в шині дорівнюють одиниці (& bus), або чи є в шині хоча б одна одиниця (| bus). Наприклад:

```
module simple_reduction_logic ( operandA,
out_reduction_and, out_reduction_or, out_reduction_xor);
    input [7:0] operandA;
    output out_reduction_and, out_reduction_or,
out_reduction_xor;
    assign out_reduction_and = &operandA;
    assign out_reduction_or = |operandA;
    assign out_reduction_xor = ^operandA;
endmodule
```

Також корисні оператори редукції: ~ | operandA означає, що в шині немає одиниць, ~ & operandA означає, що деякі біти в шині дорівнюють нулю.

Оператор умовного вибору.

Мова С має оператор «? : ». З його допомогою можна вибрати одне значення з двох за результатом логічного виразу. У Verilog теж є подібний оператор, він фактично реалізує мультиплексор. У цьому прикладі на

виході мультиплексора виявиться значення operandA, якщо сигнал sel_in одиниця. І навпаки. Якщо вхідний сигнал sel_in дорівнює нулю, то на виході мультиплексора буде значення operandB.

```
module simple_mux ( operandA, operandB, sel_in, out_mux );

    input [7:0] operandA, operandB;
    input sel_in;  //вхідний сигнал селектора
    output [7:0] out_mux;
    assign out_mux = sel_in ? operandA : operandB;

endmodule
```

Оператори порівняння.

Змінні отримають значення логічної одиниці, якщо порівняння підтвердилось, і – логічного нуля, якщо ні.

```
module simple_compare ( operandA, operandB, out_eq,
out_ne, out_gt, out_lt, out_ge, out_le);

    input [7:0] operandA, operandB;
    //виходи операцій порівняння
    output out_eq, out_ne, out_gt, out_lt, out_ge, out_le;
    assign out_eq = operandA == operandB; //рівно
    assign out_ne = operandA != operandB; //нерівно
    assign out_ge = operandA >= operandB; //більше або
рівно
    assign out_le = operandA <= operandB; //менше або
рівно
    assign out_gt = operandA > operandB; //більше
    assign out_lt = operandA < operandB; //менше

endmodule
```

Хід роботи

1. Створіть проект для додавання та віднімання двох чисел, заданих в двійковій формі, результат повинен виводитись також в двійковій формі на світлодіодні індикатори. Вибір операції (додавання чи віднімання) повинен здійснюватись перемикачем на платі. Запрограмуйте ПЛІС та перевірте роботу проекту.

2. Створіть проект для виконання бітових логічних операцій над двома числами, заданими в двійковій формі. Запрограмуйте та перевірте роботу.

3. Створіть проект для перевірки булевих логічних операцій над двійковими числами. Запрограмуйте та перевірте роботу.

4. Створіть проект для перевірки роботи операторів редукції. Запрограмуйте та перевірте роботу.

5. Створіть проект простого мультиплексора. Запрограмуйте та перевірте роботу.

6. Створіть проект для перевірки операторів порівняння. Запрограмуйте та перевірте роботу.

7. Створіть проект простого калькулятора з виводом результату в десятковій формі на 7-сегментні індикатори. Запрограмуйте та перевірте роботу.

8. Зробіть висновки про здобуті знання та навички.

Перелік контрольних запитань

1. Як відбувається додавання і віднімання чисел в мові Verilog? Напишіть тексти для додавання і віднімання чисел.

2. Опишіть роботу логічних операторів: «&»,«|»,«^»,«~».

3. Опишіть принцип роботи булевих логічних операторів: «&&»,«||»,«!».

4. Що таке оператори редукції? Опишіть їхню роботу.

5. Поясніть принцип роботи оператора умовного вибору.

6. Поясніть принцип роботи операторів порівняння.

ЛАБОРАТОРНА РОБОТА № 5
Процедурні блоки в мові Verilog

Мета роботи: освоїти лексику мови Verilog для виконання процедурних операцій. Закріпити навички зі створення проектів.

Теоретичні відомості

Мова Verilog має так звані процедурні «*always*» блоки. Використання процедурних блоків дуже схоже на програмування мовою С. Воно дозволяє виразити алгоритм так, щоб він виглядав як послідовність дій. Для опису процедурного блоку використовується такий синтаксис:

```
always @ (<sensitivity list>) <statement>
```

<*sensitivity list*> – це список всіх вхідних сигналів, до яких чутливий блок. Тобто список вхідних сигналів, зміна яких впливає на вихідні сигнали цього блоку. «A*lways*» перекладається як «завжди». Отже, такий запис можна прочитати як: «Завжди виконувати вирази <statement> при зміні сигналів, описаних у списку чутливості <sensitivity list>».

У списку чутливості імена всіх вхідних сигналів розділяються ключовим словом «or»:

```
always @ (a or b or d) <statement>
```

Іноді набагато простіше і надійніше долучити до списку чутливості всі сигнали. Це робиться так:

```
always @* <statement>
```

При описанні виразів всередині процедурних блоків комбінаторної логіки з правої сторони від знаку рівності, як і раніше, можна використовувати типи сигналів *wire* або *reg*, а з лівої сторони тепер використовується лише тип *reg*.

```
reg [3:0] c;
always @ (a or b or d)
begin
   c = <вираз, що використовує вхідні сигнали a, b, d >
end
```

Зверніть увагу, що регістри, яким надано присвоєння в таких процедурних блоках, не будуть виконані у вигляді D-тригерів після синтезу. Тут ми робимо присвоєння регістрам за допомогою оператора « = », який називається «блокуючим». Для симулятора це означає, що вираз обчислюється, його результат присвоюється регістру приймача і він тут же, негайно, може бути використаний у подальших виразах. Блокуючі присвоєння зазвичай використовуються для опису комбінаторної логіки в процедурних блоках. Не блокуючі присвоєння зазвичай використовуються для опису синхронної логіки і вже там регістри ***reg*** після синтезу будуть представлені за допомогою D-тригерів. Розрізняйте блокуючі і не блокуючі присвоєння!

Ви можете написати досить багато виразів пов'язаних між собою, синтезатор переробить їх і створить, можливо, досить довгі ланцюги з комбінаторної логіки. Наприклад:

```
wire [3:0] a, b, c, d, e;
reg [3:0] f, g, h, j;
always @(a or b or c or d or e)
begin f = a + b; g = f & c; h = g | d; j = h - e;
end
```

Те ж саме можна зробити по іншому:

```
always @(a or b or c or d or e)
begin j = (((a + b) & c) | d) - e;
end
```

Насправді, після того, як проект буде скомпільований, список всіх сигналів проекту (netlist) може сильно скоротитися. Багато сигналів, описаних програмістом, можуть зникнути – синтезатор викине їх, створивши ланцюг з оптимізованої комбінаторної логіки. У нашому прикладі сигнали *f*, *g* і *h* можуть зникнути зі списку сигналів проекту після синтезатора, все залежить від того чи використовуються ці сигнали ще десь в проекті чи ні. Синтезатор навіть може видати попередження (warning) про те, що сигналу *f* присвоєно значення, але воно ніде не використовується.

Описати простий мультиплексор можна за допомогою оператора «?»:

```
reg [3:0] c;
always @(a or b or d)
begin c = d ? (a & b) : (a + b);
end
```

Можемо написати це ж саме, але по іншому:

```
reg [3:0] c;
always @(a or b or d)
begin if (d)
begin c = a & b;
end
else begin c = a + b;
end
end
```

Якщо потрібно зробити вибір з багатьох варіантів (це мовою схемотехніки мультиплексор з багатьма входами), то можна використовувати конструкцію *case*. Конструкції *case* дуже схожі на *switch* з мови С. Базовий синтаксис такий:

```
case (selector) option1: ; option2: ; default: ;
endcase
```

Простий приклад:

```
wire [1:0] option;
wire [7:0] a, b, c, d;
reg [7:0] e;
always @(a or b or c or d or option) begin
case (option)
0: e = a;
1: e = b;
2: e = c;
3: e = d;
endcase
end
```

Далі розглянемо цикли. Тут потрібно зробити зауваження, що цикли у Verilog мають дещо інший сенс, не такий, як у мовах C або Pascal. Мовою C цикл позначає, що деяка дія має бути виконана послідовно раз за разом. Чим більше ітерацій в циклі, тим довше він буде виконуватися процесором. На мові Verilog цикл описує скільки зразків логічних функцій має бути реалізовано апаратно. Цикл повинен мати обмежене число ітерацій, яке можна однозначно визначити на етапі синтезу.

Розглянемо простий приклад – потрібно визначити номер найстаршого ненульового біта вектора (шини):

```
module find_high_bit (input wire [7:0] in_data, output reg [2:0]
high_bit, output reg valid);
integer i;
always @ (in_data)
begin
valid = |in_data; // визначимо, чи є в шині одиниці
high_bit = 0;
for (i = 0; i < 8; i = i + 1)
begin
if (in_data[i])
begin
high_bit = i; // запам'ятаємо номер біта з одиницею в шині
end
end
end
endmodule
```

У наведеному прикладі в циклі переглядаються всі біти послідовно від молодшого до старшого. Коли буде знайдений ненульовий біт, то його порядковий номер запам’ятається в регістрі *high_bit*. Тут *high_bit* – це просто змінна типу *reg*, а не апаратний регістр. Коли цикл закінчиться, то змінна *high_bit* буде містити індекс найстаршого ненульового біта в шині (якщо *valid* не нуль).

Насправді, звичайно, потрібно уявляти собі, що подібний запис циклу буде реалізований в апаратурі досить довгим ланцюжком з мультиплексорів. В цьому прикладі цикл *for* – це приблизно така логічна схема (рис. 5.1).

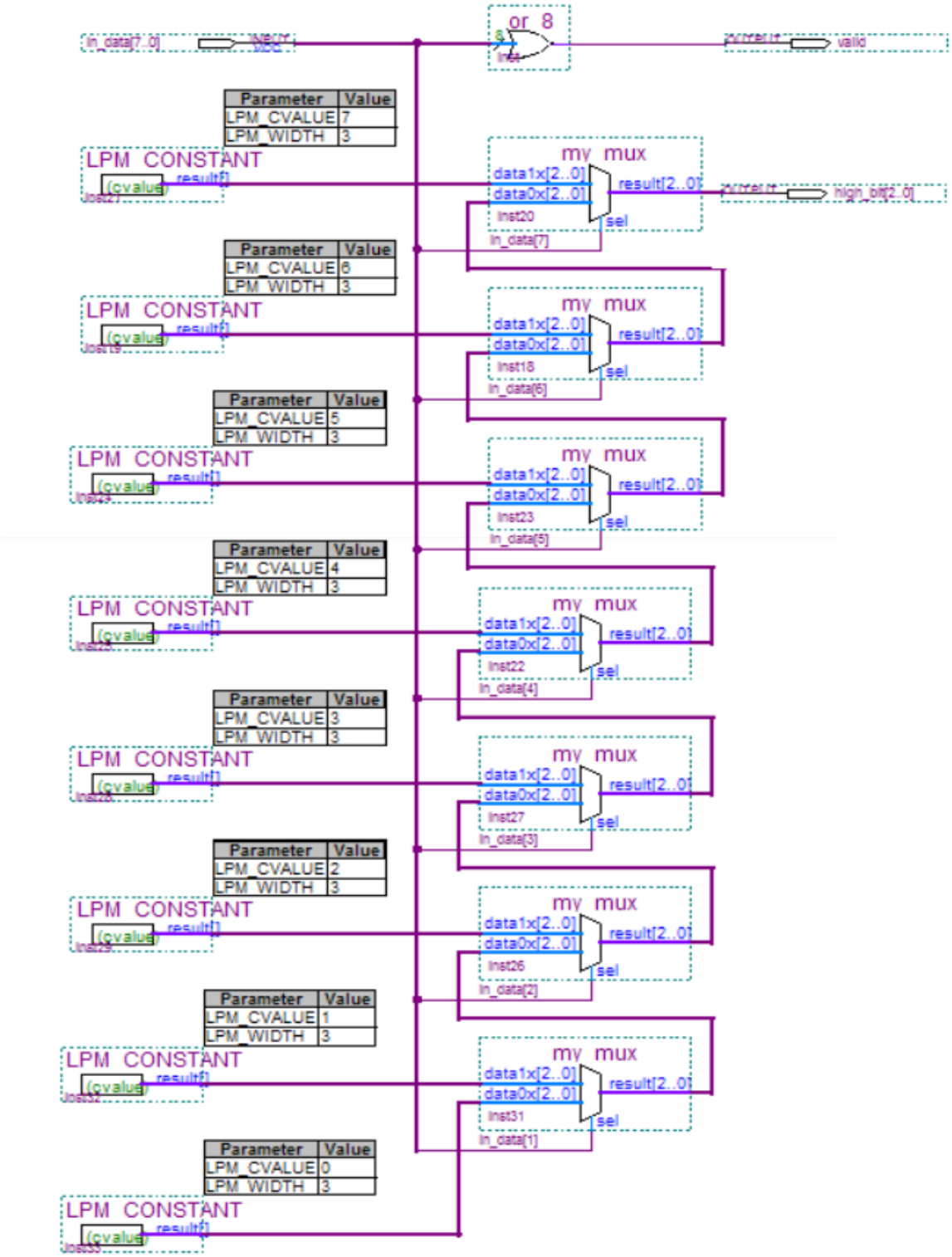


Рисунок 5.1 – Апаратна реалізація циклу з 8 ітерацій

Розглянемо інший приклад – потрібно знайти наймолодший ненульовий біт в шині. Тепер перегляд бітів йде від старшого до молодших.

```
module find_low_bit (input wire [7:0] in_data, output reg [2:0] low_bit,
output reg valid);
integer i;
always @ (in_data)
begin
   valid = |in_data; // визначимо, чи є в шині одиниці
  low_bit = 0;
   for (i = 7; i >= 0; i = i-1)
   begin
        if (in_data[i])
        begin
        low_bit = i; // запам'ятаємо номер біта з одиницею в шині
        end
    end
end
endmodule
```

Хід роботи

1. Створіть проект, що буде виконувати функцію мультиплексора з трьома трибітовими входами. Запрограмуйте ПЛІС та перевірте роботу.
2. Створіть проект для знаходження найстаршого ненульового біта в 10-бітовій шині. Номер біта виведіть на 7-сегментний індикатор в десяткові формі. Запрограмуйте ПЛІС та перевірте роботу.
3. Створіть проект для знаходження наймолодшого ненульового біта в 10-бітній шині. Номер біта виведіть на 7-сегментний індикатор в десяткові формі. Запрограмуйте ПЛІС та перевірте роботу.
4. Створіть проект для підрахунку кількості одиниць в 10-бітовій шині. Номер біта виведіть на 7-сегментний індикатор в десяткові формі. Запрограмуйте ПЛІС та перевірте роботу.

Перелік контрольних запитань

1. Опишіть поняття процедурного блоку.
2. Яким чином створити запис процедурного блоку?
3. Як Ви розумієте список чутливості процедурного блоку?
4. Що означають параметри «posedge» та «negedge» в списку чутливості?
5. Опишіть як працює умовний оператор «if». Наведіть приклад.
6. Опишіть як працює оператор циклу «for». Наведіть приклад.
7. Опишіть роботу оператора умовного вибору «?». Наведіть приклад.

ЛАБОРАТОРНА РОБОТА № 6 (два заняття)

Робота з VGA-адаптером, виведення зображень на монітор

Мета роботи: ознайомитись з особливостями роботи VGA-адаптера, навчитись створювати проекти з виводом зображень на монітор.

Теоретичні відомості

Спочатку VGA створювався для моніторів з електронно-променевою трубкою. VGA складається з таких основних підсистем.

1. Графічний контролер (Graphics Controller), за допомогою якого відбувається обмін даними між центральним процесором і відеопам'яттю. Має можливість виконувати бітові операції над переданими даними.
2. Відеопам'ять (Display Memory), у якій розміщуються дані, які відображаються на екрані монітора. 256 кБ DRAM розділені на чотири колірних шари по 64 кБ.
3. Послідовний перетворювач (Serializer або Sequencer) перетворює дані з відеопам'яті в потік бітів, що передається контролеру атрибутів.
4. Контролер атрибутів (Attribute Controller) за допомогою палітри перетворює вхідні дані в колірні значення.
5. Синхронізатор (Sequencer) керує тимчасовими параметрами відеоадаптера і перемиканням колірних шарів.
6. Контролер ЕПТ (CRT Controller) – генерує сигнали синхронізації для ЕПТ.

Як синхронізатор у нас буде генератор на 50 МГц.

Отже, завдання розбивається на реалізацію таких блоків:

1. Синхронізатор – частота 50 МГц для VGA не підходить, тому вона буде дещо змінена.
2. Контролер ЕПТ, який буде видавати сигнали синхронізації відповідно до вимог стандарту VGA.
3. Генератор зображення – блок буде створювати просте зображення і видавати її контролеру ЕПТ.

Синхронізатор

Із довідкових даних (рис. 6.1) – для VGA частота синхрогенератора має бути 25,175 МГц. Для того щоб із 50 МГц створити 25,175 МГц, можемо скористатись інструментом PLL (подільником частоти).

Parameter	Value	Unit
Pixel clock frequency	25.175	MHz[12]
Horizontal frequency	31.4686	kHz
Horizontal pixels	640	
Horizontal sync polarity	Negative	
Total time for each line	31.77	µs
Front porch (A)	0.94	µs
Sync pulse length (B)	3.77	µs
Back porch (C)	1.89	µs
Active video (D)	25.17	µs

Parameter	Value	Unit
Vertical lines	480	
Vertical sync polarity	Negative	
Vertical frequency	59.94	Hz
Front porch (E)	0.35	ms
Sync pulse length (F)	0.06	ms
Back porch (G)	1.02	ms
Active video (H)	15.25	ms

Рисунок 6.1 – Довідкові дані стандарту VGA

За допомогою Mega Wizard можна створити PLL з потрібними налаштуваннями. Для цього зайдемо в Tools / MegaWizard Plug-InManager. Там вибираємо тип вихідного файлу Verilog HDL I / O ALTPLL і налаштовуємо його – встановлюємо вхідну частоту і потім її перетворюємо. Нам потрібно отримати частоту 25,175 МГц, при цьому ми можемо тільки множити і ділити на ціле число, тобто нам потрібно перевести число 0,5035 у дріб. Можна помножити на 5035 і поділити на 10000, але зауважимо, що можна скоротити обидва числа на 5. Отримуємо дріб 1007 / 2000. Включаємо отриманий модуль у проект підключаємо до його входу генератор 50 МГц, на виході отримаємо частоту 25,175 МГц.

Контролер ЕПТ

Потрібно створити модулі, які видають синхросигнали, що задовольняють таймінгам горизонтальної і вертикальної синхронізації.

Перед видачею даних потрібно потримати лінію синхронізації у високому стані час A (рис. 6.2), потім в низькому стані час B, далі перевести у високий стан, потримати час C і тільки потім відбувається потік даних – промінь починає пробігати стрічку за час D і за цей час потрібно встигнути встановити всі пікселі з частотою 25,175 МГц, і так для кожної стрічки.

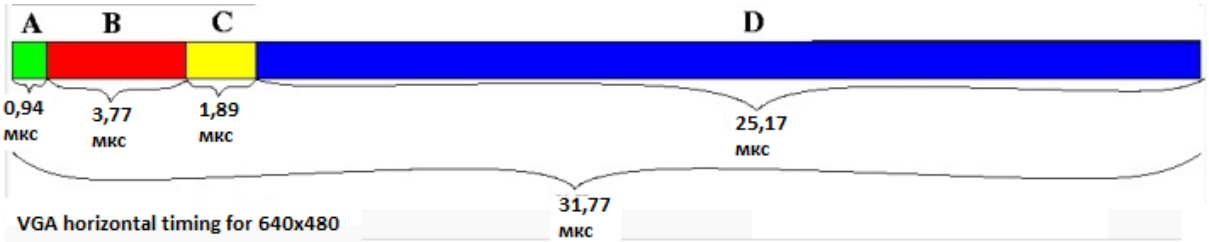


Рисунок 6.2 – Діаграма горизонтального таймінгу

Вертикальна синхронізація працює майже так само, тільки замість пікселів передаються рядки і за один період синхроімпульсу прорисовується весь кадр.

Модуль синхронізації

Для реалізації такого модуля синхронізації потрібно 24 такти на час A, 95 на час B, 48 на час C і 640 на час D. Для вертикальної синхронізації зручніше використовувати тактування не по тактах вхідної частоти 25,175 МГц, а за переднім фронтам сигналу горизонтальної синхронізації, тоді всі затримки можна рахувати в періодах прорисовки одного рядка. Крім того, модуль має повідомляти поточні координати променя для того, щоб задавати значення кольору кожному пікселю. Екземпляр модуля наведений нижче.

```
module VGA_SYNC(
                                                input CLK,
                                                input SYNC_RST_N,
                                                output reg H_SYNC_CLK,
                                                output reg V_SYNC_CLK,
                                                output wire [10:0]oCurrent_X,
                                                output wire [10:0]oCurrent_Y,
                                                output reg oSYNC_COLOR);
parameter A_TIME_H = 24;
parameter B_TIME_H = 95;
parameter C_TIME_H = 48;
parameter D_TIME_H = 640;
parameter TOTAL_TIME_H = A_TIME_H + B_TIME_H + C_TIME_H + D_TIME_H;
parameter BLANK_H = A_TIME_H + B_TIME_H + C_TIME_H;

parameter A_TIME_V = 10;
parameter B_TIME_V = 2;
parameter C_TIME_V = 33;
parameter D_TIME_V = 480;
parameter TOTAL_TIME_V = A_TIME_V + B_TIME_V + C_TIME_V + D_TIME_V;
parameter BLANK_V = A_TIME_V + B_TIME_V + C_TIME_V;


reg                         [10:0]H_Counter;
reg                         [10:0]V_Counter;

assign   oCurrent_X         =   (H_Counter >= BLANK_H) ? H_Counter-BLANK_H   : 11'h0 ;
assign   oCurrent_Y         =   (V_Counter >= BLANK_V) ? V_Counter-BLANK_V   : 11'h0 ;

always@(posedge(CLK) or negedge(SYNC_RST_N))
begin
        if(!SYNC_RST_N)
        begin
                H_Counter         <=       1'b0;
                H_SYNC_CLK        <=       1'b1;
        end
        else
        begin
                if(H_Counter < (TOTAL_TIME_H-1))
                H_Counter         <=       H_Counter + 1'b1;
                else
                        begin
                        H_Counter         <=       1'b0;
                        oSYNC_COLOR <= 1'b0;
                        end
                if(H_Counter == A_TIME_H-1)
                H_SYNC_CLK        <=       1'b0;
```

```
                if(H_Counter == A_TIME_H + B_TIME_H-1)
                H_SYNC_CLK      <=      1'b1;
                if(H_Counter == BLANK_H)
                oSYNC_COLOR <= 1'b1;
        end
end

always@(posedge(H_SYNC_CLK) or negedge(SYNC_RST_N))
begin
        if(!SYNC_RST_N)
        begin
                V_Counter       <=      1'b0;
                V_SYNC_CLK      <=      1'b1;
        end
        else
        begin
                if(V_Counter < (TOTAL_TIME_V-1))
                V_Counter       <=      V_Counter + 1'b1;
                else
                V_Counter       <=      1'b0;
                if(V_Counter == A_TIME_V-1)
                V_SYNC_CLK      <=      1'b0;
                if(V_Counter == A_TIME_V + B_TIME_V-1)
                V_SYNC_CLK      <=      1'b1;
        end
end
endmodule
```

Модуль виводу

Після того як імпульси згенеровані, координати точки відомі, залишилось лише вчасно вмикати потрібний колір, поки промінь пробігає рядок. Для цього потрібно створити модуль, якому на вхід подаються сигнали синхронізації, 3 шини кольору і координати, а вихід – 4-бітовий R-2R ЦАП для кожного кольору. Екземпляр модуля наведений нижче.

```
module VGA_OUT(RESET,
                                        SYNC_COLOR,
                                        VGA_CLK,
                                        oVGA_R,
                                        oVGA_G,
                                        oVGA_B,
                                        iVGA_R,
                                        iVGA_G,
                                        iVGA_B,
                                        Current_X,
                                        Current_Y);
input wire VGA_CLK;
input wire RESET;
input wire SYNC_COLOR;
input wire [10:0]Current_X;
input wire [10:0]Current_Y;
input wire [3:0]iVGA_R;
input wire [3:0]iVGA_G;
input wire [3:0]iVGA_B;
output reg [3:0]oVGA_R;
output reg [3:0]oVGA_G;
output reg [3:0]oVGA_B;
always@(posedge VGA_CLK or negedge RESET)
begin
        if(!RESET)
        begin
                oVGA_R   <=      0;
                oVGA_G   <=      0;
                oVGA_B   <=      0;
```

```
        end
        else
        begin
        oVGA_R <= ((SYNC_COLOR == 1)&&(Current_X > 0)&&(Current_Y > 0)) ? iVGA_R :
0;
        oVGA_G <= ((SYNC_COLOR == 1)&&(Current_X > 0)&&(Current_Y > 0)) ? iVGA_G :
0;
        oVGA_B <= ((SYNC_COLOR == 1)&&(Current_X > 0)&&(Current_Y > 0)) ? iVGA_B :
0;
        end
end
endmodule
```

Модуль генерації бітстріма (зображення на екран)

```
module   VGA_BITSTREAM    (        //      Read Out Side
                                                oRed,
                                                oGreen,
                                                oBlue,
                                                iVGA_X,
                                                iVGA_Y,
                                                iVGA_CLK,
                                                //      Control Signals
                                                iRST_n,
                                                iColor_SW        );
//      Read Out Side
output   reg      [9:0]    oRed;
output   reg      [9:0]    oGreen;
output   reg      [9:0]    oBlue;
input    [9:0]             iVGA_X;
input    [9:0]             iVGA_Y;
input                      iVGA_CLK;
//      Control Signals
input                      iRST_n;
input                      iColor_SW;

always@(posedge iVGA_CLK or negedge iRST_n)
begin
        if(!iRST_n)
        begin
                oRed     <=       0;
                oGreen   <=       0;
                oBlue    <=       0;
        end
        else
        begin
                if(iColor_SW == 1)
                begin
                        if (iVGA_Y<120)
                        begin
                                oRed     <=       (iVGA_X<40)?             0    :
                                                  (iVGA_X>=40 && iVGA_X<80)?       1    :
                                                  (iVGA_X>=80 && iVGA_X<120)?      2    :
                                                  (iVGA_X>=120 && iVGA_X<160)?     3    :
                                                  (iVGA_X>=160 && iVGA_X<200)?     4    :
                                                  (iVGA_X>=200 && iVGA_X<240)?     5    :
                                                  (iVGA_X>=240 && iVGA_X<280)?     6    :
                                                  (iVGA_X>=280 && iVGA_X<320)?     7    :
                                                  (iVGA_X>=320 && iVGA_X<360)?     8    :
                                                  (iVGA_X>=360 && iVGA_X<400)?     9    :
                                                  (iVGA_X>=400 && iVGA_X<440)?     10   :
                                                  (iVGA_X>=440 && iVGA_X<480 )?    11   :
                                                  (iVGA_X>=480 && iVGA_X<520 )?    12   :
                                                  (iVGA_X>=520 && iVGA_X<560 )?    13   :
                                                  (iVGA_X>=560 && iVGA_X<600 )?    14   :
                                                                                   15   ;
                                oGreen   <=       0;
                                oBlue    <=       0;
```

```
                        end
                        else if (iVGA_Y>=120 && iVGA_Y<240)
                        begin
                                oRed    <=      0;
                                oGreen  <=      (iVGA_X<40)?            15  :
                                                (iVGA_X>=40 && iVGA_X<80)?      14  :
                                                (iVGA_X>=80 && iVGA_X<120)?     13  :
                                                (iVGA_X>=120 && iVGA_X<160)?    12  :
                                                (iVGA_X>=160 && iVGA_X<200)?    11  :
                                                (iVGA_X>=200 && iVGA_X<240)?    10  :
                                                (iVGA_X>=240 && iVGA_X<280)?    9   :
                                                (iVGA_X>=280 && iVGA_X<320)?    8   :
                                                (iVGA_X>=320 && iVGA_X<360)?    7   :
                                                (iVGA_X>=360 && iVGA_X<400)?    6   :
                                                (iVGA_X>=400 && iVGA_X<440)?    5   :
                                                (iVGA_X>=440 && iVGA_X<480 )?   4   :
                                                (iVGA_X>=480 && iVGA_X<520 )?   3   :
                                                (iVGA_X>=520 && iVGA_X<560 )?   2   :
                                                (iVGA_X>=560 && iVGA_X<600 )?   1   :
                                                                                0   ;
                                oBlue   <=      0;
                        end
                        else if (iVGA_Y>=240 && iVGA_Y<360)
                        begin
                                oRed    <=      0;
                                oGreen  <=      0;
                                oBlue   <=      (iVGA_X<40)?            0   :
                                                (iVGA_X>=40 && iVGA_X<80)?      1   :
                                                (iVGA_X>=80 && iVGA_X<120)?     2   :
                                                (iVGA_X>=120 && iVGA_X<160)?    3   :
                                                (iVGA_X>=160 && iVGA_X<200)?    4   :
                                                (iVGA_X>=200 && iVGA_X<240)?    5   :
                                                (iVGA_X>=240 && iVGA_X<280)?    6   :
                                                (iVGA_X>=280 && iVGA_X<320)?    7   :
                                                (iVGA_X>=320 && iVGA_X<360)?    8   :
                                                (iVGA_X>=360 && iVGA_X<400)?    9   :
                                                (iVGA_X>=400 && iVGA_X<440)?    10  :
                                                (iVGA_X>=440 && iVGA_X<480 )?   11  :
                                                (iVGA_X>=480 && iVGA_X<520 )?   12  :
                                                (iVGA_X>=520 && iVGA_X<560 )?   13  :
                                                (iVGA_X>=560 && iVGA_X<600 )?   14  :
                                                                                15  ;
                        end
                        else
                        begin
                                oRed    <=      (iVGA_X<40)?            15  :
                                                (iVGA_X>=40 && iVGA_X<80)?      14  :
                                                (iVGA_X>=80 && iVGA_X<120)?     13  :
                                                (iVGA_X>=120 && iVGA_X<160)?    12  :
                                                (iVGA_X>=160 && iVGA_X<200)?    11  :
                                                (iVGA_X>=200 && iVGA_X<240)?    10  :
                                                (iVGA_X>=240 && iVGA_X<280)?    9   :
                                                (iVGA_X>=280 && iVGA_X<320)?    8   :
                                                (iVGA_X>=320 && iVGA_X<360)?    7   :
                                                (iVGA_X>=360 && iVGA_X<400)?    6   :
                                                (iVGA_X>=400 && iVGA_X<440)?    5   :
                                                (iVGA_X>=440 && iVGA_X<480 )?   4   :
                                                (iVGA_X>=480 && iVGA_X<520 )?   3   :
                                                (iVGA_X>=520 && iVGA_X<560 )?   2   :
                                                (iVGA_X>=560 && iVGA_X<600 )?   1   :
                                                                                0   ;

                                oGreen  <=      (iVGA_X<40)?            15  :
                                                (iVGA_X>=40 && iVGA_X<80)?      14  :
                                                (iVGA_X>=80 && iVGA_X<120)?     13  :
                                                (iVGA_X>=120 && iVGA_X<160)?    12  :
                                                (iVGA_X>=160 && iVGA_X<200)?    11  :
                                                (iVGA_X>=200 && iVGA_X<240)?    10  :
                                                (iVGA_X>=240 && iVGA_X<280)?    9   :
                                                (iVGA_X>=280 && iVGA_X<320)?    8   :
```

```
                                        (iVGA_X>=320 && iVGA_X<360)?    7    :
                                        (iVGA_X>=360 && iVGA_X<400)?    6    :
                                        (iVGA_X>=400 && iVGA_X<440)?    5    :
                                        (iVGA_X>=440 && iVGA_X<480 )?   4    :
                                        (iVGA_X>=480 && iVGA_X<520 )?   3    :
                                        (iVGA_X>=520 && iVGA_X<560 )?   2    :
                                        (iVGA_X>=560 && iVGA_X<600 )?   1    :
                                                                        0    ;

                        oBlue   <=      (iVGA_X<40)?                    15   :
                                        (iVGA_X>=40 && iVGA_X<80)?      14   :
                                        (iVGA_X>=80 && iVGA_X<120)?     13   :
                                        (iVGA_X>=120 && iVGA_X<160)?    12   :
                                        (iVGA_X>=160 && iVGA_X<200)?    11   :
                                        (iVGA_X>=200 && iVGA_X<240)?    10   :
                                        (iVGA_X>=240 && iVGA_X<280)?    9    :
                                        (iVGA_X>=280 && iVGA_X<320)?    8    :
                                        (iVGA_X>=320 && iVGA_X<360)?    7    :
                                        (iVGA_X>=360 && iVGA_X<400)?    6    :
                                        (iVGA_X>=400 && iVGA_X<440)?    5    :
                                        (iVGA_X>=440 && iVGA_X<480 )?   4    :
                                        (iVGA_X>=480 && iVGA_X<520 )?   3    :
                                        (iVGA_X>=520 && iVGA_X<560 )?   2    :
                                        (iVGA_X>=560 && iVGA_X<600 )?   1    :
                                                                        0    ;

                end
        end
        else
        begin
                oRed    <=      (iVGA_Y<120)    ?               3    :
                                (iVGA_Y>=120 && iVGA_Y<240)?    7    :
                                (iVGA_Y>=240 && iVGA_Y<360)?    11   :
                                                                15   ;
                oGreen  <=      (iVGA_X<80)     ?               1    :
                                (iVGA_X>=80 && iVGA_X<160)?     3    :
                                (iVGA_X>=160 && iVGA_X<240)?    5    :
                                (iVGA_X>=240 && iVGA_X<320)?    7    :
                                (iVGA_X>=320 && iVGA_X<400)?    9    :
                                (iVGA_X>=400 && iVGA_X<480)?    11   :
                                (iVGA_X>=480 && iVGA_X<560)?    13   :
                                                                15   ;
                oBlue   <=      (iVGA_Y<60)     ?               15   :
                                (iVGA_Y>=60 && iVGA_Y<120)?     13   :
                                (iVGA_Y>=120 && iVGA_Y<180)?    11   :
                                (iVGA_Y>=180 && iVGA_Y<240)?    9    :
                                (iVGA_Y>=240 && iVGA_Y<300)?    7    :
                                (iVGA_Y>=300 && iVGA_Y<360)?    5    :
                                (iVGA_Y>=360 && iVGA_Y<420)?    3    :
                                                                1    ;
        end
    end
end

endmodule
```

Головний модуль. Для узгодження роботи всіх модулів.

```
module VGA_MAIN(CLOCK_50,
                KEY,
                LEDG,
                VGA_HS,
                VGA_VS,
                VGA_R,
                VGA_G,
                VGA_B,
                SW
                );
input CLOCK_50;
```

```
input [2:0]KEY;
input [9:0]SW;
output           [9:0]LEDG;
output                VGA_HS;                                //       VGA H_SYNC
output                VGA_VS;                                //       VGA V_SYNC
output   [3:0]    VGA_R;                                     //       VGA Red[3:0]
output   [3:0]    VGA_G;                                     //       VGA Green[3:0]
output   [3:0]    VGA_B;                                     //       VGA Blue[3:0]
wire VGA_CLK;
wire H_SYNC_CLK;
wire V_SYNC_CLK;
wire RESET;
wire [10:0]Current_X;
wire [10:0]Current_Y;
wire SYNC_COLOR;
reg      [3:0]    iVGA_R;
reg      [3:0]    iVGA_G;
reg      [3:0]    iVGA_B;
wire     [3:0]    irVGA_R;
wire     [3:0]    irVGA_G;
wire     [3:0]    irVGA_B;
assign irVGA_R[3:0] = iVGA_R[3:0];
assign irVGA_G[3:0] = iVGA_G[3:0];
assign irVGA_B[3:0] = iVGA_B[3:0];
assign RESET = KEY[0];
assign VGA_HS = H_SYNC_CLK;
assign VGA_VS = V_SYNC_CLK;
VGA_PLL          u1
                 (        .inclk0(CLOCK_50),
                          .c0(VGA_CLK)
                 );
VGA_SYNC u2
(        .CLK(VGA_CLK),
         .H_SYNC_CLK(H_SYNC_CLK),
         .V_SYNC_CLK(V_SYNC_CLK),
         .SYNC_RST_N(KEY[0]),
         .oCurrent_X(Current_X),
         .oCurrent_Y(Current_Y),
         .oSYNC_COLOR(SYNC_COLOR));
VGA_OUT u3
(
.oVGA_R(VGA_R[3:0]),
.oVGA_G(VGA_G[3:0]),
.oVGA_B(VGA_B[3:0]),
.iVGA_R(iVGA_R[3:0]),
.iVGA_G(iVGA_G[3:0]),
.iVGA_B(iVGA_B[3:0]),
.VGA_CLK(VGA_CLK),
.Current_X(Current_X),
.Current_Y(Current_Y),
.SYNC_COLOR(SYNC_COLOR),
.RESET(RESET)
);
VGA_BITSTREAM u4(.oRed(irVGA_R),
                                                    .oGreen(irVGA_G),
                                                    .oBlue(irVGA_B),
                                                    .iVGA_X(Current_X),
                                                    .iVGA_Y(Current_Y),
                                                    .iVGA_CLK(VGA_CLK),
                                                    .iRST_n(RESET),
                                                    .iColor_SW(SW[0]));
endmodule
```

Загалом отримаємо таку структурну схему проекту (рис. 6.3)

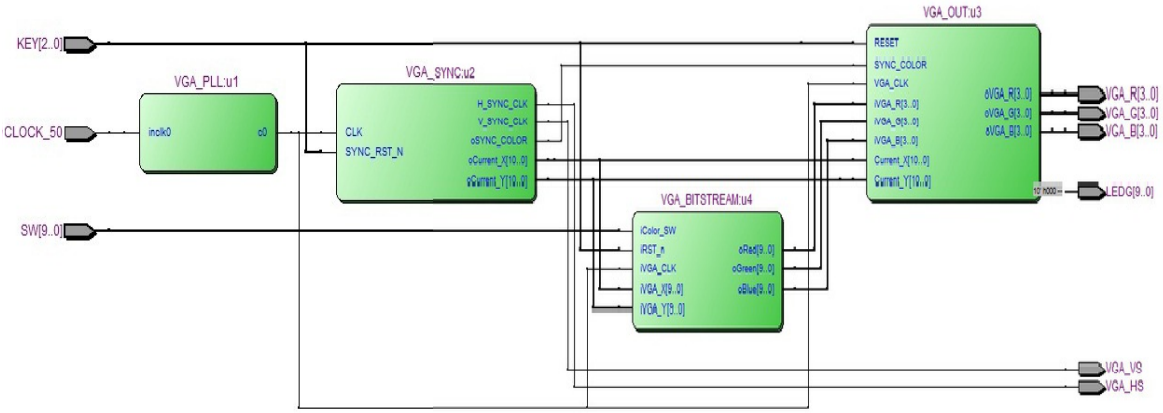


Рисунок 6.3 – Структурна схема проекту

Хід роботи

1. Створіть проект для виводу зображення на монітор.

1.1. Створіть описані вище модулі та PLL модуль, скомпілюйте весь проект, назначивши головний модуль файлом верхнього рівня.

1.2. Якщо є помилки, то виправте, якщо немає, то назначте виводи скориставшись файлом User_manual.pdf.

1.3. Скомпілюйте знову проект та збережіть.

2. Підключіть монітор до роз’єму VGA на навчальній платі.

3. Запрограмуйте ПЛІС та перевірте роботу (зображення на моніторі).

4. За завданням викладача змініть зображення на інше (наприклад, прапор якоїсь країни чи інше).

5. За завданням викладача або за власною ініціативою створіть проект з динамікою на моніторі (рухомі об’єкти, зміна кольору чи інше).

6. Зробіть висновки про здобуті знання та навички.

Перелік контрольних запитань

1. Поясніть принцип роботи синхронізатора для електронно-променевої трубки.

2. Поясніть процедуру створення об’єкта перетворення частоти типу PLL.

3. Опишіть виводи для підключення інтерфейсу VGA.

4. Яким чином формується колір окремого пікселя монітора?

СПИСОК ЛІТЕРАТУРИ

1. Цифрова схемотехніка. Частина 2. Електронні пристрої і системи: навчальний посібник / Й. Й. Білинський, П. М. Ратушний, А. О. Мельничук. – Вінниця : ВНТУ, 2016. – 171 с.

2. Зотов В. Ю. Проектирование цифрових устройств на основе ПЛИС фирмы XILINX в САПР WebPACK ISE. – М. : Горячая линия – Телеком, 2003. – 624 с.

3. Суворова Е. А. Проектирование цифрових систем на VHDL / Е. А. Суворова, Ю. Е. Шейнин – СПб. : БХВ – Петербург, 2003. – 576 с.

4. Стешенко В. Б. ПЛИС фирмы Altera: элементная база, система проектирования и языки описания аппаратуры / В. Б. Стешенко. – М. : Издательский дом «Додэка – XXI», 2007. – 576 с.

5. Поляков А. К. Языки VHDL и Verilog в проектировании цифровой аппаратуры / А. К. Поляков. – М. : Солон-Пресс, 2003. – 320 с.

6. Соловьев В. В. Проектирование цифровых систем на основе программируемых интегральных схем / Соловьев В. В. – М. : Горячая линия – Телеком, 2001. – 636 с.

7. Рябенький В. М. MAX + plus II. Основи проектування цифрових пристроїв на ПЛІС / В. М. Рябенький, Ушкаренко О. О. – К. : «Корнійчук», 2004. – 253 с.

8. Рябенький В. М. VERILOG. Практика проектування цифрових пристроїв на ПЛІС : навч. посібник / В. М. Рябенький, О. О. Ушкаренко. – Миколаїв : Іліон, 2007. – 324 с

9. Системы автоматизированного проектирования фирмы ALTERA MAX+plus II и 9. 9. Quartus II. Краткое описание и самоучитель / [Комолов Д. А., Мальк Р. А., Зобенко А. А., Филиппов А. С.]. – М. : ИП РАДИОСОФТ, 2002 – 352 с.

10. Кофанов В. Л., Осадчук О. В., Гаврілов Д. В. Проектування цифрових пристроїв на основі САПР Quartus II. Практикум / Кофанов В. Л., Осадчук О. В., Гаврілов Д. В. – Вінниця, ВНТУ, 2009. – 164 с.

11. Грушвицкий Р. И., Мурсаев А. Х., Угрюмов Е. П. Проектирование систем на микросхемах программируемой логики. – СПб. : БХВ-Петербург, 2002. – 608 с.

Навчальне видання

Ратушний Павло Миколайович
Жагловська Олена Миколаївна
Огородник Костянтин Володимирович

ПЛІС та їх програмування

Лабораторний практикум

Рукопис оформлено: О. Жагловська

Редактор: О. Ткачук

Оригінал-макет виготовлено: О. Ткачук

Підписано до друку 17.05.2018.
Формат 29,7×42¼. Папір офсетний.
Гарнітура Times New Roman.
Друк різографічний. Ум. друк. арк. 3,29.
Наклад 50 (1-й запуск 1–20) пр. Зам. № 2018-094.

Видавець та виготовлювач
Вінницький національний технічний університет,
інформаційний редакційно-видавничий центр.
ВНТУ, ГНК, к. 114. Хмельницьке шосе, 95,
м. Вінниця, 21021.
Тел. (0432) 65-18-06.
press.vntu.edu.ua;
email: kivc.vntu@gmail.com.
Свідоцтво суб'єкта видавничої справи
серія ДК № 3516 від 01.07.2009 р.