# ПРОЕКТУВАННЯ МЕДИЧНИХ КОМП'ЮТЕРНИХ СИСТЕМ НА ОСНОВІ МІКРОСХЕМ ПРОГРАМОВАНОЇ ЛОГІКИ

**ПЕРЕДМОВА** ..... **6**

**РОЗДІЛ 1. АРХІТЕКТУРА МІКРОСХЕМ ПРОГРАМОВАНОЇ ЛОГІКИ** ..... **9**

1.1. Класифікація ПЛІС ..... 10
1.2. Архітектура мікросхем сімейства Cyclone II ..... 15
1.2.1. Архітектура логічного елемента ..... 16
1.2.2. Логічний блок ..... 20
1.2.3. Конфігуруємий блок пам'яті M4K ..... 21
1.2.4. Архітектура блоку ФАПЧ ..... 22
1.2.5. Архітектура вбудованого помножувача і блоку цифрової обробки сигналу ..... 24
1.2.6. Елементи вводу-виводу і банки вводу-виводу ..... 26
1.3. Архітектура мікросхем Spartan ..... 30
1.4. Архітектура мікросхем CPLD ..... 32
1.5. Архітектура мікросхем MAX II ..... 34
1.6. Конфігурування мікросхем ПЛІС ..... 36
Література ..... 40

**РОЗДІЛ 2. МОВА ОПИСУ АПАРАТУРИ VHDL.** ..... **42**

2.1. Мови опису апаратури ..... 43
2.2. Рівні проектування мікросхем ..... 45
2.3. Структура проекту на мові VHDL. ..... 47
2.3.1. Декларація бібліотек ..... 47
2.3.2. Інтерфейс об'єкта проекту ..... 49
2.3.3. Архітектура об'єкта проекту ..... 51
2.3.4. Спрощений приклад програми на мові VHDL ..... 52
2.4. Типи даних. Літерали. ..... 54
2.4.1. Базові типи ..... 54
2.4.2. Типи, визначені користувачем ..... 58
2.4.3. Підтипи ..... 62
2.4.4. Атрибути типів. ..... 63
2.4.5. Літерали. ..... 64
2.4.6. Масиви і записи ..... 65
2.5. Константи, сигнали та змінні ..... 69
2.6. Оператори мови VHDL. ..... 71
2.6.1. Оператори присвоювання ..... 73
2.6.2. Послідовні оператори ..... 77
2.6.3. Паралельні оператори ..... 84
2.6.4. Оператор генерації ..... 93
2.6.5. Компонент ..... 95
2.7. Поради по написанню текстів на мові VHDL ..... 99

Література ..... 101

**РОЗДІЛ 3. РОЗРОБКА СИСТЕМ ЗА ДОПОМОГОЮ МОВИ VHDL 103**
3.1. Опис комбінаційних пристроїв ..... 104
3.2. Опис послідовних схем ..... 116
3.2.1. Опис тригерів ..... 116
3.2.2. Опис регістрів ..... 121
3.2.3. Опис лічильників ..... 125
3.3. Опис цифрових автоматів ..... 129
3.4. Опис пам'яті з використанням VHDL ..... 136
3.4.1. Опис постійних зап'ятовуючих прист-роїв на мові VHDL ..... 137
3.4.2. Опис оперативних запам'ятовуючих пристроїв ..... 145
3.4.3. Опис двохпортової пам'яті ..... 149
3.5. Пакети, процедури та функції ..... 155
3.5.1. Пакети ..... 155
3.5.2. Процедури і функції ..... 157
3.6. Створення файлу на мові опису апаратури в пакеті Quartus II ..... 160
Література ..... 164

**РОЗДІЛ 4. РОБОТА В ПАКЕТІ QUARTUS II ..... 165**
4.1. Знайомство з пакетом Quartus II ..... 166
4.1.1. Особливості роботи з пакетом Quartus II ..... 166
4.1.2. Цикл розробки проекту на ПЛІС ..... 167
4.1.3. Створення проекту ..... 170
4.1.4. Відкриття існуючого проекту ..... 175
4.1.5. Керування проектом ..... 177
4.2. Створення графічного файлу в пакеті Quartus II ..... 181
4.2.1. Створення нового файлу ..... 181
4.2.2. Створення елементів за допомогою майстра MegaWizard Plug-In Manager ..... 188
4.3. Компіляція проекту ..... 195
4.3.1. Звіт про компіляцію ..... 198
4.3.2. RTL Viewer та Technology Map Viewer ..... 199
4.3.3. State Machine Viewer ..... 200
4.3.4. Chip Planner ..... 201
4.4. Установки та призначення проекту ..... 202
4.5. Призначення виводів мікросхеми в Quartus II ..... 207
4.6. Часовий аналіз в Quartus II ..... 211
4.7. Побудова часових діаграм проекту ..... 215
4.7.1. Додавання діаграм у файл ..... 217
4.7.2. Встановлення параметрів сигналів ..... 221

4.8. Програмування та конфігурування в Quartus II.....225
Література.....229

**РОЗДІЛ 5. РОЗРОБКА СИСТЕМ В ПАКЕТІ QUARTUS II.....230**
5.1. Створення ієрархічного проекту.....231
5.1.1. Робота з каналами (conduit).....232
5.1.2. Робота з блоками.....233
5.1.3. Приєднання блоку до схеми.....238
5.1.3. Створення символу для файлу.....240
5.2. Стилі проектування.....241
5.3. Реалізація запам'ятовуючих пристроїв в ПЛІС.....242
5.3.1. Створення MIF-файлу.....243
5.3.2. Створення блоку пам'яті за допомогою MegaWizard Plug–In Manager.....247
5.4. Робота з цифровими автоматами.....251
5.4.1. Робота з State Machine Wizard.....251
5.4.2. Кодування станів цифрового автомата.....256
5.5. Оптимізація проектів в пакеті Quartus II.....259
5.5.1. Розподіл проекту на частини.....259
5.5.2. Аналіз повідомлень компілятора.....261
5.5.3. Помічники оптимізації.....263
5.5.4. Аналіз проекту.....266
5.5.5. Оптимізація використання ресурсів мікросхеми.....269
5.6. Засоби внутрішньосистемного налагоджування для ПЛІС Altera.....278
5.6.1. Налагоджувальні виводи (SignalProbe).....280
5.6.2. Інтерфейс для зовнішнього логічного аналізатора.....283
5.6.3. Редактор вмісту вбудованої пам'яті.....290
5.6.4. Вбудований логічний аналізатор Signal Tap II.....293
Література.....311

# ПЕРЕДМОВА

Розробка вбудованих комп'ютерних систем нерозривно пов'язана з розвитком елементної бази, на якій виконується реалізація програми. Від ефективності використання апаратних засобів залежить швидкість реалізації алгоритмів та їх ефективність. Традиційно, з цією метою використовуються спеціалізовані мікросхеми, які можна розподілити на декілька груп в залежності від того, наскільки архітектура мікросхеми залежить від її області використання.

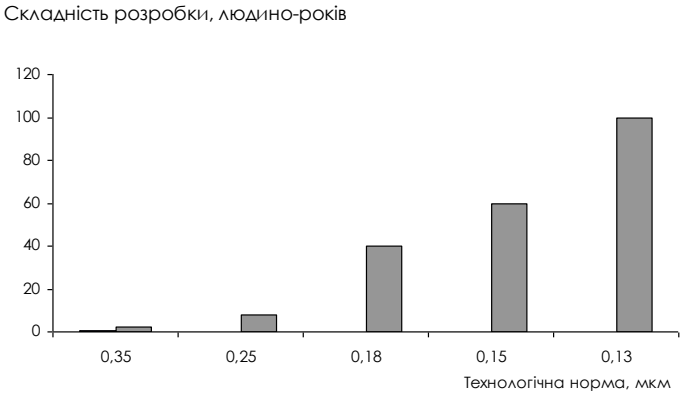
Найбільш вузькоспеціалізованими є так звані мікросхеми спеціального призначення (ASIC - application-specific integrated circuit). Такі мікросхеми розробляються для використання у конкретному середовищі і для вирішення заздалегідь визначених задач. Це дозволяє таким мікросхемам найбільш ефективно вирішувати задачі, для яких розроблялась ця мікросхема. Прикладами мікросхем ASIC можуть бути мікросхеми чіпсету материнської плати персонального комп'ютера (chipset), мікросхеми керування мобільними телефонами, кодування та декодування відеоданих.

З розвитком технології зростає і складність та вартість розробки мікросхем спеціального призначення в новому технологічному базисі. Наприклад, вартість розробки мікросхеми ASIC при технологічній нормі 180 нм складала 7-8 млн. дол., при технологічній нормі 90 нм – біля 40 млн. дол., а при технологічній нормі 65 нм – 70-80 млн. дол. При цьому зараз розробка мікросхем ASIC окупається при виробництві не менше ніж 100 000 шт., а п'ять років тому – лише 10 000 шт.

Крім зменшення технологічної норми при виробництві мікросхем ускладнюється і трудомісткість процесу проектування Середня швидкість роботи розробника апаратури складає біля 100 вентилів або 30 строк RTL коду в день. Для мікросхеми в 100 000 вентилів це становить 1000 людино-днів. Тобто 5 розробників проектують одну мікросхему за рік. А для мікросхеми в 10 мільйонів транзисторів і одного року розробки необхідно вже 500 розробників.

Як видно, використання мікросхем спеціального призначення може бути виправданим при великих партіях випуску та для великих компаній, що займаються розробкою таких систем. В той же час для об'ємів випуску, дослідних зразків чи тестових партій комп'ютерних

систем та пристроїв, що можуть змінювати свою конфігурацію під час роботи, краще використовувати програмовані логічні інтегральні мікросхеми – ПЛІС




На ринку мікросхем програмованої логіки у 2020 близько 90% займали дві компанії –Altera та Xilinx. На частку компанії Xilinx припадало 47%, а компанії альтана – 41% ринку ПЛІС.

При порівняні з ASIC мікросеми ПЛІС дозволяють значно зменшити час розробки пристрою та вартість невеликої серії мікросхем, однак при випуску більше ніж кілька тисяч пристроїв використання ПЛІС не є економічно вигідним. Ще одним недоліком ПЛІС у порівнянні з ASIC є їх більше споживання енергії при тих самих функціях.

До переваг ПЛІС слід віднести:

- можливість розробки спеціалізованої системи для обробки будь-яких даних з пропускною спроможністю до одиниць Гбіт/сек;
- можливість перепрограмування системи на платі, а значить і зміни структури апаратного модуля, що дозволяє використовувати один і той самий пристрій для виконання різних операцій;
- можливість швидкого прототипування пристроїв та відлагодження проекту на платі за допомогою вбудованого логічного аналізатора;

- побудова системи на кристалі – модуля в якомі на мікросхемі ПЛІС розміщується мікропроцесор зі змінюємою периферією, контролери різноманітних інтерфейсів (PCI, PCI-Express, USB, SPI, I2C, CAN, Ethernet), контролерів зовнішньої пам'яті (Flash, SDRAM, DDR, DDR2/3), вбудованих блоків пам'яті, спеціалізовані обчислювальні модулі (криптографічні модулі, блоки швидкого обчилювання Фур'є, цифрові фільтри та інше);
- побудова на мікросхемі ПЛІС мережі на кристалі – групи обчилювальних модулів, що з'єднані між собою локальною обчислювальною мережею.

Перший розділ присвячений розгляду принципів організації мікросхем ПЛІС FPGA та CPLD типів. Розглянуто архітектуру логічного елемента, логічного блоку, блоків фазового автопідстроювання частоти й убудованих блоків цифрової обробки сигналу. Також у першому розділі розглянуті принципи програмування й конфігурування мікросхем ПЛИС.

Другий і третій розділи присвячені розгляду основ мови опису апаратури VHDL і принципам опису різних цифрових пристроїв мовою VHDL. Наведено приклади опису різних комбінаційних і послідовністних пристроїв, модулів пам'яті, цифрових автоматів. Також показана реалізація бібліотек, процедур, функцій і компонентів, що дозволяє реалізовувати більші проекти мовою VHDL.

В четвертому та п'ятому розділах розглянута робота в пакеті Quartus II починаючи від створення файлів та проектів і закінчуючи програмуванням мікросхеми ПЛІС. Також розглянуті приклади розробки модулів пам'яті і цифрових автоматів за допомогою засобів пакета Quartus II, методика оптимізації проекту на ПЛІС і внутрісистемне налагодження.

# Розділ 1

# Архітектура мікросхем програмованої логіки


- Класифікація ПЛІС. ..... 10
- Архітектура мікросхем сімейства Cyclone II..... 15
- Архітектура мікросхем Spartan ..... 30
- Архітектура мікросхем CPLD..... 32
- Архітектура мікросхем MAX II ..... 34
- Конфігурування мікросхем ПЛІС..... 36

## 1.1. Класифікація ПЛІС

Найбільш просту побудову серед мікросхем програмованої логіки мали програмовані логічні матриці – ПЛМ. Ці мікросхеми складалися з програмованих матриць «І» та «АБО». Як приклад таких мікросхем можна навести мікросхеми РТ1, РТ2, РТ21 серії 556. На рисунку 1.1 зображена внутрішня структура мікросхеми ПЛМ [5].

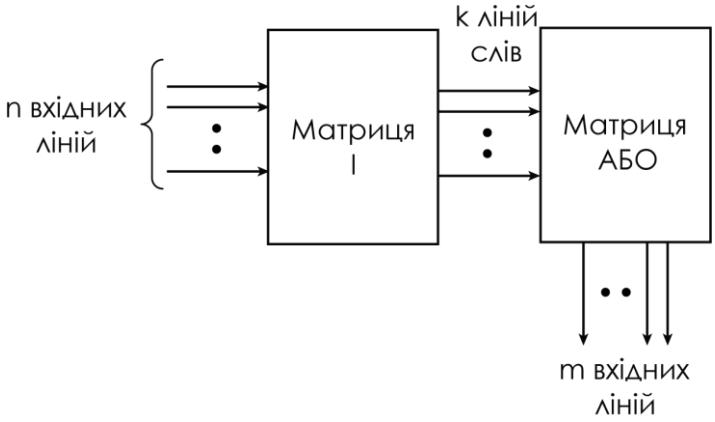



*Рисунок 1.1 – Спрощена архітектура ПЛМ*

Вхідний сигнал розрядністю *n* потрапляє до першої матриці, де над вхідними даними виконується логічна функція «І». Результати *k* кон'юнкцій входять до матриці «АБО», де над ними виконується операція логічного додавання (кон'юнкції). Таким чином у ПЛМ можна побудувати *m* функцій виду:

$$y = x_1 x_2 x_3 x_4 + x_1 x_2 x_3 x_4$$

В подальшому архітектура мікросхем програмованої логіки ускладнювалась. Змінювалась архітектура окремої чарунки, їх кількість та схема зв'язків, додавались нові функції та модулі. Перш ніж перейти до розгляду архітектури сучасних мікросхем спочатку розглянемо основні архітектурні особливості ПЛІС.

Схематично мікросхема та шляхи проходження сигналів показані на рисунку 1.2. Мікросхема містить елементи вводу-виводу, логічні елементи та лінії зв'язку – рядки і стовбці. Спрощений механізм роботи пристрою на ПЛІС виглядає таким чином: сигнал через елементи вводу-виводу потрапляє в мікросхему, де в логічному елементі над ним виконуються необхідні операції. Потім по лініях зв'язку через елементи вводу-виводу сигнал виходить з мікросхеми.

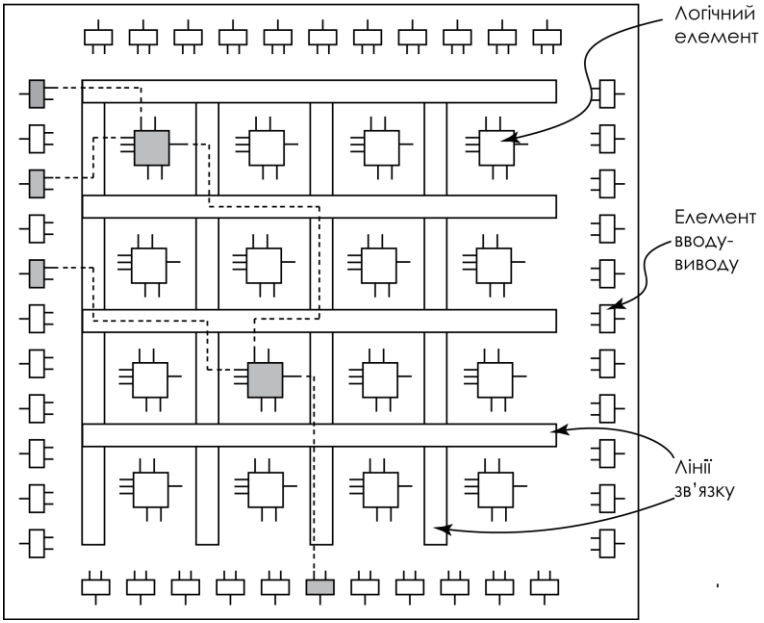



*Рисунок 1.2 – Спрощена архітектура ПЛІС*

Програмування та перепрограмування впливає на режими роботи логічних елементів, елементів вводу-виводу, зв'язки елементів в мікросхемі – як і в якому порядку з'єднуються логічні елементи, як розташовуються функції в логічних елементах. Для збереження всіх цих параметрів в мікросхемі ПЛІС існує конфігураційна пам'ять, доступ до якої користувач не може отримати, і програмування якої виконується спеціалізованим програмним забезпеченням.

Архітектура та можливості сучасних ПЛІС дуже різноманітні, але у більшості випадків можна навести декілька класифікаційних ознак:

ступінь інтеграції або логічна ємність;

архітектура логічного елемента мікросхеми;

тип запам'ятовуючого елемента для пам'яті конфігурації;

наявність вбудованих модулів (внутрішньої пам'яті, помножувачів, блоків цифрової обробки сигналів, інтерфейсів для різноманітних протоколів та інших).

*Ступінь інтеграції або логічна ємність* – найбільш важливий показник мікросхем ПЛІС. Сучасні мікросхеми включають в себе сотні мільйонів транзисторів, але мікросхема ПЛІС має дуже велику надлишковість структури, тому для більш коректного визначення об'єму мікросхем виробники використовували такий параметр як кількість еквівалентних логічних вентилів типу 2І-НІ або 2АБО-НІ, які б використовувались для реалізації пристроїв тієї ж самої складності, що і на ПЛІС. Такий спосіб розрахунку об'єму ПЛІС використовувався приблизно до 2004 року. Мікросхеми, які випускались на той час, мали обсяг від 10 тисяч еквівалентних логічних елементів (для мікросхеми Flex 10K10 фірми ALTERA) до 5,3 мільйонів еквівалентних логічних елементів (для мікросхеми APEX II фірми ALTERA). Слід відмітити, що кожна з фірм, що випускає мікросхеми програмованої логіки, розраховує кількість еквівалентних вентилів за власною методикою, що ускладнює порівняння мікросхем різних типів. Крім того, в структуру мікросхеми крім власне логічних елементів входить ще дуже багато різноманітних вузлів (елементи вводу-виводу інформації, блоки пам'яті, ресурси трасування сигналів і т.д.), кожен з яких описується різною кількістю

еквівалентних логічних елементів. Це також ускладнює розрахунок у еквівалентних вентилях.

Тому сьогодні для вимірювання ємності мікросхеми використовують такий параметр як кількість логічних елементів. Але, оскільки архітектура логічних елементів для різних мікросхем навіть одного виробника відрізняється від іншої, то порівняння мікросхем стає ще більш ускладненим. Для порівняння ПЛІС різних виробників можна порекомендувати статтю [1].

З іншого боку мікросхеми основних виробників ПЛІС можуть бути поділеними на такі умовні класи як дешеві мікросхеми (Lowest Cost), мікросхеми середнього класу (middle) та hi-end мікросхеми. Так серед продукції компанії Altera до мікросхем класу Lowest Cost відносяться мікросхеми сімейств Cyclone. Для мікросхем компанії Xilinx це сімейства Spartan [3, 7, 13, 16]. Вартість найменшої з таких мікросхем складає близько 12 долларів. До мікросхем середнього класу компанія Altera відносить мікросхеми сімейства Arria, що орієнтовані на реалізацію швидкісних інтерфейсів вводу-виводу. Вартість найменшої з цих мікросхем складає близько 170 долларів. Мікросхеми класу hi-end є найкращим досягненням виробника і до таких мікросхем відносяться мікросхеми сімейств Statix компанії Altera та Virtex компанії Xilinx [3, 8, 17, 18]. Вартість таких мікросхем може сягати кількох тисяч доларів.

За *архітектурою логічного елементу* (Logic Element – LE) найбільш часто мікросхеми розділяють на два типи [2, 4]:

– CPLD – Complex Programmable Logic Devices – пристрій зі складною програмованою логікою;

– FPGA – Field Programmable Gate Array – матриця програмуємих логічних елементів.

*Мікросхеми з архітектурою CPLD* реалізовують в логічному елементі складну логічну функцію у вигляді логічного рівняння, яке формується за допомогою функцій «І» та «АБО». Це більш ускладнений варіант ПЛМ мікросхем. Конфігураційні дані в мікросхемах цього виду зберігається у вбудуваній пам’яті типу Flash, тому для своєї роботи мікросхеми не потребує зовнішнього завантажувача конфігурації. Найбільш характерними представниками такого виду ПЛІС можуть слугувати мікросхеми сімейств MAX7000 та MAX3000 фірми ALTERA або CoolRunner та XC9500 фірми

XILINX [6, 9, 10, 12, 14]. Більш детально архітектура мікросхеми MAX3000 буде розглянута у параграфі 1.3.

Основна ідея *мікросхем FPGA* полягає в тому, що будь-яка функція може бути описана таблицею дійсності. Тому мікросхеми цього типу реалізують логічну функцію на основі таблиці перекодування (look-up table), яка за принципом роботи подібна до постійного запам'ятовуючого пристрою. Класичним прикладом такого типу мікросхем можна назвати сімейство FLEX 10K фірми Altera, яке свого часу було дуже поширене і описане в багатьох книгах та підручниках [9, 11]. Ми ж розглянемо архітектуру FPGA мікросхем на прикладі мікросхем Cyclone II фірми ALTERA в параграфі 1.1. До FPGA мікросхем також відносяться ПЛІС сімейств Stratix фірми Altera та Spartan і Virtex компанії Xilinx [3, 10].

*Тип запам'ятовуючого елемента пам'яті конфігурації* визначає можливості ПЛІС по програмуванню, перепрограмуванню і збереженню інформації при відключенні живлення.

Сучасні мікросхеми використовують EEPROM, Flash або SRAM технологією для виготовлення запам'ятовуючого елементу, що дає можливість перепрограмовувати мікросхему.

Мікросхеми, що використовують технологію EEPROM або Flash забезпечують енергонезалежне збереження конфігурації, а також і багаторазове програмування мікросхеми. Мікросхема на SRAM повинна кожен раз програмуватися при ввімкненні живлення. Конфігураційні дані в цьому випадку зберігається у зовнішньому ПЗП або у персональному комп'ютері.

*Наявність вбудованих модулів.* Окрім стандартних логічних елементів ПЛІС може містити також апаратні блоки, які виконують спеціалізовані функції. Найбільш часто в мікросхемах розміщують блоки пам'яті, помножувачі, блоки фазового автопідстроювання частоти. Кількість та наявність таких блоків визначається складністю та призначенням мікросхеми. Наявність таких блоків з одного боку ускладнює мікросхему, підвищуючи її вартість. З іншого боку апаратні ядра значно потужніші за швидкодією у порівнянні з такими самими пристроями, створеними на логічних елементах [4].

## 1.2. Архітектура мікросхем сімейства Cyclone II

Мікросхеми Cyclone відносяться до так званих «дешевих» мікросхем ПЛІС. Перші мікросхеми сімейства Cyclone з'явились у 2002 році. З того часу було випущено кілька сімейств і на сьогоднішній день фірма ALTERA випускає мікросхеми сімейства Cyclone V. У даному розділі ми розглянемо архітектуру сімейства Cyclone II [13].

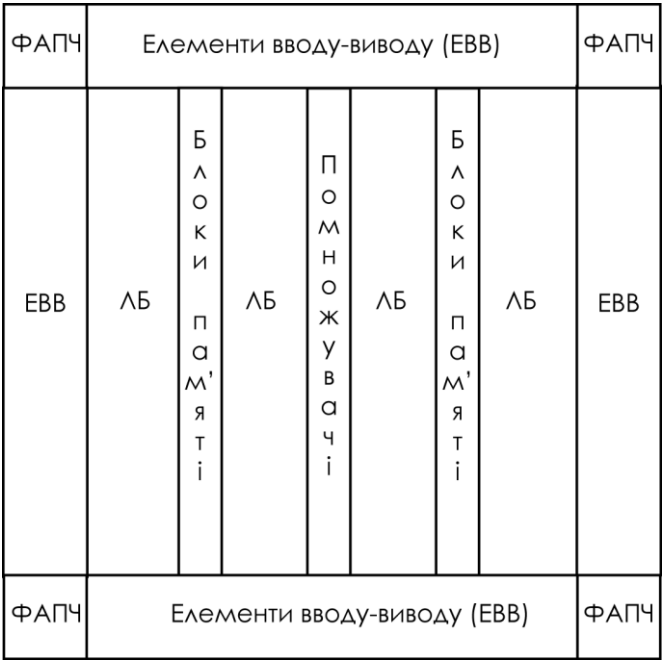



*Рисунок 1.3 – Загальна архітектура мікросхеми Cyclone II*

Загальна архітектура мікросхеми Cyclone II показана на рисунку 1.3. В структурі мікросхеми показані лише ті ресурси, які розробник може використовувати в своїх проектах і які доступні йому для програмування. Основний елемент мікросхеми це логічний блок (ЛБ), який включає в себе елементарні чарунки мікросхеми – логічні елементи (ЛЕ). В масиві логічних блоків розташовані стовпчики блоків пам'яті та помножувачів. По периметру мікросхеми розміщуються елементи вводу-виводу (ЕВВ), скрізь які виконується обмін мікросхеми інформацією з навколишнім світом. По кутах кристалу розміщені блоки фазового автопідстроювання частоти (ФАПЧ). Крім показаного на рисунку 1.3 на кристалі розміщується ще вузол конфігурування та конфігураційна пам'ять, які недосяжні користувачу. Перелік ресурсів найменшої та найбільшої мікросхем сімейства Cyclone II наведені в таблиці 1.1.

*Таблиця 1.1 – Параметри мікросхем сімейства Cyclone II*

| Параметр | EP2C5 | EP2C70 |
|---|---|---|
| Кількість логічних елементів | 4 608 | 68 416 |
| Кількість блоків пам'яті M4K | 26 | 250 |
| Загальний об'єм пам'яті, біт | 119 808 | 1 152 000 |
| Блоків ФАПЧ | 2 | 4 |
| Вбудованих помножувачів | 13 | 150 |
| Виводів, доступних користувачу | 158 | 622 |

## 1.2.1. Архітектура логічного елемента

ПЛІС Cyclone II – це FPGA мікросхема і основою логічного блоку в ній є таблиця перекодування (Look-Up Table – LUT). Архітектура логічного блоку показана на рисунку 1.4.

Таблиця перекодування може генерувати будь-яку логічну від чотирьох змінних. Крім таблиці перекодування логічний елемент містить також конфігуруємий тригер (Programmable Register) та схему переносу.

Логічний елемент може працювати у двох режимах роботи:

- нормальному режимі,
- арифметичному режимі.

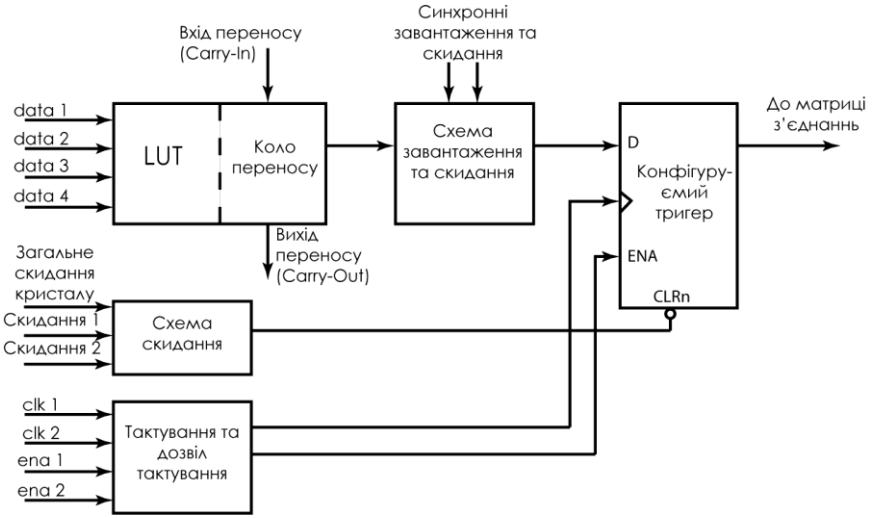



*Рисунок 1.4 – Загальна архітектура логічного елемента FPGA мікросхеми*

В нормальному режимі роботи (рисунок 1.5) на входи таблиці перекодування надходять чотири сигнали даних (data 1 .. data 4), а з виходу дані можуть поступати або на вхід конфігуруємого тригера, або через матрицю з'єднань на інші елементи мікросхеми. Крім вхідних даних в логічний елемент з логічного блоку надходять також лінії двох локальних сигналів синхронізації (clk 1, clk 2), двох сигналів дозволу роботи (ena 1, ena 2), двох сигналів скидання та сигналу загального скидання кристалу, а також сигнали синхронного завантаження та скидання. Конфігуруємий тригер, в залежності від режиму роботи, може працювати як D, T, JK або RS тригер. Вихідний сигнал тригера надходить в матрицю з'єднань і може подаватися через лінії зв'язку на будь-який елемент мікросхеми.

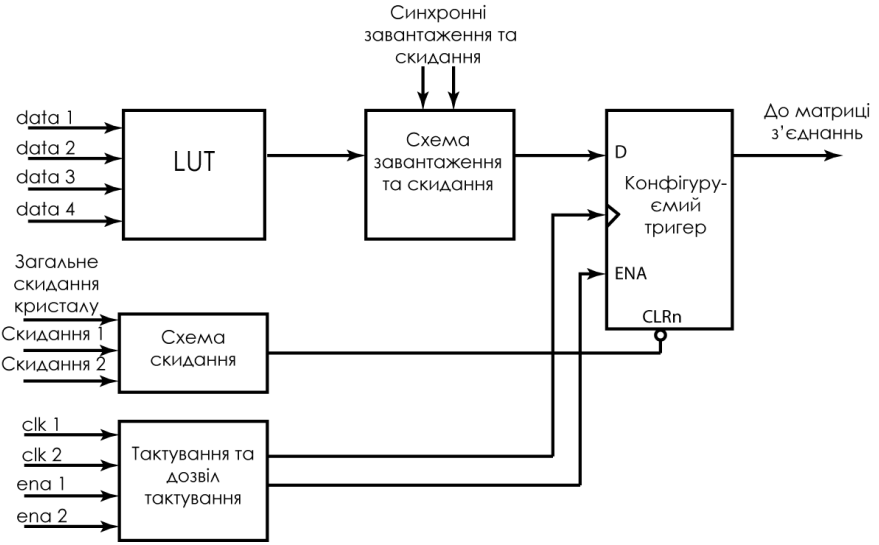



*Рисунок 1.5 – Логічний елемент FPGA мікросхеми в нормальному режимі роботи*

Перед вивченням арифметичного режиму роботи пригадаємо як виконується операція додавання у двійковому коді.

Для додавання двох багаторозрядних чисел над кожним розрядом (крім молодшого) необхідно виконати такі операції:

$$S_i = A_i \oplus B_i \oplus P_{i-1}$$

$$P_i = A_i B_i + P_{i-1}(A_i \oplus B_i)$$

де $S_i$ – сума і-го розряду,

$P_i$ – вихід переносу з і-го розряду,

$P_{i-1}$ – вхід переносу з і-1-го розряду,

$A_i$, $B_i$ – і-й розряд доданка А та В.

З наведених вище формул видно, що для виконання операції додавання необхідно використати три змінні: A, B, P, які повинні надійти до логічного елемента мікросхеми. Крім того, необхідно організувати перенос з молодшого розряду і до старшого розряду. Таким чином, необхідна інша архітектура логічного елемента, яка показана на рисунку 1.6. Така архітектура використовує таблицю перекодування, що поділена на дві трьохвходові таблиці перекодування. Одна таблиця використовується для формування сигналу суми, а друга – для формування сигналу переносу (Carry-Out). В якості сигналів даних використовуються два входи даних (data1, data2) та вхід переносу (Carry-In).

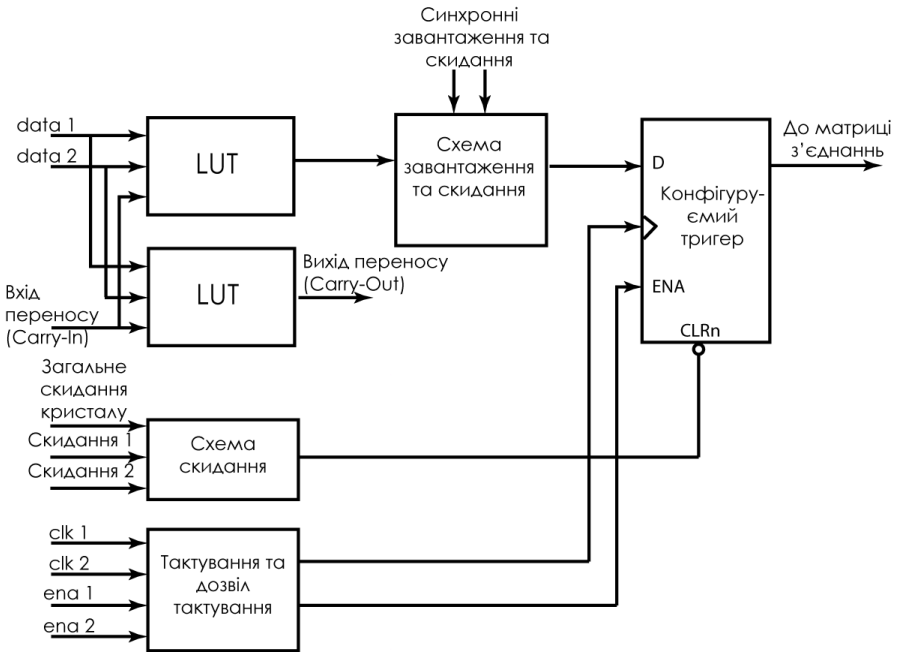



*Рисунок 1.6 – Логічний елемент FPGA мікросхеми в арифметичному режимі роботи*

Для складання багаторозрядного числа логічні елементи повинні бути об'єднані у загальну структуру, структуру якої показано на рисунку 1.7. З рисунку видно, що окремі розряди доданків надходять до сусідніх ЛЕ, які формують біти результату. Якщо розрядність доданків перевищує кількість логічних елементів у логічному блоці (для мікросхем сімейства Cyclone II – це 16), то формується сигнал переносу з одного логічного блоку до іншого (LAB Carry-Out).

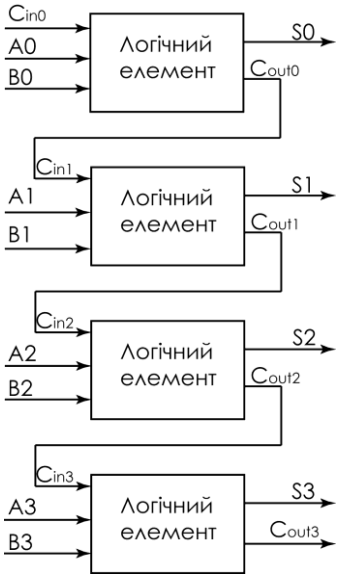



*Рисунок 1.7 – Об'єднання логічних елементів для додавання 4-розрядних чисел*

## 1.2.2. Логічний блок

Логічний блок (Logic Array Block – LAB) мікросхеми містить у собі шістнадцять логічних елементів, які з'єднані локальною матрицею з'єднань (Local interconnect), ланцюги сигналів керування логічним блоком, ланцюг переносу для логічних елементів блоку та ланцюг переносу з одного логічного блоку до іншого. На рисунку 1.8

показано два логічних блоки та з'єднання їх з глобальною матрицею з'єднань.

Логічні блоки розташовані рядками та стовбчиками і з'єднуються між собою за допомогою глобальної матриці з'єднань (MultiTrack interconnect).

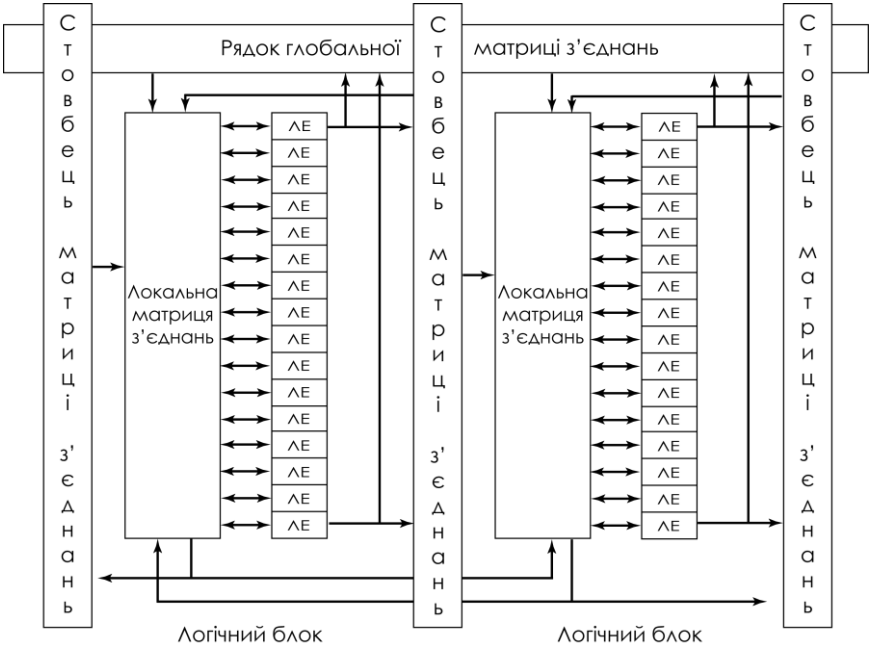



*Рисунок 1.8 – Логічні блоки мікросхеми Cyclone II*

## 1.2.3. Конфігуруємий блок пам'яті M4K

Мікросхеми сімейства Cyclone II містять в собі блоки пам'яті типу M4K, об'єм яких складає 4 098 біт. Загальний об'єм вбудованої пам'яті мікросхеми становить віт 104 кбіт до 1 Мбіта, а максимальна тактова частота роботи сягає 250 МГц.

Блок пам'яті M4K може конфігуруватись в наступні види пам'яті:

- оперативна пам’ять;
- постійний запам’ятовуючий пристрій;
- двохпортовий ОЗП;
- регістр зсуву;
- блок FIFO.

Крім того можлива робота з бітом парності.

Основною відмінністю вбудованих блоків пам’яті ПЛІС є їх конфігуруємість, тобто блок пам’яті може мати кілька різних варіантів організації, об’єм яких не перевищує об’єм самого блоку пам’яті. При роботі блок M4K допускає наступні конфігурації:

| | |
|---|---|
| 4Kx1; | 256x16; |
| 2Kx2; | 256x18; |
| 1Kx4; | 128x32; |
| 512x8; | 128x36. |
| 512x9; | |

Якщо ж необхідно використовувати модуль вбудованої пам’яті більшого об’єму, то кілька блоків об’єднуються в один модуль. В більш старих мікросхемах ПЛІС, наприклад Flex10K, блок пам’яті вважався використаним, якщо в ньому був зайнятий хоча б один біт. Невикористані біти пам’яті розробник вже не міг використовувати в своєму проекті. У мікросхемах сімейства Cyclone є можливість використовувати один модуль M4K для двох однопортових блоків пам’яті, що значно підвищує використання блоків пам’яті.

## 1.2.4. Архітектура блоку ФАПЧ

Блоки фазового автопідстроювання частоти (ФАПЧ, Phase-locked loop, PLL) можуть використовуватись для множення та ділення частоти, зсув фази тактового сигналу відносно основної частоти та отримання програмованої шпаруватості тактового сигналу.

Розглянемо основний принцип роботи системи ФАПЧ, структурна схема якої зображена на рисунку 1.9.

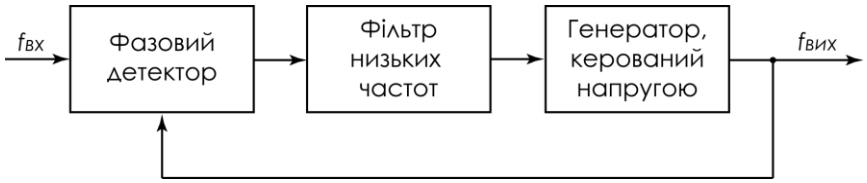



*Рисунок 1.9 – Структурна схема ФАПЧ*

До складу системи входить фазовий детектор, який визначає різницю в фазі та частоті вхідного сигналу (*fвх*) та сигналу зворотного зв'язку. Вихідний сигнал формується генератором, що керується напругою. Фільтр низьких частот необхідний для запобігання самозбудженню системи вцілому.

В результаті роботи такої системи в ідеальному випадку можна отримати на виході сигнал, що співпадає з вхідним сигналом.

При подальшому ускладненні системи (рисунок 1.10) до неї додаються три лічильники – дільники частоти: вхідний (коефіцієнт ділення M), вихідний (коефіцієнт ділення K) та лічильник у ланці зворотного зв'язку (коефіцієнт ділення N).

Якщо увімкнути у ланку зворотного зв'язку дільник зі змінюємим коефіцієнтом рахування N, то частота на виході цього дільника зменшиться в N раз у порівнянні з сигналом fд. Система фазового автопідстроювання частоти буде підтримувати значення частот на вході фазового детектора рівними один одному. А це означає, що частота на виході генератора, що керується напругою, буде збільшуватись в N разів у порівнянні з вхідною частотою fд. Змінюючи коефіцієнт рахування дільника зворотного зв'язку можна отримувати різні значення частот генератора.

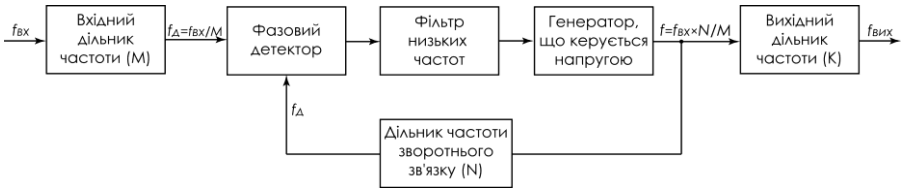



*Рисунок 1.10 – Структурна схема ФАПЧ в ПЛІС Cyclone II*

Додавання до схеми вхідного дільника частоти з постійним коефіцієнтом рахування М дозволяє отримати низьку частоту для порівняння на входах фазового детектора. На виході генератора, що керується напругою, додається ще один дільник, але вже зі змінним коефіцієнтом рахування К. В результаті вихідна частота буде визначатися формулою:

$$f_{вих} = \frac{f_{вх}}{M} N / K$$

## 1.2.5. Архітектура вбудованого помножувача і блоку цифрової обробки сигналу

Для підвищення швидкодії виконання операцій цифрової обробки сигналів у сучасних ПЛІС вбудовують або помножувачі (Embedded multiplier) або блоки цифрової обробки сигналу (DSP block). Структура вбудованого помножувача наведена на рисунку 1.11. Вбудований помножувач може працювати в одному з двох режимів: один 18-бітний помножувач або два незалежних 9-бітних помножувача. Максимальна тактова частота роботи помножувача досягає 250 МГц. Кількість вбудованих 18-бітних помножувачів в мікросхемах сімейства Cyclone II може доходити до 150.

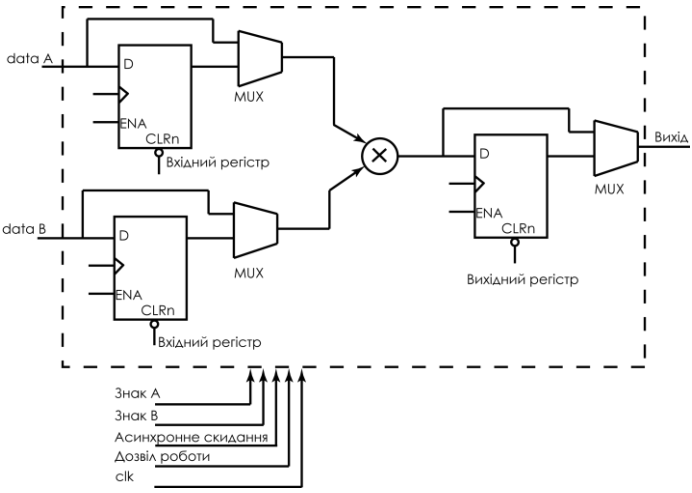



*Рисунок 1.11 – Стуктурна схема помножувача у ПЛІС Cyclone II*

Як видно з рисунку, помножувач містить регістри по входу та виходу, що дає можливість створювати синхронні схеми. За бажанням ці регістри можуть і не використовуватись в схемі. Для цього в схемі використовуються мультиплексори (MUX), які формують сигнали на вході та виході помножувача.

В мікросхемах сімейств Stratix замість вбудованих помножувачів використовуються більш складні блоки цифрової обробки сигналу (ЦОС). Основним призначенням блоку цифрової обробки сигналу є оптимізація алгоритмів ЦОС, які потребують високої пропускної спроможності. До таких алгоритмів відносяться цифрова фільтрація, швидке перетворення Фур'є, кодування даних. Типовими задачами, які потребують виконання таких операцій є цифрове телебачення, IP-телефонія та телекомунікації. Для всіх цих задач використання вбудованих блоків ЦОС дозволяє значно підвищити пропускну спроможність алгоритму та зменшити ресурси, які необхідні для реалізації алгоритму.

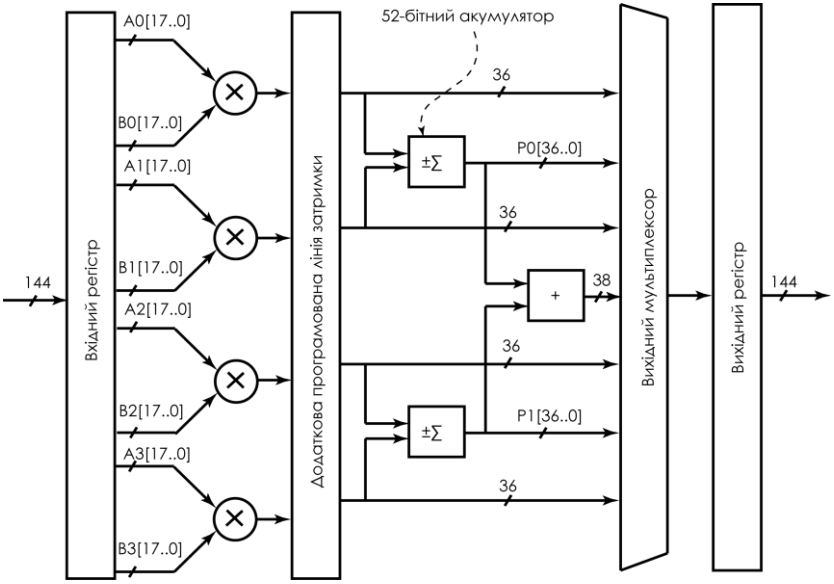



*Рисунок 1.12– Структурна схема половини блоку ЦОС*

Блок ЦОС складається з двох однакових частин, одна з яких показана на рисунку 1.12 [17]. Блок містить помножувачі з програмуємою розрядністю, суматори та регістри для збереження проміжних результатів розрахунків. Помножувачі можуть працювати у декількох режимах – 9, 12, 18 та 36-розрядному.

При використанні такого блоку при роботі помножувачів у 18-розрядному режимі на виході кожного суматора $\sum$ можна отримати наступний вираз для операції MAC (multiply-accumulate):

$$P0[36..0] = A0[17..0] \times B0[17..0] \pm A1[17..0] \times B1[17..0].$$

Крім помножувачів та суматорів блоки ЦОС містять ланцюги для переносу результатів з одного блоку ЦОС до іншого, що значно прискорює обчислення у основних операціях ЦОС.

### 1.2.6. Елементи вводу-виводу і банки вводу-виводу

Елемент вводу-виводу мікросхем сімейства Cyclone II може працювати в одному з режимів, який задається при програмуванні мікросхеми: вхід, вихід, двонаправлений вивід, вивід у високоімпедансному стані. Крім того елемент вводу-виводу забезпечує також підтримку різноманітних стандартів, увімкнення підтягуючих резисторів (pull-up) під час конфігурації мікросхеми, роботу ланцюгів утримання шини (bus-hold), програмуєму затримку сигналу для входу та виходу, підтримку необхідної сили струму для деяких стандартів, програмуєме увімкнення підтягуючих резисторів.

Елемент вводу-виводу містить вихідний буфер, вхідний, вихідний регістри та регістр дозволу роботи, який переводить вихід мікросхеми в Z-стан, які забезпечують двонаправлений режим роботи, введення даних в ПЛІС та виведення даних з мікросхеми. Архітектура елементів вводу-виводу показана на рисунку 1.13.

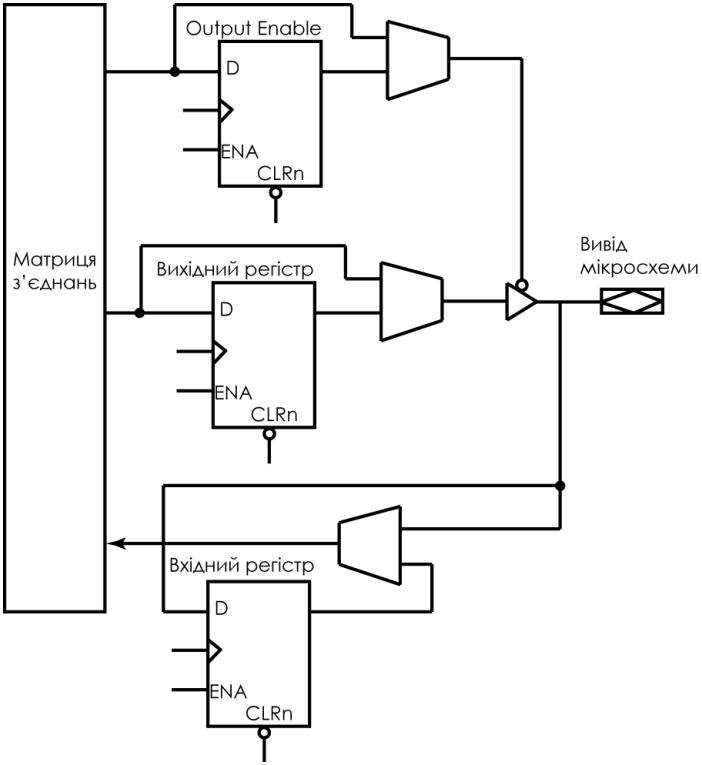



*Рисунок 1.13 – Архітектура блоку вводу-виводу*

Сучасні ПЛІС можуть працювати одночасно з кількома стандартами вводу-виводу одночасно (рисунок 1.14). Це дає можливість використовувати ПЛІС в системах з різними стандартами вводу-виводу. Так, наприклад, до мікросхеми ПЛІС можуть бути підключені сигнали з напругою живлення 2,5 В, 3,3 В, диференційні сигнали, сигнали різноманітних протоколів: PCI, PCI-X, DDR. Але слід звернути увагу на те, що ПЛІС не можуть забезпечити повноцінну роботу з стандартними ТТЛ-рівнями. В більшості випадків мікросхеми працюють з сигналами до 3,3 В. Для роботи з сигналами ТТЛ та 5В КМОН необхідно використовувати перетворювачі рівня.

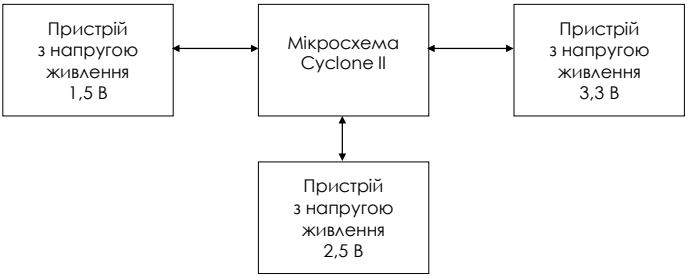



*Рисунок 1.14 – Робота мікросхеми з різними стандартами вводу-виводу*

Забезпечення підтримки декількох стандартів вводу-виводу досягається завдяки архітектурі ПЛІС та роздільним лініям живлення для різних банків (рисунок 1.15).

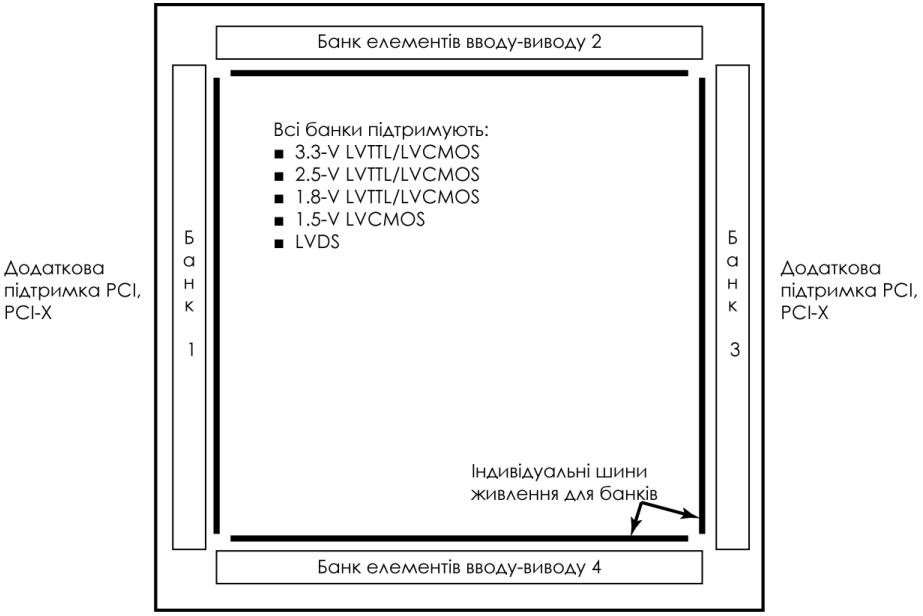



*Рисунок 1.15 – Банки вводу-виводу в мікросхемі ПЛІС*

Елементи вводу-виводу об'єднуються в групи, які називаються банками. Особливістю банків є використання окремих ліній живлення для кожного з банків. Для живлення банків елементів вводу-виводу може бути використано декілька рівнів напруг: 1,5 В; 1,8 В; 2,5 В; 3,3В. Тобто окрема лінія не може бути сконфігурована на інший стандарт, а лише весь банк вцілому, який включає кілька десятків виводів. Рівень напруги визначається стандартами вводу-виводу, які будуть використані в елементах вводу-виводу. В мікросхемах сімейства Cyclone II може бути 4 або 8 банків. Кількість банків залежить від об'єму мікросхеми та кількості виводів у корпусі (таблиця 1.2).

Використання різних типів корпусів та мікросхем різної логічної ємності дає можливість так званої вертикальної міграції, коли один і той самий корпус може використовуватись мікросхемами різної логічної ємності. При цьому розташування службових виводів та виводів живлення у мікросхем різної ємності буде однаковим. В цьому випадку можна використовувати ту ж саму друковану плату, замінивши лише мікросхему на більшу або меншу.

*Таблиця 1.2 – Кількість ліній, доступних користувачу в залежності від типу корпусу мікросхеми*

| Тип мікросхеми | Тип корпусу | | | | | |
|---|---|---|---|---|---|---|
| | TQFP, 100 виводів | TQFP, 144 виводи | PQFP, 240 виводів | BGA, 256 виводів | BGA, 324 виводи | BGA, 400 виводів |
| EP1C3 | 65 | 104 | – | – | – | – |
| EP1C4 | – | – | – | – | 249 | 301 |
| EP1C6 | – | 98 | 185 | 185 | – | – |
| EP1C12 | – | – | 173 | 185 | 249 | – |
| EP1C20 | – | – | – | – | 233 | 301 |

Крім ліній програмування та живлення мікросхеми ПЛІС мають також виводи для підключення тактових сигналів. Сигнали з цих виводів підключаються до *ліній глобального розповсюдження сигналу*. Кількість спеціалізованих тактових сигналів залежить від типу корпусу мікросхеми і може дорівнювати 8 або 20. Лінії глобального розповсюдження сигналів дозволяють прискорити проходження

сигналів по мікросхемі у порівнянні з проходженням сигналу через стандартні лінії зв'язку. Це дозволяє досягти майже одночасного спрацювання тригерів, які розташовані в різних місцях мікросхеми.

Крім того джерелом для глобальних сигналів можуть слугувати також виводи подвійного використання DPCLK (Dual-Purpose Clock), кількість яких дорівнює 8 або 16, виходи вбудованих блоків ФАПЧ (PLL) та внутрішня логіка проекту. Підключення сигналів до ліній глобального розповсюдження може виконуватись або автоматично самим компілятором пакету Quartus II (опція **Auto Global Clock** в діалозі **More Settings** сторінки **Fitter Settings** меню **Settings**) або розробником за допомогою примітива **Global** як в графічному редакторі так і при використанні мови опису апаратури.

## 1.3. Архітектура мікросхем Spartan

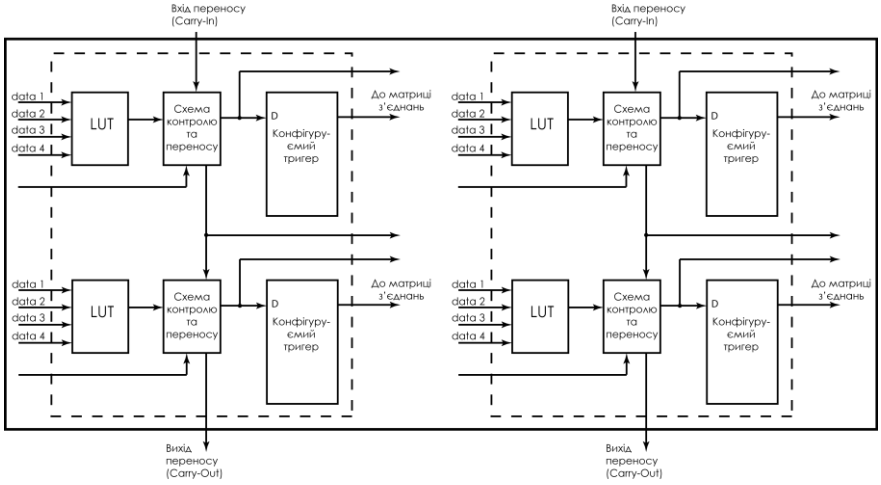



*Рисунок 1.16 – Архітектура секції (Slice) мікросхеми Spartan 3*

Мікросхеми сімейств Spartan належать до так званих дешевих мікросхем (Lowest Cost). Архітектуру FPGA мікросхем компанії Xilinx розглянемо на прикладі мікросхеми Spartan 3. Основу мікросхеми складають конфігуруємі логічні блоки, які складаються з чотирьох логічних чарунок, що організовуються у вигляді двох однакових секцій (Slice). Одна з таких секцій показана на рисунку 1.16.

Чарунка має в своєму складі таблицю перекодування (LUT) та конфігуруємий тригер. Тобто архітектура логічного елемента мікросхем сімейства Spartan дуже подібна до архітектури логічного елемента мікросхем сімейства Cyclone. Відмінність полягає в тому, що логічний блок включає в себе чотири чарунки. Чарунки згруповані в пари SliceM та SliceL, які мають спільні ланцюги переносу. Як і в мікросхемах Cyclone чарунки можуть працювати в звичайному та арифметичному режимах. Тобто мікросхеми Cyclone та Spartan мають багато спільного і тому детально зупинятись на особливостях архітектури ми не будемо. Для більш докладного ознайомлення з мікросхемами Spartan можна звернутись до [10, 16]. Параметри найменшої та найбільшої з мікросхем сімейства Spartan 3 наведені в таблиці 1.3.

*Таблиця 1.3 – Параметри мікросхем сімейства Spartan 3*

| Параметр | XC3S50 | XC3S5000 |
|---|---|---|
| Логічних елементів | 1 728 | 74 880 |
| Загальний об’єм пам’яті, Кбіт | 84 | 2 392 |
| Блоків ФАПЧ | 2 | 4 |
| Вбудованих помножувачів | 4 | 104 |
| Виводів, доступних користувачу | 124 | 784 |

## 1.4. Архітектура мікросхем CPLD

Архітектуру CPLD мікросхем розглянемо на прикладі сімейства MAX 3000 компанії Altera. На сьогоднішній день компанія Altera випускає два сімейства з класичною CPLD архітектурою: MAX 3000 та MAX 7000. В номенклатурі компанії Xilinx подібну архітектуру мають мікросхеми сімейств XC9500 та CoolRunner. Параметри мікросхем сімейства MAX 3000 наведені в таблиці 1.4 [14]. Часові параметри, що використовуються в таблиці, мають наступне значення: $t_{PD}$ – затримка проходження сигналу без тактування крізь мікросхему, $t_{CO}$ – час між приходом тактового сигналу до появи сигналу на виході мікросхеми, $f_{CNT}$ – максимальна тактова частота для тригерів.

*Таблиця 1.4 – Параметри мікросхем MAX3000*

| Параметр | EPM3032A | EPM3064A | EPM3512A |
|---|---|---|---|
| Макрочарунок | 32 | 64 | 512 |
| Логічних блоків | 2 | 4 | 32 |
| $t_{PD}$, нс | 4,5 | 4,5 | 7,5 |
| $t_{CO}$, нс | 3,0 | 3,1 | 4,7 |
| $f_{CNT}$, МГц | 227,3 | 222,2 | 116,3 |

Як видно з рисунку 1.17 основою мікросхеми MAX 3000 є макрочарунка (Macrocell), яка складається з матриці розподілу термів (Product-Term Select Matrix) та конфігуруємого тригера (Programmable Register).

Для реалізації логічної функції використовуються локальна програмуєма матриця з'єднань (Logic Array Block Local Array) та матриця розподілу термів, які можуть об'єднувати за допомогою функції «АБО» або «Виняткове АБО» результати логічних добутків.

Для реалізації логічної функції на основі декількох макрочарунок одного логічного блока використовується логічний

розширювач (Expander Product Terms). Розділяємий логічний розширювач формує інверсію терма і подає його значення на локальну програмуєму матрицю де він може використовуватись будь-якою макрочарункою логічного блоку. Паралельний логічний розширювач дозволяє використовувати терми з сусідніх макрочарунок для реалізації складних функцій, до складу яких входить більше 5 термів.

Як видно з рисунку 1.17 в матрицю з'єднань має в своєму складі надходять 36 сигналів з інших логічних блоків та 16 сигналів з розділяємого паралельного логічного розширювача. Тобто на основі одного логічного блоку можна реалізувати логічну функцію з 52 термів.

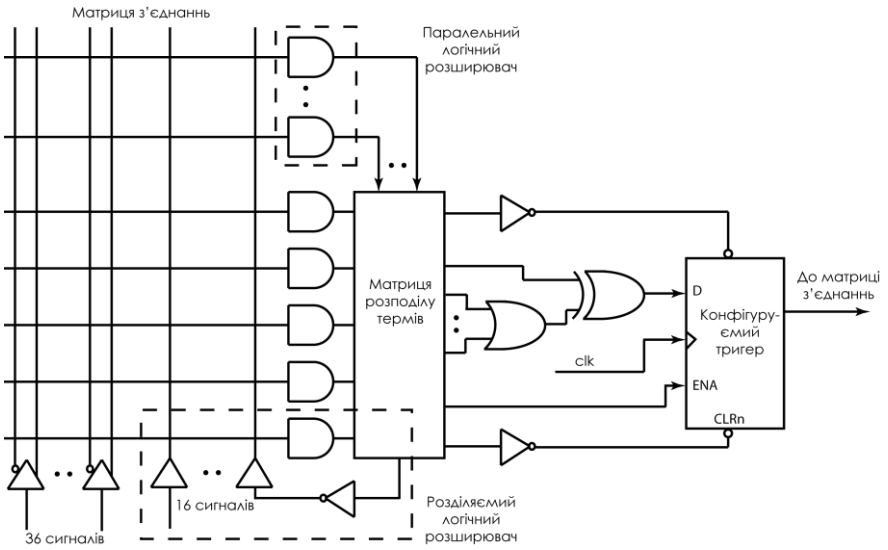



*Рисунок1.17 – Макрочарунка мікросхеми MAX 3000*

Макрочарунки об'єднуються в логічні блоки (рисунок 1.18) які, в свою чергу, з'єднані між собою та зовнішніми контактами ПЛІС за допомогою програмованої матриці зв'язків (ПМЗ).

Як і в мікросхемах типу FPGA MAX 3000 дозволяє використовувати глобальні сигнали. Але їх кількість значно менша –

лише дві лінії тактування (GCLK1, GCLK2), сигнал скидання та дві лінії дозволу роботи (OE1, OE2).

Елементи вводу-виводу мають архітектуру, подібну до архітектури цих блоків в мікросхемах Cyclone II. Тому її наводити не будемо.

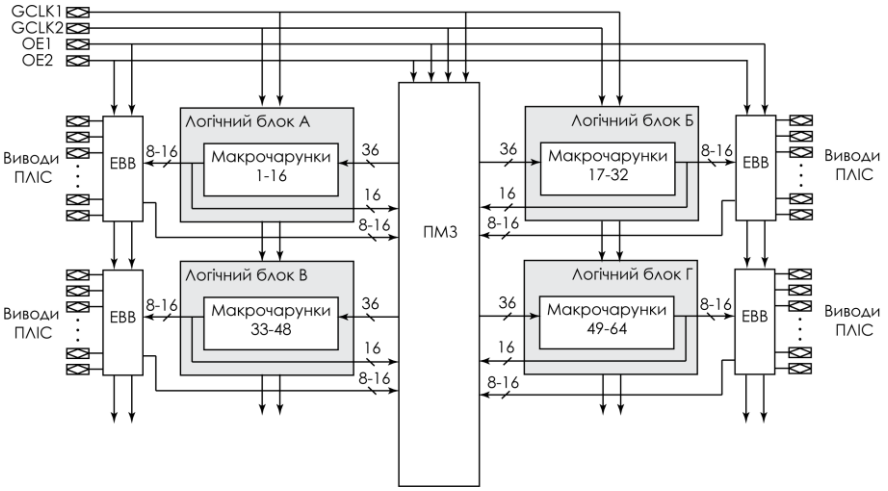



*Рисунок 1.18 – Архітектура мікросхеми MAX 3000*

## 1.5. Архітектура мікросхем MAX II

Мікросхеми сімейства MAX II займають проміжну ланку між класичними CPLD та FPGA мікросхемами. Як і всі CPLD мікросхеми у MAX II конфігураційна інформація зберігається у внутрішній Flash пам'яті і для цих мікросхем зовнішній ПЗП для зберігання конфігурації не потрібен. Логічний елемент мікросхем MAX II подібний до логічних елементів FPGA мікросхем. Робота логічних елементів описана в розділі присвяченому мікросхемам Cyclone II, тому тут на ній ми зупинятися не будемо. Також в мікросхемах

сімейства MAX II використовується архітектура програмованої матриці зв'язків подібна до FPGA. Для забезпечення зв'язку між логічними блоками використовуються рядки і стовпчики матриці зв'язків замість однієї загальної матриці. Це дозволяє значно зменшити розмір кристалу при збільшенні кількості логічних блоків у порівнянні з класичними CPLD (рисунок 1.19).

Мікросхеми сімейства MAX II Z мають низьку споживану потожність у режимі standby – 29 мкА (для мікросхеми EPM240Z) [15].

Параметри мікросхем сімейства MAX II наведені в таблиці 1.5.

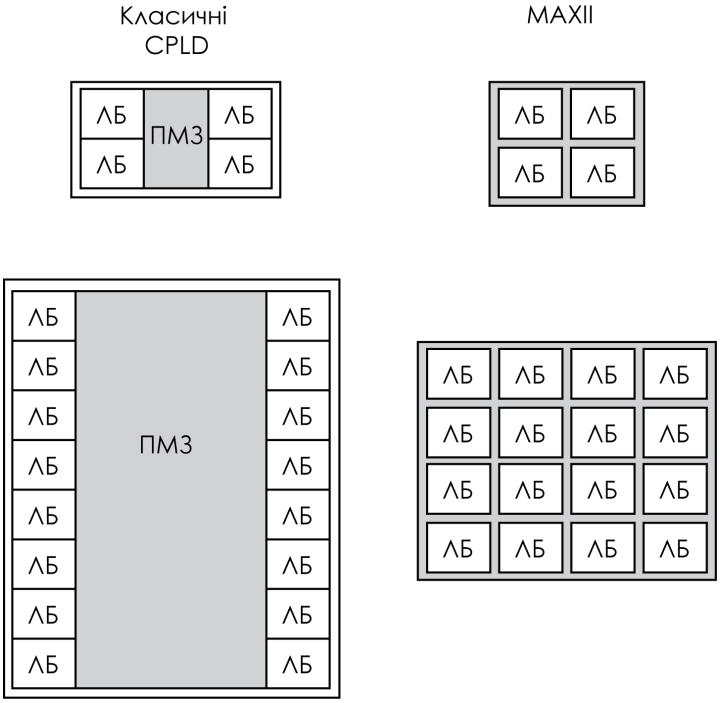



*Рисунок 1.19 – Порівняння площі кристалу для класичних CPLD та MAX II*

*Таблиця 1.5 – Параметри мікросхем сімейства MAX II*

| Параметр | EPM240 | EPM240G | EPM2210 | EPM240Z |
|---|---|---|---|---|
| Логічних елементів | 240 | 240 | 2210 | 240 |
| $t_{PD}$, нс | 4,7 | 4,7 | 7,0 | 7,5 |
| $t_{CO}$, нс | 4,3 | 4,3 | 4,6 | 6,5 |
| $f_{CNT}$, МГц | 304 | 304 | 304 | 152 |
| $I_{CC\ STANDBY}$ | 12 мА | 2 мА | 12 мА | 29 мкА |

## 1.6. Конфігурування мікросхем ПЛІС

Для роботи ПЛІС необхідно завантажити в неї конфігурацію. Як вже було зазначено раніше, конфігурація може зберігатись як статичній пам'яті (SRAM), так і в постійному запам'ятовуючому пристрої (EEPROM або Flash). До SRAM мікросхем можна віднести Flex10K, Cyclone, Stratix, Arria компанії Altera та Virtex і Spartan компанії Xilinx. До мікросхем з ПЗП: MAX3000, MAX7000, MAXII компанії Altera та CoolRunner і XC9500 компанії Xilinx.

Для програмування та конфігурування мікросхем ПЛІС можуть використовуватись програматори, зовнішні ПЗП та завантажувальні кабелі. Завантажувальні кабелі можуть використовуватись при програмуванні мікросхеми, яка вже встановлена на друковану плату. Така технологія носить назву внутрішньосхемного програмування (In-System programming – ISP).

При використані зовнішніх ПЗП конфігураційна інформація зберігається в спеціалізованих мікросхемах і при включенні живлення завантажується до ПЛІС. Слід відмітити, що конфігураційні ПЗП відрізняються від звичайних ПЗП тим, що крім збереження інформації вони ще повинні працювати з мікросхемою ПЛІС за стандартним протоколом конфігурування. Протокол залежить від режиму

конфігурування. Для мікросхем ПЛІС можливо декілька режимів програмування [13, 17]:

- пасивний послідовний (Passive Serial);
- асинхронний пасивний послідовний (Passive Parallel Asynchronous);
- пасивний паралельний (Passive Parallel);
- швидкий пасивний паралельний (Fast Passive Parallel);
- програмування через JTAG-інтерфейс;
- активний послідовний (Active Serial).

Вибір режиму конфігурування визначається комбінацією на входах MSEL. Для різних сімейств мікросхем кількість входів може бути 2 або 3. Для прикладу в таблиці 1.6 наведені режими конфігурування мікросхем сімейства Stratix.

*Таблиця 1.6 – Режими конфігурування мікросхеми Stratix*

| MSEL2 | MSEL1 | MSEL0 | Режим конфігурації |
|---|---|---|---|
| 0 | 0 | 0 | Швидкий паралельний пасивний (Fast Passive Parallel – FPP) |
| 0 | 0 | 1 | Пасивний паралельний асинхронний (Passive Parallel Asynchronous – PPA) |
| 0 | 1 | 0 | Пасивний послідовний (Passive Serial – PS) |
| 1 | 0 | 0 | Оновлення конфігурації (FPP) |
| 1 | 0 | 1 | Оновлення конфігурації (PPA) |
| 1 | 1 | 0 | Оновлення конфігурації (PS) |
| X | X | X | Програмування за допомогою JTAG інтерфейсу |

На рисунку 1.20 наведена схема конфігурування мікросхем ПЛІС за допомогою послідовного конфігураційного пристрою. В такому режимі процес конфігурування контролюється мікросхемою ПЗП. Тобто мікросхема ПЗП крім власне пам’яті також містить контролер конфігурації, який виробляє команди для конфігурування ПЛІС. У тому випадку, коли необхідно провести конфігурування

кількох мікросхем ПЛІС схема трохи змінюється і конфігурування мікросхем відбувається послідовно, одна за одною. Використання активних конфігураційних пристроїв підвищувало вартість системи, тому при розробці мікросхем сімейства Cyclone було вирішено перенести контролер конфігурації до мікросхеми ПЛІС, а зовнішню пам'ять використовувати тільки для збереження конфігураційної інформації.

Конфігурування за допомогою завантажувального кабелю є основним режимом при проведенні експериментальних досліджень та лабораторних практикумів. Схема підключення завантажувального кабелю до ПЛІС наведена на рисунку 1.21.

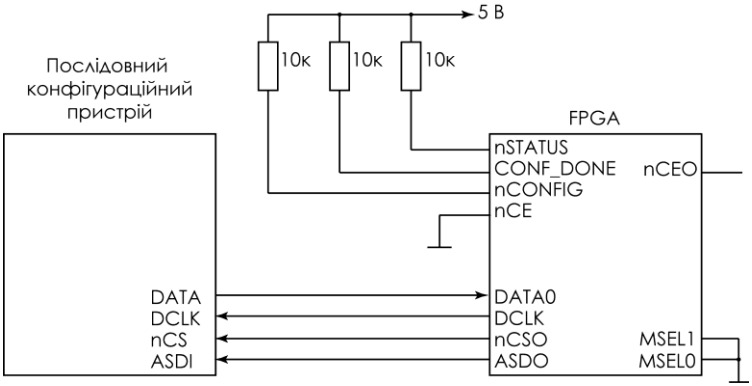



*Рисунок 1.20 – Конфігурування за допомогою послідовного конфігураційного пристрою*

В цьому випадку необхідно під'єднати кабель до паралельного інтерфейсу персонального комп'ютера. В цьому випадку використовується завантажувальний кабель ByteBlaster II або ByteBlaster MV. Сучасні мікросхеми ПЛІС також можуть програмуватися через порт порт USB за допомогою кабеля USB Blaster [17].

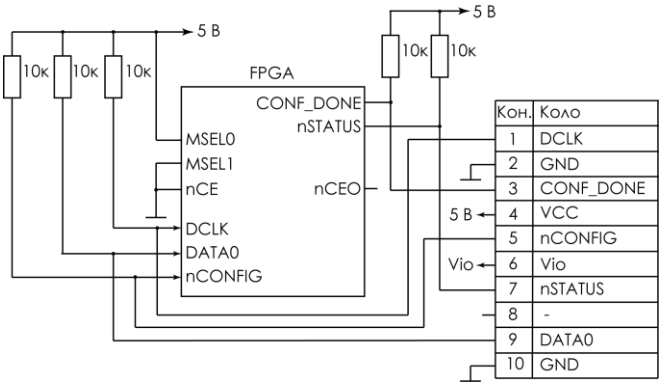



*Рисунок 1.21 – Конфігурування ПЛІС за допомогою завантажувального кабелю*

Якщо в системі викростовуються ПЛІС великого об'єму, то необхідно використовувати і велику за об'ємом конфігураційну пам'ять. Для зменшення об'єму конфігураційного файлу можна використати компресію конфігураційних даних, що зменшує об'єм конфігураційного файлу приблизно на 50 %. Для декомпресії даних в ПЛІС необхідно встановити спеціальну схему, яка буде виконувати цю операцію.

Дуже часто виникає питання: а чи можливо, прочитавши конфігураційну інформацію, отримати схему проекту або файл на мові опису апаратури? Відповідь на це питання негативна. Можна при конфігуруванні пристрою прочитати бітовий потік, що йде з конфігураційного пристрою до ПЛІС, але це дозволить отримати лише конфігураційний файл, а не схему проекту. Конфігураційна інформація не є аналогом машинних кодів у програмі для комп'ютера або контролера. Тому навіть маючи конфігураційних файл неможливо з нього отримати схему проекту або програму на мові опису апаратури.

Для унеможливлювання зчитування інформації з мікросхеми ПЛІС в мікросхемах сімейств MAX 7000, MAX 3000, MAX II використовується біт секретності, який неможливо зняти. Для мікросхем сімейств Stratix використовується кодування даних за допомогою 128-розрядного ключа, який зберігається в спеціальній

пам'яті самої мікросхеми, а з конфігураційного пристрою йде кодований потік даних, який декодується вже в самій мікросхемі.

## *Література*

1. Верма С. Как объективно оценить параметры FPGA разных производителей?// Электронные компоненты, 2009, №1 – с. 40-44.

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

3. Зотов В.Ю. Проектирование цифровых устройств на основе ПЛИС фирмы Xilinx в САПР WebPACK ISE. – М.: Горячая линия-Телеком, 2003. – 624 с.

4. Максфилд К. Проектирование на ПЛИС. Курс молодого бойца. М.: Изд.дом "Додэка-XXI", 2007. – 408 с.

5. Мальцев П. П., Долидзе Н. С., Критенко М. И. и др. Цифровые интегральные микросхемы: Справочник.М.: Радио и связь, 1994. – 240 с.

6. Перепрограммируемые в системе ПЛИС CPLD семейства XC9500. Краткое техническое описание. 35 с. http://www.plis.ru /pic/pict/ File/ 9500_rus .pdf

7. ПЛИС с архитектурой FPGA семейства Spartan™-3 (1,2 В). Краткое техническое описание. ЗАО «ИНЛАЙН ГРУП», 2005. – 33 с. http://www.plis.ru/pic/pict/File/Spartan3.pdf

8. ПЛИС с архитектурой FPGA семейства VirtexTM (2.5В). Краткое техническое описание. 27 с. http://tsdig.ru/custom/ virtex_rus.pdf

9. Стешенко В.Б. ПЛИС фирмы "Altera": элементная база, система проектирования и языки описания аппаратуры. М.: Издательский дом "Додэка-XXI", 2002. – 576 с.

10. Современные семейства ПЛИС фирмы Xilinx: справочное пособие / М.О. Кузелин, Д.А. Кнышев, В.Ю. Зотов. – М.: Горячая линия-Телеком, 2004. – 440 с.

11. Угрюмов Е.П. Цифровая схемотехника. – СПб.: БХВ-Санкт-Петербург, 2000. – 528 с.

12. CoolRunner-II CPLD Family. Product Specification. Xilinx inc., 2008. – 16 p.

13. Cyclone II Device Handbook. Altera Corporation, 2008. – 470 p.

14. MAX 3000A Programmable Logic Device Family Data Sheet. Altera Corporation, 2006. – 46 p.

15. MAX II Device Handbook. Altera Corporation, 2007. – 106 .

16. Spartan-6 Family Overview. Product Specification. Xilinx inc., 2011. – 11 p.

17. Stratix V Device Overview. Altera Corporation, 2012. – 22 p.

18. Virtex-6 Family Overview. Product Specification. Xilinx inc., 2012. – 11 p.

# Розділ 2

# Мова опису апаратури VHDL


- Мови опису апаратури ..... 43
- Рівні проектування мікросхем ..... 45
- Структура проекту на мові VHDL. ..... 47
- Типи даних. Літерали ..... 54
- Константи, сигнали та змінні ..... 69
- Оператори мови VHDL ..... 71
- Поради по написанню текстів на мові VHDL ..... 99

## 2.1. Мови опису апаратури.

Постійне ускладнення електронних пристроїв вимагає від розробників підвищення швидкості розробки, простого переходу між платформами проектування та наглядного опису проекту. Використання мов опису апаратури дозволяє забезпечити високорівневий опис складних систем, підтримку бібліотек проектування, сумісність коду, написаного для різних середовищ розробки. На сьогоднішній день найбільш часто використовуються дві мови опису апаратури: VHDL та Verilog HDL. Ці мови вважаються стандартними і підтримуються майже всіма засобами моделювання та синтезу пристроїв на ПЛІС. Крім цих мов існують менш поширені AHDL, Abel, SystemC, SystemVerilog, Verilog-ASM [1, 5, 6, 8, 10]. У цьому виданні ми зупинимось на розгляді мови VHDL.

Мова VHDL (VHSIC (Very-high Speed Integrated Circuits) Hardware Description Language) була розроблена у США з ініціативи міністерства оборони цієї країни. Первинним призначенням мови VHDL було документування проектів і електронних виробів різних рівнів складності. Це дозволяло розробникам і користувачам легко та однозначно розуміти функції різних пристроїв. Надалі коло завдань, які розв'язувались за допомогою VHDL, розширилося. В 1981 році відбулася перша конференція, на якій були сформульовані вимоги до мови. Версія VHDL 7.2, що з'явилась в серпні 1985 року, задовольняла більшості вимог і була прийнята в якості стандарту в рамках Міністерства Оборони США. У червні 1986 вийшла перша робоча версія посібника з мови (Language Reference Manual). Остаточна версія проекту мови була винесено на голосування 11 грудня 1987 року і прийнята в якості стандарту IEEE STD 1076-1987. Даний стандарт часто називають VHDL'87.

В інституті інженерів з електротехніки та електроніки (Institute of Electrical and Electronics Engineers, IEEE) усі стандарти повинні переглядатися кожні п'ять років. Перший перегляд стандарту на VHDL відбувся в 1992 році, хоча підготовка до цього почалася в червні 1990 року. Новий стандарт ANSI/IEEE Std 1076-1993 (стандарт VHDL'93) з'явився в 1993 р.

В 1999 р. затверджений стандарт Std 1076.1-1999 (або більш розповсюджене найменування VHDL-AMS), який включає розширення, що дають можливість опису моделей аналогових і змішаних (цифро-аналогових) схем. В 2000 році з'явилася доопрацьована версія IEEE Std 1076-2000. Остання на сьогоднішній день версія мови VHDL описана в стандарті 2008 року. В Росії мова VHDL описана в державному стандарті 1995 року [11].

Як говорилося, початковою метою створення мови було моделювання і опис обладнання, а функція синтезу була додана пізніше. Це призвело до того, що не всі конструкції, оператори і типи даних VHDL можуть бути автоматично перетворені в принципову схему. Тому розділяють синтезовані та несинтезовані підмножини VHDL. В цьому виданні ми більшу частину уваги звернемо саме на синтезовану підмножину мови VHDL. Тому при написанні програм на мові VHDL слід пам'ятати, що VHDL код синтезується в електричну цифрову схему, яка потім буде програмуватися в ПЛІС.

В подальшому ми будемо використовувати для роботи з мовою VHDL пакет Quartus II. Використання цього пакету не обов'язкове і читач може використовувати будь-який компілятор цієї мови. Перед використанням необхідно лише звернути увагу на те, як саме в пакеті підтримується мова, чи всі конструкції допустимі при написанні програм. Наприклад, при використанні пакету Quartus II можна звернутись до довідки у розділі Quartus II VHDL Support.

Коротко зупинимось на літературі, яка може бути рекомендована для вивчення мови VHDL.

Стандарт мови VHDL [13] та його російський аналог [11] можуть бути рекомендовані лише досвідченому розробнику. Початківцю ці стандарти використовувати буде складно.

Короткий курс VHDL наведено в книзі [10]. При невеликому обсязі в книзі наведені найбільш поширені конструкції мови. По тексту книги наводиться багато прикладів опису цифрових блоків.

Книги білоруського автора Бібіло П.Н. можуть бути використані як довідник. А книга [3] наводить велику кількість прикладів коду, готового до синтезу.

Книги [3, 7, 9] наводять повний опис мови VHDL і можуть бути корисними при вивченні як мови в цілому, так і окремих аспектів уживання, але містять приклади несинтезуємої частини мови.

Для вивчення мови Verilog можна використовувати книгу [6]. Також слід звернути увагу на цикл статей Й.Каршенбойма в журналі «Компоненти і технології» [5].

Англомовна література представлена великою кількістю видань. З них можна порекомендувати [12, 14]. Також слід звернути увагу на розділ Recommended HDL Coding Styles книги Quartus II Handbook [15].

## *2.2. Рівні проектування мікросхем*

Цифрова мікросхема в залежності від точки зору на її структуру та принципи функціонування може бути описана моделями трьох стилів [9, 10]: структурного, функціонального (поведінкового) та геометричного (топологічною). Таке представлення мікросхеми носить назву діаграми Гайського-Кана (Gajski and Kuhn). Проектування мікросхеми носить послідовний характер переходу від більш складних рівнів до простіших. При цьому виконується також і перехід від одного стилю опису до іншого.

Кожен стиль формує свою ієрархію представлення мікросхеми. Кожний наступний рівень опису в кожній ієрархії більш абстрактний, ніж попередній (рисунок 2.1). Поява кожного наступного рівня передбачає, що перехід на нижчі рівні опису виконується програмами автоматизованого проектування. Чим більш абстрактний рівень проектування, тим більш складна система розробляється, і тим вищі вимоги до програного забезпечення. Для простих систем опис відразу ж компілюється в принципову схему, для складних систем подібна операція в найкращому разі приводить до неоптимальних і громіздких реалізацій. Тому в таких випадках необхідна деталізація проекту, що дозволяє підсилити вплив розробника на компіляцію проекту і загальний результат в цілому.

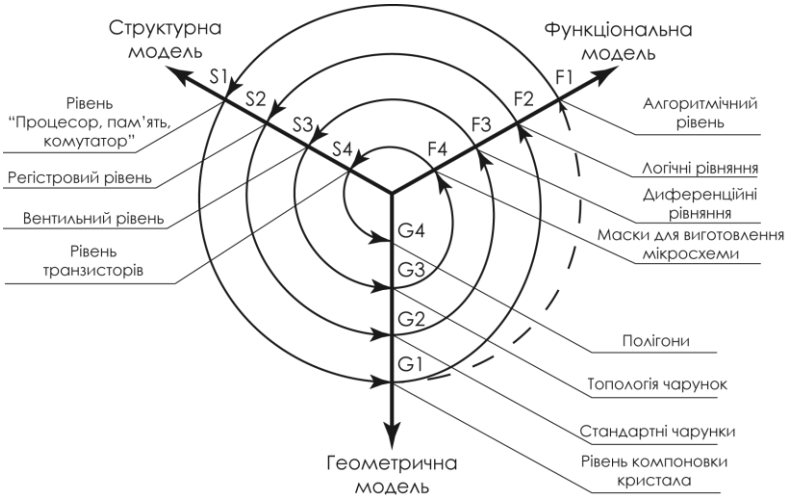



*Рисунок 2.1 – Рівні моделей при проектуванні СБІС. Діаграма Гайського-Кана*

Для **структурної моделі** виділяють такі рівні:

- *рівень транзисторів* (Transistors Level). Тут принципова схема представляється транзисторами та іншими дискретними елементами.
- *вентильний рівень* (Gates Level). На цьому рівні схема представляється як з'єднання вентилів. А перехід на рівень транзисторів виконується автоматично програмним забезпеченням.
- *регістровий рівень* (RTL – Register Transfer Level). Рівень функціональних блоків – регістрів, лічильників, дешифраторів, мультиплексорів і т.д. На цьому рівні найбільш часто виконується робота розробника і опис схеми на допомогою мов опису апаратури.
- *рівень великих функціональних блоків*. При описі цього блоку найбільш часто згадуються процесор, пам'ять і

системна шина з арбітром (Processor, Memory, Switch – PMS-level). Також цей рівень інколи називають системним рівнем. Проектування на цьому рівні виконується за допомогою програмного забезпечення для опису систем. Частково такі функції можуть виконувати пакети MATLAB/Simulink або SOPC Builder для ПЛІС Altera.

**Функціональна модель** поступово проходить від алгоритму роботи мікросхеми до масок для виготовлення окремих елементів.

**Геометрична модель** працює з топологією мікросхеми і розміщенням окремих елементів системи на кристалі. При роботі в пакеті Quartus II можна працювати лише на рівні компоновки кристала (Floor plan), що описується в розділі 4. Всі інші рівні розробнику систем на ПЛІС недосяжні.

## *2.3. Структура проекту на мові VHDL.*

Проект на мові VHDL представляється сукупністю ієрархічно зв'язаних текстових файлів, які називаються проектними модулями. Типовий текст програми мовою VHDL має наступну структуру (рисунок 2.2):

- декларація бібліотек;
- інтерфейс об’єкта проекту;
- архітектура об’єкта проекту.

### 2.3.1. Декларація бібліотек

Бібліотека у VHDL складається з пакетів, які в свою чергу – з окремих модулів – процедур, функцій, блоків і т.д. Для використання їх у тексті програми необхідно оголосити бібліотеки, які будуть використовуватись в даному проекті. Після декларації бібліотеки необхідно визначити пакети з даної бібліотеки та модулі з обраного пакету.

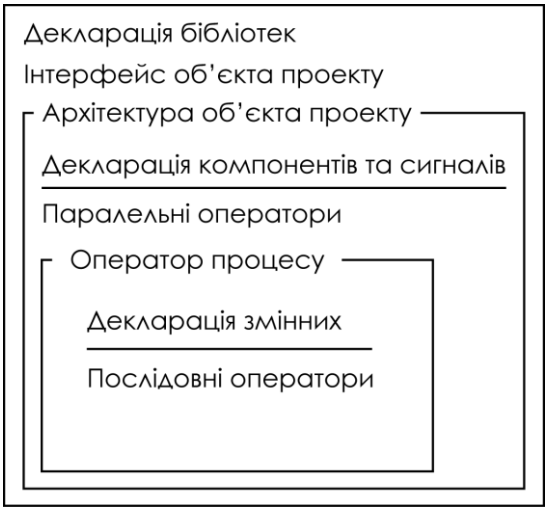



*Рисунок 2.2 – Структура проекту на мові VHDL*

```
use ім'я_бібліотеки.ім'я_пакета.ім'я_модуля;
use ім'я_бібліотеки.ім'я_пакета.all;
```

Варіант запису **use** … **all** забезпечує доступ до всіх модулів оголошеного пакета. Якщо в тексті програми використовується декілька пакетів з бібліотеки, то вони описуються без додаткової об'яви використання бібліотеки.

Наведемо приклад використання найбільш поширених бібліотек.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

В проекті за замовчуванням використовується бібліотека *std*, використання якої оголошувати не потрібно. Бібліотеки *std_logic_1164* та *numeric_std* описані в розділі 2.3.

Бібліотечний модуль може містити наступну інформацію:

– описи процедур та функцій;
– описи компонентів;
– оголошення констант;
– оголошення типів.

## 2.3.2. Інтерфейс об'єкта проекту

Інтерфейс об'єкту, сутність проекту (**entity**) визначає його ім'я, входи-виходи та параметри. Входи-виходи – це перелік портів (**port**), їх імена, напрямок передачі даних (входи – **in**, виходи – **out**, двонаправлені – **inout**), типи сигналів, які описують розрядність (один біт – *bit*, *std_logic* або вектор – *bit_vector*, *std_logic_vector*), алфавіт кодування сигналів (наприклад, 0, 1), спосіб представлення їх значень (цілий – *integer*, дійсний – *real*) і т.д.

Тіло об'єкту проекту визначає його структуру і функцію, поведінку, алгоритм. Його опис виділяється в окрему частину і наведений в описі архітектури об'єкту (**architecture**).

Спрощена форма запису декларації **entity** має вигляд:

```
entity ім'я_проекту is
port      (ім'я_сигналу:      напрямок_передачі
тип_сигналу;
ім'я_сигналу: напрямок_передачі тип_сигналу;
   …
   ім'я_сигналу:            напрямок_передачі
тип_сигналу);
end ім'я_проекту;
```

**Увага! При використанні пакету Quartus II слід пам'ятати, що ім'я проекту, яке записується після слова *entity* повинно співпадати з іменем файлу, в якому зберігається ця програма.**

Оголошення портів

```
port      (ім'я_сигналу:      напрямок_передачі
тип_сигналу;
ім'я_сигналу: напрямок_передачі тип_сигналу
) ;
```

Напрямок передачі сигналу може приймати такі значення:

**in** – вхідний сигнал на вході об'єкта;

**out** – вихідний сигнал на виході об'єкта; значення сигналу неможливо зчитувати всередині об'єкта, його значення доступне тільки об'єктам, які використовують даний об'єкт;

**inout** – двонаправлений сигнал, який може бути як вхідним так і вихідним.

Наприклад, в об'єкта проекту з іменем SM (рисунок 2.3) три вхідні (**in**) лінії: A, B, C і дві вихідні (**out**): S і P, на які можуть надходити сигнали, що мають двійкові (**bit**) значення 0 або 1. Цей об'єкт буде мати наступний інтерфейсний опис.

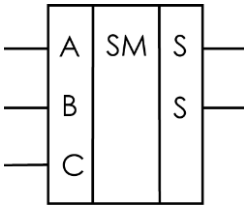



*Рисунок 2.3 – Суматор*

```
entity SM is
port (A, B, C: in bit;
   S, P: out bit);
end SM;
```

Оголошення параметрів налаштування включається в *entity* для створення об'єктів, які передбачається використовувати як фрагменти в різноманітних інших модулях. Це дозволяє створювати параметризовані модулі в яких можлива модифікація деяких властивостей фрагмента за допомогою вибору параметра з множини значень, визначеної типом параметра.

Оголошення параметрів настроювання:

```
generic (ім'я_парамера: тип:=
         значення_параметра);
```

Визначення портів задає імена ліній передачі сигналів та тип даних, які передаються через порти. Оголошення портів не є обов'язковим. Така конструкція дозволяє ефективно поєднувати опис обладнання, яке проектується та алгоритм його тестування. Програми, які створюються для налагодження та тестування, носять назву Test-bench. Зазвичай така програма включає опис як самого проектованого обладнання, так і модель зовнішнього середовища, зокрема генератора тестового впливу.

Для параметрів і вхідних портів можна задавати значення за замовчуванням, що представляється виразом, відділеним від типу знаками :=. Ці значення використовуються в тому випадку, коли відповідним параметрам або портам не присвоєні інші значення в модулях вищого рівня ієрархії. Також необхідно пам’ятати, що значення за замовчуванням для вхідних портів можуть бути присвоєні лише в тому випадку, коли порти знаходяться всередині мікросхеми ПЛІС. У тому ж випадку, коли вхідний порт підключений до зовнішнього виводу мікросхеми значення за замовчуванням присвоєне не буде.

## 2.3.3. Архітектура об’єкта проекту

Архітектурні тіла представляють змістовний опис проекту. Розрізняють структурні архітектурні тіла, що описують проект у вигляді сукупності компонентів і їх з'єднань, поведінкові архітектурні тіла, які описують проект як сукупність дій, що виконуються, і змішані тіла. Формальних ознак, по яких архітектурне тіло можна було б віднести до певного типу, не вводиться – відмінність виявляється у складі операторів, які використовуються у програмі. Визначені наступні правила запису архітектурних тіл:

```
architecture ім'я_архітектури of ім'я_entity
is
        декларація типів
        декларація сигналів
        декларація констант
        декларація функцій
        декларація процедур
```

```
        декларація компонентів
begin
        паралельні оператори
          …
        паралельні оператори
end ім'я_архітектури;
```

Тут ім'я архітектури – це будь-яке індивідуальне ім'я проектного модуля. Ім'я **entity** задає первинний модуль, якому підпорядковується архітектурне тіло. У розділі декларацій оголошуються локальні для цього модуля інформаційні одиниці – типи даних, сигнали, підпрограми і т.д. Розділ операторів описує правила функціонування та (або) конструювання обладнання в термінах мови. Одній інтерфейсній частині може відповідати декілька архітектурних тіл. Ці архітектурні тіла записуються в одному файлі, але компілятор аналізує лише останню архітектуру в файлі. Декілька архітектурних тіл в одному файлі необхідні для документування проекту. Кожна наступна архітектура є більш детальною, ніж попередня, переходячи поступово з більш абстрактних рівнів до більш деталізованих описів.

## 2.3.4. Спрощений приклад програми на мові VHDL

Нижче наведений приклад програми, яка описує D-тригер, показаний на рисунку 2.4. Тригер по фронту сигналу clk записує інформацію з входу D. Вихідний сигнал тригеру надходить до порту Q.

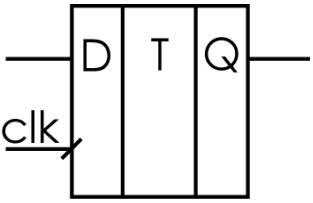



*Рисунок 2.4 – D-тригер*

**Приклад 2.1 – D-тригер.**

```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3
 4  entity d_trigger is
 5  port (
 6     d:          in    std_logic;
 7     clk:  in    std_logic;
 8     q:          out   std_logic);
 9  end d_trigger;
 10
 11 architecture arh of d_trigger is
begin
 12 process (clk)
    begin
 13    if (clk'event and clk = '1')
 14    then  q <= d; end if;
 15 end process;
 16 end arh;
```

Розберемо текст програми.

Рядки 1-2 містять оголошення бібліотеки та пакету який буде використовуватись. З обраного пакету будуть використані всі функції. Бібліотеку *std* оголошувати не потрібно – вона підключена за замовчуванням.

Рядки 5-9 містять опис інтерфейсу (**entity**) тригера. Інтерфейс носить назву *d_trigger* і містить два вхідних (**in**) порти *d* та *clk* типу *std_logic*, а також вихідний (**out**) порт *q* також типу *std_logic*.

Рядки 11-18 містять інтерфейсну частину тригеру.

Рядки 11 вводить архітектуру з назвою *arh*, яка належить до інтерфейсної частини *d_trigger*.

Рядки 12 та 18 починають (**begin**) та закінчують (**end**) опис архітектури. Після зарезервованого слова **end** необхідно вказати назву архітектури – *arh*.

Рядки 13-17 містять оператор процесу (**process**) більш докладний опис якого наведений в розділі 2.5.

# 2.4. Типи даних. Літерали.

VHDL є мовою з суворою типізацією. Це означає, що застосування різних типів даних і літералів в одній операції є помилкою. Засоби суворого контролю типів відіграють велику роль, оскільки дозволяють уточнювати наміри розробника. Мова використовується для представлення апаратних проектів у різноманітних варіантах, тому засоби контролю типізації даних мають тут особливо велике значення. Наприклад, вони дають розробнику можливість представити групу ліній (провідників) шини у вигляді масиву бітів або цілого числа.

Тип – пойменована множина значень із деякими загальними характеристиками. Кожний тип даних в VHDL має певний набір прийнятих значень, формат збереження та передачі даних і набір дозволених операцій. У мові визначена достатня кількість простих і складених типів, а також є засоби для утворення типів, визначених користувачем.

Розрізняють скалярні типи даних і агрегатні типи. Об'єкт, віднесений до скалярного типу, розглядається як закінчена одиниця інформації. Агрегат являє собою впорядковану сукупність скалярних одиниць, об'єднаних однаковим ім’ям. Крім того, типи діляться на базові та ті, які визначаються користувачем.

## 2.4.1. Базові типи

Базові типи даних описані в пакеті *standart* бібліотеки *std*. До таких типів відносяться: *integer, real, bit, bit_vector, boolean, character, string, time, severity_level, file_open_status, file_open_kind*.

Ще раз нагадаємо, зазначити, що мова VHDL містить синтезовану та несинтезовану множини мови. Синтезована частина за допомогою програмного забезпечення може синтезувати принципову схему, яка потім реалізується на ПЛІС або спеціалізованих мікросхемах. Несинтезована множина, як зрозуміло з назви, не реалізується на рівні принципової схеми і використовується для опису

схеми та її моделювання. Крім того слід пам'ятати, що може також відрізнятись і підтримка мови опису апаратури програмним забезпеченням різним фірм-виробників [2]. У цій книзі описані лише конструкції та типи мови VHDL, які можуть бути синтезовані за допомогою програмного пакету Quartus II. Це конструкції мови VHDL версій 1987 та 1993 років. Перелік розділів стандарту мови VHDL, які підтримуються пакетом Quartus II, можна знайти у розділі *Quartus II VHDL Support* довідки пакету.

Оскільки не всі типи можуть бути синтезовані в електричну схему тому для синтезу доцільно використовувати тільки такі типи даних: *integer*, *bit*, *bit_vector*, *boolean*, *character*, *string*. Для ознайомлення з всіма іншими типами можна звернутись до [13].

Дані типу *bit* можуть мати значення з множини {'0', '1'}. Тип *bit* описує лише однобітні сигнали. Наприклад:

```
signal x: bit;
```

Тип *bit_vector* визначений як необмежений масив бітів. При визначенні масиву бітів необхідно вказати межі значень і напрямок зміни номера біта – вгору (**to**) або вниз (**downto**).

Взагалі, для будь-якого типу визначення обмеженого типу підкоряється синтаксичному правилу:

```
type базовий_тип is діапазон;
```

Визначення діапазону:

```
range обмеження_1 напрямок обмеження_2
```

Обмеження проводиться за допомогою слів **downto** та **to**. Слово **to** використовується при зростанні значень, тобто *обмеження_1* менше ніж *обмеження_2*, а слово **downto** – при зменшенні: *обмеження_1* більше ніж *обмеження_2*.

Наприклад:

```
signal y: bit_vector (0 to 3);
signal z: bit_vector (7 downto 0);
```

У першому прикладі описаний сигнал *y*, який є 4 бітним вектором і старший біт (MSB – Most Significant Bit) в ньому

знаходиться справа. У другому прикладі сигнал *z* є 8-бітним вектором зі старшим бітом, який знаходиться зліва.

Наведемо приклади присвоювання значень сигналам типу *bit* та *bit_vector*.

```
x <= '1';
y <= "0111";
z <= "00010001";
```

Сигналу *x* присвоєне значення '1'. При присвоєнні значень однобітним сигналам необхідно використовувати одинарні лапки (' '). Сигналу *y* присвоєне значення "0111" і старший біт має значення 1. Сигналу *z* присвоєне значення "00010001" і старший біт дорівнює 0. При операціях присвоювання векторам необхідно використовувати подвійні лапки (" ").

Над операторами типу *bit* можуть виконуватись логічні операції, перелік яких наведено в таблиці 2.1. Слід зазначити, що операція **not** має більший пріоритет ніж інші логічні операції і виконується в першу чергу. Для більш зручного читання та аналізу виразів все ж краще виконувати групування виразів по дужках.

*Таблиця 2.1 – Операції для типу bit*

| Операція | Значення |
|---|---|
| **and** | І |
| **or** | АБО |
| **nand** | І-НІ |
| **nor** | АБО-НІ |
| **xor** | Виключне АБО |
| **xnor** | Виключне АБО-НІ |
| **not** | Інверсія |

Приклади операцій для типу *bit*.

```
y <= not a and b;                -- a'×b
y <= not (a and b);              -- (a×b)'
y <= a nand b;                   -- (a×b)'
```

Дані типу *boolean* також можуть приймати два значення: *{true, false}* і над ними визначені ті ж самі операції, що й над даними типу *bit* (таблиця 2.1). Різниця між типами *bit* і *boolean* полягає в тому, що перші використовуються для представлення рівнів логічних сигналів в апаратурі, а другі – для роботи з результатами логічних операторів, наприклад, результатів порівняння. Так, якщо змінна *select* визначена як біт, то не можна записати умовний оператор у вигляді

```
if select then ...
```

Слід записати

```
if select = '1' then...
```

Якби змінна *select* була визначена як змінна типу *boolean*, то, навпаки, перший варіант був би припустимий, а другий – ні. Дані різних типів несумісні, тому наведений нижче вираз неприпустимий:

```
'0' and true
```

Типи *integer* і *real* визначають чисельні дані – цілі та дійсні, відповідно. Діапазон представлення чисел може залежати від реалізації, але стандартними вважаються діапазони $\{-2^{31}+1, +2^{31}-1\}$ для типу *integer* і { –1.0E38, +1.0E38} для *real*.

Операції для типу *integer* наведені в таблиці 2.2.

*Таблиця 2.2 – Оператори для типу integer*

| Операція | Значення |
|---|---|
| + | Додавання |
| - | Віднімання |
| * | Множення |
| / | Ділення |
| mod | Ділення по модулю |
| rem | Залишок від ділення по модулю |
| abs | Абсолютне значення |
| ** | Степінь числа |

Не всі операції над сигналами типу *integer* можуть бути реалізовані у будь-якому апаратному базисі. Так операції додавання та віднімання без будь-яких обмежень синтезуються в принципову схему. Теж саме стосується і операції множення. Для операції ділення без обмежень реалізується операція ділення на ступінь двійки, тобто операція зсуву. Для операції розрахунку ступеня числа тільки статичні значення основи та показника можуть бути без обмежень використані для синтезу схеми. Операції *mod, rem* та *abs* зазвичай за допомогою стандартних бібліотек або зовсім не синтезуються або частково.

## 2.4.2. Типи, визначені користувачем

Користувач має можливість визначити власні типи, використовуючи декларацію типу:

```
type ім'я_типу is визначення_типу;
```

При визначенні типу можуть бути використані наступні типи: перелічювальні, цілі, дійсні, фізичні, масиви та записи.

Визначення перелічувальних типів користувача доцільно, по-перше, для контролю сумісності даних у програмах, а по-друге, для точного завдання розрядності слів, що представляють дані в об'єкті, що проектується.

Приклад:

```
type my_unsigned_short is integer range 0 to 255;
type my_data is integer range 15 downto 0;
```

Крім типів, що описані в бібліотеці *std* і описані в стандарті мови VHDL існує ще декілька типів, що не описані стандартом, але дуже розповсюджені і вже стали стандартними «де-факто». Бібліотеки з цими типами поставляються з усіма засобами моделювання та синтезу мови VHDL.

Ці типи описані в таких бібліотеках та пакетах:

Пакет *std_logic_1164* бібліотеки *ieee* містить типи: *std_logic* та *std_ulogic*, а також розширення цих типів до векторного представлення: *std_logic_vector* та *std_ulogic_vector*.

Пакет *numerc_bit* бібліотеки *ieee* описує типи *signed* та *unsigned*, які базуються на типі *bit*, та функції для взаємного перетворення цих типів.

Пакет *numeric_std* бібліотеки *ieee* описує типи *signed* та *unsigned*, які основані на типі *std_logic_vector*, та функції для конвертації цих типів.

Відразу ж слід відзначити, що бібліотеку *std_logic_arith*, яка інколи зустрічається у прикладах програм на VHDL, краще не використовувати, оскільки вона замінена на *numeric_std* і відсутня у стандартній поставці пакету Quartus II.

Наведемо визначення типу *std_ulogic*:

```
type std_ulogic is ('U', 'X', '0', '1', 'Z',
'W', 'L', 'H', '-');
```

Тип *std_ulogic* і породжуваний на його основі підтип *std_logic* використовуються для представлення сигналів у дев'ятизначному алфавіті:

'U' – не ініціалізовано (у програмі взагалі сигналу не привласнювалися будь-які значення);

'X' – активний невизначений стан;

'0', '1' – активний нуль, активна одиниця;

'Z' – лінія знаходиться у високоімпедансному стані;

'W' – слабкий невизначений стан;

'L' – слабкий нуль;

'H' – слабка одиниця;

'-' – не важливо (вибір надається компілятору).

Більшість рівнів типу *std_logic* використовується лише для моделювання проектів. Для синтезу можуть бути використані лише сигнали зі значеннями '0', '1', 'Z'. У типі *bit*, як було вказано вище, існує лише два значення '0' та '1', чого може не вистачати для опису реальних схем. Тому при написанні програм на мові VHDL більш доцільно використовувати тип *std_logic*.

Різниця між слабкими та активними станами полягає в тому, що слабкий сигнал формується від джерел, називаних драйверами, що мають підвищений вихідний опір у порівнянні з активними джерелами. У цьому випадку джерело, що генерує активний сигнал, пригнічує слабкий, якщо він не відключений. У випадку, коли з'єднані

два сигнали типу *std_logic* з однаковими рівнями для визначення результуючого значення необхідно скористатися таблицею 2.3.

*Таблиця 2.3 – Значення результуючих сигналів для типу std_logic*

| | | Вхідний сигнал | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | X | 0 | 1 | Z | W | L | H | - |
| Вхідний сигнал | X | X | X | X | X | X | X | X | X |
| | 0 | X | 0 | X | 0 | 0 | 0 | 0 | X |
| | 1 | X | X | 1 | 1 | 1 | 1 | 1 | X |
| | Z | X | 0 | 1 | Z | W | L | H | X |
| | W | X | 0 | 1 | W | W | W | W | X |
| | L | X | 0 | 1 | L | W | L | W | X |
| | H | X | 0 | 1 | H | W | W | H | X |
| | - | X | X | X | X | X | X | X | X |

Для використання типу *std_logic* необхідно застосувати наступну конструкцію:

```
library ieee;
use ieee.std_logic_1164.all;
```

Крім типу *std_logic* значного розповсюдження набули типи даних, які є похідними від типу *std_logic*. Це типи *unsigned* та *signed*, які використовуються для опису беззнакових та знакових чисел. При використанні типу *unsigned* числа не можуть бути від нуля. Наприклад "0101" представляє число 5, а "1101" – число 13. При використанні знакових чисел типу *signed* від'ємні числа представляються у додатковому коді. Тому число "0101" буде так само дорівнювати 5, а "1101", записане у додатковому коді, відповідатиме числу -3.

Типи *unsigned* та *signed* визначені як масиви типу *std_logic* та описані в пакеті *numeric_std*. Для використання цих типів обов'язково декларувати використання бібліотеки ieee та пакету *numeric_std*.

```
library ieee;
use ieee.numeric_std.all;
```

Синтаксис сигналів типу *unsigned* та *signed* дуже подібний до синтаксису *std_logic_vector* і наведений нижче.

```
signal x : signed (7 downto 0);
signal y : unsigned (0 to 16);
```

Не потрібно плутати типи *integer* та *signed*. Обидва типи при обробці сприймаються як числа та обидва синтезуються в схему. Але слід пам'ятати, що *integer* – це число, *signed* – це вектор, тобто набір бітів. І як над набором бітів над сигналами типів *unsigned* та *signed* можна виконувати логічні операції та операції зсуву. При використанні типу *integer* слід обмежувати діапазон значень сигналу. При використанні типу *signed* це виконується при описі портів та сигналів.

Для перетворення типів даних використовують наступні функції:

*to_integer* () – для перетворення типів *unsigned* або *signed* в *integer*.

*to_unsigned ()* – перетворення невід'ємного числа типу *integer* у вектор типу *unsigned* заданого розміру.

*to_signed ()* – перетворення невід'ємного числа типу *integer* у вектор типу *signed* заданого розміру.

Шляхи перетворення типів схематично показані на рисунку 2.5. На цьому рисунку типи даних показані у прямокутниках, біля яких вказана назва бібліотеки, в якій описаний цей тип. Для перетворення типів необхідно використовувати функцію, яка наведена на дузі, що з'єднує типи. Ще раз відмітимо, що бібліотеку *std* не потрібно описувати в програмі – вона під'єднується до проекту автоматично.

Для прикладу розглянемо перетворення сигналу *Address* типу *std_logic_vector* в сигнал типу *integer*. В цьому випадку необхідно провести два послідовних перетворення: з типу *std_logic_vector* до типу unsigned за допомогою функції *unsigned ()* та типу *unsigned* до типу integer за допомогою функції *to_integer ()*. Також необхідно підключити до програми бібліотеки *std_logic_1164* та *numeric_std*. В цьому випадку конструкція перетворення типу буде мати такий вигляд:

```
to_integer(unsigned(Address))
```

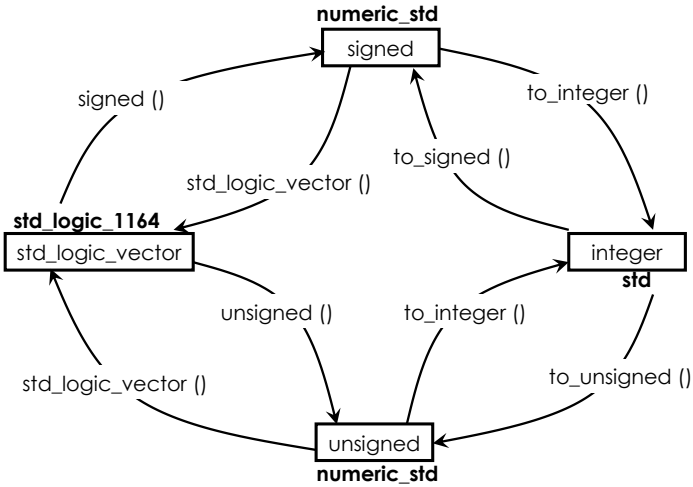



*Рисунок 2.5 – Перетворення типів у мові VHDL*

## 2.4.3. Підтипи

Підтип – підмножина значень даного типу. Об'єкти, віднесені до підтипу, зберігають сумісність із даними типу, з якого виділяється підтип так званого базового типу. Однак уведення підтипу:

- визначає множину припустимих значень даних підтипу як підмножину припустимих значень базового типу;
- дозволяє вводити додаткові функції перетворення, обумовлені тільки для даних підтипу.

Синтаксис декларації підтипу визначений у такий спосіб:

**subtype** *ім'я_підтипу* **is** *базовой_тип обмеження;*

Наприклад, визначимо підтип типу *integer*. Дані цього підтипу передбачається використовувати для індексації бітів в 5-розрядному коді, що може приймати 32 значення. Ці дані сумісні з даними типу *integer*. Однак присвоєння цим даним значень поза зазначеним діапазоном буде викликати повідомлення про помилку.

```
subtype bit_in is integer range 31 downto 0;
```

### 2.4.4. Атрибути типів.

Тип визначає множину значень і множину операцій, які можуть застосовуватись до нього. При визначенні типу одночасно визначається також множина атрибутів типу, які дозволяють одержати інформацію про діапазон значень, що входять у цей тип. Звертання до атрибутів має наступний синтаксис:

*ім'я_типу'атрибут*

Перелік деяких атрибутів скалярних типів наведено в таблиці 2.4. У ній T – ім'я типу або об'єкта даного типу.

*Таблиця 2.4 – Атрибути скалярних типів*

| Атрибут | Опис |
|---|---|
| T'left | Перше значення у типі (ліва границя інтервалу) |
| T'right | Останнє значення у типі (права границя інтервалу) |
| T'low | Найменше значення у типі |
| T'high | Найбільше значення у типі |

Розглянемо два типи даних: *index* та *index2*. У першому випадку тип описує значення від 20 до 0, а у другому – від 0 до 20. Для цих двох типів найменше та найбільше значення будуть однаковими, а от ліва та права границі типів – різними.

```
type index is range 20 downto 0;

index'left = 20;
index'right = 0;
index'low = 0;
index'high = 20;
```

```
type index2 is range 0 to 20;

index2'left = 0;
index2'right = 20;
index2'low = 0;
index2'high = 20;
```

## 2.4.5. Літерали.

Літерали – це лексичні елементи, які залишаються постійними в програмі та приймають певне значення після компіляції. Константи, операнди у виразах, ідентифікатори – всі вони є літералами.

Літерали діляться на (рисунок 2.6):

1. Числові.
    1.1. Абстрактні.
        1.1.1. Десяткові.
        1.1.2. Базові.
    1.2. Дійсні.
2. Перелічимі.
3. Строкові.
4. Рядок біт
5. Null.

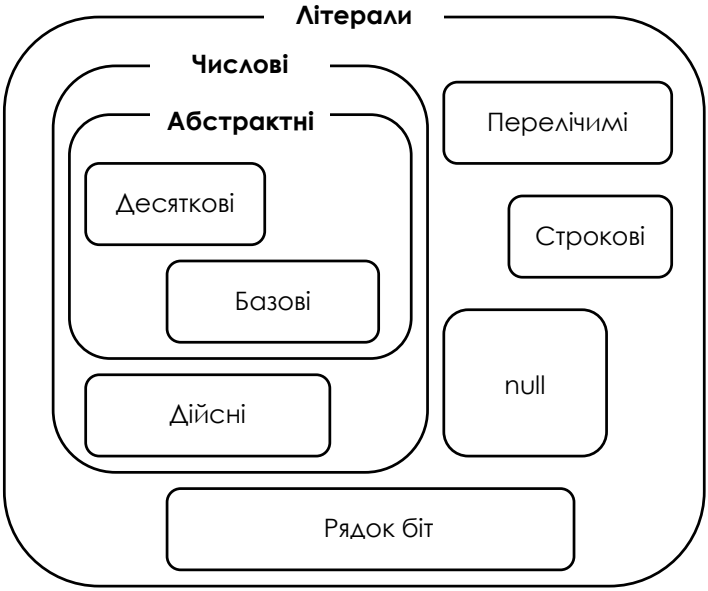



*Рисунок 2.6 – Літерали мови VHDL*

Десяткові літерали записуються у вигляді:
*ціле.ціле ступінь експонента*

Тут цілі числа задають мантису числа. При необхідності використовуєтьсья ступінь 10 для завдання показника ступеня. У числі можуть використовуватись знаки підкреслення (_) для поділу на групи.

Приклади:
Ціле: 21, 0, 1E2, 3e4, 452
Дійсне: 11.0, 0.0, 0.476, 3.14_15_926
Дійсне з експонентою: 1.34E-12, 1.0E+6,
Базові літерали записуються у вигляді:
*базис#число_в_базисі #експонента*
Базис (основа системи числення) – ціле число.

| | | |
|---|---|---|
| Двійкова | 2#1111_1100# | Всі ці літерали мають значення 252 |
| Шістнадцяткова | 16#fc# | |
| Шістнадцяткова | 016#0FC# | |
| Десяткова | 10#252# | |
| Сімкова | 7#510# | |

Літерали "строка біт":
*базис*”*число_в_базисі*”
Базис:
B – бінарный,
O – вісімковий,
X – шістнадцятковий.

```
 B”101010”, b”1001_100”
O”126”, o”236”
 X”34f”, x”FC05”
```

## 2.4.6. Масиви і записи

Масив, як і в інших мовах, – це набір даних, об'єднаних загальним іменем, доступ до їх значень відбувається за допомогою операції індексування, вжитої після імені масиву. Для того щоб

оголосити об'єкт типу масив, необхідно попередньо ввести відповідний тип на основі наступних синтаксичних правил:

**type** *ім'я_типу* **is array** (*діапазон_значень*) **of** *тип_елементів_масива*

Діапазон задає безліч допустимих значень індексу. При визначенні діапазону значень можлива зміна індексу від більшого до меншого (*downto*) або від меншого до більшого (*to*) індексу:

*початкове_значення* **to** *кінцеве_значення*
*початкове_значення* **downto** *кінцеве_значення*

Якщо діапазон заданий конструкцією **range<>**, то це є оголошенням необмеженого масиву. У цьому випадку визначається не діапазон значень індексу, а тільки тип індексної змінної. Але таке використання масиву не є рекомендованим, оскільки може призводити до невірних результатів при компіляції програми. Тому необхідно завжди визначати межі значень індексів масиву для коректного синтезу схеми.

Масиви найбільш часто використовуються при описі блоків пам'яті. Нижче наведений приклад декларації типу для опису блоку пам'яті, що складається з 32 чарунок, кожна з яких містить по 8 біт даних.

```
type ROM_Array is array (0 to 31) of
std_logic_vector(7 downto 0);
```

Декларації об'єктів, що належать наведеному типу, можуть виглядати таким чином:

```
signal ROM : ROM_Array;
```

або

```
constant Content: ROM_Array:= (
            0 => "00000001",
            1 => "00000010",
            2 => "00000011",
                . . .
            14 => "00001111",
        others => "11111111");
```

При звертанні до елементів масиву в програмі індекси містяться в дужках після імені масиву. Тип індексного виразу повинен

відповідати типу індексу, оголошеного при декларації типу масиву. Запис звертання до елемента масиву повинен мати наступний вигляд:

```
ім'я_масива(індекс_елемента)
ім'я_масива(індекс_елемента to
  індекс_елемента)
ім'я_масива(індекс_елемента downto
  індекс_елемента)
```

Приклад: звернення до конкретного елемента масиву:

```
memory (5)
byte (7)
```

Звернення до декількох елементів масиву:

```
memory (5 to 7)
byte (7 downto 5)
```

При звертанні до елемента багатомірного масиву індексні вирази записуються через коми в порядку, описаному в декларації типу.

Приклад:

```
ram_instance (2, 3) := 5;
```

Для одномірних масивів визначено кілька групових операцій, у яких масив розглядається як єдине ціле. Це, насамперед , операція конкатенації & (об'єднання рядків). Наприклад, наведена нижче послідовність операторів записує в сигнал *b* значення"11011001".

```
a:= "1001";
b<= "1101" & a;
```

Тут *a* і *b* – рядки або бітові вектори, причому *a* – змінна, *b* – сигнал. Різниця між сигналом та змінною описана в п. 2.4.

**Операції зсуву** визначені для одномірних масивів типу *bit_vector* і записуються в такий спосіб:

```
ім'я_масива символ_операції_зсуву ціле_число
```

Визначені наступні операції зсуву:

- логічні зсуви вліво та вправо *sll* і *srl*,
- арифметичні зсуви вліво та вправо *sla* і *sra*,

– циклічні зсуви вліво та вправо *rol* і *ror*.

Ціле в записі виразу для зсуву визначає число розрядів, на які здійснюється зсув коду.

Наприклад.

| | Операція | Результат |
|---|---|---|
| a = 00000011 | *b <= a sll 5* | b = 01100000 |
| a = 10110000 | *b <= a slr 5* | b = 00000101 |
| a = 00000011 | *b <= a sla 5* | b = 01111111 |
| a = 10110000 | *b <= a sla 5* | b = 11111101 |
| a = 00100011 | *b <= a rol 5* | b = 01100100 |
| a = 10010001 | *b <= a ror 5* | b = 10001100 |

**Запис** – це структура даних, кожна інформаційна одиниця якої має індивідуальне ім'я і може бути окремого типу. Зазвичай записи використовуються для агрегатування різних даних, що характеризують один об'єкт. Для використання записів, як змінних, спочатку треба оголосити відповідний тип:

```
record список_полів_запису: тип;
     список_полів_запису : тип
end record;
```

**Приклад.** Визначимо тип pixel, що представляє складові кольорів зображення крапки на екрані у форматі FULL COLOR (повна передача кольору), що передбачає восьмирозрядне представлення трьох колірних складових.

```
type pixel is
     record red,green,blue: integer range 0
     to 255;
end record;
```

Тоді тип "відеопам'ять" може бути визначений як

```
type video_ram is array(integer range
<>,integer range <>) of pixel;
```

Екземпляр відеопам'яті буде визначатися, наприклад, у такий спосіб:

```
signal vram : video_ram (639 downto 0, 479 downto 0);
```

Цей екземпляр може зберігати інформацію про зображення розміром 640 рядків по 480 елементів у рядку. Вибірка значення червоної складової верхнього лівого елемента зображення з такої пам'яті описується оператором

```
Out_red <= vram (0,0).red;
```

## 2.5. Константи, сигнали та змінні

Будь-який проект на мові VHDL є описом явищ у дискретних системах. Ці явища можуть представлятися трьома різними категоріями даних: константи, сигнали та змінні.

### Константи

Константи (**constant**) у VHDL, як і в будь-якій мові програмування, необхідні для визначення постійних значень. Після оголошення константи присвоювання їй нових значень заборонено.

Оголошення констант.

```
constant список_констант: тип := вираз;
```

```
constant period: time:= 100 ns;
constant PI:     real:= 3.14159;
constant WIDTH:  integer:= 32;
constant DEFAULT: bit_vector (0 to 3):= "0101";
```

### Сигнали

Сигнал (**signal**) – це інформація, що передається між модулями проекту або представляє вхідні та вихідні дані обладнання, яке описується проектом (рисунок 2.7). Він синтезується в провід, сигнальну лінію, значенням на якій притаманні властивості зміни в часі.

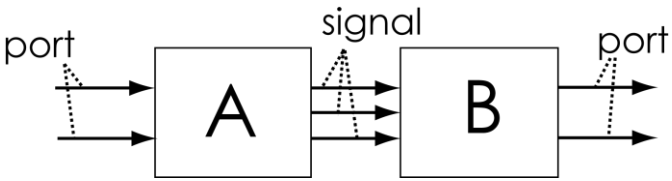



*Рисунок 2.7 – Порти і сигнали*

Загальна конструкція оголошення сигналу виглядає так:

```
signal ім'я: тип := початкове_значення;
```

Приклади оголошень сигналів:

```
 signal CLK, RESETn: bit;
signal COUNTER: integer range 0 to 31 := 25;
 signal INSTRUCTION: bit_vector (15 downto 0);
```

Для присвоювання сигналу значень використовується оператор паралельного присвоювання <=.

**Атрибути сигналів**

Атрибути сигналів визнаються так само як і атрибути типів:

```
ім'я_сигналу ' ім'я_атрибута
```

У таблиці 2.5 наведені деякі атрибути сигналів, які ми будемо використовувати в подальшому.

*Таблиця 2.5 – Атрибути сигналів*

| Ім'я атрибута | Тип атрибута | Значення |
|---|---|---|
| S'event | boolean | Сигналізує про зміну сигналу |
| S'stable | boolean | S'stable = **not** S'event |
| S'active | boolean | true, якщо присвоєння сигналу виконане, але значення ще не змінене (не кінчений часовий інтервал, заданий вираженням after) |
| S'quiet | boolean | S'active = **not** S'quiet |

### Змінні

Змінна (**variable**) – це допоміжна інформаційна одиниця, яка використовується для опису внутрішніх операцій у програмних блоках. Змінна може синтезуватись у чарунки пам'яті, що зберігають дані. Загальна конструкція оголошення змінної виглядає так:

```
variable ім'я: тип := початкове_значення;
```

Приклади:

```
variable ROM: integer range 0 to 15;
variable count: positive:= 100;
variable a, b, c, d: bit:= '1';
```

## 2.6. Оператори мови VHDL.

Оператори у мові VHDL можна розділити на дві великі групи – послідовні і паралельні.

**Послідовні оператори** (*Sequential Statement*) за характером виконання подібні до операторів традиційних мов програмування. Оператори цього типу обов'язково "вкладені" в оператор **process** або підпрограму **procedure** чи **function** і виконуються послідовно один за одним у порядку запису. Результати виконання послідовних операторів недоступні іншим програмним модулям принаймні до того, як буде виконаний оператор очікування **wait**, або не будуть виконані до кінця всі процеси, ініційовані загальною подією.

Нижче наведено повний список послідовних операторів мови:

- послідовне сигнальне присвоювання (*signal assignment*);
- присвоювання змінної (*variable assignment*);
- умовний оператор (*if*);
- оператор вибору (*case*);
- оператор повтору (*loop*);
- оператор переходу до нового циклу (*next*);
- оператор виходу із циклу (*exit*);
- оператор повернення (*return*);

- порожній оператор (*null*);
- оператор очікування (*wait*);
- виклик процедури (*procedure call*);
- оператор перевірки (*assertion*);
- оператор виведення повідомлень (*report*).

**Паралельні оператори** (*concurrent statement*) виконуються при будь-якій зміні сигналів, які використовуються у якості його вихідних даних. До паралельних операторів відносяться:

- оператор процесу (*process*);
- оператор паралельного присвоювання (*signal assignment*);
- паралельний виклик процедури (*procedure call*);
- паралельний оператор перевірки (*assertion*);
- оператор блоку (*block*);
- оператор входження компонента (*component instantiation*);
- оператор генерації (*generate*).

Виконання паралельних операторів ініціюється не по послідовному, а по подійному принципу, тобто вони виконуються тоді, коли реалізація інших операторів програми створила умови для їхнього виконання. Паралельні оператори представляють частини алгоритму, які в реальній системі можуть виконуватися одночасно. Ці частини взаємодіють між собою і з оточенням проектованої системи і є незалежно синтезованими окремими блоками.

***Оператори логічних та арифметичних операцій, операцій зсуву*** найбільш просто описують нескладні комбінаційні пристрої. Перелік таких операторів наведений в таблиці 2.6. Для опису більш складних комбінаційних і послідовністних схем зручніше використовувати оператор процесу (*process*).

*Таблиця 2.6 – Логічні, арифметичні оператори та оператори зсуву*

| Тип оператору | Оператори | Тип даних |
|---|---|---|
| Логічні | *not, and, nand, or, nor, xor, xnor* | *bit, bit_vector, std_ulogic, std_logic, std_logic_vector, std_ulogic_vector, unsigned, signed* |
| Арифметичні | +, -, *, /, *mod, rem, abs* | *integer, signed, unsigned, natural* |
| Порівняння | =, /=, <, >, <=, >= | *integer, signed, unsigned, natural* |
| Зсуву | *sll, srl, sla, sra, rol, ror* | *bit_vector, signed, unsigned* |
| | *shift_left, shift_right, rotate_left, rotate_right* | *signed, unsigned* |
| Конкатенації | *&*, (, , ,) | Так само як і для логічних операторів |

## 2.6.1. Оператори присвоювання

Для присвоювання значень сигналам, змінним та константам можуть використовуватись декілька різновидів оператора присвоювання:

- паралельне присвоювання (<=), яке використовується для присвоювання значень сигналам (**`signal`**);
- послідовне присвоювання (:=), яке використовується для присвоювання значень змінним (**`variable`**), константам (**`constant`**), визначення початкових значень та у операторі генерації (**`generic`**).

По синтаксису і правилах виконання безумовне паралельне присвоювання збігається з послідовним присвоюванням сигналу.

Варіанти різняться по локалізації в програмі і характеризуються різними умовами виконання. Найбільш істотні відмінності:

- паралельне присвоювання локалізується в загальному розділі архітектурного тіла, а послідовне – тільки в тілі процесу;
- послідовне присвоювання сигналу виконується після того, як ініційовано виконання процесу і виконані всі попередні оператори в тілі процесу;
- оператор паралельного присвоювання виконується відразу (з погляду модельного часу) після зміни сигналів у правій частині цього оператора.

**Послідовне присвоювання.**

Оператор послідовного присвоювання має такий вигляд := і використовується для присвоювання значень змінним. Цей оператор може використовуватись лише всередині процесів, а також підпрограм (процедур і функцій). Тобто лише з послідовними операторами.

**Паралельне присвоювання**

Паралельне присвоювання може мати декілька варіантів:

- безумовне паралельне присвоювання,
- умовне присвоювання,
- присвоювання за вибором.

При використанні будь-якого з варіантів присвоювання можна використовувати мітки для більш зручного читання та аналізу коду.

**Безумовне паралельне присвоювання** записується за допомогою оператора <= і вже зустрічалося раніше. В якості прикладу розглянемо запис рівнянь мультиплексора з чотирма входами *x0, x1, x2, x3* та одним виходом *y*. Для керування використовується два сигнали *s0, s1* (рисунок 2.8). Вихідний сигнал формується як результат мультиплексного рівняння від входів даних та керуючих входів. Результат моделювання мультиплексора показаний на рисунку 2.9.

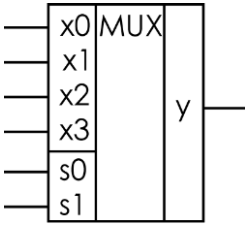



*Рисунок 2.8 - Мультиплексор*

**Приклад 2.2. Мультиплексор 4-в-1.**

```
library ieee;
use ieee.std_logic_1164.all;

entity mux is
   port(x0, x1, x2, x3: in std_logic;
      s0, s1: in std_logic;
      y       : out std_logic);
end mux;

architecture pure_logic of mux is
begin
   y <= (x0 and not s1 and not s0) or
        (x1 and not s1 and s0) or
        (x2 and s1 and not s0) or
        (x3 and s1 and s0);
end pure_logic;
```

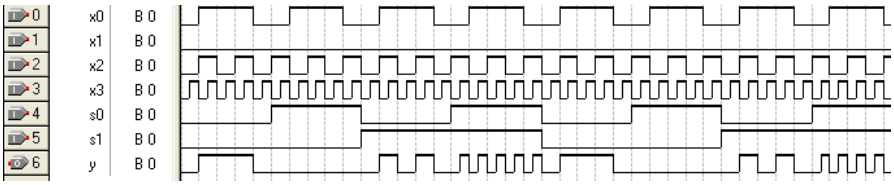



*Рисунок 2.9 – Часові діаграми роботи мультиплексора*

**Умовне присвоювання** має наступний синтаксис:

```
приймач <=
значення when умова else
значення when умова else
...;
```

**Приклад 2.3.** Мультиплексор із третім станом на виході може бути представлений наступним оператором:

```
Z_out <= x0 when (adr='0' and en='1') else
   x1 when (adr='1' and en='1') else 'Z';
```

Слід пам'ятати, що сигнал *Z_out* повинен бути типу *std_logic*, оскільки він приймає значення **'Z'**.

**Присвоювання по вибору**

```
with ключовий_вираз select
приймач <= значення when умова,
   значення when умова,
...;
```

**Приклад 2.4. Мультиплексор з третім станом на виході.**

Мультиплексор з третім станом представлений вище. У наступному прикладі використовується оператор *&*, який працює зі змінними типу *bit*. Результати моделювання показані на рисунку 2.10.

```
library ieee;
use ieee.std_logic_1164.all;

entity mux_z is
port (x0,x1: in std_logic;
   adr, en : in bit;
   Z_out: out std_logic);
end mux_z;

architecture concurrent of mux_z is
begin
with adr & en select
Z_out<= x0 when "01",
   x1 when "11",
```

```
    'Z' when others;
end concurrent;
```

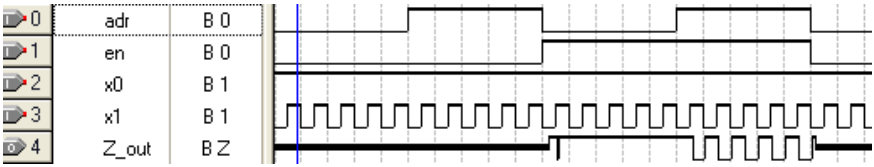



*Рисунок 2.10 – Часові діаграми роботи мультиплексора*

## 2.6.2. Послідовні оператори

### Умовний оператор і оператор вибору

Ці оператори дозволяють описувати сукупності дій, деякі з яких виконуються при виникненні певних умов у реальному обладнанні і при моделюванні, а інші за тих самих умов не виконуються.

**Умовний оператор `if`.**

```
if булевий_вираз then
    операторі;
     elsif булевий_вираз then
          операторі;
          else
               операторі;
end if;
```

Умова повинна мати булевий вираз, який після виконання дає результат типу *boolean*. В одному операторі **`if`** може бути одна (або жодної) або більше частин **`elsif`**. Ключове слово **`elsif`** слід відрізняти від слів **`else`**. Частина **`else`** також може бути тільки одна або не буде жодної. Після оператора **`elsif`** записується нова умова, при виконанні якої реалізується частина **`else`**.

**Приклад 2.5.** Спрощена логіка роботи RS-тригера.

```
if s='1' then q<='1';
     elsif r='1' then q<='0';
end if;
```

Вихідний сигнал *q* дорівнює 1 у випадку, коли вхід *s* дорівнює 1. Значення входу *r* у цьому випадку не має значення. Якщо ж *s* = 0 та *r* = 1 вихідний сигнал скидається в нуль.

**Приклад 2.6.** Лічильник з входами скидання, завантаження та дозволу рахування. У наведеному нижче прикладі умовні оператори виконуються один за одним і чим більше вкладення оператора, тим більш складним виявляється булевий вираз, який розраховується при аналізі.

```
1  if clrn = '0' then csignal <= 0;
2  elsif (clk'event and clk = '1') then
3   if load = '1' then csignal <= d;
   else
      if ena = '1' then csignal <= csignal +
1;
4        else    csignal <= csignal;
5     end if;
6   end if;
7  end if;
```

Програма складається з чотирьох вкладених операторів умови.

В першому операторі перевіряються асинхронні сигнали – тобто ті, які аналізуються без урахування тактового сигналу *clk*. До таких сигналів належить сигнал скидання *clrn*. Якщо цей сигнал дорівнює нулю, то у вихідний сигнал *csignal* записується нуль (рядок 1 прикладу) і тримається до тих пір, поки не буде знятий сигнал *clrn*. При цьому інші сигнали не можуть впливати на стан вихідного сигналу (рисунок 2.11).

В другому операторі **if** виконується перевірка синхронних сигналів, тобто тих, робота яких прив'язана до тактового сигналу *clk*. У наведеному прикладі це сигнали *load* і *ena*. Значення цих сигналів перевіряється при надходженні фронту сигналу *clk*. Всі оператори, що

описані всередині будуть виконуватись лише тоді, коли умова приходу переднього фронту сигналу *clk* буде виконана.

Перевірка стану тактового сигналу *clk* відбувається у 2 рядку прикладу. Конструкція *clk'event and clk = '1'* є логічним добутком двох логічних виразів. Перший *clk'event* дозволяє перевірити наявність будь-якої події з сигналом *clk* (див. атрибути сигналів і таблицю 2.5). Другий вираз дозволяє перевірити чи дорівнює рівень сигнала одиниці. В результаті отримуємо перевірку наявності події з сигналом *clk*, внаслідок якої сигнал прийняв значення одиниці. Така конструкція може бути заміненою на вираз *rising_edge (сигнал)* з бібліотеки *std_logic_1164*. Сигнал в дужках має бути типу *std_logic*.

Для визначення заднього фронту сигналу можна використати або конструкцію *clk'event and clk = '0'* або функцію *falling_edge (сигнал)* з бібліотеки *std_logic_1164*.

Третій рядок перевіряє наявність на вході сигналу синхронного завантаження *load*. Якщо цей сигнал дорівнює одиниці, то дані з входу *d* переписуються на вихід.

Якщо завантаження нема і синхронний сигнал дозволу роботи *ena* дорівнює одиниці, то виконується рахування (рядок 5). У випадку, коли сигнал дозволу роботи дорівнює нулю значення лічильника не змінюється (рядок 6).

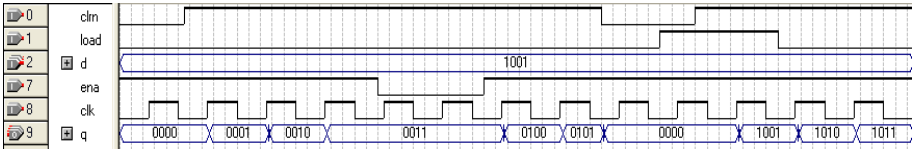



*Рисунок 2.11 – Часові діаграми роботи лічильника*

**Оператор вибору `case`.**

Цей оператор обирає один з варіантів, який визначається значенням ключового виразу. Вираз повинен бути дискретного типу або типу однорозмірного масиву символів, значення яких можуть бути представлені як рядки або рядок бітів. Варіант значення повинен бути такого ж типу, як і ключовий вираз. Усі можливі значення варіанту повинні бути перебрані. При визначенні значень варіанту можливий запис як одиночного значення так і діапазону значень. Також можливе

використання слова *others* – тобто всі інші варіанти крім тих, що були перераховані. Для випадку *others* повинне бути таке значення, яке не відповідає вже описаним вище.

```
case ключовий_вираз is
     when варіант => оператор;
     when варіант => оператор;
      …
   when others => оператор;
end case;
```

**Приклад 2.7.** Дешифратор семисегментного індикатора.

```
process (code)
   begin
      case code is
            when x"0" => led <= b"1111110";
            when x"1" => led <= b"0110000";
            when x"2" => led <= b"1101101";
            when x"3" => led <= b"1111001";
            when x"4" => led <= b"0110011";
            when x"5" => led <= b"1011011";
            when x"6" => led <= b"1011111";
            when x"7" => led <= b"1110000";
            when x"8" => led <= b"1111111";
            when x"9" => led <= b"1111011";
            when x"a" => led <= b"1110111";
            when x"b" => led <= b"1110111";
            when x"c" => led <= b"1001110";
            when x"d" => led <= b"0111101";
            when x"e" => led <= b"1001111";
            when x"f" => led <= b"1000111";
      end case;
end process;
```

**Оператор циклу `loop`**

Оператор циклу **`loop`** використовується у тому випадку, коли необхідно повторювати деяку операцію декілька разів. Якщо оператор циклу використовується сам, то буде виконуватись нескінченний цикл. Виконання може бути перерване тільки спеціальним керуючим

оператором. Цикли **loop** можуть бути вкладеними. При використанні оператора циклу необхідно використовувати мітку циклу, яка дозволяє однозначно ідентифікувати цикли.

В операторі циклу може використовувати дві ітераційні схеми: **while** і **for**. Синтаксис операторів циклу **while** і **for** показаний нижче.

```
 мітка_цикла: while умова loop
       оператор
       оператор
 end loop мітка_цикла;

мітка_цикла: for ітератор_цикла loop
       оператор
       оператор
 end loop мітка_цикла;
```

Коли в записі циклу використовується ключове слово **while**, то спочатку обчислюється умова входження до тіла циклу. Якщо умова виконується, то виконуються всі оператори в тілі циклу, інакше оператор циклу завершується і обчислення переходить на наступний оператор. Коли в записі циклу використовується ключове слово **for**, то ітератор визначає параметр циклу з базовим дискретним типом, значення якого визначаються діапазоном. Ітератор циклу є константою по відношенню до всіх дій в тілі оператора циклу і йому не може присвоюватись значення.

**Приклад 2.8.** Присвоювання вихідному вектору *data_out* розрядністю 4 біта молодших чотирьох бітів вхідного сигнала *s* розрядністю 8 біт за допомогою циклу **loop** та ключового слова **for**.

```
 clk: in std_logic;
s: in std_logic_vector (7 downto 0);
data_out: out std_logic_vector (3 downto 0);
...
process (clk)
    variable i: integer range 0 to 3;
begin
    if rising_edge (clk) then
```

```
      for i in 0 to 3 loop
            data_out(i) <= s(i);
      end loop;
   end if;
end process;
```

**Оператор переходу до нового циклу next**

Оператор **next** вживається для передчасного завершення ітерацій циклу, що задовольняють деяким умовам. При цьому блокується виконання всіх наступних операторів у поточній ітерації циклу. Виконання продовжується з початку циклу, але параметр циклу збільшується на одиницю. Якщо ж параметр циклу досяг свого найбільшого значення, то виконання циклу завершується.

**Приклад 2.9.** В процесі використовується оператор циклу, в якому виконується логічна операція «І» над бітами вхідних сигналів *a* і *b*. Для перевірки необхідності виконання цієї операції контролюється стан поточного біта з масиву *done*. Якщо він дорівнює *true*, то логічний добуток не розраховується і виконання циклу переходить до нової ітерації. Якщо ж біт масиву *done* дорівнює *false*, то його значення змінюється на *true* і виконується обчислення.

```
a, b: in std_logic_vector (7 downto 0);
data_out: out std_logic_vector (7 downto 0);
...
process (a, b)
begin
   for i in 0 to 7 loop
      if (done (i) = true) then
            next;
      else
            done (i):= true;
      end if;
      data_out (i) <= a(i) and b(i);
   end loop;
end process;
```

**Оператор виходу з циклу `exit`**

```
мітка: exit мітка_цикла when умова;
```

Оператор **`exit`** використовується для завершення виконання і закриття оператора циклу. Якщо умова (*condition*) виконується (*true*), то здійснюється вихід із циклу. При використанні вкладених циклів мітка циклу в операторі **`exit`** дозволяє виконати вихід з вказаного циклу. Якщо ж мітка не вказується, то виконується вихід з поточного циклу.

**Приклад 2.10.** Вкладені цикли.

```
outer: loop
    inner: loop
           exit   outer   when   condition_1
     інструкції;
           exit when condition_2 інструкції;
    end loop inner;
    exit when condition_3 інструкції;
end loop outer;
```

В наведеному прикладі використовується два нескінченні цикли: зовнішній *outer* і внутрішній *inner*. При виконанні першої умови (*condition_1*) виконується вихід з внутрішнього і зовнішнього циклів. При виконанні другої умови (*condition_2*) виконується вихід з внутрішнього циклу, а при виконанні третьої умови (*condition_3*) – вихід з зовнішнього циклу.

**Порожній оператор `null`**

Оператор **`null`** не породжує дій. Він вживається, щоб точно специфікувати, що нема ніяких дій. Типове застосування – в операторі **`case`**, щоб визначити дії у всіх випадках. У наведеному нижче прикладі сигнал *OPcode* може приймати будь-яке значення від 000 до 111. В операторі **`case`** аналізуються лише три стани: 001, 010 та 100. Всі інші комбінації (**`others`**) не повинні викликати жодної дії, тому використовується оператор **`null`**.

```
case OPcode is
```

```
    when "001"  => TmpData := RegA and RegB;
    when "010"  => TmpData := RegA or RegB;
    when "100"  => TmpData := not RegA;
    when others => null;
end case;
```

## 2.6.3. Паралельні оператори

Паралельні оператори можуть бути простими та складеними. Складений оператор включає в себе декілька простих операторів, для яких визначені загальні умови ініціалізації. Така сукупність операторів називається тілом складеного оператора.

Складені паралельні оператори це – оператор процесу (**process**) та оператор блоку (**block**).

### Оператор process

Оператор процесу (**process**) – це складений оператор паралельного типу. Функціонування процесів всередині архітектури відбувається паралельно, а всередині самих процесів оператори функціонують послідовно. Синтаксис оператора визначений у такий спосіб:

```
мітка_процеса:
    process (список ініціалізаторів)
    розділ декларацій
    begin
    розділ операторів
end process мітка_процеса;
```

Мітка оператора процесу може бути відсутньою, але її використання значно полегшує структурованість програми. Розділ декларацій визначає локальні об'єкти, що використовуються в наступному за ним програмному блоці, у цьому випадку – в розділі операторів. У розділі операторів використовуються послідовні оператори.

Оператор процесу починає виконуватись при зміні сигналів, що входять у список чутливості, який містить ініціалізатори (при відсутності такого списку – безумовно після виконання всіх вкладених

операторів), а результати його виконання доступні іншим паралельним операторам тільки після виконання всіх операторів, ініційованих тими ж подіями, у тому числі процесів.

Під час моделювання оператори, вкладені в оператор **process**, будуть виконуватися один за одним після виникнення в системі події, яка ініціалізує процес. Паралельні оператори в тілі процесу не визначені.

Деякі паралельні і послідовні оператори збігаються за формою запису. У цьому випадку відмінність установлюється в контексті: якщо оператор локалізований у тілі процесу, він трактується як послідовний, а якщо в іншому місці програми – як паралельний.

В операторі процесу можливе використання як сигналів так і змінних. Змінні можуть бути визначені тільки в тілі процесу, а сигнали у всьому архітектурному тілі. Найбільш явно різниця між сигналами та змінними виявляється при інтерпретації операторів послідовних присвоєнь. Для обох видів зберігається загальне для послідовних операторів правило початку виконання: перший оператор у процесі виконується після виконання умов ініціалізації процесу, а кожний наступний – відразу після виконання попереднього. Однак результат присвоєння змінної безпосередньо доступний будь-якому наступному операторові в тілі процесу. Трактування оператора послідовного присвоєння сигналу суттєво відрізняється від трактування присвоєння змінної або операторів присвоювання в традиційних мовах програмування. Присвоєння сигналу не приводить безпосередньо до зміни його значення. Нове значення спочатку заноситься в буфер, який називається *драйвером сигналу*, а сам сигнал залишається незмінним до закінчення виконання оператора процесу. Фактична зміна значення сигналу виконується тільки після виконання до кінця процесу і інших паралельних операторів, ініційованих загальною подією, або після виконання оператора зупинки **wait**.

Розглянемо приклади, що ілюструють різницю між операторами та змінними.

```
signal clk: bit;
signal cS, b: integer;
constant a,d:
integer:=10;
```

```
signal clk:bit;
signal b: integer;
constant a,d:
integer:=10;
```

```
...
process (clk)
        begin
           cS<= a;
           b<=cS+d;
--      cS<=      b;      --
неприпустимо
end process;
```

```
process (clk)
        variable cV;
begin
        cV:= a;
        b<=cV+d;
        cV:=b;
end process;
```

Якщо перед виконанням цих процесів (після зміни сигналу *clk*) сигнал *cs* і змінна *cv* мали значення 4, а *b* = 3, то після реалізації процесу в лівому стовпчику *cs* і *b* приймуть значення 14, тому що присвоєння значення сигналу *cs* у першому операторові тіла не впливає на наступні оператори.

У другому випадку при тих же вихідних даних після завершення процесу одержимо *b* = 20 і *cv* = 3.

## Оператор очікування `wait`

Існує кілька модифікацій оператора **`wait`**:

```
wait;
wait on ім'я_сигнала, ім'я_сигнала;
wait until умова;
```

Оператор очікування призводить до тимчасового припинення оператора процесу або процедури.

Варіант оператора **`wait`** без додаткових уточнюючих конструкцій відповідає "нескінченній зупинці". У цьому випадку після досягнення такого оператора процес ніколи більше не буде виконуватися. Зазначену версію можна використовувати для опису процедур ініціалізації систем, а також фрагментів, робота яких за деяких умов припиняється назавжди. Звичайно такий оператор завершує програми генерації тестів, означаючи закінчення тестової послідовності.

Список сигналів у варіанті **`wait on`** еквівалентний списку ініціалізаторів процесу: виконання буде продовжено після того, як один із сигналів списку змінить своє значення.

```
wait on clk;
```

Очікування, задане конструкцією **wait until**, закінчується, коли виконана задана оператором умова, тобто відповідний вираз приймає значення *true*.

```
wait until clk = '1';
```

**Приклад 2.11.** Використання оператора **wait**. Наведений процес буде запускатись відразу після запуску і перший ж оператор зупинить його роботу. Продовження виконання програми відбудеться тільки після зміни одного із сигналів у списку ініціалізації оператора **wait**.

```
C_1: process
begin
   wait on a,b;
   if a = '1' and b = '1' then
   q <= '1'; p<='1';
   elsif a = '0' and b = '0' then
   q <= '0'; p<='1';
   else q <= '0'; p<= '1';
   end if;
end process C_1;
```

**Приклад 2.12.** У даному модулі три оператори **wait** – тому призупинення виконання процесу буде відбуватися тричі. У першому випадку відбувається очікування моменту, коли сигнали *a* і *b* будуть рівні 1, у другому випадку – 0, а в третьому –зупинка буде тривати до зміни одного із сигналів.

```
C_2 : process
begin
   wait until a = '1' and b= '1';
   q <= '1'; p <= '1';
   wait until a = '0' and b = '0';
   q <= '0'; p<= '1';
   wait on a, b;
   q <= '0'; p <= '1';
end process C_2;
```

Загальні правила інтерпретації оператора **process** можна звести до наступних:

1. Входження до тіла процесу при зміні будь-якого сигналу, перерахованого в списку ініціалізаторів процесу. Якщо список ініціалізаторів порожній, то процес безумовно виконується при початковому запуску, а також відразу за виконанням останнього оператора в розділі операторів цього процесу. При цьому треба мати на увазі, що оператор процесу без списку ініціалізаторів обов'язково повинен містити у своєму тілі оператор очікування **wait**. Інакше виконання будь-яких інших операторів у програмі блокується безкінечним виконанням процесу.

2. Усі оператори, що знаходяться всередині оператора процесу, виконуються послідовно один за одним від початку до кінця, за винятком випадків призупинення виконання дій оператором **wait**. Тоді після призупинення може бути ініційоване виконання інших процесів і паралельних операторів, а реалізація операторів, що знаходяться за оператором **wait**, продовжиться після настання події, оголошеного в цьому операторі.

3. Одночасне використання п. 1 і 2 неприпустимо.

Проілюструємо ці правила за допомогою двох операторів процесу. Перший містить сигнал *signal_1* в списку ініціалізації, а другий – оператор **wait** і список ініціалізації у цьому випадку відсутній.

```
process (signal_1)
      statement_1;
      statement_2;
      …
end process;
```

```
process
      statement_1;
      statement_2;
      …
wait on signal_1;
end process;
```

Форма запису оператора процесу наведена нижче, неприпустима оскільки містить і список ініціалізації і оператор **wait**:

```
process (signal_1)
      statement_1;
      statement_2;
      …
      wait on signal_2;
```

```
end process;
```

Необхідно також урахувати, що записи, наведені нижче, не є еквівалентними, тобто очікування (**wait**) на початку процесу не еквівалентно очікуванню наприкінці процесу.

```
process                          process
        statement_1;             wait on signal_1;
        statement_2;                    statement_1;
        …                               statement_2;
        wait          on                …
        signal_1;                 end process;
 end process;
```

Розглянемо яким чином впливає наявність сигналу в списку ініціалізації на його аналіз в операторі процесу. В якості прикладу будемо використовувати D-тригер, наведений на рисунку 2.12. Це примітив DFF-тригера, який використовується пакетом Quartus II. Таблиця дійсності тригера наведена у таблиці 2.7. Результати моделювання – на рисунку 2.13.

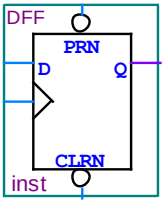



*Рисунок 2.12 – Тригер DFF*

*Таблиця 2.7 – Таблиця дійсності тригера DFF*

| PRN | CLRN | CLK | D | Q |
|---|---|---|---|---|
| L | H | X | X | H |
| H | L | X | X | L |
| L | L | X | X | L |
| H | H | ⌐ | L | L |
| H | H | ⌐ | H | H |
| H | H | L | X | $Q_0$ |
| H | H | H | X | $Q_0$ |

Як видно з таблиці, сигнали *prn* та *clrn* є асинхронними і не залежать від тактового сигналу *clk*. Сигнал даних *d* – синхронний бо його стан перевіряється при наявності фронту тактового сигналу. В список ініціалізації процесу, який буде описувати DFF-тригер, включимо асинхронні сигнали *prn* та *clrn* і тактовий сигнал *clk*. Сигнал *d* до списку ініціалізації включати не будемо.

**Приклад 2.13.** DFF-тригер.

```
library ieee;
use ieee.std_logic_1164.all;

entity d_ff is
port
    (clk    : in std_logic;
    prn, clrn: in std_logic;
    d  : in std_logic;
    q  : out std_logic);
end d_ff;

architecture rtl of d_ff is
begin
    process (clk, prn, clrn)
    begin
       if prn = '0' then q <= '1';
            elsif clrn = '0' then q <= '0';
            elsif clk'event and clk = '1'
                 then q <= d;
       end if;
    end process;
end rtl;
```

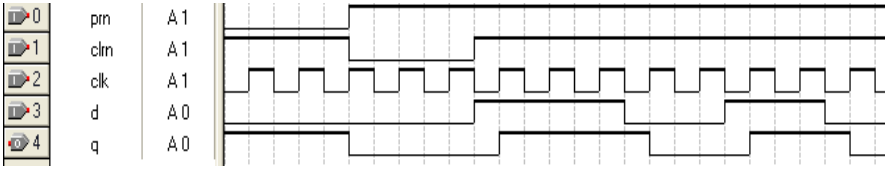



*Рисунок 2.13 – Часові діаграми роботи DFF тригера*

Як видно з програми, асинхронні сигнали повинні не тільки включатися в список ініціалізації, але й аналізуватися в процесі незалежно від тактового сигналу. Синхронний сигнал *d* аналізується в операторі **if** після перевірки наявності фронту тактового сигналу *clk*. Більш докладно про використанні та опис синхронних та асинхронних сигналів можна подивитись у статтях [5].

## Оператор блоку.

Оператор блоку (**block**) це паралельний складений оператор, який використовується для ієрархічної структуризації програми на мові VHDL, управління межами дії змінних, процедур та функцій. Існує два типи запису оператору блоку – спрощений та запис з охоронним виразом. Синтаксис оператора блоку виглядає так:

```
мітка: block (охоронний_вираз)
   розділ декларацій блока
begin
   розділ операторів блока
end block мітка;
```

Оператори тіла блоку, як і інші паралельні оператори, забезпечують можливість використання паралелізму в модельованій системі. Ці оператори ініціюються не по послідовному, а по подійному принципу, а результати їх виконання стають доступними іншим операторам як включеним у блок, так і розміщеним в інших блоках або "індивідуально", тільки після виконання всіх операторів, ініційованих однією подією.

У цьому сенсі оператори, включені в блок, не відрізняються від "індивідуальних" паралельних операторів.

Об'єднання операторів у блоки забезпечує наступні можливості:

- структуризація тексту опису, тобто можливість явного й наочного виділення сукупності операторів, що описують закінчений функціональний вузол;
- можливість оголошення в блоці локальних типів, сигналів, підпрограм і деяких інших локальних об'єктів;
- можливість присвоювання всім або деяким операторам блоку загальних умов ініціалізації.

Найбільш специфічними аспектами блокової організації є поняття охоронного вираження та оператора присвоювання, що охороняється.

Охоронний вираз – це будь-який вираз логічного типу, аргументами якого є сигнали. Будь-яка зміна сигналів, що входять в охоронний вираз, викликає обчислення значення цього виразу і присвоєння отриманого значення змінній *guard*. Область дії змінної *guard* – усе тіло блоку, і вона може використовуватися як звичайна логічна змінна у вкладених операторах блоку. Оператор присвоювання, який охороняється, використовує значення змінної *guard* без явної вказівки умови в програмі. Якщо *guard* = '0', то виконання операторів присвоювання, що містять ключове слово *guarded*, у такому блоці заборонене.

**Приклад 2.14.** Однорозрядний суматор.

```
entity add1_e is
port (b1, b2, enable : in bit;
    c1, s1 : out bit);
end add1_e;

architecture struct of add1_e is
begin
    p0: block (enable = '1')
            begin
            s1<= guarded (b1 xor b2);
            c1<= guarded (b1 and b2);
       end block p0;
end struct;
```

Охоронним виразом блоку є вираз *enable = ‘1’*. Якщо цей вираз дорівнює значенню *true* (істина), то конструкції присвоювання сигналів, що охороняються, виконуються, тобто однорозрядний суматор складає числа, якщо ж значення виразу дорівнює *false* (хибно), то присвоювання сигналів, що охороняється, не виконуються, тобто суматор не складає числа *b1, b2*. Присвоювання сигналів, що охороняється, виконується з використанням ключового слова **guarded**.

**Приклад 2.15.** D-тригер з асинхронним скиданням у вигляді блоку з охоронним виразом (*clk = ‘1’ or clr = ‘1’*).

```
D_LATCH: block (clk ='1' or clr = '1')
begin
       Q <= guarded '0' when clr = '1';
              else D when clk = '1';
              else Q;
 end block D_LATCH;
```

У даному прикладі *clk* – вхід синхронізації, *clr* – асинхронне скидання, *D* – вхід даних, *Q* – вихід тригера. Коли охоронний вираз приймає значення «хибно», то сигнал *Q* у лівій частині зберігає своє колишнє значення. Сигнал асинхронного скидання *clr* має пріоритет стосовно сигналу *clk*.

Підсумовуючи зазначене вище можна порівняти оператори **block** та **process**. Оператори блоку та процесу – це складені паралельні оператори, але в тілі оператора процесу знаходяться послідовні оператори, а тілі оператора блоку – паралельні. Запуск оператора процесу відбувається при зміні сигналів у списку ініціалізації, а оператора блоку – при зміні охоронного виразу.

## 2.6.4. Оператор генерації

Даний оператор використовується для генерації схем, що мають регулярну, повторювану структуру. Наприклад, суматорів та регістрів. Синтаксис оператора генерації:

```
мітка: for параметр_генерації generate
   паралельні оператори
```

```
end generate;
```

або

```
мітка: if умова_генерації generate
   паралельні оператори
end generate;
```

**Приклад 2.16.** Опишемо створення регістру на основі D-тригера. В цьому прикладі для генерації використовується компонент D_FF, який використовується чотири рази для створення чотирьохрозрядного паралельного регістра. Використання оператора компонента описане нижче.

```
entity reg is
port (clk : in bit;
      d: in bit_vector (3 downto 0);
      Q: out bit_vector (3 downto 0));
end entity reg;
architecture STRUCT of reg is
component D_FF
  port (D, clk : in bit; Q: out bit);
end component D_FF;
begin
  Gen: for i in  0 to 3 generate
        D_FF_i  :  D_FF  port  map  (d(i),  clk,
q(i));
    end generate;
end STRUCT;

entity D_FF is
port (D,clk : in bit;
      Q : out bit);
end entity D_FF;
architecture F of D_FF is
begin
    process(clk)
    begin
      if clk = '1' and clk'Event
        then Q <= D;
```

```
      end if;
    end process;
end architecture F;
```

## 2.6.5. Компонент

Поняття компонента вводиться для структурної організації проекту. Під компонентом розуміється окрема підсхема даного проекту, що описується в VHDL як **entity** і має своє ім'я. Підсхема має свої порти, імена яких можуть збігатися або не збігатися з іменами портів або сигналів у проекті, в якому вживається схема.

Та сама підсхема може входити в проект кілька разів, тому для різних екземплярів компонента використовують різні імена. Так як схема може мати різні зв'язки, то для визначення зв'язків використовується оператор входження компонента.

Інтерфейс компонент декларується в такий спосіб:

```
component ім'я_компонента
   generic (список_параметрів);
   port (список_портів);
end component ім'я_компонента;
```

Визначення компонентів може відбуватися в декларативній частині програми – разом з описом типів, підтипів, а може знаходитись в окремому файлі або бібліотеці.

Опис компонентів дає так би мовити структурну або ієрархічну структуру проекту. Для використання компонента необхідно використовувати оператор входження компонента:

```
мітка: ім'я_компонента
      generic (список_параметрів)
      port map (список_портів);
```

Мітка компонента є обов'язковою, тому що вона є унікальним ім'ям даного екземпляра компонента. Ім'я компонента – це фактично ім'я типу, а мітка – це ім'я об'єкта певного типу.

Для завдання портів при створенні компонента використовують:

- ключову відповідність із використанням оператора "=>";
- позиційну відповідність.

Для використання екземплярів компонента з будь-яким варіантом відповідності використовується конструкція **port map.**

При ключовій відповідності до портів компонента під'єднуються сигнали з використанням конструкції «*порт* => *сигнал*». Порт є внутрішнім об'єктом для компонента, а сигнал – зовнішнім об'єктом для компонента. Порядок перерахування портів в цьому випадку не відіграє ніякого значення.

Позиційна відповідність вимагає щоб порядок перерахування сигналів співпадав з порядком портів при оголошенні компонента.

**Приклад 2.17.** Розглянемо схему, показану на рисунку 2.14. Тут знаходяться три компоненти: два екземпляри компонента A і один екземпляр компонента B.

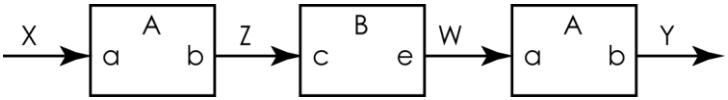



*Рисунок 2.14 – Схема для прикладу відповідності портів компонентів*

Опишемо компоненти.

```
 component a
port(a : in bit;
     b : out bit);
 end component;

 component b
port(c : in bit;
     e : out bit);
 end component;
```

Тепер опишемо екземпляри компонентів з використанням ключової відповідності.

```
 P1: A port map (a=>X, b=>Z);
P2: B port map (e=>W, c=>Z);
 P3: A port map (a=>W, b=>Y);
```

Тепер позиційна відповідність без явного значення ключа:

```
 P1: A port map (X, Z);
```

```
P2: B port map (Z, W);
 P3: A port map (W, Y);
```

**Приклад 2.18.** Компонент *AND4* реалізує функцію *I,* компонент *OR3 – АБО* (рисунок 2.15*)*. Компонент *AND4* має два екземпляри: DD1, DD2. Компонент *OR3* – один (DD3).

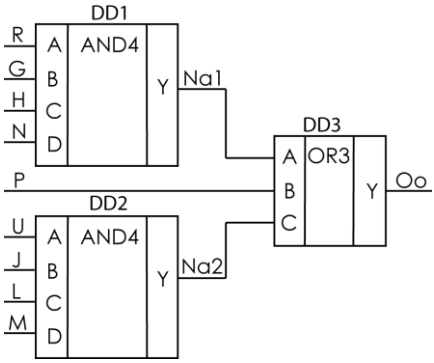



*Рисунок 2.15 – Схема для прикладу відповідності портів компонентів*

Так як оператор входження компонента є паралельним, то не має значення порядок опису компонентів. Структура програми може бути представлена так:

```
entity для_всієї_схеми
port для_всієї_схеми

architecture для_всієї_схеми
    оголошення компонентів і сигналів
begin
       оголошення і використання компонентів
 end для_всієї_схеми

 entity першого_компонента
architecture першого_компонента
entity другого_компонента
architecture другого_компонента
```

Текст програми:

```
entity comp_example is
    port(r, g, h, n, p, u, j, l, m: in bit;
        oo   : out bit);
end comp_example;

architecture a of comp_example is
    component AND4
      port(a, b, c, d : in bit;
            y : out   bit);
    end component;
    component OR3
      port(w, x, v    : in bit;
            y : out   bit);
    end component;
    signal na1, na2 : bit;
begin
    dd3: OR3 port map (na1, p, na2, oo);
    dd1: AND4 port map (r, g, h, n, na1);
    dd2:  AND4  port  map  (a=>u,  b=>j,  c=>l,
            d=>m, y=>na2);
end a;
entity AND4 is
      port(a, b, c, d : in  bit;
            y : out    bit);
end AND4;
architecture beh1 of AND4 is
begin
    y<=a and b and c and d;
end beh1;

entity OR3 is
      port(w, x, v    : in  bit;
            y : out bit);
end OR3;
architecture beh2 of OR3 is
begin
    y<=w or x or v;
 end beh2;
```

# 2.7. Поради по написанню текстів на мові VHDL

В цьому параграфі будуть наведені загальні рекомендації щодо оформлення програм мовою VHDL. Більшість наведених рекомендацій вже знайома читачу з курсів програмування, але, як показує практика, ці рекомендації майже завжди ігноруються. Хоча їх використання значно полегшує вивчення та розуміння програми, робить можливим повторне використання коду, а також значно скорочує іншим розробникам час, потрібний для усвідомлення роботи пристрою [4].

1. При написанні програм мовою VHDL необхідно пам'ятати, що будь-яка програма та конструкція в результаті компіляції буде синтезована у певний цифровий вузол.
2. Неприпустиме використання назв, які не мають зв'язку з логічними функціями самого сигналу або змінної, архітектури або всієї програми.
3. Вживайте в іменах сигналів, констант, змінних, типів і параметрів тільки маленькі і великі літери для покращення сприйняття імен сигналів. Наприклад, *LocalReadEnable*. Іншим варіантом розділу слів є використання підкреслювання між словами: *Local_Read_Enable, local_read_enable.*
4. Для найменування всіх тактових сигналів застосовуйте *clk,* як в якості префікса або частини назви сигналу. Використання такої назви для тактових сигналів інформує про те, що всі вони мають одне спільне джерело тактового сигналу: *clk_n, clk_50MHz*. Якщо тактових сигналів два або більше, тоді до назви сигналу слід додавати порядковий номер або назву функціонального вузла, який тактує даний сигнал – *clk2*, clk3 і відповідно *clk2_n, clk3_50MHz.*
5. Для сигналів скидання використовуйте префікси *rst, reset.*
6. Використовуйте послідовний порядок бітів від більшого до меншого при описі багаторозрядних змінних або сигналів, тобто від (n-1) до 0. Це дозволить не помилятись при зв'язуванні багаторозрядних змінних між собою.
7. Додавайте *_n* або *n* для того, щоб показати інверсний сигнал. Наприклад, *we_n, nWE*. У тому випадку, коли імена двох

сигналів відрізняються лише символами *_n* (*n*), то необхідно впевнитись, чи не є ці сигнали інверсією один одного.

8. Застосовуйте константи в тексті блоку або програми замість безпосередніх значень.

9. Вживайте імена файлів, які пояснюють їх призначення. Імена файлів не слід роботи занадто довгими. Це ж саме стосується і назв архітектур та сутностей програми.

10. Назва файлу повинна співпадати з назвою сутності (*entity*).

11. При написанні програми за можливістю слід використовувати стандартні бібліотеки *ieee*. Це дозволить отримати код, який буде незалежним від конкретного виробника програмного забезпечення.

12. Декларацію портів треба робити логічною та без протиріч. Назва порту повинна відповідати його призначенню. Кожен порт необхідно описувати в окремому рядку. І додавати до нього стислий коментар, якщо його призначення незрозуміло з назви. Порти слід групувати по функціональному призначенню: входи, виходи, двонаправлені.

13. Не використовуйте режим передачі **`buffer`** для портів.

14. Не використовуйте порти типу **`inout`** для внутрішніх сигналів проекту, оскільки вони замінюються мультиплексорами всередині мікросхеми. Тип порту **`inout`** можна використовувати лише для портів, які відключені до виводів ПЛІС.

15. Використовуйте позначки (*label:*), які стисло характеризують призначення процесу. Крім того, використання позначок на початку та наприкінці оператора процесу дозволяє швидко орієнтуватись в тексті програми.

16. Робіть список чутливості процесів найбільш повним. Коли список чутливості неповний – результати моделювання схеми до синтезу (pre-synthesis) можуть не збігатися з результатами моделювання схеми після синтезу (post-synthesis). У комбінаційних пристроях список чутливості повинен включати в себе всі сигнали, які використовуються в процесі. В послідовністних пристроях необхідно додавати до списку сигнал тактування та асинхронні сигнали. Надлишковість у списку чутливості призводить до збільшення часу, який витрачається на моделювання проекту.

17. Не можна писати до одного сигналу безпосередньо за допомогою декількох процесів, оскільки це еквівалентно з'єднанню виходів разом. Читати один сигнал можна у різних модулях.

18. Не намагайтеся писати великі процеси чи оператори високого ступеня вкладеності, бо це заважає читанню й відлагодженню коду та приводить до важкосинтезованих чи неоднозначно працюючих конструкцій.

19. Під час опису схем необхідно описувати всі варіанти роботи схеми. Навіть тоді, коли такий варіант роботи є дуже малоймовірним. Це передбачає використання для кожного **`if`** частини, **`else`** частини, а також конструкції **`when others`** для оператора **`case`**.

20. Для файлу верхнього рівня використовуйте графічний редактор. В цьому випадку проект буде більш наглядним, а інтерфейсна частина файлів нижнього рівня може бути отримана автоматично з графічних символів за допомогою пункту меню **File** ⇒ **Create/Update** ⇒ **Create HDL Design File for Current File** [15].

## *Література*

1. Антонов А.П. Язык Описания цифровых устройств AlteraHDL. М. РадиоСофт, 2002. – 240 с.

2. Бибило П.Н. О несинтезируемых конструкциях языка VHDL.// Современная электроника, 2008. – №5. – с. 68-71.

3. Бибибло П.Н. Синтезлогических схем с использованием язика VHDL. – М.: СОЛОН-Р, 2002. – 384 с.

4. Денисов А. Несколько советов по проектированию цифровых устройств на VHDL для ПЛИС.//Компоненты и технологии, 2009. – № 12. – С. 31-34.

5. Каршенбойм И.Г. Краткий курс HDL. //Компоненты и технологии, 2008-2009.

6. Поляков А.К. Языки VHDL и VERILOG в проектировании цифровой аппаратуры. – М.: СОЛОН-Пресс, 2003. – 320 с.

7. Сергиенко А.М. VHDL для проектирования вычислительных устройств. – К.: ЧП «Корнейчук», ООО «ТИД»ДС», 2003. – 208 с.

8. Стешенко В.Б. ПЛИС фирмы "Altera": элементная база, система проектирования и языки описания аппаратуры. М.: Издательский дом "Додэка-XXI", 2002. – 576 с.

9. Суворова Е.А., Шейнин Ю.Е. Проектирование цифровых схем на VHDL. – СПб.: БХВ-Петербург, 2003. – 576 с.

10. Уэйкерли Д.Ф.Проектирование цифровых устройств. В 2 томах. – М.: Постмаркет, 2002. – 1088 с.

11. Язык описания аппаратуры цифровых систем – VHDL. Описание языка. – М.: Госстандарт России, 1995. – 142 с.

12. Chu Pong P. RTL hardware design using VHDL. Wiley-Interscience, 2006. – 680 p.

13. IEEE Standard VHDL Language Reference Manual, IEEE Std 1076, 2000 Edition.

14. Perry D.L. VHDL: Programming by Example. McGraw-Hill, 2002. – 480 p.

15. Quartus II Handbook. Version 9.1. Altera Corp., 2009. – 1820 p.

# Розділ 3

# Розробка систем за допомогою мови VHDL


- Опис комбінаційних пристроїв ..... 104
- Опис послідовних схем ..... 116
- Опис цифрових автоматів ..... 129
- Опис пам'яті з використанням VHDL ..... 136
- Пакети, процедури та функції ..... 155
- Створення файлу на мові опису апаратури в пакеті Quartus II .... 160

## 3.1. Опис комбінаційних пристроїв

При описі комбінаційних пристроїв розглянемо наступні методи опису: алгебраїчний опис та табличний опис.

Реалізація комбінаційної схеми на основі алгебраїчної форми запису логічної функції використовує оператор присвоювання, у правій частині якого записується еквівалентний логічний вираз. Однак, слід звернути увагу, що при використанні паралельного оператора присвоювання ініціалізація операторів відбувається по подійному принципу. Порядок запису операторів в цьому випадку не має значення.

**Приклад 3.1.** Програма представляє опис комбінаційної логічної схеми із двома виходами, з використанням паралельних присвоювань.

```
library ieee;
use ieee.std_logic_1164.all;
entity simple_logic is
port (
   a,b,c,d    : in  std_logic;
   out1, out2 : out std_logic);
end simple_logic;

architecture concurrent of simple_logic is
signal a_and_b: std_logic;
begin
      out1<= a_and_b or (c and d and not b)
      or (not a and not b and d);
      out2<= a_and_b or (not a and c and d)
      or
      (c and not b and not d) or (b and not c
      and d);
      a_and_b <= a and b;
end concurrent;
```

Можна застосовувати також і послідовну форму запису правила функціонування з використанням оператора процесу. Архітектурне тіло описаного в цій формі обладнання наведено у прикладі 3.5.

**Приклад 3.2.** Спочатку відзначимо, що всі вхідні сигнали комбінаційної схеми повинні бути включені в список ініціалізаторів процесу для того, щоб будь-яка зміна їх значень викликала виконання оператора присвоювання. Крім того, у цьому випадку неприпустимо *a_and_b* декларувати як сигнал. Це обов'язково повинна біти змінна, причому її розрахунок задається оператором, що передує оператором обчисленню результуючих сигналів. Інакше буде спостерігатись некоректна поведінка схеми, яка полягає в тому, що використовуються значення не безпосередньо отримані в процесі поточного виконання оператора `process`, а значення, обчислені раніше, після попередньої зміни одного із вхідних сигналів.

```
architecture sequential of  simple_logic is
       -- signal a_and_b: std_logic;
       -- неприпустимо у цьому випадку
begin
process (a,b)
       variable a_and_b: std_logic;
begin
       a_and_b := a and b;
       out1 <= a_and_b or (c and d and not b)
       or (not a and not b and d);
       out2 <= a_and_b or (not a and c and d)
       or
    (c and not b and not d) or (b and not c
and d);
end process;
end sequential;
```

**Приклад 3.3.** Якщо a_and_b це все-таки сигнал (наприклад, необхідний для передачі інформації іншим операторам програми або до портів), то слід виділити його обчислення в окремий процес або паралельний оператор.

```
 architecture two_processes of  simple_logic
is
```

```
        signal a_and_b: std_logic;
begin
        process (a,b)
        begin
              a_and_b <= a and b;
        end process;
        process (a,b,c,d,a_and_b)
        begin
              out1<= a_and_b or (c and d and not
              b) or (not a and not b and d);
              out2<= a_and_b or (not a and c and
              d) or
              (c and not b and not d) or (b and
              not c and d);
        end  process;
 end  two_processes;
```

Реалізація таблиці дійсності логічної функції може бути виконана в такий спосіб:

- за допомогою паралельного умовного присвоювання **when**…**else**;
- з використанням паралельного присвоювання за вибором **with**…**select**;
- з використанням умовного оператора **if**…**then**;
- за допомогою оператора вибору **case**.

Приклади використання паралельного умовного присвоювання та присвоювання за вибором наведені у параграфі 2.5. Нижче ми розглянемо використання послідовних операторів для реалізації комбінаційних пристроїв.

**Приклад 3.4. Буфер з третім станом на виході.** Розглянемо простий восьми-розрядний буфер з третім станом на виході. Цей буфер за своєю логікою роботи нагадує мікросхему К555АП5 (74LS244): якщо сигнал en дорівнює одинці – вхідна інформація надходить на вихід. Якщо ж сигнал en дорівнює нулю – вихід відключається і переходить у третій стан.

Реалізація такої програми можлива лише при роботі з вихідними портами мікросхеми ПЛІС (див. параграф 1.1). При роботі такого модуля з внутрішніми сигналами ПЛІС буфер з третім станом буде автоматично замінений або на мультиплексори або на примітиви wire в залежності від контексту реалізації.

```
library ieee;
use ieee.std_logic_1164.all;
entity tristate_dr is
port (
   d_in  : in  std_logic_vector(7 downto 0);
   en    : in  std_logic;
   d_out : out std_logic_vector(7 downto 0));

end tristate_dr;
architecture behavior of tristate_dr is
begin
  process(d_in, en)
  begin
    if en='1' then
      d_out <= d_in;
    else
      d_out <= "ZZZZZZZZ";
    end if;
  end process;
end behavior;
```

Результат компіляції буфера з третім станом у RTL viewer показаний на рисунку 3.1, а часові діаграми роботи – на рисунку 3.2.

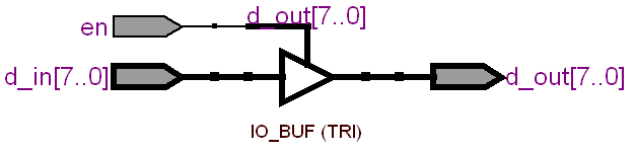



*Рисунок 3.1 – Буфер з третім станом у RTL viewer*

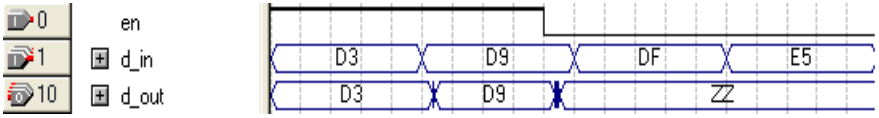



*Рисунок 3.2 – Часові діаграми роботи буфера з третім станом*

При описі комбінаційних пристроїв можливе використання як паралельних операторів умовного присвоювання так і оператора процесу.

**Приклад 3.5.** Опис мультиплексора за допомогою оператора процесу.

На входи мультиплексора надходить чотири трьох розрядних шини *I3*, *I2*, *I1*, *I0*, які комутуються на один трьохрозрядний вихід *O*. Оскільки цей приклад описує асинхронний пристрій, то у списку ініціалізації оператора процесу присутні всі вхідні сигнали – I3, I2, I1, I0, S. Зміна будь-якого з них призведе до запуску процесу і зміни вихідного сигналу.

```
library ieee;
use ieee.std_logic_1164.all;
entity Mux is
port(   I3 :      in       std_logic_vector   (2
downto 0);
    I2 :     in  std_logic_vector (2 downto 0);
    I1 :     in  std_logic_vector (2 downto 0);
    I0 :     in  std_logic_vector (2 downto 0);
    S  :in  std_logic_vector (1 downto 0);
    O  :out std_logic_vector (2 downto 0)
);
end Mux;
architecture beh of Mux is
begin
    process(I3, I2, I1, I0, S)
    begin
        case S is
           when "00" =>O <= I0;
           when "01" =>O <= I1;
```

```
            when "10" =>O <= I2;
            when "11" =>O <= I3;
            when others =>    O <= "ZZZ";
        end case;
    end process;
end beh;
```

Результат компіляції у RTL viewer показаний на рисунку 3.3. Як видно з нього програма зкомпілювалась у три мультиплексори – за розрядністю вихідного порту. Кожен з мультиплексорів має по чотири інформаційних входи – за кількістю вхідних шин, і два керуючих входи S [1..0]. Результати симуляції мультиплексора показані на рисунку 3.4.

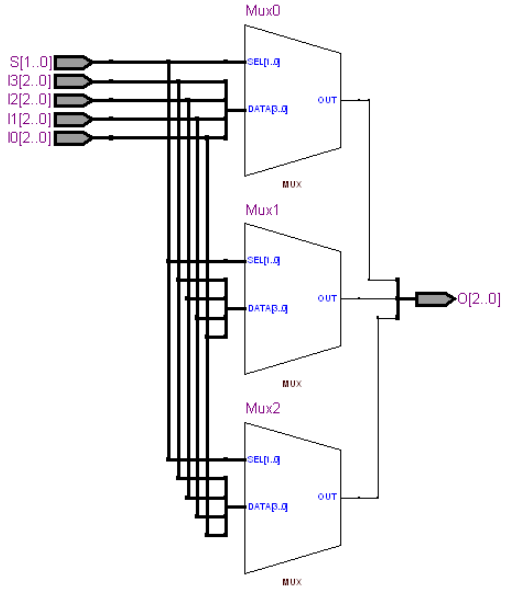



*Рисунок3.3 – Реалізація мультиплексора у RTL viewer*

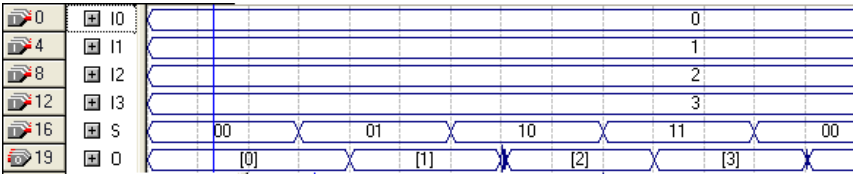



*Рисунок3.4 – Результати симуляції мультиплексора*

**Дешифратор**. Для опису дешифратора знову використаємо оператор процесу і оператор case. В такому випадку програма буде дуже схожа на програму мультиплексора. Необхідно лише правильно описати логіку роботи дешифратора за допомогою оператора case.

**Приклад 3.6.** Дешифратор.

```
 library ieee;
use ieee.std_logic_1164.all;

entity decoder is
port (
   I : in  std_logic_vector (1 downto 0);
   O : out std_logic_vector (3 downto 0));
end decoder;

architecture beh of decoder is
begin
process (I)
    begin
        case I is
           when "00" => O <= "0001";
           when "01" => O <= "0010";
           when "10" => O <= "0100";
           when "11" => O <= "1000";
           when others => O <= "XXXX";
      end case;
    end process;
 end beh;
```

Результат компіляції дешифратора у RTL viewer показаний на рисунку 3.5. Як видно, дешифратор був реалізований за допомогою чотирьох мультиплексорів, які керуються вхідним сигналом I[1..0]. На інформаційні входи кожного з мультиплексорів надходить чотирьохрозрядне число. Наприклад, для першого мультиплексора, який керує виходом O[3], це число 8 або 1000 у двійковому коді. Для другого – число 4 або ж 0100 у двійковому коді. Тобто кожен з мультиплексорів на інформаційному вході має набор вихідних значень для одного виходу.

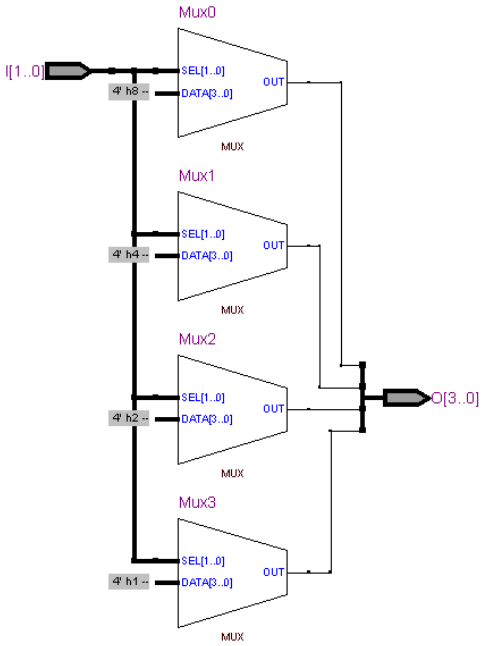



*Рисунок 3.5 – Дешифратор у RTL viewer*

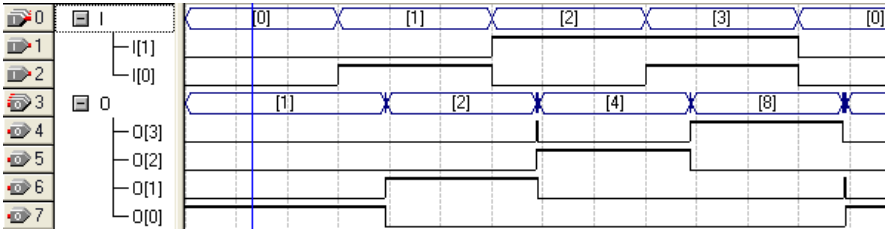



*Рисунок 3.6 – Часові діаграми роботи дешифратора*

**Компаратори.** При описі компараторів розглянемо два приклади – беззнаковий та знаковий компаратор.

**Приклад 3.7. Беззнаковий компаратор.** На його входи надходить два трьохбітних сигнали: A і B. Компаратор містить три вихідних сигнали: менше (less), дорівнює (equal) та більше (greater). Робота компаратора описується таблицею 3.1.

Таблиця 3.1 – Таблиця дійсності беззнакового компаратора

| | less | equal | Greater |
|---|---|---|---|
| A<B | 1 | 0 | 0 |
| A=B | 0 | 1 | 0 |
| A>B | 0 | 0 | 1 |

```
library ieee;
use ieee.std_logic_1164.all;

entity Comparator is
generic (n: natural :=2);
port (
    A        : in  std_logic_vector(n-1 downto
0);
    B        : in  std_logic_vector(n-1 downto
0);
    Less     : out std_logic;
    equal    : out std_logic;
    greater : out std_logic);
```

```
end Comparator;
architecture behv of Comparator is
begin
    process(A,B)
    begin
        if (A<B) then
        less <= '1';
        equal <= '0';
        greater <= '0';
    elsif (A=B) then
        less <= '0';
        equal <= '1';
        greater <= '0';
else
        less <= '0';
        equal <= '0';
        greater <= '1';
    end if;
    end process;
end behv;
```

Результат компіляції у RTL viewer показаний на рисунку 3.7. Компаратор представлений двома компараторами – дорівнює (Equal0) та менше (LessThan0), які формують сигнал одиниці при виконанні умови. Розглянемо роботу цієї схеми більш докладно.

Коли сигнал A менше ніж сигнал B, то на виході компаратора Equal0 сигнал нуля, а компаратора LessThen0 – одиниці. Тоді обидва мультиплексори перемикаються на передачу нижнього вхідного сигналу і ми отримуємо таку комбінацію вихідних сигналів: greater – 0, equal – 0, less – 1.

Коли сигнал A дорівнює сигналу B, то на виході компаратора Equal0 сигнал одиниці, а компаратора LessThen0 – нуля. Мультиплексори виводять сигнали з верхнього порту і ми отримаємо таку комбінацію вихідних сигналів: greater – 0, equal – 1, less – 0.

Останній випадок – сигнал A більше ніж сигнал B. Як і у попередньому випадку мультиплексори виводять дані з верхнього порту. На виході обох компараторів сигнали нуля і вихідні порти мають такий стан: greater – 1, equal – 0, less – 0.

Як бачимо, компаратор працює так, як і було описано в таблиці 3.1. Часові діаграми роботи компаратора наведені на рисунку 3.8.

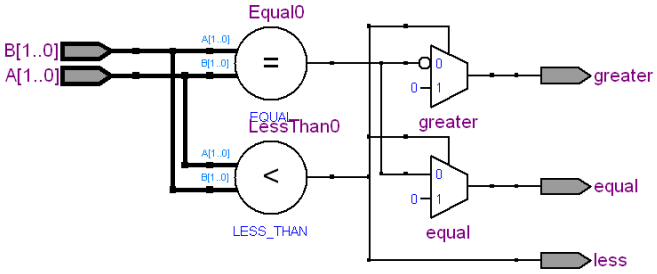



*Рисунок3.7 – Беззнаковий компаратор у RTL viewer*

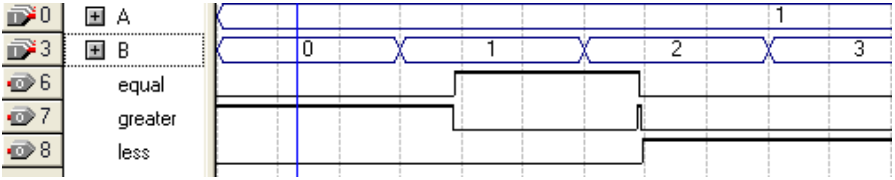



*Рисунок 8 – Часові діаграми роботи роботи компаратора*

**Приклад 3.8. Знаковий компаратор.** Опис знакового компаратора зробимо на основі беззнакового. Для цього змінимо лише типи вхідних портів: тип std_logic_vector для портів A і B замінимо на signed. Для цього також необхідно додати декларацію пакету numeric_std з бібліотеки ieee. Архітектурне тіло програми залишиться незмінним тому його наводити не будемо.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity Signed_Comparator is
generic (n: natural :=2);
port (
```

```
        A       : in  signed (n-1 downto 0);
        B       : in  signed (n-1 downto 0);
        Less    : out std_logic;
        equal   : out std_logic;
        greater : out std_logic
    );
    end Signed_Comparator;
```

Результат компіляції знакового компаратора у RTL viewer наведений на рисунку 3.9. Як видно він відрізняється від беззнакового лише оператором «менше»: на верхній порт находять сигнали B[1] та A[0], а на нижній – A[1] та B[0]. Це пов'язано з необхідністю роботи компаратора у додатковому коді.

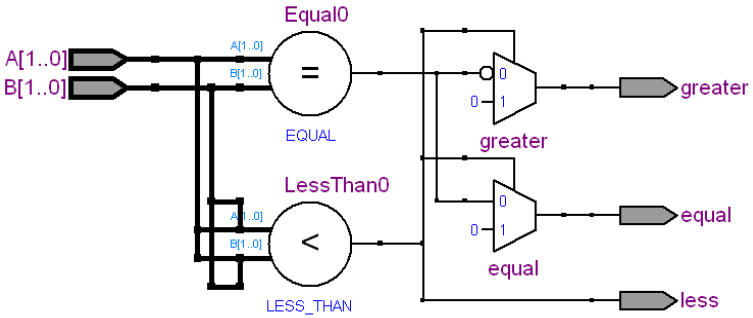



*Рисунок 3.9 – Знаковий компаратор у RTL viewer*

Часові діаграми роботи знакового компаратора наведені на рисунку 3.10.

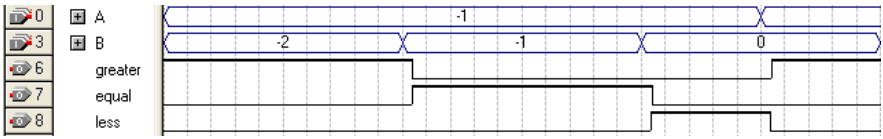



*Рисунок 3.10 – Часові діаграми роботи знакового компаратора*

## 3.2. Опис послідовних схем

### 3.2.1. Опис тригерів

Тригер зручно описувати в VHDL за допомогою оператора PROCESS, причому керуючі сигнали краще включати до списку ініціалізаторів, а не до оператора **wait**. Використання тактового сигналу у списку ініціалізаторів та конструкції типу **clk'event and clk = '1'** автоматично синтезує тригер. Також необхідно пам'ятати, що в пакеті Quartus II всі тригери реалізуються за допомогою D-тригерів. Для реалізації всіх інших тригерів використовується D-тригер та додаткова логічна схема.

Розглянемо наступні способи керування тригерами:

- асинхронне керування;
- динамічне керування, або керування фронтом;
- змішані варіанти.

**Приклад 3.9. RS-тригер.** Асинхронне керування характеризується тим, що події на інформаційних входах безпосередньо викликають зміну стану тригера. Єдиний тригер з асинхронним керуванням це RS-тригер, функціональна модель якого може бути представлена в такий спосіб:

```
entity rs_trigger is
    port(
       r, s : in  bit;
       q    : out bit);
end rs_trigger;
architecture a of rs_trigger is
begin
      process (r, s)
      begin
            if s='1' then q<='1';
            elsif r='1' then q<='0';
            end if;
```

```
        end process;
end a;
```

Результати компіляції у RTL-viewer RS-тригера показані на рисунку 3.11. Як видно, для реалізації тригера використовується тільки асинхронне керування, а з вбудованого тригера – тільки асинхронні входи установки (pre) і скидання (clr).

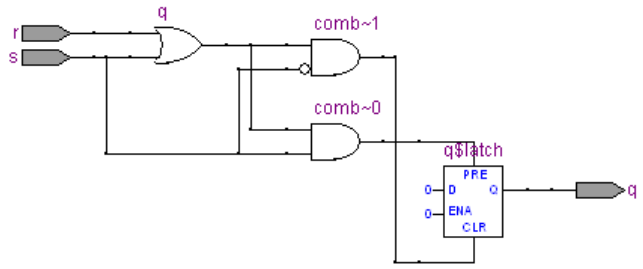



*Рисунок 3.11 – RS-тригер у RTL viewer*

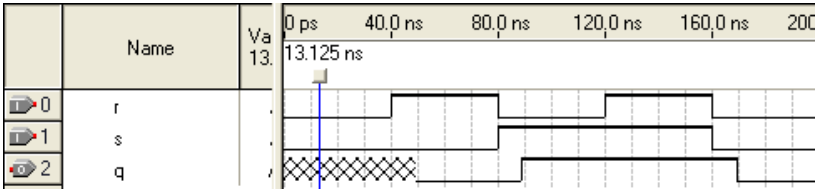



*Рисунок 3.12 – Результат симуляції RS-тригера*

**Синхронний тригер**

Більшість реальних тригерів є синхронними пристроями, які працюють або по передньому або по задньому фронту тактової частоти. Для опису таких пристроїв необхідно включити до списку ініціалізації процесу тактовий сигнал та всі асинхронні сигнали.

**Приклад 3.10.** В цьому прикладі наведений код D-тригера, що працює за переднім фронтом тактової частоти. Для перевірки переднього фронту використовується конструкція `clk'event and clk= '1'`, в якій перевіряються наявність події у сигналі `clk` (перевіряється атрибут сигналу **`event`**)та значення тактового сигналу.

```
 library ieee;
use ieee.std_logic_1164.all;
entity d_trigger is
    port (
       d    : in  std_logic;
       clk  : in  std_logic;
       q    : out std_logic);
end d_trigger ;

architecture a of d_trigger is
begin
    process (clk)
    begin
       if (clk'event and clk= '1')
                  then  q<= d;
            end if;
    end process;
 end a;
```

Результати моделювання роботи тригера наведені на рисунку 3.13.

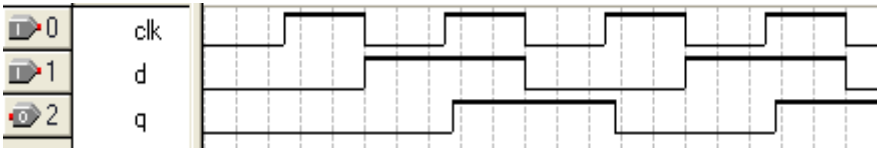



*Рисунок 3.13 – Часові діаграми роботи D-тригера*

**Приклад 3.11. JK-тригер.** Розглянемо більш складний приклад, в якому крім синхронних сигналів будуть також і асинхронні. У прикладі розглядається опис JK-тригера. Для визначення фронту сигналу в цьому прикладі використовується конструкція rising_edge, яка описана в бібліотеці std_logic_1164.

Для опису стану тригера використовується внутрішній сигнал state, який після компіляції буде реалізований на основі програмуємого тригера, що входить до складу логічного елемента мікросхеми ПЛІС (рисунок 3.14).

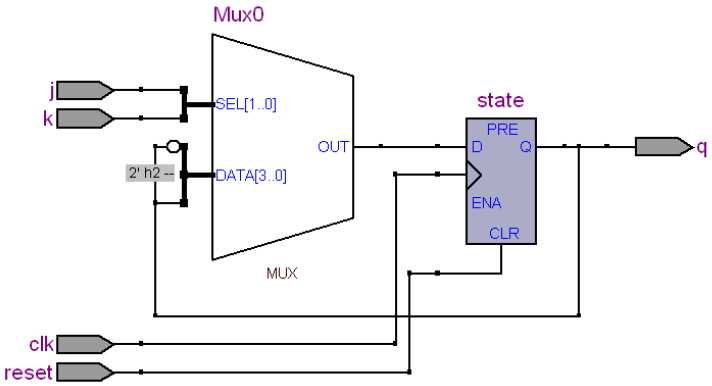



*Рисунок 3.14 – JK тригер у RTL viewer*

Для реалізації таблиці дійсності тригера формується двохбітний сигнал input, старший біт якого являє собою сигнал j, а молодший – k. В операторі процесу значення цього внутрішнього сигналу керують вибором наступного значення сигналу state в операторі **case**. На рисунку 3.14 Видно, що оператор **case** синтезується у чотирьохвходовий мультиплексор.

У списку ініціалізації процесу записані два сигнали – тактовий clk та сигнал скидання reset. Сигнал скидання асинхронний, тому його значення в операторі процесу перевіряється раніше за перевірку тактового сигналу.

Результати моделювання JK-тригера наведені на рисунку 3.15.

```
library ieee;
```

```
use ieee.std_logic_1164.all;
entity jk_ff is
port (
    clk   : in  std_logic;
    j, k  : in  std_logic;
    reset : in  std_logic;
    q     : out std_logic
);
end jk_ff;
architecture behv of jk_ff is
    signal state: std_logic;
    signal  input:  std_logic_vector(1  downto
0);
begin
       input <= j & k;
       process(clk, reset) is
       begin
       if (reset='1') then state <= '0';
       elsif (rising_edge(clk)) then
           case (input) is
                 when  "11"  =>  state  <=  not
            state;
                 when "10" =>    state <= '1';
                 when "01" => state <= '0';
                 when others => null;
            end case;
       end if;
       end process;
       q <= state;
end behv;
```

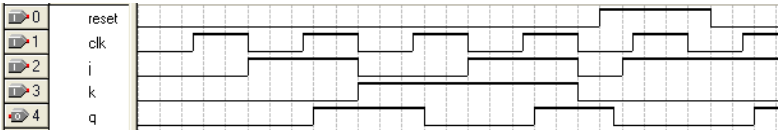



*Рисунок 3.15 – Часові діаграми роботи JK-тригера*

## 3.2.2. Опис регістрів

Опис паралельних регістрів нічим не відрізняється від опису D-тригерів. Єдина відмінність – це зміна розрядності вхідного та вихідного сигналів.

**Приклад 3.11. Восьмиразрядний регістр зі скиданням і дозволом запису.** Крім того, зробимо цей опис параметризованих, тобто визначимо параметр, змінюючи який можна отримувати регістра необхідної розрядності. При виконанні операції скидання необхідно записати у драйвер вихідного сигналу нульові значення. Оскільки модуль параметризуємий то кількість розрядів змінюється. Тому для запису нульових значень використовуються оператори перетворення типів, описані у параграфі 2.3. Оператор to_unsigned для визначення розрядності використовує параметр data_width, який і визначає загальну розрядність пристрою.

Регістр використовує змішане керування: асинхронне керування по входу скидання reset, синхронне керування по тактовому входу clk й керування рівнем по входу дозволу запису ena. Список ініціалізації процесу містить лише сигнали скидання та тактування. Всі інші сигнали аналізуються при надходженні фронту тактової частоти. Запис у регістр відбувається при таких значеннях керуючих сигналів: сигнали скидання reset та дозволу запису ena дорівнюють одиниці.

Результати моделювання регістра показані на рисунку 3.16, а результат компіляції у RTL viewer – на рисунку 3.17. Як видно з результатів компіляції регістр реалізується на D-тригерах, а сигнали скидання та дозволу роботи використовують керуючі входи вбудованого тригера логічного елемента.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity reg is
    generic (data_width: integer:= 32);
    port(
```

```
    d:    in    std_logic_vector    (data_width-1
downto 0);
    clk   : in std_logic;
    reset: in std_logic;
    ena   : in std_logic;
    q:    out   std_logic_vector    (data_width-1
downto 0));
end reg;

architecture beh of reg is
begin
process (clk, reset)
    begin
       if reset = '0' then
            q <= std_logic_vector (to_unsigned
(0, data_width));
       elsif (clk'event and clk = '1') then
            if ena = '1' then
                 q <= d;
            end if;
       end if;
    end process;
end beh;
```

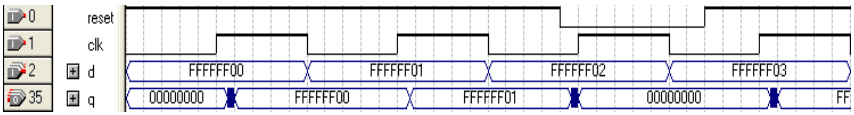



*Рисунок 3.16 – Часові діаграми роботи паралельного регістра*

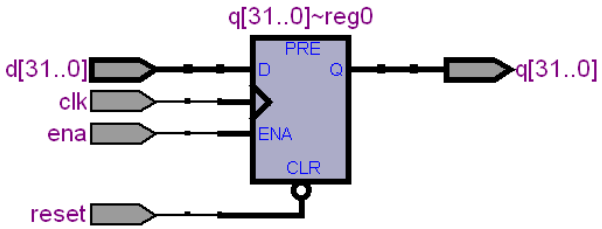



*Рисунок 3.17 – Паралельний регістр у RTL viewer*

**Приклад 3.12. Регістр зсуву.** Цей регістр має одно розрядний вхід та одно розрядний вихід. Вхідні дані поступово зсуваються всередині регістра і загальна затримка даних складає 8 тактів. Для того, щоб виконати зсув даних створимо сигнал, що являє собою масив з восьми чарунок типу std_logic. При надходженні переднього фронту тактової частоти виконується зсув даних вправо. При цьому молодші сім розрядів масиву переписуються у старші сім розрядів sr(7 **downto** 1) <= sr(6 **downto** 0), а у молодший розряд записується вхідний сигнал.

```
 library ieee;
use ieee.std_logic_1164.all;
entity shift_register is
    port (
      clk    : in  std_logic;
      enable : in  std_logic;
      sr_in  : in  std_logic;
      sr_out : out std_logic);
end entity;
architecture rtl of shift_register is
      type sr_length is array (7 downto 0) of
      std_logic;
signal sr: sr_length;
begin
      process (clk)
      begin
      if (rising_edge(clk)) then
```

```
                if (enable = '1') then
                      sr(7 downto 1) <= sr(6 downto
0);
                      sr(0) <= sr_in;
                end if;
          end if;
          end process;
          sr_out <= sr(7);
    end rtl;
```

Результат компіляції регістру зсуву у Technology Map Viewer показаний на рисунку 3.18, а часові діаграми роботи – на рисунку 3.19.

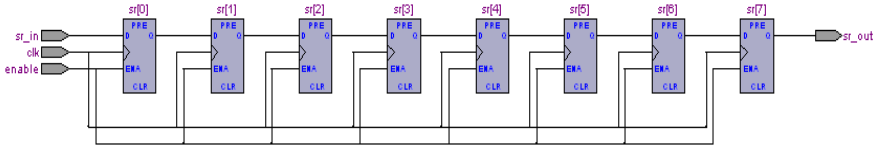



*Рисунок 3.18 - Регістру зсуву у Technology Map Viewer*

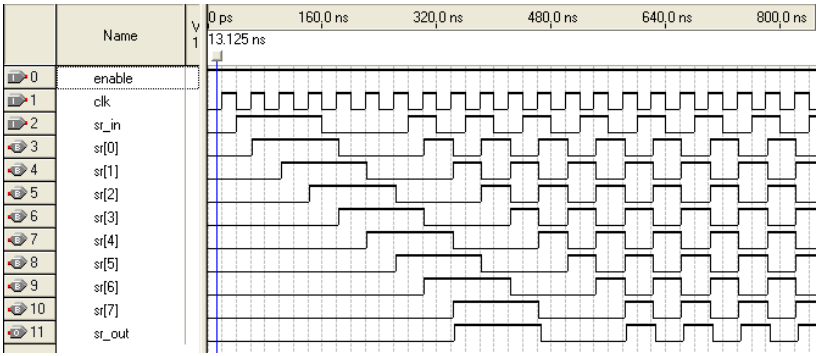



*Рисунок 3.19 – Часові діаграми роботи регістру зсуву*

## 3.2.3. Опис лічильників

**Приклад 3.13. Простий сумуючий лічильник.** Він буде мати лише один тактовий вхід і чотирьох розрядний вихід. Вихідний порт у цьому прикладі описаний за допомогою типу integer з діапазоном значень від 0 до 15. Для роботи лічильника необхідно описати внутрішній сигнал csignal такого ж самого типу, що і вихідний порт. Сама операція інкременту описується як додавання одиниці до попереднього значення. Оскільки вихідний порт не може виступати в якості джерела сигналу, то і використовується внутрішній сигнал.

```
library ieee;
use ieee.std_logic_1164.all;
entity count is
    port    (clk     : in  std_logic;
       out_data : out integer range 0 to 15  );
end count;
architecture a of count is
    signal  csignal: integer range 0 to 15;
begin
    process (clk)
    begin
       if (clk'event and clk = '1') then
            csignal <= csignal + 1;
       end if;
    end process;
    out_data<= csignal;
 end a;
```

Результат компіляції лічильника у RTL viewer показаний на рисунку 3.20. Як видно, лічильник реалізований у вигляді чотирьохрозрядного суматора, що додає одинцю до попереднього результату та чотирьох тригерів для збереження даних. Часові діаграми лічильника показані на рисунку 3.21.

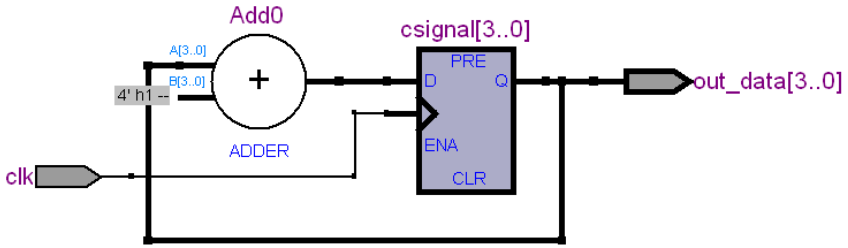



*Рисунок 3.20 – Лічильник у RTL viewer*

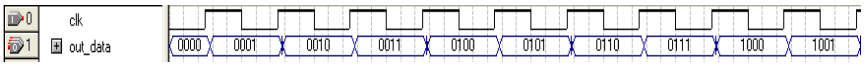



*Рисунок 3.21 – Часові діаграми роботи лічильника*

**Приклад 3.13. Лічильник з асинхронним скиданням, синхронним попереднім записом і сигналом дозволу лічби.** Наступний приклад зробимо дещо складнішим – реалізуємо параметризований лічильник, в якому буде змінюватись розрядність лічильника.

Для того щоб зробити модуль параметризованих додамо конструкцію з словом **generic**, в якій визначимо параметр width, який буде визначати розрядність лічильника, тобто лічильник буде рахувати від нуля до $2^{width}$-1.

Як було описано в попередньому прикладі, для роботи лічильника необхідно описати внутрішній сигнал csignal, розрядність якого дорівнює розрядності вихідного порта і визначається параметром width. Після компіляції цей сигнал буде реалізований на вбудованих тригерах (рисунок 3.23).

Оскільки лічильник містить лише оди асинхронний сигнал – скидання, то у список ініціалізації процесу включимо лише два сигнали – тактовий clk та скидання clrn. Сигнал clrn в процесі аналізуємо до надходження тактового і якщо він дорівнює нулю, то у внутрішній сигнал csignal буде записане нульове значення.

При надходженні переднього фронту тактового сигналу аналізуємо стани ще двох керуючих сигналів – завантаження load та дозволу роботи ena. Якщо сигнал load дорівнює одиниці, то у внутрішній сигнал записуємо дані з вхідного порта d. Сигнал ena дозволяє або забороняє інкремент значення внутрішнього сигналу csignal. Часові діаграми роботи лічильника показані на рисунку 3.22.

```
library ieee;
use ieee.std_logic_1164.all;

entity count is
    generic (width: natural:= 4);
    port(
    d              :   in integer range 0 to
2**width-1;
    clk       : in  std_logic;
    clrn      : in  std_logic;
    ena       : in  std_logic;
    load      : in  std_logic;
    out_data   :  out  integer  range  0  to
2**width-1
    );

end count;

architecture a of count is
    signal  csignal:  integer  range  0  to
2**width-1;
begin
    process (clk, clrn)
    begin
       if clrn = '0' then csignal <= 0;
       elsif (clk'event and clk = '1') then
            if load = '1' then csignal <= d;
            else
                 if ena = '1' then
                      csignal <= csignal + 1;
                 else
```

```
                        csignal <= csignal;
                    end if;
                end if;
            end if;
        end process;
        out_data<= csignal;
end a;
```

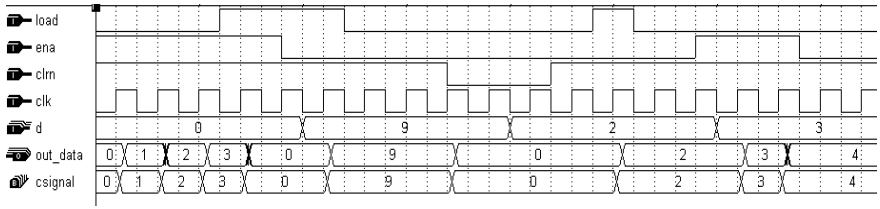



*Рисунок 3.22 – Часові діаграми роботи лічильник зі скиданням, попереднім завантаженням та дозволом роботи*

На рисунку 3.23 показаний вигляд лічильника у RTL-viewer. Для збільшення на одиницю використовується суматор Add0, значення якого подається на мультиплексор, що керується сигналом дозволу роботи ena. Якщо лічба заборонена на вхід D-тригерів подається попереднє значення. Якщо ж лічба дозволена, то на вхід тригерів надходить сигнал з виходу суматора. Наступний мультиплексор керується сигналом завантаження load і подає на вхід D-тригерів або значення з вхідного порта d або з виходу попереднього мультиплексора. Скидання лічильника забезпечується за рахунок сигналів скидання тригерів clr.

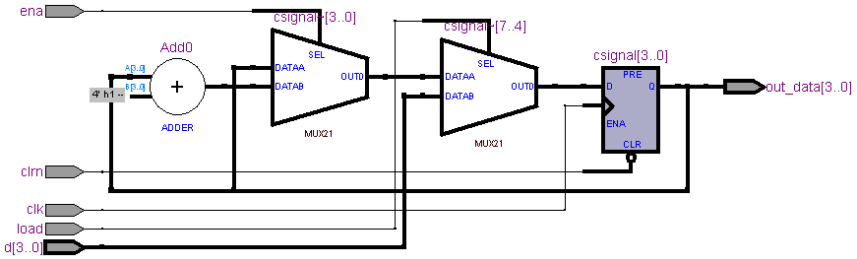



*Рисунок 3.23 – Лічильник зі скиданням, попереднім завантаженням та дозволом роботи у RTL-viewer*

## *3.3. Опис цифрових автоматів*

Схематично цифровий автомат може бути представлений за допомого двох взаємопов'язаних частин (рисунок 3.24). Нижня частина містить послідовнісну частину (тригери), а верхня – комбінаційну логіку цифрового автомата.

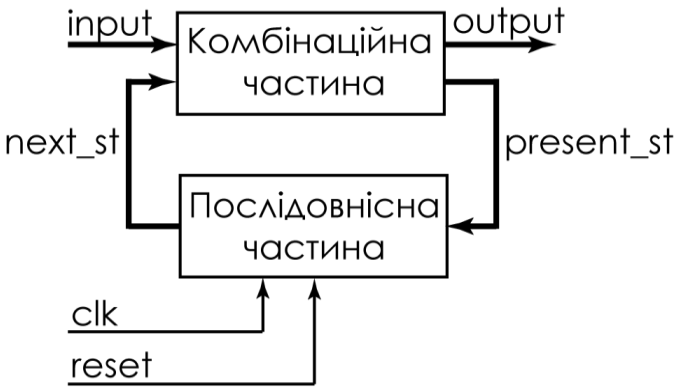



*Рисунок 3.24 – Схематичне зображення цифрового автомата*

В послідовнісну частину входить три входи: тактовий (clk), скидання (reset) та код наступного стану (next_st). Тактовий сигнал та сигнал скидання призначені для тригерів, які зберігають стан цифрового автомата. Вихідний сигнал нижньої частини кодує поточний стан цифрового автомата (present_st).

Комбінаційна частина на свій вхід отримує код поточного стану цифрового автомата та зовнішні сигнали (input). Виходом верхнього блока буде наступний стан автомата та його вихідні сигнали (output).

Якщо вихідний сигнал цифрового автомата залежить лише від поточного стану, то такий автомат називається автоматом Мура, якщо ж вихідний сигнал залежить від поточного стану та вхідних сигналів, то такий цифровий автомат носить назву автомата Мілі.

Розподіл цифрового автомата на дві частини дозволяє описати його за допомогою двох процесів, які описують верхню і нижню частини. Такий варіант опису цифрового автомата дозволяє розробнику розділяти логіку переходів від логіки формування вихідних сигналів. А це, в свою чергу, дозволяє більш просто описувати і відлагоджувати цифровий автомат.

Нижня частина в списку ініціалізаторів процесу містить сигнали clk та reset. При надходженні сигналу reset цифровий автомат переходить до початкового стану, з якого починається робота всього автомата. Передній фронт сигналу clk призводить до запису поточного стану ЦА, тобто сигналу present_st.

Для опису станів цифрового автомата необхідно використовувати перелічимий тип. Для цього описується тип, значеннями якого є стани цифрового автомата. Потім описується сигнал цього, в якому і буде зберігатись поточний стан автомата.

```
type state_type is (Init, state1, state2, …);
signal state: state_type;
```

Оскільки за збереження стану цифрового автомата відповідає нижня частина схеми на рис. 3.24. При реалізації буде отримана схема з декількох тригерів. В залежності від методу кодування станів кількість тригерів може змінюватись, що впливає на швидкодію та об’єм схеми. Більш докладно про методи кодування див. параграф 5.4.2.

Процес, що описує послідовністну частину цифрового автомата може бути описаний за допомогою такого шаблона процесу:

```
process (clk, reset)
begin
    if reset = '1' then present_st <= Init;
    elsif (rising_edge(clk)) then
            case state is
                when stateX =>
                    present_st <= next_st;
                …
            end case;
    end if;
end process;
```

Описаний процес дуже простий і містить лише аналіз сигналу скидання reset та тактової частоти clk. По сигналу скидання цифровий автомат переходить до початкового стану Init. Всі інші переходи виконуються по передньому фронту тактової частоти. По фронту тактової частоти значення наступного стану (next_st) записується до тригера, який зберігає стан цифрового автомата (present_st). Для опису роботи цифрового автомата в цілому необхідно проаналізувати всі стани цифрового автомата, тобто таблицю переходів. Для цього можна використовувати оператор **case.**

Верхня частина, яка описує комбінаційну схему за допомогою оператора процесу, повинна в списку ініціалізаторів містити вхідні сигнали та сигнал наступного стану (next_st).

**Приклад 3.14.** Розглянемо в якості прикладу цифровий автомат, який описує роботу світлофору. Цей цифровий автомат має п’ять станів: початковий (Init), червоний (R), зелений (G) та два жовтих – один для переходу з червоного до зеленого (RG), а другий з зеленого до червоного (GR). У тому випадку, коли вхід cnt дорівнює нулю ніяких переходів не виконується, коли вхід дорівнює одиниці то відбувається перехід в наступний стан. Граф цифрового автомата зображений на рисунку 3.25.

Цей же автомат може бути представлений за допомогою таблиці переходів (таблиця 3.2) та таблиці виходів (таблиця 3.3). У таблиці переходів у кожній клітинці записується наступний стан, в який

переходить автомат в залежності від умови переходу та поточного стану. Таблиця виходів ілюструє відповідність між станами автомата та його виходами.

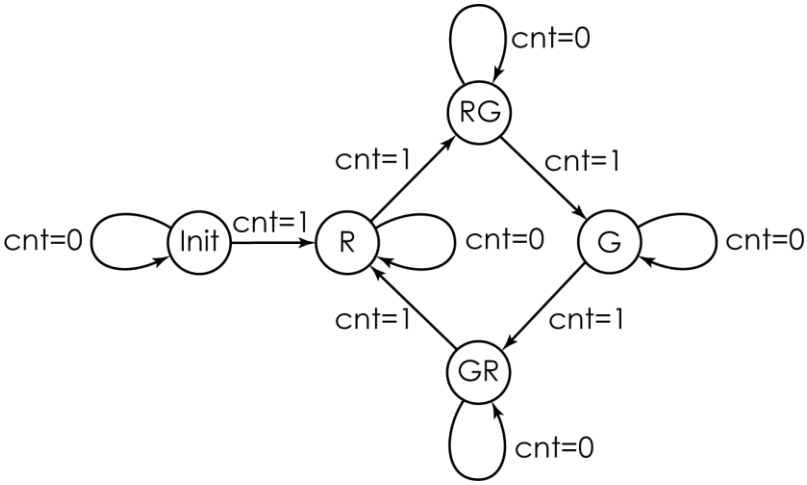



*Рисунок 3.25 – Граф цифрового автомата світлофора*

*Таблиця 3.2 – Таблиця переходів цифрового автомата*

| Поточний стан (present_st) | Наступний стан (next_st) | Умова переходу |
|---|---|---|
| Init | Init | $\overline{cnt}$ |
| Init | R | cnt |
| R | R | $\overline{cnt}$ |
| R | RG | cnt |
| RG | RG | $\overline{cnt}$ |
| RG | G | cnt |
| G | G | $\overline{cnt}$ |
| G | GR | cnt |
| GR | GR | $\overline{cnt}$ |
| GR | R | cnt |

*Таблиця 3.3 – Таблиця виходів цифрового автомата*

| Поточний стан (present_st) | Вихідний сигнал |
|---|---|
| Init | 000 |
| R | 100 |
| RG | 010 |
| G | 001 |
| GR | 010 |

Для опису станів ЦА необхідно описати перелічимий тип, в якому будуть перераховані всі стани. У наведеному прикладі вводиться тип *state_type*, який містить п’ять значень: *Init*, *R*, *RG*, *G*, *GR*. Для роботи конкретного екземпляра цифрового автомата потрібно описати сигнал цього типу. У прикладі це сигнал *state*. При виконанні процесів цей сигнал буде приймати значення поточного стану цифрового автомата.

```
library ieee;
use ieee.std_logic_1164.all;

entity traffic is
    port(clk       : in    std_logic;
       cnt : in    std_logic;
       reset       : in    std_logic;
       output      : out   std_logic_vector(2
downto 0));
end entity;
architecture rtl of traffic is
    type state_type is (Init, R, RG, G, GR);
    signal state: state_type;
begin

      -- опис послідовністної частини
state_proc: process (clk, reset)
begin
```

```
    if reset = '1' then state <= Init;
    elsif (rising_edge(clk)) then
       case state is
           when Init => if cnt = '1'
                        then state <= R;
                        else state <= Init;
                 end if;
           when R => if cnt = '1'
                        then state <= RG;
                        else state <= R;
                 end if;
           when RG => if cnt = '1'
                        then state <= G;
                        else state <= RG;
                 end if;
           when G => if cnt = '1'
                        then state <= GR;
                        else state <= G;
                 end if;
           when GR => if cnt = '1'
                        then state <= R;
                        else state <= GR;
                 end if;
       end case;
    end if;
end process;

       -- опис комбінаційної частини
out_proc: process (state)
begin
    case state is
       when Init =>    output <= "000";
       when R =>  output <= "100";
       when RG => output <= "010";
       when G =>  output <= "001";
       when GR => output <= "010";
    end case;
```

```
end process;
end rtl;
```

Часові діаграми роботи цифрового автомата наведені на рисунку 3.26. На рисунку крім портів показані також стани цифрового автомата. Для імен станів використовується такий синтаксис: *ім'я_сигнала.значення_стану*. Наприклад, state.R.

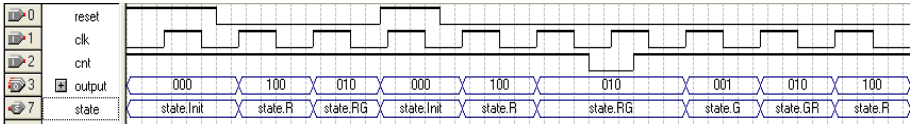



*Рисунок 3.26 –Часові діаграми роботи цифрового автомата керування світлофором*

Описаний вище цифровий автомат формує асинхронні вихідні сигнали. Однак велика кількість систем потребує щоб вихідні сигнали були синхронними відносно фронту сигналу тактової частоти. Для того, щоб отримати синхронні вихідні сигнали процес `out_proc` повинен містити в списку ініціалізаторів крім сигналу стану цифрового автомата також сигнал тактової частоти.

```
out_proc: process (state, clk)
begin
    if (rising_edge(clk)) then
            case state is
            when Init =>    output <= "000";
            when R =>  output <= "100";
            when RG => output <= "010";
            when G =>  output <= "001";
            when GR => output <= "010";
       end case;
    end if;
 end process;
```

# 3.4. Опис пам'яті з використанням VHDL

## Загальні положення

Пам'ять на мові VHDL описується як масив векторів. Розрядність вектора визначається розрядністю чарунки пам'яті, а кількість векторів кількістю чарунок в модулі пам'яті.

Наприклад, для модуля пам'яті з 32 чарунок, кожна з яких містить 8 біт необхідно оголосити масив, в якому 32 вектори, кожен з яких є восьмирозрядним.

```
type mem is array (0 to 31) of
std_logic_vector (7 downto 0);
```

Далі необхідно описати входи адреси, входи та виходи даних. Тип портів даних повинен співпадати з типом даних окремої чарунки. Для наведеного вище приклада – це *std_logic_vector (7 downto 0).*

```
data_in: in std_logic_vector (7 downto 0);
data_out: out std_logic_vector (7 downto 0);
```

Тип даних для адреси – *integer* або засновані на ньому типи. Тип *integer* необхідний тому, що адреса використовується як індекс масиву пам'яті.

```
addr: in integer range 0 to 31;
```

Опис пам'яті краще виконувати за допомогою параметризованих модулів. Це дозволяє повторно використовувати написаний код. Нижче наведений приклад параметризованого модуля розміром 32×8. У прикладі для опису модуля пам'яті використовується параметри *addr_width* та *data_width*, які задають розрядність шин адреси та даних відповідно. Кількість чарунок в блоці пам'яті в цьому випадку визначається як $2^{addr\_width}$, а їх розрядність дорівнює *data_width.*

```
generic (addr_width: natural:= 5;
data_width: natural:=8);
```

```
port (addr: in integer range 0 to
2**addr_width - 1;
     data_in: in std_logic_vector (data_width-
1 downto 0)
     data_out: out std_logic_vector
(data_width-1 downto 0)
);
 type mem is array (2**addr_width-1 downto 0)
 of std_logic_vector (7 downto 0);
```

### 3.4.1. Опис постійних зап'ятовуючих пристроїв на мові VHDL

При описі постійних запам'ятовуючих пристроїв вміст чарунок необхідно визначати при написанні програми. Можливо використання декількох варіантів визначення вмісту пам'яті:

- створення константи або сигналу типу «масив»;
- використання оператора case;
- використання *.mif файлу та атрибутів синтезу.

З трьох варіантів два перші можуть бути реалізовані на мікросхемах ПЛІС будь-якого виробника, а третій можливий лише у пакеті Quartus II.

#### Визначення вмісту пам'яті за допомогою константи або масиву.

При використанні цього варіанту спочатку необхідно оголосити тип, що буде відповідати розміру блока пам'яті. Потім оголошується константа цього типу і визначається вміст всіх чарунок масиву.

Наприклад, оголосимо новий тип ROM, який являє собою масив з 8 чарунок, кожна з яких має розмір 8 біт. Потім визначимо константу Content типу ROM.

```
type ROM is array (0 to 7) of
std_logic_vector
 (7 downto 0);
constant Content: ROM := (
```

```
    0 => "00000001",
    1 => "00000010",
    2 => "00000011",
    3 => "00000100",
    4 => "00000101",
    5 => "00000110",
    6 => "00000111",
    7 => "00001000",
);
```

Для використання такої константи необхідно просто адресувати необхідну чарунку в масиві за допомогою вхідних даних з ліній адреси. При цьому необхідно, щоб адресний порт мав тип integer, або похідний з нього. Вихідний порт даних повинен мати такий самий тип, як і тип чарунки блоку пам'яті. Для наведеного вище приклада вихідний порт Data_out повинен мати тип std_logic_vector (7 downto 0). Доступ до вмісту пам'яті буде виглядати таким чином:

```
Data_out <= Content (Addr);
```

**Приклад 3.15.** Розглянемо приклад повного опису блоку **ПЗП з використанням константи**. Блок пам'яті, що відповідає цьому опису, показаний на рисунку 3.27.

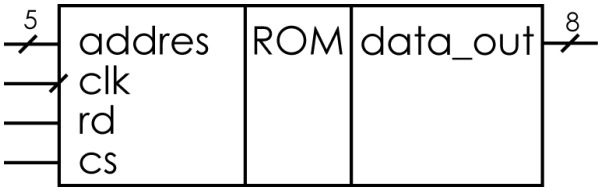



*Рисунок 3.27 – Блок пам'яті, описаний у прикладі 3.15*

Рядки 13 та 14 оголошують тип масив з 32 чарунок, кожна з яких містить 8 біт. Рядки з 15 по 23 задають значення чарунок масиву. Окремо визначаються значення тільки для перших 16 чарунок – рядки з 16 по 22. Всі інші чарунки заповнюються однаковим значенням «1111 1111» за допомогою слова `others` – рядок 23.

Робота модуля пам'яті описується за допомогою оператор апроцеса, у список ініціалізації якого входять сигнали `clk`, `cs`, `address` – тактовий, вибору кристала та адреси відповідно. Якщо сигнал `cs` дорівнює одиниці вихідні лінії ПЗП переходять у Z-стан (рядки 27 та 28). Якщо ж сигнал `cs` дорівнює нулю, то виходи переходять до робочого стану і відбувається робота мікросхеми.

Рядок 29 перевіряє наявність переднього фронту тактового сигналу `clk`.

Рядки 30-35 описують процес читання інформації з ПЗП. Якщо сигнал `rd` дорівнює одиниці дозволяється читання інформації, якщо ж він дорівнює нулю – вихідні лінії переходять у Z-стан (рядок 34). Для доступу до конкретної чарунки в модулі пам'яті використовуються рядки 31 та 32. Константа `content` має тип даних чарунки `std_logic_vec`tor, що відповідає типу вихідного сигналу `data_out`. Сигнал адреси в модулі пам'яті також тип `std_logic_vector`, тому для адресації чарунки в масиві `content` необхідне перетворення типу `std_logic_vector` до типу `integer`, що виконується за допомогою конструкції `to_integer (unsigned (address)`. В цій конструкції спочатку сигнал типу `std_logic_vector` перетворюється до сигналу типу `unsigned`, а вже потім – до типу `integer`. Більш докладно про перетворення типів див. параграф 2.3.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ROM is
port (clk    : in  std_logic;
   cs      : in  std_logic;
   rd      : in  std_logic;
   address : in  std_logic_vector(4 downto 0);
   data_out: out std_logic_vector(7 downto 0));
end ROM;

architecture behav of ROM is
```

```
   type ROM_array is array (0 to 31)
   of std_logic_vector(7 downto 0);

constant content: ROM_array := (
   0 => "00000001",
   1 => "00000010",
   2 => "00000011",
   . . .
   12 => "00001101",
   13 => "00001110",
   14 => "00001111",
   others => "11111111");
begin
   process(clk, cs, address)
   begin
      if(cs = '1' ) then
         data_out <= "ZZZZZZZZ";
         elsif (clk'event and clk = '1') then
         if rd = '1' then
            data_out <= content(to_integer
                  (unsigned (address)));
         else
            data_out <= "ZZZZZZZZ";
         end if;
      end if;
   end process;
end behav;
```

Результати моделювання наведені на рисунку 3.28.

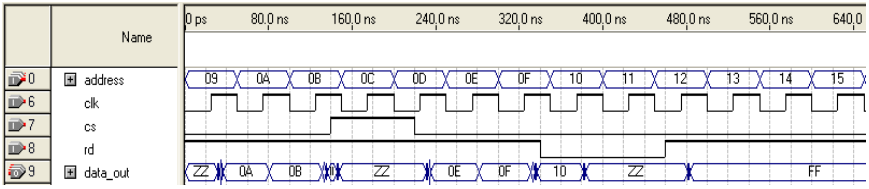



*Рисунок 3.28 – Результати моделювання блоку пам'яті*

### Визначення вмісту пам'яті за допомогою оператора case.

При використанні оператора case необхідно оголосити порти, а визначення вмісту пам'яті відбувається в архітектурному тілі. Кожному значенню адреси ставиться у відповідність вміст цієї чарунки пам'яті. Конструкція буде мати такий вигляд:

```
when адреса => вихідний_порт <=
                    вміст_чарунки;
```

**Приклад 3.16.** В якості прикладу розглянемо **опис модуля пам'яті об'ємом 256×6**. Адрес описується восьмирозрядним вектором типу std_logic, вихід даних – шестирозрядним вектором типу std_logic. За допомогою оператора case окремо визначається вміст перших десяти чарунок блоку пам'яті, всі інші визначаються разом за допомогою операторів **when others**.

```
library ieee;
use ieee.std_logic_1164.all;

entity mem is
port (
     clock    : in  std_logic;
     address  : in  std_logic_vector (7 downto
0);
     data_out : out std_logic_vector (5 downto
0));
end mem;

architecture rtl of mem is
begin
process (clock)
begin
    if rising_edge (clock) then
   case address is
      when "00000000" => data_out <= "000111";
      when "00000001" => data_out <= "000110";
```

```
        when "00000010" => data_out <= "000010";
        when "00000011" => data_out <= "100000";
        when "00000100" => data_out <= "100010";
        when "00000101" => data_out <= "001110";
        when "00000110" => data_out <= "111100";
        when "00000111" => data_out <= "110111";
        when "00001000" => data_out <= "111000";
        when "00001001" => data_out <= "100110";
        when others => data_out <= "101111";
    end case;
     end if;
end process;
end rtl;
```

Результати моделювання блоку пам'яті з прикладу 3.16 наведені на рисунку 3.29.

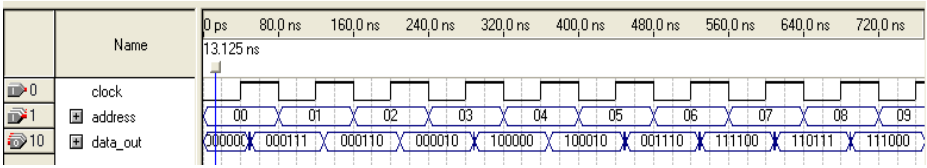



*Рисунок 3.29 – Моделювання блоку пам'яті з прикладу 3.16*

## Визначення вмісту пам'яті за допомогою mif файлу.

Цей варіант визначення вмісту пам'яті працює лише з продукцією компанії Altera, але й дозволяє дуже швидко змінювати вміст пам'яті. Для опису необхідно підключити бібліотеку атрибутів синтезу компанії Altera altera_syn_attributes та використовувати атрибут ram_init_file. За замовчуванням бібліотека знаходиться у папці *папка_quartus*\libraries\vhdl\altera. Цей атрибут задає mif файл, який містить інформацію про початковий вміст пам'яті.

Для використання цього атрибута необхідно декларувати атрибут синтеза, як рядковий тип:

```
attribute ram_init_file : string;
```

Створити зв'язок атрибута ram_init_file з сигналом, що описує блок пам'яті. Значення атрибута повинно співпадати з іменем *.mif файлу:

```
attribute ram_init_file of rom : signal is "mem.mif";
```

**Приклад 3.17.** В прикладі розглядається опис блоку пам'яті об'ємом 256×8. Рядки 1-4 описують бібліотеки та модулі. Видно, що у рядках 1 та 4 описується бібліотека атрибутів синтезу.

Рядки 5-9 описують інтерфейсну частину модуля.

У рядках 11 та 12 вводяться тип mem_t та сигнал rom цього типу, що описують модуль пам'яті.

В рядку 13 задається атрибут ram_init_file типу рядок, а у рядку 14 цей атрибут пов'язується з сигналом rom і робиться посилання на файл mem.mif, в якому наведений вміст модуля пам'яті.

Використання модуля пам'яті описане в рядку 19 і являє собою лише виведення на вихідний порт вмісту чарунки, що визначається адресним сигналом.

```
1  library ieee, altera;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use altera.altera_syn_attributes.all;
5  entity mem is
6   port (clk: in std_logic;
7    addr: in natural range 0 to 255;
8    q: out std_logic_vector (7 downto 0));
9  end entity;
10 architecture rtl of mem is
11  type mem_t is array (255 downto 0) of
                std_logic_vector(7 downto 0);
12 signal rom: mem_t;
13 attribute ram_init_file: string;
14  attribute ram_init_file of rom: signal is
                "mem.mif";
15 begin
```

```
16 process(clk)
17 begin
18 if(rising_edge(clk)) then
19  q <= rom(addr);
20 end if;
21 end process;
22 end rtl;
```

Вміст mif файлу визначається за допомогою редактора mif файлів, який описаний в параграфі 5.3.1. Для наведеного прикладу вікно з вмістом пам'яті показане на рисунку 3.30, а результати модулювання – на рисунку 3.31.

mem.mif

| Addr | +0 | +1 | +2 | +3 | +4 | +5 | +6 | +7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 |
| 8 | 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 |
| 16 | 32 | 34 | 36 | 38 | 40 | 42 | 44 | 46 |
| 24 | 48 | 50 | 52 | 54 | 56 | 58 | 60 | 62 |
| 32 | 64 | 66 | 68 | 70 | 72 | 74 | 76 | 78 |
| 40 | 80 | 82 | 84 | 86 | 88 | 90 | 92 | 94 |
| 48 | 96 | 98 | 100 | 102 | 104 | 106 | 108 | 110 |
| 56 | 112 | 114 | 116 | 118 | 120 | 122 | 124 | 126 |
| 64 | 128 | 130 | 132 | 134 | 136 | 138 | 140 | 142 |
| 72 | 144 | 146 | 148 | 150 | 152 | 154 | 156 | 158 |
| 80 | 160 | 162 | 164 | 166 | 168 | 170 | 172 | 174 |
| 88 | 176 | 178 | 180 | 182 | 184 | 186 | 188 | 190 |
| 96 | 192 | 194 | 196 | 198 | 200 | 202 | 204 | 206 |

*Рисунок 3.30 – Вміст модуля пам'яті*

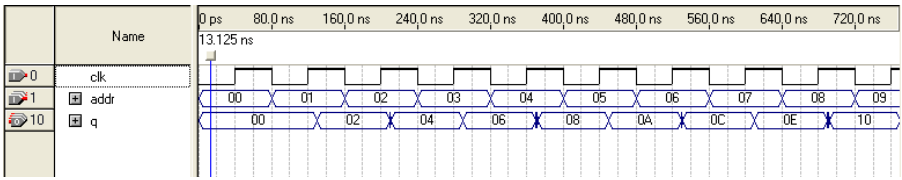



*Рисунок 3.31 – Часові діаграми роботи модуля пам'яті з прикладу 3.17*

## 3.4.2. Опис оперативних запам'ятовуючих пристроїв

Опис оперативних запам'ятовуючих пристроїв (ОЗП) відрізняється від опису постійних запам'ятовуючих пристроїв лише тим, що в ОЗП можна проводити запис. За необхідністю визначення початкового вмісту пам'яті в ОЗП можна проводити так само як і в ПЗП.

**Приклад 3.18. Синхронний ОЗП**, показаний на рисунку 3.32. Його робота описується таблицею 3.4.

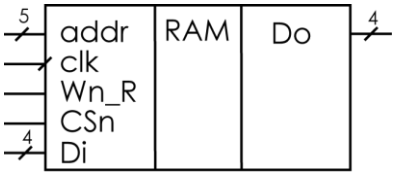



*Рисунок 3.32 – Блок пам'яті, описаний у прикладі 3.18*

Як і в попередніх прикладах для роботи з пам'яттю визначаємо новий тип як масив, що має розміри необхідного модуля пам'яті. Більш докладно про тип внутрішнього сигналу та портів див. приклад 3.15.

*Таблиця 3.4 – Таблиця дійсності ОЗП*

| Wn_R | CSn | Do[3..0] | Режим роботи |
|---|---|---|---|
| 0 | 0 | ZZZZ | Запис |
| 1 | 0 | Вихідні дані | Читання |
| × | 1 | ZZZZ | Збереження інформації |

```
library ieee;
    use ieee.std_logic_1164.all;
    use ieee.std_logic_unsigned.all;
    use ieee.numeric_std.all;
```

```
    entity mem is
      port (clk  : in  std_logic;
            Wn_R : in  std_logic;
            CSn  : in  std_logic;
            addr : in  std_logic_vector(4 downto
  0);
            Di   : in  std_logic_vector(3 downto
  0);
            Do   : out std_logic_vector(3 downto
  0));
    end mem;

    architecture syn of mem is
      type ram_type is array (31 downto 0) of
  std_logic_vector (3 downto 0);
      signal RAM : ram_type;
    begin
      process (clk, CSn)
      begin
        if CSn = '0'  then
       if (clk'event and clk = '1') then
          if (Wn_R = '0') then
             RAM(to_integer(unsigned(addr))) <=
  Di;
             Do <= "ZZZZ";
          else Do <=
  RAM(to_integer(unsigned(addr)));
          end if;
       end if;
      else
       Do <= "ZZZZ";
        end if;
      end process;
end syn;
```

Результати роботи стимулятора модуля пам'яті зображені на рисунку 3.33. Тут необхідно звернути увагу на те, що початковий вміст всіх чарунок пам'яті дорівнює нулю. Це видно під час читання чарунок з номерами 6-8 значення, що виводиться на вихід дорівнює нулю.

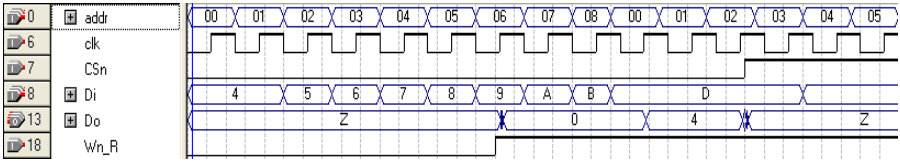



*Рисунок 3.33 – Часові діаграми роботи ОЗП*

**Приклад 3.19. Пам'ять з двонаправленим портом даних.** В якості прикладу будемо використовувати пам'ять з такою ж самою логікою роботи та об'ємом, як була описана вище. Але в цьому прикладі два порти – вхідний Di та вихідний Do будуть представлені одним двонаправленим портом DIO.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity bidirmem is
  port (clk  : in    std_logic;
    Wn_R : in   std_logic;
    CSn  : in   std_logic;
    addr : in   std_logic_vector(4 downto 0);
    DIO : inout std_logic_vector(3 downto 0));
end bidirmem;

architecture syn of bidirmem is
  type ram_type is array (31 downto 0) of
     std_logic_vector (3 downto 0);
  signal RAM : ram_type;
begin
```

```
  process (clk, CSn)
  begin
    if CSn = '0'  then
      if (clk'event and clk = '1') then
        if (Wn_R = '0') then
          RAM(to_integer(unsigned(addr))) <=
DIO;
        else
          DIO                              <=
RAM(to_integer(unsigned(addr)));
        end if;
      end if;
    else
      DIO <= "ZZZZ";
    end if;
  end process;
 end syn;
```

Результат симуляції пам'яті зображений на рисунку 3.34. Перші три такти частоти clk інформація записується в пам'ять, потім один такт пам'ять переводиться в третій стан сигналом CSn. Три останні такти виконується читання записаної інформації з пам'яті. Таким чином перші три такти інформацію на виводах DIO генерує зовнішнє джерело сигналу, а потім це джерело відключається і інформація виводиться з мікросхеми.

В результаті симуляції з'явиться додаткова група вихідних сигналів, які називаються DIO~result. На виводах DIO відображені дані, які генерує зовнішнє джерело сигналу – це перші три такти частоти clk. Потім зовнішнє джерело сигналу відключається, що зображено значеннями ZZZZ на виводах DIO. На виводах DIO~result – реальне значення, яке існує на виводах мікросхеми. Воно може керуватись як зовнішнім так і внутрішнім джерелом в залежності від режиму роботи – вводу або виводу даних.

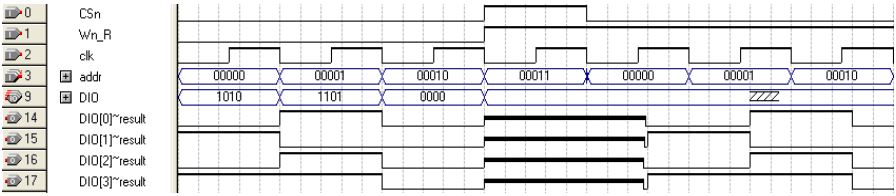



*Рисунок 3.34 – Часові діаграми пам'яті з двонаправленим портом даних*

## 3.4.3. Опис двохпортової пам'яті

**Приклад 3.20.** Для прикладу двохпортової пам'яті візьмемо так звану дійсно двохпортову пам'ять з одним тактовим сигналом (True Dual-Port RAM with Single Clock). Це означає що блок пам'яті (рисунок 3.35) містить два незалежних порти доступу до спільної пам'яті – A і B та один загальний тактовий сигнал clk. Поведінка двохпортового модуля пам'яті визначається мікросхемою ПЛІС, в якій реалізовано проект. Подальші результати моделювання показані для мікросхем сімейств Cyclone II і вбудованого блоку пам'яті M9K.

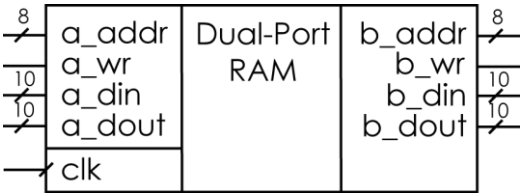



*Рисунок 3.35 – Модуль двохпортової пам'яті*

Особливістю двохпортової пам'яті є необхідність реалізації одночасного доступу до вмісту пам'яті від двох портів. При цьому можливо декілька варіантів роботи пам'яті:

– запис або читання портами різних чарунок в модулі пам'яті. В цьому випадку можна сприймати двохпортову пам'ять як два різних модулі пам'яті, які не впливають один на одний.

- Читання портами однієї і тієї ж чарунки в модулі пам'яті. Ситуація буде аналогічна попередній. Вплив портів відсутній.
- Запис інформації в чарунку одним портом і читання з цієї ж чарунки іншим портом. Такий режим називають читання під час запису (Read-During-Write Mode). Поведінка блоку пам'яті в цьому випадку визначається можливостями мікросхеми ПЛІС, що використовується для проекту. Тому розглянемо два найбільш вірогідних варіанти поведінки мікросхем:
    - Старі дані (Old Data) – при читанні на виході відображаються та дані, які були записані в чарунку пам'яті. Нові дані записуються в чарунку і на виході порта, що читає дані, з'являться в наступному такті.
    - Не важливо (Don't Care) – на виході в режимі симуляції буде показаний сигнал «Don't Care», тобто якесь невідоме значення.
- Одночасний запис двома портами інформації в одну чарунку пам'яті. Така ситуація призводить до конфлікту, тому її результат неможливо передбачити. Оскільки блок пам'яті M9K не містить логіки, що передбачає вирішення таких конфліктів, то її необхідно реалізовувати окремо в логічних елементах мікросхеми ПЛІС.

Для прикладів використаємо модуль пам'яті об'ємом один кілобайт. Модуль буде параметризованим, що дозволить легко змінювати його об'єм. При описі портів двохпортової пам'яті описуємо обидва порти. Виводи порту A будуть містити букви «a_» перед навзою, а потру B – «b_». В іншому порти однакові. Для тактування використовується загальний тактовий сигнал clk.

Як і при описі ОЗП і ПЗП оголошуємо сигнал типу масив, що має розмір 1K×8. Він буде представляти загальний блок пам'яті.

Архітектурна частина програми включає в себе два процеси, кожен з яких описує роботу окремого порту. Вони однакові, тому розглянемо роботу лише одного оператора процесу – для порту A.

При надходженні переднього фронту тактової частоти проводиться аналіз сигналу запису a_wr. Якщо він дорівнює одиниці, то виконується запис інформації з порту a_din в чарунку. На вихід a_dout виводиться інформація з вхідного порту. Коли ж сигнал запису a_wr дорівнює нулю на вихід a_dout виводиться вміст чарунки, що адресується портом a_addr.

Як видно, при описі окремого порту A жодним чином не описується робота порту B. Тобто жодних конфліктних ситуацій не описується і поведінка двохпортової пам'яті цілком буде визначатись архітектурою вбудованого модуля пам'яті M9K.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity dualport is
generic (
    DATA    : integer := 8;
    ADDR    : integer := 10
);
port (
    clk    : in  std_logic;
    -- Port A
    a_wr   : in  std_logic;
    a_addr : in  std_logic_vector(ADDR-1
        downto 0);
    a_din  : in  std_logic_vector(DATA-1
        downto 0);
    a_dout : out std_logic_vector(DATA-1
        downto 0);

    -- Port B
    b_wr   : in  std_logic;
    b_addr : in  std_logic_vector(ADDR-1
        downto 0);
    b_din  : in  std_logic_vector(DATA-1
        downto 0);
```

```
    b_dout : out std_logic_vector(DATA-1
        downto 0)
);
end dualport;

architecture rtl of dualport is
    type mem_type is array ( (2**ADDR)-1
        downto 0 ) of std_logic_vector(DATA-1
        downto 0);
    signal mem : mem_type;
begin

-- Port A
process(clk)
begin
  if(clk'event and clk='1') then
    if(a_wr='1') then
       mem(to_integer(unsigned(a_addr))) <=
a_din;
       a_dout <= a_din;
    else
       a_dout <=
mem(to_integer(unsigned(a_addr)));
    end if;
  end if;
end process;

-- Port B
process(clk)
begin
  if(clk'event and clk='1') then
    if(b_wr='1') then
      mem(to_integer(unsigned(b_addr))) <=
b_din;
      b_dout <= b_din;
    else
```

```
        b_dout <=
mem(to_integer(unsigned(b_addr)));
    end if;
  end if;
end process;

end rtl;
```

Результати симуляції двохпортової пам'яті показані на рисунку 3.36. На ньому розглядається одночасна робота двох портів A і B з однією чарункою 2BC. Попередня інформація в чарунці – 00.

В першому такті виконується запис числа 5A в чарунку з порту A і читання чарунки з порту B. На вихідний порт a_dout через деякий час надходить нове записане число 5A, а на вихідному порті b_dout – стара інформація 00.

В другому такті виконується читання двома портами інформації з однієї чарунки і на обох вихідних портах однакова інформація – 5A.

В третьому такті порт A читає дані, а порт B – записує число 02. На виході порту A – теж саме число 5A, а порту B – нова інформація – 02.

Четвертий такт – читання з чарунки двома портами. Ситуація подібна до описаної в другому такті – на обох вихідних портах однакова інформація – 02.

П'ятий такт – одночасний запис різної інформації в одну чарунку з двох портів. Порт A записує число 5E, порт B – число 04. В цьому такті на виходах портів та інформація, яку вони записують в чарунку – на виході a_dout – число 5E, на виході b_dout – число 04. Така ситуація повністю відповідає програмі, що описує блок двохпортового ОЗП.

Шостий такт – одночасне читання двома портами цієї ж чарунки. На вихідних портах невизначена інформація. Тобто реальна ситуація буде визначатись часовими співвідношеннями між вхідними сигналами обох портів і кожного разу буде різною. Тому для запобігання таким конфліктним ситуаціям необхідно передбачити окрему схему арбітражу.

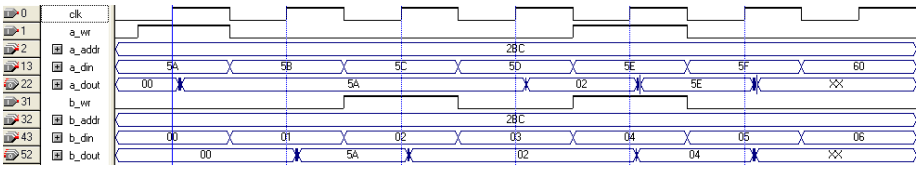



*Рисунок 3.36 – Часові діаграми роботи двохпортової пам'яті*

Як було вказано вище, для арбітражу під час одночасного запису різними портами інформації однієї чарунки необхідно створити додаткову схему. У наведеному нижче прикладі така схема аналізує вхідні адреси з обох портів і, якщо вони однакові, дозволяє запис інформації з порту A. Інформація з порту B ігнорується. Нижче наведена лише частина програми, що описує порт B. Інша частина програми не зазнала змін.

```
  -- Port B
process(clk)
begin
  if(clk'event and clk='1') then
    if(b_wr='1') then
      if a_addr = b_addr and a_wr='1' then
       b_dout <=
mem(to_integer(unsigned(b_addr)));
      else
       mem(to_integer(unsigned(b_addr))) <=
b_din;
       b_dout <= b_din;
      end if;
    else
       b_dout <=
mem(to_integer(unsigned(b_addr)));
    end if;
  end if;
end process;
```

В результаті компіляції такої програми в мікросхемі ПЛІС з'явиться компаратор, що буде порівнювати адреси портів A і B. Така схема для 10-розрядних адрес потребує більше 24 тисяч логічних

елементів у мікросхемі Cyclone III. В той же час схема без арбітражу адрес зовсім не потребувала логічних елементів для реалізації – вся схема була реалізована за рахунок вбудованого блоку пам'яті.

На рисунку 3.37 наведені часові діаграми роботи, з якого видно, що у шостому такті під час читання інформації на всіх вихідних портах буде однакова інформація 5E.

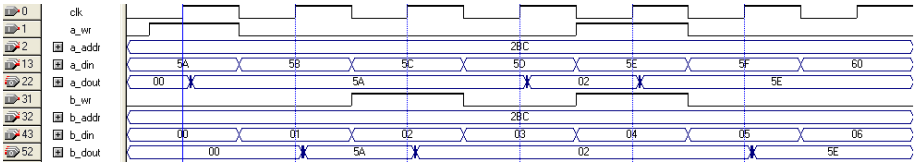



*Рисунок 3.37 – Часові діаграми роботи двохпортової пам'яті зі схемою арбітражу одночасного запису*

## *3.5. Пакети, процедури та функції*

### 3.5.1. Пакети

Пакет дозволяє згрупувати описи блоків, типів, підпрограм, які часто використовуються в єдину сукупність, яка може далі багаторазово використовуватися в різних проектах. Пакет може використовуватися як у різних частинах однієї моделі, так і в різних моделях.

Пакет (**package**) – це блок, який може включати безліч декларацій:

- типів,
- підтипів,
- констант,
- компонентів,
- процедур,
- функцій,
- файлів.

Опис пакета складається з декларативної частини і тіла пакета.

Опис декларативної частини має наступний синтаксис:

```
package ім'я_пакета is
   Розділ декларацій:
      декларація підпрограм;
      декларація типів та підтипів;
      декларація констант;
      декларація сигналів;
            декларація компонентів
end ім'я_пакета;
```

Опис тіла пакета має наступний вигляд:

```
package body ім'я_пакета is
            декларація підпрограм;
      декларація типів та підтипів;
      декларація констант;
end ім'я_пакета;
```

Декларація тіла пакета дуже подібна до декларації самого пакета за виключенням зарезервованого слова **body** після слова **package**. Тіло пакета повинно містити декларацію підпрограм, якщо вони оголошені у декларації пакета. Але необхідно пам'ятати, що ці декларації повинні мати таку ж саму кількість параметрів, їх тип, як і оголошені в декларації пакета. Крім цього тіло пакета може містити реалізацію наведених підпрограм. При описі в тілі пакета типів, підтипів, констант важливо враховувати, що ці об'єкти не будуть досяжні за межами тіла пакета.

**Приклад – 3.21.** В якості прикладу розглянемо пакет auxiliary, який описує два додаткових типи MUX_input та operation_set, підтип MUX_Address, функцію Compute_Address та константу Deferred_Con. При декларації типу описуються всі ці сутності, а при описі тіла пакету описується лише реалізацію функції та значення константи. Для використання цього пакету його необхідно під'єднати до проекту як бібліотеку.

```
library ieee;
use ieee.std_logic_1164.all;

package auxiliary is
```

```
    type MUX_input is array (integer range<>)
            of std_logic_vector (0 to 7);
  type operation_set is (SHIFT_LEFT, ADD);
  subtype MUX_Address is positive;
    function Compute_Address (IN1 : MUX_input)
            return MUX_address;
  constant Deferred_Con : integer;
end auxiliary;
package body auxiliary is
  function Compute_Address (IN1 : MUX_input)
return MUX_address is
  begin
    .............
  end;
  constant Deferred_Con : integer := 177;
end package body AUXILIARY;
```

## 3.5.2. Процедури і функції

Процедури і функції в мові VHDL виконують таку ж саму функцію, що і в інших мовах програмування – вони описують частину алгоритму роботи схеми, яка може потім використовуватися в інших програмних модулях.

Загальний вигляд опису **процедури**

```
procedure ім'я_процедури (параметри) is
    розділ декларацій
begin
    послідовні оператори
end procedure ім'я_процедури;
```

Список параметрів, які передаються до процедури може містити сигнали (**signal**), змінні (**variable**) та константи (**constant**). Для параметрів необхідно вказувати напрямок передачі інформації – вхід (**in**), вихід (**out**), двонаправлений сигнал (**inout**). За замовчуванням вхідні параметри вважаються константами, вихідні та двонаправлені – змінними. Не рекомендується використовувати процедури без списка

параметрів. В тілі процедури можуть записуватись тільки послідовні оператори.

**Приклад 3.22.** Розглянемо процедуру, яка описана в тілі процесу.

```
process (a1, a2, a3)
23 variable b1, b2, b3: integer;

24 procedure sort (variable x1, x2: inout
   integer)
25 is
26   variable t: integer;
27 begin
28   if x1>x2 then
29        return;
30   else
31        t:=x1; x1:= x2; x2:= t;
32   end if;
33 end procedure;

34 begin
35 b1:= a1; b2:= a2; b3:= a3;
36 sort (b2, b3);
37 sort (b1, b2);
38 sort (b2, b3);
39 c<= b1; c2<= b2; c3<= b3;
40 end process;
```

Рядки 1 і 2 оголошують процес зі списком ініціалізації (*a1, a2, a3*) та змінні (*b1, b2, b3*) цілого типу, які використовуються в процесі та вкладених в нього процедурах. Рядки 3 і 4 описують процедуру *sort*, в яку передається дві двонаправлені змінні цілого типу (*x1, x2*). Рядок 5 оголошує внутрішню змінну процедури (*t*). Рядки 7 і 8 описують варіант дострокового переривання роботи процедури – команду *return*. Тепер розглянемо приклад використання процедури. Це рядки 15-17. В процедуру можна передавати параметри тільки типу **inout**. При використанні процедури використовується позиційна відповідність.

**Приклад 3.23**. Передавання до процедури не всіх параметрів. Спочатку опишемо процедуру *and4*. Список параметрів процедури містить змінні та сигнал. Напрямок передачі для змінних – вхід, для сигналу – вихід. В цьому прикладі *y* – вихідний сигнал.

```
procedure and4
     (variable x1, x2, x3, x4: in bit:= '1';
     signal y: out bit) is
begin
     y<= x1 and x2 and x3 and x4;
end procedure;
```

Виклик процедури з неповним списком параметрів буде виглядати наступним чином:

```
 and4 (a1, a2, a3, open, b);
```

Ключеве слово **open** вказує на те, що замість нього необхідно встановити значення за замовчуванням, тобто одиницю.

Загальний вигляд опису **функції**:

```
function ім'я_функції
     список параметрів
return тип_параметра is
     розділ декларацій
begin
     послідовні оператори
end function ім'я_функції;
```

Ім'я функції може бути як звичайним іменем так і символом операції. Наприклад, «+», як буде показано нижче. Список параметрів має таку саме мету як і в процедурі, але він не може включати режими **out** і **inout**. Після слова **return** вказується тип параметра, який повертає функція. При використанні функцій слід враховувати наступні особливості:

- змінна не може виступати в якості параметра функції;
- виконання функції повинно закінчуватись оператором **return**;
- функція, на відмінність від процедури, може повертати лише один параметр;

– параметр, який повертає функція можна використовувати як операнд у виразах.

В якості прикладу розглянемо стандартну функцію з бібліотеки *numeric_std*. Це функція додатку *«+»*.

```
function "+" (l: unsigned; r: natural)
      return unsigned is
begin
      return l + to_unsigned(r, l'length);
end "+";
```

Як видно з опису в якості імені використовується символ «+». В якості вхідних параметрів функція отримує дві змінні різного типу: *unsigned* та *natural*. Оскільки функція повертає значення типу *unsigned*, то для отримання двох операндів одного типу необхідно виконати перетворення типів для змінної типу *natural*.

**Перезавантаження процедур та функцій**

Можлива ситуація, коли в програмі описано декілька процедур або функцій з однаковими іменами. В цьому випадку буде виконуватись перезавантаження (overloading) функції. Тобто використання найкращої з точки зору сумісності типів даних та кількості параметрів підпрограми. Прикладом перезавантаження функцій є використання операції додавання з однаковим позначенням «+». При використанні такої операції з типом *integer* використовується операція додавання з бібліотеки *std*, а при використанні типів *signed* та *unsigned* використовується бібліотека *numeric_std*. В такій ситуації компілятор обирає необхідну функцію.

## 3.6. Створення файлу на мові опису апаратури в пакеті Quartus II

Створення проекту, загальні правила роботи в пакеті Quartus II описані в розділі 4. Тут ми зупинимось лише на тому, яким чином створити файл на мові опису апаратури. Описані методи підходять для

створення файлу на будь-якій мові опису апаратури в пакеті Quartus II.

Для створення файлу на мові VHDL необхідно з меню **File → New**… вибрати пункт **Design Files** і вибрати пункт **VHDL File** (рисунок 3.38).

В результаті буде відкрите вікно з пустим аркушем, в якому можна набирати тексти на мові опису апаратури.

Крім набору тексту програм в редакторі є можливість вставляти готові шаблони (template). Для цього є декілька шляхів (рисунок 3.39):

- вибір пункту меню **Edit → Insert Template…**
- вібір пункту **Insert Template…** з контекстного меню редактора;
- вставка шаблона за допомогою кнопки  .

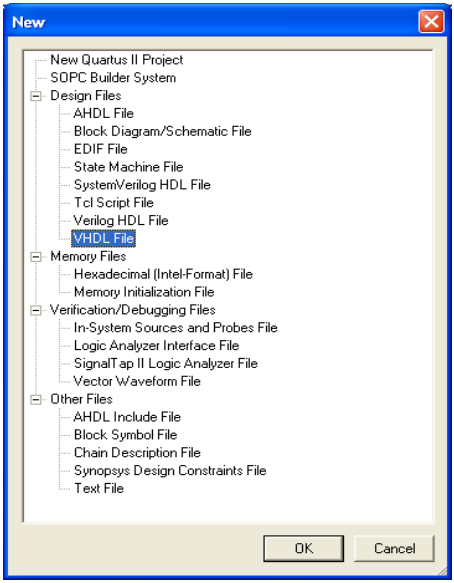



*Рисунок 3.38 – Створення нового файлу*

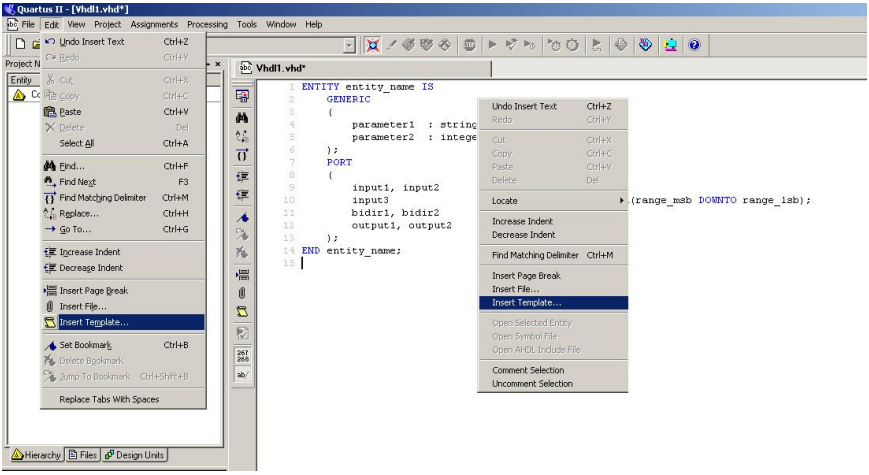



*Рисунок 3.39 - Вставка шаблонів програм*

В результаті буде відкритий діалог вставки шаблона, показаний на рисунку 3.40.

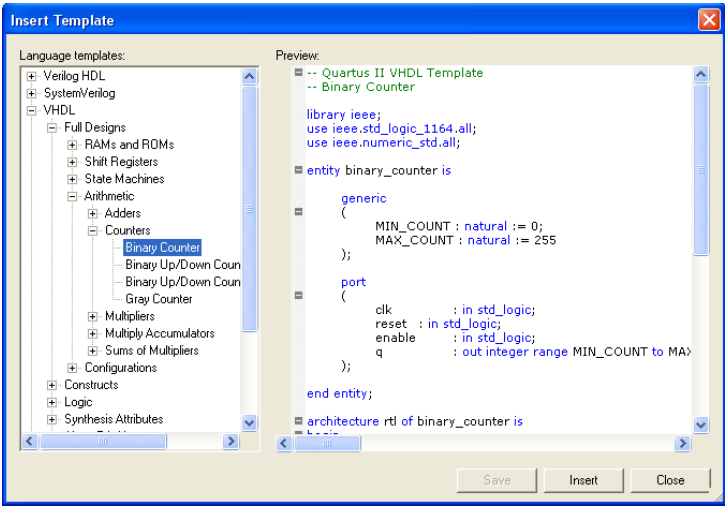



*Рисунок 3.40 – Діалог вставки шаблону*

Як видно з рисунку розробник може використовувати або повністю готові шаблони простих модулів (**Full Design**) або часткові шаблони, які описують окремі конструкції мови VHDL (**Constructs**) або окремі модулі (**Logic**). Крім того можна додавати атрибути синтезу, що описані в бібліотеці *altera.altera_syn_attributes* (**Synthesis Primitives**), та примітиви буферів та тригерів компанії Altera (**Altera Primitives**).

Часто виникає необхідність створення символу з тексту програми на мові VHDL. Для цього необхідно відкрити файл на мові опису апаратури і обрати пункт меню **File → Create/Update → Create Symbol Files for Current File**. Після чого буде створений символ, який можна використовувати в графічному редакторі. При використанні в графічному редакторі параметризованих блоків, в яких була використана конструкція **generic ()** можна змінювати параметри не редактуючи текст програми.

В якості прикладу розглянемо роботу з символом паралельного регістру, що був описаний в параграфі 3.2. Після створення для нього символу, як було описано вище, вставимо цей символ у графічний файл пакету Quartus II. У контекстному меню оберемо пункт **Proporties** і закладку **Parameters** (рисунок 3.41).

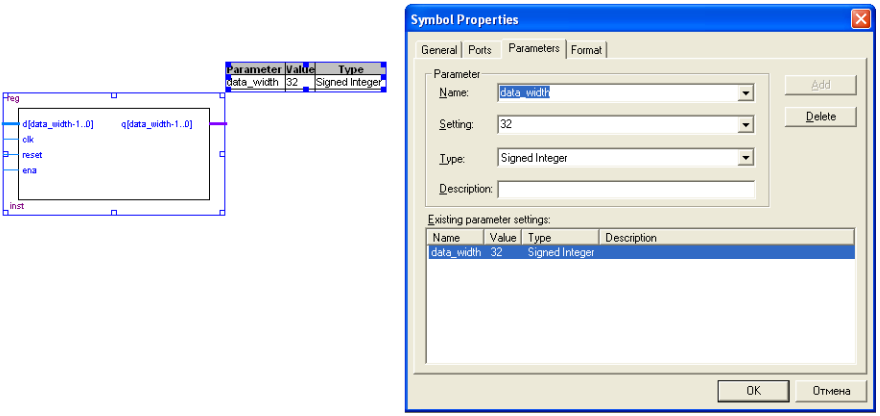



*Рисунок 3.41 – Зміна параметрів файлу*

# Література

1. Сергиенко А.М. VHDL для проектирования вычислительных устройств. – К.: ЧП «Корнейчук», ООО «ТИД»ДС», 2003. – 208 с.
2. Суворова Е.А., Шейнин Ю.Е. Проектирование цифровых схем на VHDL. – СПб.: БХВ-Петербург, 2003. – 576 с.
3. Уэйкерли Д.Ф.Проектирование цифровых устройств. В 2 томах. – М.: Постмаркет, 2002. – 1088 с.
4. Chu Pong P. RTL hardware design using VHDL. Wiley-Interscience, 2006. – 680 p.
5. Perry D.L. VHDL: Programming by Example. McGraw-Hill, 2002. – 480 p.
6. Quartus II Handbook. Version 9.1. Altera Corp., 2009. – 1820 p.

# Розділ 4

# Робота в пакеті Quartus II


- Знайомство з пакетом Quartus II ..... 166
- Створення графічного файлу в пакеті Quartus II..... 181
- Компіляція проекту..... 195
- Установки та призначення проекту..... 202
- Призначення виводів мікросхеми в Quartus II ..... 207
- Часовий аналіз в Quartus II..... 211
- Побудова часових діаграм проекту ..... 215
- Програмування та конфігурування в Quartus II ..... 225

# 4.1. Знайомство з пакетом Quartus II

## 4.1.1. Особливості роботи з пакетом Quartus II

Quartus II – це інтегроване середовище для розробки приоектів на ПЛІС фірми Altera. Цей пакет дозволяє проводити розробку систем будь-якої складності, включаючи системи на кристалі (system-on-a-programmable-chip SOPC) від вводу проекту до генерації конфігураційного файлу і програмування мікросхеми. Quartus II підтримує командну розробку (Incremental Compilation), відлагодження проекту на кристалі (SignalTap II), розміщення проекту на кристалі.

Існує декілька версій пакету Quartus II: вільно розповсюджувана версія Quartus II Web Edition, яку можна завантажити з сайту та комерційна версія пакету (Subscription Edition). Версія Web Edition підтримує всі ті можливості, що описані в цій книзі. Різницю між версіями пакету можна подивитись в [2].

До середини 2000-х років компанія Altera випускала пакет MAX+PLUS II, який на сьогоднішній день повністю заміщений пакетом Quartus II і нові сімейства мікросхем не підтримує. У книзі [1] наведений приклад роботи з цим пакетом, який фактично є перекладом Getting Started.

Слід відмітити декілька особливостей пакету Quartus II.

1. Пакет не допускає використання російських букв і пробілів, як в іменах файлів, так і в іменах каталогів.

2. Небажане маніпулювання із системною датою комп'ютера, оскільки пакет перевіряє дату створення та зміни проекту.

3. При створенні проекту необхідно пам'ятати, що при роботі пакет створює велику кількість файлів. Тому бажано кожний із проектів створювати в окремому каталозі.

Для вивчення пакету можна порекомендувати розділ Getting Started Tutorial довідки пакету Quartus II, книгу з пакету [3] та її частковий переклад на російську мову [5].

На сайті компанії Altera існує розділ з прикладами різноманітних проектів, які доступні для завантаження [4].

## 4.1.2. Цикл розробки проекту на ПЛІС

Цикл розробки проекту на базі ПЛІС показаний на рисунку 4.1.

Цикл розробки містить у собі декілька етапів:

Створення проекту. При створенні проекту можуть бути використані графічні (.gdf, .bdf) і текстові (.tdf, .vhd, .v) файли. Можливий також імпорт файлів з інших засобів проектування та використання спеціалізованих бібліотек.

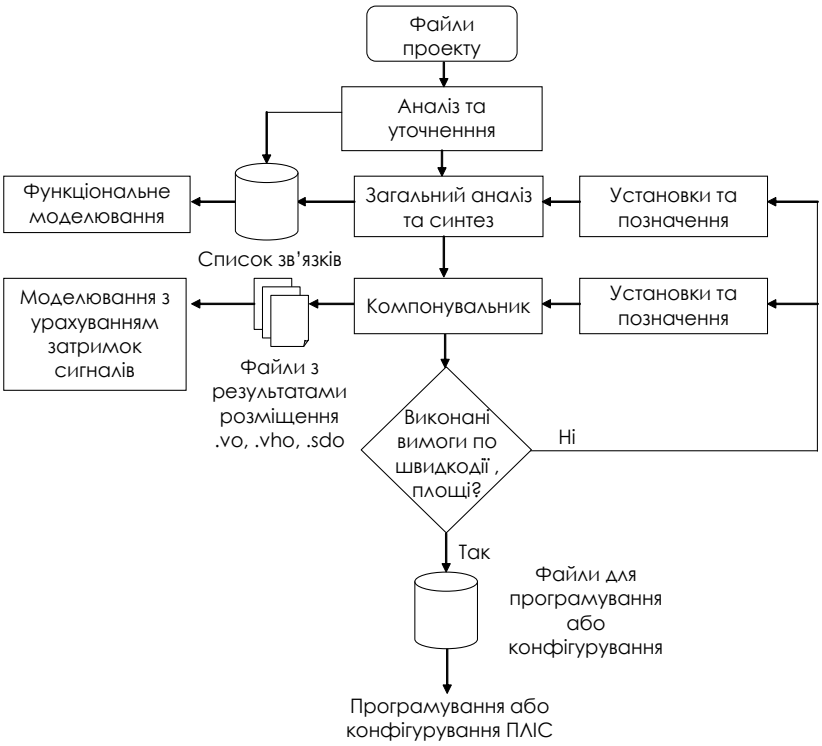



*Рисунок 4.1 – Цикл розробки проекту на базі ПЛІС в пакеті Quartus II*

У результаті аналізу та компіляції файлів проекту (**Analysis & Synthesis**) буде отриманий список зв'язків, що дозволяє провести функціональне моделювання проекту, тобто моделювання без урахування затримок сигналів всередині мікросхеми. Даний етап є дуже важливим, оскільки дозволяє проаналізувати працездатність алгоритму.

Далі компанувальник (**Fitter**) розміщує проект всередині мікросхеми. Після цього з'являється можливість отримати дані про затримки сигналів, максимальні тактові частоти. Ці дані дозволяють провести моделювання з урахуванням затримок сигналу (**Timing Analysis**), розрахувати спожиту потужність пристрою (**PowerPlay Power Analysis**) і т.д.

Якщо вимоги по використаній площі кристалу і швидкодії виконані, то генеруються файли для програмування або конфігурування мікросхеми ПЛІС (**Assembler (Generate programming file)**). Якщо ж вимоги не виконані, то робляться додаткові призначення та установки (**Settings & Assignments**, **Floorplan Location Assignments**), а компіляція виконується повторно.

В результаті роботи компілятора на різних етапах генерується декілька файлів, що містять списки зв’язків проекту (**Netlist**). Розглянемо їх детально, оскільки ці файли використовуються в подальшому для симуляції, часового аналізу та відлагодження проекту.

**Pre-synthesis netlist** – список зв’язків, який буде отриманий після аналізу проекту, перевірки синтаксису файлів та правильності з’єднань. До цього списку не застосовуються будь-які оптимізації і він містить всі введені користувачем імена.

**Post-synthesis netlist** – список зв’язків, який отримується після синтезу проекту. Його склад залежить від опцій оптимізації і в ньому можуть бути відсутні деякі сигнали, що були у pre-synthesis netlist.

**Post-fitting netlist** – список тих зв’язків, що залишись після оптимізації проекту та його розміщення на кристалі ПЛІС. Цей список містить найбільш реальну картину проекту і дозволяє найбільш точно проводити симуляцію та відлагодження.

При розробці проекту на основі ПЛІС можливе спільне використання декількох систем автоматичного проектування. При цьому сполучною ланкою для мікросхем ПЛІС фірми Altera буде пакет Quartus II.

Вигляд пакета Quartus II після запуску показаний на рисунку 4.2.

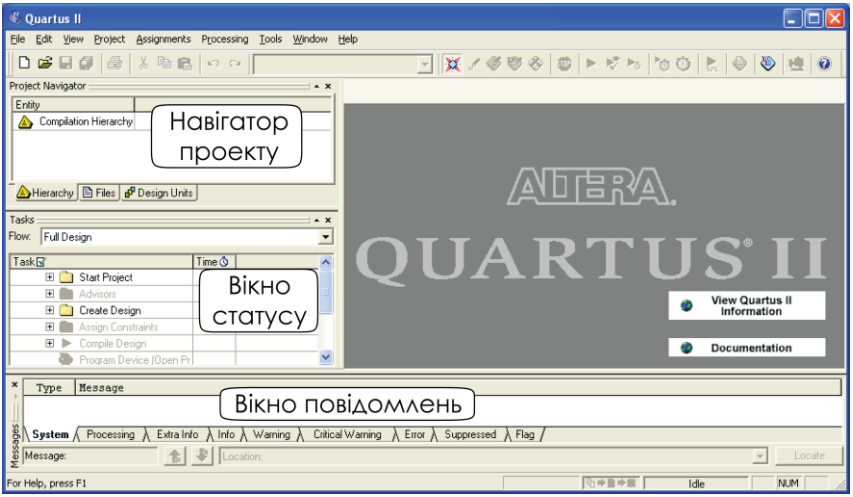



*Рисунок 4.2 – Вигляд пакета Quartus II після запуску*

Вікна в пакеті можуть переміщатися незалежно від основного інтерфейсу. Для цього необхідно скористатися кнопкою прив'язки вікна  . Для повернення вікна в основний інтерфейс необхідно також скористатися цією кнопкою (рисунок 4.3).

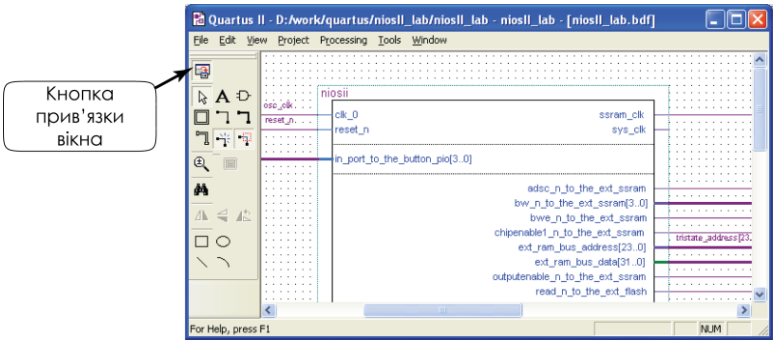



*Рисунок 4.3 – Прив'язка вікна*

Крім звичного інтерфейсу з кнопковими формами, Quartus II також підтримує спеціальну мову програмування – TCL (**Tool Command Language**), на якій можуть бути написані програми (скрипти), які дозволяють прискорити роботу з пакетом та визначення опцій проекту. Для відкриття вікна TCL скриптів (рисунок 4.4) необхідно вибрати наступні пункти меню **View** ⇒ **Utility** ⇒ **Windows** ⇒ **TCL console** (Alt+2).

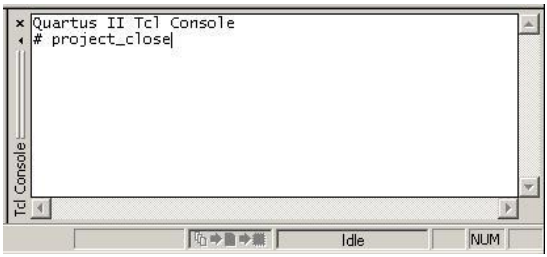



*Рисунок 4.4 – TCL console*

## 4.1.3. Створення проекту

При створенні проекту необхідно визначити кілька головних параметрів:

1. Каталог, у якому буде збережено проект.

2. Назву проекту.
3. Назву файлу верхнього рівня ієрархії. Зазвичай це ім'я збігається з назвою проекту.
4. Використовувану мікросхему.
5. Файли додаткових бібліотек.
6. Додаткове програмне забезпечення, яке використовується для симуляції та верифікації проекту.

Для створення нового проекту необхідно вибрати пункт меню **File ⇒ New Project Wizard...** (рисунок 4.5). При цьому буде запущений майстер, за допомогою якого можна буде визначити основні параметри проекту. Розглянемо послідовно призначення діалогів у вікнах майстра проекту. Перехід між вікнами виконується при натисканні на кнопку **Next >**.

**Увага!** Пункт меню **File ⇒ New**… за замовчуванням та кнопка створюють новий файл, а не новий проект!

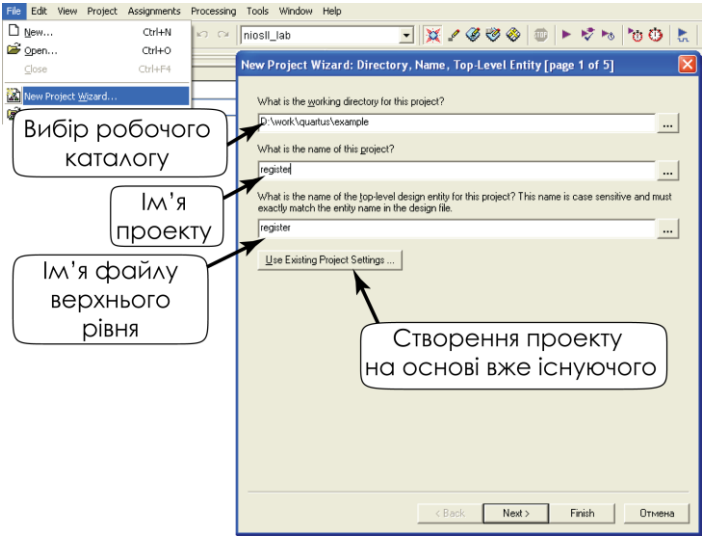



*Рисунок 4.5 – Вікно майстра створення проекту*

## Перше вікно майстра створення проекту

**What is the working directory for this project?** – вибір робочого каталогу проекту.

**What is the name of this project?** – визначення імені проекту. Це може бути будь-яке ім'я, але рекомендується використовувати ім'я файлу верхнього рівня.

**What is the name of the top–level design entity for this project?** – визначення імені файлу верхнього рівня. При введенні імені слід пам'ятати, що пакет чутливий до зміни регістру.

**Use Existing Project Settings...** – визначення параметрів проекту на основі існуючого проекту.

## Друге вікно майстра створення проекту

У цьому вікні задаються додаткові бібліотеки та файли користувача, які необхідно підключити до проекту (рисунок 4.6). До проекту можуть бути додані файли наступних типів: графічні (.bdf, .gdf), опис схем за допомогою мов опису апаратури (AHDL, VHDL, Verilog), а також файли типу EDIF. Необхідно пам'ятати, що файли, які знаходяться у робочому каталозі проекту, додавати не потрібно. Крім цього можливе додавання бібліотек користувача за допомогою кнопки **User Libraries...**

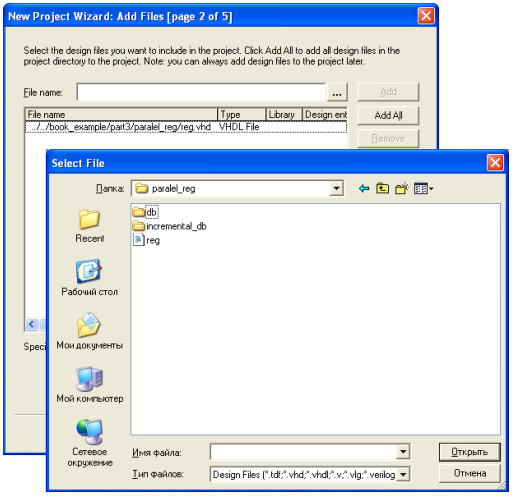



*Рисунок 4.6 – Додавання файлу до проекту*

Для додавання файлу до проекту необхідно натиснути на кнопку  , вибрати потрібний файл і натиснути кнопку "Открыть". Обраний файл з'явиться в рядку **"File name:"**. Після цього необхідно натиснути кнопку **"Add..."** і файл буде доданий до проекту.

## Третє вікно майстра створення проекту

В цьому вікні задаються сімейство мікросхем та параметри мікросхеми з даного сімейства (рисунок 4.7).

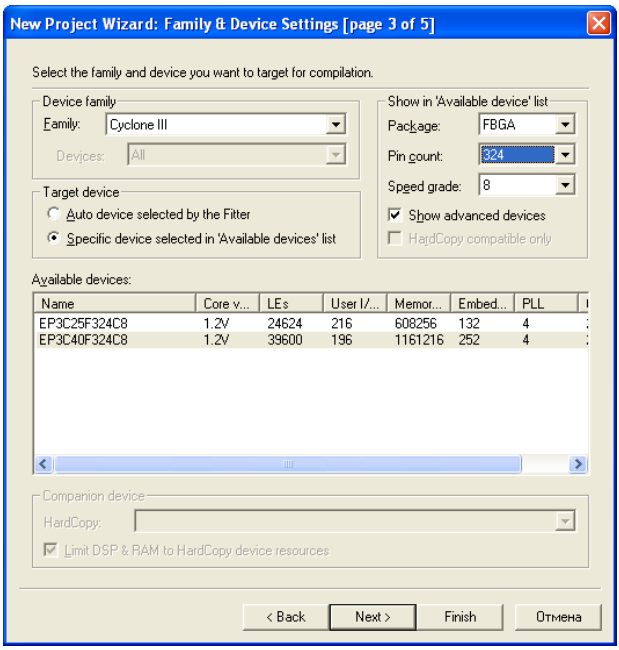



*Рисунок 4.7 – Вікно вибору мікросхеми*

**Family** – задається сімейство мікросхем.
**Target device** – вибір мікросхеми:

- **Auto device selected by the Fitter** – автоматичний вибір мікросхеми компанувальником пакету Quartus II.
- **Specific device selected in 'Available devices' list** – вибір конкретної мікросхеми зі списку доступних.

**Show in 'Available devices' list** – вибір параметрів мікросхем, за якими буде формуватися список доступних мікросхем. Користувачу доступні такі фільтри відображення списку мікросхем:

- **Package** – вибір типу корпусу мікросхеми.
- **Pin count** – вибір кількості виводів у корпусі.
- **Speed grade** – визначення градації швидкості.
- **Core voltage** – відображається напруга живлення ядра мікросхеми.
- **Show advanced devices** – при включенні цієї опції відображаються тільки мікросхеми з найкращими параметрами.

## Четверте вікно майстра створення проекту

В цьому вікні робиться вибір додаткових засобів налагодження, верифікації та синтезу проекту (рисунок 4.8). Оскільки при роботі буде використовуватися тільки пакет Quartus II, тому додаткові засоби проектування визначати не будемо.

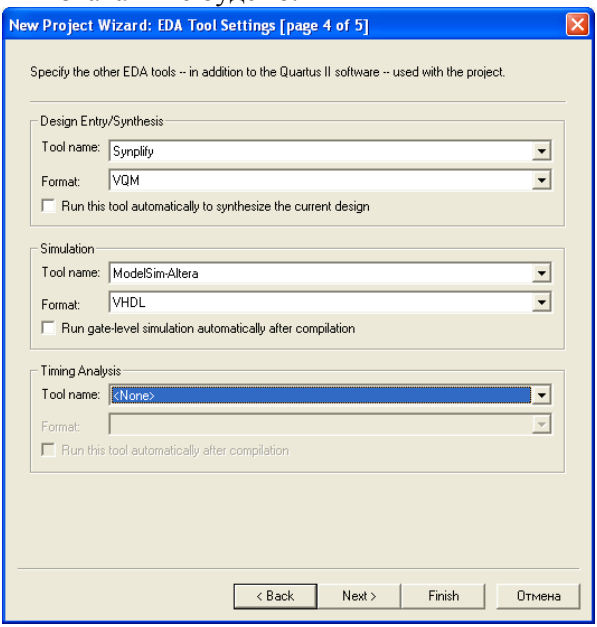



*Рисунок 4.8 – Вікно додавання додаткового програмного забезпечення*

### П'яте вікно майстра створення проекту

Це вікно призначене для відображення сумарного результату створення нового проекту (рисунок 4.9). Ніяких дій тут виконувати не потрібно. Натискання на кнопку **Finish** призводить до створення проекту та закриття майстра.

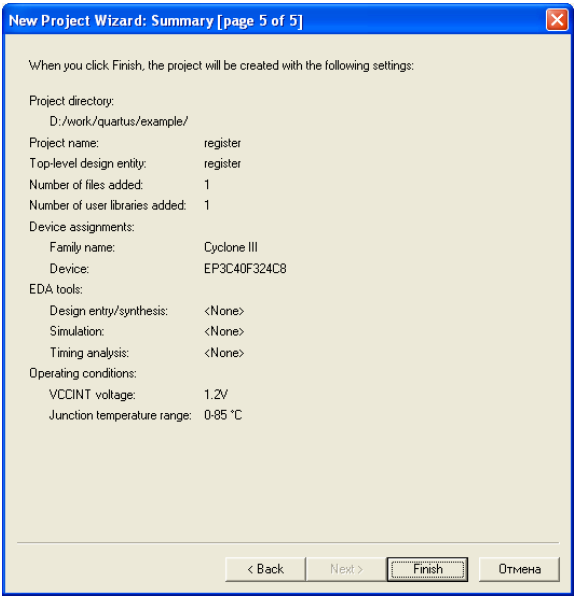



*Рисунок 4.9 – Результат роботи майстра створення проекту*

## 4.1.4. Відкриття існуючого проекту

Відкриття нового проекту призводить до закриття попереднього проекту. Тому необхідно зберегти всі зміни в поточному проекті перед відкриттям нового. Відкрити існуючий проект можна декількома способами:

1. За допомогою меню **File ⇒ Open Project...** (рисунок 4.10). У відкритому вікні, що з'явилося, необхідно вибрати файл із розширенням *.qpf.

2. Вибрати проект зі списку проектів, які вже відкривалися, за допомогою пункту меню **File ⇒ Recent Projects**.

3. Використати TCL команду `project_open` *`ім'я_проекту`*.

Необхідно пам'ятати, що відкриття файлу (меню **File ⇒ Open...**) та відкриття проекту (меню **File ⇒ Open Project...**) не є ідентичними командами.

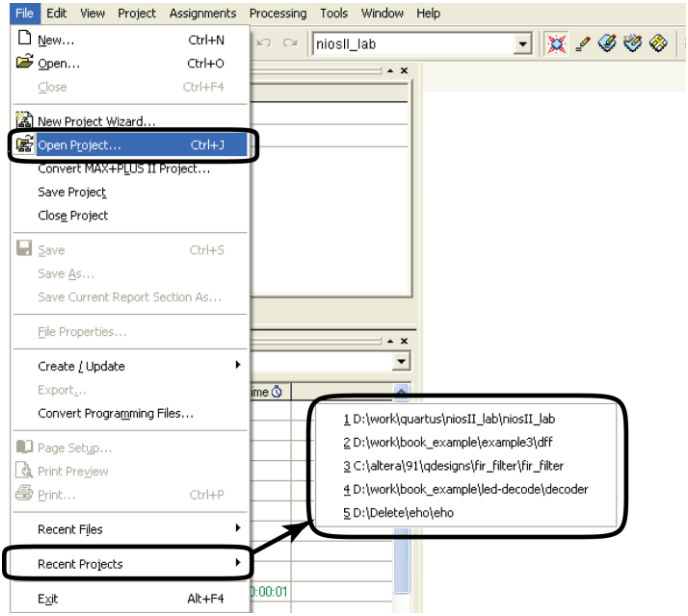
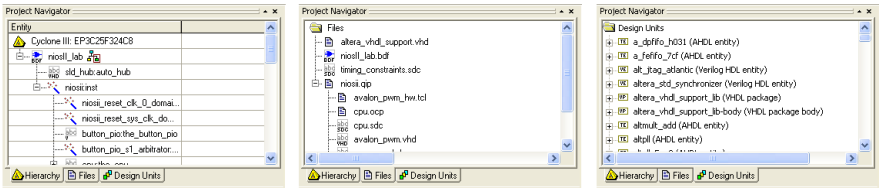



*Рисунок 4.10 – Способи відкриття проекту*

Відкриття проекту призводить до появи у вікні навігатора проекту його ієрархії (рисунок 4.11).




*а* *б* *в*

*Рисунок 4.11 – Закладки вікна навігатора проекту*

Навігатор проекту дає можливість переглянути його ієрархію, файли та модулі, що входять у проект. Вікно навігатора проекту складається із трьох закладок:

– **Hierarchy** – ієрархія проекту (рисунок 4.11 а). Ця закладка використовується для визначення файлу верхнього рівня ієрархії, перегляду використаних ресурсів мікросхеми, пошуку окремої частини проекту в різних редакторах і планувальниках.

– **Files** – файли (рисунок 4.11 б). Ця закладка дозволяє відкрити файли, видалити файли із проекту, визначити новий файл верхнього рівня ієрархії, вибрати спеціальні засоби синтезу для файлу.

– **Design Units** – модулі проекту (рисунок 4.11 в). Відображає різні модулі проекту – інтерфейсні та архітектурні тіла різних блоків на мовах опису апаратури.

## 4.1.5. Керування проектом

Крім роботи з файлами проекту пакет Quartus II дає також можливості управління проектом вцілому, а саме:

– архівація проекту;
– створення копії проекту;
– створення версій проекту.

### Архівація проекту

Під час роботи пакету в каталозі проекту створюється велика кількість файлів. Тільки частина з них потрібна для опису проекту. Інші файли є допоміжними та необхідні для роботи пакету Quartus II.

Тому для переносу проектів між робочими станціями більш зручно використовувати архівний проект. Архівація та відновлення проекту з архіву робиться в середовищі Quartus II з використанням меню **Project ⇒ Archive Project...** та **Project ⇒ Restore Archived Project...** При архівації створюються два файли: файл із архівом проекту *.qar і протокол архівації *.qarlog.

Відмінність архіву, що створює пакет Quartus II, від архіву каталогу із проектом полягає в тому, що в архів проекту входять не тільки файли з каталогами проекту, але також і файли з бібліотек користувача, підключених до проекту, а також налагодження пакета, прийняті при роботі з даним проектом.

TCL команда для створення архіву: `project_archive` *`ім'я_проекту`*.

TCL команда для розпакування архіву: `project_restore` *`ім'я_проекту`*.

## Створення копії проекту

Якщо в процесі роботи над проектом необхідно зробити копіювання його в іншу папку, то в цьому випадку необхідно скористатися командою **Project ⇒ Copy Project...** У цьому випадку в іншу папку будуть скопійовані:

– файл, що зберігає версію пакета і дату, а також зміни, що відбуваються із проектом (*.qpf);
– файли проекту;
– файли установок і призначень.

## Створення версій проекту

Версія проекту – це група призначень та установок проекту. Використання різних версій проекту дозволяє визначити, як конкретно установки впливають на результат компіляції проекту. Для виклику діалогу створення версії необхідно скористатися меню **Project ⇒ Revisions...** Діалог створення версії проекту показаний на рисунку 4.12.

Для створення нової версії проекту необхідно натиснути на кнопку **"Create..."**. В результаті буде відкрите вікно **"Create Revision"**.

– **Revision name** – ім'я нової версії проекту.

– **Based on revision** – тут можна вибрати версію проекту, на основі якої буде створюватися нова. При цьому в нову версію за допомогою пункту **"Copy database"** можна включити базу даних проекту.

– **Description** – опис версії проекту, наприклад, дата і час створення.

– **Set as current revision** – вибір нової версії в якості робочої.

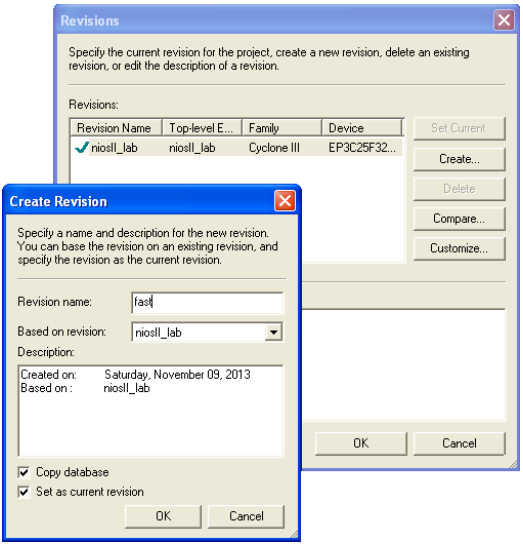



*Рисунок 4.12 – Створення версії проекту*

При роботі в пакеті Quartus II можна вибирати версію проекту за допомогою списку, який розташований на панелі інструментів (рисунок 4.13).

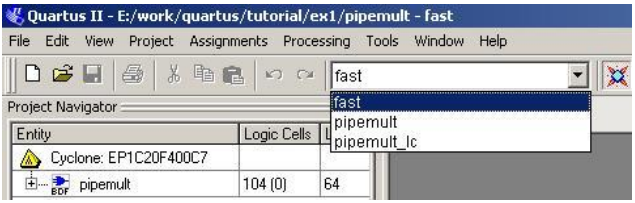
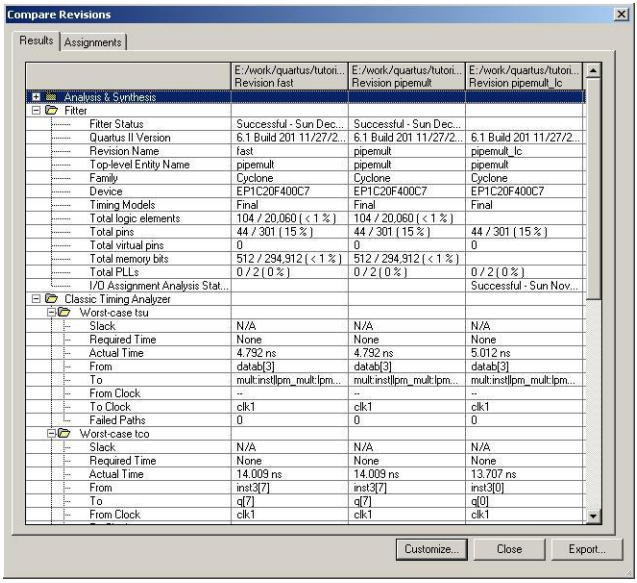



*Рисунок 4.13 – Вибір версії проекту*

Результат роботи компілятора при різному налаштуванні можна переглянути за допомогою кнопки **Compare...** вікна **Revisions** (меню **Project**). Результат порівняння різних версій налаштуваннь показаний на рисунку 4.14.




| | E:/work/quartus/tutori... Revision fast | E:/work/quartus/tutori... Revision pipemult | E:/work/quartus/tutori... Revision pipemult_lc |
|---|---|---|---|
| Analysis & Synthesis | | | |
| Fitter | | | |
| Fitter Status | Successful - Sun Dec... | Successful - Sun Dec... | |
| Quartus II Version | 6.1 Build 201 11/27/2... | 6.1 Build 201 11/27/2... | 6.1 Build 201 11/27/2... |
| Revision Name | fast | pipemult | pipemult_lc |
| Top-level Entity Name | pipemult | pipemult | pipemult |
| Family | Cyclone | Cyclone | Cyclone |
| Device | EP1C20F400C7 | EP1C20F400C7 | EP1C20F400C7 |
| Timing Models | Final | Final | Final |
| Total logic elements | 104 / 20,060 ( < 1 % ) | 104 / 20,060 ( < 1 % ) | |
| Total pins | 44 / 301 ( 15 % ) | 44 / 301 ( 15 % ) | 44 / 301 ( 15 % ) |
| Total virtual pins | 0 | 0 | 0 |
| Total memory bits | 512 / 294,912 ( < 1 % ) | 512 / 294,912 ( < 1 % ) | |
| Total PLLs | 0 / 2 ( 0 % ) | 0 / 2 ( 0 % ) | 0 / 2 ( 0 % ) |
| I/O Assignment Analysis Stat... | | | Successful - Sun Nov... |
| Classic Timing Analyzer | | | |
| Worst-case tsu | | | |
| Slack | N/A | N/A | N/A |
| Required Time | None | None | None |
| Actual Time | 4.792 ns | 4.792 ns | 5.012 ns |
| From | datab[3] | datab[3] | datab[3] |
| To | mult:inst\|lpm_mult:lpm... | mult:inst\|lpm_mult:lpm... | mult:inst\|lpm_mult:lpm... |
| From Clock | -- | -- | -- |
| To Clock | clk1 | clk1 | clk1 |
| Failed Paths | 0 | 0 | 0 |
| Worst-case tco | | | |
| Slack | N/A | N/A | N/A |
| Required Time | None | None | None |
| Actual Time | 14.009 ns | 14.009 ns | 13.707 ns |
| From | inst3[7] | inst3[7] | inst3[0] |
| To | q[7] | q[7] | q[0] |
| From Clock | clk1 | clk1 | clk1 |

*Рисунок 4.14 – Порівняльна таблиця різних версій проекту*

За допомогою кнопки **Customize...** можна провести порівняння з іншим проектом, а за допомогою кнопки **Export... –** провести збереження даних в CSV файл, який може бути відкритий в пакетах MATLAB або Excel.

# 4.2. Створення графічного файлу в пакеті Quartus II

## 4.2.1. Створення нового файлу

Для створення нового файлу необхідно скористатися пунктом меню **File ⇒ New…** або кнопкою  на панелі інструментів. У результаті буде створене вікно вибору типу файлу, показане на рисунку 4.15.

Вікно містить список, з якого обирають різні типи створюваних файлів. Розглянемо основні типи файлів, які будуть використовуватися і надалі в цій книзі.

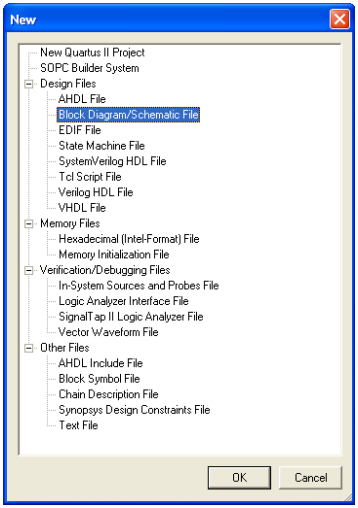



*Рисунок 4.15 – Типи файлів у Quartus II*

Закладка **Device Design Files**:

- **New Quartus II Project** – створення нового проекту.
- **AHDL File** – файл мовою Altera HDL.
- **Block Diagram/Schematic File** – файл-схема.

– **SOPC Builder System** – створення системи на програмувальному кристалі.
– **Verilog HDL File** – файл мовою Verilog.
– **VHDL File** – файл мовою VHDL.

Закладка **Other Files**:

– **Memory Initialization File** – файл, що містить "прошивку" для пам'яті.
– **Vector Waveform File** – файл часових діаграм для моделювання роботи проекту.

## Створення графічного файлу

Для створення графічного файлу необхідно вибрати пункт **Block Diagram/Schematic File** у закладці **Device Design Files**. Це призведе до створення нового файлу, вигляд вікна редагування якого показаний на рисунку 4.16.

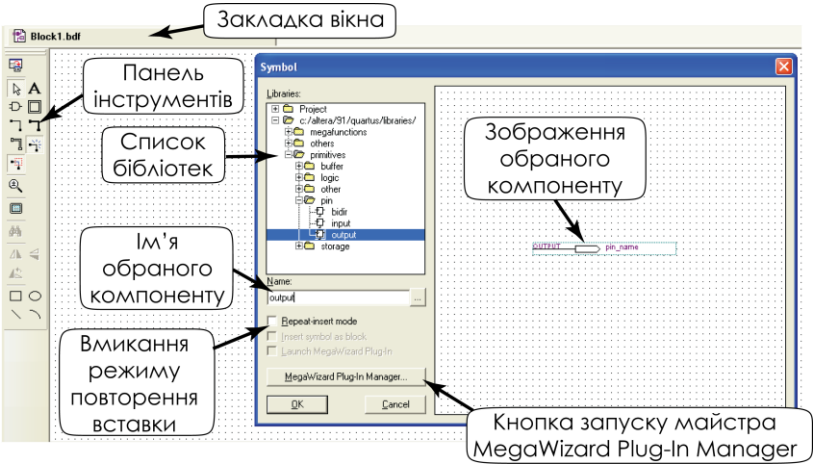



*Рисунок 4.16 – Вікно вставки символу*

Всі вікна в Quartus II містять закладки, подібні закладкам аркуша в Microsoft Excel. Активна закладка має більш темні кольори. Ліворуч від вікна графічного редактора розташована панель

інструментів. Закладка може розташовуватись і окремо від загального вікна пакету Quartus II. Для цього необхідно натиснути кнопку прив'язки вікна (рисунок 4.3). Для повернення вікна назад, до загального вікна пакету необхідно натиснути кнопку прив'язки ще раз.

Саме поле графічного редактора заповнене сіткою, параметри якої можуть бути встановлені за допомогою категорії **Block/Symbol Editor** меню **Tools ⇒ Options...** Основні пункти цієї категорії наведені в таблиці 4.1.

*Таблиця. 4.1 – Категорія Block/Symbol Editor меню Options*

| Назва категорії | Опис |
|---|---|
| **Show guidelines** | Відображає лінії сітки на екрані. |
| **Guideline spacing** | Встановлення кроку сітки. |
| **Snap to grid** | Прив'язка об'єктів до сітки. Використовується тільки в символьному редакторі. |
| **Use rubberbanding** | "Нерозривність" ліній. Дозволяє переміщувати об'єкти не розриваючи зв'язку між ними. Але ця опція не працює, якщо об'єкт перевернути або відобразити дзеркально. |
| **Use partial line selection** | Дозволяє вибирати частину лінії, а не всю лінію цілком. |
| Опції, які починаються зі слова **Show** | Керують відображенням різних параметрів об'єктів схеми. |

Призначення кнопок на панелі інструментів графічного редактора наведено в таблиці 4.2.

*Таблиця. 4.2 – Призначення кнопок панелі інструментів графічного редактору*

| | | | |
|---|---|---|---|
|  | Кнопка прив'язки вікна |  | Малювання каналу |
|  | Кнопка вибору |  | "Нерозривність" ліній |
|  | Вставка тексту |  | Частковий вибір лінії |
|  | Вставка графічного символу |  | Масштаб |
|  | Вставка блоку |  | Повноекранний режим |
|  | Малювання провідника |  | Пошук |
|  | Малювання шини | | |

Для вставки символу у файл необхідно натиснути кнопку вставки символу  або виконати подвійне клацання по порожньому місцю на робочому листку. При цьому з'явиться вікно вставки символу, показане на рисунку 4.16. Для вибору символу можна ввести його ім'я в поле **Name:** або вибрати символ зі списку доступних бібліотек, що відображуються в полі **Libraries**. У полі **Libraries** відображаються символи, які містяться в робочому каталозі проекту, символи з інших бібліотек, а також символи зі стандартних бібліотек, які встановлюються разом з пакетом.

За замовчуванням користувачу доступні такі бібліотеки:

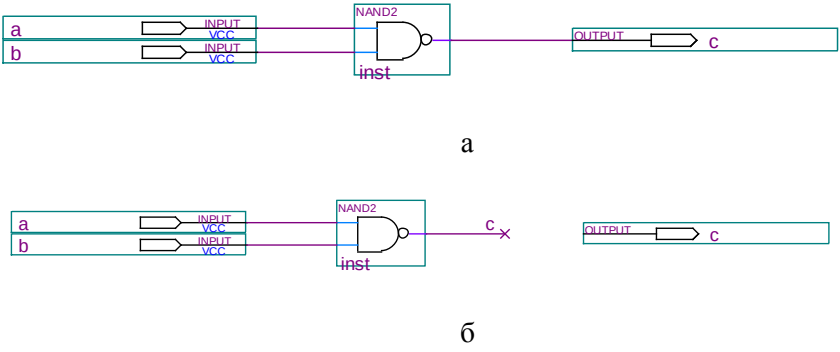
1. **primitives** – бібліотека примітивів:
   - **buffer** – буфери SOTF, WIRE, LCELL, GLOBAL і т.д.
   - **logic** – логіка: елементи І, АБО, НІ і т.д.
   - **other** – примітиви землі (GND), живлення (VCC), константи;
   - **pin** – виводи: вхід (INPUT), вихід (OUTPUT), двонаправлений (BIDIR);

– **storage** – тригери;

2. **other** – інші примітиви.
   – **maxplus2** – примітиви 74 серії. Не рекомендується для застосування.

3. **megafunctions** – мегафункції, які використовуються майстром MegaWizard. Склад бібліотеки буде описаний нижче.

Як приклад розглянемо побудову схеми, показаної на рисунку 4.17.




а

б

*Рисунок 4.17 – Приклади схеми*

Для малювання цієї схеми знадобляться наступні примітиви:
NAND2 – елемент 2I–HI;
INPUT – примітив входу;
OUTPUT – примітив виходу.

## Створення схеми

Створіть графічний файл (Block Diagram/Schematics File).

Виконайте подвійне клацання та вставте необхідні примітиви. При роботі в редакторі можливе використання миші в режимі "натиснути і перетягнути".

З'єднання елементів. Для з'єднання елементів схеми немає необхідності переходити в режим малювання провідника. Якщо підвести курсор до виводу елемента, біля курсору з'явиться

зображення значка на кнопці малювання провідників: ┐. Натискаючи на ліву кнопку миші, з'єднайте елементи так, як показано на рисунку 4.17 а.

Крім звичного всім з'єднання елементів схеми провідниками можливе використання імен провідників для з'єднання. У цьому випадку два провідники вважаються з'єднаними, якщо вони мають однакові імена (рисунок 4.17, б).

Для визначення імені провідника необхідно вибрати провідник і натиснути праву кнопку миші. У контекстному меню, що відкрилося, необхідно вибрати пункт "Властивості" (**Properties**) (рисунок 4.18). У вікні, що відкрилося, **Node Properties** у вкладці "Загальні" (**General**) у поле **Name** необхідно ввести ім'я провідника. У випадку використання групи провідників необхідно використовувати ті ж правила, що прийняті для контактів.

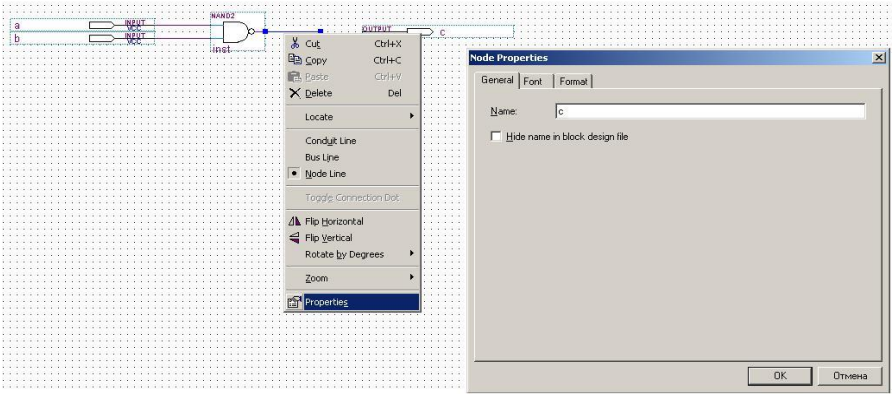



*Рисунок 4.18 – Редагування імені провідника*

Перейменування входів та виходу. Для перейменування входів та виходу необхідно виконати подвійне клацання по входу, що призведе до відкриття діалогу, показаного на рисунку 4.19.

Тут:

– **Pin name(s)** – ім'я контакту або контактів.

– **Default value** – значення за замовчуванням, тобто значення, що буде подаватися на контакт у тому випадку, якщо він

залишився непідключеним. При цьому необхідно пам'ятати, що таке значення стосується тільки внутрішніх вузлів схеми. Зовнішні виводи повинні управлятися елементами вводу–виводу.

У тому випадку, коли необхідно задати групу контактів, то їхню назву вводять у наступному форматі: *ім'я_контакту [старший_розряд .. молодший_розряд]*.

Наприклад: addr[7..0], count[16..8].

У записах груп контактів у програмному забезпеченні фірми Altera прийнято записувати старший значущий біт ліворуч, а молодший значущий біт – праворуч. Зворотний порядок призводить до появи попередження (Warning).

Для звернення до окремих контактів із групи варто користуватися наступним форматом: *ім'я_контакту [номер_розряду]*.

Наприклад: addr[7], addr[5], count[10].

Введіть у поле **Pin name(s)** значення відповідно до рисунка 4.17.

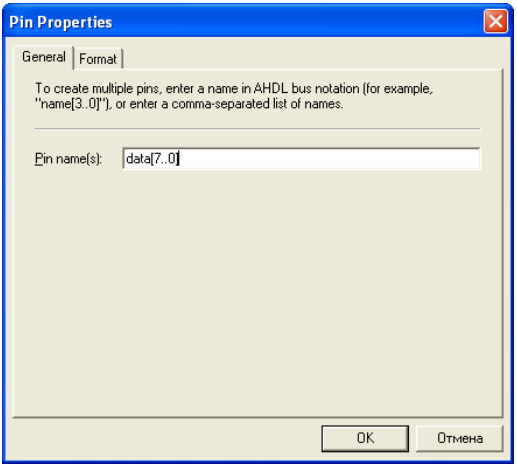



*Рисунок 4.19 – Редагування імені виводу ПЛІС*

На рисунку 4.20 показана робота з шиною (bus) та каналом (conduit). Шина – це група ліній, що має одне ім’я та різні цифрові індекси. Наприклад, addr [15..0]. При роботі з шиною можна оперувати як з усією групою в цілому так і з окремими сигналами.

При з’єднанні шин необхідно контролювати, щоб розрядність шин була однаковою. Канал – ще група ліній, які можуть мати різні імена та розрядності, тобто канал може включати в себе як окремі провідники так і шини. Більш докладно про роботу з каналами див. параграф 5.1.

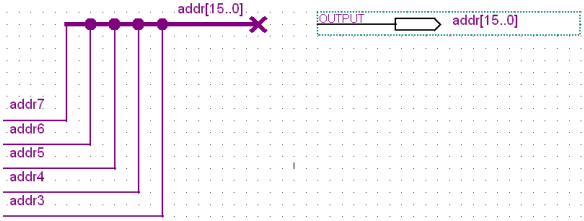



*Рисунок 4.20 – Робота з шиною та каналом*

**Збереження схеми**. При збереженні Quartus II запропонує ім'я файлу, яке співпадатиме з ім'ям проекту. Якщо файл є файлом верхнього рівня ієрархії (Top-Level Entity), то змінювати його не потрібно. В усіх інших випадках бажано, щоб ім’я файлу було унікальним і не повторювало назви компонентів у проекті і самому файлі.

## 4.2.2. Створення елементів за допомогою майстра MegaWizard Plug-In Manager

Майстер MegaWizard Plug-In Manager дозволяє, використовуючи параметризовані функції, створювати велику кількість різних цифрових блоків: починаючи від найпростіших логічних елементів закінчуючи системами на кристалі. У прикладі, що розглядається нижче буде побудований 8-розрядний інкрементуючий лічильник.

Майстер являє собою послідовність діалогових вікон, у яких користувач вводить параметри обраного блоку. Кількість кроків може значно різнитися в залежності від складності елемента. При створенні елементів бажано зберігати їх в робочому каталозі проекту.

Запуск майстра MegaWizard Plug–In Manager можливий наступними методами: вибрати меню **Tools ⇒ MegaWizard Plug–In Manager...** або виконати подвійне клацання по порожньому полю в графічному редакторі і у вікні, що відкрилося (рисунок 4.16), натиснути кнопку MegaWizard Plug–In Manager...

Перше вікно майстра показане на рисунку 4.21.

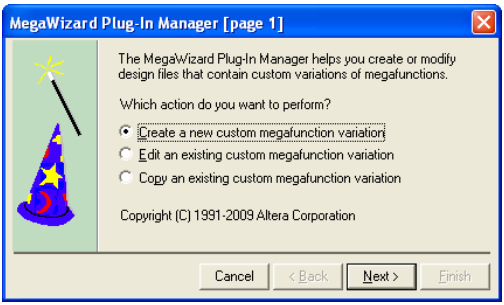



*Рисунок 4.21 – Вікно запуску MegaWizard Plug–In Manager*

Тут необхідно вибрати одну з трьох дій:

1. **Create a new custom megafunction variation** – створити нову мегафункцію користувача.
2. **Edit an existing custom megafunction variation** – редагувати існуючу мегафункцію.
3. **Copy an existing custom megafunction variation** – копіювати існуючу мегафункцію.

Вибираємо перший пункт – створюємо нову мегафункцію. Натискаємо кнопку **Next**. Це призводить до відкриття другого вікна майстра (рисунок 4.22). В ньому необхідно вибрати з бібліотеки необхідну мегафункцію. Список бібліотек може відрізнятися від вказаного на рисунку 4.21 і залежить від встановлених бібліотек. Деякі елементи бібліотеки можуть бути неактивними, тому що вони не підтримуються мікросхемами обраного сімейства.

Стандартна бібліотека мегафункцій включає наступні розділи:

– **Arithmetic** – арифметичні пристрої: суматори, помножувачі, компаратори і т.д.

- **DSP** – блоки цифрової обробки сигналу: фільтри, блоки швидкого перетворення Фур'є.
- **Gates** – вентилі: елементи І, АБО і т.д. Основна відмінність від елементів, що входять до складу бібліотеки примітивів – це масштабованість, тобто можливість вибору розрядності вхідної шини.
- **I/O** – інтерфейси різних типів пам'яті (DDR, EPCS), трансиверів і т.д.
- **Interfaces** – інтерфейс мікросхем DDR SDRAM, DDR2 SDRAM.
- **Memory Compiler** – різні типи пам'яті: ОЗП, ПЗП.
- **Storage** – регістри та тригери.

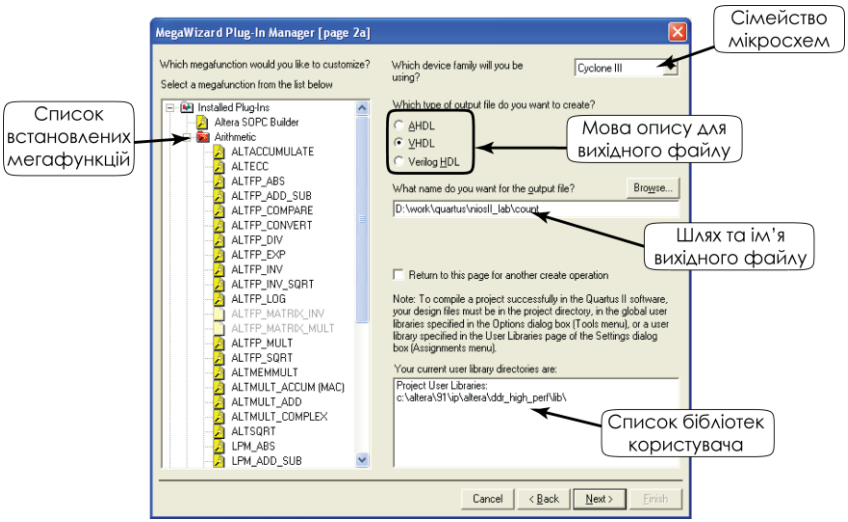



*Рисунок 4.22 – Вікно вибору мегафункції*

Розглянемо поля, що розташовані у цьому вікні:

**Which device family will you be using?** – вибір відповідного сімейства мікросхем за допомогою списку, що розгортається.

**Select a megafunction from the list below** – вибір необхідної мегафункції зі списку встановлених мегафункцій. Для створення

лічильника необхідно вибрати функцію LPM_COUNTER розділу Arithmetics.

**Which type of output file do you want to create?** – вибір мови, на якій буде написана відповідна мегафункція. Як мова опису може бути обрана одна з мов: AHDL, VHDL, Verilog HDL. Якщо вибір мови не має особливого значення, то краще залишити вибір за замовчуванням.

**What name do you want for the output file?** – Визначення імені і шляху розташування вихідного файлу. При необхідності вибору каталогу для розміщення файлів можна скористатися кнопкою **Browse...**

**Next** – перехід до наступного вікна.

Три наступні вікна визначають параметри обраної мегафункції. В якості прикладу розглянемо створення восьмирозрядного інкрементуючого лічильника.

1. **Parameter settings** – установка параметрів (рисунок 4.23).

Дана закладка призначена для визначення параметрів функції: розрядності, необхідних портів, констант і т.д. На цій вкладці показується символ функції, який буде використовуватись в графічному редакторі та кількість використаних ресурсів.

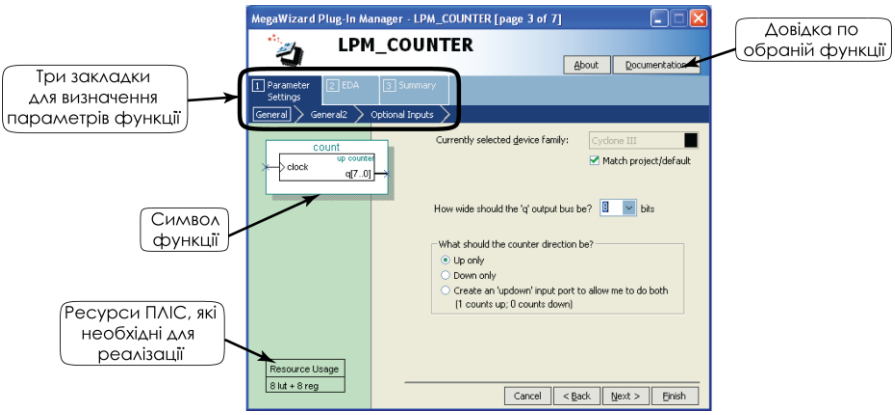



*Рисунок 4.23 – Вибір розрядності та напряму рахування лічильника*

Для створення лічильника в даній закладці встановлюємо розрядність яка дорівнює 8 у поле **"How wide should the 'q' output bus be?"**.

У поле **"What shoud the counter direction be?"** – визначаємо напрямок підрахунку:

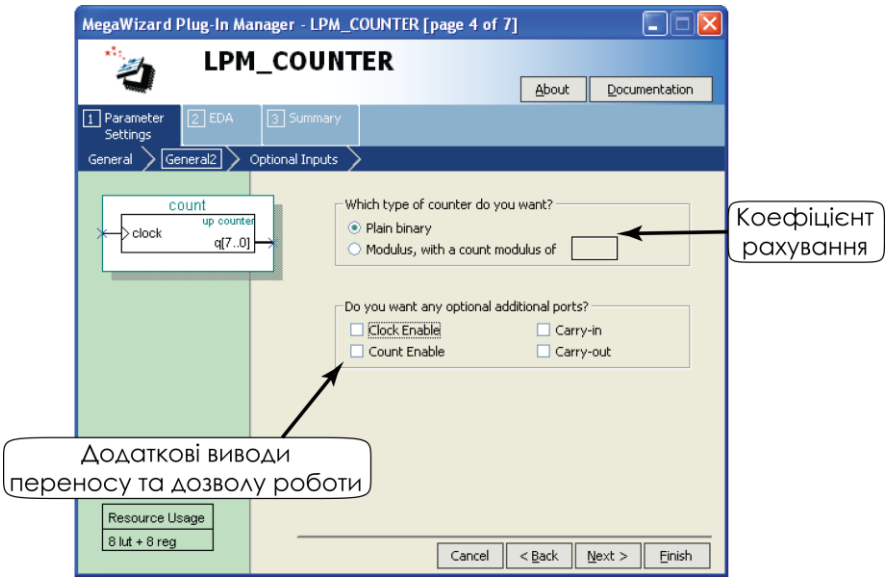
– **Up only** – інкрементуючий лічильник

– **Down only** – декрементуючий лічильник

– **Create an 'updown' input port to allow me to do both (1 counts up; 0 counts down)** – реверсивний лічильник: 1 – рахунок вгору, 0 – вниз.

Вибираємо **Up only** і натискаємо кнопку **Next**.

Наступне вікно (рисунок 4.24) визначає коефіцієнт рахування, а також додаткові порти переносу та дозволу рахування. Тут залишаємо все без змін і натискаємо кнопку **Next**.




*Малюнок 4.24 – Визначення додаткових параметрів лічильника*

Вікно **"Optional Inputs"** визначає ще одну групу додаткових виводів: синхронних та асинхронних входів установки, скидання та

завантаження. У цьому вікні залишаємо все без змін і натискаємо кнопку **Next**.

2. Наступна закладка – **Simulation Library** – призначена для визначення файлів, необхідних для симуляції даної мегафункції в інших засобах проектування (рисунок 4.25), наприклад, ModelSim.

Тут не вибираємо ніяких опцій і натискаємо кнопку **Next**.

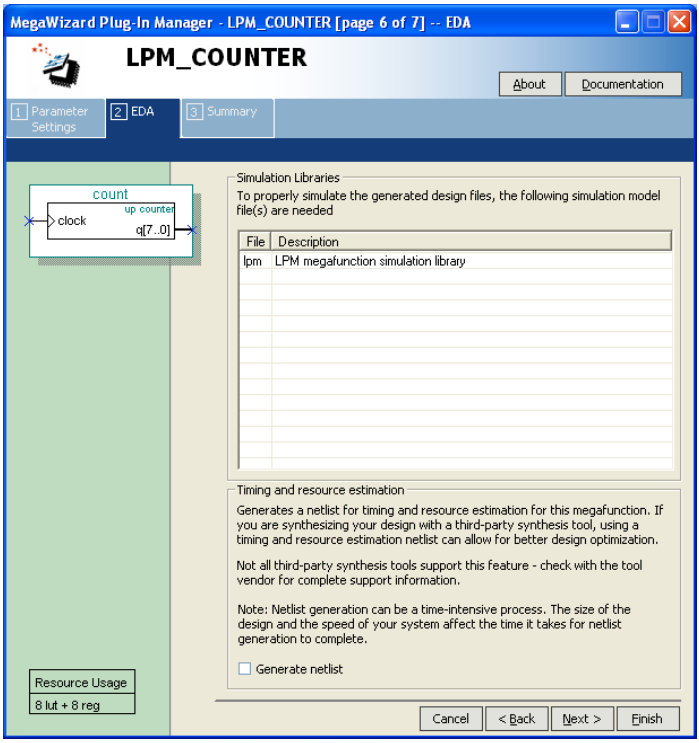



*Рисунок 4.25 – Визначення файлів для симуляції лічильника*

3. Вкладка **Summary** (рисунок 4.26). Тут відображається список файлів, які будуть згенеровані майстром MegaWizard Plug–In Manager (табл. 4.3).

*Таблиця. 4.3 – Список файлів, згенерованих MegaWizard Plug–In Manager*

| Метод введення проекту | Файли, згенеровані MegaWizard Plug–In |
|---|---|
| VHDL | mult.vhd, mult.cmp & mult_inst.vhd |
| Verilog | mult.v & mult_inst.v |
| Schematic | mult(.vhd or .v) & mult.bsf |

У цій закладці залишаємо значення за замовчуванням і натискаємо кнопку **Finish**.

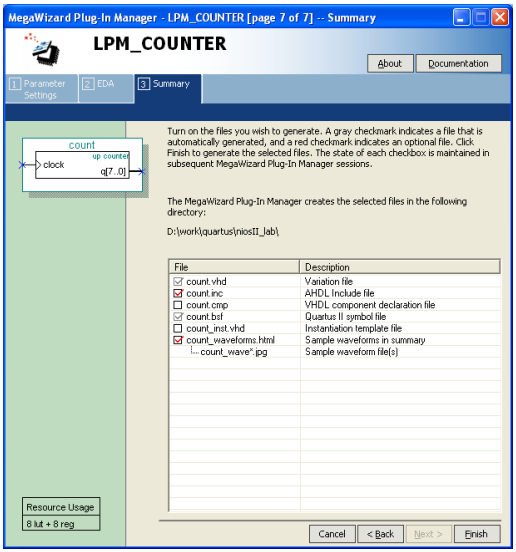



*Рисунок 4.26 – Визначення додаткових файлів для моделювання*

Натискання на кнопку **Finish** призводить до закриття майстра і поверненню у вікно вибору елемента (рисунок 4.27). При цьому у вікні буде відображений символ розробленої мегафункції. Натискання на кнопку **Ok** приведе до включення отриманого символу в графічний файл.

Якщо виникне необхідність змінити вже створену мегафункцію, то повторно відкрити вікно майстра MegaWizard Plug–In Manager можна за допомогою подвійного клацання по символу функції.

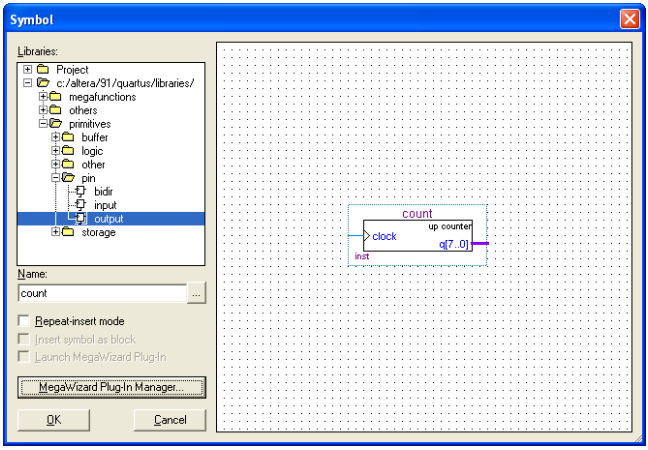



*Рисунок 4.27 – Символ розробленого лічильника*

## *4.3. Компіляція проекту*

Процес компіляції в пакеті Quartus II у своєму типовому вигляді може бути описаний за допомогою алгоритму, показаного на рисунку 4.1. До цього процесу можуть бути додані додаткові модулі, призначення яких визначається програмним забезпеченням інших виробників.

Запуск компілятора здійснюється або при натисненні кнопки  , або вибором пункту меню **Processing ⇒ Start Compilation**. У цьому випадку буде виконана повна компіляція. Однак, можливий запуск окремих складових компілятора. Для цього необхідно вибрати пункт меню **Processing ⇒ Start** (рисунок 4.28).

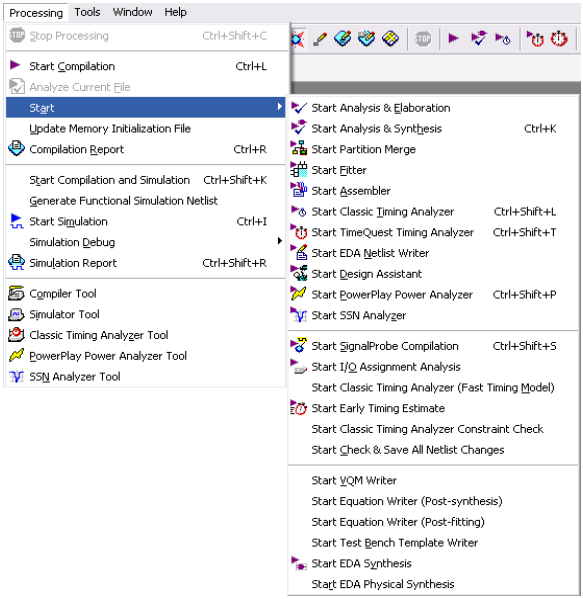



*Рисунок 4.28 – Запуск модулів компілятора*

Розглянемо деякі модулі компілятора, доступ до яких здійснюється за допомогою меню:

– **Start Analysis & Elaboration** – зберігає всі файли проекту, запускає перевірку синтаксису та семантики проекту. Після роботи цього модуля можливий перегляд ієрархії проекту та перегляд файлів проекту в **RTL Viewer** (див. нижче).

– **Start Analysis & Synthesis** – включає всі дії **Analysis & Elaboration**, формує файл зв'язків для створення бази даних проекту, також виконує синтез проекту.

– **Start Fitter** – розміщує проект на кристалі ПЛІС. Перед запуском цього модуля необхідно виконати синтез проекту за допомогою модуля **Analysis & Synthesis**.

– **Start Assembler** – модуль, що створює з результатів роботи компанувальника (**fitter**) файли для програмування або конфігурування ПЛІС.

– **Start Classic Timing Analyzer** або **TimeQuest Timing Analyzer** – запуск модулів для розрахунку часових параметрів і побудови часових діаграм проекту.

– **Start Design Assistant** – модуль, що перевіряє проект на відповідність правилам проектування.

В процесі роботи статус компіляції та її результати виводяться у відповідних вікнах пакету (рисунок 4.29).

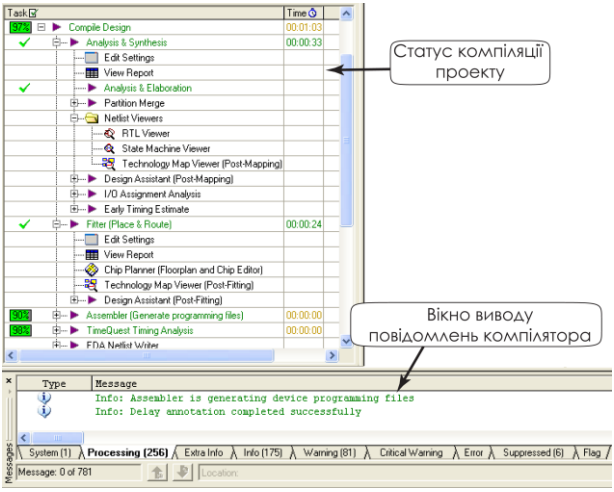



*Рисунок 4.29 – Вікно повідомлень та статусу компіляції*

Результати компіляції проекту можуть бути переглянуті декількома способами:

1. З використанням звіту про компіляцію (**Compilation Report**).
2. Використовуючи спеціальні переглядачі:

– переглядач проекту у вигляді структурної схеми та технологічних блоків (**RTL & Technology Map**);

– перегляд графів цифрових автоматів (**State Machine Viewer**).

3. З використанням планувальника ресурсів мікросхеми (**Chip Planner**).

Для перегляду окремих елементів проекту в зазначених модулях необхідно вибрати потрібний елемент схеми і у контекстному меню вибрати пункт **Locate** (Розташування). Далі відкриється список, який дозволить отримати доступ до зазначеного вище модуля.

### 4.3.1. Звіт про компіляцію

Звіт про компіляцію являє собою вікно (рисунок 4.30), яке автоматично відкривається після компіляції та містить всю інформацію про результати компіляції:

- використанні ресурси мікросхеми;
- виводи мікросхеми;
- налаштування та призначення, а також їхнє виконання;
- інформаційні повідомлення, списки попереджень і помилок.

Також результати компіляції можуть бути переглянуті в текстових файлах, які розташовуються в робочому каталозі проекту. Це наступні файли: *ім'я_проекту*.fit.rpt та *ім'я_проекту*.map.rpt.

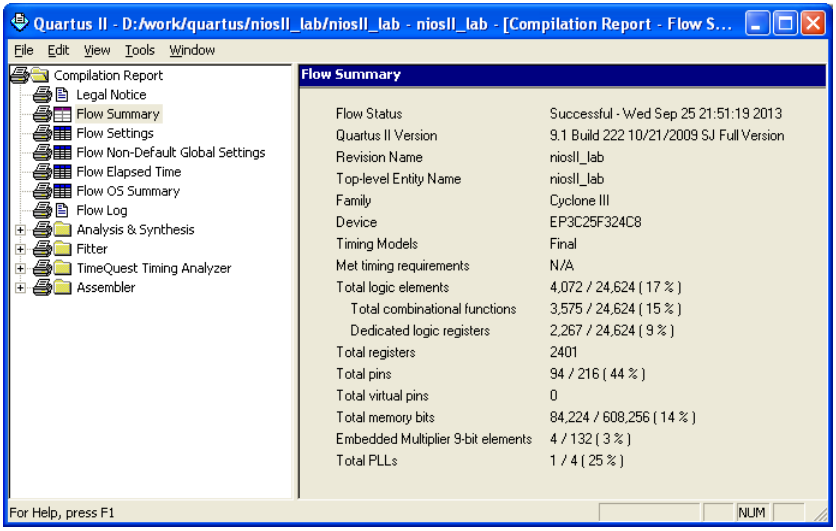



*Рисунок 4.30 – Звіт про результати компіляції*

## 4.3.2. RTL Viewer та Technology Map Viewer

Графічне представлення вузлів проекту в базисі ПЛІС може бути отримане при використанні **RTL Viewer** (рисунок 4.31). Для відкриття цього переглядача необхідно вибрати пункт меню **Tools ⇒ Netlist Viewer ⇒ RTL Viewer.** Необхідно пам'ятати, що результати у вигляді функціональної схеми можна побачити тільки після запуску модулів **Analysis & Elaboration** або **Analysis & Synthesis**.

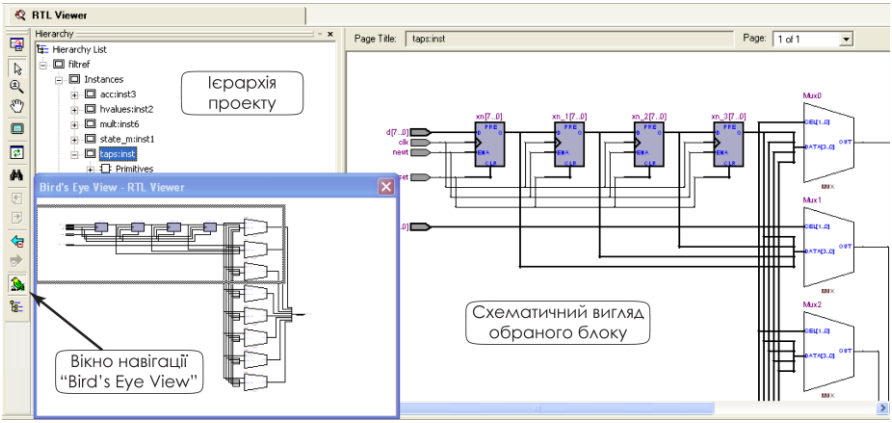



*Рисунок 4.31 – Вікно RTL Viewer*

**RTL Viewer** показує схематичне зображення внутрішньої структури мікросхеми, отримане в результаті компіляції. При цьому є можливість подивитися як проект використовує ресурси мікросхеми, тобто логічні блоки, елементи вводу–виводу, мультиплексори, тригери.

**Technology Map Viewer** – дозволяє визначити, як окремі елементи проекту використовують ресурси логічних елементів: таблиці перекодування, конфігуруємі тригери, мультиплексори і т.д. (рисунок 4.32).

Різниця між представленням в **RTL Viewer** та **Technology Map Viewer** полягає в тому, що в першому відображається подання проекту у вигляді функціональних блоків, а в другому – у вигляді ресурсів логічних елементів.

Ці редактори можуть застосовуватися для визначення критичних шляхів при оптимізації проекту по швидкодії або для виявлення помилок в Verilog або VHDL коді.

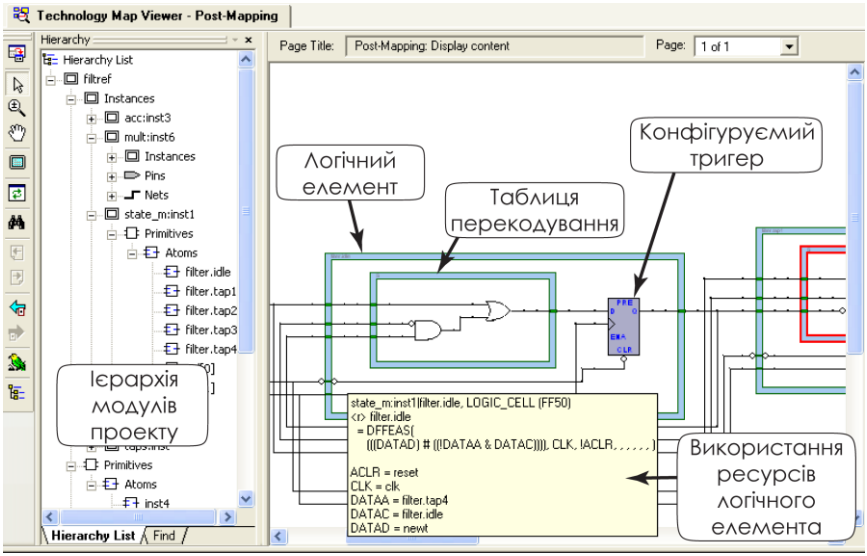



*Рисунок 4.32 – Вікно Technology Map Viewer*

### 4.3.3. State Machine Viewer

Даний модуль дозволяє побачити графічне представлення цифрового автомата, а також таблицю переходів цифрового автомата (рисунок 4.33). Для відкриття цього переглядача необхідно вибрати пункт меню **Tools ⇒ Netlist Viewer ⇒ State Machine Viewer**. Робота з цифровими автоматами описана в розділі 5.

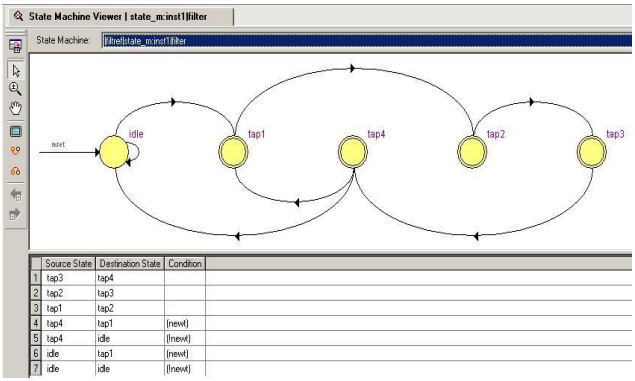



*Рисунок 4.33 – Вікно State Machine Viewer*

## 4.3.4. Chip Planner

**Chip Planner** (планувальник кристала) дає графічне представлення кристалу використовуваної мікросхеми (рисунок 4.34). Для відкриття цього планувальника необхідно вибрати пункт меню **Tools ⇒ Chip Planner ⇒Floorplan & Chip Editor** або натиснути кнопку  на панелі інструментів.

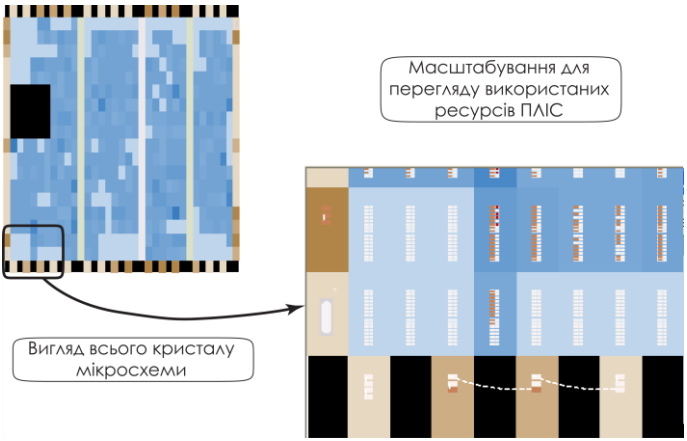



*Рисунок 4.34 – Вікно планувальника кристала*

Планувальник кристала дозволяє побачити графічне зображення ресурсів мікросхеми, а також канали з'єднання між елементами мікросхеми (рисунок 4.35).

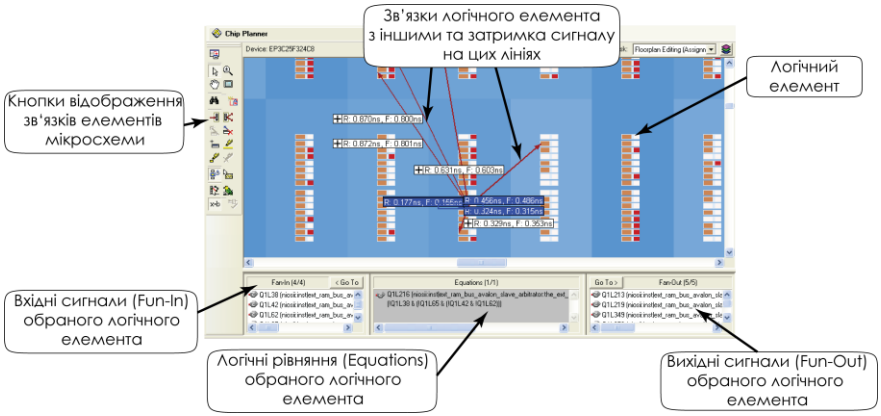



*Рисунок 4.35 – Відображення зв'язків між елементами мікросхеми*

Також для обраного елемента мікросхеми будуть відображені вхідні (**Fan–In**), вихідні (**Fan–Out**) зв'язки та логічні рівняння, які відображають зв'язок входів і виходів (**Equations**).

## *4.4. Установки та призначення проекту*

Функціонування пристрою на ПЛІС багато в чому визначається установками компілятора та призначеннями, які виконуються для визначення виводів мікросхеми та стилів синтезу проекту. Етап визначення опцій синтезу проходить за алгоритмом, наведеним на рисунку 4.36. Установки, виконані за допомогою діалогу **Settings**, редактора призначень (**Assignment Editor**), планувальника виводів (**Pin Planner**) і часових аналізаторів підсумовуються у файлі установок пакету Quartus II (Quartus II Settings File – *.qsf). Ці дані, нарівні з файлами, що описують проект, визначають результат компіляції.

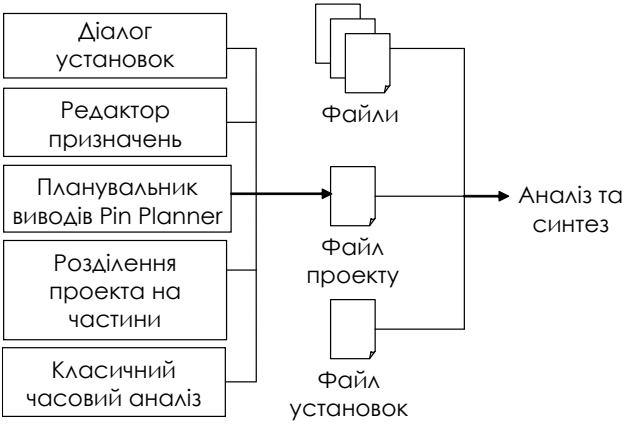



*Рисунок 4.36 – Визначення параметрів проекту*

Для відкриття діалогу **Settings** необхідно вибрати пункт меню **Assignments ⇒ Settings** (рисунок 4.37).

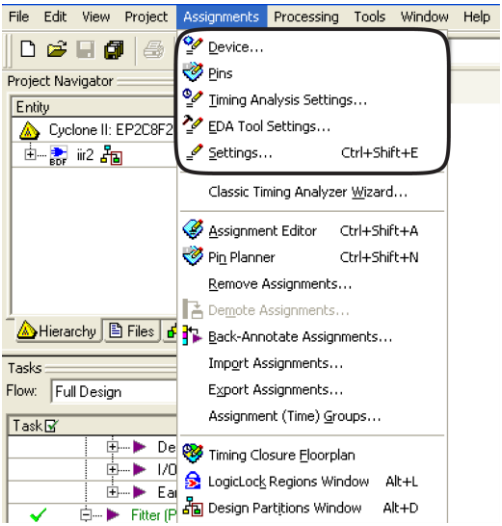



*Рисунок 4.37 – Меню призначень проекту*

За допомогою діалогу **Settings** можна виконати установки, які виконувалися при створенні проекту за допомогою майстра (див. параграф 4.1) – це пункти **General, Files, User Libraries (Current Project), Device, EDA Toll Settings** (рисунок 4.38).

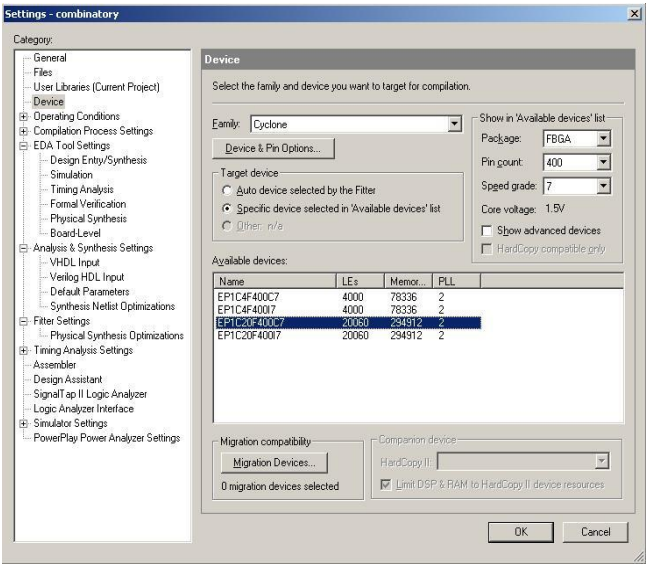



*Рисунок 4.38 – Вікно вибору мікросхеми*

Керування процесом компіляції виконується у вкладці **Compilation Process Settings** (рисунок 4.39). На цій вкладці найбільш важливим є пункт **Use Smart compilation** (Використати "розумну" компіляцію). Ця опція дозволяє зменшити час компіляції, але при цьому збільшується обсяг зайнятого проектом дискового простору. Після проведення першої такої компіляції всі наступні компіляції проекту будуть проводитися таким чином, щоб процес компіляції використовував тільки необхідні модулі компілятора. Якщо ж були проведені зміни в логіці проекту, то проводиться повна компіляція. Тобто дана опція повинна використовуватись на етапі налагодження проекту, коли проводиться зміна налагоджень та установок проекту.

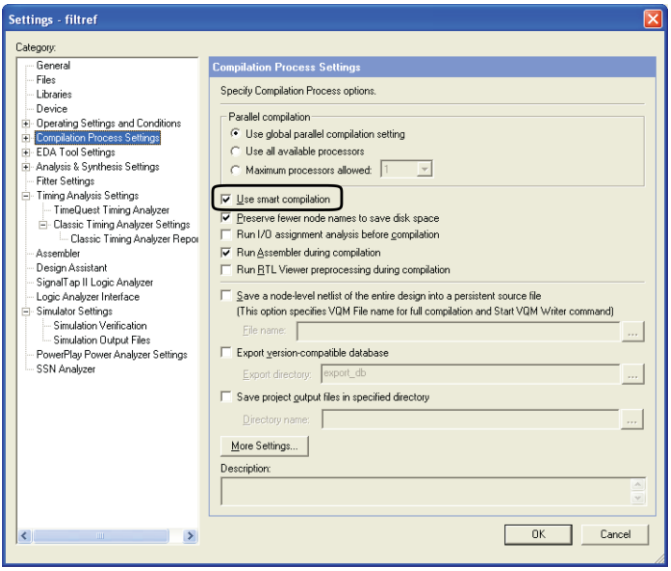



*Рисунок 4.39 – Вікно опцій компілятора*

Результат компіляції проекту багато в чому залежить і від техніки синтезу, що встановлюється у вкладці **Analysis & Synthesis Settings** (рисунок 4.40).

Тут потрібно звернути увагу на пункт **Optimization Technique** (Техніка оптимізації):

– **Speed** – оптимізація проекту по швидкості. У цьому випадку проект буде займати більшу площу кристала.

– **Area** – оптимізація проекту по займаній площі кристала. При цьому проект може працювати повільніше.

– **Balanced** – середнє значення між швидкістю та використовуваною площею кристалу.

Опції компанувальника встановлюються за допомогою вкладки **Fitter Settings** (рисунок 4.41). Найбільш важливим тут є пункт **Fitter effort** (Зусилля компанувальника), що містить наступні пункти:

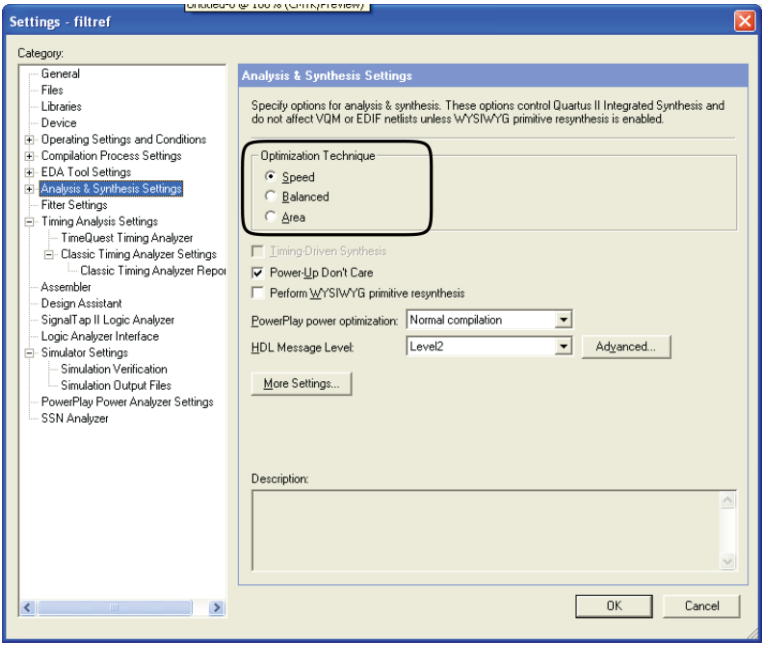



*Рисунок 4.40 – Вікно опцій логічного синтезу*

– **Standard Fit** – ця опція націлена на збільшення максимальної тактової частоти проекту. При цьому час компіляції збільшується.

– **Fast Fit** – зменшення часу компіляції приблизно на 50%. Але це призводить до зменшення максимальної тактової частоти проекту в середньому на 10%

– **Auto Fit** – компілятор працює також, як й у випадку **Fast Fit**, але якщо є установки часових параметрів або параметрів розміщення проекту, то час компіляції збільшується для виконання вимог проектувальника. Цей стиль роботи компанувальника краще використовувати для нових проектів.

Настроювання параметрів компанувальника та стилів синтезу використовується при оптимізації проекту по швидкодії або об’єму, що займає проект. Ці дії описані в розділі 5.

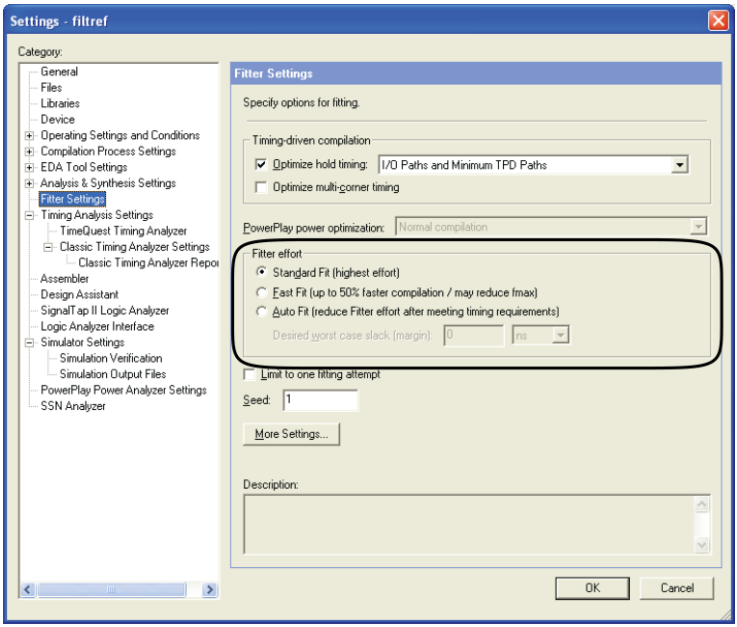



*Рисунок 4.41 – Вікно параметрів компанувальника*

## 4.5. Призначення виводів мікросхеми в Quartus II

Призначення виводів у пакеті Quartus II може бути виконано декількома способами. Найбільш часто використовуються наступні методи:

- з використанням планувальника виводів (**Pin Planner**);
- імпорт із таблиці в CSV форматі;
- редагуванням файлу призначень проекту (QSF);
- за допомогою програми.

У цій книзі буде розглянутий тільки метод призначення виводів за допомогою планувальника виводів.

Планувальник виводів – це інтерактивний засіб, який може бути відкритий за допомогою вибору пункту меню **Assignments ⇒ Pin**

**Planner** або кнопки  (рисунок 4.42). Його вікно складається із трьох частин:

– списку всіх виводів;
– списку груп виводів;
– зображення мікросхеми – виду згори (**Top view**) або виду знизу (**Bottom view**).

Список виводів містить наступні поля:

– **Node Name** – ім'я виводу.
– **Direction** – напрямок передачі даних: вхід, вихід або двонаправлений.
– **Location** – розташування, тобто номер виводу в корпусі.
– **I/O Bank** – банк вводу–виводу.
– **Vref Group** – опорна напруга для групи виводів.
– **I/O Standart** – стандарт вводу–виводу.
– **Reserved** – резервування, тобто вказівка на те що необхідно робити з тими виводами, які не використовуються.
– **Group** – група виводів.

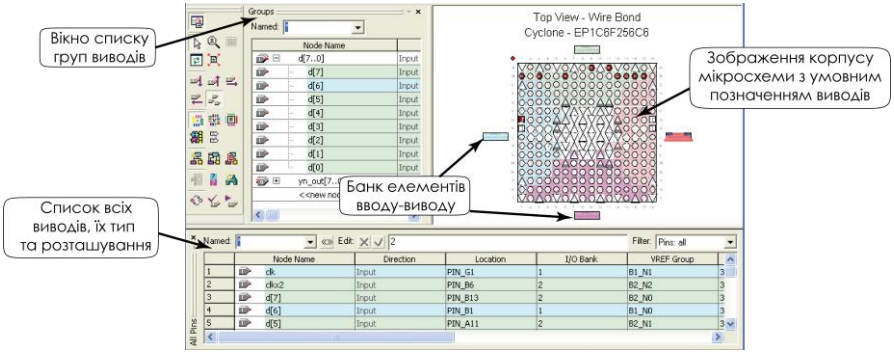



*Рисунок 4.42 – Вікно редагування виводів проекту*

## Порядок призначення виводів у пакеті Quartus II

Розглянемо порядок призначення виводів за допомогою **Pin Planner**.

1. Відкриваємо проект.
2. Підготуємо планувальник до роботи.

2.1. Відкриваємо планувальник виводів за допомогою меню **Assignments ⇒ Pin Planner**.

2.2. Переконаємося, що для вигляду мікросхеми встановлений вид згори (**Top view**). Якщо ні, то його необхідно встановити за допомогою пункту меню **View ⇒ Show ⇒ Package Top**.

2.3. Переконаємося, що обрано режим відображення банків вводу–виводу (**Show I/O bank**) – кнопка  натиснута. Більш докладно про банки вводу-виводу див. у розділі 1.

3. Призначення виводів.

3.1. У вікні списку груп виводів (Group) планувальника **Pin Planner** вибираємо групу входів, наприклад, **data[7..0].** Утримуючи цю групу, за допомогою лівої клавіші перетягуємо її на позначення банку елементів вводу–виводу **IOBank_2** у вікні зображення мікросхеми. При цьому Quartus II автоматично призначить виводам проекту доступні виводи мікросхеми. Такий метод використовується в тому випадку, коли ще немає топології друкованої плати, тобто не існує жорсткої прив'язки виводів проекту.

3.2. У тому випадку, коли потрібно призначати окремі виводи, можна користуватися наступними способами:

3.2.1. Вибрати необхідний вивід у списку всіх виводів і перетягнути на потрібний вивід у вікні зображення мікросхеми.

3.2.2. Вибрати необхідний вивід у списку всіх виводів та у таблиці параметрів зробити подвійне клацання по полю **Location** (рисунок 4.43). В списку вибрати потрібний вивід.

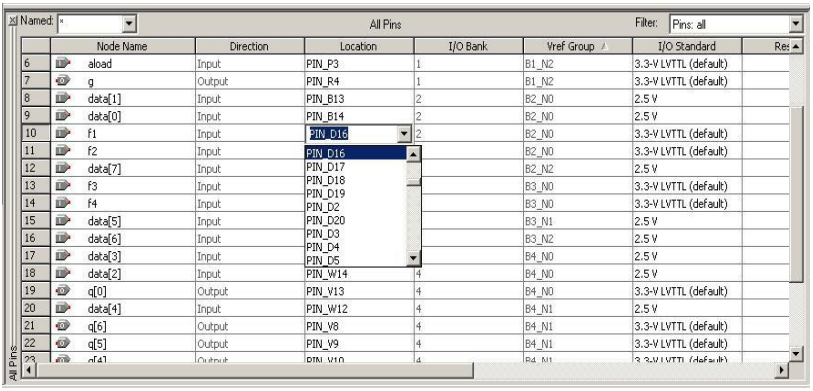



*Рисунок 4.43 – Таблиця виводів мікросхеми*

4. Призначення властивостей виводів. Визначення властивостей перш за все передбачає визначення стандарту вводу–виводу для одного або декількох виводів. Для доступу до вікна властивостей необхідно виконати одну з описаних дій:

– У вікні списку груп виводів (Group) натиснути праву кнопку на шині і вибрати пункт **Node Properties**, що призведе до відкриття вікна, показаного на рисунку 4.44.

– У вікні списку всіх виводів вибрати один або кілька виводів та у контекстному меню вибрати пункт **Node Properties**.

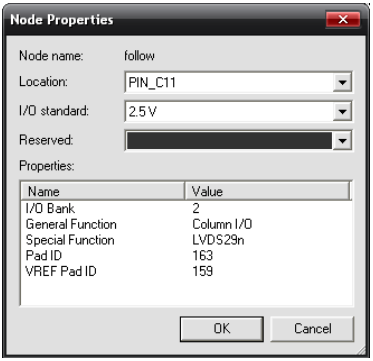



*Рисунок 4.44 – Вікно властивостей виводу*

Вікно властивостей містить наступні поля:

**Location** – вказується розташування виводу. Для декількох виводів може вказуватися банк елементів введення – виведення.

**I/O standart** – стандарт введення – виведення.

**Reserved** – зарезервований. Використовується в тому випадку, коли необхідно даний вивід зарезервувати на майбутнє.

**Properties** – список властивостей виведення.

5. У відкритому вікні необхідно вибрати потрібний стандарт введення-виведення. Для мікросхем сімейства Cyclone стандарт вводу-виводу – 3,3 V LVTTL.

### Перевірка правильності призначень виводів

Перевірка правильності призначення виводів робиться або автоматично при повній компіляції проекту, або запуском модуля компілятора, відповідального за таку перевірку.

Для перевірки призначення виводів необхідно вибрати пункт меню **Processing ⇒ Start ⇒ Start I/O Assignment Analysis** або натиснути кнопку  на панелі інструментів планувальника виводів. Це дозволить перевірити правильність призначень без повної компіляції. Для запуску перевірки призначення виводів необхідно тільки зробити призначення виводів (напрямок передачі даних, стандарти введення – виведення, банки).

Результати перевірки можуть бути переглянуті в розділі Fitter звіту компіляції. Для цього можна скористатися пунктами **Pin–Out File, Resource Section**. Також необхідно ще раз перевірити стан невикористаних виводів, що можна подивитись у закладці **Unused Pins** діалогу **Device and Pin Options** з закладки **Assignments ⇒Device**.

## *4.6. Часовий аналіз в Quartus II*

Часовий аналіз у пакеті Quartus II може бути виконаний або класичним методом (**Classic Timing Analysis**) або за допомогою

програми **TimeQuest**. У цьому розділі буде розглянутий класичний часовий аналізатор.

Часовий аналіз є складовою частиною повної компіляції. Результати часового аналізу дозволяють оцінити швидкодію проекту та виявити найбільш критичні ділянки, які потім можна оптимізувати схемотехнічно або задати їм інші опції компілятора. Повторна компіляція проекту покаже зміну часових параметрів змінених блоків.

Параметри часового аналізатора встановлюються за допомогою закладки **Classic Timing Analyzer Settings** діалогу **Settings** (меню **Assignments ⇒ Settings...**), яка показана на рисунку 4.45.

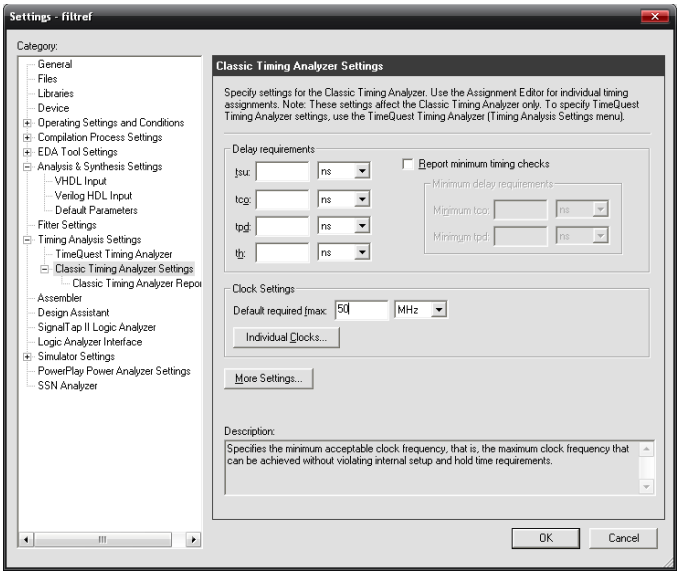



*Рисунок 4.45 – Встановлення часових параметрів*

При роботі часовий аналізатор використовує наступні налагодження пакета:

**$f_{MAX}$** – максимальна тактова частота;

**$t_{SU}$** – час передвстановлення (setup time) для регістрів;

**$t_{PD}$** – час затримки проходження сигналу від входу через внутрішню комбінаційну логіку до виходу (pin-to-pin delay);

**$t_{CO}$** – затримка вихідного сигналу щодо тактової частоти (clock to output delay);

**$t_H$** – час утримання (clock hold time).

Обмеження часових параметрів числовими значеннями збільшує час компіляції, оскільки Quartus II спочатку виконує компіляцію проекту, потім перевіряє його на відповідність встановленим параметрам і, при наявності невідповідності, виконує оптимізацію проекту.

Для перегляду результатів часового аналізу необхідно відкрити вікно результатів компіляції та вибрати пункт **Timing Analyzer** (рисунок 4.46).

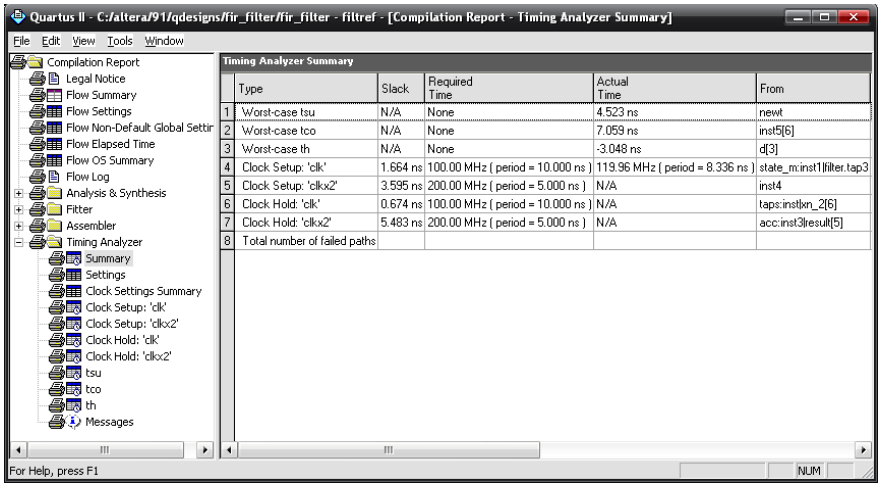



*Рисунок 4.46 – Результати часового аналізу*

Дана закладка результатів компіляції містить наступні пункти:

**Summary** – загальне зведення результатів часового аналізу. Тут вказуються найгірші з можливих значень показників (worst–case) часу передустановки, утримання, максимальне значення тактових частот і т.п.

**Settings** – налагодження, які використовуються в процесі часового аналізу. Для їхньої зміни необхідно використовувати діалог **Settings**.

– **Clock Settings Summary** – збірка установок по всіх тактових частотах, які використовуються в проекті.

– **Clock Setup** – час предвстановлення сигналів даних, відносно сигналу тактової частоти. Відображається в полі **Slack**. Максимально можлива частота для даного ланцюга відображається в полі **Actual fmax**.

**Clock Hold** – час утримання сигналів даних відносно періоду тактової частоти.

– **tsu, tco, th** – часи передустановки, тактовий сигнал-вихід та утримання для обраних вузлів схеми.

– **Messages** – повідомлення, що з'являються в результаті часового аналізу.

Для додавання вузлів у таблицю часового аналізу необхідно вибрати пункт **Advanced List Paths...** у контекстному меню (рисунок 4.47).

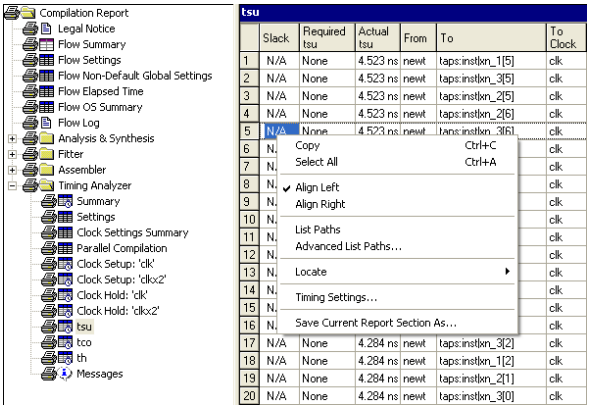



*Рисунок 4.47 – Перегляд результатів часового аналізу*

Вікно, котре відкрилося (рисунок 4.48) містить діалог, у якому необхідно вказати:

– Кількість шляхів, які додаються в схему – **Path to report**.

– Джерело сигналу (**From**) та його одержувач (**To**).

– Таблиці, у яких буде проводитися аналіз (**Report window sections**).

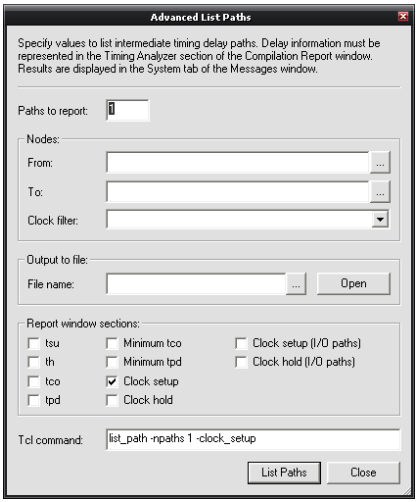



*Рисунок 4.48 – Додавання шляхів до часового аналізу*

## *4.7. Побудова часових діаграм проекту*

Симуляція проекту – це побудова його часових діаграм роботи. Це дозволяє промоделювати роботу системи при певних вхідних впливах, визначити значення внутрішніх та вихідних сигналів проекту.

Пакет Quartus II підтримує наступні методи визначення вхідних сигналів для симуляції:

– VWF (vector waveform file) – файл із часовими діаграмами;

– CVWF (compressed vector waveform file) – стисла бінарна версія VWF файлу;

– VCD (value charge dump file), VEC (vector file) – текстове подання файлів;

– TCL скрипт.

Розглянемо побудову VWF файлу часових діаграм.

## Встановленя параметрів симуляції

Установка параметрів симуляції робиться у вікні, що відкривається з меню **Assignments ⇒ Settings ⇒ Simulator Settings** (рисунок 4.49).

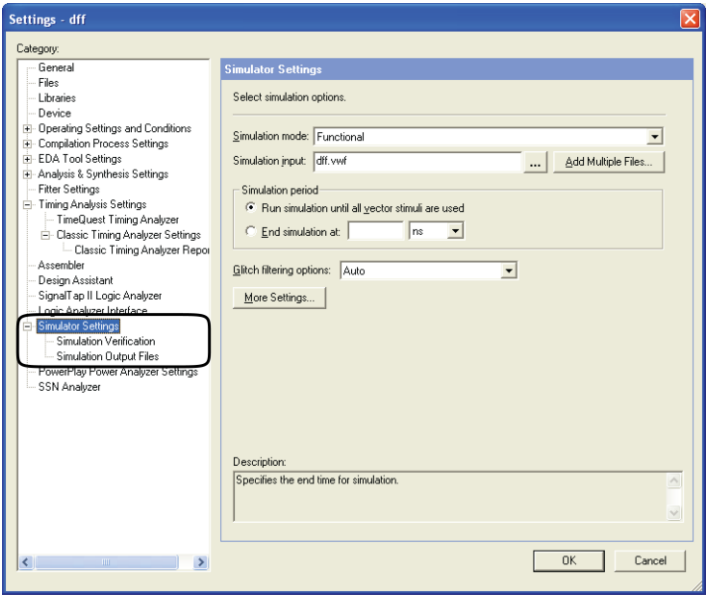



*Рисунок 4.49 – Визначення опцій симулятора*

В полі **Simulation mode** (метод симуляції) можна вибрати один з варіантів:

– **Functional** – функціональна симуляція, при якій не враховуються затримки на елементах схеми, а перевіряється тільки працездатність алгоритму в цілому.

– **Timing** – симуляція з урахуванням реальних затримок сигналу у вузлах мікросхеми.

– **Timing using Fast Timing Model** – симуляція з урахуванням найбільш сприятливої ситуації швидкодії мікросхеми.

Поле **Simulation input** – задається файл, що містить вектори вхідних сигналів.

Поле **Simulation period** (час симуляції) – встановлює кінцевий час симуляції до тих пір, поки існують сигнали (**Run simulation until all vector stimuli are used**) або до якогось певного часу (**End simulation at**).

Відзначимо також пункт **Glitch detection,** що дозволяє визначати "голки" в сигналах заданої тривалості та виводить повідомлення про них у вікно повідомлень.

Для створення нового файлу часових діаграм необхідно вибрати пункт меню **File ⇒ New ⇒ Vector Waveform File** (закладка **Other Files**).

### 4.7.1. Додавання діаграм у файл

Для додавання часових діаграм необхідно вибрати пункт меню **Edit ⇒ Insert ⇒ Insert Node or Bus …** або зробити подвійне клацання в полі **Name**. Відкриється вікно, показане на рисунку 4.50.

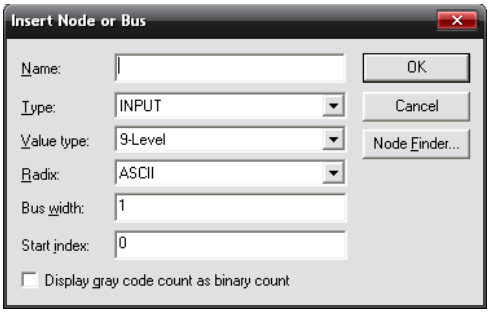



*Рисунок 4.50 – Додавання виводу в файл симуляції*

Вікно містить наступні поля:

1. **Name** – ім'я виводу або шини.

2. **Type** – тип виводу або виводів. Можливий вибір одного з наступних типів:

– **Bidir** – двонаправлений,

– **Buried** – внутрішній,

– **Comb** – комбінаційний, тобто отриманий в результаті роботи комбінаційної схеми,

– **Input** – вхід,

– **Machine** – стан цифрового автомата,

– **Memory** – вміст пам'яті,

– **Output** – вихід,

– **Reg** – регістровий вивід, тобто вихід регістра або тригера.

**Value type** – тип значень, які може приймати сигнал:

**9–Level** або **Enum** – типи даних мов VHDL та Verilog. Опис типу 9–Level наведений в главі 2 при описі типу *std_logic*.

3. **Radix** – система числення, у якій буде відображатися результат:

– **ASCII** – коди таблиці ASCII;

– **Binary** – двійковий код;

– **Fractional** – дробове число;

– **Hexadecimal** – шістнадцятковий код;

– **Octal** – вісімковий код;

– **Signed Decimal** – знакове десяткове;

– **Unsigned Decimal** – беззнакове десяткове.

4**. Bus width** – розрядність шини. Якщо задається шина, то можна вказати її розрядність, якщо провідник – то це поле недоступне.

5. **Start index** – початковий індекс нумерації виводів шини. Можливе використання будь–якого цілого числа.

6. **Display gray code count as binary count** – відображення коду Грея у вигляді звичайного двійкового коду.

Для введення даних зручніше користуватися діалогом, який відкривається при натисканні на кнопку **Node Finder...** (рисунок 4.51).

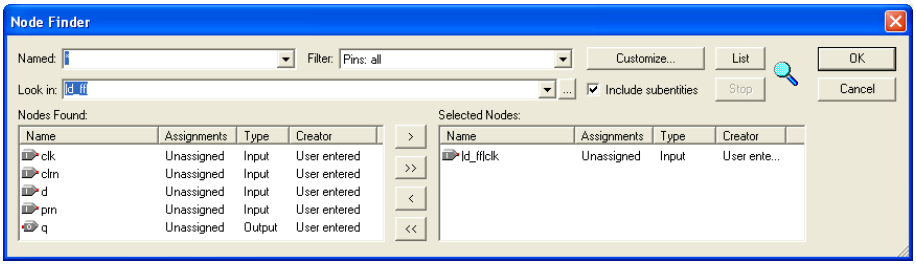



*Рисунок 4.51 – Пошук сигналів за допомогою Node Finder*

Для вставки сигналів або шин необхідно виконати наступні дії:

1. Використати фільтр (**Filter**). В списку, потрібно вибрати потрібний тип виводу:

– **Pins: input** – знаходить всі входи проекту;

– **Pins: output** – знаходить всі виходи проекту;

– **Pins: bi–directional** – знаходить двонаправлені виводи проекту;

– **Pins: all** – всі виводи проекту;

– **Pins: virtual** – віртуальні виводи – використовуються при інкрементальній організації проекту.

– **Registers: pre-synthesis** – всі виходи регістрів та тригерів, які існують в проекті після проведення операції **Analysis & Synthesis** до этапу фізичного синтезу.

– **Registers: post-fitting** – всі виходи регістрів та тригерів, які існують в проекті після этапу фізичного синтезу **Fitting.**

– **Design Entry (all names)** – всі імена, що були введені користувачем.

– **Post-Compilation** – всі вузли, які залишились після оптимізації проекту.

– **SignalTap II** – виводи для логічного аналізатора SignalTap II.

– **SignalProbe** – виводи, які використовуються в SignalProbe.

2. Натиснути кнопку **List**.

3. В полі **Node Found** буде відображений список сигналів, відібраних за допомогою фільтра.

4. Вибрати потрібні сигнали й натиснути кнопку >. Обрані сигнали з'являться в списку **Selected Nodes**.

5. Натиснути кнопку **Ok**.

Далі необхідно встановити кінцевий час моделювання (рисунок 4.52). Для цього потрібно вибрати пункт меню **Edit ⇒ End Time ...** У вікні **End Time** в полі **Time** встановити кінцевий час моделювання. При цьому використовуються наступні одиниці виміру часу: s – с, ms – мс, us – мкс, ns – нс, ps – пс.

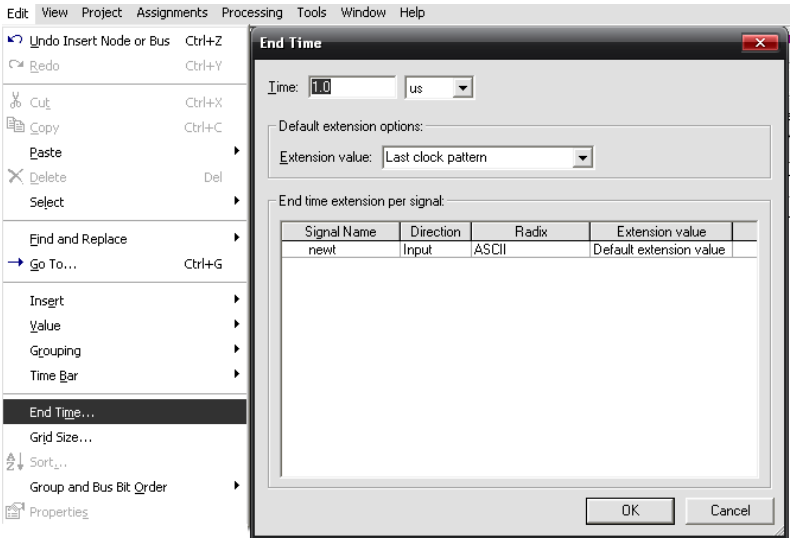



*Рисунок 4.52 – Встановлення кінцевого часу симуляції*

## 4.7.2. Встановлення параметрів сигналів

В результаті дій, описаних вище, вікно буде мати вигляд, що наведений на рисунку 4.53. Призначення кнопок панелі інструментів, яка знаходиться ліворуч від часових діаграм, наведено в таблиці 4.4.

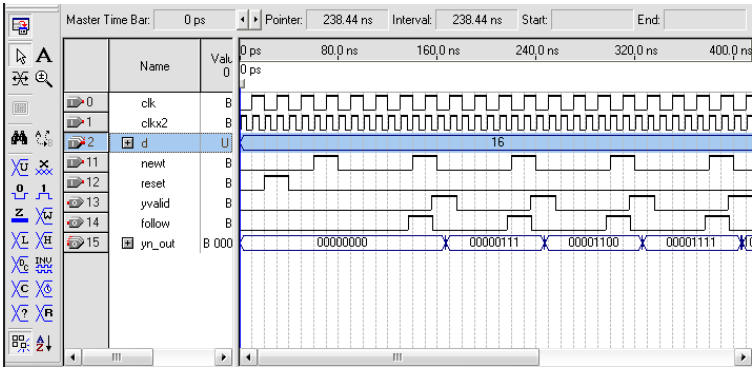



*Рисунок 4.53 – Вікно симулятора*

Таблиця 4.4 – Призначення кнопок на панелі інструментів

| Кнопка | Виконувана функція |
|---|---|
|  | Виділення частини часових діаграм |
|  | Вставка коментарю до діаграм |
|  | Uninitialized – неініціалізовано |
|  | Forcing unknown – сильний невідомий сигнал |
|  | Forcing low – "сильний" нуль |
|  | Forcing high – "сильна" одиниця |
|  | High impedance – третій стан |
|  | Weak unknown – слабкий невідомий сигнал |
|  | Weak low – "слабкий" нуль |
|  | Weak high – "слабка" одиниця |

| Кнопка | Виконувана функція |
|---|---|
| | Don't care – без різниці |
| | Invert – інверсія сигналу |
| | Count value – лічильник (див. нижче) |
| | Clock – тактові імпульси (див. нижче) |
| | Arbitrary value – значення групи |
| | Random – випадковий сигнал |

Побудова часових діаграм може бути виконана декількома методами:

1. Виділення частини однієї або декількох діаграм за допомогою лівої кнопки миші та задання необхідного значення за допомогою кнопок панелі інструментів (таблиця 4.4).
2. Виділення всієї діаграми натисканням на кнопку  та задання значення за допомогою панелі інструментів.

Для задання послідовності тактових імпульсів потрібно натиснути кнопку  на панелі інструментів. В результаті буде відкрите діалогове вікно, показане на рисунку 4.54. Воно містить наступні поля:

1. **Time range** – діапазон часу, до якого застосовуються параметри:
   - **Start time** – час початку.
   - **End time** – час закінчення.
2. **Base waveform on** – параметри сигналу:
   - **Time period** – період сигналу
   - **Period** – тривалість періоду.
   - **Offset** – зсув початку імпульсу відносно початку періоду.
   - **Duty cycle (%)** – шпаруватість.

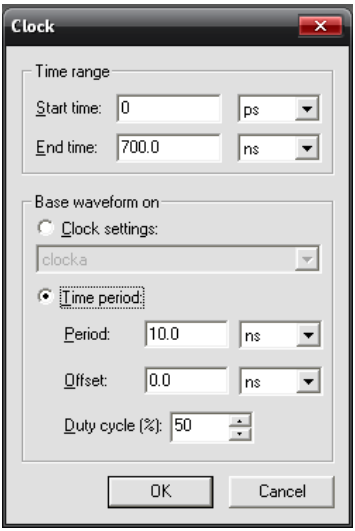



*Рисунок 4.54 – Встановлення параметрів тактового сигналу*

Для завдання часових діаграм послідовності чисел з певним законом зміни використовують діалогове вікно **Count Value** (рисунок 4.55). Перша закладка – **Radix** – містить наступні поля:

- **Radix** – система числення;
- **Start value** – початкове значення;
- **End value** – кінцеве значення;
- **Increment by**: коефіцієнт рахування;
- **Count type – binary** – двійковий код, **gray code** – код Грея.

Друга закладка (**Timing**) містить наступні поля:

- **Start time, End time** – опис аналогічний наведеному вище;
- **Count every** – час, через який будуть відбуватися зміни сигналу на наступні значення.
- **Multiplied by** – коефіцієнт множення, тобто на скільки потрібно помножити число в полі **Count every**, щоб визначити час наступної зміни.

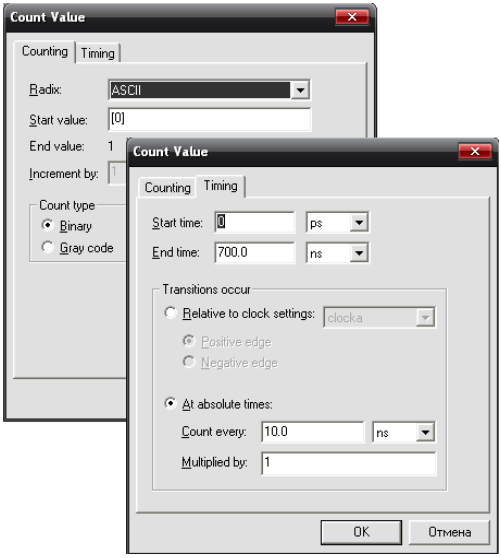



*Рисунок 4.55 – Встановлення параметрів групи сигналів*

В тому випадку, коли симуляції використовуються групи сигналів, можна результати симуляції представити у вигляді аналогового сигналу (рисунок 4.56). Для цього необхідно виділити потрібну групу сигналів і у контекстному меню або у меню **View** обрати пункт **Display Format → Analog Waveform**.

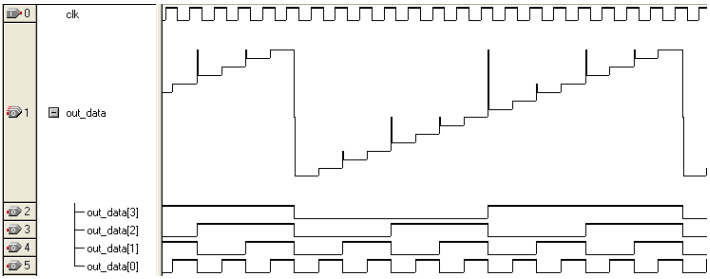



*Рисунок 4.56 – Аналогове представлення сигналів*

# 4.8. Програмування та конфігурування в Quartus II

## Визначення режимів роботи мікросхеми

Перед проведенням операції конфігурування необхідно зробити призначення режимів роботи мікросхеми в процесі конфігурації та після її закінчення. Для цього необхідно вибрати пункт меню **Assignments ⇒ Device ⇒ Device & Pin Options...** (рисунок 4.57).

Дане вікно має велику кількість вкладок. Розглянемо найбільш необхідні з них:

– **General** – загальні параметри (рисунок 4.58). У цій закладці необхідно зробити активним опцію **Auto–restart configuration after error** – авто рестарт при помилці конфігурування. У цьому випадку мікросхема буде заново переконфігурована у випадку помилки завантаження конфігурації. У випадку відсутності даної опції при помилці конфігурування необхідно буде робити запуск процесу конфігурації зовнішнім сигналом.

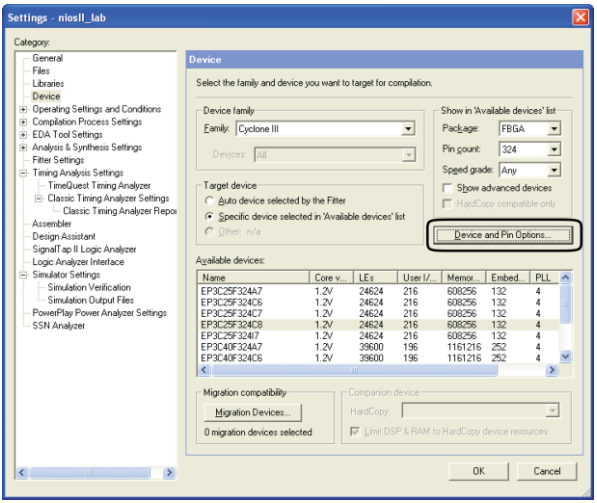



*Рисунок 4.57 – Встановлення додаткових параметрів ПЛІС*

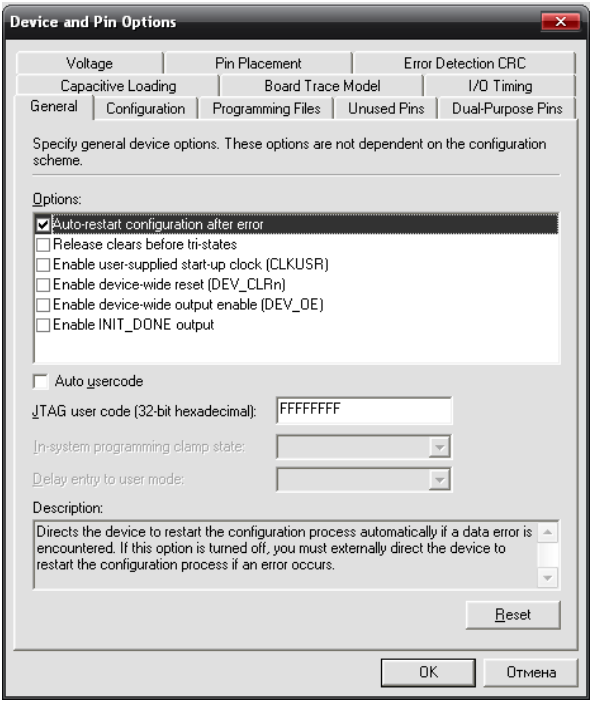



*Рисунок 4.58 – Встановлення режимів роботи мікросхеми після компіляції*

**Unused pins** – виводи, які не використовуються. Визначає в якому режимі будуть перебувати невикористовувані виводи мікросхеми після програмування. З доступних варіантів необхідно вибрати **As inputs tri–stated** – як входи із трьома станами. Тобто всі невикористовувані виводи будуть працювати на введення інформації в ПЛІС і переключені в третій стан.

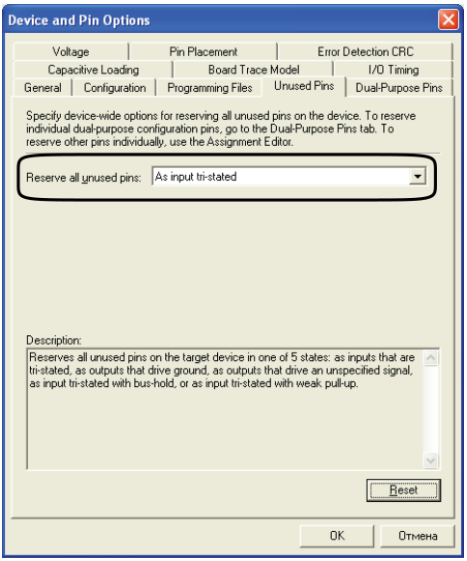



*Рисунок 4.59 – Встановлення правил роботи з непід'єднаними виводами*

## Конфігурування ПЛІС

При конфігуруванні ПЛІС необхідно використати конфігураційний файл, для створення якого використовується модуль **Assembler**. Для його запуску необхідно або виконати повну компіляцію проекту або вибрати пункт меню **Processing ⇒ Start ⇒ Start Assembler**. Файли створені в процесі роботи модуля **Assembler,** необхідно завантажувати в ПЛІС за допомогою модуля **Programmer**.

У результаті буде відкрите вікно програматора, яке показано на рисунку 4.60. Робоче поле модуля **Programmer** містить наступні опції програмування:

– **Program/configure** – програмування або конфігурування мікросхеми.

– **Verify, Blank–check, Examine, Erase** – перевірка конфігурації в мікросхемі, стирання, контроль стирання конфігурації –

для мікросхем сімейств MAX II, MAX 7000, MAX 3000 та конфігураційних ПЗП.

– **Security bit** – встановлення біта таємності для мікросхем сімейств MAX II, MAX 7000, MAX 3000.

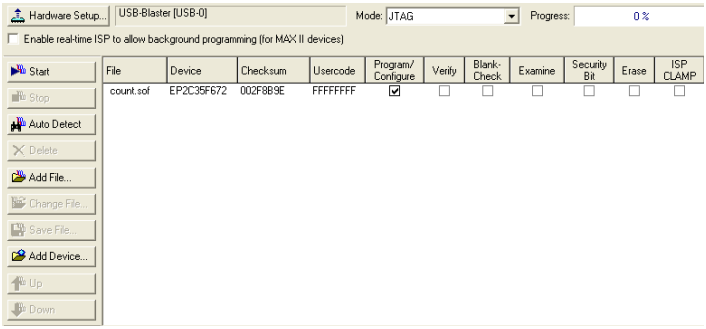



*Рисунок 4.60 – Вікно програматора*

Даний модуль також дозволяє визначити використовуваний завантажувальний кабель (рисунок 4.61).

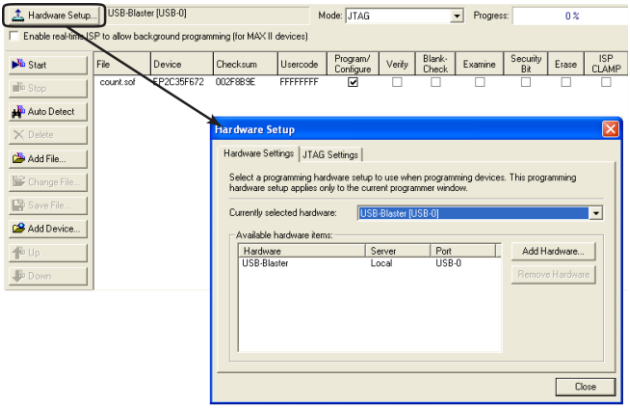



*Рисунок 4.61 – Додавання конфігураційного кабелю*

При роботі з пакетом Quartus II можливе використання наступних завантажувальних кабелів: Byteblaster II, Byteblaster MV, Usb-blaster, Masterblaster cable. Ці кабелі можуть підключатися до різних апаратних портів вводу–виводу персонального комп'ютера: LPT, COM, USB.

При першому ввімкненні модуля Programmer необхідно зробити вибір використовуваного завантажувального кабелю. Для цього необхідно натиснути кнопку **Hardware Setup...** У вікні, що відкрилося, можна вибрати підключений кабель зі списку **Currently selected hardware** (поточний обраний апаратний засіб). Для додавання обраного завантажувального кабелю необхідно натиснути кнопку **Add Hardware...** Подальші операції з конфігурування містять в собі наступні етапи:

– Вибір необхідного конфігураційного файлу (за замовчуванням відкривається файл, необхідний для завантаження в обрану мікросхему. Тому даний пункт найчастіше немає необхідності виконувати).
– Установка опції **Program/Configure.**
– Натискання кнопки **Start**.

## *Література*

1. Комолов Д.А., Мяльк Р.А., Зобенко А.А., Филлипов А.С. Системы автоматизированного проектирования фирмы Altera MAX+plus II и Quartus II. Краткое описание и самоучитель. – М.: РадиоСофт, 2002. – 352 с.
2. Altera Quartus II Software v12.0 — Subscription Edition vs. Web Edition. http://www.altera.com/literature/po/ss_quartussevswe.pdf
3. Quartus II Handbook. Version 9.1. Altera, 2009. – 1820 p.
4. Design Examples http://www.altera.com/support/examples/exm-list.jsp
5. http://www.naliwator.narod.ru/index.html

# Розділ 5

# Розробка систем в пакеті Quartus II


- Створення ієрархічного проекту ..... 231
- Стилі проектування..... 241
- Реалізація запам'ятовуючих пристроїв в ПЛІС ..... 242
- Робота з цифровими автоматами. ..... 251
- Оптимізація проектів в пакеті Quartus II. ..... 259
- Засоби внутрішньосистемного налагоджування для ПЛІС Altera ..... 287

# 5.1. Створення ієрархічного проекту

Ієрархічна будова проекту передбачає наявність в проекті декількох частин, які зв'язані в одне ціле за допомогою файлу верхнього рівня. Ієрархія проекту в пакеті Quartus II відображається у вікні навігатора проекту (Project Navigator) закладка (Hierarchy) – рисунок 5.1. Ім'я файлу верхнього рівня зазвичай збігається з іменем проекту.

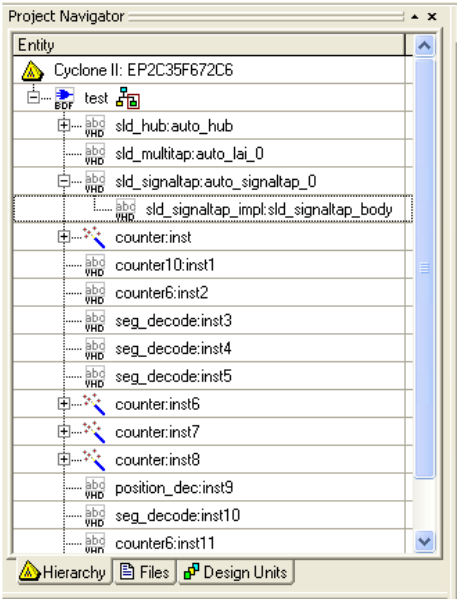



*Рисунок 5.2 – Ієрархія проекту*

### 5.1.1. Робота з каналами (conduit)

При роботі з різними об’єктами в Quartus II необхідно використовувати відповідні провідники для з'єднання частин проекту між собою. Quartus II містить наступні типи провідників: провід **wire**, шина **bus** та канал **conduit**. При їх використанні необхідно дотримуватись правил з'єднання, наведених в таблиці 5.1.

Таблиця 5.1 – Правила з'єднання в Quartus II

| Об’єкти для з'єднання | Тип лінії зв’язку |
|---|---|
| Блок та вивід | Канал |
| Блок та символ | Провід або шина |
| Блок та блок | Канал |
| Вивід та символ | Провід або шина |
| Символ та символ | Провід або шина |

На рисунку 5.2 показаний приклад підключення сигналу (key_clk) та шини (control[1..0]) до каналу.

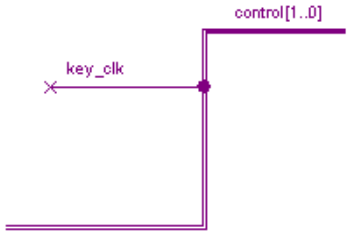



*Рисунок 5.3 – Канал з підключеними шиною та проводом*

Для визначення властивостей каналу у контекстному меню необхідно обрати пункт **Conduit Properties**

Вкладка загальних властивостей (**General**) визначає ім'я каналу в полі **Conduit name**. Його можна не вводити.

Вкладка **Signals** (сигнали) визначає які сигнали проходять через канал і до яких портів у блоці приєднані ці сигнали (рисунок 5.3).

Для додавання сигналу необхідно в полі **Signal** потрібно ввести його ім'я. Після натискання кнопки **Add** цей сигнал з'явиться в таблиці **Connections** – з'єднання. В тому випадку, коли сигнал приєднаний до блоку, то в таблиці **Connections** у рядку з іменем сигналу з'явиться також і ім'я порту блоку.

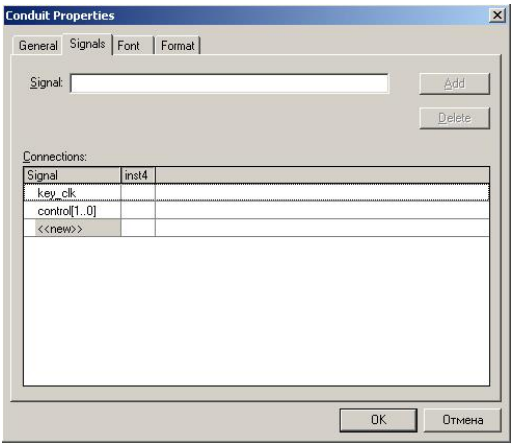



*Рисунок 5.4 – Додавання сигналів до каналу*

Закладки **Font** (шрифт) та **Format** (формат) дозволяють редагувати зовнішній вигляд каналу.

## 5.1.2. Робота з блоками

Блок разом з символами елементів представляє графічне зображення якої-небудь частини схеми. Для створення блоку необхідно натиснути кнопку  на панелі інструментів, а потім за допомогою миші розташувати блок на вільному полі в редакторі графічних файлів. В результаті буде отримана рамка з пустою таблицею (рисунок 5.4). Для використання блока в описі проекту

необхідно призначити йому ім'я, порти, а також приєднати порти блоку до схеми.

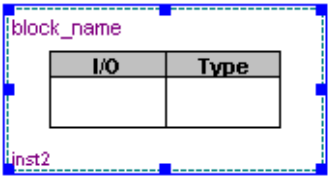



*Рисунок 5.5 – Символ нового блоку*

Призначення імені блоку та його портів проводиться за допомогою контекстного меню (рисунок 5.5). Перша вкладка **General** (рисунок 5.6) містить два поля: **Name** (ім'я блоку) та **Instance name** (ім'я елементу графічного файлу). Для коректної роботи пакету можна не змінювати значення у полях, але для більшої зручності в роботі з проектом бажано в полі **Name** ввести таке ім'я, яке відповідає функції блоку. Поле **Instance name** можна не змінювати.

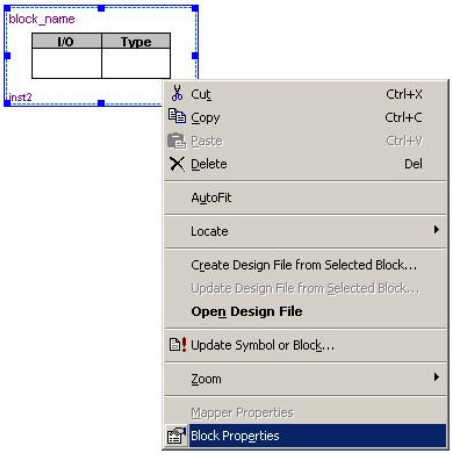



*Рисунок 5.6 – Контекстне меню блока*

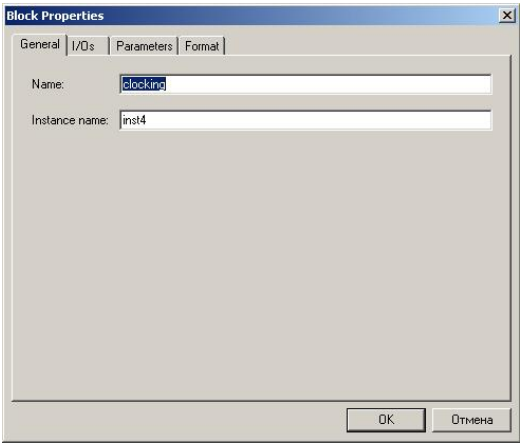



*Рисунок 5.7 – Перша вкладка меню властивостей блока*

Друга вкладка **I/Os** (рисунок 5.7) містить групу полів **I/O** з полями **Name** (ім'я порту) та **Type** (тип порту). Типи портів бувають **Input** (вхід), **Output** (вихід), **Bidir** (двонаправлений порт).

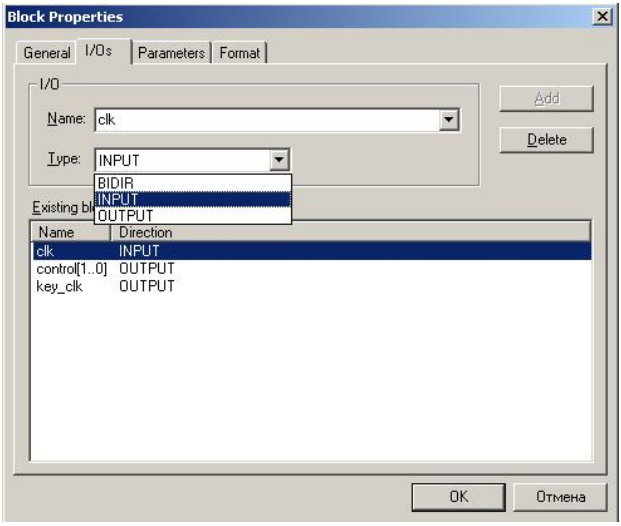



*Рисунок 5.8 – Друга вкладка меню властивостей блока*

У таблиці **Existing block I/Os** відображаються ті порти, які вже є в даному блоці.

Для додавання порту до блоку необхідно ввести його ім'я в полі **Name**, за допомогою списку **Type** визначити напрямок передачі даних і натиснути кнопку **Add**. Новий порт з'явиться у таблиці **Existing block I/Os**.

Вкладка **Parameters** – параметри – містить числові значення параметрів блоку, які потрібні при використанні параметризованих блоків, тобто блоків зі змінною структурою.

Вкладка **Format** дозволяє змінювати стиль оформлення блоку.

В результаті роботи з діалогом властивостей буде отриманий блок, показаний на рисунку 5.8.

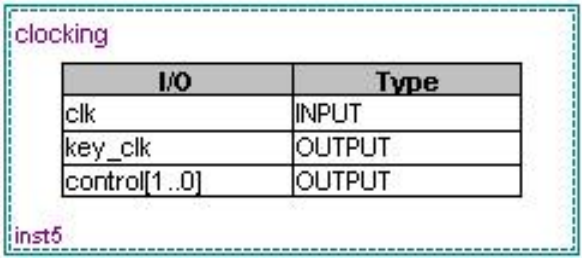



*Рисунок 5.9 – Вигляд створеного блоку*

Блок можна використати для створення файлу нижчого рівня. Для цього в контекстному меню блока потрібно вибрати пункт **Create Design File from Selected Block …** – створити новий файл проекту з обраного блоку. Це призведе до появи діалогу, наведеного на рисунку 5.9.

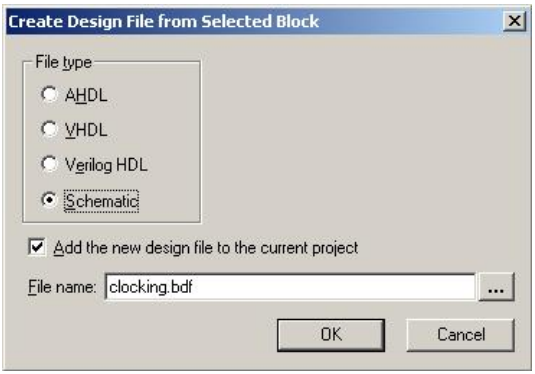



*Рисунок 5.10 – Діалог створення нового файлу*

В цьому діалозі можна побачити такі поля:

**File type** – тип нового файлу:

– **AHDL**, **VHDL**, **Verilog HDL** – файли на одній з мов опису апаратури. В цьому випадку буде створена інтерфейсна частина файлу.

– **Schematic** – файл графічного редактора.

**Add the new design file to the current project** – додати новий файл до проекту.

**File name** – ім'я нового файлу. Два останніх поля краще залишити такими, які встановлено за замовчуванням.

Результатом роботи з наведеним діалогом буде створення нового файлу з портами (рисунок 5.10).

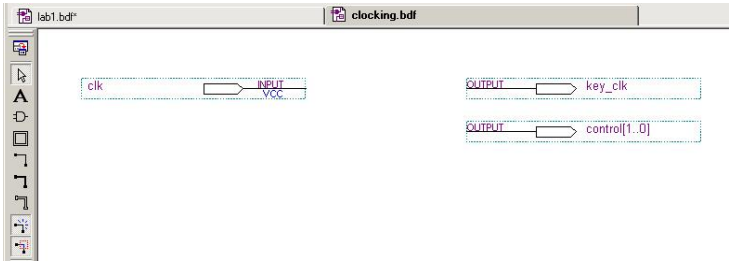



*Рисунок 5.11 – Файл, створений з блоку*

На рисунку 5.11 показане вікно зі створеним файлом на мові VHDL. Отриманий модуль буде мати такі самі порти, як і блок з верхнього рівня ієрархії. Змінювати порти не можна, оскільки назви портів блоку на верхньому рівні та файлу нижнього рівня повинні співпадати.

```
1   // WARNING: Do NOT edit the input and output ports in this file in a text
2   // editor if you plan to continue editing the block that represents it in
3   // the Block Editor! File corruption is VERY likely to occur.
4   
5   //  Module Declaration
6   module clocking
7   (
8       // ((ALTERA_ARGS_BEGIN)) DO NOT REMOVE THIS LINE!
9       clk, key_clk, control
10      // ((ALTERA_ARGS_END)) DO NOT REMOVE THIS LINE!
11  );
12  // Port Declaration
13  
14      // ((ALTERA_IO_BEGIN)) DO NOT REMOVE THIS LINE!
15      input clk;
16      output key_clk;
17      output [1:0] control;
18      // ((ALTERA_IO_END)) DO NOT REMOVE THIS LINE!
19  
20  
21  
22  endmodule
23
```

*Рисунок 5.11 – Згенерований текст на мові VHDL*

## 5.1.3. Приєднання блоку до схеми

Для приєднання блоку до схеми необхідно перейти в режим рисування проводу, шини або каналу та провести з'єднувальні ліній до блоку. В результаті на кордоні блоку з'являться перетворювачі сигналу (**mapper**) зеленого кольору, що працюють в режимі двонаправленого порту (рисунок 5.12).

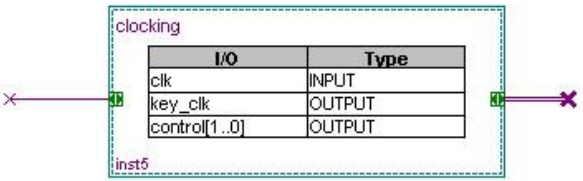



*Рисунок 5.12 – Блок з створеними перетворювачами сигналу*

***Увага! Для роботи з перетворювачем сигналу необхідно задавати імена сигналів на провідниках, що приєднані до блоку.***

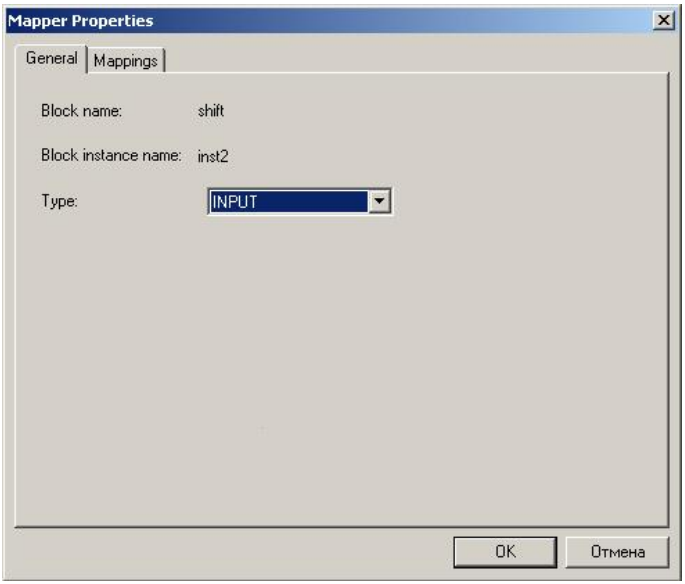



*Рисунок 5.13 – Перша вкладка властивостей перетворювача сигналу*

Для визначення сигналів, що приєднані до блоку через конкретний перетворювач сигналу необхідно підвести курсор до перетворювача і клацнути на ньому правою кнопкою миші. З контекстного меню потрібно вибрати пункт властивостей перетворювача (**Mapper Properties**). Це призведе до відкриття діалогу, наведеного на рисунку 5.13. Перша вкладка **General** дозволяє визначити напрямок сигналу – вхід, вихід або двонаправлений сигнал.

Друга вкладка **Mappings** (рисунок 5.14) дозволяє визначити відповідність сигналів на провідниках каналу та сигналів у блоці.

Результат роботи з властивостями перетворювача сигналів показаний на рисунку 5.15.

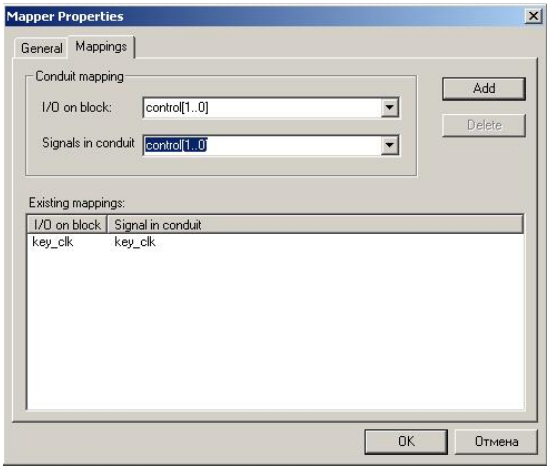



*Рисунок 5.14 – Друга вкладка властивостей перетворювача сигналу*

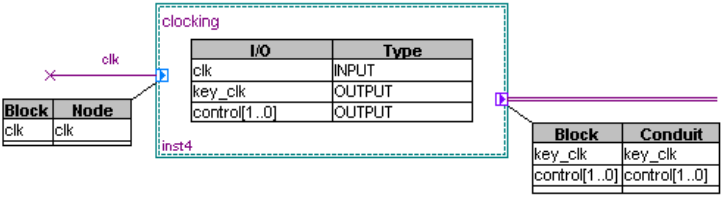



*Рисунок 5.15 – Блок з під'єднаними провідниками*

## 5.1.3. Створення символу для файлу

Якщо один з файлів проекту необхідно використовувати як частину більш складного графічного файлу, то в пакеті Quartus II є можливість створення графічного символу для відповідного файлу. Для цього потрібно відкрити необхідний файл. В меню **File** пункті **Create/Update** необхідно вибрати пункт **Create Symbol Files for Current File**. В результаті новий файл символу буде створено і розміщено в папці проекту.

## 5.2. Стилі проектування

При створенні проекту може використовуватись два стилі проектування: "знизу-вгору" (Bottom-Up) або "згори-донизу" (Top-Down).

Перший стиль проектування передбачає створення складових частин проекту, які потім складаються в більш загальні і на верхньому рівні об'єднуються файлом верхнього рівня. Цей стиль використовують в тому випадку, коли розробка проекту виконується однією людиною і розмір проекту порівняно малий. Вся інформація про алгоритм роботи міститься в одному проекті, а управління проектом виконується дуже легко. Однак, при зміні будь-якої з частин або параметрів проекту весь проект компілюють заново, а це може змінити параметри вже відкомпільованих частин.

При використанні стилю "згори-донизу" спочатку створюють файл верхнього рівня, який потім розбивають на складові частини і деталізують алгоритм роботи. Цей стиль зручно використовувати при великих проектах, над якими працює група розробників. Це дає можливість одночасно і незалежно працювати над декількома частинами проекту, включаючи їх компіляцію і розміщення на кристалі. На верхньому рівні проектування виконується інтеграція окремих модулів у проект. Але для такого стилю розробки можлива ситуація, коли частини проекту, які працюють окремо одна від одної разом можуть не працювати.

При використанні стилю розробки "згори-донизу" можливо використання так званих LogicLock ділянок. Це прямокутна ділянка на кристалі мікросхеми, яка закріплена за окремою частиною проекту. Розмір та розташування цієї ділянки можуть змінюватись або автоматично програмним забезпеченням або розробником. Підтримка LogicLock ділянок можлива тільки в порівняно нових сімействах мікросхем фірми Altera: Stratix, Cyclone, MAX II, Arria.

Використання ділянок LogicLock та розробка проекту по частинах в пакеті Quartus II забезпечується режимом, який має назву Incremental Design. Такий стиль передбачає розподіл проекту на

окремі частини та окрему розробку кожної частини. При цьому процес відлагодження для кожної частини проходить окремо. Але оскільки проект компілюється не весь, а лише його частина, то час, витрачений на компіляцію зменшується приблизно на 40 відсотків. А якщо врахувати одночасну роботу декількох розробників, то загальний час на розробку проекту значно скорочується. При розподілі проекту на частини слід використовувати правила, які наведені у параграфі 5.5.1.

## *5.3. Реалізація запам'ятовуючих пристроїв в ПЛІС*

Більшість сучасних мікросхем ПЛІС містить на кристалі модулі пам'яті, які можуть використовуватись в різних формах: постійних або оперативних запам’ятовуючих пристроїв. Опис таких блоків наведено у параграфі 1.1.3.

Для визначення типу вбудованої пам'яті необхідно використовувати майстер **MegaWizard Plug–In Manager**. В пакеті Quartus II [7] є можливість декілька типів пам'яті, але їх кількість залежить від обраного сімейства мікросхем:

- FIFO – пам’ять типу «перший ввійшов – перший вийшов».
- FIFO partioner – інструмент для розміщення декількох блоків FIFO на одному кристалі ПЛІС.
- RAM: 1 PORT – однопортовий оперативний запам’ятовуючий пристрій (ОЗП).
- RAM: 2 PORT – двохпортовий синхронний ОЗП.
- RAM: 3 PORT – трьохпортовий ОЗП.
- ROM: 1 PORT – однопортовий постійний запам’ятовуючий пристрій (ПЗП).
- ROM: 2 PORT – двохпортовий ОЗП.
- Shift register (RAM-based) – регістр зсуву, виконаний на основі ПЗП.

### 5.3.1. Створення MIF-файлу

Для роботи з ПЗП необхідно дані, що будуть в нього записуватись представити у вигляді hex- або mif-файлів. В цьому параграфі опишемо створення mif-файлу, а також роботу в редакторі mif-файлів.

Для створення mif-файлу необхідно в меню **File** вибрати пункт **New**… Далі відкрити закладку **Other Files**, в якій вибрати пункт **Memory Initialization File** (рисунок 5.16).

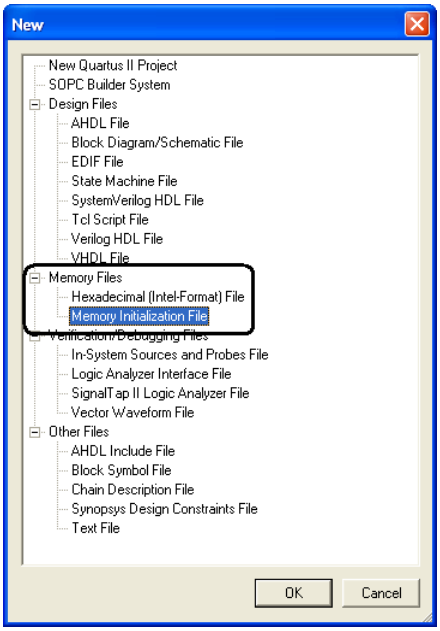



*Рисунок 5.16 – Створення mif-файлу*

В наступному вікні потрібно вказати кількість слів (**Number of words**) та розмір слова (**Word size**) – рисунок 5.17. В нашому випадку це 32 слова по 8 біт в кожному.

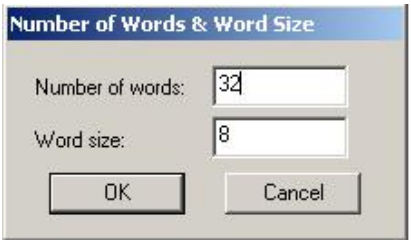



*Рисунок 5.17 – Параметри блока пам'яті*

Після цього відкриється редактор mif-файлів, який має вигляд, показаний на рисунку 5.18. Він має вигляд таблиці, в якій в заголовках рядків вказаний адрес чарунки в пам'яті, а в заголовках стовпців – зміщення відносно цього адресу. Сама таблиця містить інформацію ПЗП.

Mif1.mif

| Addr | +0 | +1 | +2 | +3 | +4 | +5 | +6 | +7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 24 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

*Рисунок 5.18- Редактор вмісту пам'яті*

При роботі з редактором є можливість змінювати його вигляд за допомогою пунктів меню **View** (рисунок 5.19): кількість чарунок у рядку (**Cells Per Row**), система числення для адреси пам'яті (**Address Radix**) та даних в пам'яті (**Memory Radix**). Останній пункт містить такі варіанти:

- **Binary** – двійковий код,
- **Hexadecimal** – шістнадцятковий код,
- **Octal** – вісімковий код,
- **Decimal** – десятковий код,
- **Signed Decimal** – знаковий десятковий код,
- **Unsigned Decimal** – беззнаковий десятковий код.

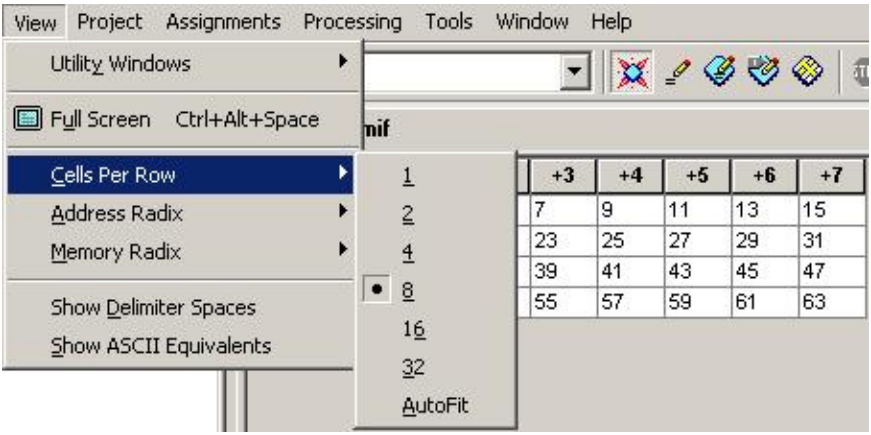



*Рисунок 5.19 – Зміна параметрів вигляду вмісту пам'яті*

Зміст чарунок в редакторі можна корегувати декількома способами:

1. Вибрати потрібну чарунку та записати туди необхідне значення.
2. Вибрати діапазон чарунок та з контекстного меню (рисунок 5.20) вибрати один з пунктів:
   - **Fill Cells with 0's** – заповнити чарунки нулями;
   - **Fill Cells with 1's** – заповнити чарунки одиницями;
   - **Custom Fill Cells** – редагування послідовностей для заповнення чарунок (див. нижче);
   - **Reverse Address Contents** – інверсія змісту чарунок.
3. Вставити значення чарунок з іншої таблиці або програми, наприклад, Excel.

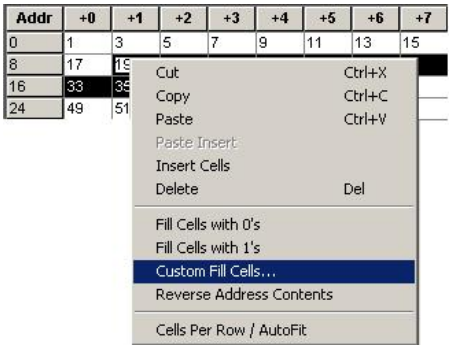



*Рисунок 5.20 – Контекстне меню редактора mif-файлів*

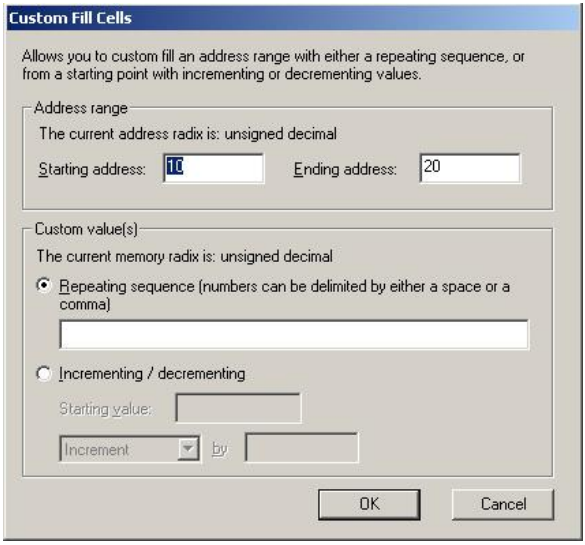



*Рисунок 5.21 – Заповнення пам'яті послідовністю чисел*

Діалог **Custom Fill Cells** дозволяє заповнювати чарунки пам'яті послідовними наборами чисел. Цей діалог містить наступні поля:

**Address range** – адресний діапазон чарунок, з якими буде проводитись робота. **Starting address** – початкова адреса, **Ending address** – кінцева адреса.

**Custom value(s)** – значення, які задає користувач.

**Repeating sequence** – послідовність, що повторюється. Числа в послідовності повинні розділятися пропуском або комою.

**Incrementing/decrementing** – інкремент або декремент числа. При цьому вказують початкове значення (**Starting value**) напрямок зміни чисел, а також крок зміни значень.

## 5.3.2. Створення блоку пам'яті за допомогою MegaWizard Plug–In Manager

Для створення нового блоку пам'яті необхідно використати **MegaWizard Plug–In Manager**. У відкритому вікні вибрати пункт **Memory Compiler**, а потім – вибрати необхідний тип модуля пам'яті. Вигляд вікна **MegaWizard Plug–In Manager** може відрізнятись від наведеного на рисунку 5.22 і залежить від обраного сімейства мікросхем (пункт **Which device family will you be using?**)

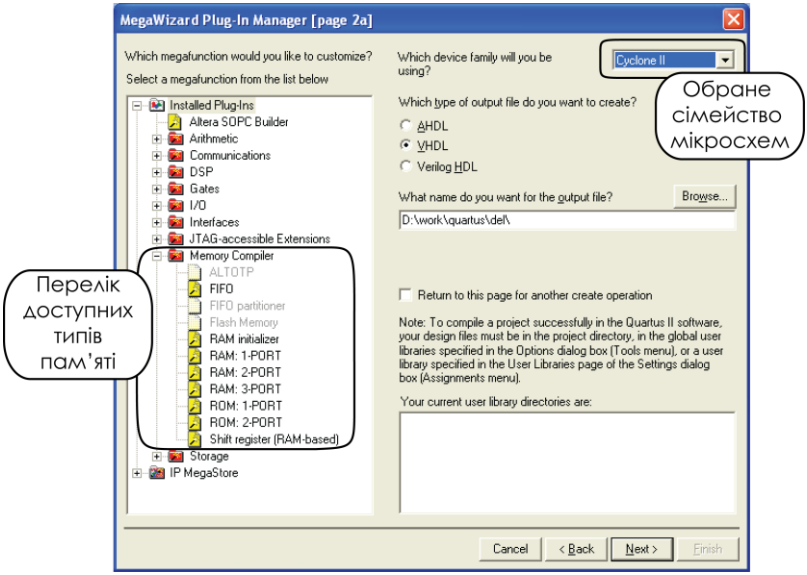



*Рисунок 5.22 – Вікно MegaWizard Plug-In Manager при створенні модуля пам'яті*

Після натискання кнопки **Next** відкриється діалог створення блока пам'яті. У цьому параграфі ми розглянемо створення блоку ОЗП для чого у відкритому вікні **MegaWizard Plug–In Manager** необхідно обрати **RAM: 1-PORT**. Буде відкрите перше вікно майстра створення блоку оперативної пам'яті (рисунок 5.23).

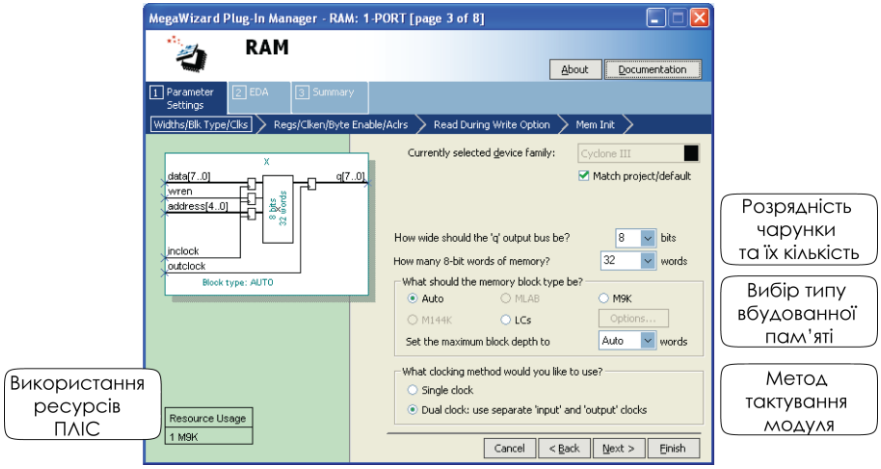



*Рисунок 5.23 – Перше вікно майстра проекту*

У різних вікнах MegaWizard Plug–In Manager потрібно визначити розрядність шини даних (вихід **'q'**), кількість чарунок в пам'яті (рисунок 5.23), наявність синхронізації по входу **'address'** та виходу **'q'** (рисунок 5.24), а також початковий зміст модуля (рисунок 5.25).

У першому вікні діалогу необхідно відмітити наступні поля:

**Currently selected device family** – обране сімейство мікросхем.

**How wide should the 'q' output bus be?** – якої розрядності повинен бути вихід?

**How many 8-bit words of memory?** – яка кількість слів міститься в пам'яті?

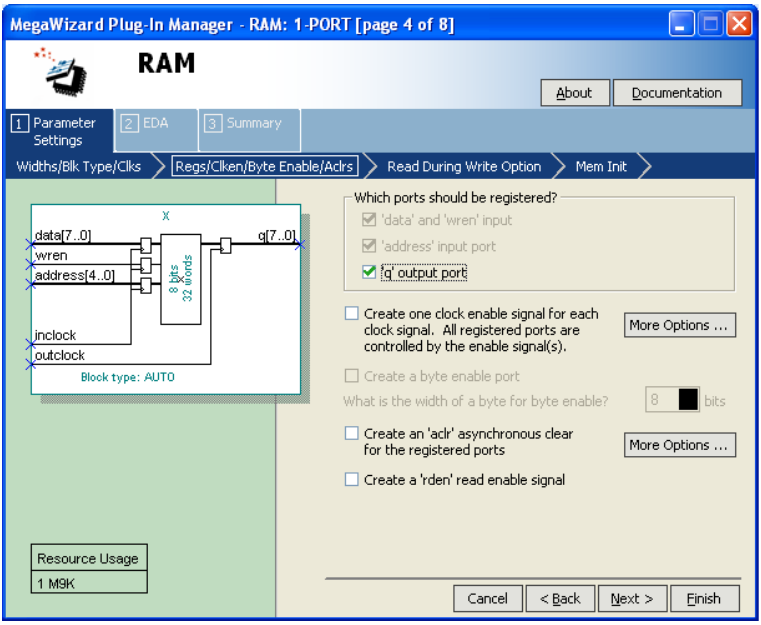



*Рисунок 5.24 – Друге вікно майстра проекту*

**What should the RAM block type be?** – який тип вбудованої пам'яті потрібно використати? Цей параметр залежить від обраного сімейства мікросхем і може приймати такі значення:

- **Auto** – вибір типу для розміщення пам'яті відбувається за допомогою компілятора;
- **MLAB** – блок пам'яті розміром 640 біт;
- **M9K** – блок пам'яті розміром 9 кбіт;
- **M144K** – блок пам'яті розміром 144кбіта;
- **LCs** – реалізація блоку пам'яті за допомогою логічних елементів.

Друге вікно майстра проекту використовується для визначення, які з портів пам'яті будуть синхронними (рисунок 5.24).

**Which ports should be registered?** – які порти будуть тактуватися?

**'data' input port** – вхідні дані

**'address' input port** - адреса

**'q' output port** – вихідні дані

Далі необхідно визначити властивості тактових сигналів, які будуть використовуватись для тактування вхідних та вихідних сигналів.

Наступна вкладка містить діалог вибору початкового вмісту пам'яті. Тобто ОЗП може одразу після запуску містити якусь інформацію, яку потім можна видалити (рисунок 5.25).

**Do you want to specify the initial content of the memory? –** Чи бажаєте Ви визначити початковий вміст пам'яті.

**No, leave it blank –** Ні, залишити пустим.

**Yes, use this file for the memory content data –** Так використати файл для завантаження пам'яті.

У полі **File** name потрібно вибрати створений ***.mif** або ***.hex** файл. При цьому необхідно звернути увагу на те, щоб розрядність пам'яті та розмір даних в файлі співпадали.

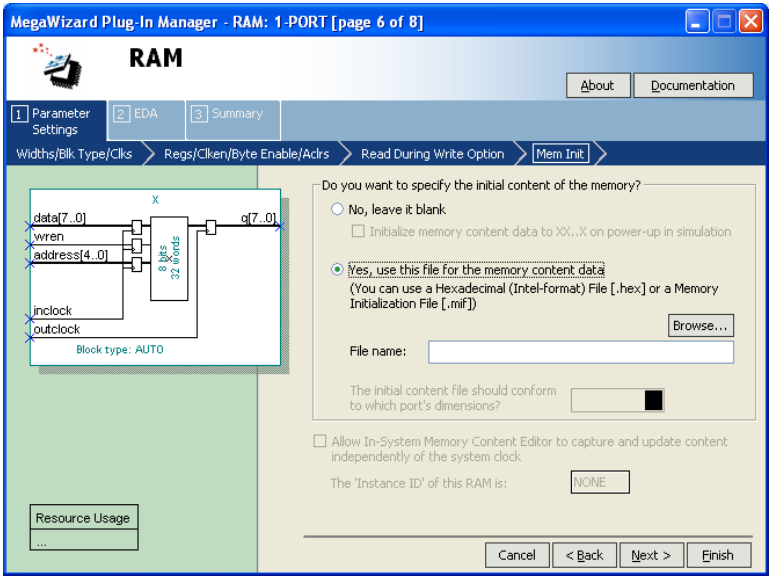



*Рисунок 5.25 – Визначення початкового вмісту пам'яті*

# 5.4. Робота з цифровими автоматами

Цифрові автомати зазвичай описуються на мові опису апаратури. Для створення цифрового автомату можна використати декілька способів:

- написати код цифрового автомату самостійно на мові опису апаратури (опис цифрового автомата на мові VHDL наведений у параграфі 3.3);
- нарисувати граф цифрового автомату в редакторі;
- побудувати граф цифрового автомату за допомогою помічника **State Machine Wizard**.

При використанні графу цифрового автомату необхідно попередньо побудувати таблиці переходів та станів та таблицю виходів. Після цього з графу цифрового автомату може бути згенерований текст опису автомату на мові опису апаратури [5-8].

Розглянемо рисування графу в редакторі. Для створення нового файлу цифрових автоматів вибираємо пункт меню **File → New**, в якому обираємо пункт **State Machine File**.

## 5.4.1. Робота з State Machine Wizard.

Для запуску **State Machine Wizard** необхідно відкрити редактор цифрових автоматів (**State Machine Viewer**) і вибрати пункт меню **Tools → State Machine Wizard** або натиснути кнопку  . Запуск цього майстра призводить до появи вікна, показаного на рисунку 5.26.

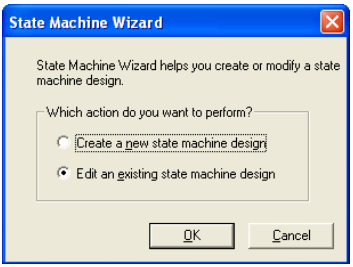



*Рисунок5.26 – Запуск **State Machine Wizard***

Тут розробнику необхідно вибрати один з двох варіантів: створення нового цифрового автомата (**Create a new state machine design**) або редагування вже існуючого (**Edit an existing state machine design**).

Перше вікно **State Machine Wizard** (рисунок 5.27) дозволяє визначити синхронізацію сигналу скидання (**Which reset mode do you want to use?**), його активний рівень (**Reset is active high**) та додавання регістрів у вихідні порти (**Register the output ports**).

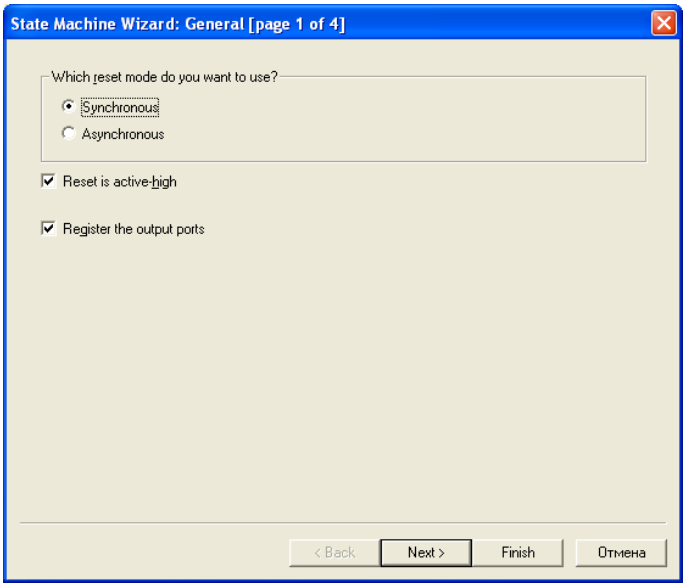



*Рисунок 5.27 – Перше вікно **State Machine Wizard***

У другому вікні (рисунок 5.28) перераховуються стани цифрового автомата (таблиця **States**), вхідні порти (таблиця **Input ports**) та наводиться таблиця переходів (**State transitions**). Опція **Transition to source state if not all transition conditions are specified** вказує, що необхідно переходити у вихідний стан, якщо інші переходи не визначені.

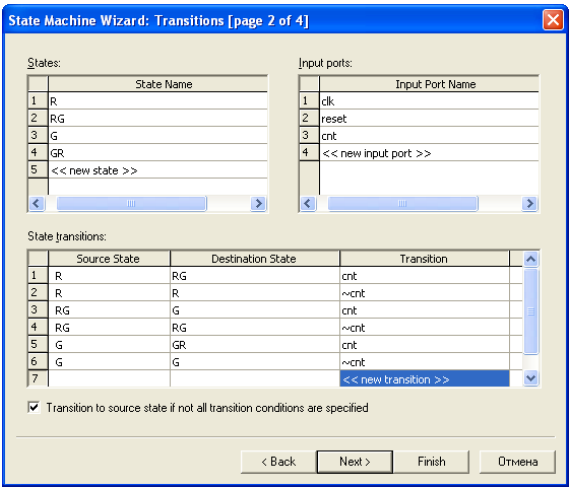



*Рисунок 5.28 – Друге вікно* ***State Machine Wizard***

У третьому вікні (рисунок 5.29) **State Machine Wizard** визначаються назви вихідних портів та таблиця виходів цифрового автомата.

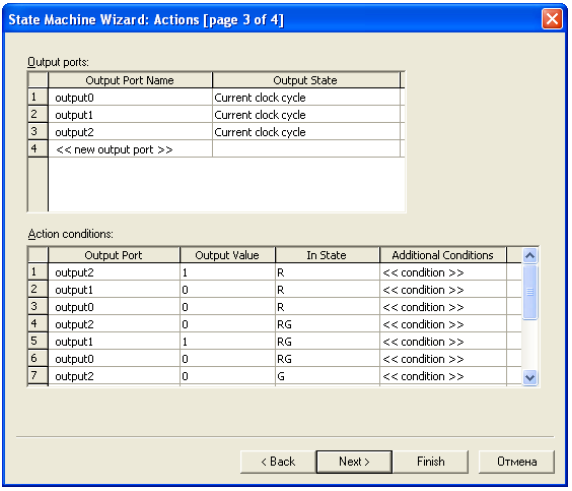



*Рисунок 5.29 – Третє вікно* ***State Machine Wizard***

В останньому вікні **State Machine Wizard** представлена сумарна інформація про розроблений цифровий автомат.

В результаті роботи **State Machine Wizard** буде отриманий цифровий автомат, який показаний на рисунку 5.30. Цей автомат може бути збережений у окремий файл типу *.smf.

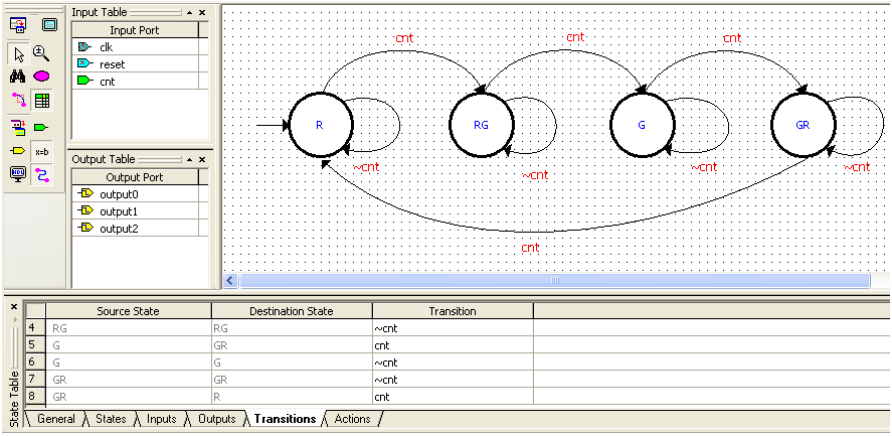



*Рисунок 5.30 – Результуючий цифровий автомат у редакторі цифрових автоматів*

Редактор цифрових автоматів є графічним пакетом, який дуже подібний до редактора для побудови графічних файлів. Вигляд цього редактора показаний на рисунку 5.30. Зліва від поля редактора розміщуються таблиці входів та виходів, а під полем редактора розміщуються таблиці, перелік яких наведений в таблиці 5.2.

З отриманого цифрового автомату може бути згенерований код на мові опису апаратури. Для цього необхідно вибрати пункт меню **Tools → Generate HDL File** або натиснути кнопку  на панелі інструментів редактора цифрових автоматів.

Призначення інструментів редактора цифрових автоматів наведене в таблиці 5.3.

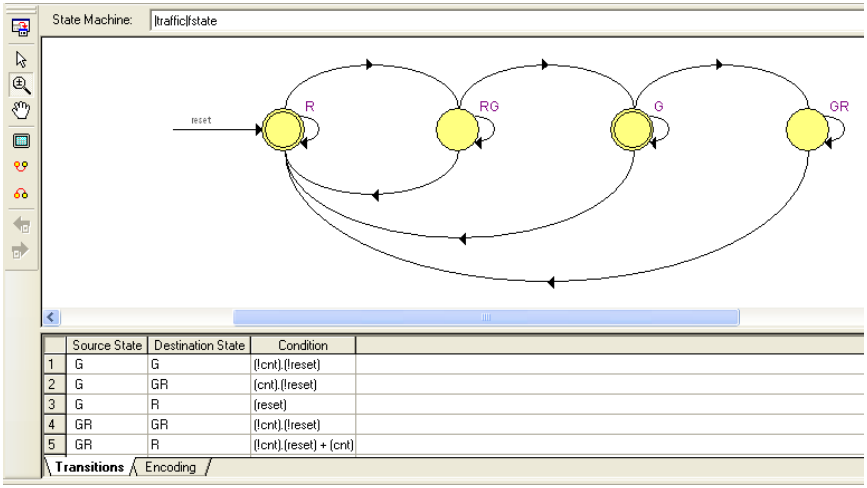
*Таблиця 5.2 – Вкладки вікна редагування цифрового автомата*

| Назва закладки | Призначення таблиці |
|---|---|
| General | Вказується тип сигналу скидання (reset) – синхронний або асинхронний та його активний рівень – високий або низький. |
| States | Перераховуються стани ЦА |
| Inputs | Перераховуються вхідні порти та їх сигнали керування |
| Outputs | Перераховуються вихідні порти |
| Transitions | Перераховуються попередні стани та результуючі стани, а також умови переходів |
| Actions | Перераховуються вихідні порти, їх значення та стани ЦА, які відповідають цим вихідним значенням. Також тут можуть перераховуватись додаткові умови. |

*Таблиця 5.3 – Інструменти редактора цифрових автоматів*

| Іконка інструменту | Назва інструменту | Призначення інструменту |
|---|---|---|
|  | State Tool | Інструмент для рисування станів ЦА |
|  | Transition Tool | Інструмент для рисування переходів ЦА |
|  | Input Port Tool | Додавання вхідних портів |
|  | Output Port Tool | Додавання вихідних портів |
|  | Rubberbanding Tool | Перемикання гнучкості ліній зв’язку |

В результаті компіляції згенерованого HDL коду буде отриманий цифровий автомат, граф якого можна побачити у вікні **State Machine Viewer**. Як видно з рисунку 5.31, згенерований цифровий автомат має ще й переходи у перший стан, які обумовлюються сигналом скидання.




| | Source State | Destination State | Condition |
|---|---|---|---|
| 1 | G | G | (!cnt).(!reset) |
| 2 | G | GR | (cnt).(!reset) |
| 3 | G | R | (reset) |
| 4 | GR | GR | (!cnt).(!reset) |
| 5 | GR | R | (!cnt).(reset) + (cnt) |

Transitions | Encoding

*Рисунок 5.31 – Цифровий автомат у **State Machine Viewer***

## 5.4.2. Кодування станів цифрового автомата

При кодуванні станів цифрового автомата можливе використання кількох методів, які доступні при призначенні опцій компіляції:

- **auto** – автоматичний вибір методу кодування в залежності від кількості станів цифрового автомата. Якщо станів менше ніж п’ять, то використовується метод кодування sequential. Якщо кількість станів більша п’яти але менша 50, то використовується метод one-hot. В усіх інших випадках використовується метод з використанням коду Грея (gray).
- **one-hot** – цей метод потребує N біт для кодування N станів цифрового автомата. У будь-якому стані лише один з бітів

має значення 1. Всі інші – 0. Цей стиль використовується за замовчуванням.

- **gray** – кодування станів за допомогою коду Грея. В цьому випадку сусідні коди відрізняється лише одним бітом і N бітний код Грея може кодувати 2N станів цифрового автомата.
- **Johnson** – кодування за допомогою коду Джонсона. Цей код подібний до коду Грея, але потребує меншу кількість логіки для реалізації кодування.
- **minimal bits** – компілятор буде мінімізувати кількість станів цифрового автомата, а відповідно і кількість біт, що необхідна для їх кодування.
- **sequential** – використовується звичайний двійковий код для опису станів цифрового автомата.
- **user-encoded** – використовується кодування, що визначає розробник. В цьому випадку необхідно описати коди для опису станів цифрового автомата самостійно.

В таблиці 5.4 наведені коди для чисел від 0 до 7 в різних кодах

*Таблиця 5.4 – Кодування чисел в різних системах*

| Двійковий код | Код Грея | Код Джонсона | Код «one-hot» |
|---|---|---|---|
| 000 | 000 | 00000 | 0000 0001 |
| 001 | 001 | 00001 | 0000 0010 |
| 010 | 011 | 00011 | 0000 0100 |
| 011 | 010 | 00111 | 0000 1000 |
| 100 | 110 | 01111 | 0001 0000 |
| 101 | 111 | 11111 | 0010 0000 |
| 110 | 101 | 11110 | 0100 0000 |
| 111 | 100 | 11100 | 1000 0000 |

Стиль кодування визначається глобально для всього проекту пунктом **State Machine Processing**, що вибирається при натисканні на кнопку **More Settings** сторінки **Analysis & Synthesis Settings** діалогу **Settings** або локально для окремого цифрового автомата у **Assignment Editor**.

При використанні мови VHDL для визначення стилю кодування використовується атрибут синтезу `enum_encoding`. Порядок використання атрибуту наступний:

1. Визначити атрибут `enum_encoding` типу рядок (**`string`**).
2. Зв'язати визначений атрибут з перелічимим типом, що використовується для станів цифрового автомата.
3. Значення атрибута повинно бути рядком, який може кодувати стани цифрового автомата або приймати значення `"default"`, `"sequential"`, `"gray"`, `"johnson"`, `"one-hot"`.

Наведемо приклад кодування станів цифрового автомата, що описує світлофор.

```
type color is (Red, RedGreen, Green, GreenRed);
attribute enum_encoding: string;
attribute enum_encoding of color: type is "gray";
```

Якщо використовується метод **user encoding**, то необхідно визначити перелічимий тип, а потім присвоїти значення типу std_ulogic кожному об'єкту перелічимого типу. В прикладі нижче визначається тип color, який має значення Red, RedGreen, Green, GreenRed.

```
type color is (Red, RedGreen, Green, GreenRed);
attribute enum_encoding: string;
attribute enum_encoding of color: type is "11 01 10 00";
```

В результаті компіляції значення типу color будуть закодовані наступним чином:

```
Red = "11"
RedGreen = "01"
Green = "10"
GreenRed = "00"
```

# 5.5. Оптимізація проектів в пакеті Quartus II

Оптимізація проекту має за мету покращення його характеристик: швидкодії, зменшення об'єму, що займає проект, споживаної потужності. Це необхідно у випадку, коли потрібно зменшити собівартість проекту, спробувати вмістити проект у меншу мікросхему та підвищити його тактову частоту [2, 3, 7].

Спочатку ми опишемо ті дії, які необхідно виконати **до оптимізації проекту**, оскільки вони в подальшому дозволять безболісно вносити зміни до проекту. Найперше, необхідно при розробці проекту правильно розділити його на частини та визначити вимоги до проекту. А після того як проект вже зроблено необхідно проаналізувати всі попередження, які виводить пакет Quartus II, а також проаналізувати систему тактування проекту.

## 5.5.1. Розподіл проекту на частини.

Під розподілом проекту мається на увазі не incremental compilation, а розбивка проекту на функціональні блоки.

Така розбивка дуже важлива, оскільки :

- дозволяє використати певні методи оптимізації для різних частин проекту;
- потім застосувати incremental compilation;
- розташувати на кристалі функціональні блоки після компіляції;
- спрощується наступна зміна схеми проекту.

Розглянемо більш докладно деякі **правила розподілу проекту на частини**.

1. Необхідно попередньо обміркувати, які ресурси необхідні даному проекту. При цьому необхідно завжди пам'ятати, що яким би способом не робився опис проекту (схема, мови опису апаратури) у результаті компіляції буде отримана схема, що повинна бути реалізована в заданому елементному базисі, тобто в ПЛІС обраного сімейства. А це означає, що необхідно визначити передбачуваний обсяг вбудованої пам'яті ПЛІС, необхідність блоків ФАПЧ,

помножувачів або блоків ЦОС, а також вимоги до інтерфейсів портів вводу-виводу і їх кількості. Після проведення подібного аналізу з'являється більш ясне бачення вимог до мікросхеми ПЛІС.

2. Розбивку найбільш зручно робити відповідно до функціонального призначення вузлів пристрою. У результаті кожен окремий функціональний вузол буде описуватися окремою частиною проекту. При подібній розбивці зручним може виявитися використання графічного подання верхнього рівня проекту.

3. Необхідно мінімізувати кількість зв'язків між блоками верхнього рівня. Це дозволить потім більш просто реалізувати колективу розробку проекту.

4. Необхідно мінімізувати кількість обмежень (**construct**) до кожного конкретного блоку. Використання обмежень для окремого блоку дозволяє використати індивідуальну техніку оптимізації для кожного блоку. Однак, призначення індивідуальних вимог до блоку потрібно використовувати дуже обачно, щоб не збільшувати час компіляції проекту. Про призначення індивідуальних вимог до блоку параграф 5.5.4.

5. Мінімізувати кількість тактових доменів для кожного блоку. Це дозволить оптимально використати ланцюги тактування ПЛІС. При цьому навіть невеликий блок може використати свою тактову частоту.

6. Розташувати цифрові автомати в окремих блоках. Це підсилить контроль за правильністю кодування станів цифрового автомата, а також робити оптимізацію даного блоку по швидкості.

7. Звести всі критичні по швидкодії ланцюги в один блок. Це полегшить розміщення таких ланцюгів на кристалі, проведення оптимізації файлу зв'язків (**physical synthesis optimization**).

8. Використати регістри для всіх входів і виходів функціональних блоків проекту. Таке рішення дозволяє зменшити залежність блоків один від одного в часовому сенсі, тому що призведе до синхронного запису та зчитування даних. Цю вимогу необхідно застосовувати для більш високих рівнів ієрархії проекту.

9. Не використовувати функції із третім станом та двонаправлені виводи всередині проекту, оскільки вони фізично не існують у ПЛІС. Виключення становлять тільки виводи ПЛІС, що виходять назовні.

**Вимоги до параметрів проекту.**

1. Необхідно визначити всі вимоги до часових параметрів проекту. Це дозволяє модулям пакету (**fitter**, **timing analyzer**) більш чітко працювати з проектом. У цьому випадку компонувальник (**fitter**) робить оптимізацію критичних по швидкодії ланцюгів, а повідомлення системи дозволяють визначити виконання або невиконання даних вимог.

2. Перевірка ланцюгів тактування. Необхідно уникати появи комбінаційних функцій у ланцюгах тактування (т.зв. **gated clock**). Зазвичай при тактові сигнали проходять по мікросхемі за допомогою ліній глобального розповсюдження сигналів. Це дозволяє зменшити затримки тактових сигналів. У тому ж випадку, коли в ланцюзі тактування використовується комбінаційна схема, яка керує появою тактового сигналу, то тактовий сигнал може бути трактований компілятором як звичайний сигнал. Це може призвести до того, що він буде розповсюджуватись за допомого матриць з'єднань, а це в свою чергу може привести до появи гонок сигналів. Також необхідно звернути увагу на ланцюги, які працюють по передньому та задньому фронтам тактового сигналу, оскільки ланцюги з таким тактуванням фактично працюють із подвоєною частотою.

3. Необхідно перевірити те, яким чином відбувається перетинання сигналами границь тактових доменів.

## 5.5.2. Аналіз повідомлень компілятора.

1. Аналіз повідомлень компілятора необхідний для того щоб визначити, що саме робить компілятор. Також необхідно пам'ятати, що не всі повідомлення компілятора показують проблеми в проекті. Тому необхідно відібрати повідомлення за допомогою фільтра.

Розглянемо процес відбору повідомлень. Така фільтрація дозволить сховати повідомлення, як при поточній компіляції, так і при наступних. Відфільтровані повідомлення будуть виводитись в окремій закладці, а самі правила фільтрації зберігаються у файлі <ім'я проекту>.srf.

Для фільтрації повідомлень необхідно вибрати потрібне повідомлення та у контекстному меню вибрати пункт **Suppress** Це контекстне меню містить наступні пункти:

**Suppress Exact Selected Messages** – відбір саме такого повідомлення;

**Suppress All Similar Messages** – відбір схожих повідомлень;

**Message Suppression Manager** – виклик менеджера фільтрації повідомлень;

**Import Message Suppression Rule File, Export Message Suppression Rule File** – імпорт та експорт фільтрів повідомлень в/з файлів.

Відібрані повідомлення виводяться в окремій закладці, що носить назву **Suppressed** (рисунок 5.32).

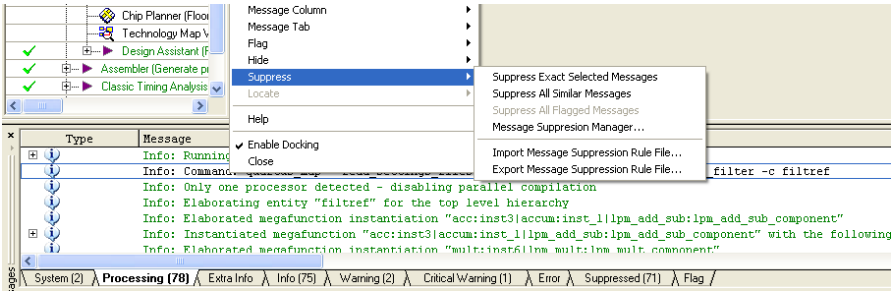



*Рисунок 5.32 – Фільтрація повідомлень компілятора*

**Менеджер фільтрації** повідомлень являє собою діалог вибору повідомлень для фільтрації та вікно роботи з фільтрами. Вигляд цього діалогу показаний на рисунку 5.33.

При аналізі повідомлень компілятора не слід забувати про вбудовану допомогу пакету Quartus II. Доступ до допомоги з кожного конкретного повідомлення можливий за допомогою контекстного меню, в якому необхідно вибрати пункт **Help**. Допомога побудована на принципі – опис проблеми (**CAUSE**) – шляхи вирішення (**ACTION**).

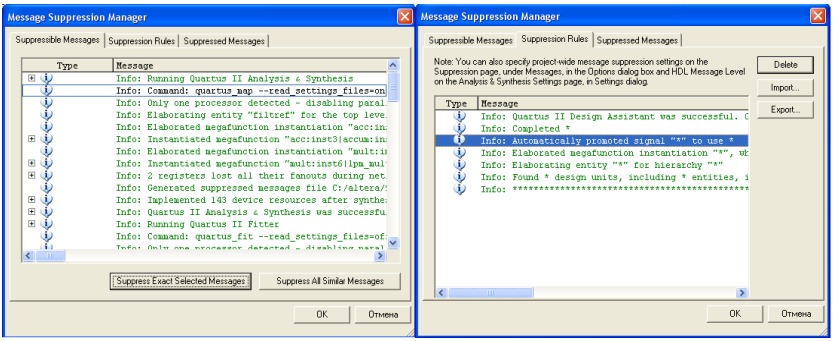



*Рисунок 5.33 – Менеджер фільтрації повідомлень*

## 5.5.3. Помічники оптимізації

У меню **Tools** знаходиться пункт **Advisors**, що містить кілька модулів, призначених для допомоги проектувальнику при роботі з проектом (рисунок 5.34). Ці модулі дають рекомендації з оптимізації різних аспектів проекту.

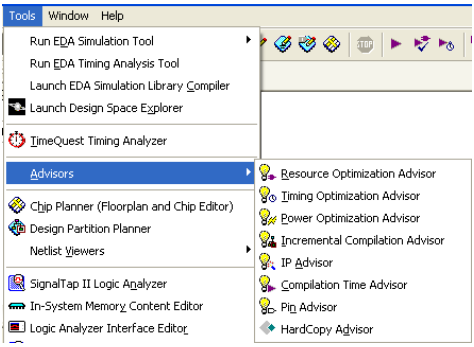



*Рисунок 5.34 – Помічники оптимізації проекту*

У даному пункті меню доступні наступні помічники:

**Resource Optimization Advisor** – помічник оптимізації використаних ресурсів,

**Timing Optimization Advisor** – помічник оптимізації часових параметрів,

**Power Optimization Advisor** – помічник оптимізації споживаної потужності,

**Incremental Compilation Advisor** – помічник оптимізації при компіляції проекту по частинах,

**IP Advisor** – помічник оптимізації при використанні IP ядер,

**Compilation Time Advisor** – помічник оптимізації часу компіляції,

**Pin Advisor** – помічник оптимізації виводів мікросхеми,

**HardCopy Advisor** – помічник оптимізації при використанні мікросхем типу HardCopy.

Всі зазначені вище види помічників можуть бути згруповані в три типи: оптимізація часових параметрів, оптимізація ресурсів, оптимізація споживаної потужності.

Розглянемо як приклад роботу помічника оптимізації часових параметрів (рисунок 5.36).

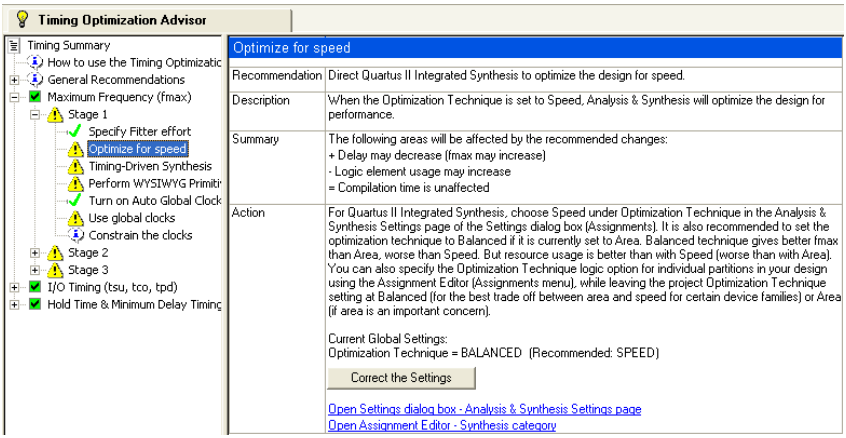



*Рисунок 5.36 – Помічник оптимізації часових параметрів проекту*

Ліва частина вікна містить список різних проблем, що виникають при часовому аналізі: загальні рекомендації (**General recommendations**), максимальна тактова частота (**Maximum frequency ($f_{MAX}$)**), часові параметри елементів вводу-виводу (**I/O timing ($t_{SU}$, $t_{CO}$, $t_{PD}$)**), час утримання та мінімальна затримка (**Hold time and minimum delay timing**).

Права частина вікна містить таблицю, що складається з наступних розділів:

**Description** – опис причини рекомендації, інформацію про установки та призначення, які можуть використовуватися. Також даний розділ може містити посилання на інші розділи довідки або частини книги Quartus II Handbook.

**Summary** – описується ефект, що може принести дана рекомендація у проекті. Ефект від рекомендації позначається наступними символами:

( + ) – рекомендація має позитивний ефект для проекту;

( - ) – рекомендація має негативний ефект для проекту;

( = ) – рекомендація може не дати ефекту для проекту.

**Action** – описуються дії, які необхідно зробити для виконання даної рекомендації, а також поточний стан параметра, що рекомендується змінювати. Додатково розділ містить посилання на відповідний розділ діалогу **Settings** або категорію в редакторі призначень **Assignment Editor**.

Список іконок помічника, які використовуються для позначення відносини даної рекомендації до поточних

Кожен помічник відображає результати компіляції у зведеному виді. Для цього необхідно звернутися до розділу **Summary**. При використанні помічників необхідно пам'ятати, що поради по оптимізації є загальними для всіх типів проектів і не враховують особливостей конкретного проекту. Тому не завжди результати використання порад помічника можуть давати кращий результатустановок проекту. Список цих іконок наведений у таблиці 5.5.

*Таблиця 5.5 – Іконки статусу порад у помічнику*

| Іконка | Опис |
|---|---|
|  | Вказує на проблему в даній категорії порад. |
|  | Вказує на опції, настроювання, установки проекту які не відповідають рекомендованим. |
|  | Показує, що призначення відповідають значенням за замовчуванням, але ефект від їхнього використання мінімальний. |
|  | Показує, що призначення відповідають значенням за замовчуванням. |
|  | Показує, що помічник оптимізації не може визначити ефект від даних призначень у проекті. |

## 5.5.4. Аналіз проекту

Після проведення компіляції пакет Quartus II видає звіт, аналіз якого дозволить виявити шляхи оптимізації проекту. Можна виділити наступні етапи аналізу:

1. Аналіз ресурсів, що використовуються в проекті.
2. Аналіз затримок сигналів.
3. Аналіз максимальної тактової частоти.

**Аналіз ресурсів** необхідно використовувати у тому випадку, коли проект не вміщується у використовувану мікросхему. Для того, щоб визначити обсяг використовуваних ресурсів необхідно скористатися звітом, генерованим компілятором. В пакеті Quartus II звіт можна побачити відкривши його **Processing → Compilation Report → Fitter → Resource Section → Resource Usage Summary**.

В цьому звіті можна побачити, яким чином використовуються ресурси мікросхеми. Ресурси групуються за типами і для більш докладної інформації з кожного типу ресурсів можна переглянути окремі розділи звіту R**esource Section** (рисунок 5.37).

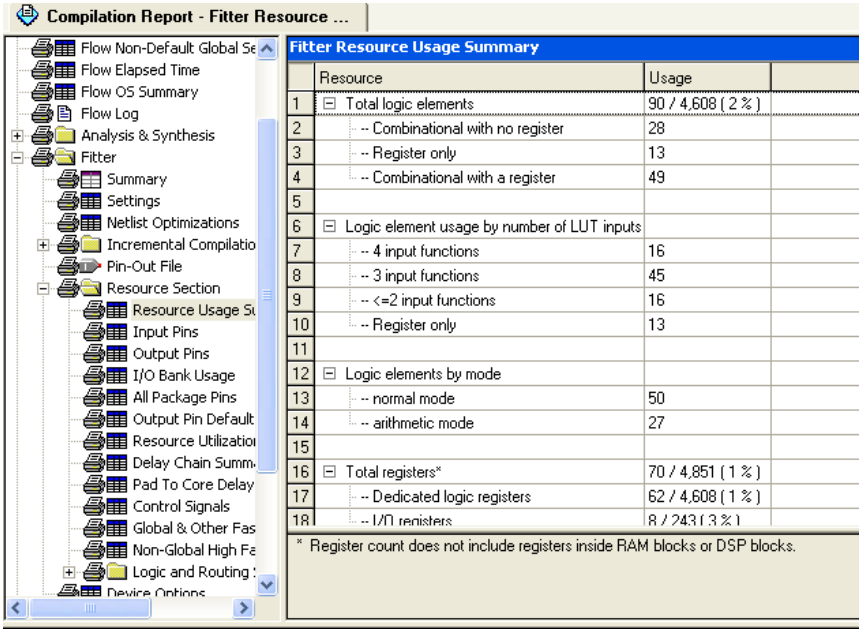



*Рисунок 5.37 – Звіт про використання ресурсів мікросхеми*

**Аналіз затримок сигналів.**

Для того, щоб переглянути звіт зі значеннями затримок сигналів необхідно відкрити закладку **Timing Analyzer**: **Processing → Compilation Report → Timing Analyzer** (рисунок 5.38).

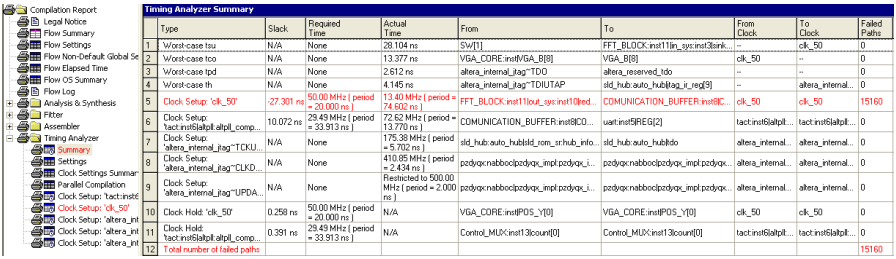



*Рисунок 5.38 – Звіт часового аналізатора*

Сумарний звіт часового аналізатора впорядкований спочатку за типом часового параметру (**Type**), потім за відхиленням розрахованого значення від необхідного (**Slack**) і в останню чергу – за значенням цього відхилення.

В полі типів часових параметрів можливі такі значення:

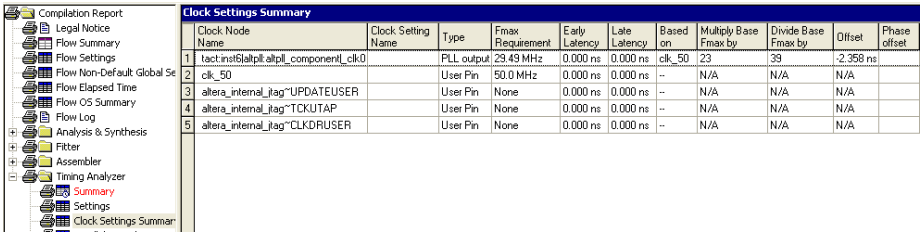
- найгірший випадок для часу передустановлення сигналу (**Worst-case $t_{su}$**);
- найгірший випадок для часу утримання сигналу (**Worst-case $t_h$**);
- найгірший випадок для часу, що проходить від надходження тактового сигналу до моменту, коли з'явиться сигнал на виході (**Worst-case $t_{co}$**);
- найгірший випадок для часу, що проходить від надходження асинхронного сигналу до моменту, коли сигнал з'явиться на виході (**Worst-case $t_{pd}$**);
- час передустановлення для тактового сигналу (**Clock Setup**). Якщо тактових сигналів у проекті декілька, то виводиться час передустановлення для кожного тактового сигналу окремо;
- час утримання для кожного тактового сигналу (**Clock Hold**);
- загальна кількість шляхів сигналів, на яких не виконані вимоги по часовим параметрам (**Total number of failed path**).

Значення відхилення (**Slack**) відображається чорним, якщо воно входить у відведений діапазон, або червоним, коли перевищує його.

**Аналіз максимальної тактової частоти.**

Для того, щоб переглянути звіт з тактових частот проекту необхідно відкрити закладку **Clock Setup**: **Processing → Compilation Report → Clock Setup** (рисунок 5.39). Таблиця тактових частот містить список тактових частот проекту (**Clock Node Name**), їх джерела (**Type**), параметри тактових сигналів: максимальну тактову частоту (**Fmax Requirement**) та вимоги до кожного з сигналів. В залежності від джерела тактового сигналу можуть відображатися додаткові параметри. Так, наприклад, на рисунку 5.39 показаний сигнал, що генерується блоком ФАПЧ (PLL output), для якого показані джерело сигналу (clk_50), коефіцієнти множення (**Multiply Base Fmax**

**by**) та ділення (**Divide Base Fmax by**) для блоку ФАПЧ, зсув сигналу відносно основного тактового сигналу (**Offset**).




**Clock Settings Summary**

| | Clock Node Name | Clock Setting Name | Type | Fmax Requirement | Early Latency | Late Latency | Based on | Multiply Base Fmax by | Divide Base Fmax by | Offset | Phase offset |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | tact:inst6\|altpll:altpll_component\|_clk0 | | PLL output | 29.49 MHz | 0.000 ns | 0.000 ns | clk_50 | 23 | 39 | -2.358 ns | |
| 2 | clk_50 | | User Pin | 50.0 MHz | 0.000 ns | 0.000 ns | -- | N/A | N/A | N/A | |
| 3 | altera_internal_jtag~UPDATEUSER | | User Pin | None | 0.000 ns | 0.000 ns | -- | N/A | N/A | N/A | |
| 4 | altera_internal_jtag~TCKUTAP | | User Pin | None | 0.000 ns | 0.000 ns | -- | N/A | N/A | N/A | |
| 5 | altera_internal_jtag~CLKDRUSER | | User Pin | None | 0.000 ns | 0.000 ns | -- | N/A | N/A | N/A | |

*Рисунок 5.39 – Аналіз тактових частот*

### 5.5.5. Оптимізація використання ресурсів мікросхеми.

Оптимізацію використання ресурсів мікросхеми проводять декількома шляхами:

- оптимізація коду на мові опису апаратури;
- оптимізація використання блоків пам’яті;
- оптимізація використання блоків ЦОС.

Також для проведення будь-якої оптимізації необхідно визначити глобальну та локальну техніку оптимізації.

**Оптимізація коду на мові опису апаратури.**

При написанні коду на мові опису апаратури необхідно бути впевненим, що пам’ять, вузли цифрової обробки сигналу будуть реалізовані на вбудованих блоках пам’яті та ЦОС, а не на логічних елементах. Для цього компілятор пакету Quartus II повинен розпізнати в коді, що описує вищеназвані блоки саме пам’ять та блок ЦОС. Для того, щоб бути впевненим в цьому необхідно використати **Chip Planner** для перегляду розташування частин проекту на кристалі. Використання **Chip Planner** описане в параграфі 4.3.

Також необхідно подивитись результати компіляції та оптимізації цифрових автоматів. Для цього потрібно використати пункт **Processing → Compilation Report → Analysis & Synthesis →**

**State Machines**. Більш детально про опис цифрових автоматів на мові VHDL див. параграф 3.3.

Також необхідно за можливістю використовувати шаблони пакету Quartus II (див. параграф 3.6).

Для оптимізації ресурсів мікросхеми необхідно визначити **глобальну техніку оптимізації**. Попередньо необхідно впевнитись, що всі призначення, які встановлені для проекту коректні та актуальні. Це призначення мікросхеми, часових параметрів. Для визначення глобальної техніки оптимізації необхідно вибрати закладку **Analysis and Synthesis Settings** діалогу **Settings** (рисунок 5.40)

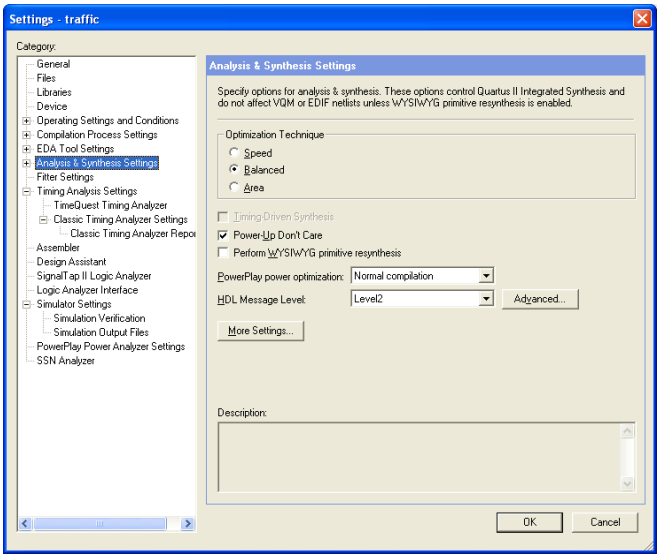



*Рисунок 5.40 – Параметри глобальної оптимізації*

Техніка глобальної оптимізації (**Optimization Technique**) обирається з трьох варіантів:

- **Speed** – оптимізація за швидкодією.
- **Area** – оптимізація за площею, що займає проект на кристалі.

– **Balanced** – середнє значення між швидкодією та площею – значення за замовчуванням.

Глобальна техніка оптимізації стосується всього проекту в цілому. Але для окремих блоків можуть бути встановлені свої опції синтезу. В такому випадку при компіляції опції оптимізації локального об'єкту відміняють глобальні опції оптимізації. Для визначення локальних параметрів оптимізації для окремого блоку необхідно виділити необхідний блок і в контекстному меню обрати пункт **Locate → Locate in Assignment Editor** (рисунок 5.41).

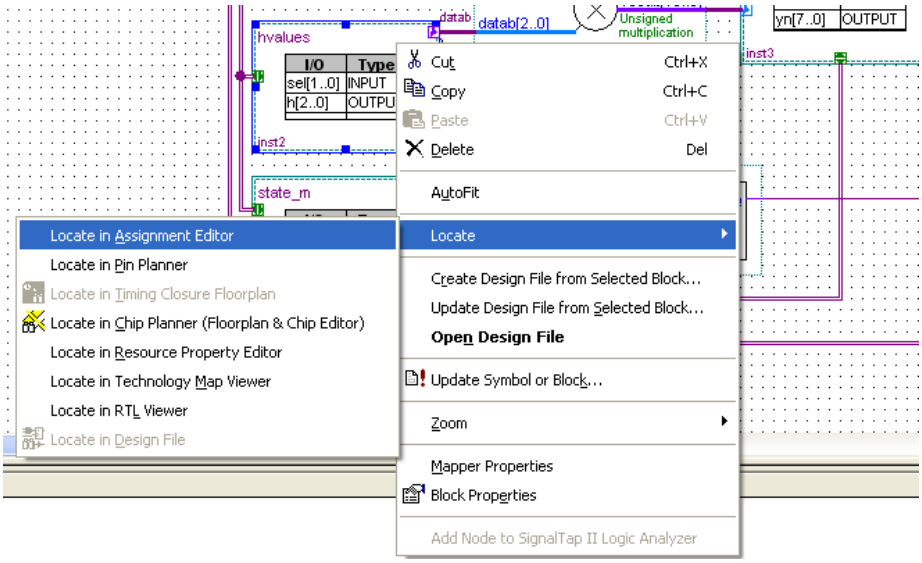



*Рисунок 5.41 – Запуск **Assignment Editor***

Після цього відкриється редактор, який дозволяє виконувати різні призначення параметрів проекту (рисунок 5.42). В полі **Assignment Name** оберемо параметр **Optimization Technique** – техніка оптимізації. Значення цього параметра, яке встановлюється в поле **Value** приймає ти ж самі значення, що й при глобальній оптимізації: **Speed**, **Balanced**, **Area**.

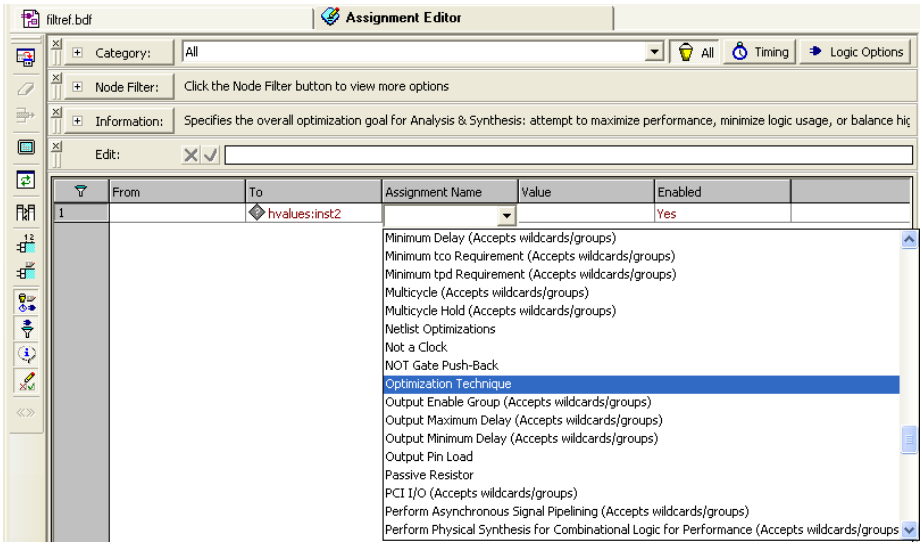
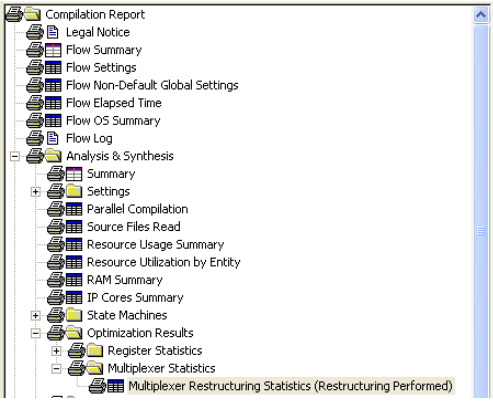



*Рисунок 5.42 – Редактор призначень* ***Assignment Editor***

Велику частину будь-якого проекту займають мультиплексори, які утворюються при використанні системних шин в мікропроцесорних системах, реалізації операцій **`if`** … **`then`**, **`case`**, цифрових автоматів. Для оптимізації таких схемотехнічних конструкцій необхідно використовувати опцію компілятора **Restructure Multiplexers**, яка доступна або в **Assignment Editor** або в списку, що відкривається при натисканні кнопки More Settings на сторінці **Analysis & Synthesis Settings** діалогу **Settings**. Результат використання цієї опції можна побачити на сторінці **Multiplexer Restructuring Statistics** звіту компіляції (**Analysis & Synthesis → Optimization Result**) і він може давати до 20% зменшення розміру схеми мультиплексорів (рисунок 5.43) - поле **Saving if Restructed**.




Multiplexer Restructuring Statistics (Restructuring Perfor

| | Multiplexer Inputs | Bus Width | Baseline Area | Area if Restructured | Saving if Restruct... |
|---|---|---|---|---|---|
| 1 | 5:1 | 32 bits | 96 LEs | 32 LEs | 64 LEs |
| 2 | 6:1 | 16 bits | 64 LEs | 32 LEs | 32 LEs |
| 3 | 9:1 | 15 bits | 90 LEs | 60 LEs | 30 LEs |
| 4 | 6:1 | 13 bits | 52 LEs | 26 LEs | 26 LEs |
| 5 | 9:1 | 14 bits | 84 LEs | 70 LEs | 14 LEs |
| 6 | 20:1 | 4 bits | 52 LEs | 40 LEs | 12 LEs |
| 7 | 3:1 | 10 bits | 20 LEs | 10 LEs | 10 LEs |
| 8 | 4:1 | 9 bits | 18 LEs | 9 LEs | 9 LEs |
| 9 | 8:1 | 2 bits | 10 LEs | 4 LEs | 6 LEs |
| 10 | 6:1 | 2 bits | 8 LEs | 2 LEs | 6 LEs |
| 11 | 3:1 | 4 bits | 8 LEs | 4 LEs | 4 LEs |
| 12 | 6:1 | 2 bits | 8 LEs | 4 LEs | 4 LEs |
| 13 | 5:1 | 4 bits | 12 LEs | 8 LEs | 4 LEs |
| 14 | 9:1 | 2 bits | 12 LEs | 8 LEs | 4 LEs |
| 15 | 3:1 | 3 bits | 6 LEs | 3 LEs | 3 LEs |
| 16 | 3:1 | 3 bits | 6 LEs | 3 LEs | 3 LEs |
| 17 | 5:1 | 3 bits | 9 LEs | 6 LEs | 3 LEs |

*Рисунок 5.43 – Результат використання оптимізації мультиплексорів*

Дуже часто при розробці виникає ситуація, коли у логічному елементі використовується лише таблиця перекодування або лише тригер. В такому випадку можливе використання таблиці перекодування та тригера одного логічного елемента для різних операцій (рисунок 5.44). Така операція називається пакування регістрів (**Register Packing**). Використання цієї можливості приводить до зменшення використовуваних ресурсів і до зменшення робочої частоти. Для мікросхем сімейств Stratix, Stratix GX, Cyclone типове зменшення займаного обсягу до 90% від початкового, а тактової частоти до 94 % від початкової.

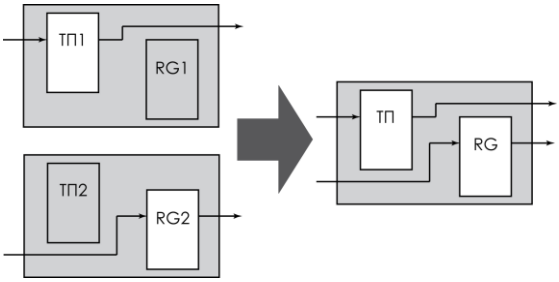



*Рисунок 5.44 – Пакування регістрів*

Для ввімкнення цієї опції необхідно вибрати **Auto Packed Registers** на сторінці **Fitter Settings** (рисунок 5.45).

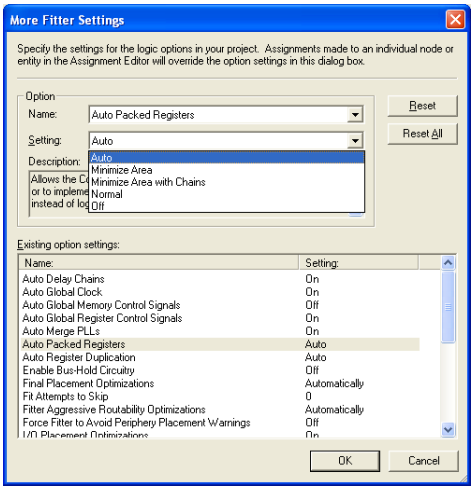



*Рисунок 5.45 – Визначення опцій пакування регістрів*

Для вибору доступні декілька варіантів оптимізації:

- **Off** – компілятор не розміщує таблицю перекодування та тригер в одному логічному елементі.
- **Nornal** – розміщення таблиці перекодування та тригера в одному логічному елементі тільки у випадку коли це не впливає на швидкодію проекту.
- **Minimize Area with Chains** – компілятор зводить таблицю перекодування та тригер для ланцюгів переносу в арифметичних операціях.
- **Minimize Area** – компілятор розміщує таблицю перекодування та тригер разом з метою зменшення обсягу проекту незважаючи на зменшення швидкодії.
- **Auto** – автоматичний вибір методу.

Як було вже вказано вище, можливе використання не тільки глобальних опцій компіляції, але й локальних. Приклад призначення локальних опцій пакування регістрів показаний на рисунку 5.46.

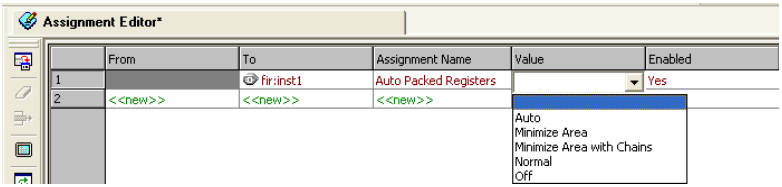



*Рисунок 5.46 – Визначення локальних опцій пакування регістрів*

## Оптимізація використання блоків пам’яті

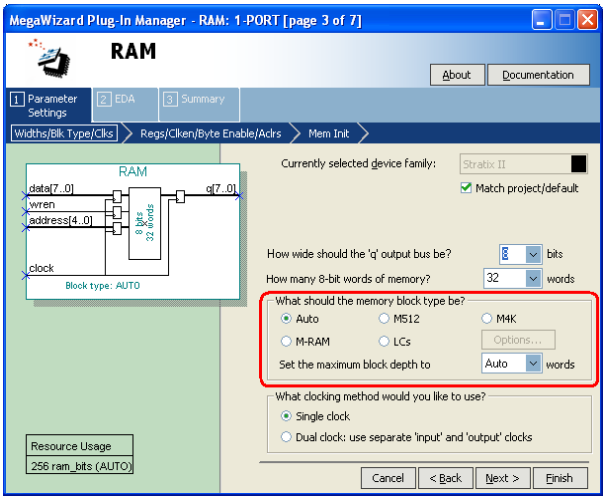



*Рисунок 5.47 – Визначення типу блоку пам’яті*

При використання блоків пам’яті слід контролювати відповідність об’єму модуля пам’яті який ми хочемо реалізувати і блоку вбудованої пам’яті. Сучасні мікросхеми ПЛІС мають в своєму складі декілька різних за розміром блоків пам’яті. Тому необхідно, щоб модуль пам’яті займав якомога більшу частину блоку вбудованої пам’яті. А для невеликих модулів пам’яті іноді доцільно використовувати логічні елементи. Всі ці процедури вимагають

відключення автоматичного розташування блоків пам'яті і визначення цих опцій в процесі проектування.

При розробці модуля пам'яті за допомогою **MegaWizard Plug-In Manager** необхідно визначати тип блоків, який буде використаний для даного модуля пам'яті (рисунок 5.47), а при використанні мови опису апаратури потрібно використовувати атрибут синтезу **`ramsyle`**.

Також необхідно відключити опції автоматичного розміщення пам'яті. Для всього проекту в цілому це опції **Auto RAM Replacement** та **Auto ROM Replacement**, які доступні при натисканні кнопки **More Settings** на сторінці **Analysis & Synthesis Settings** діалогу **Settings**. Для локальних опцій конкретних блоків необхідно використовувати **Assignment Editor**.

**Оптимізація використання блоків ЦОС.**

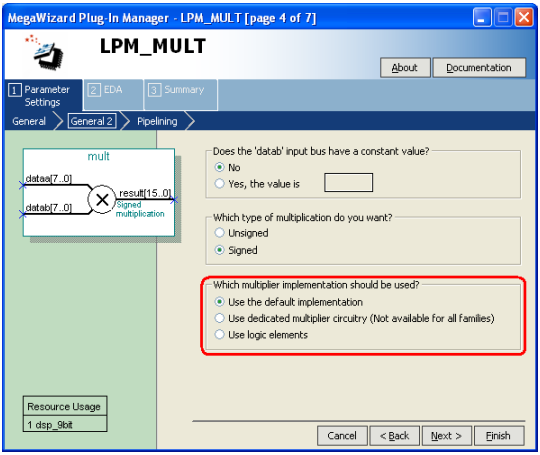



*Рисунок 5.48 – Визначення варіанту реалізації помножувача*

При використанні ПЛІС для виконання великої кількості розрахунків можлива ситуація, коли може не вистачати блоків цифрової обробки сигналів (DSP block). Невеликі за обсягом або не критичні до швидкодії блоки можливо розміщувати у логічних

елементах. Для цього необхідно використовувати або призначення в **MegaWizard Plug-In Manager** (рисунок 5.48) або опцію **DSP Block Balancing** (рисунок 5.49).

На рисунку 5.48 показане створення помножувача за допомогою **MegaWizard Plug-In Manager**. Видно, що при компіляції можна використовувати значення за замовчуванням (**Use the default implementation**), яке дозволяє враховувати розрядність помножувача для вибору ресурсів. На рисунку показано, що знаковий помножувач 8×8 використовує один блок ЦОС – пункт **Resource Usage**. В тому ж випадку, коли блоки ЦОС відсутні в мікросхемі ПЛІС то для реалізації будуть використані вбудовані помножувачі. Наступне значення опції – це примусове використання блоків ЦОС (**Use dedicated multiplier circuitry**). Ця опція може бути недоступною для деяких сімейств. Останній варіант – розміщення помножувача в логічних елементах (**Use logic elements**).

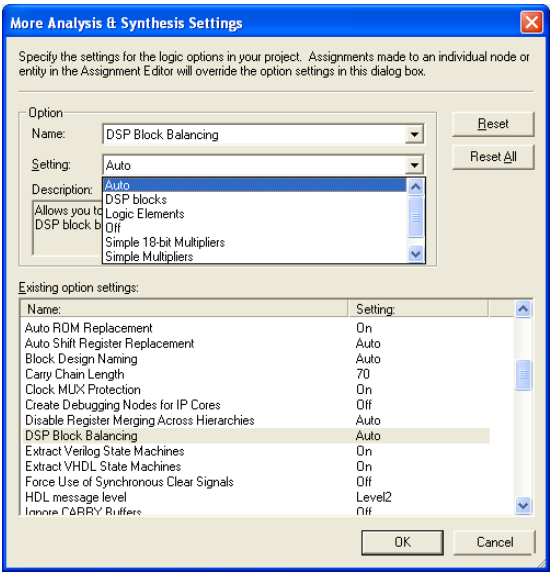



*Рисунок 5.49 – Визначення опцій розташування блоків ЦОС*

Для повного контролю над процесом розміщення ЦОС блоків у ПЛІС можна використовувати глобальні або локальні опції розміщення блоків ЦОС. Для цього необхідно обрати пункт **DSP Block Balancing**, який доступний при натисканні на кнопку **More Settings** на сторінці **Analysis & Synthesis Settings** діалогу **Settings**. Або обрати цю опцію в **Assignment Editor**.

## 5.6. Засоби внутрішньосистемного налагоджування для ПЛІС Altera

Налагоджування проекту на ПЛІС на платі в оточенні реальних пристроїв дозволяє виявити проблеми, пов'язані з друкованою платою, виявити неузгодженість інтерфейсних частин проекту та взагалі перевірити функціонування всього пристрою. Налагодження проекту на платі має декілька переваг перед симуляцією проекту за допомогою програмного забезпечення, серед яких слід виділити швидкість верифікації, оскільки програмна симуляція займає більше часу, а також програмна верифікація обмежена в достовірності моделі вхідного впливу у порівнянні з реальним сигналом. Найбільш часто при налагоджуванні електронних пристроїв використовують такі прибори як осцилограф, логічний аналізатор та різноманітні генератори сигналів [1, 4, 7].

Пакет Quartus II надає комплекс засобів для системного відлагодження проектів, які дозволяють як замінити зовнішні прилади виміру та аналізу, так і полегшити підключення цих зовнішніх приладів. До засобів системного налагодження пакету Quartus II (рис. 5.50) відносяться:

- редактор відлагоджувальних виводів (**SignalProbe Pins**);
- редактор інтерфейсу для зовнішнього логічного аналізатора (**Logic Analyzer Interface Editor**);
- редактор вмісту пам'яті в системному оточенні (**In-System Memory Content Editor**);

– вбудований логічний аналізатор **Signal Tap II** (**Signal Tap II Logic Analyzer**).

При використанні засобів системного відлагодження САПР Quartus II рекомендується виконувати інкрементальну компіляцію (**Incremental Compilation**). При цьому додавання засобів налагодження не буде вносити змін у розміщення та розведення проекту і його характеристики залишаться незмінними. Крім того, у випадку зміни настроювань засобів налагодження час повторної компіляції буде значно скорочено.

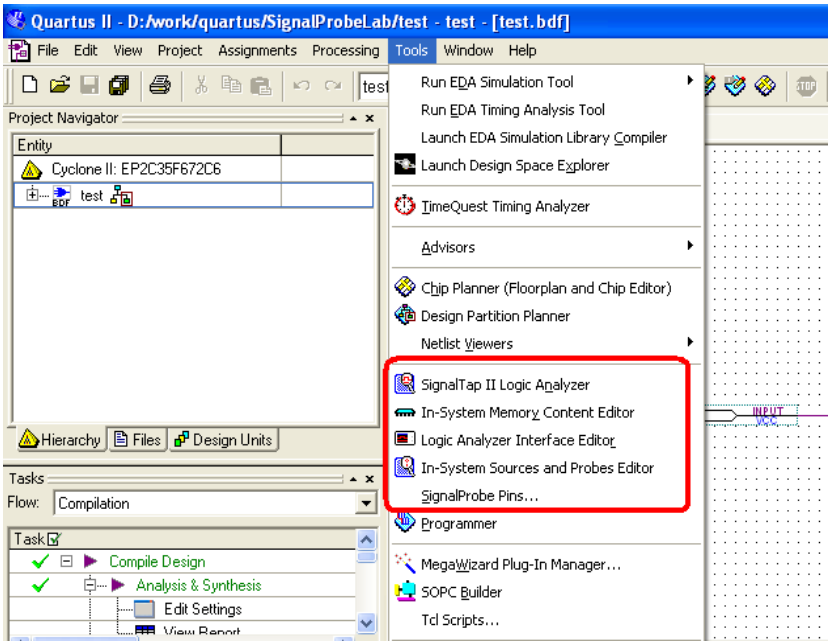



*Рисунок 5.50 – Засоби внітрішньосистемного відлагоджування в Quartus II*

Вищеперелічені засоби налагоджування відрізняються впливом на цільовий проект, кількістю необхідних ресурсів ПЛІС – логічних елементів та вбудованих блоків пам'яті.

## 5.6.1. Налагоджувальні виводи (SignalProbe)

Налагоджувальні виводи найбільш економним варіантом внутрішньосистемного налагоджування. Цей метод не потребує внутрішніх ресурсів ПЛІС і використовує лише вільні виводи мікросхеми для комутації внутрішніх сигналів проекту на виходи мікросхеми. Перш ніж розпочати знайомство з цим методом необхідно відмітити його відмінність від підключення внутрішніх сигналів проекту до виводів мікросхеми. При підключенні сигналу до виводу ПЛІС вузол, який підключається залишається в проекті і не може бути мінізований. Як вже описувалось раніше (див. параграф 4.3) при компіляції проекту на етапі аналізу та синтезу проекту (**Analysis & Synthesis**) виконується мінімізація логічних рівнянь. В результаті частина внутрішніх сигналів може бути відсутньою у скомпільованому проекті. В тому ж випадку коли сигнал підключається до зовнішнього виводу ПЛІС його наявність в результуючому проекті є обов'язковою. Це може призвести зміни поведінки та швидкодії проекту.

При використанні засобу SignalProbe підключення відбувається до вже скомпільованого проекту і його внутрішня структура не змінюється. При використанні засобу від лагодження SignalProbe слід пам'ятати, що він може використовуватись лише з новими сімействами мікросхем ПЛІС Altera: Arria GX, Stratix, Cyclone, MAX II.

Розглянемо порядок роботи з засобом SignalProbe, яка наведена на рисунку 5.51 [4].

Робота з SignalProbe виконується на тому етапі, коли проект вже скомпільовано і виконані всі призначення в проекті.

Сигнал для SignalProbe може бути як комбінаційним, так і регістровим. Для додавання сигналу в список SignalProbe необхідно відкрити діалог **SignalProbe Pins**, який доступний через пункт меню **Tools → SignalProbe Pins**. Вигляд вікна показаний на рисунку 5.52. Розглянемо елементи цього вікна.

**Current and potential SignalProbe pins** – поточні та потенційні виводи мікросхеми ПЛІС, що можуть використовуватись для SignalProbe.

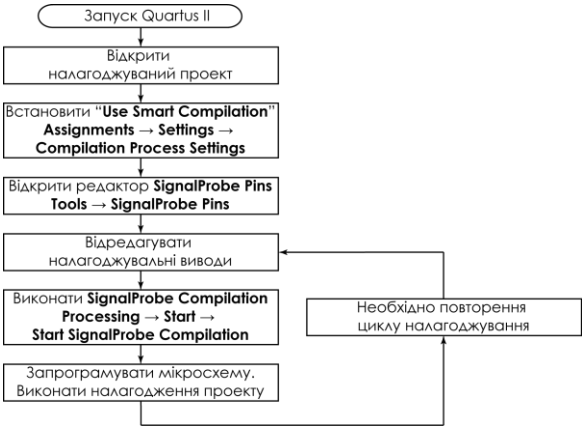



*Рисунок 5.51 – Послідовність роботи з SignalProbe*

**SignalProbe pin settings** – налаштування для виводів SignalProbe. Для додавання виводу в SignalProbe необхідно визначити його ім'я в полі **Pin name**, джерело сигналу для цього виводу (**Source**), а також додаткові параметри, які визначають доступність цього сигналу для роботи (**SignalProbe enable**), тактовий сигнал для синхронізації виводу (**Clock**), кількість регістрів, що будуть розташовані між джерелом сигналу та виводом мікросхеми (**Registers**), логічні рівні на виході мікросхеми (**I/O standard**).

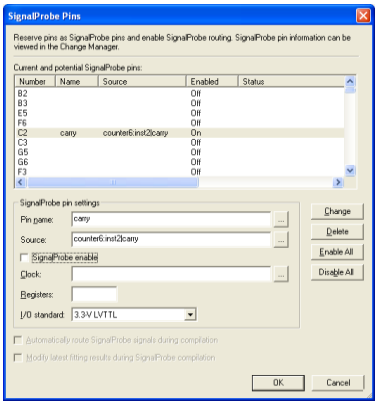



*Рисунок 5.52 – Діалог додавання виводів* ***SignalProbe Pins***

Для визначення джерела сигналу використовується той самий діалог Node Finder, який розглядався в параграфі 4.7 при описі стимулятора пакету Quartus II. Відмінність полягає у тому, що у фільтрі Node Finder необхідно обрати пункт SignalProbe, який обирає вузли, що доступні для SignalProbe після компіляції проекту (рисунок 5.53).

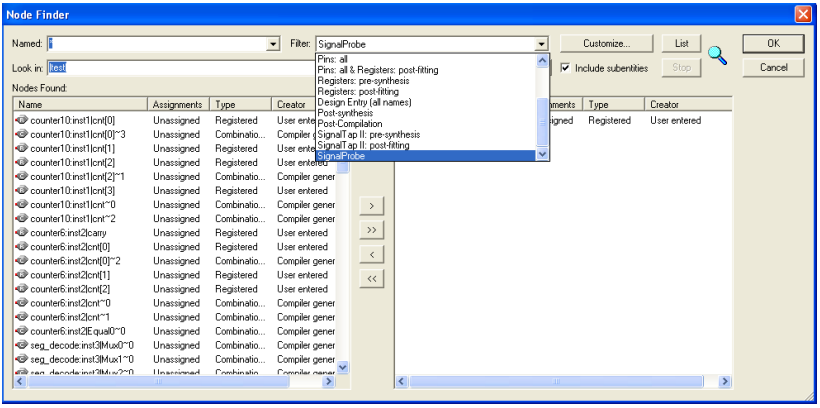



*Рисунок 5.53 – Фільтрація вузлів SignalProbe у Node Finder*

Така фільтрація необхідна оскільки частина вузлів зникає після компіляції проекту і не всі вузли можуть бути розведені як SignalProbe виводи. Наприклад, регістри та вузли гігабітного трансивера у мікросхемі Stratix IV не можуть використовуватись, оскільки нема можливості вивести ці сигнали на виводи мікросхеми.

Після призначення виводів SignalProbe необхідно виконати компіляцію SignalProbe Compilation. Вона відрізняється тим, що компіляція проводиться лише для доданих виводів SignalProbe і лишає незмінною сам проект.

## 5.6.2. Інтерфейс для зовнішнього логічного аналізатора

Логічний аналізатор – це пристрій, що призначений для запису та аналізу цифрових послідовностей. Структурна схема логічного аналізатора зображена на рисунку 5.54.

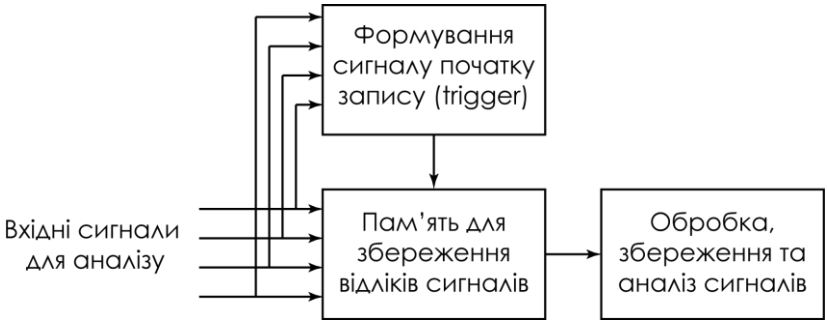



Рисунок 5.54 – Структурна схема логічного аналізатора

Логічний аналізатор містить:

- Пам’ять для запису відліків вхідних сигналів;
- Засоби для керування записом вхідних сигналів в пам’ять;

– Засоби для підключення до сигналів, які будуть досліджуватись;
– Засоби відображення та аналізу сигналів.

Для запису заданої кількості відліків послідовності вхідних цифрових сигналів подається команда початку запису (Trigger). Відліки із заданою тактовою частотою записуються до пам'яті логічного аналізатора, а потім виводяться для відображення та аналізу.

Пакет Quartus II дозволяє використовувати зовнішній логічний аналізатор для налагодження проекту. При цьому можливе використання двох типів логічних аналізаторів: логічного аналізатора SignalTap II та інтерфейсу зовнішнього логічного аналізатора. Опишемо особливості обох логічних аналізаторів.

**Кількість відліків для аналізу.**

При використанні зовнішнього логічного аналізатора кількість сигналів та їх відліків обмежується лише самим логічним аналізатором. При використанні SignalTap II доступно лише 128Кбіт пам'яті.

**Робота з даними в реальному часі.**

Зовнішній логічний аналізатор дозволяє працювати з даними в реальному часі. SignalTap II працює з тими даними, які записані по тактовому імпульсу.

**Кількість необхідних ресурсів мікросхеми ПЛІС.**

Інтерфейс зовнішнього логічного аналізатора потребує значно менше ресурсів, ніж логічний аналізатор SignalTap II.

**Використання виводів ПЛІС.**

Зовнішній логічний аналізатор потребує стільки виводів, скільки сигналів буде виведено зовні. Логічний аналізатор SignalTap II не потребує додаткових виводів ПЛІС окрім ттих, що використовувались при програмування мікросхеми через інтерфейс JTAG.

**Тактова частота.**

При роботі з SignalTap II можливе тактування внутрішніх регістрів з частотою до 200 МГц. У зовнішньому логічному аналізаторі для роботи з такими частотами необхідно виконувати узгодження ліній для боротьби зі спотвореннями сигналів.

**Необхідність додаткового обладнання.**

При роботі з SignalTap II ніяке додаткове обладнання не потрібно. Відліки сигналів передаються напряму до пакету Quartus II. При роботі із зовнішнім логічним аналізатором необхідний власне сам логічний аналізатор, який не завжди може бути під рукою.

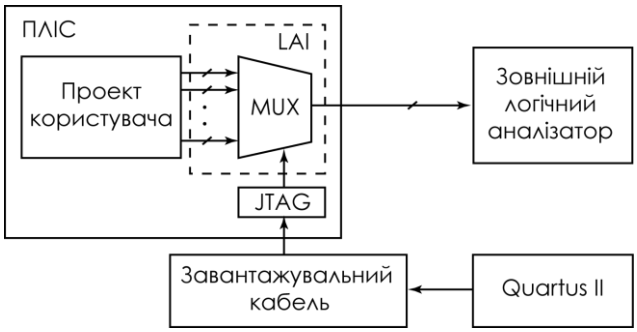



*Рисунок 5.55 – Інтерфейс для роботи з зовнішнім логічним аналізатором*

Спрощена схема підключення зовнішнього логічного аналізатора виглядає наступним чином (рисунок 5.55): в мікросхему ПЛІС вбудовується інтерфейс зовнішнього логічного аналізатора (Logic Analyzer Interface – LAI), який вже видає сигнали на виводи мікросхеми і далі на зовнішній логічний аналізатор. Кожен такий модуль являє собою параметризований блок, в якому можна змінювати кількість входів та їх розрядність. Керування інтерфейсом логічного

аналізатора відбувається через JTAG порт або за допомогою пакету Quartus II або самим зовнішнім логічним аналізатором.

Алгоритм роботи з LAI показаний на рисунку 5.56.

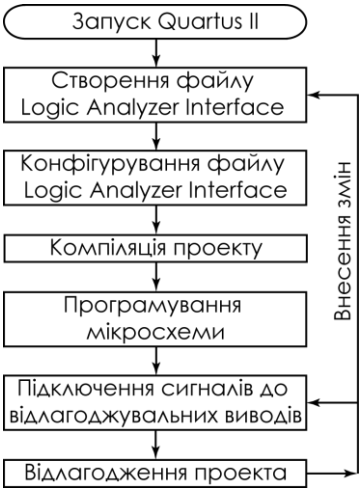



*Рисунок 5.56 – Алгоритм роботи з Logic Analyzer Interface*

Розглянемо всі етапи роботи с інтерфейсом логічного аналізатора. LAI вбудовується у вже готовий та відкомпільований проект. Для додавання модуля в проект можна використати один з двох варіантів:

1. Обрати пункт меню **File → New**. У списку необхідно обрати пункт **Logic Analyzer Interface File** розділу **Verification/Debagging Files**.

2. Обрати пункт меню **Tools → Logic Analyzer Interface Editor**.

У результаті з’явиться вікно інтерфейсу логічного аналізатора, показане на рисунку 5.57.

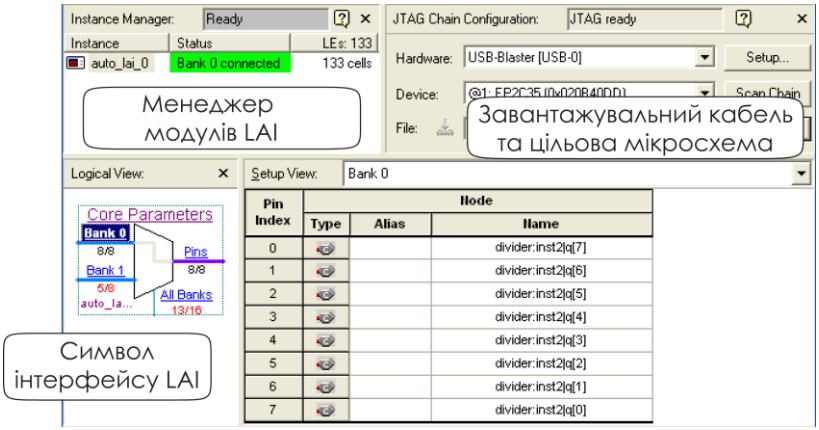



*Рисунок 5.57 – Вікно інтерфейсу логічного аналізатора*

Вікно містить наступні панелі.

**Менеджер блоків (Instance Manager)**. В цьому вікні відображається список модулів LAI, які існують у проекті (**Instance**), їх статус (**Status**) та кількість ресурсів, які необхідні для реалізації конкретного модуля. В даному випадку необхідні лише логічні елементи (**LEs**).

**Вигляд обраного модуля (Logical View)**. Ця панель використовується для зміни параметрів всього блоку в цілому, входів та виходів. Для переходу між різними об'єктами одного модуля необхідно виконати подвійне клацання по обраному елементу. В цьому випадку праворуч відкриються параметри цього об'єкта (**Setup View**).

Для всього блока вцілому (**Core Parameters**) доступні такі параметри:

1. Кількість виводів, що підключаються до зовнішнього логічного аналізатора (**Pin count**). Цей параметр може приймати значення від 1 до 255 і обмежений кількістю доступних виводів мікросхеми у конкретному проекті.

2. Кількість вхідних шин, що підключаються до мультиплексора LAI (**Bank count**). Їх розрядність визначається розрядністю вихідної шини (**Pin count**).

3. Режим роботи логічного аналізатора (**Output/Capture mode**). Аналізатор може працювати у двох режимах: регістровому (**Registered/State**) та комбінаційному (**Combinational/Timing**).

Регістровий режим використовує зовнішній сигнал від логічного аналізатора для запису даних. Оскільки цей тактовий сигнал є асинхронним до проекту на ПЛІС, то необхідно чітко визначити його значення. Такий режим найбільш часто використовується для визначення часових параметрів сигналів, наприклад, для визначення затримки сигнали при передачі його по каналу зв'язку. При виборі цього варіанту роботи необхідно визначити тактовий сигнал у пункті Clock. Це може бути будь-який сигнал проекту, але бажано, щоб він синхронним з основною тактовою частотою проетку.

Комбінаційний режим використовує для тактування внутрішній сигнал від системи і є повністю синхронним для проекту на ПЛІС. Цей режим використовують для перевірки правильності функціонування проекту.

4. Стан виводів при ввімкненні системи (**Power-up state**).

Параметри входів (**Banks**) показані на рисунку 5.58.

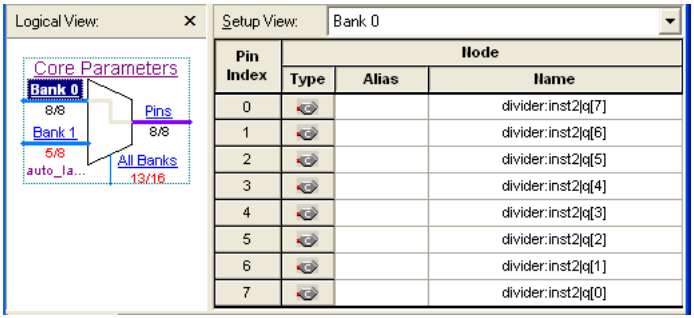
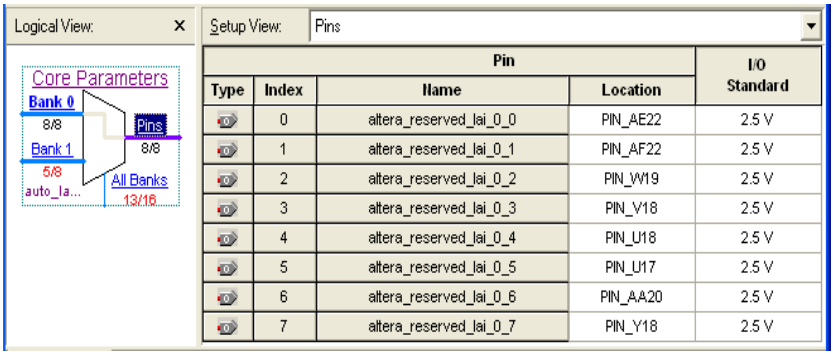



*Рисунок 5.58 – Параметри вхідних сигналів LAI*

Для кожної шини вхідних сигналів необхідно визначити джерело сигналу. Якщо виводу з шини не призначено джерело сигналу, то він буде під'єднаний до землі. В стовпці **Alias** можуть бути вказані синоніми назв сигналів для спрощення сприйняття.

Визначення параметрів виходів (**Pins**) потребує лише призначення виводів ПЛІС аналогічно тому, як це описано в параграфі 4.5. В результаті буде отримано таблицю, аналогічну показаній на рисунку 5.59.




| Pin | | | | I/O Standard |
|---|---|---|---|---|
| Type | Index | Name | Location | |
| | 0 | altera_reserved_lai_0_0 | PIN_AE22 | 2.5 V |
| | 1 | altera_reserved_lai_0_1 | PIN_AF22 | 2.5 V |
| | 2 | altera_reserved_lai_0_2 | PIN_W19 | 2.5 V |
| | 3 | altera_reserved_lai_0_3 | PIN_V18 | 2.5 V |
| | 4 | altera_reserved_lai_0_4 | PIN_U18 | 2.5 V |
| | 5 | altera_reserved_lai_0_5 | PIN_U17 | 2.5 V |
| | 6 | altera_reserved_lai_0_6 | PIN_AA20 | 2.5 V |
| | 7 | altera_reserved_lai_0_7 | PIN_Y18 | 2.5 V |

*Рисунок 5.59 – Параметри вихідних сигналів LAI*

Після виконання всіх призначень у інтерфейсі зовнішнього логічного аналізатора необхідно перекомпілювати проект. Для підвищення швидкості компіляції доцільно використовувати Incremental Compilation.

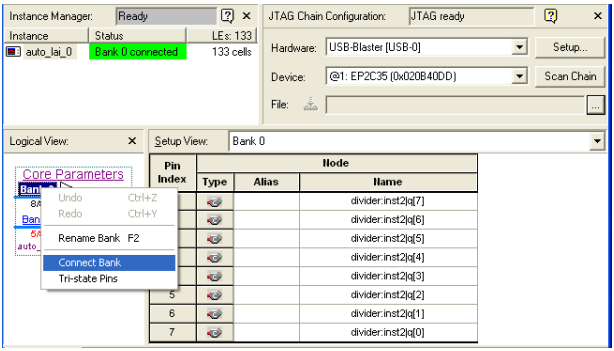



*Рисунок 5.60 – Підключення вхідної шини до виходу*

Після компіляції і програмування мікросхеми ПЛІС можна проводити налагодження проекту. Для цього необхідно під'єднати логічний аналізатор до пристрою і у вікні інтерфейсу логічного аналізатора обрати ту шину вхідних сигналів, яка буде подаватися на вихід. Для цього необхідно просто клацнути правою кнопкою миші по назві вхідної шини і обрати пункту **Connect Bank** (рисунок 5.60).

## 5.6.3. Редактор вмісту вбудованої пам'яті

Редактор вбудованої пам'яті (In-System Memory Content Editor) дозволяє швидко переглянути та редагувати вміст пам'яті в працюючій системі, наприклад, вміст комунікаційного пакету, підготовленого для передачі, або вміст пам'яті вбудованого контролера. Інше призначення цього редактора – генерація тестових векторів для проекту. Для цього використовується вільний блок пам'яті, в який підключається до того модуля, який необхідно протестувати. Додавши просту схему тактування та генерації адреси можна отримати генератор сигналів для тестового модуля.

Для своєї роботи редактор вміщує однопортовий модуль пам'яті у двохпортовий. У двохпортовому модулі один порт використовується для роботи самої пам'яті, а другий – підключається до JTAG порту для перегляду і редагування.

Тому для редагування можна використовувати такі категорії модулів: LPM_CONSTANT, RAM: 1-PORT, ROM: 1-PORT, ALTSYNCRAM, LPM_RAM_DQ, LPM_ROM.

Для роботи редактора вбудованої пам'яті необхідно виконати такі операції:

1. Визначити які модулі будуть редагуватись.
2. Додати у визначені модулі можливість їх редагування в системі.
3. Виконати повну компіляцію проекту.
4. Запрограмувати мікросхему.
5. Запустити In-System Memory Content Editor.
6. Підключитися до мікросхеми через JTAG-порт.
7. Виконати редагування.

Розглянемо послідовно необхідні операції. Роботу з редактором розглянемо на прикладі однопортового ПЗП.

Для додавання у модуль пам'яті можливості редагування необхідно використовувати **Mega-Wizard Plug-In Editor**. На рисунку 5.61 показане вікно **Mega-Wizard Plug-In Editor**, в якому ввімкнена можливість редагування вмісту пам'яті.

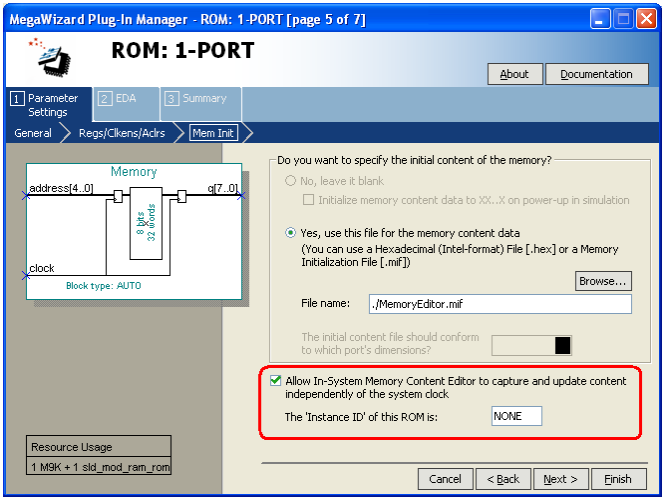



*Рисунок 5.61 – Вмикання редактора вмісту пам'яті у Mega-Wizard Plug-In Editor*

Після компіляції та програмування мікросхеми необхідно запустити редактор вбудованої пам'яті. Для цього обрати пункт меню **Tools → In-System Memory Content Editor**. З'явиться вікно, зображене на рисунку 5.62.

Як видно, налаштування JTAG в редакторі вбудованої пам'яті такі ж самі, як і в інтерфейсі логічного аналізатора. Відрізняється область, в якій відображається вміст вбудованої пам'яті. Тут відображається вміст, статус та характеристики того модуля пам'яті, який обраний для аналізу. Редактор дозволяє одноразове (**Read Data from In-System Memory**) та циклічне читання (**Continuously Read Data from In-System Memory**), зупинку циклічного читання (**Stop In-System Memory Analysis**) та запис в пам'ять (**Write Data from In-System Memory**). Для виконання цих операцій не потрібне переконфігурування мікросхеми ПЛІС.

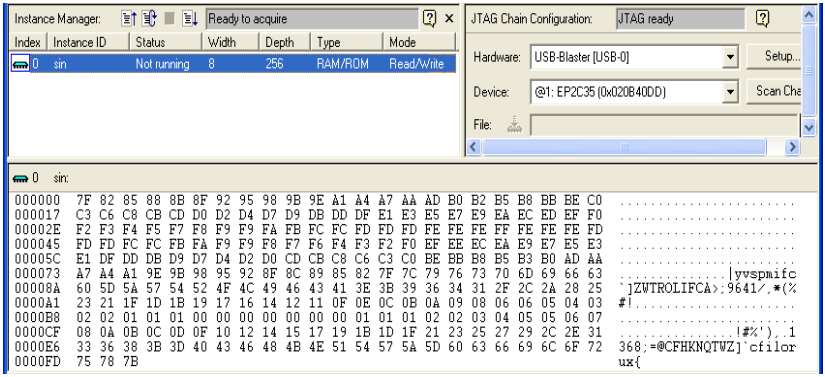



*Рисунок 5.62 – Редактор вмісту вбудованої пам'яті*

Дані в редакторі пам'яті відображаються у шістнадцятковому коді та у вигляді ASCII символів. У тому випадку, коли код не може бути відображений у вигляді ASCII символу, він замінюється крапкою (.). Значення в області даних відображаються різними кольорами в залежності від статусу даних:

- чорним кольором, якщо дані, що зчитані з ПЛІС співпадають з попередніми даними в редакторі;
- червоним, якщо дані в редакторі та зчитані дані відрізняються;
- синім, якщо дані змінені, але ще не записані у пам'ять мікросхеми.

## 5.6.4. Вбудований логічний аналізатор Signal Tap II

Вбудований логічний аналізатор Signal Tap II, як і звичайний логічний аналізатор, включає в себе пам'ять для збереження відліків сигналів та систему керування (рисунок 5.63). Для своєї роботи Signal Tap II використовує вбудовану пам'ять ПЛІС для збереження відліків сигналів та схему керування, реалізовану всередині ПЛІС, яка визначає алгоритм запису відліків сигналів. Для відображення інформації використовується персональний комп'ютер, який під'єднаний до відлагоджувальної плати за допомогою завантажувального кабелю. Керування логічним аналізатором виконується за допомогою пакету Quartus II.

Слід відмітити деякі особливості логічного аналізатора Signal Tap II. По-перше, Signal Tap II може використовуватись лише в мікросхемах типу FPGA. По-друге, для використання вбудованого логічного аналізатора необхідна деяка кількість логічних елементів та вбудованої пам'яті ПЛІС. Кількість ресурсів логіки залежить від кількості сигналів, що аналізуються та умов зчитування даних. Об'єм пам'яті, що буде використаний логічним аналізатором, залежить від кількості каналів та їх розрядності. Тому необхідно розуміти, що можливості логічного аналізатора напряму залежать від вільних ресурсів мікросхеми. По-третє, Signal Tap II зчитує внутрішні сигнали лише за переднім фронтом тактового сигналу. Тому при його використанні доступний лише функціональний аналіз роботи проекту. Будь-який аналіз роботи з реальними затримками сигналів в цьому випадку не буде доступний.

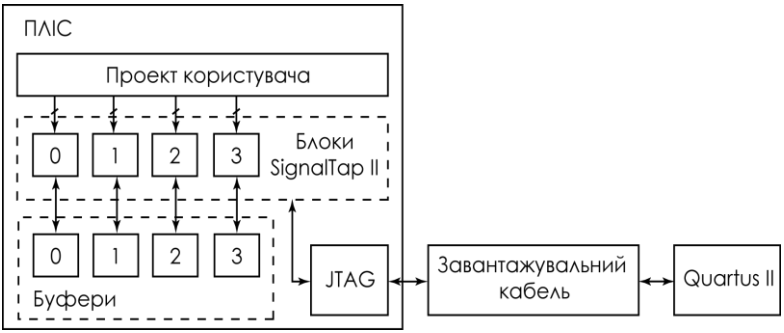



*Рисунок 5.63 – Структура логічного аналізатора Signal Tap II*

Для використання Signal Tap II в своєму проекті необхідно виконати таку послідовність дій:

1. Створити новий або відкрити існуючий проект.
2. Додати модуль Signal Tap II до проекту та сконфігурувати його.
3. Визначити умови початку захоплення даних (Data Triggering).
4. Скомпілювати проект.
5. Завантажити проект до мікросхеми ПЛІС.
6. Запустити Signal Tap II в пакеті Quartus II.
7. Виконати запис даних у буферну пам’ять (Data Capture).
8. Провести аналіз записаних даних (Data Analysis).

Для того щоб додати модуль Signal Tap II до проекту необхідно використати MegaWizard Plug-In Manager (рисунок 5.64) або створити новий файл логічного аналізатора. Створений файл автоматично буде доданий до відкритого проекту. Другий варіант є більш розповсюдженим і є рекомендованим для роботи у більшості випадків. Тому в цій книзі ми будемо розглядати саме цей варіант.

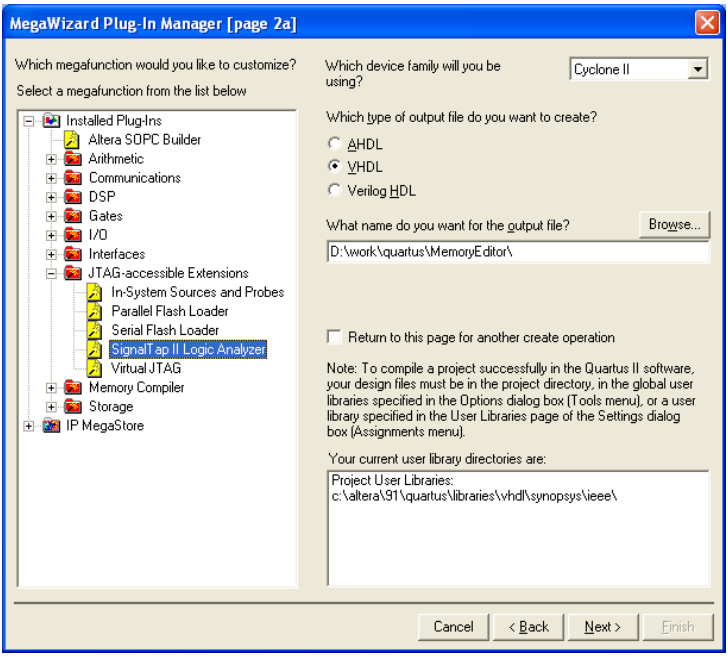
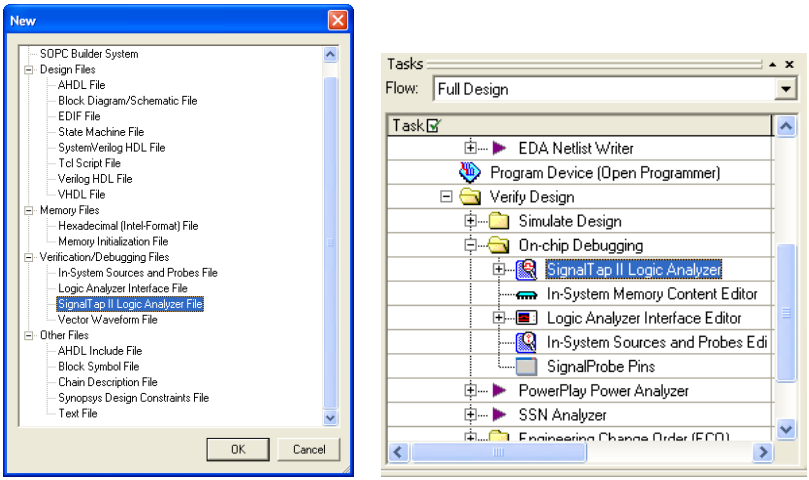



*Рисунок 5.64 – Запуск Signal Tap II за допомогою MegaWizard Plug-In Manager*

Для створення нового файлу логічного аналізатора можна використати один з трьох варіантів: пункт меню **File → New → Signal Tap II Logic Analyzer** File (рисунок 5.65 а), обрати пункт меню **Tools →** або обрати пункт **Signal Tap II Logic Analyzer** списку задач пакету Quartus II (рисунок 5.65. б).




*а* *б*

*Рисунок 5.65 – Створення нового файлу Signal Tap II*

У будь-якому випадку це призведе до відкриття вікна логічного аналізатора (рисунок 5.66).

В цьому вікні є вже як знайомі так і нові елементи. Діалоги **Instance Manager** та **JTAG Chain Configuration** вже використовувались вище при описі **SignalProbe** і тут описуватись не будуть. Зупинимось більш докладно на двох діалогах – списку сигналів з закладками **Data** та **Setup** і конфігурація сигналів (**Signal Configuration**).

Список сигналів з закладки **Setup** показаний на рисунку 5.67. Для додавання сигналів необхідно двічі клацнути в полі **Node**. Буде відкритий стандартний діалог **Node Finder**, в якому виконується фільтрація сигналів **Signal Tap II: post-fitting** або **Signal Tap II: pre-synthesis**. У першому випадку будуть показані сигнали, які наявні в проекті після синтезу проекту та його розміщення на мікросхемі. Такі сигнали будуть показані синім. У другому випадку будуть показані сигнали, які отримані після першого етапу компіляції – перевірки синтаксису та побудови ієрархії проекту. Такі сигнали будуть показані

чорним. Після повної компіляції та оптимізації проекту можлива ситуація, коли частина сигналів зникає. Такі сигнали будуть показані у списку сигналів червоним.

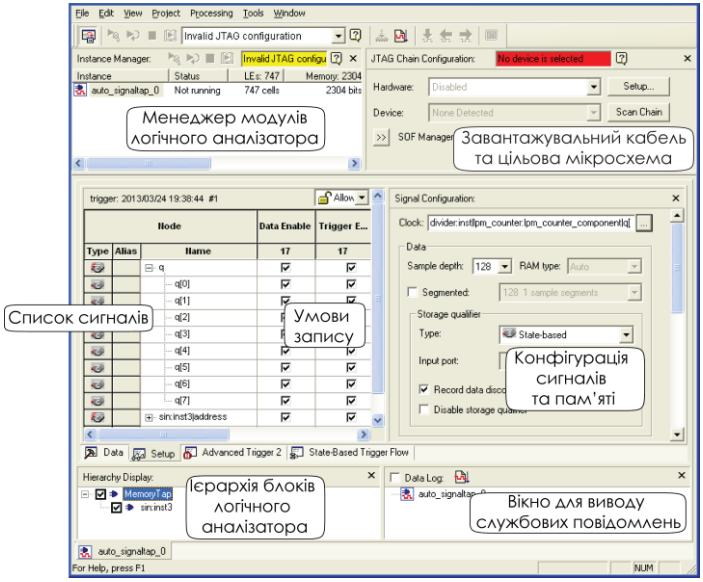



*Рисунок 5.66 – Вікно логічного аналізатора Signal Tap II*

Після того, як сигнал буде додано, з'являється можливість конфігурації того, як буде проводитись аналіз сигналу. Для цього використовуються кнопки **Data Enable** та **Trigger Enable**. Кнопка **Data Enable** дозволяє або забороняє запис даних з обраного сигналу. Це дає можливість збільшувати або зменшувати об'єм пам'яті, що використовується логічним аналізатором. Якщо кнопка **Trigger Enable** відключена, то це означає, що обраний сигнал не може використовуватись у створенні умов запису. В свою чергу це зменшує логічну схему аналізатора. Сигнал, у якого буде відключена кнопка **Data Enable**, але ввімкнена **Trigger Enable** не записується в пам'ять, але включається в умову запису інших сигналів.

Тут необхідно зупинитись і з'ясувати значення поняття тригер (**Trigger**). При своїй роботі дані проходять через логічний аналізатор, але не виконується зупинка запису. Як тільки умова запису виконується запис сигналів припиняється і на екран виводяться записані сигнали. Умова запису і називається тригером.

Далі у вікні розташовується перелік умов запису сигналів. Для кожного сигналу може бути обрано до десяти різних умов запису. Кількість умов запису визначається у діалозі параметрів тригера і описана нижче. Можна обрати один з двох варіантів умов запису – загальний (**Basic**) або розширений (**Advanced**).

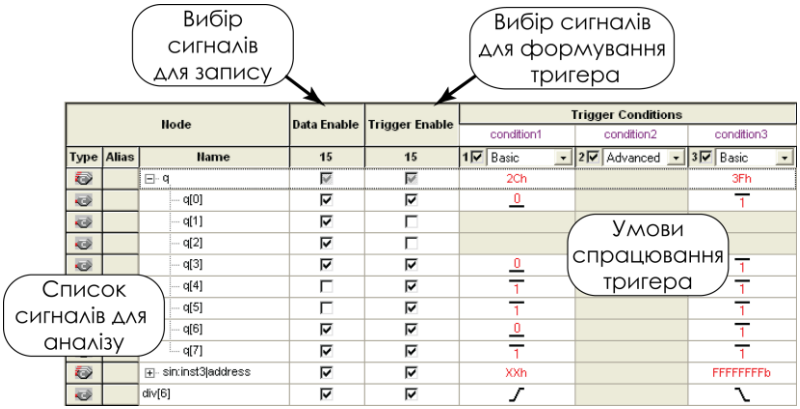



*Рисунок 5.67 – Список сигналів у логічному аналізаторі* ***Signal Tap II***

Загальний метод запису передбачає виконання логічної функції І над всіма сигналами або групами, які знаходяться у даному стовпчику. Для зміни значення сигналу в контекстному меню можна обрати такі значення (рисунок 5.68):

- **Don't Care** – рівень сигналу не має значення
- **Low** – низький рівень сигналу;
- **Falling Edge** – задній фронт сигналу;
- **Rising Edge** –передній фронт сигналу;

– **High** – високий рівень сигналу;
– **Either Edge** – будь-яка зміна сигналу;
– **Insert Value** – введення значення групи сигналів.

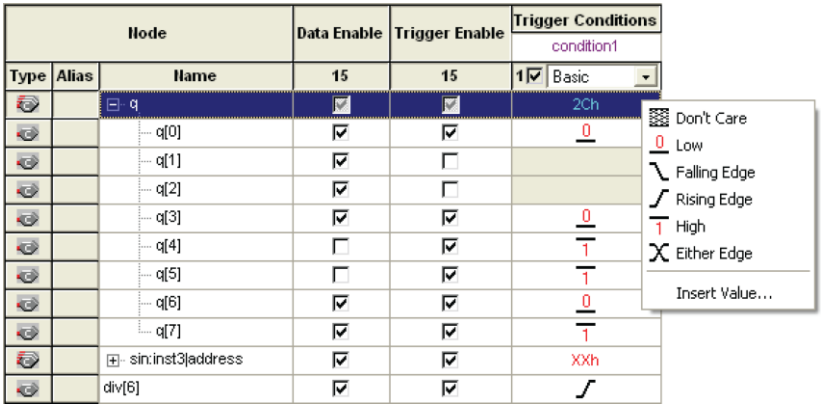



*Рисунок 5.68 – Загальні умови запису сигналів*

Запис обраних сигналів буде відбуватись кожного разу, коли під час фронту тактового сигналу будуть виконуватись всі умови, що записані у стовбці.

Розширений варіант (**Advanced**) дає можливість створювати більш складні логічні рівняння для запуску запису сигналів. При виборі цього варіанту відкривається графічний редактор, в якому за допомогою простих блоків будується логічне рівняння (рисунок 5.69).

Вікно конфігурації сигналу (**Signal Configuration**), яке знаходить праворуч від списку виводів (рисунок 5.70) дозволяє визначити тактовий сигнал, розмір буфера даних та його властивості, умови запису. Розглянемо ці пункти докладніше.

Найперше це призначення тактового сигналу (**Clock**). Зчитування сигналів логічним аналізатором буде відбуватись по кожному фронту тактового сигналу. Бажано обирати сигнал, який виконує функцію тактового в тому тактовому демені, до якого належить сигнал, що досліджується. Чим більшу частоту

має тактовий сигнал, тим більшу точність буде забезпечувати логічний аналізатор. Разом з цим це призведе до збільшення обсягу пам'яті, що необхідна для запам'ятовування відліків сигналу.

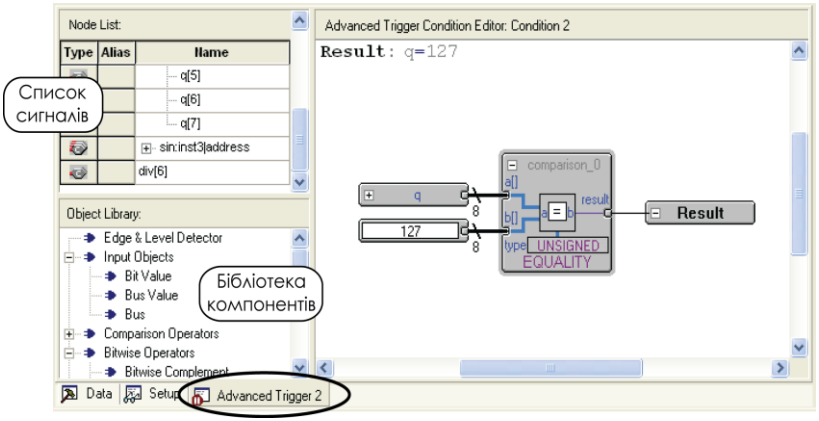



*Рисунок 5.69 – Редактор розширених опцій запису сигналів*

Об'єм пам'яті (**Sample depth**) та її тип (**RAM type**) дають змогу визначити розмір буферу для запам'ятовування відліків сигналу та тип елементів вбудованої пам'яті. Розмір буферу може змінюватись від 0 до 128 К. Але не бажано встановлювати максимальне значення, оскільки це може призвести до того, що проект з логічним аналізатором не вміститься в мікросхему. Тип пам'яті залежить від того сімейства мікросхем, що використовується для реалізації. Наприклад, для мікросхем сімейства Stratix IV це блоки пам'яті типів MLAB, M9K, M144K, а для сімейства Cyclone IV – M9K. Якщо ж тип пам'яті не важливий, то можна залишити значення **Auto**.

Наступний параметр, який необхідно визначити – це тип буфера. За замовчуванням встановлено циклічний (**Circular**)

буфер. Розробник також може обрати сегментований буфер (**Segmented**) та визначити розмір сегментів.

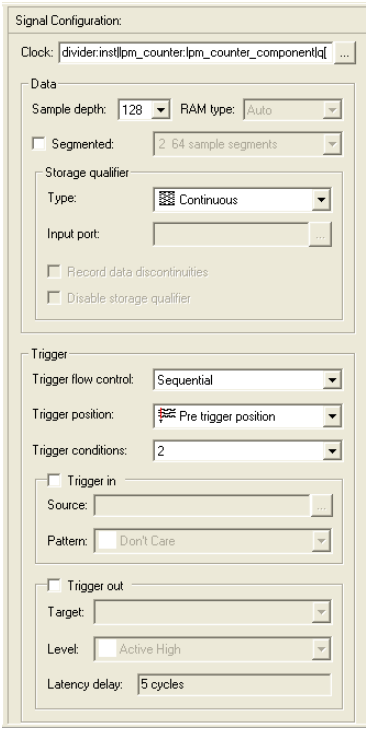



*Рисунок 5.70 – Параметри сигналів у логічному аналізаторі Signal Tap II*

Розглянемо більш докладно циклічний та сегментований буфери.

**Циклічний буфер** за своєю структурою нагадує регістр типу FIFO – First In – First Out. На рисунку 5.71 розмір циклічного буфера дорівнює 128. Дані входять в буфер і проходять крізь нього. Коли створюються умови для спрацювання тригера запис даних проводиться до заповнення буфера. В буфері дані будуть розташовані таким чином, що

точка, в якій спрацював тригер знаходиться всередині буфера (рисунок 5.71). Ті дані, які були до спрацювання умов запису (**pre-trigger**), записуються перед даними на яких відбулось спрацювання. Дані, які були після спрацювання (**post-trigger**), розташовуються після точки спрацювання.

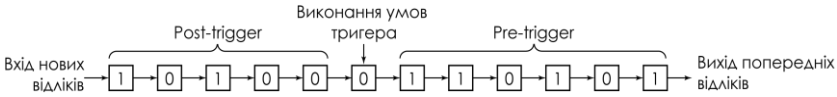



*Рисунок 5.71 – Циклічний буфер*

Приклад роботи циклічного буфера показаний на рисунку 5.72. Умова запису даних співпадає з показаною вище на рисунку 5.69 – значення вхідної шини дорівнює 127. На рисунку момент спрацювання показаний штрихованою лінією.

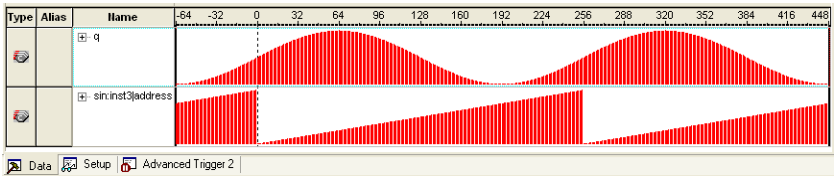



*Рисунок 5.72 – Записані дані у циклічному буфері*

**Сегментований буфер** являє собою декілька невеликих циклічних буферів. Об'єм пам'яті, який був виділений для розташування буфера логічного аналізатора поділяється не декілька невеликих (рисунок 5.73).


*Рисунок 5.73 – Сегментований буфер*

Кожен сегмент працює як окремий циклічний буфер. Якщо умова запису для всіх сегментів буфера єдина, то спочатку запис виконується в поточний сегмент до повного заповнення буфера. При новому виникненні умови запису виконується запис у наступний сегмент. Якщо ж умов запису декілька, то запис у наступний сегмент буде виконаний при повторі останньої умови запису.

Для встановлення сегментованого буфера необхідно активувати пункт **Segment** в діалозі конфігурації сигналу (**Signal Configuration**). На рисунку 5.74 загальний об'єм поділений на 4 сегменти по 32 відліку в кожному.

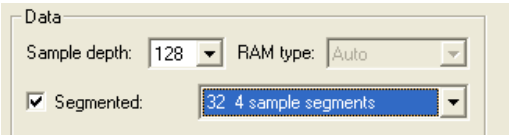



*Рисунок 5.74 – Визначення розміру сегментованого буфера*

На рисунку 5.75 показаний приклад роботи сегментованого буфера. Загальний об'єм буфера поділений на чотири сегменти і запис в окремі сегменти відбувається у випадку, коли вхідний сигнал *q[7..0]* дорівнює нулю.

Наступна група параметрів (**Storage qualifier**) уточнює параметри зберігання даних у буфері і дозволяє зберігати місце у буфері лише для тих даних, які необхідні для аналізу. Для того, щоб розібратись з роботою цих параметрів розглянемо як виконується запис даних у буфер.

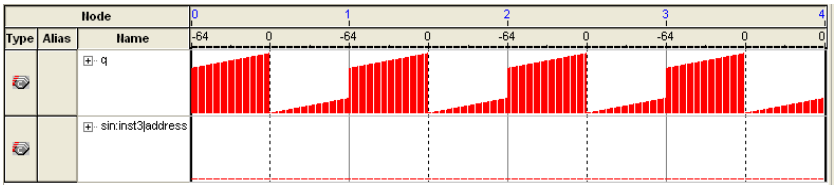



*Рисунок 5.75 – Приклад роботи сегментованого буфера*

При виконанні умов спрацювання тригера буфер заповнюється даними, що надходять з кожним тактовим імпульсом, до тих пір, поки не буде повністю заповнена вся пам'ять, яка виділена для буфера (рисунок 5.76, а). Однак, в цих даних можуть бути присутні непотрібні для аналізу відліки сигналів, а ті, що необхідні не вмістилися в буфер. Для того, щоб записати необхідні дані в буфер треба пропустити запис непотрібних даних (рисунок 76, б). Це дозволить записати додаткові дані в буфер (рисунок 76, в).

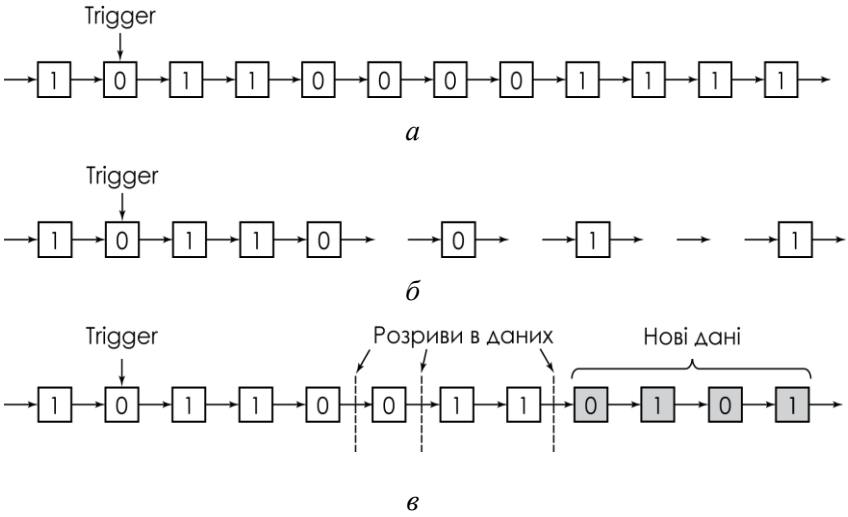



*Рисунок 5.76 – Усунення непотрібних даних з буфера та додавання нових*

Параметри спрацювання тригера можуть бути наступними:

**Continuous** – записуються усі дані, що надходять до буфера. Це режим за замовчуванням.

**Input Port** – в якості сигналу дозволу запису використовується додатковий вхідний порт. Запис виконується коли вхідний сигнал дорівнює одиниці.

**Transitional** – запис виконується в тому випадку, коли один з визначених сигналів змінює своє значення. Сигнали обираються у додатковому списку.

**Conditional** – запис виконується при виконанні логічної умови, яка формується над обраними сигналами.

**Start/Stop** – умова подібна до попередньої, але формується два набори правил – один для початку запису у буфер, другий – для зупинення запису.

**State-based** – запис контролюється цифровим автоматом і цей метод запису дає найбільші можливості по керуванню процесом запису до буфера.

Параметри спрацювання тригера – це значний перелік опцій, які будуть розглянуті нижче.

Перший параметр – це положення моменту спрацювання у буфері (**Trigger position**). Можливе використання одного з варіантів:

**Pre** – зберігаються, в основному, дані що з'явилися після спрацювання тригера. В цьому випадку 12% об'єму буфера займають дані до спрацювання тригера, 88% - після спрацювання.

**Center** – об'єм буфера поділяється на дві рівні частини – до і після спрацювання тригера.

**Post** – увага звертається на ті дані, що існували до спрацювання тригера. 88% буфера виділяється на дані до спрацювання, 12% - після.

На рисунку 5.77 показане розміщення моменту спрацювання тригера в загальному об'ємі буфера.

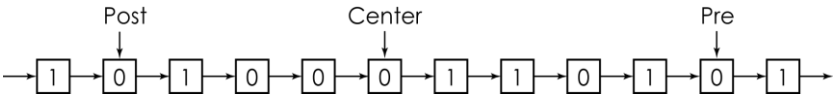



*Рисунок 5.77 – Положення моменту спрацювання тригера*

Умови роботи тригера (**Trigger flow control**). Існує два механізми регулювання умов спрацювання тригера:

послідовний (**Sequential**) та на основі цифрового автомата (**State-based**).

**Послідовний механізм** спрацювання тригера передбачає, що логічний аналізатор послідовно розраховує умови спрацювання тригера – **Trigger Conditions**. Якщо виконується перша умова, то розраховується друга. Коли вона виконується – логічний аналізатор переходить до наступної. І так до того часу поки не буде виконана остання умова спрацювання тригера. Тоді буде виконаний запис до буфера. Максимальна кількість умов спрацювання – десять і можлива робота як з основними (**Basic**) та і з розширеними (**Advanced**) умовами спрацювання тригера.

Існує різниця між роботою послідовного механізму при роботі з циклічним та сегментованим буферами. При роботі з циклічним буфером після запису інформації до буфера виконується перехід до аналізу першої умови і процес повторюється знову. При роботі з сегментованим буфером умови спрацювання також аналізуються послідовно, одна за одною. Коли ж виконується остання умова виконується запис до першого сегмента. Після цього переходу до першої умови не відбувається, а аналізується остання умова спрацювання. Коли вона знову виконується робиться запис до другого сегменту буфера. І так відбувається до тих пір, поки не буде заповнений весь сегментований буфер. Тільки після цього відбувається перехід до аналізу першої умови спрацювання.

Механізм **спрацювання тригера на основі цифрового автомата** дає можливість повного контролю над записом інформації до буфера. Для реалізації цієї можливості створюється до 20 станів цифрового автомата, під керівництвом якого і виконується запис до буфера. У кожному зі станів цифрового автомата розраховується одна або декілька умов спрацювання тригера, перевіряється значення лічильника. На основі цих даних формується таблиця переходів цифрового автомата. У циклічному буфері один зі станів, зазвичай це останній стан, після виконання необхідних умов запускає запис у буфер. Для сегментованого буфера будь-який стан може записувати інформацію до сегменту буфера.

При виборі у списку умов тригера механізму спрацювання тригера на основі цифрового автомата з'являється нова закладка **State-Based Trigger Flow**, у якій визначається кількість станів цифрового автомата, прапорців та лічильників. Умови переходу до наступного стану описуються за допомогою спеціальної мови опису цифрових автоматів (**Trigger Flow Description Language**). Приклад побудованого цифрового автомата для контролю запису до буфера показаний на рисунку 5.78.

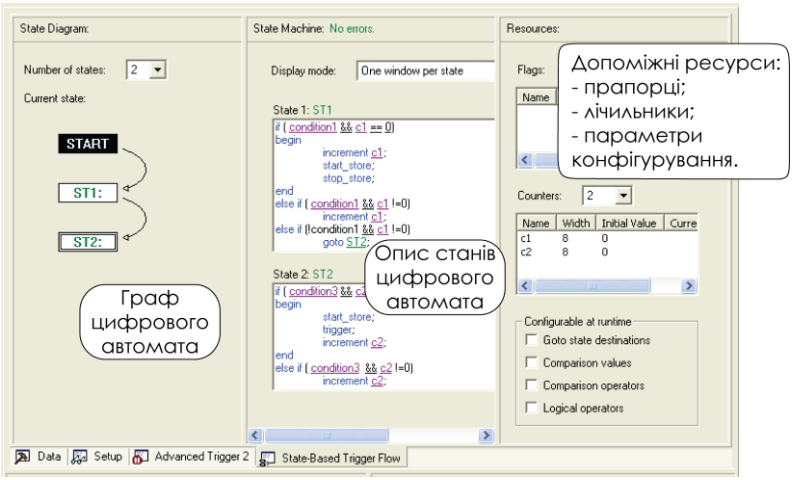



*Рисунок 5.78 – Цифровий автомат умов запису до буфера*

Після визначення всіх умов роботи логічного аналізатора створюється файл .stp, який автоматично під'єднується до проекту. Для роботи логічного аналізатора необхідно виконати повну компіляцію проекту. В результаті буде створено два нових об'єкти: sld_signaltap та sld_hub, які забезпечують логіку роботи логічного аналізатора та інтерфейс з JTAG-портом (рисунок 5.79).

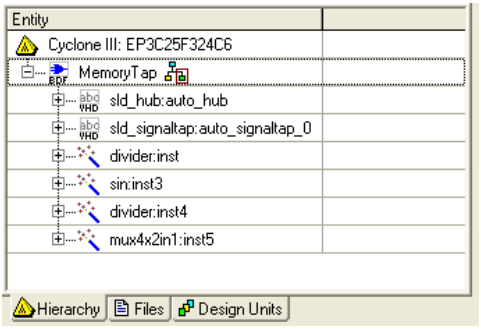



*Рисунок 5.79 – Модулі логічного аналізатора в ієрархії проекту*

Після компіляції у звіті можна переглянути список сигналів, що під'єднанні до логічного аналізатора (рисунок 5.80). Він знаходиться у пункті **Connection to In-System Debugging Instance** на сторінці **In-System Debugging** папки **Analysis & Synthesis**.

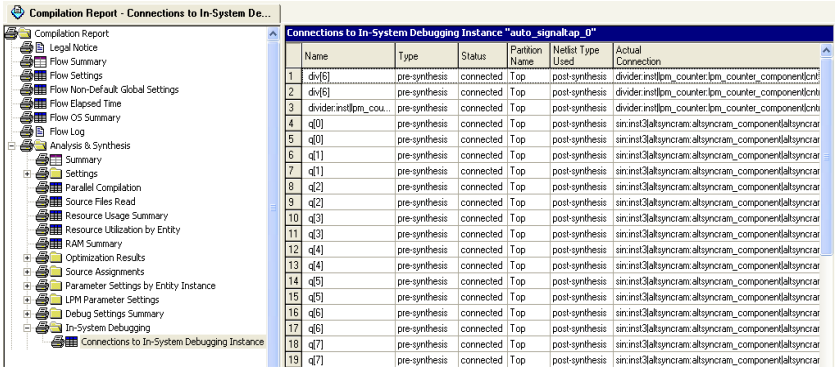



*Рисунок 5.80 - Список сигналів, що під'єднанні до логічного аналізатора*

Програмування проектів з логічним аналізатором нічим не відрізняється від звичайного програмування ПЛІС, яке описане в параграфі 4.8.

**Запуск та робота з логічним аналізатором.**

Всі операції з логічним аналізатором контролюються за допомогою чотирьох кнопок на панелі інструментів головного вікна логічного аналізатора (таблиця 5.6).

На рисунках 5.72 та 5.75 показані діаграми сигналів, що були отримані з логічного аналізатора. По горизонтальній вісі наводяться номера відліків сигналів. Відлік, на якому відбулося спрацювання тригера має номер 0. Попередні дані мають від'ємні номери, а наступні – додатні. Діаграми можуть бути збережені у вигляді файлів різних форматів (.csv, .tbl, vwf, jpg, bmp) за допомогою меню **File → Export** (рисунок 5.81).

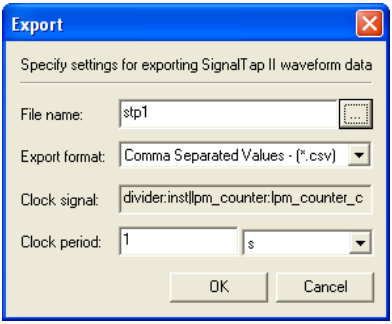



*Рисунок 5.81 – Діалог експорту даних з логічного аналізатора*

Крім того, сигнали можна переглянути дані у вигляді текстового файлу. Для цього необхідно у контекстному меню діаграм обрати пункт **Create SignalTap II List File.** Приклад отриманого файла наведений нижче.

```
Signal Legend:
Key         Signal Name
0         = q
1         = sin:inst3|address
Data Table:
Signals-> 0   1
sample
```

```
-64        0    192
-63        0    193
-62        0    194
-61        0    195
-60        1    196
-59        1    197
-58        1    198
-57        2    199
-56        2    200
-55        3    201
-54        4    202
-53        5    203
```

*Таблиця 5.6 – Кнопки управління логічним аналізатором*

| Кнопка | Опис функції |
|---|---|
|  | Запуск обраного екземпляра логічного аналізатора. Робота продовжується до тих пір, поки не буде виконана умова спрацювання тригера або логічний аналізатор не зупиниться. |
|  | Запуск логічного аналізатора у режимі повтору, постійно оновлюючи інформацію на екрані дисплея. |
|  | Зупинка логічного аналізатора і відображення інформації на дисплеї. |
|  | Зчитування інформації з буфера до персонального комп’ютера. Використовується у тому випадку, коли інші способи зчитування інформації з буфера не спрацьовують. |

Під час своєї роботи у діалозі **Instance Manager** відображається статус обраного екземпляра логічного аналізатора. Перелік статусів показаний у таблиці 5.7.

*Таблиця 5.7 – Статуси екземпляра логічного аналізатора*

| Статус | Значення повідомлення |
|---|---|
| Not Running | Нема з'єднання JTAG або не підключена мікросхема ПЛІС. |
| Ready to Acquire | З'єднання виконане, очікується старт аналізатора. |
| Waiting for trigger | Аналізатор запущений, очікується умови спрацювання тригера. |
| Acquiring pre-trigger data | Очікується заповнення буфера перед подією спрацювання тригера. |
| Acquiring post trigger data | Умова спрацювання тригера виконана, закінчується заповнення буфера. |
| Offload acquired data | Передаються дані з буфера до файлу логічного аналізатора в ПК. |

## *Література*

1. Грушвицкий Р.И., Михайлов М.М. Проектирование в условиях временных ограничений: отладка проектов//Компоненты и технологии, 2007. - № 6, 8, 9.

2. Грушвицкий Р.И., Шашкин П.М. Проектирование в условиях временных ограничений: компиляция проектов//Компоненты и технологии, 2011. - № 1-3.

***3.*** Грушвицкий Р.И., Шашкин П.М. Проектирование в условиях временных ограничений: компиляция проектов//Компоненты и технологии, 2008. - № 1. – С. 102-105.

4. Грушвицкий Р.И., Шашкин П.М. Проектирование в условиях временных ограничений: верификация проектов//Компоненты и технологии, 2008. - № 3, 5, 6, 8.

5. Строгонов А. Проектирование цифровых автоматов//Компоненты и технологии, 2008. - № 4. – С. 149-152.

6. Строгонов А., Быстрицкий А. Эффективность разработки конечных автоматов в базисе ПЛИС FPGA//Компоненты и технологии, 2013. - № 1. – С. 134-139.

7. Quartus II Handbook. Version 9.1. Altera, 2009. – 1820 p.

8. Taylor A. How to implement state machines in your FPGA//XCell, 2012. - № . – P. 52 – 57.

Наукове видання

**Іванець** Сергій Анатолійович,
**Зубань** Юрій Олександрович,
**Казимир** Володимир Вікторович,
**Литвинов** Віталій Васильович

# ПРОЕКТУВАННЯ КОМП'ЮТЕРНИХ СИСТЕМ НА ОСНОВІ МІКРОСХЕМ ПРОГРАМОВАНОЇ ЛОГІКИ

**Монографія**

| | |
|---|---|
| Редактор | В. В. Казимир |
| Комп'ютерне верстання | С. А. Іванця |
| Художнє оформлення обкладинки | С. Ю. Цегельникова |

Формат 60x84/16. Ум. друк. арк. 18,14. Обл.-вид. арк. 16,01. Тираж 300 пр. Замовлення №

Видавець і виготовлювач
Сумський державний університет,
вул. Римського-Корсакова, 2, м. Суми, 40007
Свідоцтво суб'єкта видавничої справи ДК № 3062 від 17.12.2007.