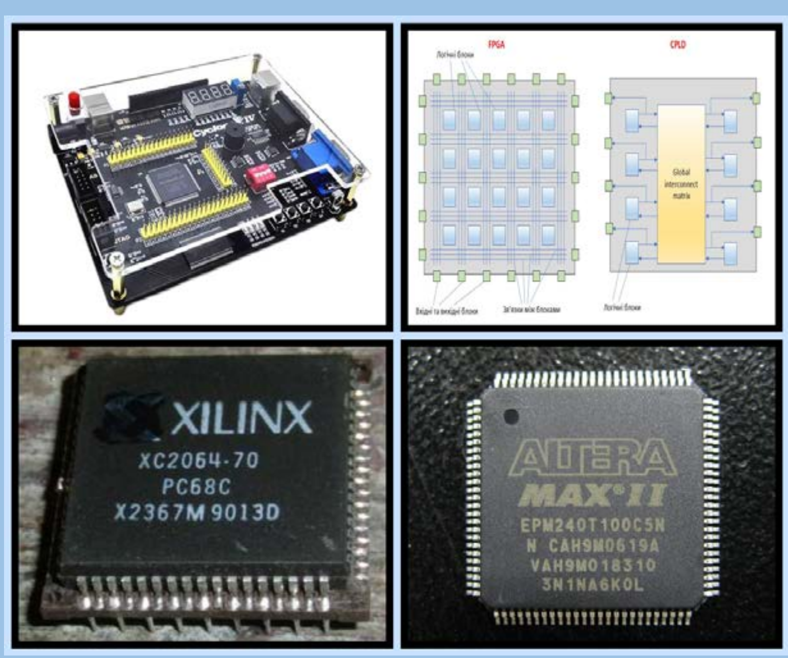
М. О. Притула, І. А. Дудатьєв, Я. О. Осадчук

ПРОГРАМОВАНІ ЛОГІЧНІ ІНТЕГРАЛЬНІ СХЕМИ ТА ЇХ ПРОГРАМУВАННЯ



Міністерство освіти і науки України
Вінницький національний технічний університет

М. О. Притула, І. А. Дудатьєв, Я. О. Осадчук

ПРОГРАМОВАНІ ЛОГІЧНІ ІНТЕГРАЛЬНІ СХЕМИ ТА ЇХ ПРОГРАМУВАННЯ

Електронний навчальний посібник

Вінниця
ВНТУ
2026

УДК 621.382.333
П75

Рекомендовано до видання Вченою Радою Вінницького національного технічного університету Міністерства освіти і науки України (протокол № 12 від 30.04.2026 р.)

Рецензенти:
С. К. Підченко, доктор технічних наук, професор
Д. В. Михалевський, доктор технічних наук, професор
А. П. Саміла, доктор технічних наук, професор

Притула, М. О.

П75 Програмовані логічні інтегральні схеми та їх програмування: навчальний посібник [Електронний ресурс] / Притула М. О., Дудатьєв І. А., Осадчук Я. О. – Вінниця : ВНТУ, 2026. – (PDF, 176 с.)

ISBN 978-617-8163-89-1 (PDF)

Посібник присвячений матеріалам курсу з дисципліни «Програмовані логічні інтегральні схеми в РЕА» для здобувачів вищої освіти, що навчаються за освітньо-професійною програмою «Радіотехніка» денної форми навчання.

Мета посібника – надати здобувачам можливість більш детально вивчити аудиторний матеріал і підготуватися до іспиту, а також застосувати отримані знання для подальшої фахової роботи.

Перелік та зміст тем відповідає програмі вказаної вище дисципліни.

УДК 621.382.333

ISBN 978-617-8163-89-1 (PDF)

© ВНТУ, 2026

Зміст

1 ПРОГРАМОВАНІ ЛОГІЧНІ ІНТЕГРАЛЬНІ СХЕМИ........................... 5
1.1 Історія виникнення та розвитку ПЛІС 5
1.2 Технології програмування апаратних засобів 7
1.3 Програмовані схеми ПЛМ, PAL, GAL... 20
1.4 Класифікація спеціалізованих замовних ВІС (ASIC)......................... 25
1.5 Архітектура базисних модулів структурованих ASIC 26
1.6 Технології проектування ПЛІС.. 29
1.6.1 Базисний модуль ПЛІС.. 29
1.6.2 Архітектура базисного модуля ПЛІС. Реалізація на комірках ОЗП та на мультиплексорах 33
1.6.3 Архітектура базисного модуля, що конфігурується (логічна комірка фірми Xilinx) 38
1.6.4 Конфігурація логічних блоків ПЛІС (комірки, секції, логічні масиви, функціональні модулі) 39
1.7 Методи та засоби програмування архітектури ПЛІС......................... 50
1.8 Тенденції розвитку сучасних ПЛІС.. 59
1.9 Архітектура ПЛІС компаній Altera та Xilinx.. 61
1.10 Контрольні питання до розділу 1 68
2 МОВА ПРОГРАМУВАННЯ VERILOG 71
2.1 Основи мови програмування Verilog 71
2.1.1 Лексичні правила 71
2.1.2 Вирази та операнди.. 71
2.1.3 Оператори 72
2.1.4 Ідентифікатори та ключові слова 80
2.1.5 Типи даних.. 82
2.1.6 Системні задачі та директиви компілятора 90
2.2 Модулі та порти.. 94
2.2.1 Модулі 94
2.2.2 Порти 97
2.2.3 Ієрархічні імена у Verilog.. 103
2.3 Поведінкове моделювання 104
2.3.1 Структуровані процедури 104
2.3.2 Процедурні присвоєння... 107
2.3.3 Керування часом... 111
2.3.4 Умовні оператори.. 117
2.3.5 Багатоваріантне розгалуження 118
2.3.6 Цикли.. 121
2.3.7 Послідовні та паралельні блоки 125
2.3.8 Generate-оператори 128
2.4 Таски та функції 135
2.4.1 Відмінності між Tasks і Functions.. 135
2.4.2 Таски 136

2.4.3 Функції 142
2.5 Контрольні питання до розділу 2 150
3 ПРОГРАМУВАННЯ ПЛІС НА ПРАКТИЦІ 153
3.1 Опис та склад Altera-Cyclone-IV-board-V3.0....................................... 153
3.2 Встановлення драйвера USB-Bluster.. 160
3.3 Приклад створення логічного елементу 4І 164
ПЕРЕЛІК РЕКОМЕНДОВАНИХ ДЖЕРЕЛ 173

1 ПРОГРАМОВАНІ ЛОГІЧНІ ІНТЕГРАЛЬНІ СХЕМИ

1.1 Історія виникнення та розвитку ПЛІС

Історія ПЛІС (програмованих логічних інтегральних схем) містить певні періоди розвитку: від перших програмованих логічних матриць до сучасних гетерогенних платформ для штучного інтелекту (ШІ) та 5G [1].

Витоки: 1970-ті — початок 1980-х

- PLA/PROM/PAL. Перед появою FPGA інженери користувалися програмованими масивами логіки (PLA), постійною пам’яттю (PROM) та PAL/ GAL. Вони дозволяли реалізовувати фіксовану комбінаційну логіку й прості автомати, але гнучкість та масштабованість були обмежені.
- CPLD-підхід. Далі з’явилися CPLD – більш «жорсткі» структури з макроосередками і прогнозованими затримками; чудово підходили для «клею» між мікросхемами, проте гірше масштабувалися для великих дизайнів.

Народження FPGA: середина 1980-х

- 1984–1985. Заснування Xilinx і вихід першої комерційної FPGA XC2064 на SRAM-конфігурації з таблицями істинності (LUT) і програмованою міжз’єднувальною матрицею. Ключова ідея — багаторазово перезаписувана конфігурація «в полі».
- Антиф’юз як альтернатива. Наприкінці 1980-х Actel запровадила антиф’юзні FPGA – одноразово програмовані, з дуже низькими витратами на маршрути й високою надійністю (актуально для космосу/безпеки).

Розквіт і спеціалізація: 1990-ті

- Більші LUT і швидкі перенесення. Зростає розмір LUT, з’являються «carry chains» для прискорення арифметики.
- Block RAM і PLL. У корпус FPGA інтегрують блочну пам’ять та тактові ресурси; дизайни стають складнішими й швидшими.
- Гігабітні трансивери. Кінець 90-х – початок 2000-х: у топових сімействах з’являються вбудовані SERDES для PCIe, Ethernet, високошвидкісних посилань [2].

Системний рівень: 2000-ні

- DSP-блоки. Стандартом стає наявність апаратних множників/ MAC для сигнал-процесінгу.
- Soft-CPU. Поширюються «м’які» процесори (Nios/Nios II, MicroBlaze) і дедалі більше готових IP (контролери пам’яті, інтерфейси).
- Flash/SRAM/антиф’юз. На ринку співіснують три основні технології конфігурації: SRAM (гнучкість), flash (енергонезалежність, простота), антиф’юз (надійність, одноразове програмування).

- Інструменти. Розвиваються P&R-алгоритми, часткова реконфігурація, верифікація; стандарт JTAG де-факто всюдисущий для програмування й тесту.

SoC-FPGA та HLS: 2010-ті

- ARM у кристалі. Покоління SoC-FPGA (напр., Xilinx Zynq-7000; у Intel/Altera – Cyclone V/Arria V SoC) об’єднує програмований логічний масив і ядра ARM з периферією – один крок до гетерогенних систем.
- 3D та FinFET. Перехід на 28/20/16 нм, 2.5D-інтеграція, багатокристальні рішення, HBM – усе це масштабно піднімає пропускну здатність і щільність.
- HLS та OpenCL. З’являються масово придатні високорівневі інструменти (HLS, OpenCL/oneAPI), що знижують поріг входу для розробників алгоритмів.
- Консолідація ринку. Altera входить до складу Intel (2015); Actel раніше придбала Microsemi (2010), пізніше Microsemi – Microchip; формується чітка «вища ліга» постачальників [3].

Гетерогенність і ШІ: кінець 2010-х — 2020-ті

- ACAP/Versal, Agilex, Speedster. Нові сімейства фокусуються на гетерогенних обчисленнях: поєднання логіки LUT, DSP, прискорювачів матричних обчислень, інколи – векторних процесорів і внутрішніх NoC (мереж на кристалі) для передбачуваної доставки даних.
- Плаваюча кома та bfloat16. Апаратура для FP16/bfloat16/FP32 стає типовою для задач ШІ та 5G-фізики; множники/акумулятори підтримують змінну точність.
- Інтеграція пам’яті. HBM/LPDDR4/5 на платі чи в корпусі закриває «голод» за пропускною здатністю, що був вузьким місцем ML/відео.
- Безпека і довіра. Шифрування бітстріму, корені довіри, PUF, контроль ланцюга постачання – вбудована вимога для телекомів та дата-центрів.
- Відкриті інструменти. Проєкти на кшталт Yosys/nextpnr та відкриті бітстріми для окремих сімейств (наприклад, iCE40) сприяють академії й мейкерству.

Затребуваність FPGA визначають такі характеристики:

- Гнучкість і час на ринок. Можна швидко оновити «залізо» прошивкою без переробки кремнію – цінно для стандартів, що еволюціонують (PCIe, Ethernet, 5G NR).
- Продуктивність на ват. Для вузькоспеціалізованих задач (фільтрація, кодування/декодування, інференс) FPGA часто дають кращу енергоефективність і латентність, ніж CPU/GPU.
- Кастомізація. Мікроархітектура «під задачу»: від ширини шин до схеми пам’яті, від конвеєрів до рівня паралельності.

Таким чином, можна виділити ключові технологічні періоди

- 1985: перша комерційна FPGA на SRAM.
- 1990-ті: block RAM, швидкі перенесення, перші вбудовані множники.
- ~2000: гігабітні трансивери, DSP-блоки, soft-CPU.
- 2010+: SoC-FPGA (ARM + логіка), 2.5D/3D інтеграція, HLS.
- Сьогодні: NoC усередині кристала, FP16/bfloat16, HBM, прискорювачі ШІ, індустріальний фокус на 5G/датах-центрах/edge.

Далі галузь рухатиметься до ще більшої гетерогенності (chiplet-и, eFPGA в ASIC/SoC), тіснішої інтеграції з пам'яттю та мережами, а також до розумніших САПР, які автоматично переносять алгоритми з високого рівня (Python/C++/ML-фреймворки) на оптимізовані апаратні потоки. Водночас безпека, вартість розробки й доступність компетенцій залишатимуться визначальними факторами успіху [4].

Отже, ПЛІС пройшли шлях від простих програмованих логік до універсальних гетерогенних платформ, які стоять у центрі сучасних телекомів, обчислень і штучного інтелекту – і їхня роль лише зростатиме.

1.2 Технології програмування апаратних засобів

Програмовані логічні інтегральні схеми (ПЛІС), або FPGA (field-programmable gate array), – це цифрові ІС, побудовані з набору конфігурованих логічних блоків і програмованих міжз'єднань між ними. Завдяки змінюваній конфігурації інженери можуть реалізовувати на цих пристроях широкий спектр функцій і рішень.

За технологією виготовлення ПЛІС бувають як одноразово програмовані, так і такі, що допускають багаторазове перепрограмування.

У цифровій електроніці існує чимало класів мікросхем: від «розсипної логіки» – невеликих компонентів із кількома фіксованими логічними функціями – до пам'яті та мікропроцесорів. У цьому контексті цікавими є програмовані логічні пристрої (ПЛП), спеціалізовані замовні інтегральні схеми ASIC (application-specific integrated circuit), стандартні вироби для певних застосувань ASSP (application-specific standard parts) та ПЛІС. Під ПЛП зазвичай мають на увазі два підтипи: прості ПЛП (SPLD) і складні ПЛП (CPLD) [5].

Архітектура програмованих логічних пристроїв (ПЛП) задається виробником так, щоб їх можна було перепрограмовувати безпосередньо на місці експлуатації для виконання різних функцій. Порівняно з ПЛІС, такі мікросхеми містять менше логічних вентилів і зазвичай застосовуються для невеликих, відносно простих завдань.

Водночас існують ASIC і ASSP, у яких кількість вентилів сягає сотень мільйонів, що дозволяє виконувати надзвичайно складні й масштабні функції. Обидва класи проєктуються на спільних принципах і

виготовляються за подібними технологіями. Різниця в призначенні: ASIC створюють під конкретне замовлення певної компанії, тоді як ASSP пропонують як готові вироби для широкого ринку.

Попри високу інтеграцію, продуктивність і здатність розв’язувати складні задачі, розроблення та виробництво замовних мікросхем триває довго і коштує дорого. Після випуску кристала функціональність фактично «застигає в кремнії», тож будь-яка зміна потребує нової версії виробу.

ПЛІС посідають проміжну нішу між ПЛП і замовними інтегральними схемами. З одного боку, їхню логіку можна оперативно конфігурувати відповідно до вимог користувача; з іншого – сучасні ПЛІС уміщують мільйони вентилів і здатні реалізовувати дуже великі та складні проєкти, які раніше були під силу лише ASIC [6].

Щодо вартості, ПЛІС зазвичай дешевші за ASIC на етапі розробки та за невеликих обсягів, хоча в масовому виробництві кінцева ціна ASIC часто нижча. Крім того, у ПЛІС легше вносити зміни й швидше виводити продукт на ринок. Це робить їх привабливими як для великих компаній, так і для невеликих інженерних команд.

Нині ключовими ринковими нішами ПЛІС є [7]:

* Замовні ІС (ASIC). Сучасні ПЛІС дозволяють створювати пристрої того самого класу складності, який раніше був доступний лише «замовним» мікросхемам.

* Системи цифрової обробки сигналів. Раніше високошвидкісну обробку виконували спеціалізовані процесори DSP. Тепер у ПЛІС є вбудовані помножувачі, швидкі ланцюжки перенесення та значні обсяги внутрішньокристальної пам’яті; разом із притаманним паралелізмом це дає приріст швидкодії, що може перевищувати показники найшвидших DSP у 500+ разів.

* Вбудовані мікроконтролерні системи. Типові задачі керування вирішують недорогі мікроконтролери з прошивкою, ПЗП команд, таймерами та інтерфейсами введення/виведення на одному кристалі. Натомість навіть прості ПЛІС здатні вмістити «м’яке» (програмне) процесорне ядро з потрібними периферійними блоками, тож їх дедалі частіше використовують як платформу для реалізації функцій мікроконтролерів.

* Мікросхеми фізичного рівня передавання даних. ПЛІС традиційно виконують роль «клейової» логіки між чипами та вищими рівнями протоколів. Завдяки наявності багатьох високошвидкісних передавачів сучасні ПЛІС дозволяють умістити мережеві й комунікаційні функції в одному пристрої.

* Системи обчислень з переналаштованою архітектурою (reconfigurable computing, RC). Паралельність і можливість оперативного перенастроювання ПЛІС дають «апаратне прискорення» програмних алгоритмів. На їх основі розробляють великі перенастроювані

обчислювальні системи для дуже різних задач – від моделювання апаратури до криптоаналізу й пошуку нових лікарських засобів.

Для роботи з такими пристроями потрібен механізм конфігурування, який дозволяє запрограмувати підготовлений кремнієвий кристал. [8]

На рис. 1.1 подано приклад елементарної програмованої логічної функції з двома входами (a та b) та одним виходом. На входах встановлено інвертори (операція «НЕ»), що дозволяє подавати сигнал як у прямому, так і в інверсному вигляді.

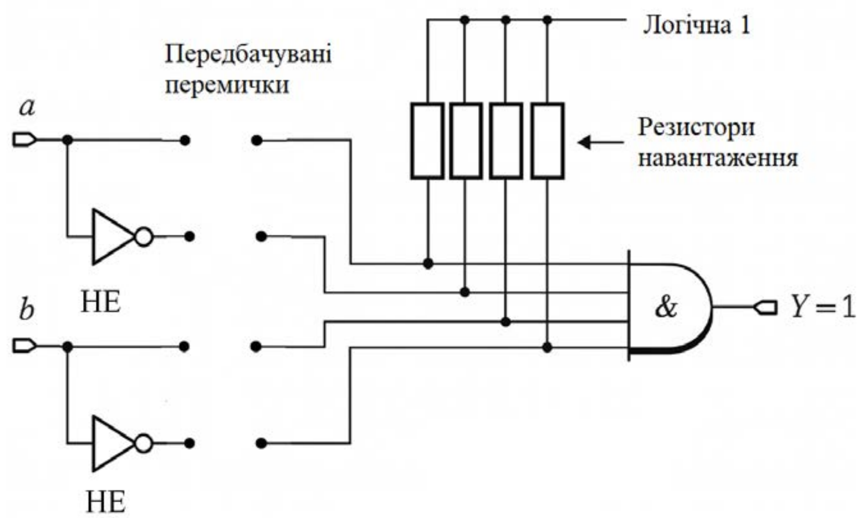


Рисунок 1.1 – Проста функція, що програмується

Варто звернути увагу на місце розташування передбачених перемичок. Якщо жодна з них не встановлена, то через навантажувальні резистори на всі входи елемента «І» надходить логічна 1. У такому випадку вихід пристрою постійно перебуватиме в стані логічної 1. Для реалізації програмованої функції необхідно мати механізм, який забезпечує можливість встановлення однієї або кількох перемичок [9].

Метод плавких перемичок був першим способом, що надав користувачам змогу створювати власні конфігурації пристрої. У цьому випадку схема спочатку виготовляється з усіма можливими з’єднаннями, кожне з яких реалізоване у вигляді плавкої перемички (рис. 1.2).

Плавкі перемички працюють подібно до звичайних запобіжників. Наприклад, якщо з певних причин пристрій починає споживати надмірну потужність, запобіжник перегорає, внаслідок чого відбувається розрив електричного кола, що запобігає пошкодженню пристрою. Плавкі перемички мають дуже малі розміри, адже виготовляються безпосередньо на кристалі кремнію з використанням тих самих технологічних процесів, що й під час створення транзисторів або провідних доріжок.

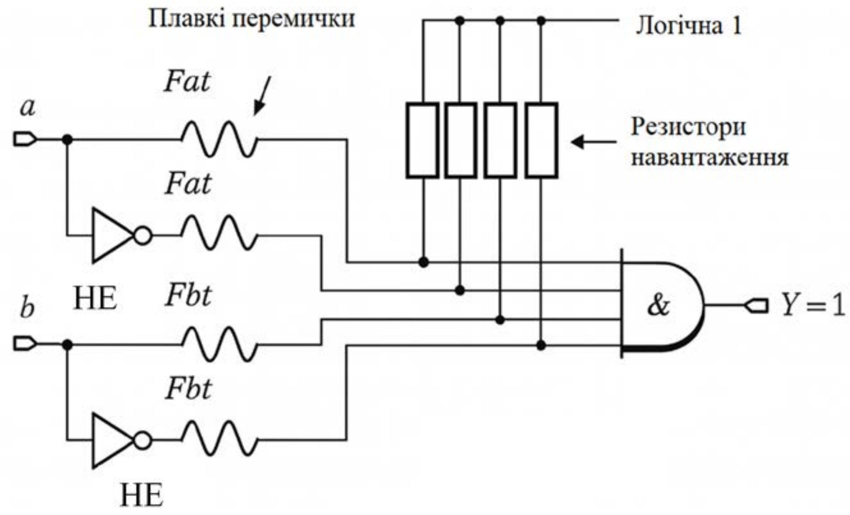


Рисунок 1.2 – Заповнення плавкими перемичками

На початковому етапі у програмованих пристроях усі перемички залишаються неушкодженими. Це означає, що в незапрограмованому стані вихід розглянутої функції завжди дорівнює логічному нулю. Достатньо наявності нуля хоча б на одному з входів елемента «І», щоб і вихід залишався на рівні 0. Наприклад, якщо на вхід a подано 0, то вихід також дорівнюватиме 0. У випадку, коли на вхід a подається 1, то відповідний інверсний вихід $\bar{a}$ все одно встановлюється у стан 0, що знову ж таки дає 0 на виході елемента «І». Аналогічна ситуація відбувається і для входу b.

Таким чином, під час програмування можна цілеспрямовано «спалювати» непотрібні перемички, прикладаючи до них короткочасні імпульси підвищеної напруги і струму. Наприклад, якщо вилучити перемички *Fat* і *Fbt*, схема змінить свою логіку роботи відповідно до заданої конфігурації [10].

Після вилучення зазначених перемичок інвертований вхід a і прямий вхід b від’єднуються від елемента «І»; завдяки резисторам підтягування на цих входах фіксується рівень логічної «1». У такій конфігурації пристрій починає реалізовувати нову логічну функцію: a & not b. Процес вибіркового «перегорання» плавких перемичок зазвичай і називають програмуванням пристрою; також уживають формулювання «пропалювання перемичок» або «пропалювання мікросхеми».

Пристрої, у яких конфігурація задається плавкими перемичками, належать до одноразово програмованих (OTP, one-time programmable): після пропалювання така перемичка не підлягає відновленню й повернути її у початковий стан неможливо. Способи програмування ПЛІС можуть відрізнятися, проте метод плавких перемичок у сучасних мікросхемах практично не застосовують; його наводять як найпростіший ілюстративний приклад [11].

Метод нарощуваних перемичок є протилежним до плавких. Тут кожне потенційне з’єднання має «нарощувану» перемичку: у

незапрограмованому стані її опір настільки великий, що еквівалентний розімкненому колу, як було схематично показано на рис. 1.1. Нарощувані перемички можна вибірково формувати, подаючи на потрібні виводи короткі імпульси підвищеної напруги та струму. Якщо створити такі перемички на шляху інвертованого входу *a* та прямого входу *b*, схема почне реалізовувати функцію *Y = not a & b* (рис. 1.3).

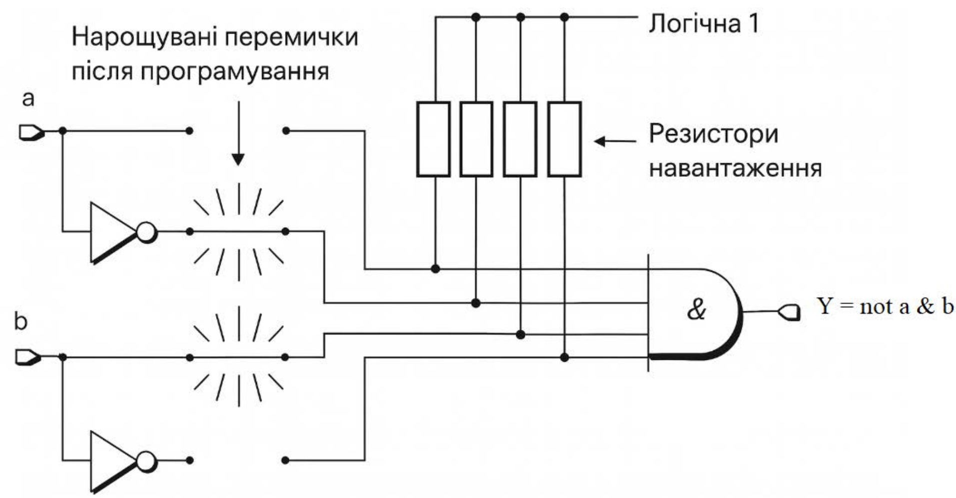
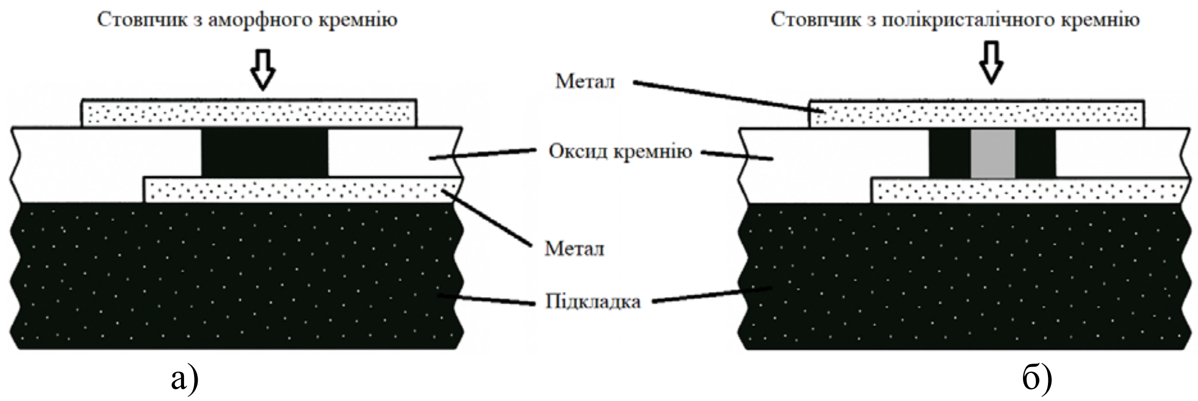


Рисунок 1.3 – Нарощувані перемички після програмування

У початковому, «незапрограмованому» стані нарощувана перемичка – це мікроскопічний стовпчик аморфного (некристалічного) кремнію між двома металевими провідниками. Аморфний кремній працює як діелектрик із опором до мільярда Ом (рис. 1.4, а) [12].

Програмування окремого з'єднання передбачає вирощування провідного містка: під дією імпульсу аморфний кремній локально перетворюється на полікристалічний, який проводить струм, утворюючи постійний електричний контакт (рис. 1.4, б).



а) б)

Рисунок 1.4 – Вирощування перемичок:
а – до програмування; б – після програмування

Пристрої, у яких конфігурацію формують методом нарощуваних перемичок, належать до одноразово програмованих: створений місток не підлягає руйнуванню чи поверненню в початковий високоомний стан. В електронних системах, зокрема в комп'ютерах, використовують два базові типи пам'яті: постійну (ПЗП) та оперативну (ОЗП). ПЗП є енергонезалежною – її вміст зберігається навіть без живлення; інші вузли можуть лише зчитувати дані, але не записувати нові. ОЗП, навпаки, допускає і запис, і читання, однак є енергозалежною: після вимкнення живлення вся інформація втрачається.

Типові мікросхеми ПЗП також називають масково-програмованими, оскільки вміст «вшивають» під час виробництва за допомогою фотошаблонів. Ці фотошаблони задають топологію транзисторів і металевих з'єднань (шарів металізації) [6].

Як приклад розглянемо бітову комірку ПЗП (рис. 1.5). Звичайний ПЗП являє собою матрицю рядків (адресні лінії) і стовпців (лінії даних). Кожен стовпець має резистор навантаження, що утримує на виході рівень логічної «1». У вузлах перетину рядків і стовпців розміщено транзистори та, за потреби, контактну перемичку. Наявність або відсутність цієї перемички визначається фотошаблоном і кодує збережений біт.

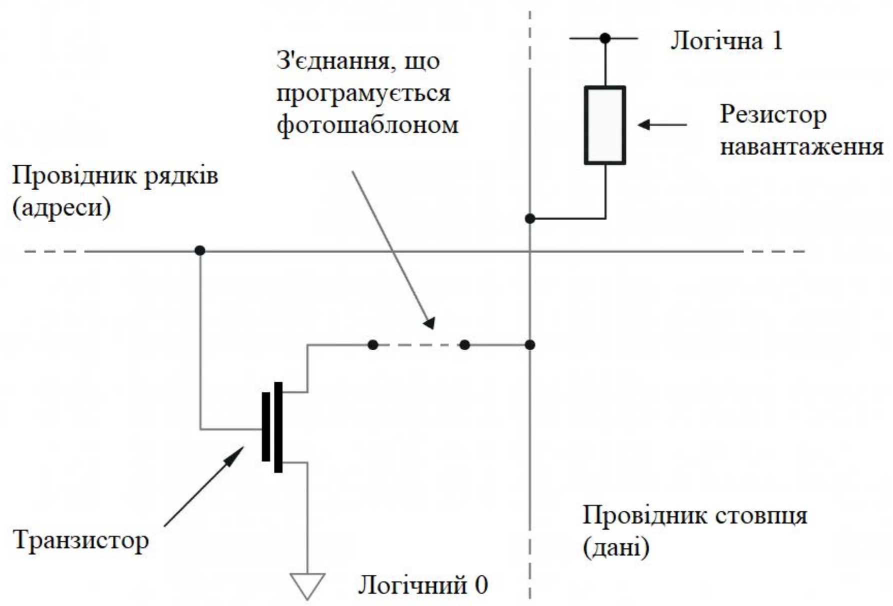


Рисунок 1.5 – Транзисторна комірка ПЗП, що програмується фотошаблоном

Більшість мікросхем ПЗП спершу випускають у масовій, типово налаштованій конфігурації. Коли ж потрібен особливий вміст, виготовляють індивідуальний фотошаблон, який задає, у яких комірках буде зроблено контакт, а де його не передбачено [8].

Подання активного рівня на лінію рядка переводить пов'язані з нею транзистори у провідний стан. Якщо в конкретній комірці маскою

закладено з’єднання, «відкритий» транзистор підтягує лінію стовпця до логічного 0, тим самим формуючи нуль на виході. Якщо ж контакту не закладено, транзистор не впливає на стовпець, і завдяки навантажувальному резистору на виході зберігається логічна 1.

Недолік масково-програмованих рішень у тому, що їх виготовлення економічно виправдане лише для дуже великих тиражів; на етапах розробки, коли склад даних часто змінюється, такі мікросхеми майже не використовують [13].

Перші програмовані постійні запам’ятовувальні пристрої (ППЗП, PROM) створила компанія Harris Semiconductor. У них застосовували ніхромові плавкі перемички. Спрощений приклад бітової комірки ППЗП на транзисторі з плавкою перемичкою показано на рис. 1.6.

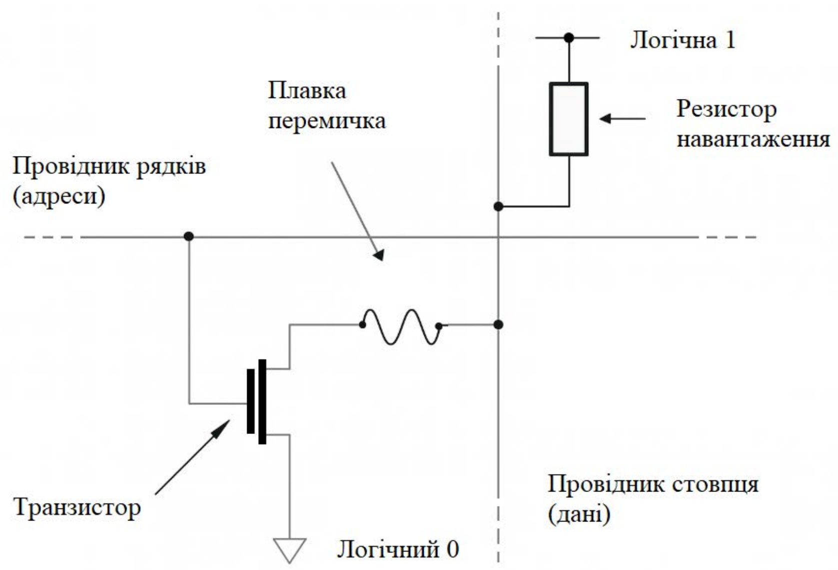


Рисунок 1.6 – Транзисторна комірка ПЗП, що програмується фотошаблоном

Мікросхема, що постачається виробником, є «чистою»: усі передбачені конструкцією плавкі перемички присутні, але нічого ще не запрограмовано. Якщо активувати певний рядок, відкриються всі транзистори, під’єднані до нього, і відповідні стовпці через ці транзистори опиняться на рівні логічного 0. Інженер може вибірково пропалювати зайві перемички, подаючи на виводи короткі імпульси підвищеної напруги та струму. Після видалення перемички вихід комірки фіксується на логічній «1». Раніше такі мікросхеми призначалися як носії постійних даних – для зберігання програм і констант (звідси назва ПЗП). Згодом їх почали використовувати і для реалізації простих логічних функцій – таблиць відповідності та кінцевих автоматів.

З часом з’явилися ПЛП загального призначення, що ґрунтувалися на технологіях пропалюваних і нарощуваних перемичок. Однак обидва підходи дають лише одноразове програмування: після пропалювання або

«нарощення» змінити налаштування майже неможливо, а будь-які доробки потребують часу; іноді можна задіяти ще невикористані перемички, але це трапляється рідко. Це підштовхнуло до створення пристроїв, які можна не лише запрограмувати, а й стерти та запрограмувати повторно.

Одним із перших рішень стали ППЗП зі стиранням (СППЗП). Транзистор СППЗП має структуру, подібну до звичайного МОП, але доповнену «плаваючим» полікремнієвим затвором, ізольованим шарами діоксиду кремнію (рис. 1.7) [14].

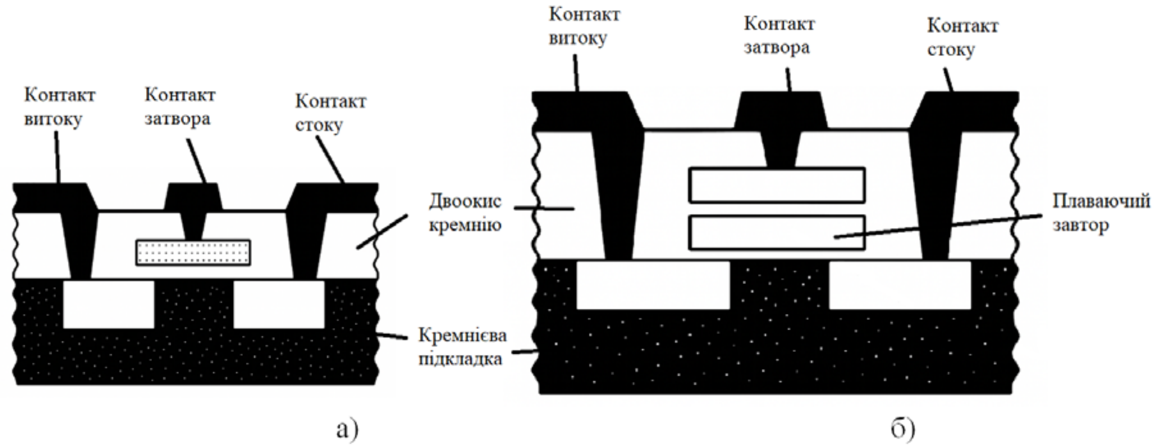


Рисунок 1.7 – Порівняння МОП та СППЗП транзисторів:
а – МОП-транзистор; б – СППЗП-транзистор

У незапрограмованій комірці плаваючий затвор не має заряду й не впливає на роботу керувального затвора. Щоб виконати програмування, до затвора та стоку МОП-транзистора подають підвищену напругу – орієнтовно 12 В. За цих умов транзистор переходить у режим інтенсивної провідності, і частина носіїв набуває високої енергії: «гарячі» електрони долають бар'єр діоксиду кремнію та потрапляють у плаваючий затвор (процес інжекції гарячих/високоенергетичних електронів).

Після припинення програмувального імпульсу негативний заряд залишається захопленим у плаваючому затворі. За дотримання умов експлуатації він зберігається стабільно протягом понад десяти років. Накопичений заряд зсуває поріг спрацьовування й фактично блокує нормальне керування каналом звичайним затвором, що й дозволяє розрізняти запрограмовані та стерті комірки. Завдяки цьому ефекту такі транзистори слугують основою елементів пам'яті (рис. 1.8) [15].

Елемент пам'яті цього типу не потребує ані плавких, ані нарощуваних перемичок, ані масково запрограмованих з'єднань. На етапі постачання всі плаваючі затвори в СППЗП-транзисторах не заряджені. Тому, коли активується певний рядок, «відкриваються» всі транзистори, під'єднані до нього, і відповідні стовпці через ці транзистори скидають сигнал до логічного нуля. Програмування здійснюють подачею імпульсів

на входи для заряджання плаваючих затворів вибраних транзисторів – це блокує їх провідність, і в таких комірках фіксується логічна «1».

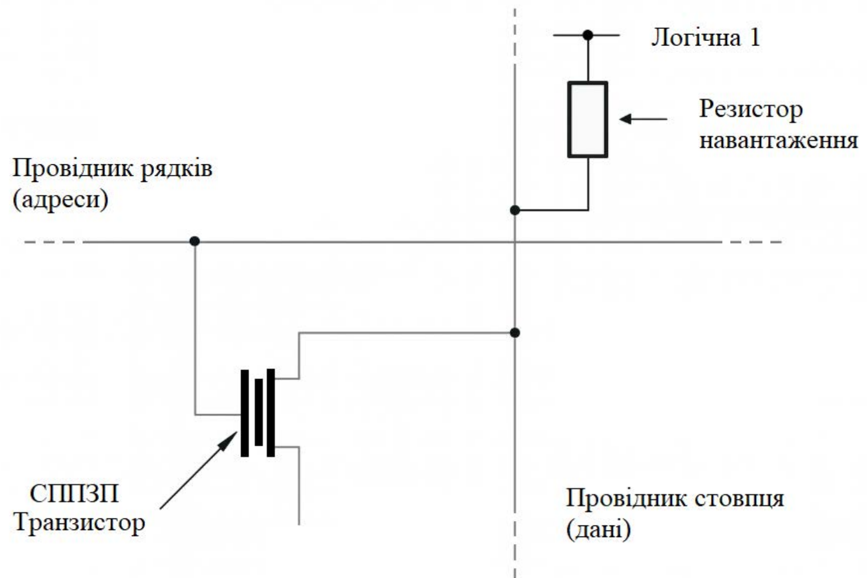


Рисунок 1.8 – Комірка пам'яті на основі СППЗП-транзистора

З погляду розміщення на кристалі СППЗП-комірки значно компактніші за аналоги на плавких перемичках, однак їх головна перевага – можливість стирання та повторного запису. Стирання полягає у виведенні електронів із плаваючого затвора; потрібну енергію забезпечує ультрафіолет. Такі мікросхеми випускають у керамічних або пластмасових корпусах із маленьким кварцовим «віконцем» зверху, яке зазвичай заклеюють непрозорою стрічкою. Щоб виконати стирання, мікросхему знімають з плати, відкривають віконце й поміщають у закритий контейнер з інтенсивним УФ-опроміненням.

До недоліків СППЗП належать дорожчий корпус із кварцовим вікном і тривалість стирання – приблизно 20 хвилин. Із подальшим зменшенням геометрії транзисторів частка металізації на поверхні кристала зростає, що ускладнює проникнення ультрафіолету до комірок і змушує збільшувати час експозиції.

Спершу ці вироби слугували саме програмованими постійними запам’ятовувальними пристроями (ППЗП), що відображено в їхній назві. Згодом ту саму технологію інтегрували в універсальніші ПЛП – так з’явилися ПЛП із можливістю стирання. Наступним етапом еволюції стали ППЗП з електричним стиранням (ЕСППЗП, EEPROM) [16].

Комірка ЕСППЗП має більшу площу – орієнтовно у 2,5 раза понад розмір аналогічної комірки СППЗП, оскільки формується двома рознесеними транзисторами. ЕСППЗП-транзистор містить плаваючий затвор, оточений надтонким шаром діоксиду кремнію, а другий транзистор використовується для електричного стирання комірки. Спочатку ЕСППЗП застосовували як комп’ютерну пам’ять. Згодом цю ж технологію

інтегрували в ПЛП, утворивши ПЛП з електричним стиранням (ЕСПЛП) – фактично різновид програмованих логічних інтегральних схем.

Технологія Flash виросла з підходів, використаних у СППЗП та ЕСППЗП. Назву «Flash» запровадили, аби підкреслити радикально коротший час стирання порівняно зі СППЗП. Такі компоненти реалізують у різних архітектурах. В одних комірки виконані на одному транзисторі з плаваючим затвором (площа близька до СППЗП), але з набагато тоншим оксидом – їх можна стирати електрично, здебільшого цілими кристалами або великими блоками. В інших – застосовано двотранзисторні комірки, подібні до ЕСППЗП, що дозволяє стирати чи перезаписувати інформацію на рівні слів.

Перші Flash-пам’яті зберігали один біт на комірку. До початку 2002 року інженери продемонстрували способи підвищення щільності: або за рахунок розрізнення кількох рівнів заряду у плаваючому затворі (MLC), або створивши два окремі запам’ятовувальні вузли під затвором.

Оперативна напівпровідникова пам’ять буває двох головних типів: динамічна (DRAM) і статична (SRAM). У DRAM кожен біт реалізовано парою «транзистор–конденсатор», що займає мінімальну площу на кристалі. «Динамічна» – бо конденсатор поступово розряджається, тож дані потрібно періодично відновлювати (регенерація), що потребує складних допоміжних схем. Це виправдано лише за великих місткостей – десятки мільйонів бітів на кристал – але для програмованої логіки DRAM зазвичай не є пріоритетною. Натомість у статичній пам’яті (SRAM) записане значення зберігається без регенерації, доки його не змінять або не вимкнуть живлення.

Комірка містить багатотранзисторний вузол SRAM, вихід якого керує додатковим транзистором. Залежно від записаного біта (0 чи 1) цей керувальний транзистор буде вимкнений (закритий) або увімкнений (відкритий) [17].

Серед мінусів пристроїв, що використовують SRAM-комірки: велика площа на кристалі, адже кожна комірка – це 4–6 транзисторів, зібраних у схему тригера/засувки. Другий недолік – втрата конфігураційних даних після знеструмлення системи.

Перелік типів програмованих пристроїв і відповідних технологій програмування наведено в табл. 1.1.

Необхідно враховувати, що з’являються нові підходи до пам’яті. Одна з них – магнітна оперативна пам’ять (MRAM, Magnetic RAM). Її розвиток стартував із створення компанією IBM магнітного тунельного переходу – тришарової структури з двох феромагнітних плівок, розділених тонким ізоляційним шаром. Осередки MRAM формують у точках перетину ліній рядків і стовпців, між якими розміщують такий тунельний перехід. MRAM поєднує швидкодію, притаманну SRAM, масштабовану місткість на кшталт DRAM, енергонезалежність, як у Flash, і водночас дуже низьке споживання енергії.

Таблиця 1.1 – Технології програмування

Технології	Умовне позначення	Галузь використання
Плавкі перемички		Прості ПЛП
Нарощувані перемички		ПЛІС
СППЗП		Прості та складні ПЛП
ЕСППЗП та Flash		Прості та складні ПЛП (деякі ПЛІС)
Статичний ОЗП	SRAM	ПЛІС (деякі складні ПЛП)

Схематичну класифікацію архітектурних особливостей ПЛП подано на рис. 1.9. Перші прості ПЛП фактично ґрунтувалися на ППЗП-мікросхемах. Їх зручно уявляти як фіксований масив елементів «І», виходи якого під’єднані до програмованого масиву елементів «АБО».

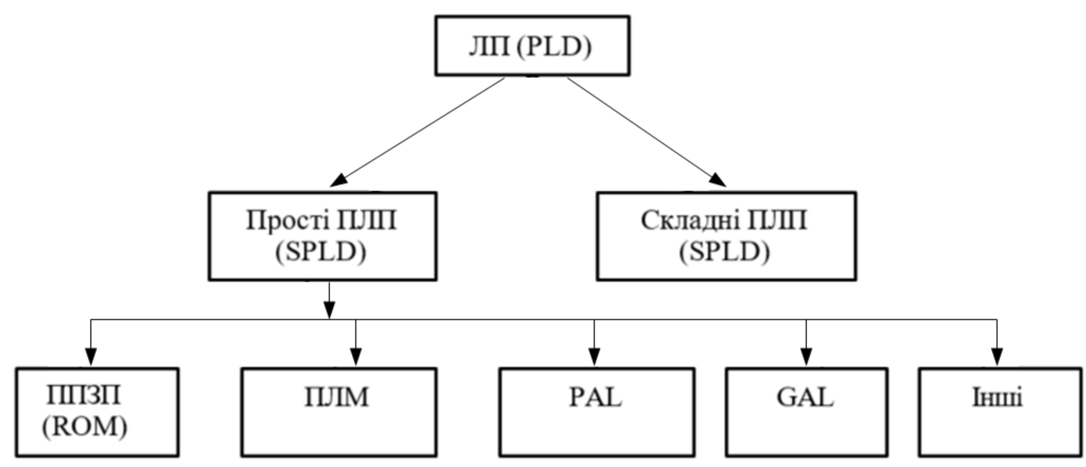


Рисунок 1.9 – Класифікація архітектури побудови ПЛП

Для прикладу розглянемо ППЗП із трьома входами та трьома виходами (рис. 1.10).

У ролі програмованих з’єднань у блоці «АБО» можуть використовуватися плавкі перемички, а також елементи на базі СППЗП чи комірок ЕСППЗП — відповідно для відповідних технологій. Схема на рис. 1.10 показує лише принцип роботи й не є реальною топологією: на практиці кожен елемент «І» має три входи, що можуть підключатися до прямих або інверсних сигналів a, b, c. Аналогічно, кожен елемент «АБО» має вісім входів, пов’язаних із виходами «І»-масиву.

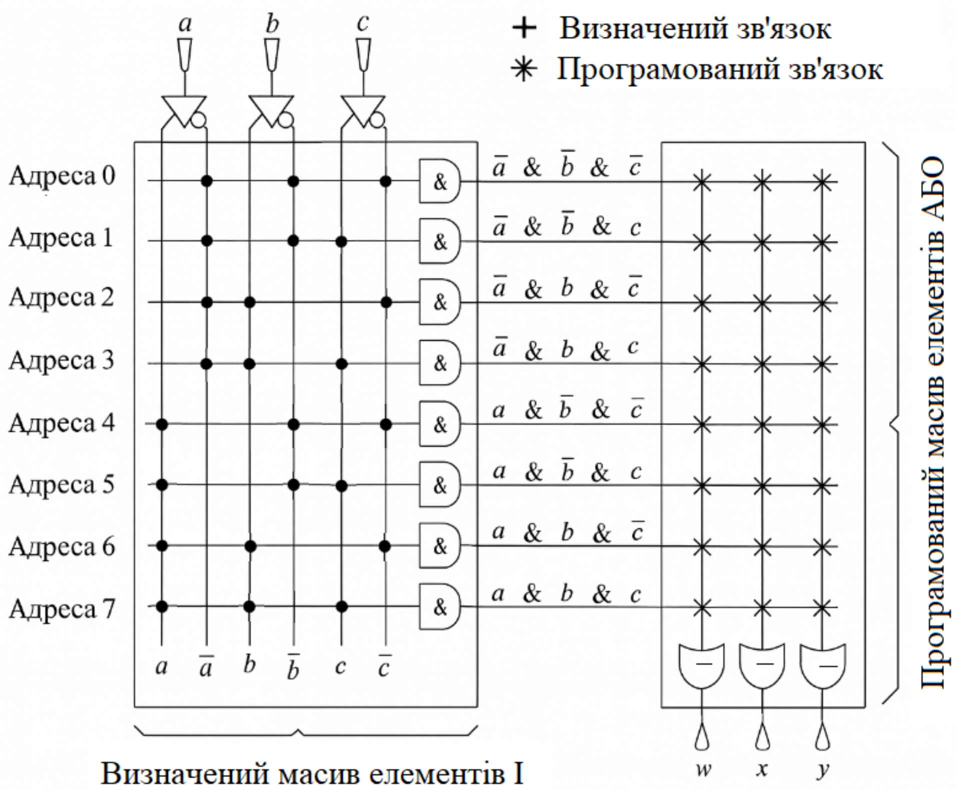
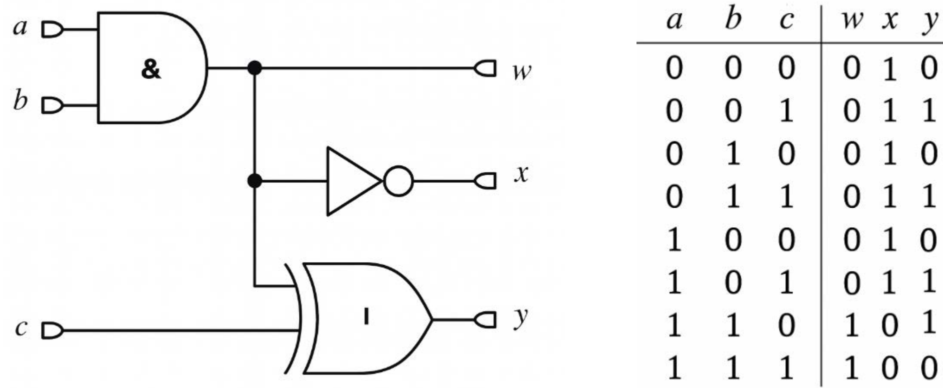


Рисунок 1.10 – Незапрограмована мікросхема ППЗП

Первісне призначення ППЗП – зберігання програм і констант, тобто функції пам’яті. Водночас їх часто застосовують для реалізації нескладної логіки – таблиць відповідності та кінцевих автоматів. Загалом ППЗП придатні для побудови будь-якого комбінаційного вузла з відносно малою кількістю входів і виходів. Наприклад, простий ППЗП на три входи й три виходи з рис. 1.10 здатний синтезувати будь-яку комбінаційну функцію з не більш ніж трьома вхідними та трьома вихідними сигналами. Щоб проілюструвати принцип дії, далі розглянемо невеликий логічний фрагмент, зображений на рис. 1.11 (схема наведена лише для цього прикладу) [18].



a	b	c	w	x	y
0	0	0	0	1	0
0	0	1	0	1	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	0	1	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	0	0

Рисунок 1.11 – Блок комбінаційної логіки

Логічний блок (рис. 1.11) може бути замінений мікросхемою ППЗП з трьома входами і трьома виходами. Для цього потрібно всього лише запрограмувати відповідні зв'язки в масиві логічних елементів АБО (рис. 1.12)

У виразах на рис. 1.12 використано такі позначення:

«&» – операція І (AND),

«|» – операція АБО (OR),

«^» – операція ВИКЛЮЧНОГО АБО (XOR),

«¯» – операція НЕ (NOT).

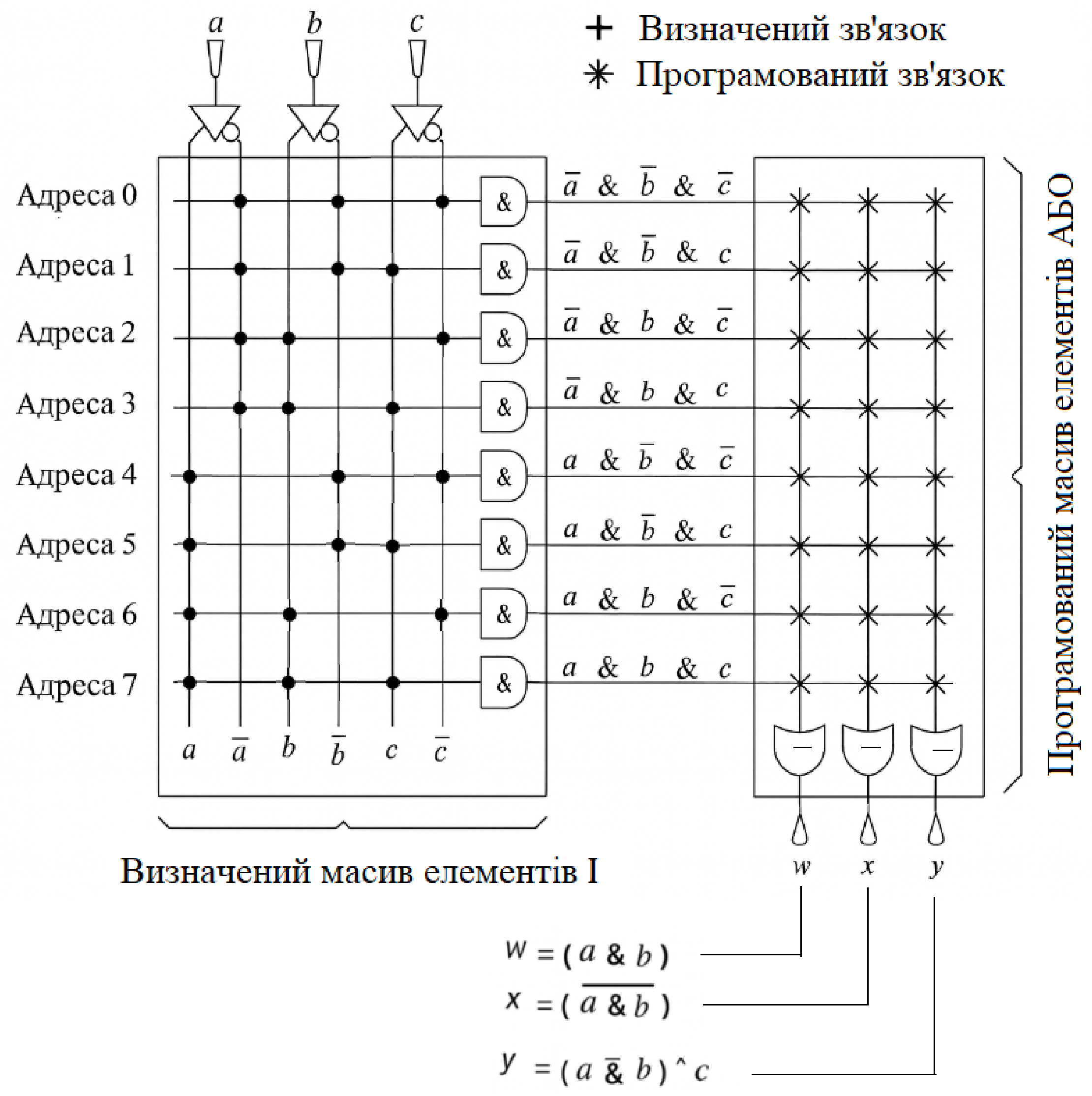


Рисунок 1.12 – Запрограмована мікросхема ППЗП

Раніше наведений приклад ППЗП був спрощеним. Насправді такі мікросхеми мають значно більше ліній введення/виведення й підходять для реалізації великих фрагментів комбінаційної логіки. Ту саму логіку традиційно будували з дискретних ІС на кшталт серії 74 від Texas Instruments.

Те, що десятки окремих логічних мікросхем можна замінити одним ППЗП, дає низку переваг: менша й простіша плата, нижча вартість і підвищена надійність (адже кожен паяний контакт – потенційне місце відмови). Крім того, якщо в схемі знайдено помилку (скажімо, замість «І–НЕ» використали «І»), її легко виправити: достатньо перепрошити інший ППЗП або стерти й заново запрограмувати варіанти на СППЗП чи ЕСППЗП. Це суттєво зручніше, ніж ловити похибки в друкованій платі, складеній з багатьох окремих ІС [19].

У логічних формулах оператор «І» (&) інтерпретують як логічне множення (добуток), а «АБО» (|) – як логічну суму (додавання). Коли маємо вираз на кшталт

$$y = (a \& b \& c) | (a \& b \& c) | (a \& b \& c) | (a \& d \& c),$$

термін «літерал» означає окрему змінну в прямій або інверсній формі (a, b тощо), а група літералів, поєднана оператором «&», – це «добуток». Наприклад, добуток (a&b&c) містить три літерали – a, b і c; і увесь вираз є «сумою добутків».

Таким чином, ППЗП особливо зручні для реалізації функцій типу «сума добутків» із великою кількістю термів, але з відносно малим числом входів. Це пов’язано з тим, що всі вхідні комбінації апаратно закодовані в матриці «І» та постійно дешифруються.

1.3 Програмовані схеми ПЛМ, PAL, GAL

Наступний етап еволюції ПЛП полягав у подоланні обмежень, притаманних архітектурі ППЗП. Перші програмовані логічні матриці (ПЛМ, PLA) з’явилися близько 1975 року. Їх часто використовували як «просунуті» прості ПЛП, оскільки програмуваними були обидва масиви – і «І», і «АБО».

Розгляньмо ПЛМ із трьома входами та трьома виходами в початковому (незапрограмованому) стані (рис. 1.13). На відміну від ППЗП, кількість елементів «І» у відповідному масиві не прив’язана до числа вхідних сигналів; за потреби нові добутки («І»-терми) додають простим розширенням масиву рядками. Аналогічно, кількість елементів «АБО» не залежить ані від числа входів, ані від розміру масиву «І»; додаткові суматори формують шляхом додавання нових стовпців у масиві «АБО».

Це досягається відповідним програмуванням з’єднань на рис. 1.14.

У ПЛМ немає жорсткої вимоги під’єднувати масив «АБО» безпосередньо до виходів масиву «І». Тож інколи застосовували альтернативні схеми, наприклад, «І» → «АБО-НЕ» — які давали прийнятні результати. Втім на практиці конфігурації на кшталт «АБО-І», «І-НЕ-АБО», «І-НЕ-АБО-НЕ» зустрічались нечасто або взагалі не використовувались. Основна причина, чому ПЛМ здебільшого будують за

архітектурою «І–АБО» (або «І–АБО–НЕ»), у тому, що логічні рівняння найчастіше записують у формі «суми добутків». Вирази типу «добуток сум» стандартно перетворюють до цієї форми засобами булевої алгебри (зазвичай це робить ПЗ).

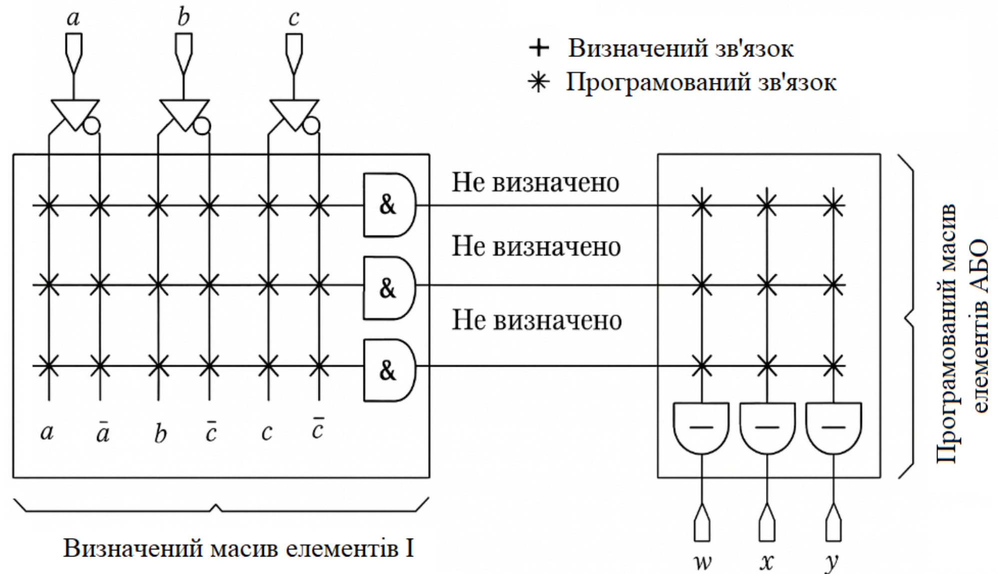


Рисунок 1.13 – Незапрограмована ПЛМ

Поставимо задачу, що потрібно реалізувати в ПЛМ три вирази:

w= (a & c) | (b & c), x = (a & b & c) | (b & c) та y = (a & b & c).

ПЛМ особливо корисні для великих схем, у яких багато вихідних рівнянь містять одні й ті самі добутки. Наприклад, на рис. 1.14 терм (b & c) використовується двічі – для формування *x* та *y*. Таке повторне використання називають «розподілом добутків».

Мінус у тому, що проходження сигналів через програмовані зв’язки повільніше, ніж через фіксовані, тож ПЛМ зазвичай працюють повільніше за ППЗП, оскільки в них програмовані обидва масиви – і «І», і «АБО». Щоб підвищити швидкодію, з’явився клас PAL (Programmable Array Logic) – по суті «дзеркальна» ідея відносно ППЗП: програмованим є лише масив «І», а масив «АБО» – фіксований. Різновидом цієї ідеології стали GAL (Generic Array Logic) від Lattice Semiconductor – КМОП-пристрої з електричним стиранням [20].

Як приклад розглянемо простий PAL на три входи й три виходи (рис. 1.15). Порівняно з ПЛМ такі ІС зазвичай швидші, бо з двох масивів програмується лише один. Водночас функціональна гнучкість PAL обмеженіша: число доступних добутків фіксоване й не може довільно зростати.

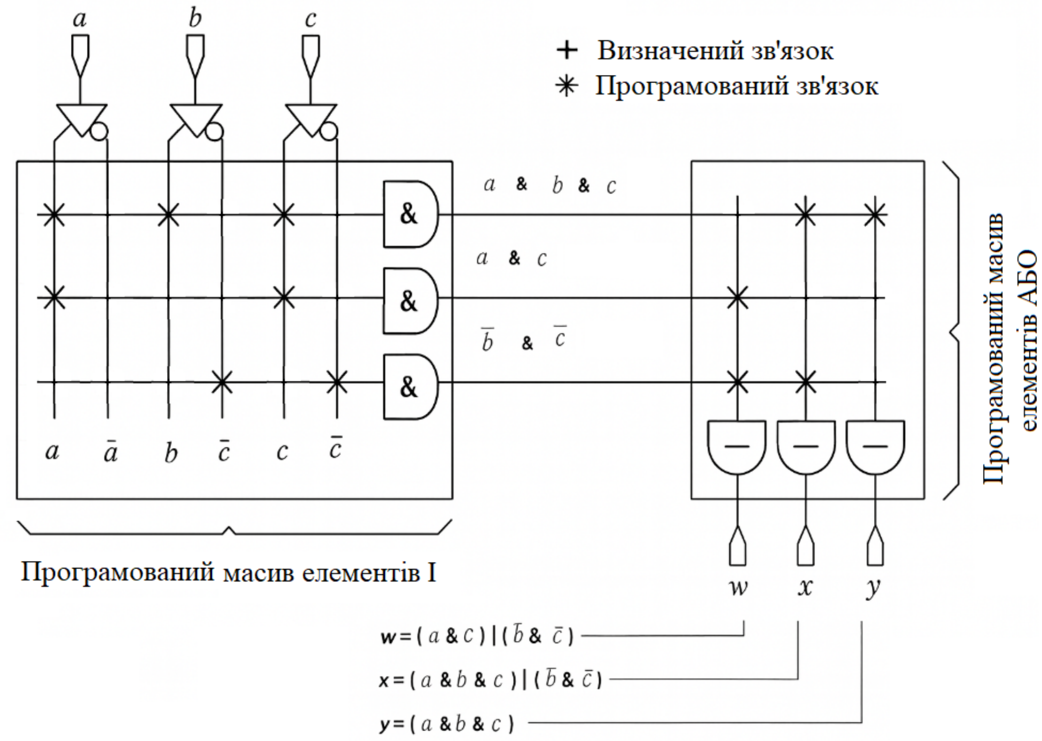


Рисунок 1.14 – Запрограмована ПЛМ

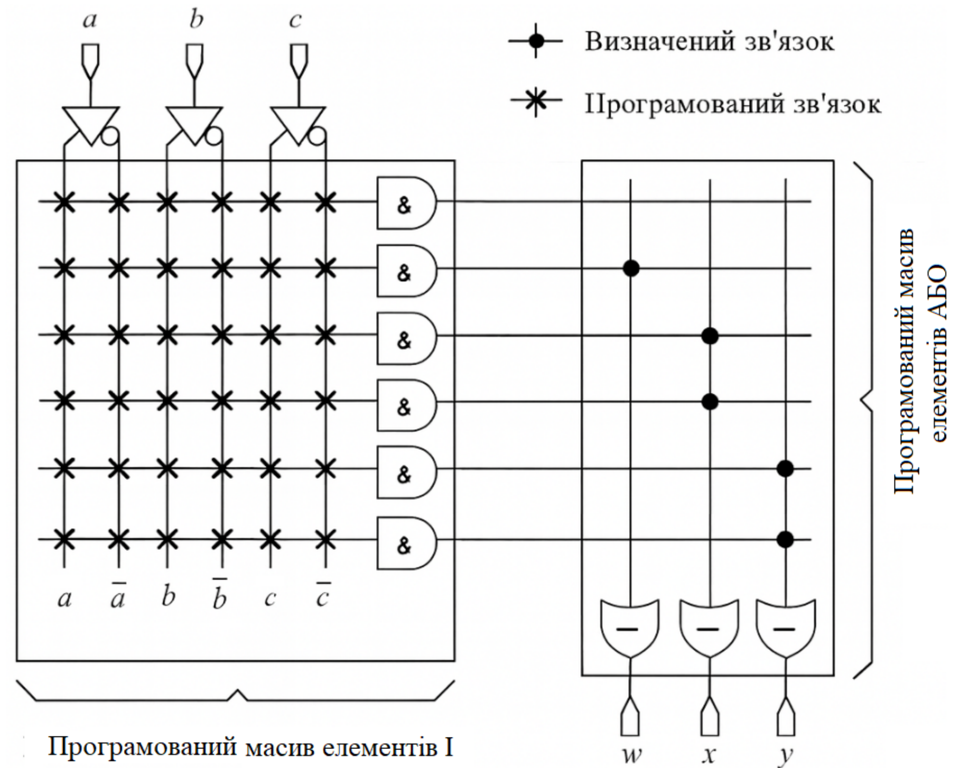


Рисунок 1.15 – Запрограмований пристій PAL

Реальні ПЛМ і PAL значно масштабніші за наведені приклади: мають багато входів, виходів і внутрішніх ліній, а також низку додаткових програмованих опцій – інверсію виходів, тристанові буфери або обидві можливості одночасно. Частина виробів підтримує «зареєстровані» виходи (засувки/тригери) й програмовані мультиплексори, що дають змогу для кожного виводу вибирати – використовувати зареєстрований чи «прямий» сигнал. У деяких чипів окремі виводи можна сконфігурувати як виходи або, за потреби, як додаткові входи [21].

Складність у тому, що різні моделі пропонують різні набори функцій, тож вибір оптимального компонента для конкретного застосування ускладнюється. Інженери або звужують перелік допустимих ІС і підлаштовують схему під них, або користуються САПР, щоб підібрати пристрій, який найкраще відповідає вимогам. Класичне інженерне завдання – досягти максимальної функціональності за мінімальних габаритів, вартості та енергоспоживання. На цьому тлі наприкінці 1970-х – на початку 1980-х з'явилися складні програмовані логічні пристрої (CPLD, complex PLD). Справжній прорив стався у 1984 році: Altera запропонувала CPLD, що поєднав КМОП із СППЗП. Слава Altera була не лише у вдалому міксі технологій. Коли архітектуру простих ПЛП масштабували до великих систем на зразок Mega-PAL, вважали, що центральна програмована комутаційна матриця має забезпечувати 100 % з'єднаність із усіма входами/виходами блоків. Це різко погіршувало швидкість, збільшувало енергоспоживання і вартість. Altera застосувала матрицю з частковою з'єднаністю: складніше ПЗ для проєктування, зате прийнятні швидкодія, потужність і ціна [22].

Попри різні реалізації у виробників, типова CPLD – це кілька блоків простих ПЛП (зазвичай PAL), поєднаних спільною програмованою комутаційною матрицею (рис. 1.16). Програмуються як самі блоки, так і з'єднання між ними через цю матрицю.

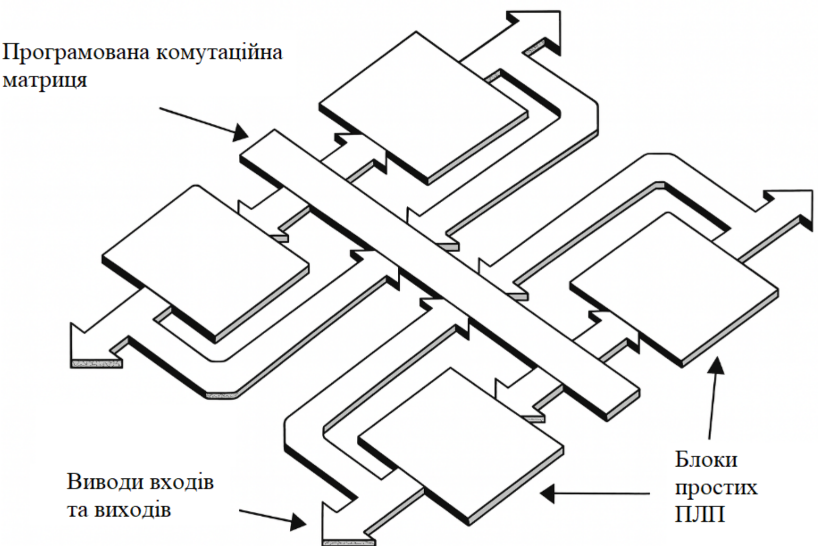


Рисунок 1.16 – Загальна структура складного ПЛП

Зображення рис. 1.16 ілюструє лише загальний принцип дії ПЛП (CPLD) і не містить усіх допоміжних вузлів. У реальному виробі всі елементи виконані на одному кристалі кремнію. Наприклад, програмована комутаційна матриця може мати дуже широку шину – скажімо, близько 100 ліній, тоді як окремий блок простого ПЛП здатний під'єднати лише обмежену кількість сигналів, наприклад 30. Зв'язок між такими блоками та матрицею забезпечують своєрідні програмовані мультиплексори (рис. 1.17) [23].

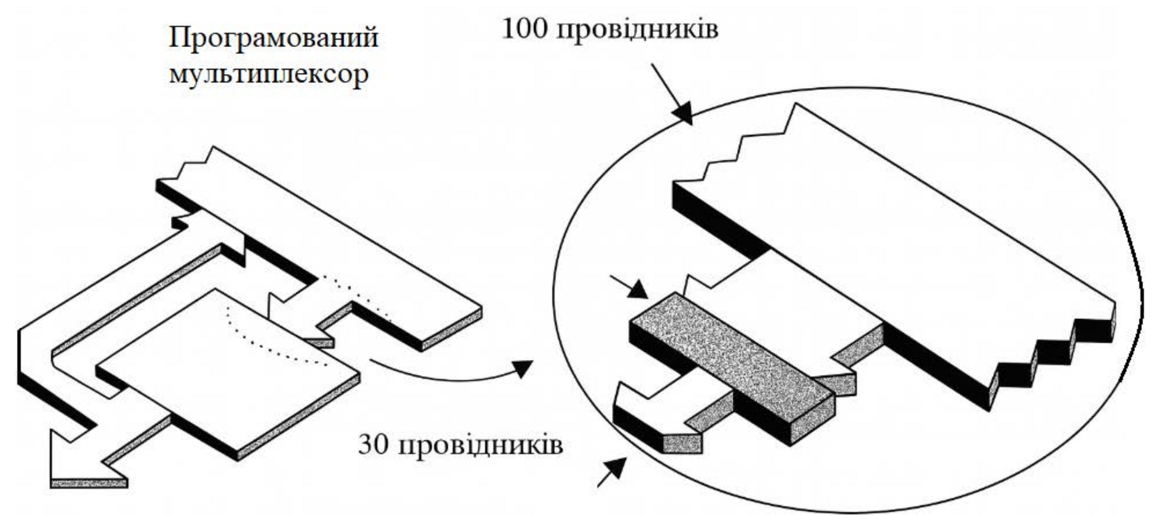


Рисунок 1.17 – Використання програмованого мультиплексора

У складних ПЛП тип перемикачів залежить від виробника та моделі: їх реалізують на комірках СППЗП, ЕСППЗП, Flash або на SRAM. Використання SRAM дає змогу підвищити гнучкість – одна й та сама пам'ять може слугувати і як програмовані перемикачі, і як робоча оперативна пам'ять [24].

У 1980 році підрозділ Асоціації електронної промисловості США – комісія JEDEC – запропонував стандарт текстових форматів для файлів, що застосовуються під час програмування ПЛП.

Приблизно тоді ж Джон Біркнер (John Birkner) – автор перших PAL і рушій їхнього розвитку – створив PALASM (PAL Assembler). Це поєднання мови опису апаратури (HDL) та прикладних утиліт: інженер описував функціональність схеми у вигляді текстового файлу з булевими рівняннями у формі «суми добутків», а PALASM зчитував цей опис і автоматично генерував програмний файл для запису в пристрій.

Попри революційність, ранні версії PALASM працювали лише з виробами MMI (Monolithic Memories Inc.) і не містили механізмів мінімізації чи оптимізації логіки. Щоб закрити ці прогалини, у 1983 році Data I/O представила мову ABEL (Advanced Boolean Expression Language), а невдовзі Assisted Technology випустила пакет CUPL (Common Universal

tool for Programmable Logic). Обидва рішення поєднували HDL і інструменти розробки, підтримували опис кінцевих автоматів і автоматичну мінімізацію логіки, а також працювали з багатьма типами ПЛП від різних виробників [25].

Окрім PALASM, ABEL і CUPL, існували й інші ранні HDL-напрямки, наприклад AMAZE (Automated Map and Zap of Equation) від Signetics. Саме ці прості мови та супровідні інструменти підготували ґрунт для високорівневих HDL – Verilog і VHDL – та для сучасних засобів проєктування (логічного синтезу тощо), що нині застосовуються у створенні як замовних ІС, так і схем на ПЛІС.

1.4 Класифікація спеціалізованих замовних ВІС (ASIC)

Замовні інтегральні схеми (ASIC, application-specific integrated circuit) зазвичай поділяють на чотири головні категорії, розташовані за зростанням складності: вентильні матриці (часто звані базовими матричними кристалами, БМК), структуровані ASIC, рішення на стандартних комірках і повністю індивідуальні (full-custom) мікросхеми (рис. 1.18).

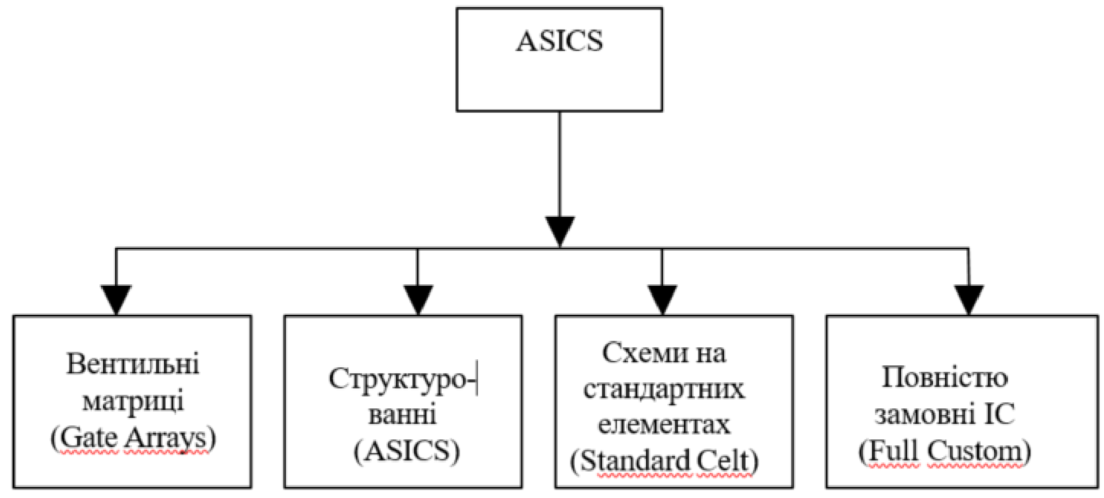


Рисунок 1.18 – Різні типи спеціалізованих замовних ІС

На зорі цифрової мікроелектроніки (окрім пам’яті) фактично існували дві групи ІС. Перша – порівняно прості стандартні мікросхеми, які масово випускали й продавали як готові компоненти компанії на кшталт Texas Instruments чи Fairchild. Друга – спеціалізовані замовні рішення, зокрема мікропроцесори, що проєктувалися й виготовлялися під конкретні потреби. У повністю замовних проєктах розробники кінцевого виробу самі відповідали за маски кожного технологічного шару. Постачальники не пропонували заздалегідь підготовлених «цеглинок» – бібліотек вентилів чи функціональних блоків – і не формували на кристалі жодних типових структур. Маючи відповідні САПР, інженери визначали

геометрію окремих транзисторів і будували з них складніші функції. Наприклад, щоб прискорити вентиль, можна було змінити розміри транзисторів, що його утворюють. Інструменти для такого конструювання часто розробляли безпосередньо в компаніях-замовниках [26].

Хоч повноіндивідуальне проєктування надзвичайно складне й трудомістке, результатом стають мікросхеми з потрібною кількістю вентилів і мінімальними «порожніми» площами на кристалі.

1.5 Архітектура базисних модулів структурованих ASIC

Концепцію матриць логічних елементів (вентильних матриць) обговорювали ще наприкінці 1960-х у IBM, Fujitsu та інших компаніях. Спершу такі рішення застосовували виключно всередині корпорацій, а для широкого ринку вони стали доступні лише в середині 1970-х, коли з’явилися КМОП-реалізації. Перші з них мали назву ULA (uncommitted logic array – «незакомутовані логічні матриці»), але згодом цей термін вийшов із ужитку [27].

Серцем вентильної матриці є базова комірка – набір попередньо сформованих, але не з’єднаних між собою транзисторів і резисторів. Кожен виробник ASIC сам визначає склад і конфігурацію такої комірки (рис. 1.19).

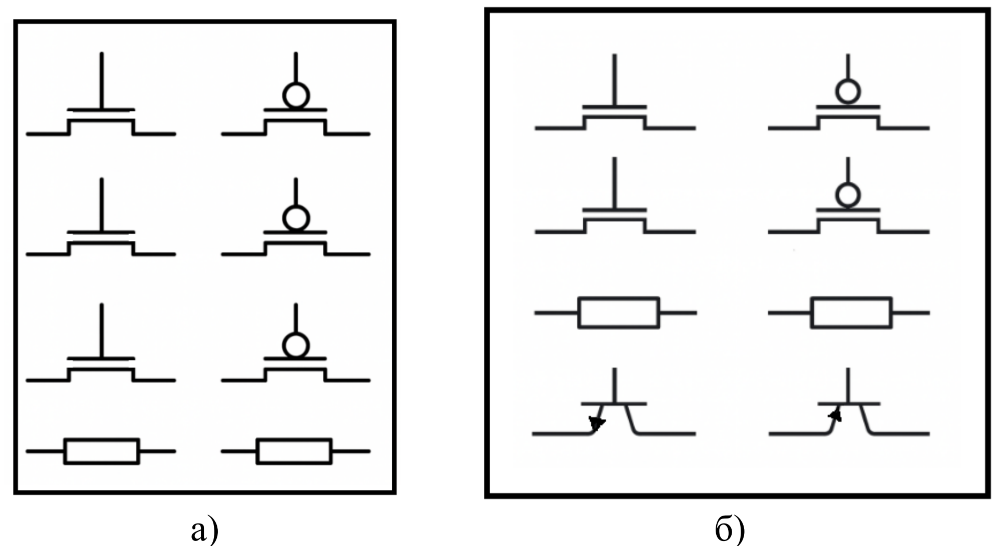


Рисунок 1.19 – Приклади простих базисних комірок вентильної матриці: а) – однорідна КМОП; б) – змішана біполярна КМОП

Виробничий процес зазвичай починався з випуску кристалів, що містили велику сітку базових комірок. Для так званих «канальних» матриць ці комірки шикували в один або два стовпці, залишаючи між стовпцями вільні смуги – канали для подальшої металізації та розведення з’єднань (рис. 1.20).

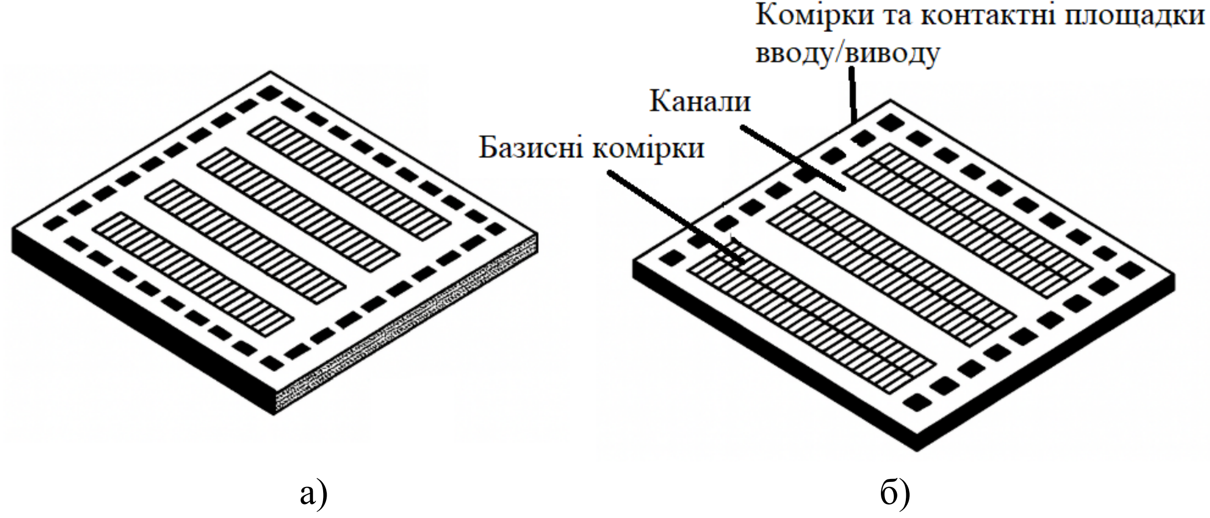


Рисунок 1.20 – Архітектура канальних логічних матриць:
а) – одностовпцеві масиви, б) – двостовпцеві масиви

Коли канали не передбачено, комірки розміщують суцільним масивом по всій площі кристала. Така структура не має спеціально виділених трас для внутрішніх з’єднань і називається безканальною.

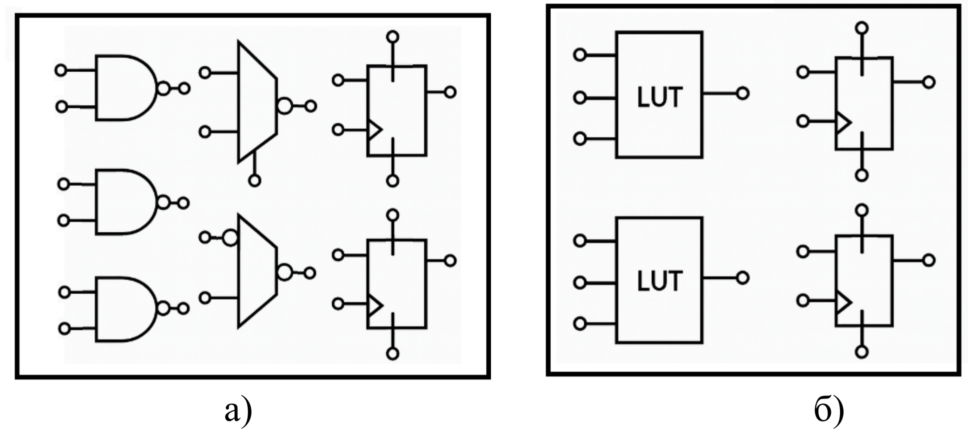
Проєктувальники користуються набором готових логічних блоків, який надає виробник: це можуть бути базові вентилі, мультиплексори, регістри тощо. Кожен такий функціональний блок вважають окремою «коміркою» (не плутати з базовою коміркою матриці). Сукупність доступних блоків формує бібліотеку елементів. Розробники отримують нетлист на рівні вентилів – перелік використовуваних вентилів і зв’язків між ними. Далі застосовують САПР для зіставлення логічних вентилів із базовими комірками, їх розміщення та трасування внутрішніх з’єднань. На підставі результатів генерують фотошаблони шарів металізації, які з’єднують елементи всередині базових комірок, а також комірки між собою та з виводами чипа.

Вартість вентильних матриць порівняно помірна: транзистори та інші структури формуються заздалегідь, а індивідуально виготовляють лише металізацію. Недолік у тому, що значна частина внутрішніх ресурсів лишається невикористаною; розміщення вентилів фіксоване, а маршрути часто далекі від оптимальних – це погіршує швидкодію й підвищує споживання. На початку 1990-х ці проблеми частково розв’язали схеми на стандартних елементах (standard-cell). Як і у випадку вентильних матриць, постачальник надає бібліотеку логічних комірок, а також набори апаратних/програмних макросів: процесорні ядра, інтерфейси зв’язку, блоки ОЗП/ПЗП тощо. Команди можуть використовувати власні напрацювання або придбавати IP-блоки; найскладніші модулі, зокрема мікропроцесори, називають «ядрами» [28].

Сучасні САПР завершують проєкт створенням нетлиста, після чого, на відміну від вентильних матриць, стандартно-коміркові ІС не спираються на заздалегідь підготовлені базові комірки на кристалі. Засоби розміщення та трасування індивідуально розташовують кожний вентиль і оптимізують з'єднання, а потім генеруються фотошаблони для всіх технологічних шарів. Такий підхід мінімізує надлишковість, покращує заповнення площі кристала і дає значно ефективнішу реалізацію, ніж вентильні матриці.

Архітектури структурованих ASIC у різних постачальників відрізняються, але спільні риси такі. Пристрій будується з базових «плиток» (модулів/тайлів), кожна з яких може містити заздалегідь сформовані блоки загальної логіки (вентилі, мультиплексори, таблиці відповідності), один чи кілька регістрів і, інколи, невелику локальну пам'ять (рис. 1.21). Масив таких елементів фабрично формується по всій площі кристала [29].

В альтернативних підходах вихідною одиницею є «базова комірка/модуль/тайл», що містить лише логічні примітиви (вентилі, мультиплексори, LUT), а вже поєднання масиву таких одиниць (наприклад 4×4, 8×8, 16×16) зі спеціалізованими модулями – регістрами, невеликими пам'яттями тощо – утворює підсистемний блок. Усі ці масиви також створюють заводським способом на всьому кристалі.



а) б)

Рисунок 1.21 – Модулі спеціалізованих структурованих ІС:
а) – модуль, що містить вентилі, мультиплексори та тригери;
б) – модуль, що містить таблиці відповідності (LUT) та тригери

Додаткові блоки – наприклад, вбудовані ОЗП, генератори тактових сигналів, логіка периферійного сканування та інші модулі – зазвичай формують фабрично, переважно вздовж країв кристала (рис. 1.22) [30].

Під час кастомізації таких мікросхем, як і у випадку вентильних матриць, індивідуально виготовляють переважно лише шари металізації. Різниця в тому, що через вищу інтегрованість модулів у структурованих ASIC більша частина металізаційних шарів уже визначена на етапі

фабричного виробництва. Тому для доведення до готового виробу часто достатньо лише двох–трьох шарів металу, а інколи – навіть одного шару з перехідними отворами. Це помітно здешевлює й пришвидшує виготовлення решти фотошаблонів [31].

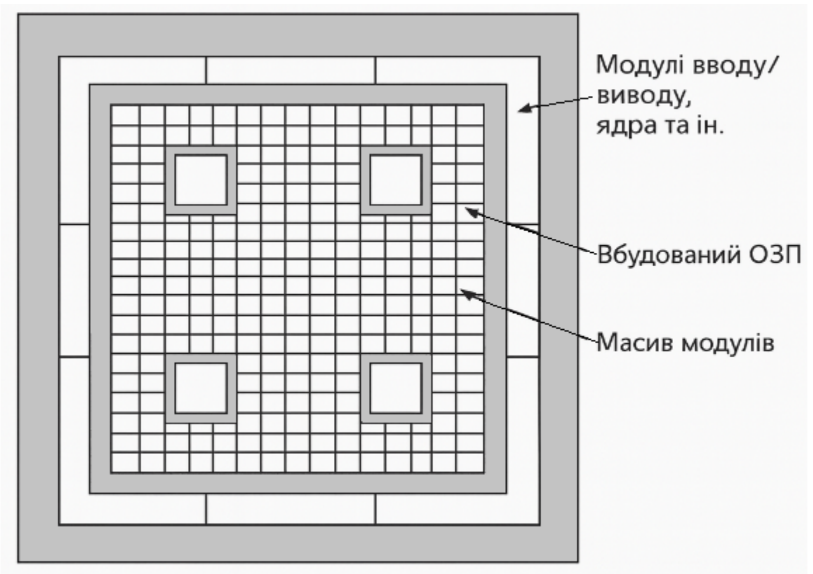


Рисунок 1.22 – Типова спеціалізована структурована ІС

Точні цифри залежать від конкретної архітектури, однак зазвичай попередньо визначена логіка структурованих рішень, порівняно зі стандартно-комірковими, споживає більше енергії, працює швидше й займає більшу площу на кристалі. У середньому для реалізації однакових функцій структурованим ASIC потрібно приблизно утричі більше площі кристала і в 2–3 рази більше потужності, ніж стандартно-комірковим реалізаціям; на практиці ці співвідношення значно варіюють залежно від типу й архітектури пристрою.

1.6 Технології проєктування ПЛІС

1.6.1 Базисний модуль ПЛІС

Ядром ранніх ПЛІС був програмований логічний блок, до складу якого входили 3-входова таблиця відповідності (LUT), тригер/засувка (регістр), мультиплексор та інші допоміжні вузли. На рис. 1.23 подано спрощений приклад такого блока (у сучасних ПЛІС він значно складніший).

Кожна ПЛІС містить багато програмованих логічних блоків. Конфігуруючи відповідні комірки SRAM, кожен блок можна налаштувати на виконання потрібної функції. Під час конфігурації для кожного регістра задають початковий стан (логічний 0 або 1) і визначають режим роботи – як тригер чи як засувка (рис. 1.24).

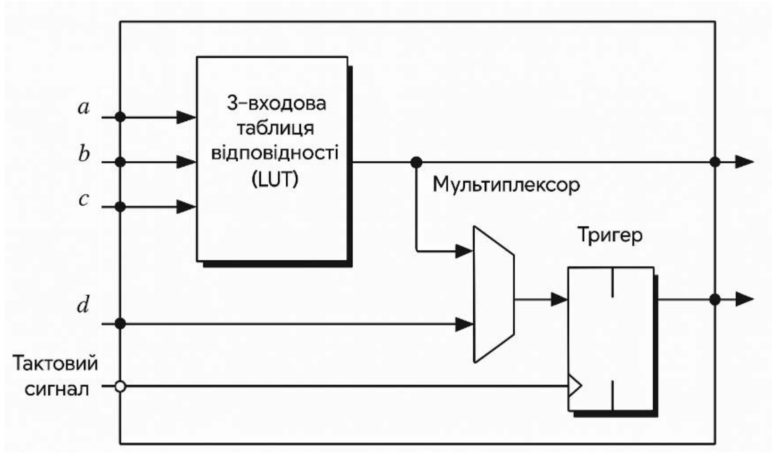


Рисунок 1.23 – Простий програмований логічний блок

Якщо вибрано режим тригера, додатково вказують, за яким фронтом тактового сигналу (позитивним або негативним) відбувається перемикання. Тактові лінії спільні для всіх блоків. Мультиплексор на вході тригера можна налаштувати так, щоб він подавав або результат із LUT, або окремий зовнішній вхід блока. Сама LUT конфігурується під будь-яку булеву функцію з трьома входами та одним виходом [32].

Наприклад, якщо потрібно реалізувати функцію *y = (a & b) | not c,* у таблицю відповідності завантажують такий вміст, щоб для кожної комбінації *a,b,c* на виході формувалося відповідне значення цієї функції (рис. 1.24).

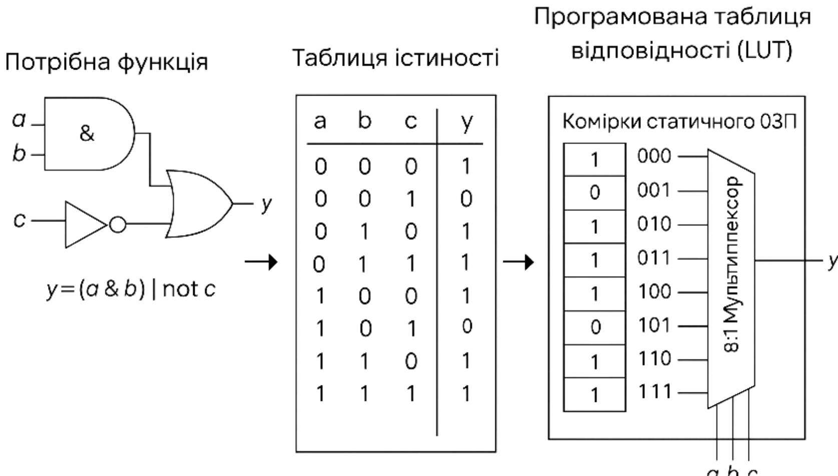


Рисунок 1.24 – Конфігурація таблиці відповідності

Таблиця відповідності разом із мультиплексором 8:1, показана на рис. 1.24, – це лише спрощена ілюстрація реальних LUT. Сучасна ПЛІС містить велику кількість таких програмованих логічних блоків, поєднаних між собою через конфігуровану внутрішню комутацію (рис. 1.25).

Спрощена схема лише приблизно відтворює архітектуру ПЛІС. Насправді всі транзистори та внутрішня комутація формуються на одному кремнієвому кристалі стандартними технологічними процесами.

Окрім локальних з'єднань (рис. 1.25), у кристалі передбачено й глобальні – високошвидкісні – траси, які передають сигнали через всю мікросхему, оминаючи численні локальні комутатори.

Пристрій також має контактні майданчики та виводи для інтерфейсу із зовнішнім світом (на рис. 1.25 не показані). Конфігураційна пам'ять SRAM визначає внутрішні маршрути так, щоб входи чипа під'єднувалися до входів одного чи кількох програмованих логічних блоків, а їхні виходи – або подавалися на наступні блоки, або безпосередньо на зовнішні виводи .[33]

ПЛІС стали містком між ПЛП і замовними ASIC. Вони поєднують високу гнучкість та короткий цикл внесення змін (як у ПЛП) із можливістю реалізовувати великі, складні функції, що раніше потребували ASIC. І хоча класичні ASIC залишаються вибором для максимально ємних і швидких систем, із розвитком технологій ПЛІС дедалі частіше займають їхню нішу у програмованій логіці.

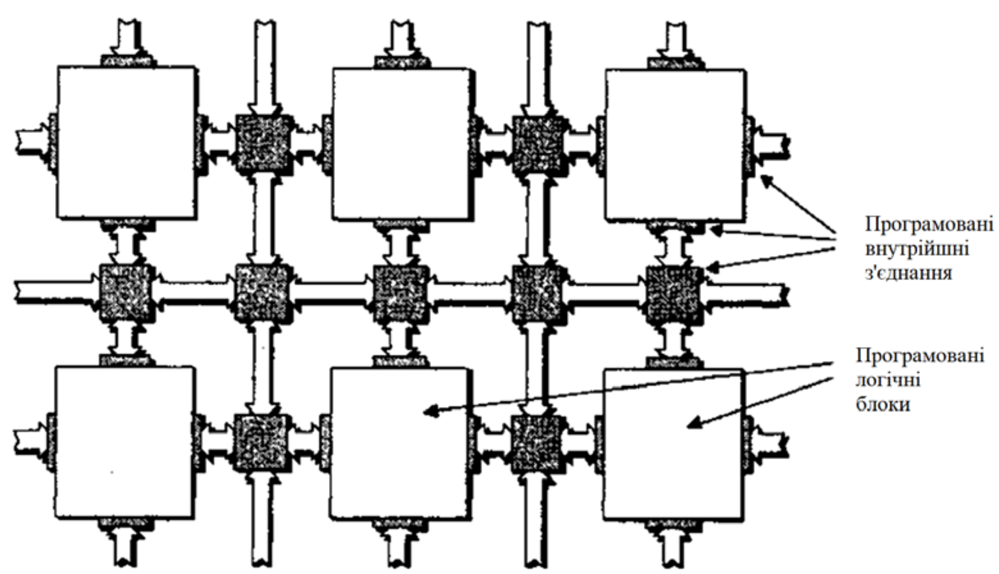


Рисунок 1.25 – Спрощена архітектура ПЛІС

Підхід «платформної» розробки давно використовували на рівні друкованих плат: маючи базову конфігурацію, створювали низку похідних виробів. Сучасні ПЛІС, крім значних ресурсів програмованої логіки, мають вбудовані ОЗП, процесорні ядра, швидкі інтерфейси вводу/виводу тощо; до того ж доступні численні IP-блоки. Це й сформувало концепцію ПЛІС-платформи: компанія може багаторазово застосовувати один спроєктований базовий чип для внутрішніх продуктів або пропонувати його як основу іншим для створення кастомних рішень.

Вбудовувати елементи «повністю замовної» логіки всередину ПЛІС зазвичай недоцільно – це повертає класичні проблеми ASIC: високу вартість, довгі терміни та складне виробництво. Водночас інколи частину логіки ПЛІС інтегрують до стандартно-коміркових ASIC або поєднують великі спеціалізовані мікросхеми з ПЛІС на одній платі, розміщуючи їх максимально близько для додавання гнучких можливостей (рис. 1.26) [34].

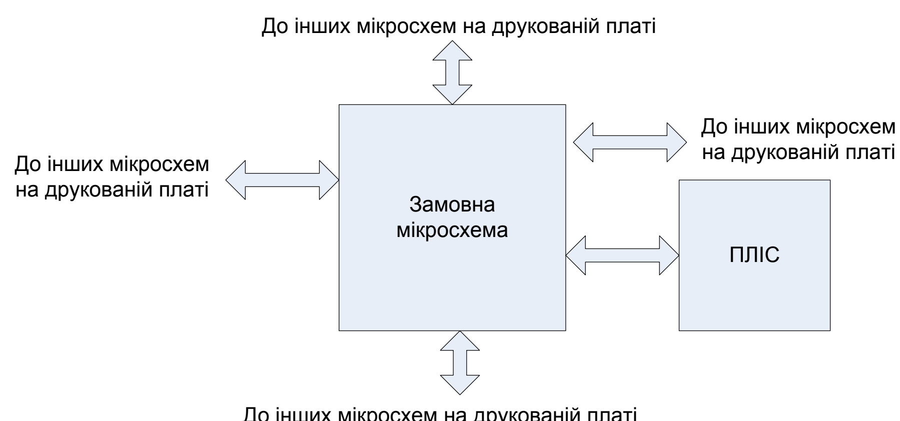


Рисунок 1.26 – Використання ПЛІС в комплексі
із замовними мікросхемами

Застосування такої схеми виправдане тим, що виправлення помилок і зміна функціоналу в системах на ASIC є дуже трудомісткими й дорогими. Навіть коли в ASIC-проєкті хиб немає, ПЛІС зручно використати для «дрібного тюнінгу» та оновлень на низькому рівні. Мінус підходу – додаткова затримка сигналів між ASIC і ПЛІС. Її можна зняти, інтегрувавши ядро ПЛІС безпосередньо в ASIC, утворюючи гібрид FPGA–ASIC. Варто пам’ятати, що САПР і методики проєктування для ASIC та ПЛІС суттєво різняться. ASIC зазвичай будують у «дрібномодульній» парадигмі – на рівні елементарних вентилів; відповідно підходять класичні методи логічного синтезу, розміщення та трасування. ПЛІС же – «середньо- великомодульні», адже фізично складаються з високорівневих блоків (програмованих логічних блоків), тож краще працюють спеціалізовані, ПЛІС-орієнтовані алгоритми синтезу, плейсменту та роутингу.

Найбільш природне поле для гібридів типу FPGA–ASIC – структуровані ASIC, бо вони також базуються на блоковій архітектурі. Постачальники таких рішень здебільшого вибирають ПЛІС-спрямовані інструменти синтезу та трасування, тож для гібридів на базі структурованих ASIC можна користуватися єдиним набором засобів: ті самі методи модульного синтезу, розміщення й маршрутизації застосовні як до «ASIC-частини», так і до «FPGA-частини». Частини кристала, схожі на програмовані логічні блоки або базові комутаційні структури, роблять «вручну» за full-custom підходом: там змагаються за кожен мікрон площі й

кожну частку наносекунди. Це невеликі, багаторазово повторювані фрагменти (транзистори, провідники), які після проєктування тисячами тиражуються по кристалу. Натомість допоміжні вузли – наприклад, конфігуровані контролери – зазвичай унікальні, до них не висувають жорстких вимог щодо площі чи швидкодії; їх реалізують методами стандартно-коміркових схем.

1.6.2 Архітектура базисного модуля ПЛІС. Реалізація на комірках ОЗП та на мультиплексорах

У ПЛІС застосовують уже відомі підходи: нарощувані перемички, конфігураційні комірки на SRAM, EEPROM/Flash тощо. До складу мікросхеми можуть входити вбудовані модулі – мультиплексори, суматори, блоки пам'яті, процесорні ядра та ін.

У більшості ПЛІС конфігурація зберігається в комірках статичної пам'яті (SRAM), які можна багаторазово перепрограмовувати. Головна перевага – швидке впровадження та перевірка нових ідей і гнучка адаптація під стандарти й протоколи. Ба більше, під час запуску система може спершу завантажити тимчасову конфігурацію (наприклад, для самотестування), а вже потім – робочу.

Ще один плюс SRAM – зрілість технології. Над її розвитком працюють провідні виробники пам'яті, а сама SRAM виготовляється тією ж КМОП-технологією, що й інша логіка ПЛІС, без екзотичних процесів. Раніше роль «тестових» виробів для нових техпроцесів виконували чипи пам'яті; нині через масштаб і складність сучасних ПЛІС їх часто використовують і для відпрацювання технологій. Наприклад, під час освоєння 90-нм (0,09 мкм) процесу компаніями IBM і UMC першими з'явилися саме ПЛІС Xilinx.

Мінус SRAM-варіанта – потрібне перезавантаження конфігурації під час кожного ввімкнення. Це потребує зовнішнього ПЗП/Flash (дорожче і займає місце на платі) або наявності вбудованого процесора чи іншого механізму ініціалізації [34].

На відміну від SRAM-рішень, ПЛІС на нарощуваних перемичках програмують у вимкненому стані через програматор. Переваги: енергонезалежність (готовність до роботи одразу після подачі живлення), відсутність зовнішньої пам'яті, «жорстка» внутрішня комутація і вища стійкість до радіації – важливо для військових і космічних застосувань. Комірки SRAM можуть «схибити» під дією випромінювання, тоді як сформована перемичка незмінна. Водночас тригери всередині все одно вразливі, тож у радіаційних умовах застосовують потрійне резервування: три копії кожного регістра з голосуванням більшістю.

Ключова перевага «перемичкових» ПЛІС – прихованість конфігурації. Програматор, звісно, може зчитувати стан під час запису для верифікації, але після завершення можна активувати додатковий захист,

що блокує читання. Навіть у разі фізичного розкриття чипа відрізнити запрограмовані й незапрограмовані перемички практично неможливо, до того ж вони розташовані у внутрішніх шарах металізації. Основний недолік – одноразовість: перепрограмувати такий пристрій не можна.

У ПЛІС на EEPROM/Flash (як і на SRAM) конфігураційні комірки утворюють «ланцюжок» на кшталт зсувного регістра. Програмування можливе офлайн програматором; деякі моделі підтримують запис «на борту», але він у кілька разів повільніший, ніж у SRAM-пристроїв. Дані після запису енергонезалежні, тож чип готовий відразу після ввімкнення. Для захисту інколи застосовують багатобітні ключі (десятки – сотні бітів): без правильного ключа через JTAG зчитати чи змінити конфігурацію неможливо; перебір зайняв би неприйнятно багато часу на типових частотах JTAG ~20 МГц.

Двотранзисторні комірки EEPROM/Flash більші за однотранзисторні приблизно у 2,5 раза, але все ж менші за SRAM-комірки. Це дозволяє зменшити площу виробу та внутрішні затримки. Натомість такі рішення потребують кількох додаткових технологічних кроків поверх стандартного КМОП-процесу, що уповільнює впровадження порівняно з SRAM-виробами; ще один мінус – помітне статичне споживання через численні навантажувальні резистори [35].

Деякі виробники комбінують технології: кожен конфігураційний біт – це пара Flash (або EEPROM) + пов’язаний SRAM. Flash попередньо прошивають, а після подачі живлення її вміст масово (паралельно) копіюється в SRAM. Маємо енергонезалежний старт, як у перемичкових ПЛІС (миттєва готовність), і гнучкість SRAM для перепрограмування «на льоту». За потреби Flash також можна оновити як у включеному стані, так і офлайн програматором.

ПЛІС поділяють на дрібномодульні та крупномодульні. Головна ідея: структура складається з безлічі програмованих логічних блоків і мережі комутації (див. рис. 1.25). У дрібномодульній архітектурі кожен блок виконує дуже просту функцію. Такі ПЛІС добре підходять для «клейової» логіки, кінцевих автоматів і систолічних алгоритмів із масовим паралелізмом; вони добре узгоджені з класичними методами синтезу, розрахованими на дрібномодульні ASIC.

Пік інтересу до дрібномодульних рішень припав на середину 1990-х, але згодом їх майже повністю витіснили крупномодульні архітектури, де один блок містить помітно більше логіки – наприклад, чотири 4-входові LUT, чотири мультиплексори, чотири D-тригери й швидкі ланцюжки перенесення. Чим більша «модульність», тим менше внутрішніх з’єднань у блоці – а саме комутація зазвичай і визначає основні затримки сигналів у ПЛІС.

У термінах класифікації є нюанс: деякі виробники створюють «крупномодульні» пристрої з вузлами, що реалізують складні алгоритмічні функції (ШПФ, навіть CPU). Формально це вже не ПЛІС у класичному

розумінні, тому LUT-архітектуру часто називають «середньомодульною», залишаючи «крупномодульні» для «тайлових» систем [36].

Програмовані логічні блоки в середньомодульних ПЛІС реалізують двома головними способами: на мультиплексорах (MUX) і на таблицях відповідності (LUT). Для MUX-підходу приклад 3-входової функції *y=(a&b)|c* показано на рис. 1.27: блок складається винятково з мультиплексорів; кожному входу можна подати *0/1*, прямий або інверсний сигнал (*a, b* чи *c*), що дає величезну кількість конфігурацій (позначка x на центральному MUX означає довільне джерело).

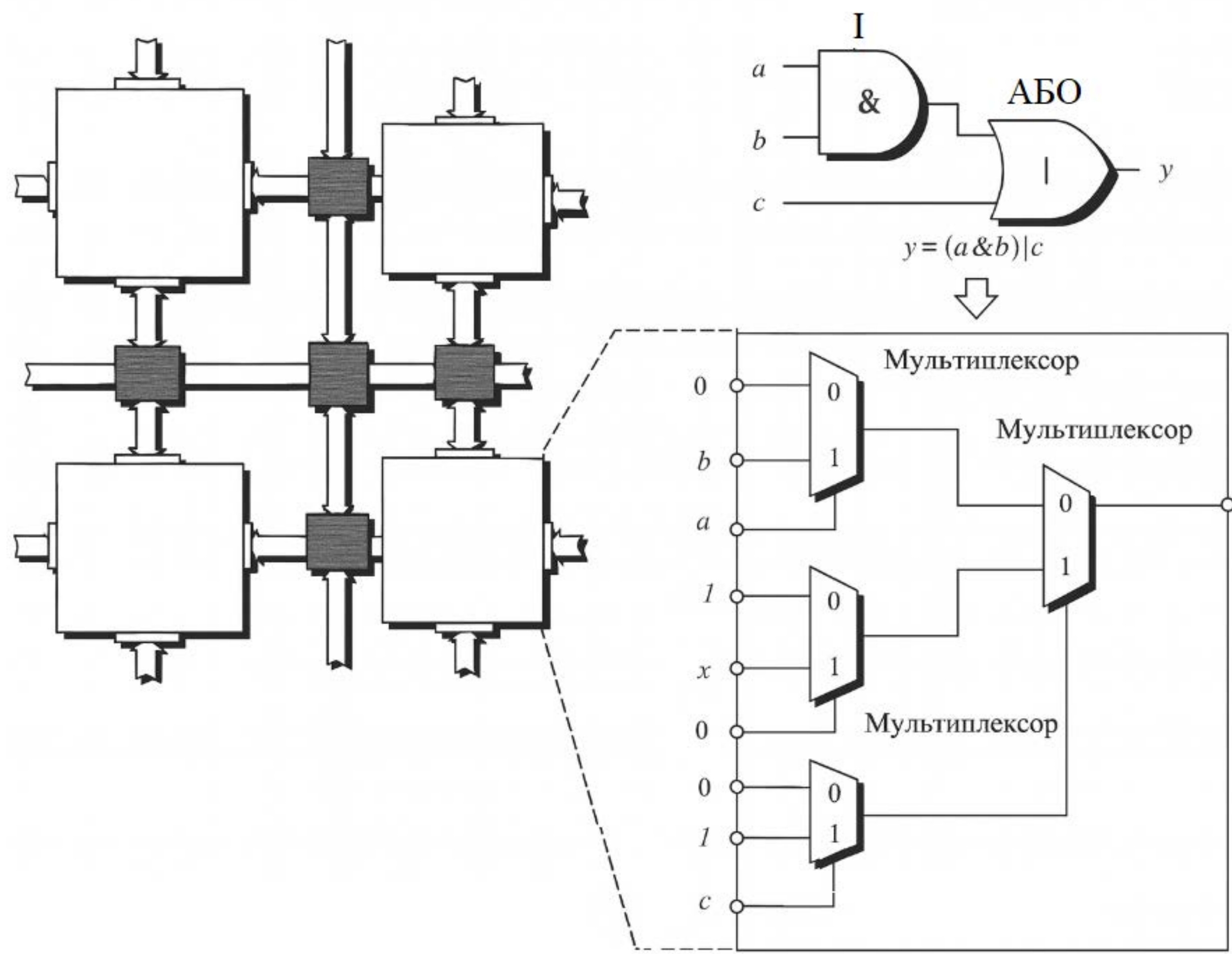
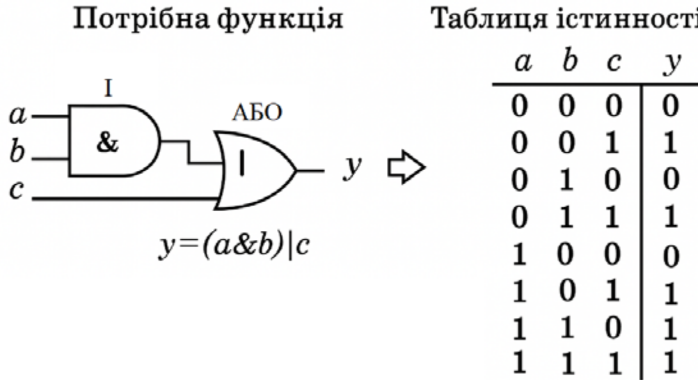


Рисунок 1.27 – Логічний блок на мультиплексорах

Концепція LUT проста: вхідна комбінація виступає адресою, а вміст таблиці містить відповідний вихід. Ту саму функцію *y=(a&b)|c* реалізують, записавши у LUT істинну таблицю для всіх наборів *a,b,c* (рис. 1.28).

Коли потрібно реалізувати логіку з кількома послідовними рівнями (шарами), LUT виявляється дуже вигідною і за використанням ресурсів, і за затримкою поширення сигналу. Під «глибиною» мають на увазі кількість елементарних логічних операцій на шляху від входу до виходу (у прикладі на рис. 1.29 – два рівні). Водночас у LUT-архітектури є мінус:

навіть для зовсім простої функції, скажімо 2-входового елемента «І», доводиться задіювати всю таблицю, через що затримка для такої дрібної операції виходить відчутно більшою, ніж хотілося б.



a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Рисунок 1.28 – Логічна функція та відповідна до неї таблиця істинності

Щоб реалізувати функцію, у 3-входову LUT потрібно записати відповідні вихідні значення. Таку таблицю можна побудувати на базі комірок SRAM, а також реалізувати за допомогою нарощуваних перемичок, EEPROM або Flash-пам’яті. Вибір потрібної комірки здійснюється вхідними сигналами через каскад передавальних ключів (рис. 1.29). Зауважте, що комірки SRAM для конфігураційних даних зазвичай об’єднують у довгий зсувний ланцюжок; на рис. 1.29 ці ланцюжки опущено для спрощення [37].

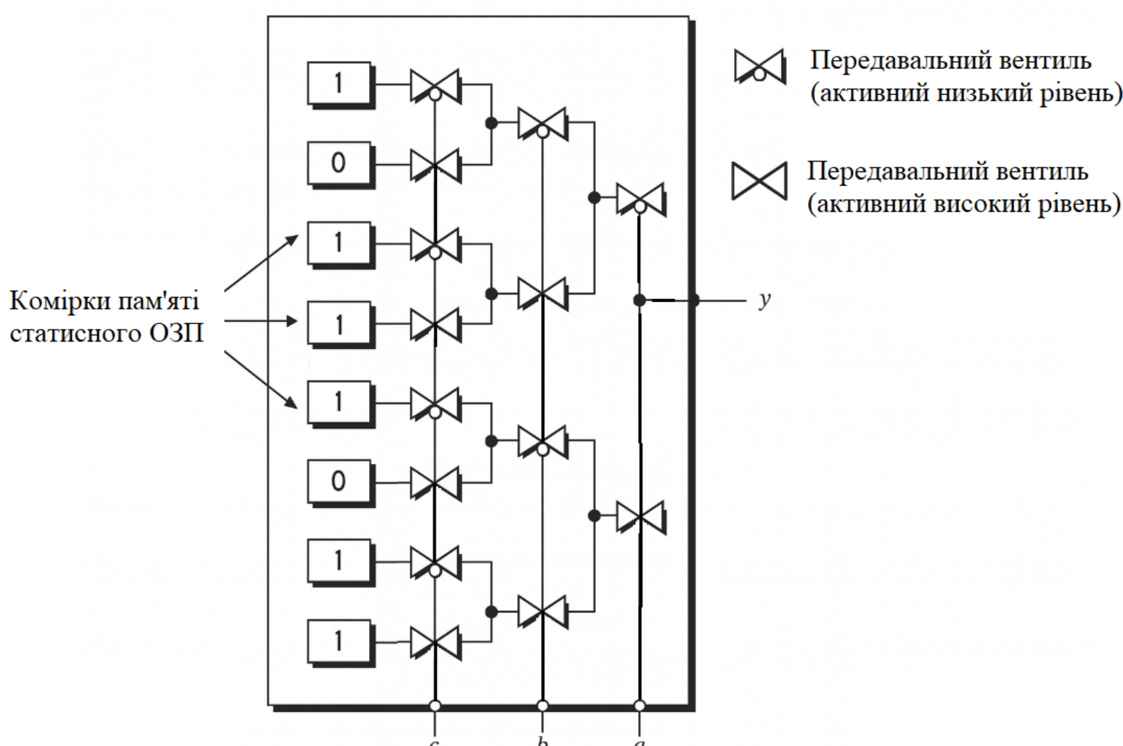


Рисунок 1.29 – Таблиця істинності на основі передавальних вентилів

У відкритому (активному) стані передавальний ключ пропускає сигнал з входу на вихід; у закритому – ізолює свій вихід від приєднаної лінії. Ключі з невеликим «колом» на позначенні спрацьовують за рівня логічного 0 на керувальному вході, а без «кола» – за логічної 1. Виходячи з цього, легко простежити як різні поєднання вхідних сигналів адресують потрібну комірку пам'яті. До епохи потужних САПР, коли схеми часто проєктували вручну, вважали, що мультиплексорна архітектура може дати кращі результати, особливо в керувальній логіці. Однак окремі її реалізації не підтримують швидкі ланцюжки перенесення, тому для арифметики LUT-рішення зазвичай переважають [38].

Оскільки ядро LUT у SRAM-рішеннях побудоване з низки пам'ятних бітів, їх можна використовувати й нетипово. Деякі ПЛІС дозволяють залучати ці біти як невеликі ОЗП: наприклад, 4-входова LUT (16 біт) працює як пам'ять 16×1. Такі ділянки називають розподіленим ОЗП, бо вони розсіяні по всьому кристалу й відрізняються від великих блоків пам'яті. Інша опція випливає з того, що всі конфігураційні біти, включно з тими, що утворюють LUT, логічно поєднані в довгий «зсувний» ланцюг (рис. 1.30).

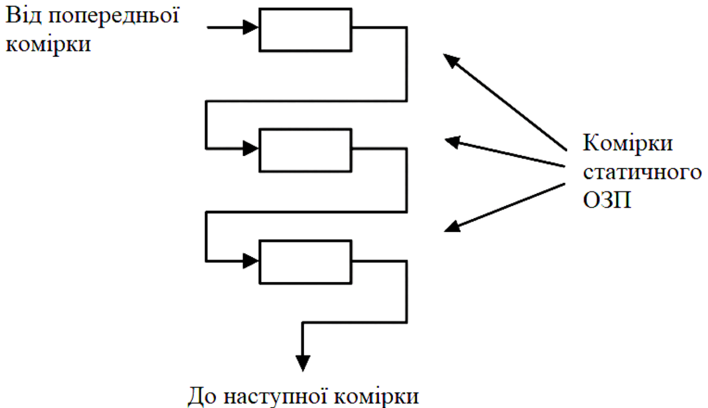


Рисунок 1.30 – Конфігураційні комірки, зв'язані в ланцюжок

Порівняно з LUT, у змішаних MUX-вузлах (мультиплексори + вентилі) частіше є доступ до проміжних вузлів сигналу, тож під час реалізації простих функцій можна відключати зайві блоки. Через це MUX-архітектура інколи виграє за швидкодією та використанням площі, коли потрібно багато дрібних незалежних логічних функцій. У 1990-х ПЛІС масово увійшли в мережеві й телеком-застосунки з великими потоками даних – тут LUT-архітектура показала себе якнайкраще. Із зростанням пропускної здатності та складності синтезу ручні методи й «чисто MUX» підходи відступили; більшість сучасних ПЛІС побудовані на LUT. Ключова властивість багатовходової LUT – здатність реалізувати будь-яку w-вхідну комбінаційну функцію. Що більше входів, то складнішу логіку можна вмістити, але кожен доданий вхід подвоює кількість комірок SRAM у таблиці. Перші ПЛІС мали 3-входові LUT. Порівняльні дослідження

3/4/5/6-входових варіантів показали, що найкращий компроміс дають 4-входові LUT. Деякі виробники експериментували зі змішаними розмірами (3+4 входи) для підвищення ефективності, але більшість інженерів воліє покладатися на синтез, тому нині успішні архітектури здебільшого стандартизовані на 4-входових LUT. Втім, із розвитком САПР «гібридні» підходи можуть повернутися [39].

У низці ПЛІС бітові осередки, що утворюють LUT, після конфігурування можна від'єднати від основного зсувного ланцюга й задіяти як окремий регістр зсуву (рис. 1.30). Відтак кожна LUT фактично стає багатофункційним елементом.

1.6.3 Архітектура базисного модуля, що конфігурується (логічна комірка фірми Xilinx)

У сучасних ПЛІС Xilinx базовим елементом є логічна комірка (logic cell). Вона містить 4-входову LUT, яку можна задіяти як пам'ять 16×1 або як 16-бітний зсувний регістр, а також має мультиплексор і регістр (рис. 1.31).

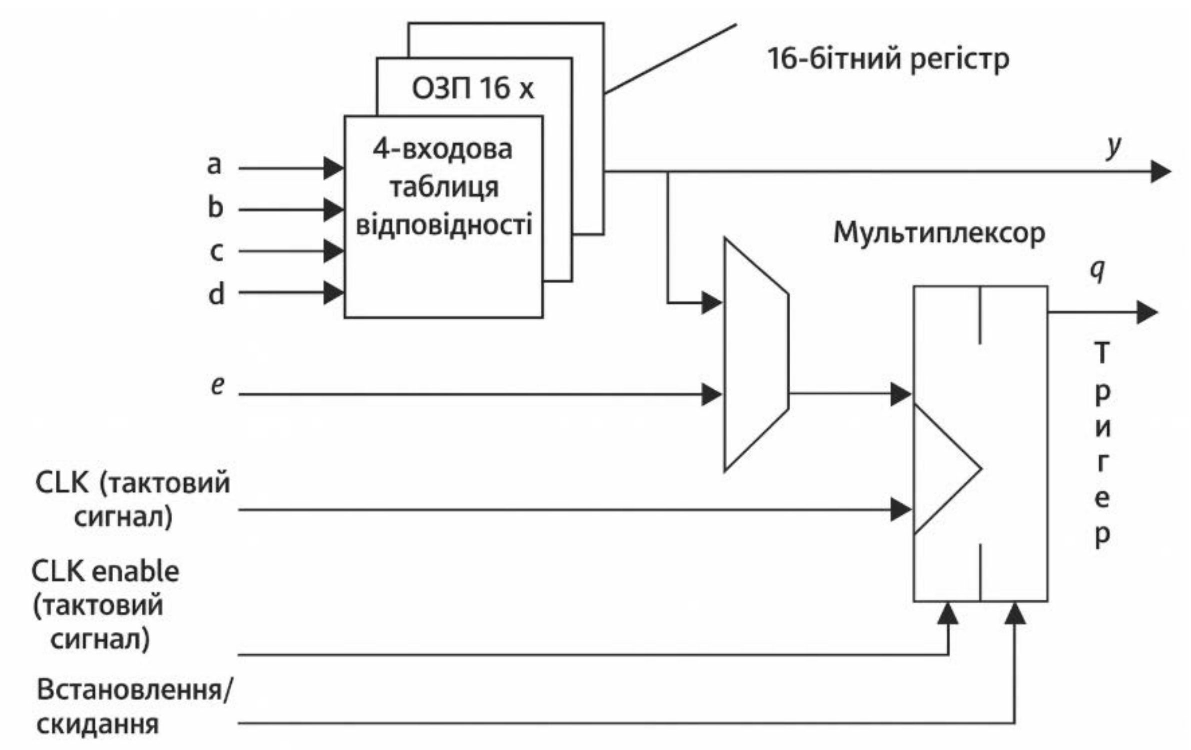


Рисунок 1.31 – Спрощений вигляд логічної комірки Xilinx

Схема на рис. 1.31 істотно спрощена, але достатня для пояснень у цьому розділі. Вбудований регістр можна налаштувати як тригер або як засувку; полярність спрацювання (по фронту чи по спаду такту), а також активні рівні сигналів «дозвіл такту» та «встановлення/скидання» задаються конфігураційно. Окрім LUT, мультиплексорів і регістрів,

логічна комірка містить і інші допоміжні вузли, зокрема швидкі ланцюжки перенесення для арифметики [40].

У ПЛІС компанії Altera базовою одиницею є логічний елемент (logic element). Хоч між «логічною коміркою» Xilinx і «логічним елементом» Altera є помітні відмінності реалізації, загальна ідеологія будови дуже подібна.

1.6.4 Конфігурація логічних блоків ПЛІС (комірки, секції, логічні масиви, функціональні модулі)

У ієрархії ПЛІС наступний рівень над логічною коміркою те, що Xilinx називає «секцією» (slice). На момент запровадження терміна секція містила дві логічні комірки (рис. 1.32). В інших виробників – зокрема Altera – для цього рівня використовують власні назви. Важливо розуміти, що LUT, мультиплексори та регістри кожної комірки мають окремі входи/виходи даних, а спільними для всієї секції є лінії такту, дозволу такту та встановлення/скидання [41].

Ще вище за рівнем у Xilinx – конфігурований логічний блок (CLB, configurable logic block); у Altera його аналог називається LAB (logic array block). Інші вендори застосовують власні, але змістовно подібні терміни. Склад CLB еволюціонував: спочатку це були дві 3-входові LUT і один регістр; згодом – дві 4-входові LUT і два регістри; далі – по дві чи чотири секції, у кожній з яких розміщувалися дві 4-входові LUT та два регістри. Рух у бік зростання ресурсів триває.

На практиці деякі лінійки Xilinx мають по дві секції в кожному CLB, інші – по чотири. Умовно CLB можна уявити як «клітинку» програмованої логіки з налаштовуваними внутрішніми з’єднаннями (рис. 1.32).

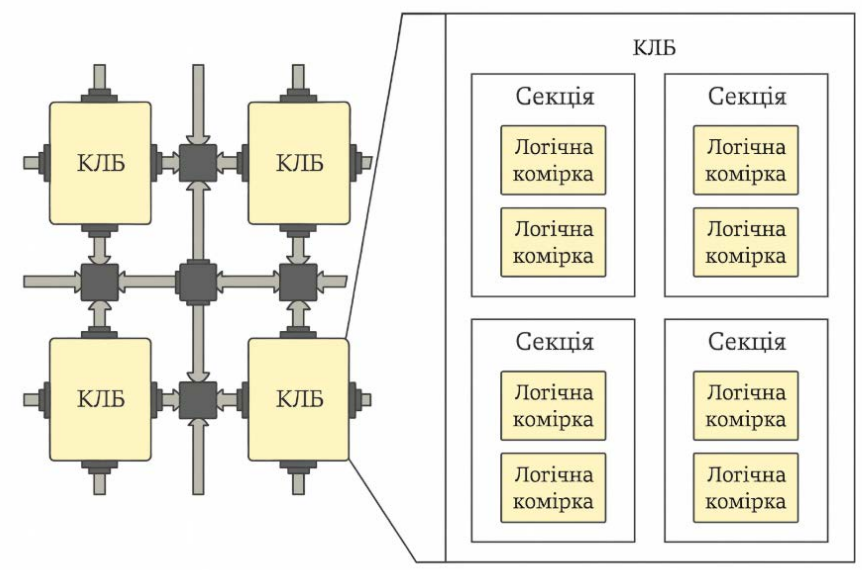


Рисунок 1.32 – КЛБ, що містить чотири секції

Усередині кожного логічного блока передбачено швидкі програмовані з'єднання, якими пов'язують сусідні секції (на рис. 1.32 їх опущено для спрощення). Ієрархія «логічна комірка → секція (дві комірки) → КЛБ із кількома секціями» відповідає ієрархії комутації: найшвидші лінії діють між комірками всередині секції, повільніші – між секціями в межах блока, і ще повільніші – між самими блоками. Така побудова дає збалансований компроміс між простотою з'єднань і мінімізацією затримок сигналів [42].

Кожна 4-входова LUT може працювати як пам'ять 16×1. Якщо взяти КЛБ із чотирма секціями (рис. 1.32), усі LUT у ньому можна налаштувати на такі режими:

- однопортове ОЗП 16×8;
- однопортове ОЗП 32×4;
- однопортове ОЗП 64×2;
- однопортове ОЗП 128×1;
- двопортове ОЗП 16×4;
- двопортове ОЗП 32×2;
- двопортове ОЗП 64×1.

«Порт» – це сукупність ліній даних і керування. В однопортовій пам'яті читання й записування відбуваються тією самою шиною; у двопортовій – окремими, зазвичай із власними адресними лініями, тож читання і записування можуть виконуватись одночасно. Кожну 4-входову LUT можна також використати як 16-бітовий зсувний регістр. Для цього існують спеціальні внутрішні зв'язки в межах секції та між секціями, які дозволяють подавати «останній» біт одного регістра на «перший» біт наступного без виходу через LUT. Саму LUT можна залучити для спостереження за довільним бітом 16-бітового регістра.

Об'єднуючи LUT усередині КЛБ, реально побудувати зсувний регістр до 128 біт. Важлива риса сучасних ПЛІС – наявність прискореного перенесення: у кожній логічній комірці є спеціальна логіка перенесення, доповнена виділеними трасами між комірками секції, між секціями всередині блока та між блоками. Такі швидкі шляхи суттєво прискорюють лічильники, суматори й інші арифметичні вузли. У поєднанні з LUT-регістрами зсуву, вбудованими множниками та іншими спеціальними блоками це забезпечує потрібну базу для задач цифрової обробки сигналів.

Оскільки більшості застосунків потрібна пам'ять, сучасні ПЛІС містять великі вбудовані масиви – блоки пам'яті. Їхнє розташування залежить від архітектури: уздовж периметра кристала, «розсіяно» по площі з відносною ізоляцією або впорядковано стовпцями (рис. 1.33).

Залежно від моделі пристрою, розмір вбудованих блоків ОЗП може коливатися від кількох тисяч до кількох десятків тисяч бітів. В одній мікросхемі зазвичай розміщують від кількох десятків до кількох сотень

таких блоків. У підсумку загальна місткість пам’яті сягає діапазону від сотень тисяч до кількох мільйонів бітів.

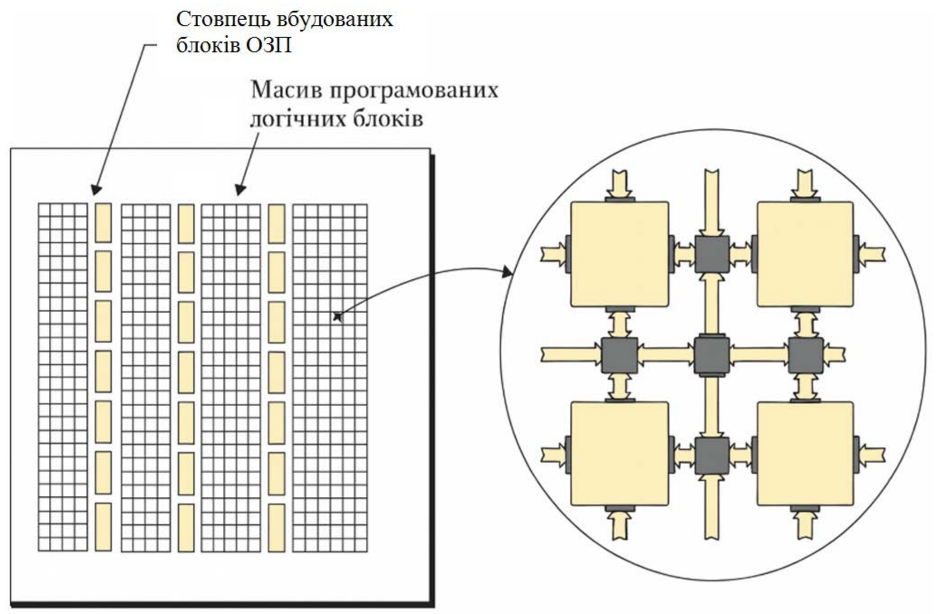


Рисунок 1.33 – Кристал із стовпцями вбудованих блоків ОЗП

Кожний блок ОЗП може працювати як окремий запам’ятовувальний модуль або об’єднуватися з іншими блоками для формування великих масивів пам’яті. Їх застосовують у різних ролях: як стандартні одно- чи двопортові блоки, як черги FIFO (first-in first-out), для реалізації кінцевих автоматів тощо.

Окремі функції, зокрема множення, під час побудови лише з програмованих логічних блоків виходять відносно повільними. Оскільки подібні операції широко використовуються, у багатьох ПЛІС передбачено спеціалізовані апаратні множники. Їх зазвичай розміщують поруч із вбудованими блоками ОЗП, адже ці ресурси часто працюють у парі (рис. 1.34).

Деякі виробники ПЛІС додають ще й спеціалізовані суматори. Водночас однією з базових операцій у задачах цифрової обробки сигналів є «множення з накопиченням» (MAC, multiply-and-accumulate) (рис. 1.35). Як випливає з назви, модуль спершу перемножає дві величини, а потім додає отриманий добуток до значення, яке вже зберігається в акумуляторі.

Якщо ПЛІС оснащена лише вбудованими помножувачами, то для реалізації MAC потрібно з’єднати цей помножувач із суматором, побудованим на кількох програмованих логічних блоках. Підсумкове значення можна зберігати у тригерах цих блоків, у вбудованих блоках ОЗП або в розподіленій пам’яті. Реалізація значно спрощується, коли в ПЛІС уже інтегровані апаратні суматори; деякі сімейства навіть мають готові вузли «помножувач + акумулятор», що безпосередньо підтримують MAC.

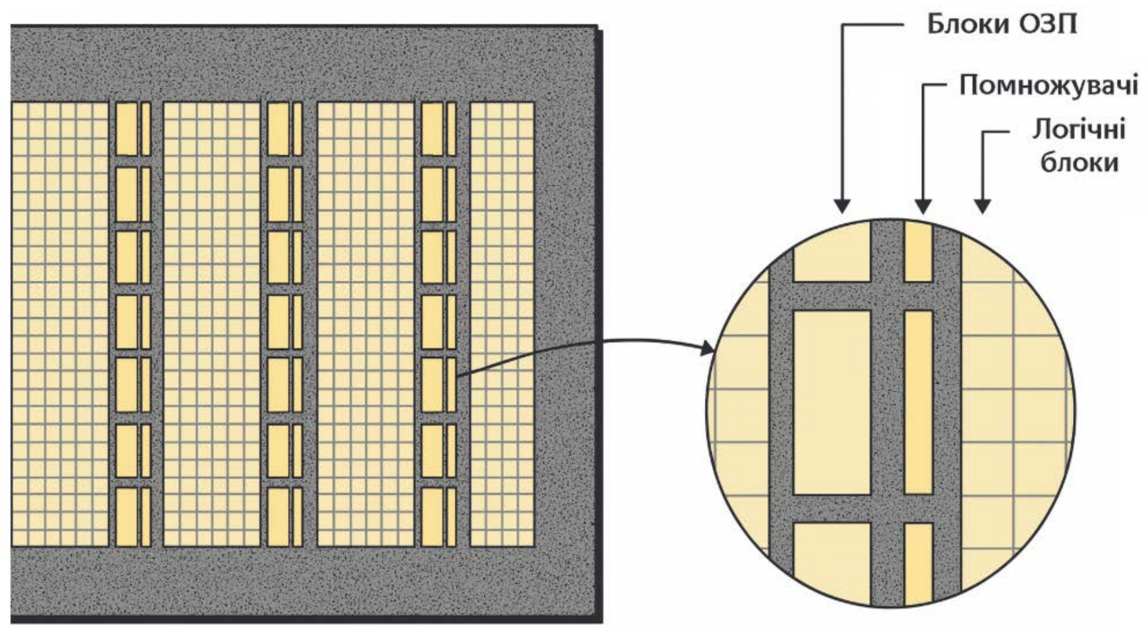


Рисунок 1.34 – Кристал із вбудованими помножувачами та блоками ОЗП

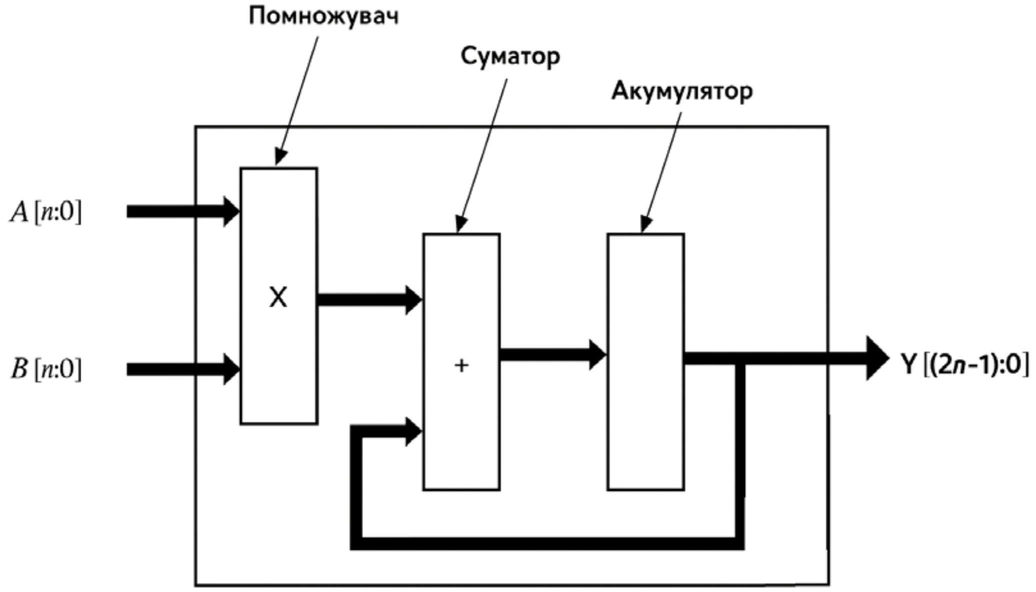


Рисунок 1.35 – Функції, що формують операцію множення
з накопиченням

Важлива риса ПЛІС полягає в тому, що майже будь-який компонент електронної системи можна реалізувати двома шляхами: апаратно (на логічних вентилях, регістрах тощо) або програмно (як набір інструкцій для мікропроцесора). Ключовий критерій вибору між цими підходами – вимоги до часу виконання [61]:

- логіка з пікосекундними та наносекундними затримками має працювати максимально швидко й тому реалізується апаратно всередині ПЛІС;

- логіка з мікросекундними масштабами є помірно швидкою й може бути як апаратною, так і програмною; зазвичай найбільше часу тут витрачається на вибір потрібного шляху обробки;

- логіка з мілісекундними інтервалами характерна для інтерфейсних задач (опитування перемикачів, керування світлодіодами). Тут часто доводиться навмисне «сповільнювати» апаратну частину – наприклад, великими лічильниками для створення затримок. Через це подібні функції нерідко зручніше виконувати на процесорі: він природно працює повільніше за апаратну логіку, хоча інколи програмна реалізація може виявитися доволі нетривіальною [43].

Більшість сучасних пристроїв у тій чи іншій формі використовують мікропроцесори. Якщо раніше це були окремі мікросхеми на платі, то тепер з'явилися ПЛІС високого рівня інтеграції з одним або кількома вбудованими процесорами – так званими мікропроцесорними ядрами. Використовуючи такі ПЛІС, має сенс перенести завдання, що виконувалися зовнішнім процесором, на вбудоване ядро. Це дає низку переваг: зменшення вартості, скорочення кількості доріжок, посадкових місць і виводів на друкованій платі, а також зменшення її габаритів і маси.

Мікропроцесорні ядра постачаються як готові апаратні блоки. Інтегрувати їх у ПЛІС можна різними способами. Один підхід – розміщення ядра «смугою» (stripe) уздовж однієї зі сторін основної області кристала, тобто головної структури ПЛІС (рис. 1.36).

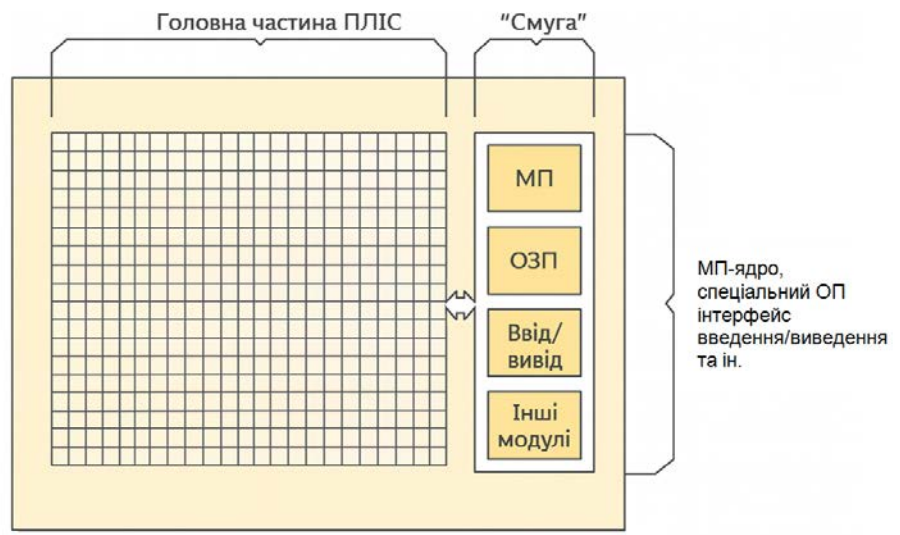


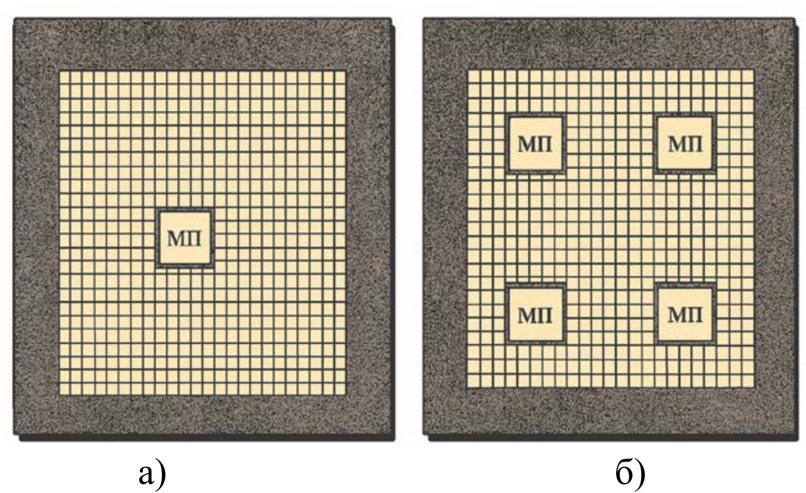
Рисунок 1.36 – Кристал з вбудованим ядром, що знаходиться за межами головної частини

За такого підходу всі вузли зазвичай інтегрують на одному кремнієвому кристалі, хоча можливий варіант із двома кристалами, складеними в багатокристальний модуль (Multichip Module, MCM). Основна область ПЛІС також містить вбудовані блоки пам'яті, помножувачі та інші згадані раніше елементи, які на схемі опущено для спрощення. Перевага цієї архітектури в тому, що «ядро» ПЛІС

залишається однаковим як для версій із вбудованим мікропроцесорним ядром, так і без нього. Додатково виробники можуть згрупувати допоміжні ресурси – пам’ять, інтерфейси введення/виведення та інші – у єдину «смугу», доповнюючи процесорне ядро. [44]

Інший варіант – розміщення одного чи кількох процесорних ядер безпосередньо всередині головної логічної області ПЛІС. Існують рішення з одним, двома, а подекуди й чотирма ядрами (рис. 1.37). Основна частина ПЛІС у цьому випадку, як і раніше, містить вбудовані ОЗП-блоки, помножувачі та інші функціональні модулі, які на ілюстрації опущено задля ясності.

Під час такого підходу інструменти проєктування мають враховувати наявність мікропроцесорів усередині кристала. Пам’ять для ядра формується з вбудованих ОЗП-блоків, а функції з’єднання реалізують за допомогою груп універсальних програмованих логічних блоків. Прихильники цієї архітектури наголошують: близьке розташування процесорного ядра до основної логічної області ПЛІС дає виграш у швидкодії.



а) б)

Рисунок 1.37 – Кристал з ядром, вбудованим до головної частини:
а) – з одним вбудованим ядром; б) – з чотирма вбудованими ядрами

Окрім фізичного вбудування процесора, можна сконфігурувати певну кількість логічних блоків так, щоб вони виконували роль процесора. Такі рішення зазвичай називають «програмними ядрами», які точніше поділяють на програмні та мікропрограмні – залежно від того, як саме за допомогою логіки реалізовано процесорні функції. Подібні ядра простіші й повільніші за апаратні аналоги, але мають суттєву перевагу: за потреби можна інстанціювати одне або кілька ядер доти, доки вистачає ресурсів ПЛІС. Типова швидкодія програмного ядра становить близько 30–50 % від швидкості апаратного. Усі синхронні елементи ПЛІС (наприклад, регістри всередині логічних блоків), які конфігуруються як тригери, мають працювати від тактового сигналу. Зазвичай такт формується поза мікросхемою, подається на спеціальні тактові входи, після чого розводиться внутрішніми мережами синхронізації до потрібних регістрів.

На рис. 1.38 зображено спрощену схему дерева розподілу такту (самі програмовані логічні блоки для наочності не показано).

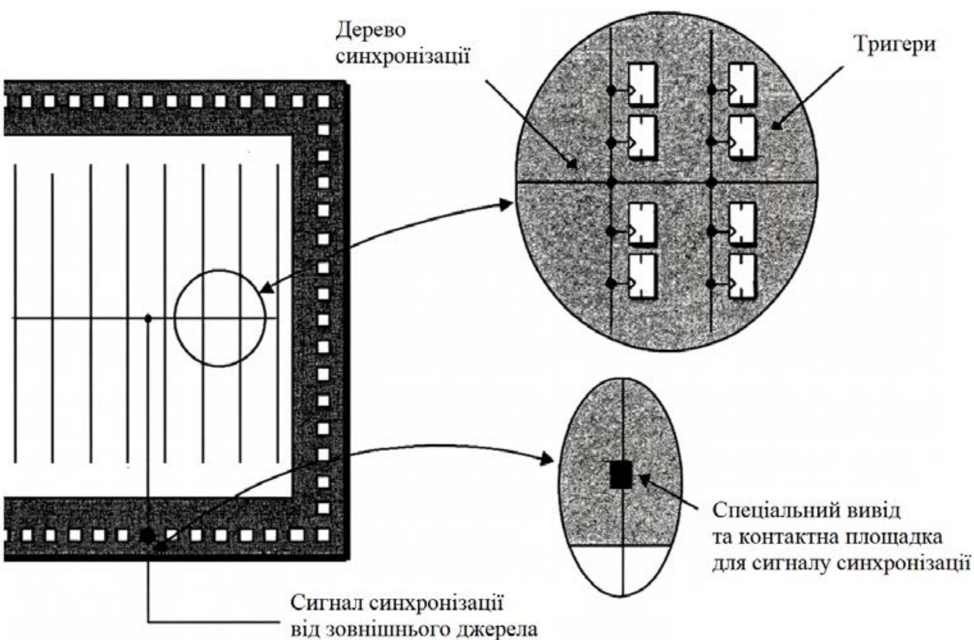


Рисунок 1.38 – Просте дерево синхронізації

Назву «дерево синхронізації» дали через те, що головний тактовий сигнал розгалужується, немов крона: на кінцях «гілок» розташовані тригери – своєрідні «листки». Така топологія допомагає забезпечити максимально одночасне надходження такту до всіх тригерів. Якби ж сигнал ішов одним довгим провідником послідовно, елементи, ближчі до тактового входу мікросхеми, одержували б імпульс значно раніше за дальні, що створювало б фазовий зсув і пов’язані з ним проблеми. Втім, навіть у дереві синхронізації невеликі різниці фаз можливі як між регістрами на одній гілці, так і між різними гілками.

Дерево такту прокладають спеціальними трасами, відокремленими від загальних внутрішніх з’єднань. Наведений опис суттєво спрощений: на практиці мікросхеми мають кілька тактових входів (невикористані часто можна задіяти як GPIO), а всередині пристрою існують кілька тактових доменів – тобто кілька дерев синхронізації.

Зовнішній тактовий вхід зазвичай не під’єднують безпосередньо до дерева. Спершу його подають на вузол керування тактуванням – «диспетчер синхронізації», який формує похідні (дочірні) тактові сигнали (рис. 1.39).

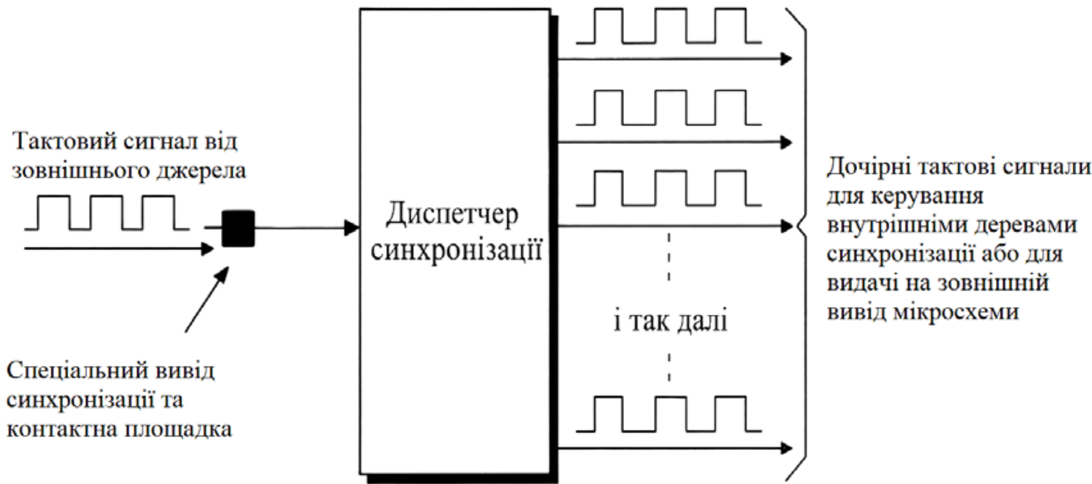


Рисунок 1.39 – Диспетчер синхронізації, що генерує дочірні тактові сигнали

Дочірні тактові сигнали можуть керувати внутрішніми деревами (доменами) тактування або подаватися на зовнішні виводи мікросхеми – для синхронізації інших вузлів на друкованій платі. У кожного сімейства ПЛІС є власна реалізація «диспетчера синхронізації», і в межах одного кристала зазвичай присутні кілька таких модулів. Залежно від типу, диспетчери можуть підтримувати всі або частину можливостей: придушення джитера (флуктуацій), синтез частоти, зсув фази та автоматичну корекцію фазового зсуву.

Усунення флуктуацій. Для простоти візьмімо тактову частоту 1 МГц (реально вона часто значно вища). В ідеалі кожний фронт імпульсу надходить рівно через одну мільйонну секунди після попереднього. У реальних умовах фронт може з’явитися трохи раніше або трохи пізніше. Щоб унаочнити цей ефект, який називають флуктуацією (джитером), розташуємо послідовні фронти один під одним: їхня суперпозиція виглядатиме як «розмазаний» у часі тактовий сигнал (рис. 1.40).

Диспетчер синхронізації в ПЛІС здатен виявляти й компенсувати такі флуктуації, «очищаючи» дочірні тактові сигнали перед їх використанням усередині кристала.

Частотний синтез. Буває, що зовнішня опорна частота, подана на ПЛІС, не збігається з потрібною для конкретного завдання. У такому разі диспетчер синхронізації генерує похідні такти, отримуючи їх діленням або множенням базового сигналу.

Для наочності розглянемо три дочірні тактові послідовності (рис. 1.41):

- із частотою, яка дорівнює початковій;
- із частотою, що дорівнює подвоєній початковій;
- із частотою, що становить половину від початкової.

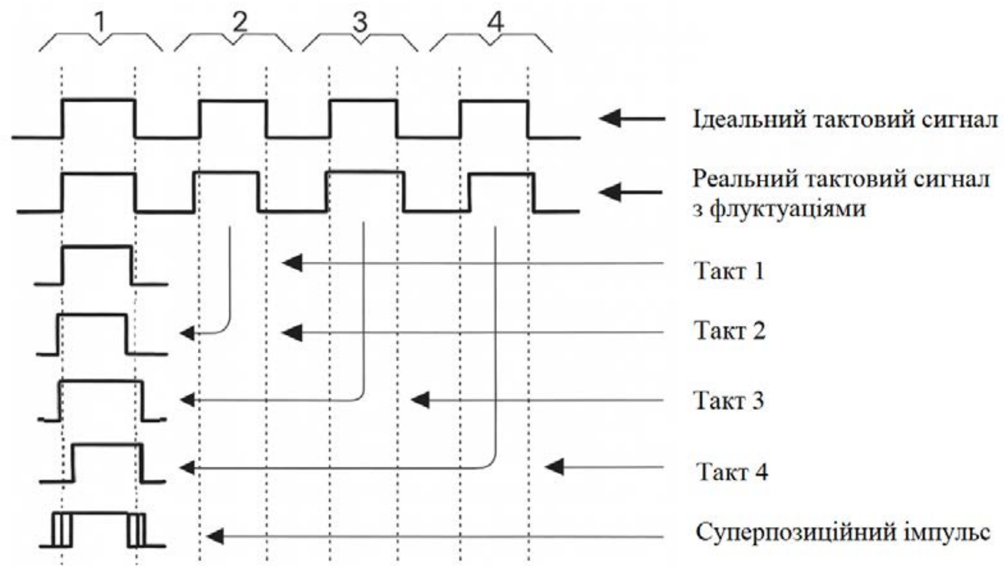


Рисунок 1.40 – Флуктуація як результат «змазування» тактового сигналу

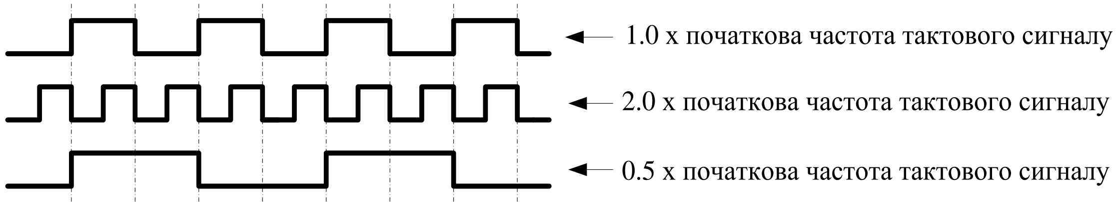


Рисунок 1.41 – Використання диспетчера синхронізації для частотного синтезу

На рис. 1.41 показано спрощений випадок. У реальних ПЛІС можна синтезувати найрізноманітніші внутрішні тактові частоти, наприклад 4/5 від базової.

Фазовий зсув. У низці систем потрібні тактові сигнали, зсунутих один відносно одного за фазою (із заданою затримкою). Частина диспетчерів синхронізації пропонує набір фіксованих зсувів, скажімо 120° і 240° для трифазних схем або 90°, 180° і 270° для чотирифазних. Інші дають змогу налаштовувати точне значення зсуву для кожного дочірнього такту за вимогою користувача.

Припустімо, з опорної послідовності формують чотири внутрішні такти: перший збігається за фазою з базовим сигналом, другий зсунений на 90°, третій – на 180°, і так далі (рис. 1.42).

Автокорекція зсуву. Для простоти вважатимемо, що дочірні тактові сигнали генеруються з тією ж частотою та фазою, що й опорний такт на вході ПЛІС. Насправді ж диспетчер синхронізації неминуче додає певну затримку. Додаткові затримки вносять і логічні елементи з внутрішніми з’єднаннями під час розведення такту.

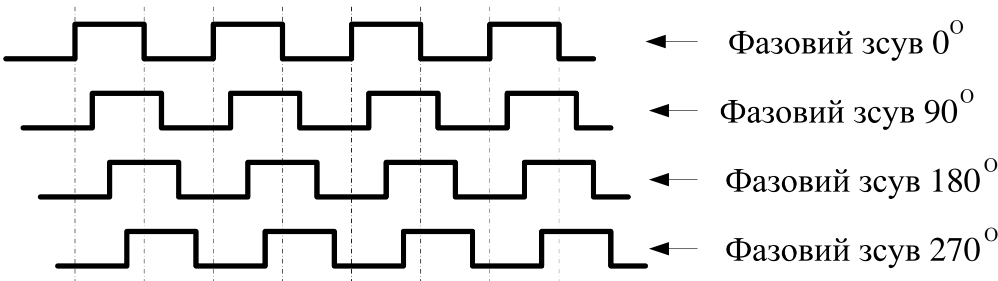


Рисунок 1.42 – Використання диспетчера синхронізації
для фазового зсуву дочірніх тактових сигналів

Без спеціальної компенсації дочірні такти відставатимуть від вхідного на деяку величину – це і є фазовий зсув. Оскільки взаємне використання головного й дочірніх тактових сигналів у ПЛІС та на платі може бути чутливим до такої різниці фаз, у диспетчері часто передбачають окремий вхід для зворотного підведення одного з дочірніх тактів. Порівнюючи його з опорним, вузол керування точно додає потрібну затримку до дочірнього сигналу, вирівнюючи його фазу з первинним. У результаті «очищається» саме базовий дочірній такт із нульовим зсувом, а решта похідних сигналів уже вирівнюються відносно нього.

Деякі тактові модулі ПЛІС будуються на основі фазового автопідстроювання частоти (ФАПЧ), інші – на цифрових петлях вирівнювання затримки (DLL, digital delay-locked loop). ФАПЧ можуть мати як аналогову, так і цифрову реалізацію, тоді як DLL за своєю суттю – виключно цифрові. Прихильники ФАПЧ відзначають кращу точність, стабільність, нижче енергоспоживання та меншу чутливість до шуму і джитера. Сучасні ПЛІС мають до тисячі й більше виводів, рознесених по периметру корпусу та підключених до кристала. Сам кристал часто монтують «догори ногами» (flip-chip), що дозволяє підводити землю, живлення, тактові та сигнальні лінії до будь-яких ділянок поверхні. Уявімо, що всі контакти кристала розміщені вздовж контуру так робили раніше [23].

З погляду інженерів друкованих плат, вибір інтерфейсу передачі даних залежить від завдання, типів компонентів і умов експлуатації. Під «стандартом» тут мають на увазі електричні рівні сигналів (наприклад, напруги логічних 0 та 1). Оскільки стандартів багато, робити окремі ПЛІС під кожен недоцільно. Тому лінії введення/виведення загального призначення в ПЛІС можна налаштувати на різні стандарти приймання та передавання сигналів. Ці виводи групують у кілька «банків». Для прикладу припустімо, що є вісім банків із номерами від 0 до 7 (рис. 1.43).

Кожен банк можна окремо налаштувати під конкретні стандарти інтерфейсів введення/виведення. Завдяки цьому ПЛІС здатна взаємодіяти з пристроями, що використовують різні електричні стандарти, а також

виступати «містком» між ними, забезпечуючи сумісність протоколів, які спираються на відмінні рівні сигналів.

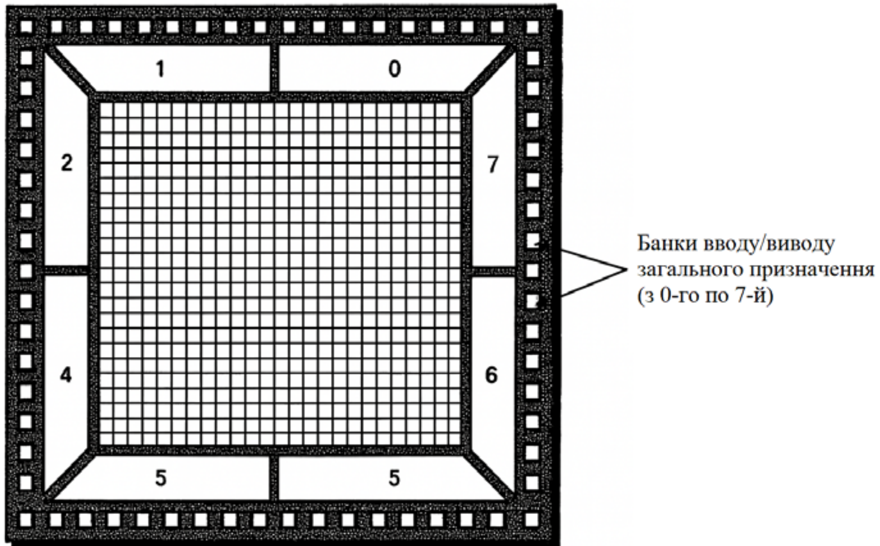


Рисунок 1.43 – Вид на кристал з банками вводу/виводу

У сучасних друкованих платах фронти сигналів дуже круті – тобто перехід між логічними рівнями відбувається за малий час. Щоб уникнути відбиттів і пов'язаних із цим коливань, на входах/виходах необхідно застосовувати узгоджувальні резистори (термінатори). Раніше такі резистори встановлювалися як дискретні елементи на платі поза ПЛІС. Але зі зростанням кількості виводів і зменшенням кроку між ними це стало незручним. Тому в сучасних ПЛІС передбачено вбудовані термінатори з налаштовуваним опором, що дозволяє узгоджувати лінії для різних інтерфейсів та стандартів без додаткових компонентів на платі. Історично (до приблизно 1995 року) більшість цифрових мікросхем працювали від 0 В (земля) та +5 В (живлення), а логічні рівні на I/O перемикалися між 0 В (логічний нуль) і +5 В (логічна одиниця). Зі зменшенням геометрії транзисторів – що робило їх дешевшими, швидшими й енергоефективнішими – знадобилося знижувати робочі напруги. Еволюцію напруг живлення відображено в табл. 1.2 [18].

Таблиця 1.2 – Залежність напруги живлення від технологічного процесу

Рік	Джерело живлення (напруга ядра) [В]	Технологічний процес [нм]
1998	3,3	350
1999	2,5	250
2000	1,8	180
2001	1,5	150
2003	1,2	130

Джерела живлення, наведені в цій таблиці, живлять внутрішню логіку ПЛІС, тож відповідну напругу називають напругою ядра (фактично для підведення живлення й «землі» використовують кілька контактів мікросхеми). Оскільки стандарти інтерфейсів введення/виведення оперують рівнями, що можуть суттєво відрізнятися від напруги ядра, кожен банк I/O зазвичай має окрему лінію живлення. Починаючи з техпроцесу 350 нм, значення напруги ядра зазвичай зменшувалося разом із переходом на тонші норми. Водночас існують фізичні обмеження, які не дозволяють опустити її набагато нижче ~1 В: це пов'язано з технологією транзисторів – порогами спрацювання та необхідними запасами по напрузі.

1.7 Методи та засоби програмування архітектури ПЛІС

У різних виробників ПЛІС – своя термінологія, підходи та інструменти. Способи програмування істотно відрізняються між окремими сімействами мікросхем. Тому далі розглядатимемо лише загальні принципи програмування ПЛІС.

Конфігураційні файли

Інструменти та методики для проєктування схем і побудови систем на ПЛІС дають змогу розробнику згенерувати конфігураційний (бітовий) файл – набір даних, який завантажується в мікросхему, щоб запрограмувати її на виконання потрібних функцій [6].

Коли ПЛІС зберігає конфігурацію в комірках статичного ОЗП, такий файл містить два типи інформації: самі конфігураційні біти, що визначають стани елементів програмованої логіки, і конфігураційні команди – інструкції, які підказують пристрою, як інтерпретувати та застосовувати ці біти. У процесі завантаження до мікросхеми цей потік даних часто називають конфігураційним двійковим потоком [2].

Пристрої на ЕСППЗП або Flash-прPam'яті програмуються подібним чином до ПЛІС зі статичною пам'яттю. Водночас у мікросхемах, що використовують технологію нарощуваних перемичок, конфігураційний файл містить лише конфігураційні дані, які безпосередньо керують формуванням цих перемичок.

Конфігураційні комірки

Базова ідея програмування ПЛІС доволі пряма: у мікросхему завантажують конфігураційний файл. Функціональність таких пристроїв визначається спеціальними конфігураційними комірками. У більшості ПЛІС це комірки статичного ОЗП; у деяких – ЕСППЗП чи Flash; є й варіанти на основі нарощуваних перемичок. Незалежно від технології,

всередині ПЛІС передбачено багато «зв'язувальних» комірок, за допомогою яких налаштовують маршрути: з'єднують зовнішні входи/виходи з логічними блоками та поєднують самі блоки між собою. Модулі введення/виведення теж мають набір таких комірок – для конфігурації під потрібні інтерфейсні стандарти та інші параметри. Припустімо, що кожен програмований логічний блок містить лише 4-входову таблицю істинності (LUT), мультиплексор і регістр (рис. 1.44).

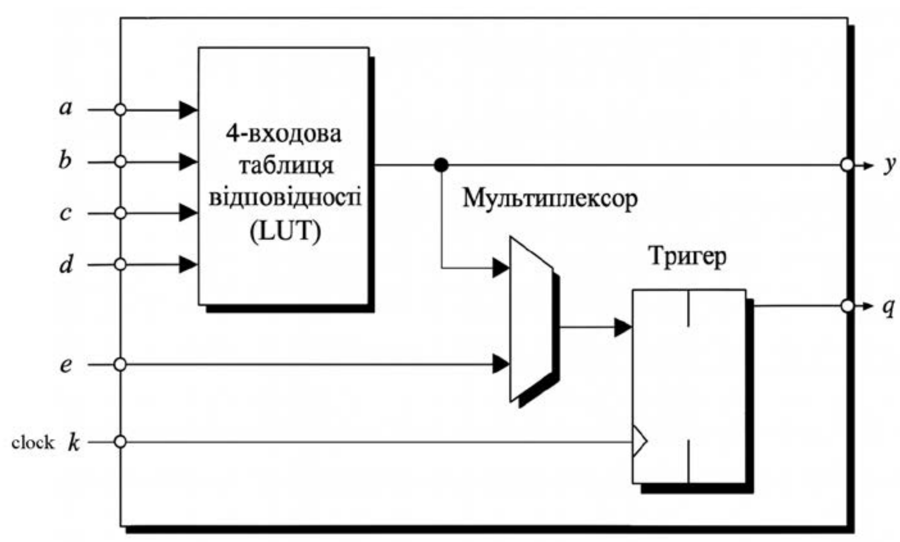


Рисунок 1.44 – Простий логічний блок, що програмується

Мультиплексору потрібна окрема конфігураційна комірка, яка задає, з якого входу брати сигнал на вихід. Для регістра необхідні конфігураційні комірки, що визначають, чи він буде:

- працювати як тригер чи як защіпка (клямка);
- перемикатися по передньому чи по задньому фронту тактового імпульсу (в режимі тригера) або за високим/низьким рівнем (у режимі клямки);
- ініціалізуватися значенням логічного 0 чи логічної 1.

Крім того, 4-входова таблиця відповідності (LUT) містить 16 конфігураційних комірок.

ПЛІС на нарощуваних перемичках

У ПЛІС із нарощуваними перемичками конфігураційні комірки зазвичай розміщені по всій площі кристала вздовж ключових напрямків. Для прошивання мікросхему встановлюють у спеціальний програматор, куди з комп'ютера завантажують бітовий (конфігураційний) файл. Далі програматор, керуючись цим файлом, подає на визначені виводи імпульси підвищеної напруги та струму, послідовно «нарощуючи» потрібні перемички.

Щоб інтуїтивно уявити процес, припустімо, що кожній перемичці відповідає пара віртуальних координат x–y на поверхні кристала, заданих числами. Одна група контактів мікросхеми подає значення x, інша – значення y. Насправді процедура складніша, але така модель добре передає її принцип [9].

Після формування всіх перемичок мікросхему знімають з програматора та монтують на друковану плату. Важливо: такі ПЛІС є одноразово програмованими – щойно процес стартував, змінити конфігурацію вже неможливо.

ПЛІС на комірках статичного ОЗП

Розгляньмо ПЛІС із конфігурацією на статичному ОЗП. Такі мікросхеми енергозалежні: їх прошивають прямо в системі (на платі) і щоразу після подачі живлення.

Уявно всі конфігураційні комірки SRAM можна звести до одного довжелезного регістра зсуву. Припустімо, що початок (вхід) і кінець (вихід) цього ланцюжка напряму з'єднані з зовнішніми контактами чипа – таке можливе в режимі послідовного завантаження через комунікаційний порт. У серійному режимі ПЛІС працює як «ведуча» (master) або як «ведена» (slave) [61]. Вихідні контакти конфігураційних даних зазвичай задіюють лише у разі каскадування кількох ПЛІС (послідовне опитування) або коли потрібно зчитати конфігурацію з пристрою.

ПЛІС на Flash (і ЕСППЗП) програмуються подібно до SRAM-пристроїв, але є енергонезалежними: вони зберігають конфігурацію без живлення, тож повторне прошивання під час ввімкнення не обов'язкове (хоч за потреби можливе). Такі мікросхеми можна прошивати як на платі, так і окремим програматором.

Найпростіше уявити конфігураційні комірки SRAM як довгий регістр зсуву на тригерах, з'єднаних у ланцюг та синхронізованих спільним тактом. Проблема в тому, що кількість комірок гігантська: ще у 2003 році топові ПЛІС містили до 25 млн конфігураційних комірок. Оскільки тригер потребує 8 транзисторів, а клямка – лише 4, у практиці конфігураційні комірки виконують на клямках. Для 25 млн комірок це економить близько 100 млн транзисторів [13].

Втім, з клямок не збудуєш наддовгий регістр зсуву. Постачальники вирішують це так: формують «фрейм» – групу, скажімо, з 1024 тригерів під спільним тактом, що працює як класичний регістр зсуву. Всі 25 млн клямкових комірок також ділять на фрейми, кожен завдовжки як тригерний фрейм. Ззовні виглядає, ніби в ПЛІС послідовно вливають 25 млн бітів. Усередині ж після завантаження перших 1024 бітів у тригерний фрейм спеціальна логіка паралельно копіює ці біти в перший клямковий фрейм; наступні 1024 біти – у другий, і так далі, доки не

заповнять увесь чип. Під час зчитування все відбувається у зворотній послідовності.

Час конфігурації може бути відчутним: для ПЛІС із 25 млн SRAM-комірок послідовне завантаження на тактовій частоті 25 МГц триватиме приблизно одну секунду.

Програмування вбудованих блоків розподіленого ОЗП та інших ОЗП

Якщо в ПЛІС є великі вбудовані блоки пам'яті, їхні осердя будують на клямках, реалізованих на комірках статичної пам'яті. Кожна така клямка водночас виступає конфігураційною коміркою й входить до уявного «ланцюжка» регістра зсуву, про який ішлося раніше.

4-входову LUT можна налаштувати в один із трьох режимів: як власне таблицю відповідності, як невеликий сегмент розподіленої пам'яті 16×1, або як 16-бітовий регістр зсуву. У кожному з цих випадків задіюється набір із 16 клямок SRAM, і кожна з них – це конфігураційна комірка з того самого регістрового кола [32].

Фішка схеми в тому, що тут застосовують ємнісні клямки, які за низкою характеристик випереджають класичні варіанти. Подібні принципи свого часу використовували інженери, створюючи зовнішні тригери з дискретних транзисторів, резисторів і конденсаторів ще на початку 1960-х років.

Мультипрограмування конфігураційних ланцюжків

Оскільки число конфігураційних комірок може сягати десятків мільйонів, відповідний ланцюг виходить надзвичайно довгим. У низці ПЛІС його ділять на сегменти, і саме ці окремі відрізки під'єднують до конфігураційного порту. Це дає змогу налаштовувати частини кристала незалежно та полегшує впровадження підходів на кшталт модульного чи поетапного проєктування [39].

Регістри в програмованих логічних блоках пов'язані з конфігураційними комірками, де зберігаються їхні початкові стани (логічний 0 або 1). Кожне сімейство ПЛІС має свій механізм – наприклад, окремий вивід ініціалізації, який, коли його активувати, повертає регістри до заданих початкових значень. Зверніть увагу: такий механізм не ініціалізує ні вбудовані блоки пам'яті, ні розподілене ОЗП.

Конфігураційний порт

Перші ПЛІС налаштовували через так званий конфігураційний порт. Навіть нині, попри наявність складніших підходів на кшталт JTAG, цей порт лишається популярним завдяки простоті.

Почнемо з невеликого набору спеціальних контактів, що задають режим конфігурації мікросхеми. У ранніх ПЛІС для цього було всього два виводи, які дозволяли вибрати один із чотирьох режимів (табл. 1.3) [5].

Таблиця 1.3 – Чотири первинних режими ініціалізації

Виводи режиму установлення	Найменування режиму
00	Послідовне завантаження, ПЛІС у режимі «ведучий»
01	Послідовне завантаження, ПЛІС у режимі «ведений»
10	Паралельне завантаження, ПЛІС у режимі «ведучий»
11	Паралельне завантаження, ПЛІС у режимі «ведений»

Варто пам’ятати, що назви режимів і відповідність кодів на контактних лініях наведені лише як приклад – у кожного виробника свої позначення та коди. Контакти вибору режиму зазвичай фіксують на платі в станах логічного 0 або 1. Теоретично їх можна під’єднати до зовнішньої логіки для динамічної зміни режиму програмування, але на практиці так роблять рідко.

Окрім ліній вибору режиму, є окремий вивід, який сповіщає ПЛІС про старт конфігурації, і ще один – що сигналізує про її завершення. Передбачені й сигнали індикації помилок під час прошивання. За потреби ПЛІС можна повторно ініціалізувати, повернувши початкові конфігураційні дані [7].

Конфігураційний порт також задіює додаткові контакти для керування процесом завантаження й приймання даних; їхня кількість залежить від вибраного режиму. Важливо, що після завершення конфігурації більшість цих ліній може бути перевикористана як універсальні входи/виходи (GPIO).

Послідовне завантаження, ПЛІС у режимі «ведучий»

Це найпростіший із режимів прошивання. Колись для нього застосовували зовнішні ППЗП, згодом – СППЗП, потім ЕСППЗП, а нині зазвичай – мікросхеми на flash-пам’яті. Такі спеціалізовані пам’яті мають один вихід даних, який під’єднують до входу конфігураційних даних ПЛІС (рис. 1.45).

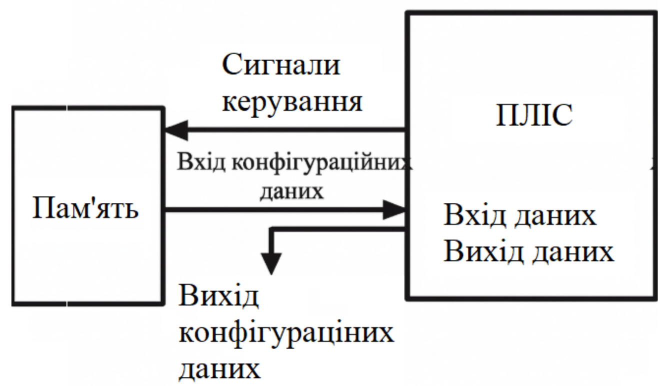


Рисунок 1.45 – Послідовне завантаження,
ПЛІС у режимі «ведучий»

ПЛІС також формує кілька керувальних сигналів для зовнішньої пам'яті. Серед них – сигнал скидання, який мікросхема подає, коли готова розпочати зчитування, та тактовий сигнал, що синхронізує передавання конфігураційних даних. У цьому режимі не потрібна пам'ять із послідовною адресацією: ПЛІС подає простий імпульс скидання, щоб почати читання з початку конфігураційної послідовності, а далі керує видачею даних серією тактових імпульсів.

За потреби вихідний канал ПЛІС можна використати для зчитування її конфігураційних даних. Це актуально, коли на платі встановлено кілька ПЛІС: кожну можна конфігурувати з окремої пам'яті незалежно (рис. 1.45) або з'єднати мікросхеми в каскад і спільно використовувати один зовнішній носій (рис. 1.46).

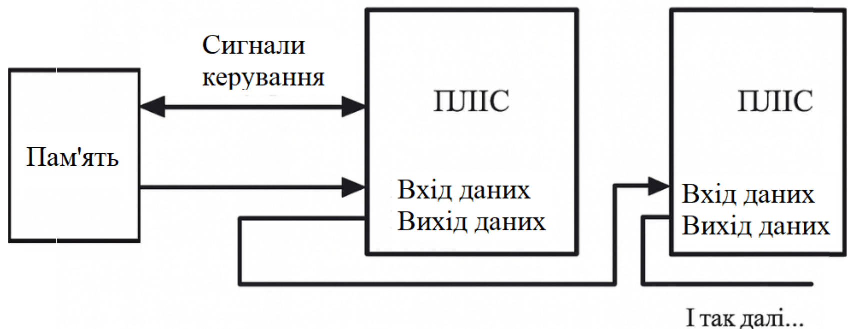


Рисунок 1.46 – Послідовне включення мікросхем FPGA

У конфігурації з рис. 1.46 перша мікросхема ПЛІС, що безпосередньо під'єднана до зовнішньої пам'яті, працює в послідовному режимі як «ведуча» (master). Усі наступні ПЛІС у цьому ланцюжку мають бути налаштовані на послідовний режим «ведених» (slave).

Паралельне завантаження, ПЛІС у режимі «ведучий»

У багатьох аспектах цей режим схожий на попередній, але дані з пам'яті зчитуються вже 8-бітовими порціями, причому сама пам'ять має вісім ліній даних. Такі групи з восьми бітів широко відомі як байти.

Окрім керувальних сигналів, ПЛІС подає на зовнішню пам'ять адресні сигнали, які вказують, який саме байт конфігурації потрібно завантажити на наступному такті (рис. 1.47) [25].

Робота цієї схеми передбачає наявність у ПЛІС внутрішнього лічильника, що генерує адреси для пам'яті. У ранніх моделях це були 24-бітові лічильники, які давали змогу адресувати до 16 мільйонів байтів. На старті конфігурації лічильник обнулявся; після зчитування кожного байта значення лічильника інкрементувалося для вибору наступної адреси. Процес триває, доки вся конфігурація не буде повністю завантажена в мікросхему.

Рисунок 1.47 – Паралельне завантаження в режимі «ведучий» (перший спосіб)

Здавалося б, паралельне завантаження має бути швидшим за послідовне. Та спершу це не працювало: у ранніх ПЛІС зчитаний байт усе одно подавався далі послідовно у внутрішній конфігураційний регістр зсуву. У сучасних чипах цю недоліковість усунули.

Нині спеціалізовані зовнішні пам'яті для ПЛІС – відносно дешеві, виконані за Flash-технологією й придатні до багаторазового перепрограмування. Нові ПЛІС застосовують удосконалені схеми паралельного завантаження: використовують пам'ять, яка не потребує зовнішньої адресації, тож внутрішній лічильник адрес у ПЛІС більше не потрібен (рис. 1.48).

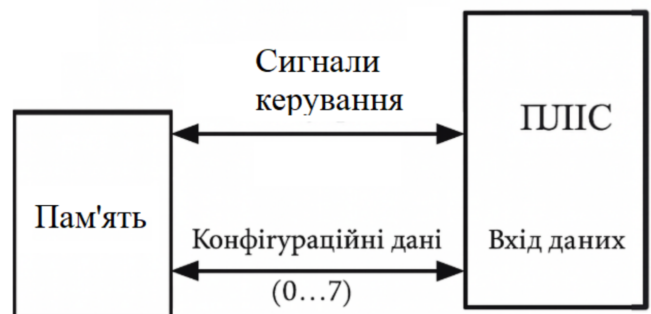


Рисунок 1.48 - Паралельне завантаження в режимі «ведучий» (сучасна реалізація)

У цій схемі, подібно до послідовного режиму, ПЛІС подає імпульс скидання на зовнішню пам’ять, сигналізуючи про готовність почати зчитування від самого початку конфігураційної послідовності, а далі генерує тактові імпульси, якими синхронізує приймання конфігураційних даних із пам’яті.

Паралельне завантаження, ПЛІС у режимі «ведений»

Режими, у яких ПЛІС працює як «ведуча» (master), привабливі своєю простотою та невибагливістю: фактично достатньо самої ПЛІС і однієї мікросхеми зовнішньої пам’яті.

Втім, на багатьох платах уже присутні мікропроцесори, що виконують різні внутрішні функції. У таких випадках розробники часто вибирають саме мікропроцесор для завантаження конфігураційних даних у ПЛІС (рис. 1.49).

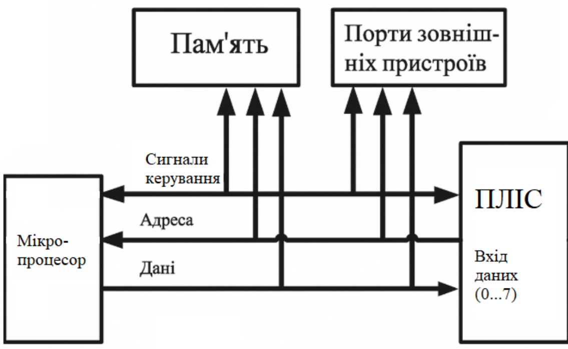


Рисунок 1.49 – Паралельне завантаження у режимі «ведений»

Суть підходу в тому, що керує процесом мікропроцесор. У цьому режимі він спершу сповіщає ПЛІС про готовність до конфігурації, далі зчитує з визначеного джерела – пам’яті, зовнішнього пристрою чи іншого обладнання – байт конфігураційних даних, записує його в ПЛІС, потім бере наступний байт і знову завантажує – і так до завершення прошивання [31].

Переваг у такого рішення кілька. Зокрема, мікропроцесор може опитувати обладнання сусідніх систем і, за заданими критеріями, вибирати, який саме конфігураційний набір потрібно завантажити в ПЛІС.

Послідовне завантаження, ПЛІС у режимі «ведений»

Цей спосіб подібний до паралельного, але дані подаються в ПЛІС поодинці – по одному біту. Мікропроцесор, як і раніше, зчитує з пам’яті байт, після чого перетворює його на послідовність бітів і послідовно завантажує в ПЛІС.

Головна перевага – менша потреба у контактів мікросхеми: для конфігурації достатньо лише одної лінії введення/виведення, яку через додатковий провід під'єднують до шини даних мікропроцесора.

JTAG-порт

Багато сучасних мікросхем, зокрема ПЛІС, оснащені інтерфейсом JTAG – це порт, стандартизований як IEEE 1149.1 (ініціатива Joint Test Automation Group), спочатку створений для периферійного сканування під час тестування плат і цифрових ІС.

У нашому випадку ПЛІС має кілька контактів під JTAG: один слугує для приймання даних, інший – для їх виведення, решта керують послідовними JTAG-регістровими ланцюжками (тригерами). Суть сканування така: через JTAG-порт послідовно подають дані у JTAG-регістри, пов'язані з входами мікросхеми; ПЛІС опрацьовує ці дані, записує результати у регістри, прив'язані до виходів, після чого результати послідовно зчитують назад через JTAG [15].

Пристрої з JTAG містять також допоміжну керувальну логіку. У ПЛІС цей порт, окрім сканування, здатен виконувати додаткові дії – наприклад, приймати спеціальні інструкції, що завантажуються у командний JTAG-регістр. Одна з таких команд під'єднує внутрішній конфігураційний регістр зсуву до JTAG-ланцюжка, завдяки чому JTAG можна використати для прошивання ПЛІС. Тож сучасні ПЛІС підтримують уже п'ять режимів програмування, і для їх вибору потрібні три лінії вибору режиму (табл. 1.4; у майбутньому можливе розширення).

Таблиця 1.4 – П'five сучасних конфігураційних режимів

Виводи режиму установлення	Найменування режиму
000	Послідовне завантаження, ПЛІС у режимі «ведучий»
001	Послідовне завантаження, ПЛІС у режимі «ведений»
010	Паралельне завантаження, ПЛІС у режимі «ведучий»
011	Паралельне завантаження, ПЛІС у режимі «ведений»
1xx	Використовується тільки JTAG-порт

Варто підкреслити, що JTAG-порт доступний завжди: спочатку пристрій можна конфігурувати через звичайний конфігураційний порт, обравши один зі стандартних режимів, а за необхідності — перезапрограмувати через JTAG. Також можливе повне програмування виключно через JTAG-інтерфейс.

1.8 Тенденції розвитку сучасних ПЛІС

Нещодавно Intel почала відвантаження сімейства Agilex; Xilinx ще в червні надала перші Versal ACAP; Achronix анонсувала зразки Speedster 7t до кінця року. Отже, на ринку з'являються три різні за підходом платформи верхнього класу з передовими техпроцесами й помітними унікальними можливостями.

Драйверами попиту є:

- 5G розгортається дедалі швидше. Більшість ранніх вузлів 5G побудовані на програмованій логіці попередніх поколінь, а нові ПЛІС ще більше прискорять впровадження – трафік мобільного абонента з великою ймовірністю проходить через декілька FPGA [14];
- ШІ/ML – другий великий напрям: усі три виробники роблять акцент на прискоренні нейромереж [16].

Техпроцеси Xilinx і Achronix використовують TSMC 7 нм, Intel – власний 10 нм для Agilex. Формальна різниця «7 проти 10» не є вирішальною: за якістю та метриками ці вузли вважають зіставними. Водночас виграші від кожного нового переходу за законом Мура зменшуються, а вартість розробки зростає, тож усе більшого значення набувають архітектура, інструменти САПР і стратегія.

Логіка (LUT) і місткість. Виробники рахують ресурси по-різному, тому часто переводять у «еквівалентні 4-входові LUT»:

- 1×LUT6 ≈ 2,2 LUT4; 1×LUT8 ≈ 2,99 LUT4.
- Achronix Speedster 7t: ~363 тис.–2,6 млн LUT6 (≈ 800 тис.–5,76 млн LUT4).
- Intel Agilex: 132–912 тис. ALM (≈ 395 тис.–2,7 млн LUT4).
- Xilinx Versal: ~246–984 тис. CLB (≈ 541 тис.–2,2 млн LUT4).

Реальна складність проєкту визначається не лише кількістю LUT, а й тим, наскільки інструменти здатні розмістити й розвести велику частку цих ресурсів.

Перевага FPGA в ML – масовий паралелізм MAC-операцій:

- Achronix: до 41 тис. мультиплікаторів для int8 (або 82 тис. для int4).
- Intel Agilex: ~2–17 тис. множників 18×19.
- Xilinx Versal: близько 0,5–3 тис. множників у складі DSP58 (27×24) з новою апаратною FP-підтримкою.

Щодо плаваючої коми, то всі троє посилили FP:

- Achronix запровадила MLP (Machine Learning Processor): до 32 MAC із цілими числами 4–24 біт та режимами FP (зокрема bfloat16). MLP розміщено поряд із пам'яттю, що дає ~750 МГц без очікування даних.
- Intel розвиває DSP зі змінною точністю і підтримкою FP16/bfloat16, маючи одну з найбільш зрілих FP-екосистем.
- Xilinx перейшла від DSP48 до DSP58 (27×24) з апаратною FP; також додала програмований векторний процесор — матрицю до ~400

VLIW-SIMD ядер із локальною пам'яттю (>1 ГГц), що спрощує використання великих паралельних обчислень у стилі GPU. [19]

Підтримувані формати (макс. орієнтовні кількості):

- FP32: Versal — до ~2,1 тис. множників; Agilex — до ~8,7 тис.
- FP16: усі три — Versal ~2,1 тис., Agilex ~17,1 тис., Speedster ~5,1 тис.
- bfloat16: Agilex ~17,1 тис., Speedster ~5,1 тис.
- FP24: Speedster — до ~2,6 тис. (Versal/Agilex імовірно використовують FP32-блоки).
- Block-floating: Speedster — до ~81,9 тис. множників.

Теоретична продуктивність (індикативно):

- INT8 TOPS: Versal ~171 (≈133 від векторного процесора + 12 від DSP + 26 від логіки); Speedster ~86 (≈61 MLP + 25 логіка); Agilex ~92 (≈51 DSP + 41 логіка).
- bfloat16 TFLOPS: Agilex ~40, Versal ~9, Speedster ~8.
- Block-floating TFLOPS: Speedster ~123, Agilex ~41, Versal ~15.

Ці значення – теоретичні вершини; у практиці зазвичай досягають частку від максимуму (орієнтовно 50–90 % у FPGA).

Маршрутизація й затримки. Зі зростанням масштабів ПЛІС складність P&R різко підвищується.

- Xilinx і Achronix інтегрують NoC (мережу на кристалі) поверх традиційної фабрики логіки, розбиваючи систему на менші синхронні домени й розвантажуючи звичайну маршрутизацію.
- Intel давно застосовує HyperFlex-регістри (друге покоління в Agilex) для агресивної ретаймінг/конвеєризації, досягаючи подібної мети іншим шляхом.

Пропускна здатність NoC (приклади):

- Achronix: 2-вимірний AXI-NoC; у кожному рядку/стовпці по два 256-бітні однонапрямні канали на 2 ГГц (≈512 Гбіт/с у кожен бік), загалом ~197 портів підключення й сумарно ≈27 Тбіт/с.
- Xilinx Versal: точні числа не оприлюднені; за оцінками з ~28 портами це може бути близько 1,5 Тбіт/с.

Більшість ключових провідних виробників ПЛІС базуються у США: Achronix, Actel (нині Microsemi), Altera (Intel), Atmel (нині Microchip), Lattice, Xilinx; TSMC (Тайвань) – контрактний виробник кремнію й лідер у просунутих вузлах (≈90 нм → 5 нм і далі).

Xilinx традиційно очолювала ринок (історично понад 50 %). Продуктові лінійки:

- Virtex – флагман із високошвидкісними трансиверами, PCIe, Ethernet MAC, DSP, FIFO, ECC;
- Kintex – баланс ціни/продуктивності;
- Artix – енергоефективний «молодший» сегмент;
- Spartan – масовий доступний клас;

- Zynq-7000 – SoC (FPGA + ARM Cortex-A9) із підтримкою HLS для високорівневого проєктування;
- EasyPath – швидке виведення дизайну в серію;
- Versal – 7-нм архітектура для гетерогенних обчислень, ІІ та вбудованих систем.

У 2020 р. AMD повідомила про угоду щодо купівлі Xilinx.

Altera (Intel) – другий за масштабом постачальник; із 2015 р. входить до Intel. Випускає FPGA та SoC-FPGA (ARM), використовує вузли 40/28/14 нм, розробляє PowerSoC (вбудовані DC-DC для високої щільності та низького шуму), IP-ядра й замовні рішення. Більшість їхніх ПЛІС – SRAM-конфігуровані (після знеструмлення потрібне перезавантаження).

Achronix – високошвидкісні FPGA (частоти до ~1,5 ГГц), зокрема радіаційностійкі варіанти; виробництво – контрактне (раніше на потужностях Intel).

Actel → Microsemi – історично відома нерухомо-конфігурована/флеш-FPGA (не втрачає прошивку без живлення).

Atmel → Microchip – виробник MCU, флеш-пам'яті та FPGA; разом з Actel відомий flash-подібними ПЛІС.

Lattice – енергоощадні й компактні FPGA для інтерфейсів, відео та «краєвих» застосунків.

Отже, нинішній ринок ПЛІС рухають ШІ та 5G, тож апаратні обчислювальні блоки й підтримка операцій з плаваючою комою стають стандартом. Через зростання масштабів ресурсів розміщення й трасування дедалі частіше перетворюються на вузьке місце, тому потрібні мережі на кристалі (NoC), механізми на кшталт HyperFlex і потужні САПР. Перехід на передові техпроцеси підвищує вартість, тож вирішальними дедалі більше стають архітектура та інструментарій, а не сам вузол. На цьому тлі попит на FPGA зростає, і ринку потрібні фахівці, які однаково впевнено працюють і на рівні HDL/RTL, і з високорівневими інструментами для ШІ та цифрової обробки сигналів.

1.9 Архітектура ПЛІС компаній Altera та Xilinx

ПЛІС за способом зберігання конфігурації поділяються на SRAM-based, Flash-based та Antifuse-based ПЛІСи.

SRAM-based ПЛІС – це один із найпоширеніших типів програмованих логічних інтегральних схем. У таких пристроях інформація про конфігурацію зберігається в комірках статичної пам'яті (SRAM), які виготовляються за стандартною CMOS-технологією. Основною перевагою цього типу є можливість багаторазового перепрограмування мікросхеми. Це дає змогу змінювати конфігурацію логіки необмежену кількість разів під час розробки або експлуатації. Водночас є і певні недоліки: швидкодія таких схем не є максимальною, а після вимкнення живлення конфігурація втрачається. Тому після кожного ввімкнення пристрою необхідно знову

завантажувати програму конфігурації. Для цього на платі зазвичай використовують додатковий пристрій – наприклад, мікросхему FLASH-пам’яті або мікроконтролер, що дещо підвищує вартість кінцевого виробу.

У мікросхемах Flash-based ПЛІС дані конфігурації зберігаються у вбудованій пам’яті FLASH або EEPROM. Головна перевага таких ПЛІС полягає в тому, що інформація не зникає після відключення живлення. Після подачі живлення пристрій одразу готовий до роботи без додаткового завантаження конфігурації. Однак ця технологія має й певні обмеження. Реалізація FLASH-пам’яті всередині CMOS-мікросхеми є технологічно складною, оскільки необхідно поєднати два різні виробничі процеси. Через це такі мікросхеми зазвичай коштують дорожче. Крім того, пам’ять FLASH має обмежену кількість циклів перепрограмування, що також слід враховувати під час використання.

Antifuse-based ПЛІС – це спеціальний тип програмованих логічних схем, які програмуються лише один раз. Процес конфігурації полягає у створенні постійних електричних з’єднань усередині кристала шляхом руйнування спеціальних перемичок (antifuse). Внаслідок цього формується необхідна логічна структура. Основний недолік такого підходу полягає в тому, що після програмування змінити конфігурацію вже неможливо. Крім того, сама процедура програмування може займати певний час. Водночас ці мікросхеми мають важливі переваги: вони здатні працювати на високих частотах, характеризуються високою швидкодією та підвищеною стійкістю до радіаційних впливів. Це пояснюється тим, що конфігурація реалізується у вигляді фізичних з’єднань, а не додаткової логіки пам’яті, як у ПЛІС типу SRAM [42].

У документації компанії Altera використовується термін Logic Array Block (LAB), що перекладається як масив логіки. У мікросхемах ПЛІС компанії Xilinx застосовується схоже поняття – Configurable Logic Block (CLB). Конфігурований логічний блок є основною структурною одиницею програмованих логічних інтегральних схем. Усередині такого блока можуть реалізовуватися прості логічні функції або зберігатися результати обчислень у регістрах (тригерах).

Будова та складність конфігурованого логічного блока визначаються виробником конкретної ПЛІС. Теоретично такий блок може мати різний рівень складності. У найпростішому випадку він може відповідати дуже елементарному компоненту, наприклад одному транзистору. У протилежному випадку блок може бути настільки складним, що за своєю структурою нагадуватиме цілий процесор. Це два крайні варіанти реалізації [44].

Якщо використовувати дуже прості елементи, то для побудови необхідної цифрової схеми доведеться застосовувати велику кількість програмованих з’єднань між ними. Якщо ж блок буде надто складним, кількість з’єднань може зменшитися, проте знизиться гнучкість під час створення користувацьких схем.

Саме тому у більшості сучасних ПЛІС конфігуровані логічні блоки мають середній рівень складності. Вони достатньо функціональні, щоб реалізовувати певні логічні операції, але водночас досить компактні, щоб на одному кристалі можна було розмістити велику кількість таких блоків і поєднати їх між собою у складну цифрову систему.

Таким чином, під час проєктування архітектури ПЛІС виробники змушені знаходити баланс між різними характеристиками: площею кристала, швидкодією, енергоспоживанням та гнучкістю конфігурації.

Конфігурований логічний блок може містити один або кілька базових логічних елементів. В англомовній літературі їх називають Basic Logic Element (BLE) або просто Logic Element (LE). У більшості сучасних ПЛІС такі базові елементи будуються на основі LUT (Look-Up Table), тобто таблиць пошуку, які дозволяють реалізовувати різні логічні функції шляхом зберігання відповідних значень у пам'яті. На рис. 1.50 показаний приклад традиційного базового логічного елемента.

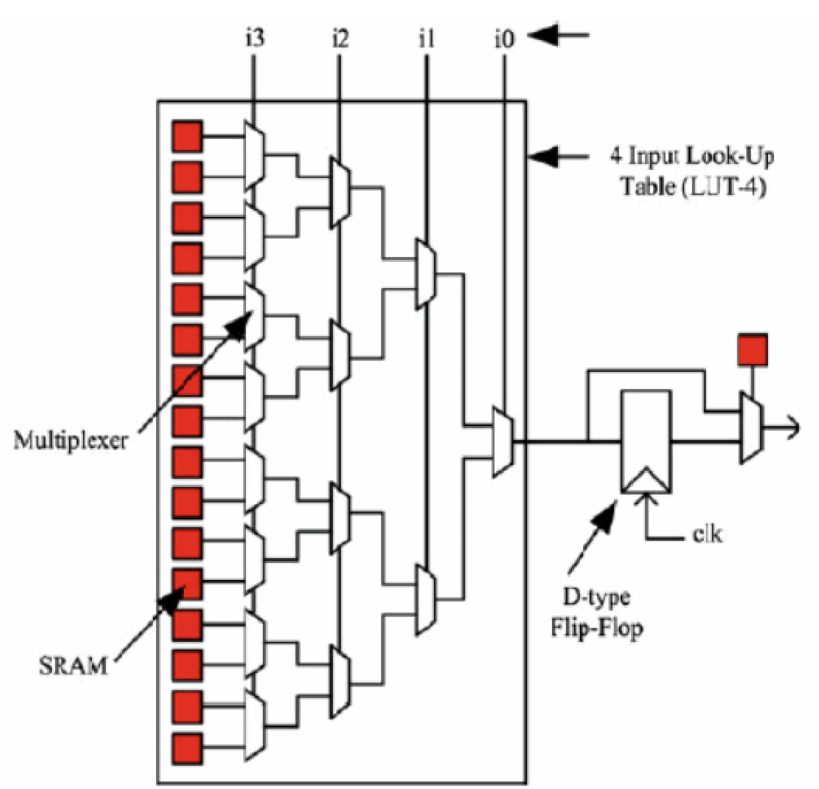


Рисунок 1.50 – Приклад традиційного базового логічного елемента

LUT (Look-Up Table) – це таблиця відповідності або таблиця пошуку, яка використовується для реалізації логічних функцій у ПЛІС. Принцип її роботи полягає в тому, що для певної комбінації вхідних сигналів у пам'яті заздалегідь зберігається відповідний вихідний результат [38].

Наприклад, на рис. 1.50 показано чотирибітний LUT, який входить до складу базового логічного елемента. У цьому випадку чотирибітному набору сигналів на вході відповідає один біт на виході. Червоні квадратики

на схемі позначають програмовані елементи, тобто конфігураційні регістри. Саме в цих комірках пам’яті зберігається конфігурація ПЛІС, яка визначає роботу логічної функції.

Для реалізації LUT з чотирма входами необхідно 16 конфігураційних регістрів, оскільки кількість можливих комбінацій входів дорівнює 16. Значення, записані в цих регістрах, фактично задають логічну функцію, яка реалізується всередині базового логічного елемента.

Крім того, використовується ще один конфігураційний регістр (на схемі він позначений окремим червоним квадратом). Він визначає, який сигнал подавати на вихід: безпосередньо результат із LUT або значення, зафіксоване у D-тригері. Використання тригерів у цифрових схемах є важливим, оскільки вони забезпечують збереження та синхронізацію даних.

Якщо розглядати таку структуру як приклад типового базового логічного елемента, стає помітною значна надлишковість архітектури сучасних ПЛІС, особливо тих, що базуються на SRAM. Конфігураційні регістри, показані на схемі, не використовуються безпосередньо в користувацькій цифровій схемі. Вони потрібні лише для зберігання конфігураційної інформації, яка визначає роботу логіки. Таким чином, для реалізації одного D-тригера у користувацькому проєкті може використовуватися більше ніж шістнадцять тригерів, що зберігають конфігурацію самої ПЛІС [33].

На рис. 1.51 зображено базовий логічний елемент CPLD MAX II компанії Altera. Тут добре показано LUT і D-тригер збереження результату. На рис. 1.52 подано базовий логічний елемент FPGA Cyclone III (Altera). В мікросхемах Altera в одному LAB може міститись 10–16 LE.

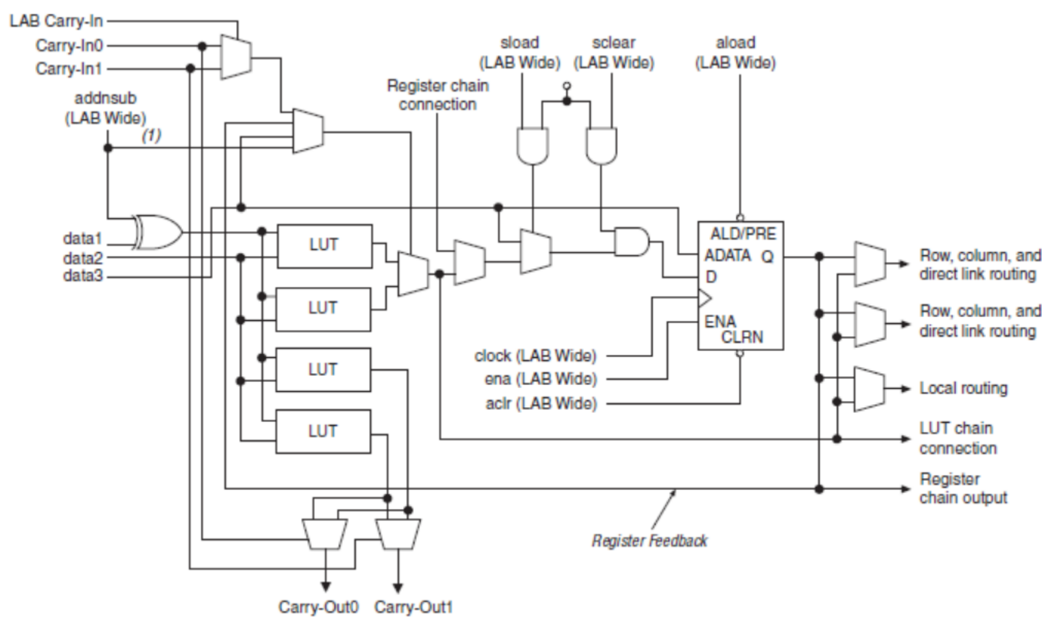


Рисунок 1.51 – Базовий логічний елемент
CPLD MAX II компанії Altera

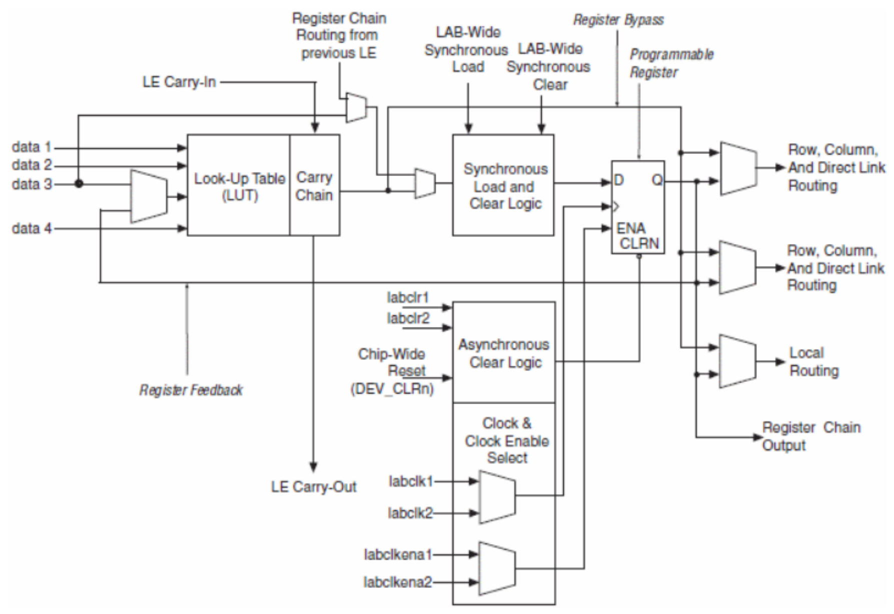


Рисунок 1.52 – Базовий логічний елемент
FPGA Cyclone III компанії Altera

В мікросхемах компанії Xilinx Virtex-6 базовий логічний елемент – це так званий Slice. В одному CLB всього два Slice. Один Slice – це дуже складний пристрій, який зображено на рис. 1.53 [27].

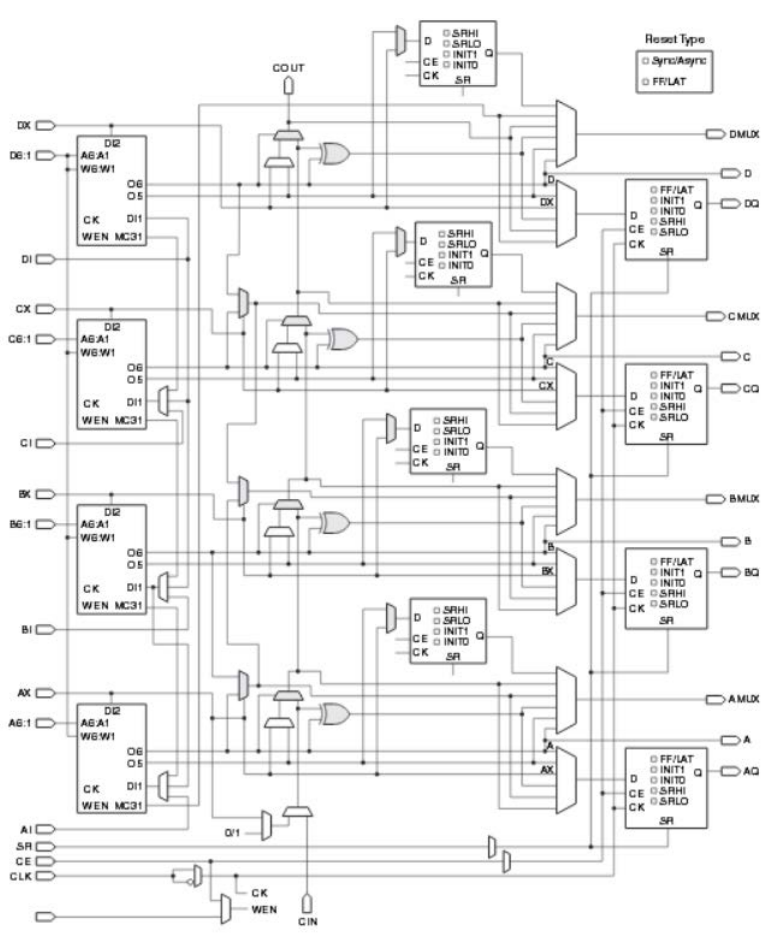


Рисунок 1.53 – Базовий елемент Xilinx Virtex-6 Slice

Насправді базові логічні елементи у різних типах ПЛІС зазвичай мають значно складнішу структуру, ніж наведено на спрощеній схемі. У технічній документації різних виробників можна знайти приклади архітектур логічних елементів із додатковими функціями та розширеними можливостями.

Наприклад, у конфігурованому логічному блоці CLB мікросхем серії Virtex-6 міститься 8 LUT-таблиць, 16 D-тригерів, а також низка додаткових елементів. Така структура демонструє досить складну архітектуру логічного блока [14].

Для реалізації цифрової схеми в ПЛІС недостатньо лише налаштувати окремі логічні блоки. Необхідно також створити зв’язки між ними, щоб сигнали могли передаватися від одного блока до іншого.

Саме для цього в архітектурі ПЛІС передбачено конфігуровані комутаційні елементи, які дозволяють програмно визначати маршрути сигналів.

У технічній літературі англійською мовою такі системи з’єднань часто називають FPGA Routing Architecture або Programmable Routing Interconnect. Обидва терміни описують структуру програмованих міжз’єднань у ПЛІС.

Загалом існують дві основні архітектури організації таких з’єднань:

- острівна архітектура (island-style architecture);
- ієрархічна архітектура (hierarchical architecture).

Кожен із цих підходів має свої особливості, які впливають на швидкодію, гнучкість конфігурації та ефективність використання ресурсів ПЛІС. На рис. 1.54 зображено архітектуру острівної ПЛІС

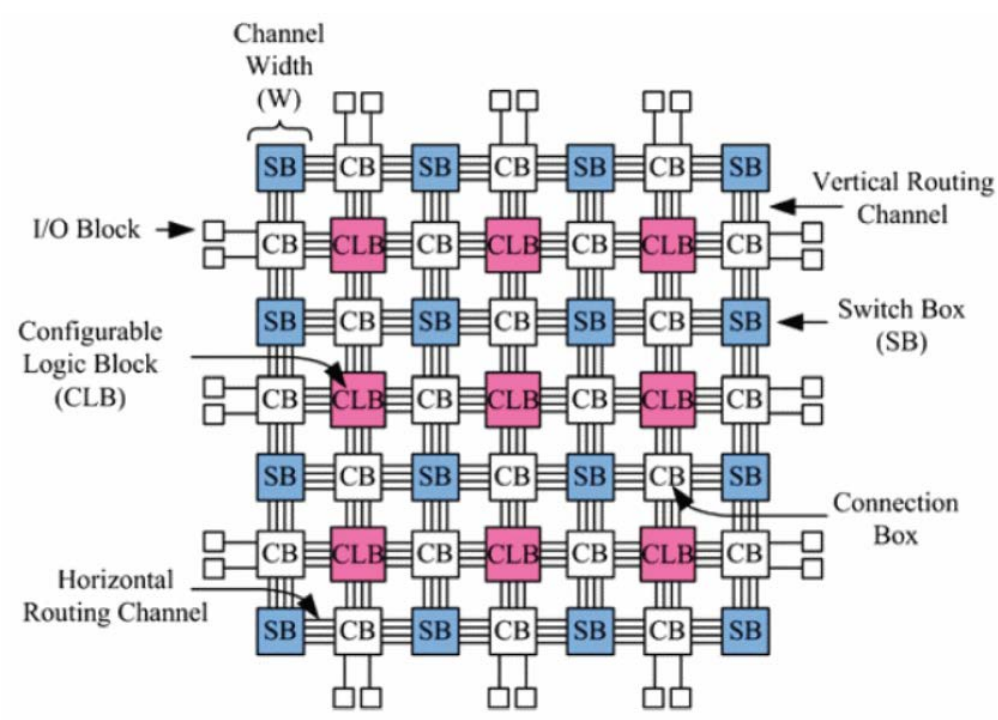


Рисунок 1.54 – Острівна ПЛІС

Острівна архітектура ПЛІС

Острівна архітектура отримала свою назву через особливості розташування конфігурованих логічних блоків. У такій структурі всі логічні блоки є однаковими за функціональністю і розміщуються на кристалі подібно до «островів», які оточені мережею комутаційних вузлів і ліній зв’язку [9].

На схемах такої архітектури зазвичай позначаються елементи CB (Connection Box) і SB (Switch Box). Фактично, це програмовані комутаційні вузли або мультиплексори, які забезпечують з’єднання одного конфігурованого логічного блока (CLB) з іншим через систему провідників усередині ПЛІС.

Подібну структуру часто називають island-style FPGA або mesh-based FPGA, оскільки вона утворює своєрідну сітку з логічних блоків і комутаційних елементів. Типовими представниками такої архітектури є мікросхеми Altera серій Cyclone та Stratix.

На рис. 1.55 зображено архітектуру ієрархічної ПЛІС

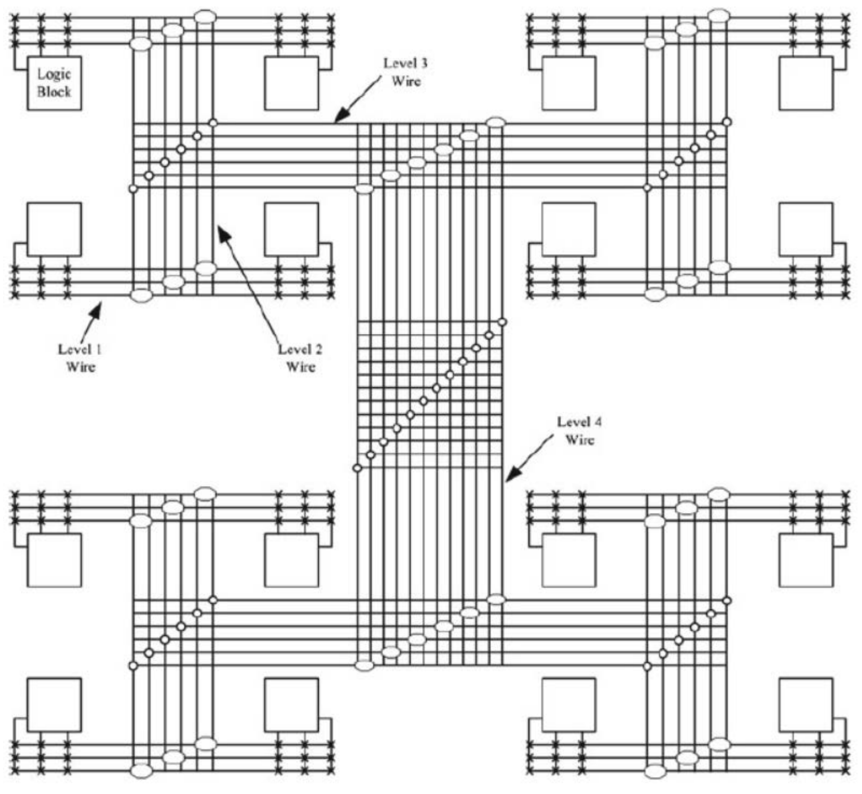


Рисунок 1.55 – Ієрархічна ПЛІС

Ієрархічна архітектура ПЛІС

Інший підхід до організації ПЛІС – це ієрархічна архітектура. Вона базується на припущенні, що в цифрових схемах окремі функціональні частини часто взаємодіють між собою інтенсивніше, ніж із віддаленими модулями.

У такій структурі логічні блоки, розташовані поруч, можуть бути з'єднані між собою відносно просто – із використанням невеликої кількості комутаційних елементів. Завдяки цьому локальні з'єднання працюють швидко та ефективно. Якщо ж потрібно об'єднати більші функціональні блоки, сигнал передається на вищий рівень ієрархії, після чого спрямовується до іншого кластера логічних елементів.

Не можна однозначно сказати, що один підхід кращий за інший. І острівна, і ієрархічна архітектури мають власні переваги та недоліки, які впливають на швидкодію, гнучкість проєктування та ефективність використання ресурсів [2].

Прикладами ПЛІС з ієрархічною структурою є мікросхеми Altera серій Flex10K та APEX.

В таблиці 1.5 наведено порівняння архітектур ПЛІС.

Таблиця 1.5 – Порівняння архітектур ПЛІС

Характеристика	Острівна архітектура	Ієрархічна архітектура
Розташування блоків	регулярна сітка	групування у кластери
Гнучкість	дуже висока	середня
Швидкодія локальних зв'язків	середня	висока
Складність маршрутизації	нижча	вища
Масштабованість	хороша	хороша, але залежить від ієрархії
Приклади	Cyclone, Stratix	Flex10K, APEX

Острівна архітектура забезпечує більшу універсальність і гнучкість під час проєктування цифрових схем. Ієрархічна архітектура, у свою чергу, дозволяє досягти вищої швидкодії завдяки ефективним локальним з'єднанням. Вибір конкретної архітектури залежить від вимог до продуктивності, складності схеми та особливостей проєкту.

1.10 Контрольні питання до розділу 1

1. Що таке програмовані логічні інтегральні схеми (ПЛІС)?
2. У чому полягає відмінність ПЛІС від замовних ВІС (ASIC)?
3. Які основні сфери застосування ПЛІС?
4. Які переваги та недоліки ПЛІС порівняно з ASIC?
5. Які етапи розвитку ПЛІС можна виділити?
6. Причини та передумови розвитку ПЛІС.
7. Які перші програмовані логічні пристрої були створені?

8. У чому полягали обмеження ранніх ПЛІС?
9. Яку роль відіграла компанія Xilinx у розвитку ПЛІС?
10. Як змінилася архітектура ПЛІС із розвитком технологій?
11. Які сучасні тенденції розвитку ПЛІС?
12. Які існують технології програмування апаратних засобів?
13. У чому різниця між одноразовим і багаторазовим програмуванням?
14. Які технології використовуються для зберігання конфігурації?
15. Що таке конфігураційна пам'ять?
16. Які способи завантаження конфігурації існують?
17. Що таке ПЛМ?
18. Яка структура PAL?
19. Чим відрізняється GAL від PAL?
20. Які переваги GAL над PAL?
21. У яких випадках застосовуються ПЛМ, PAL і GAL?
22. Що таке ASIC?
23. Які існують типи ASIC?
24. У чому різниця між стандартними та повністю замовними ASIC?
25. Які переваги ASIC перед ПЛІС?
26. Які недоліки ASIC порівняно з ПЛІС?
27. Що таке структурований ASIC?
28. Що входить до складу базисного модуля структурованого ASIC?
29. Які переваги структурованого підходу?
30. Які обмеження має структурований ASIC?
31. У яких випадках доцільно застосовувати структурований ASIC?
32. З яких основних компонентів складається ПЛІС?
33. Що таке логічний блок ПЛІС?
34. Яке призначення комутаційної мережі?
35. Що таке базисний модуль ПЛІС?
36. Як реалізуються логічні функції на комірках ОЗП?
37. Як працює LUT (таблиця істинності)?
38. Як реалізуються функції на мультиплексорах?
39. Які переваги LUT-підходу?
40. Які обмеження має реалізація на ОЗП-комірках?
41. У чому відмінність між реалізацією на LUT і мультиплексорах?
42. З яких елементів складається логічна комірка Xilinx?
43. Яку роль відіграють тригери у логічній комірці?
44. Що таке конфігурований логічний блок (CLB)?

45. Які режими роботи може підтримувати логічна комірка?
46. Як здійснюється конфігурація логічної комірки?
47. Що таке логічна комірка?
48. Що таке секція та логічний масив?
49. Яке призначення функціональних модулів у ПЛІС?
50. Як відбувається маршрутизація сигналів між блоками?
51. Які фактори впливають на продуктивність ПЛІС?
52. Що таке конфігураційний файл?
53. Що таке конфігураційні комірки?
54. Що таке JTAG і яке його призначення?
55. У чому різниця між послідовним і паралельним завантаженням?
56. Що означають режими «ведучий» та «ведений»?
57. Що таке мультипрограмування конфігураційних ланцюжків?
58. Яке призначення конфігураційного порту?
59. Які способи програмування через ОЗП використовуються?
60. Які ключові риси сучасних ПЛІС?
61. Які архітектурні особливості сучасних ПЛІС забезпечують їх високу продуктивність?

2 МОВА ПРОГРАМУВАННЯ VERILOG

2.1 Основи мови програмування Verilog

2.1.1 Лексичні правила

Базові лексичні правила у Verilog HDL подібні до правил у мові програмування C. Verilog оперує потоком токенів. Токени можуть бути коментарями, роздільниками, числами, рядками, ідентифікаторами та ключовими словами. Verilog чутлива до регістру. Усі ключові слова пишуться малими літерами [4].

Пробіли (\b), табуляції (\t) і символи нового рядка (\n) утворюють пропуски (whitespace). Пропуски у Verilog ігноруються, окрім випадків, коли вони відокремлюють токени. У рядках пропуски не ігноруються.

Коментарі можуть бути вставлені в код для читабельності та документації. Є два способи написання коментарів. Однорядковий коментар починається з «//». Verilog пропускає цю точку до кінця рядка. Коментар із кількома рядками починається з «/*» і закінчується на «*/». Кілька рядків коментарів не можуть бути вкладеними. Однак коментарі з одним рядком можуть бути вбудовані в коментарі з кількома рядками.

```
a = b && c; // Це однорядковий коментар

/* Це багаторядковий
коментар */

/* Це /* некоректний */ коментар */

/* Це є // допустимий коментар */
```

2.1.2 Вирази та операнди

Вирази – це конструкції, що поєднують оператори та операнди для отримання результату.

```
// Приклади виразів
a ^ b
addr1[20:17] + addr2[20:17]
in1 | in2
```

Операндами можуть бути будь-які типи даних. Деякі конструкції приймають лише певні типи операндів. Операнди можуть бути константами, цілими числами, дійсними числами, провідниками (nets), регістрами, часом, бітовим вибором (один біт з шини бітів або векторного регістра), частковим вибором (вибрані біти з шини бітів або векторного регістра), а також пам'яттю або викликами функцій [11].

```
integer count, final_count;
final_count = count + 1;// count це цілочисельний тип
real a, b, c;
c = a - b; // a і b є дійсними операндами
reg [15:0] reg1, reg2; reg [3:0] reg_out;
reg_out = reg1[3:0] ^ reg2[3:0];
// reg1[3:0] і reg2[3:0] є
// операнди вибірки частини регістра
reg ret_value;
ret_value = calculate_parity(A, B);
// calculate_parity є операндом типу функції
```

2.1.3 Оператори

Verilog має багато різних типів операторів. Оператори можуть бути арифметичними, логічними, порівняння, рівності, побітовими, редукційними, зсуву, конкатенації або умовними. Деякі з цих операторів подібні до операторів, що використовуються в мові програмування С. Кожен тип оператора позначається символом. У табл. 2.1 наведено повний перелік символів операторів, класифікованих за категоріями [4].

Арифметичні оператори

Є два типи арифметичних операторів: бінарні та унарні.

Бінарні арифметичні оператори – це множення (*), ділення (/), додавання (+), віднімання (-), піднесення до степеня (**) та остача від ділення (%).

Бінарні оператори приймають два операнди.

```
A = 4'b0011; B = 4'b0100;
// A і B є багаторозрядними регістрами
D = 6; E = 4; F=2
// D і E є цілочисельними типами

A * B // Множення A і B. Дає результат 4'b1100
D / E // Результат 1. Відкидає дробову частину.
A + B // Додавання A і B. Дає результат 4'b0111
B - A // Віднімання A від B. Результат 4'b0001
F = E ** F; //E в степені F, повертає 16
```

Якщо будь-який біт операнда має значення x, тоді результат усього виразу буде x. Це виглядає інтуїтивно зрозумілим, оскільки якщо значення операнда невідоме точно, результат також має бути невідомим.

```
in1 = 4'b101x; in2 = 4'b1010;
sum = in1 + in2;
// сума буде обчислена як значення 4'bx
```

Таблиця 2.1 – Типи операторів класифікованих за категоріями

<table>
<tr><th>Тип оператора</th><th>Позначення</th><th>Операція</th><th>Кількість операндів</th></tr>
<tr><td>Арифметичний</td><td>*
/
+
-
%
**</td><td>множення
ділення
додавання
віднімання
остача від ділення
x у степені y</td><td>2
2
2
2
2
2</td></tr>
<tr><td>Логічний</td><td>!
&&
||</td><td>логічне заперечення
логічне І
логічне АБО</td><td>1
2
2</td></tr>
<tr><td>Порівняння</td><td>>
<
>=
<=</td><td>більше ніж
менше ніж
більше або дорівнює
менше або дорівнює</td><td>2
2
2
2</td></tr>
<tr><td>Рівності</td><td>==
!=
===
!==</td><td>логічна рівність
логічна нерівність
покомпонентна рівність
покомпонентна нерівність</td><td>2
2
2
2</td></tr>
<tr><td>Побітові</td><td>~
&
|
^
^~ or ~^</td><td>заперечення
І
АБО
виключне АБО
еквівалентність</td><td>1
2
2
2
2</td></tr>
<tr><td>Оператори редукції</td><td>&
~&
|
~|
^
^~ or ~^</td><td>редукційне AND
редукційне NAND
редукційне OR
редукційне NOR
редукційне XOR
редукційне XNOR</td><td>1
1
1
1
1
1</td></tr>
<tr><td>Зсуву</td><td>>>
<<
>>>
<<<</td><td>зсув вправо
зсув вліво
арифметиний зсув вправо
арифметиний зсув вліво</td><td>2
2
2
2</td></tr>
<tr><td>Конкатенація</td><td>{ }</td><td>конкатенація</td><td>будь-яке</td></tr>
<tr><td>Реплікація</td><td>{ { } }</td><td>реплікація</td><td>будь-яке</td></tr>
<tr><td>Умовний</td><td>?:</td><td>Умова</td><td>3</td></tr>
</table>

Оператори остачі від ділення (modulus operators) повертають остачу від ділення двох чисел. Вони працюють подібно до оператора остачі від ділення в мові програмування С.

```
13 % 3 // Дає результат 1
16 % 4 // Дає результат 0
-7 % 2 // Результат -1, приймає знак першого операнда
7 % -2 // Результат +1, приймає знак першого операнда
```

Унарні оператори. Оператори + та - також можуть бути унарними. Вони використовуються для вказання додатного або від'ємного знака операнда. Унарні оператори + або - мають вищий пріоритет, ніж бінарні оператори + або - [16].

```
-4   // Мінус 4 (від'ємне число 4)
+5   // Плюс 5 (додатне число 5)
```

Від'ємні числа у Verilog внутрішньо подаються у вигляді доповнення до двійки (2's complement). Рекомендується використовувати від'ємні числа лише типів integer або real у виразах. Проєктувальникам потрібно уникати від'ємних чисел у вигляді <sss>'<base><nnn> у виразах, оскільки вони перетворюються у беззнакові числа в поданні 2's complement і, таким чином, дають неочікувані результати.

```
// Рекомендується використовувати типу integer або real
-10 / 5   // Обчислюється як -2
// Не потрібно використовувати числа у вигляді
// <sss>'<base><nnn>
-'d10 / 5
// Це еквівалент (2's complement від 10)/5 = (2^32 - 10)/5
// де 32 — це ширина машинного слова за замовчуванням.
//Обчислюється до некоректного та неочікуваного результату
```

Логічні Оператори

Логічні оператори – це логічне І (&&), логічне АБО (||) та логічне НЕ (!). Оператори && та || є бінарними. Оператор ! є унарним.

Логічні оператори підпорядковуються таким умовам [18]:

1. Логічні оператори завжди обчислюються до 1-бітного значення: 0 (хибність), 1 (істина) або x (невизначеність).

2. Якщо операнд не дорівнює нулю, він еквівалентний логічній 1 (умова істинна). Якщо операнд дорівнює нулю, він еквівалентний логічному 0 (умова хибна). Якщо будь-який біт операнда дорівнює x або z, результат вважається x (невизначений стан) і зазвичай трактується симуляторами як хибність.

3. Логічні оператори приймають як операнди змінні або вирази.

Використання дужок для групування логічних операцій наполегливо рекомендується – для покращення читабельності та щоб уникнути необхідності запам'ятовувати пріоритет операторів.

```
// Логічні операції
A = 3; B = 0;
A && B   // Обчислюється як 0(логічна-1 && логічний-0)
A || B   // Обчислюється як 1(логічна-1 || логічний-0)
!A       // Обчислюється як 0. Еквівалент not(логічна-1)
!B       // Обчислюється як 1. Еквівалент not(логічний-0)

// Невідомі значення
A = 2'b0x; B = 2'b10;
A && B   // Обчислюється як x. Еквівалент (x && логічна-1)

// Вирази
(a == 2) && (b == 3)
// Обчислюється як 1, якщо одночасно a == 2 та b == 3.
// Обчислюється як 0, якщо хоча б одна з умов хибна.
```

Оператори порівняння: більше (>), менше (<), більше або дорівнює (>=) і менше або дорівнює (<=). Якщо оператори порівняння використовуються у виразі, такий вираз повертає логічне значення 1, якщо умова істинна, і 0, якщо умова хибна. Якщо в операндах присутні будь-які біти зі значенням x або z, вираз приймає значення x. Ці оператори працюють точно так само, як і відповідні оператори у мові програмування C.

```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx

A <= B // Дає результат a логічний 0
A > B // Дає результат логічний 1
Y >= X // Дає результат логічний 1
Y < Z // Дає результат x
```

Оператори рівності: логічна рівність (==), логічна нерівність (!=), рівність з урахуванням станів (===) та нерівність з урахуванням станів (!==). Коли вони використовуються у виразі, оператори рівності повертають логічне значення 1, якщо умова істинна, і 0, якщо хибна. Ці оператори порівнюють два операнди побітно, з доповненням нулями, якщо операнди мають різну довжину.

Важливо звернути увагу на різницю між логічними операторами рівності (==, !=) та операторами рівності з урахуванням станів (===, !==).

Логічні оператори рівності (==, !=) повертають значення x, якщо будь-який з операндів містить біти x або z.

Оператори рівності з урахуванням станів (===, !==) порівнюють операнди побітно, включно біти x та z.

Результат дорівнює 1, якщо операнди збігаються повністю, включно біти x та z.

Результат дорівнює 0, якщо операнди не збігаються повністю.

Оператори рівності з урахуванням станів ніколи не повертають x.

```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101
// Z = 4'b1xxz, M = 4'b1xxz, N = 4'b1xxx

A == B      // Результат: логічний 0
X != Y      // Результат: логічна 1
X == Z      // Результат: x (через біти x)
Z === M
// Результат: 1 (усі біти збігаються, включно з x і z)
Z === N
// Результат: 0 (наймолодший біт не збігається)
M !== N     // Результат: логічна 1
```

Побітові оператори: заперечення (~), І (&), АБО (|), виключне АБО (^), еквівалентність (XNOR: ^~, ~^). Побітові оператори виконують операцію побітово над двома операндами. Вони беруть кожен біт одного операнда й виконують операцію з відповідним бітом іншого операнда. Якщо один операнд коротший за інший, він доповнюється нулями зліва, щоб довжина збігалася з довжиною довшого операнда. Значення z у побітових операціях трактується як x. Винятком є унарний оператор заперечення (~), який приймає лише один операнд і діє на всі його біти. [24].

```
// X = 4'b1010, Y = 4'b1101
// Z = 4'b10x1

~X       // Заперечення. Результат: 4'b0101
X & Y    // Побітове І. Результат: 4'b1000
X | Y    // Побітове АБО. Результат: 4'b1111
X ^ Y    // Побітове XOR. Результат: 4'b0111
X ^~ Y   // Побітове XNOR. Результат: 4'b1000
X & Z    // Результат: 4'b10x0
```

Важливо розрізняти побітові оператори ~, &, | від логічних операторів !, &&, ||.

Логічні оператори завжди повертають логічне значення 0, 1 або x, тоді як побітові оператори повертають значення побітово.

Логічні оператори виконують логічну операцію, а не побітову.

```
// X = 4'b1010, Y = 4'b0000

X | Y    // Побітова операція. Результат: 4'b1010
X || Y   // Логічна операція. Еквівалент: 1||0. Результат=1
```

Оператори редукції: AND (&), NAND (~&), OR (|), NOR (~|), XOR (^) та XNOR (~^, ^~).

Оператори редукції приймають лише один операнд. Вони виконують побітову операцію над усім векторним операндом і повертають результат у вигляді 1-бітного значення.

Логічні таблиці для цих операторів збігаються з тими, що і для побітових операторів [4].

Відмінність полягає в тому, що побітові операції виконуються над бітами двох різних операндів, тоді як операції редукції – над усіма бітами одного операнда.

Оператори редукції обробляють біти справа наліво. Редукційний NAND обчислюється як інверсія результату редукційного AND. Редукційний NOR – як інверсія редукційного OR. Редукційний XNOR – як інверсія редукційного XOR.

```
// X = 4'b1010

&X   // Еквівалент: 1 & 0 & 1 & 0. Результат: 1'b0
|X   // Еквівалент: 1 | 0 | 1 | 0. Результат: 1'b1
^X   // Еквівалент: 1 ^ 0 ^ 1 ^ 0. Результат: 1'b0

// Редукційні XOR або XNOR можуть використовуватись для
// генерації парності (even/odd parity) вектора.
```

Використання схожих символів для: логічних операторів (!, &&, ||), побітових операторів (~, &, |, ^), операторів редукції (&, |, ^) на початку може бути дещо заплутаним. Різниця полягає у кількості операндів, які приймає оператор, та у типі значення, яке він повертає.

Оператори зсуву: це зсув вправо (>>), зсув вліво (<<), арифметичний зсув вправо (>>>) та арифметичний зсув вліво (<<<). Звичайні оператори зсуву переміщують векторний операнд вправо або вліво на задану кількість бітів. Операндами є сам вектор і кількість бітів для зсуву.

Коли відбувається зсув, звільнені бітові позиції заповнюються нулями. Зсув не має циклічного ефекту (тобто біти не «переносяться» з одного краю в інший) [26].

Арифметичні оператори зсуву використовують контекст виразу, щоб визначити, якими значеннями заповнювати звільнені біти.

```
// X = 4'b1100

Y = X >> 1;  // Y = 4'b0110. Зсув вправо на 1 біт. MSB
заповнений 0.
Y = X << 1;  // Y = 4'b1000. Зсув вліво на 1 біт. LSB
заповнений 0.
Y = X << 2;  // Y = 4'b0000. Зсув вліво на 2 біти.

integer a, b, c;
a = 0;
```

```
b = -10;          // у двійковій формі: 00111...10110
c = a + (b >>> 3); // Результат: -2 у десятковій системі
                   // завдяки арифметичному зсуву
```

Оператори зсуву корисні тим, що дозволяють розробнику моделювати: операції зсуву, алгоритми типу «shift-and-add» для множення та інші корисні операції.

Оператор конкатенації ({ , }) забезпечує механізм для об’єднання кількох операндів. Операнди мають бути визначеного розміру. Невизначені за розміром операнди не допускаються, оскільки розмір кожного операнда має бути відомий для обчислення розміру результату. Конкатенація записується як список операндів у фігурних дужках, розділених комами.

Операндами можуть бути:
- скалярні провідники (nets) або регістри,
- векторні провідники або регістри,
- вибірка біта (bit-select),
- вибірка частини (part-select),
- константи з визначеним розміром.

```
// A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110
Y = {B , C}  // Результат: Y = 4'b0010
Y = {A,B,C,D,3'b001}  // Результат: Y = 11'b10010110001
Y = {A , B[0], C[1]}  // Результат: Y = 3'b101
```

Повторювана конкатенація одного й того самого числа може бути виражена за допомогою константи реплікації. Константа реплікації вказує, скільки разів потрібно повторити число всередині фігурних дужок ({ }).

```
reg A;
reg [1:0] B, C;
reg [2:0] D;

A = 1'b1;
B = 2'b00;
C = 2'b10;
D = 3'b110;

Y = { 4{A} }              // Результат: Y = 4'b1111
Y = { 4{A} , 2{B} }       // Результат: Y = 8'b11110000
Y = { 4{A} , 2{B} , C }  // Результат: Y = 8'b1111000010
```

Умовний оператор (?:) приймає три операнди.

Синтаксис: condition_expr ? true_expr : false_expr;

Спочатку обчислюється умова (condition_expr). Якщо результат істина (логічна 1), тоді обчислюється true_expr. Якщо результат хибність

(логічний 0), тоді обчислюється false_expr. Якщо результат x (невизначеність), тоді обчислюються обидва вирази (true_expr і false_expr), і їх результати порівнюються побітно: якщо біти різні → у цій позиції результат дорівнює x; якщо біти однакові → у цій позиції результат дорівнює значенню цього біта. Дія умовного оператора подібна до мультиплексора. Альтернативно його можна порівняти з конструкцією if-else. Умовні оператори часто застосовуються в dataflow-моделюванні для опису умовних присвоєнь. Умовний вираз діє як сигнал керування перемиканням.

```
// Моделювання функціоналу трьохстанового буфера
assign addr_bus = drive_enable ? addr_out : 36'bz;
// Моделювання функціоналу мультиплексора 2-до-1
assign out = control ? in1 : in0;
```

Умовні операції можна вкладати одна в одну.

Кожен true_expr або false_expr сам може бути умовною операцією.

У наступному прикладі переконайся сам, що (A == 3) і control – це два сигнали вибору 4-до-1 мультиплексора з входами n, m, y, x та виходом out:

```
assign out = (A == 3) ? ( control ? x : y )
                      : ( control ? m : n );
```

Для операторів характерним є поняття пріоритету. Якщо у виразах не використовуються дужки для розділення частин, у Verilog діє такий порядок пріоритетів. Оператори, наведені в табл. 2.2, розташовані у порядку від найвищого пріоритету до найнижчого.

Таблиця 2.2 – Пріоритет операторів

Оператори	**Символи операторів**	**Пріоритет**
Унарні	+ - ! ~	найвищий пріоритет
Множення, ділення, остача	* / %	
Додавання віднімання	+ -	
Зсуву	<< >>	
Relational	< <= > >=	
Порівняння	== != === !==	
Reduction	&, ~&	
Логічні	&&	
Умовні	?:	найнижчий пріоритет

Рекомендується завжди використовувати дужки для розділення виразів, окрім випадків з унарними операторами або коли немає жодної двозначності.

2.1.4 Ідентифікатори та ключові слова

Ключові слова – це спеціальні ідентифікатори, зарезервовані для визначення конструкцій мови. Ключові слова пишуться малими літерами. Список основних ключових слів у Verilog наведено у табл. 2.3 [24].

Таблиця 2.3 – Список ключових слів

always	ifnone	rnmos	endspecify	use
and	incdir	rpmos	endtable	vectored
assign	include	rtran	endtask	wait
automatic	initial	rtranif0	event	wand
begin	inout	rtranif1	for	weak0
buf	input	scalared	force	weak1
bufif0	instance	showcancelled	forever	rcmos
bufif1	integer	signed	fork	real
case	join	small	function	realtime
casex	large	specify	generate	reg
casez	liblist	specparam	genvar	release
cell	library	strong0	highz0	repeat
cmos	localparam	strong1	highz1	rcmos
config	macromodule	supply0	if	wor
deassign	medium	supply1	endspecify	xnor
default	module	table	endtable	xor
defparam	nand	task	endtask	endfunction
design	negedge	time	event	endgenerate
disable	nmos	tran	posedge	endmodule
edge	nor	tranif0	primitive	endprimitive
else	noshowcancelled	tranif1	pull0	triand
end	not	tri	pull1	trior
endcase	notif0	tri0	pulldown	trireg
endconfig	notif1	tri1	pullup	unsigned
or	parameter	pulsestyle_onevent while		
output	pmos	pulsestyle_ondetect wire		

Ідентифікатори – це назви, які надаються об'єктам, щоб можна було звертатися до них у проєкті. Ідентифікатори складаються з буквено-цифрових символів, підкреслення (_) або знака долара ($). Вони чутливі до регістру. Ідентифікатори починаються з літери або підкреслення. Вони не можуть починатися з цифри або знака $ (символ $ на початку зарезервований для системних задач, про які йтиметься далі).

```
reg value;   // reg — ключове слово; value — ідентифікатор
input clk;   // input — ключове слово; clk — ідентифікатор
```

Екрановані ідентифікатори починаються зі зворотної похилої риски (\ ) та закінчуються пробілом, табуляцією або символом нового рядка. Усі символи між початковою рискою та пропуском обробляються буквально. Будь-які друковані ASCII-символи можуть бути віднесені до екранованих ідентифікаторів. Ні сама зворотна риска, ні символ завершення не вважаються частиною ідентифікатора.

```
\a+b-c
\**my_name**
```

Нижче наведено список ключових слів, які часто використовуються симуляторами Verilog для імен системних задач і функцій. Список упорядкований за алфавітом.

```
$bitstoreal         $countdrivers      $display
$fclose             $fdisplay          $fmonitor
$fopen              $fstrobe           $fwrite
$finish             $getpattern        $history
$incsave            $input             $itor
$key                $list              $log
$monitor            $monitoroff        $monitoron
$nokey
```

Нижче наведено список ключових слів, які часто використовуються симуляторами Verilog для задання директив компілятора.

```
'accelerate              'autoexpand_vectornets
'celldefine              'default_nettype
'define                  'define
'else                    'elsif
'endcelldefine           'endif
'endprotect              'endprotected
'expand_vectornets       'ifdef
'ifndef                  'include
'noaccelerate            'noexpand_vectornets
'noremove_gatenames      'nounconnected_drive
'protect                 'protected
'remove_gatenames        'remove_netnames
'resetall                'timescale
'unconnected_drive
```

2.1.5 Типи даних

У Verilog є два способи подання чисел: із зазначеним розміром і без нього. Verilog підтримує чотири значення та вісім рівнів для моделювання функціональності реального обладнання. Чотири рівні значень наведено в табл. 2.4 [26].

Таблиця 2.4 – Рівні значень

Значення	Стан в апаратних схемах (англ.)	Стан в апаратних схемах (укр.)
0	Logic zero, false condition	Логічний нуль, хибна умова
1	Logic one, true condition	Логічна одиниця, істинна умова
x	Unknown logic value	Невизначене логічне значення
z	High impedance, floating state	Високий імпеданс, стан «плаваючий»

Окрім логічних значень, рівні сили сигналу часто використовуються для розв'язання конфліктів між драйверами з різною потужністю в цифрових схемах. Рівні значень 0 і 1 можуть мати різні сили сигналу, наведені в табл. 2.5 [4].

Таблиця 2.5 – Рівні значень

Рівень сили	Тип	Ступінь
supply	Керуючий (Driving)	найсильніший
strong	Керуючий (Driving)	↑
pull	Керуючий (Driving)	
large	Збереження (Storage)	
weak	Керуючий (Driving)	
medium	Збереження (Storage)	
small	Збереження (Storage)	
highz	Високий імпеданс (High Impedance)	найслабший

Якщо два сигнали з нерівними рівнями сили подаються на одну шину, переважає сильніший сигнал. Наприклад, якщо два сигнали з рівнями strong1 і weak0 конфліктують, результат визначається як strong1. Якщо два сигнали з однаковою силою подаються на шину, результат є невизначеним (x). Якщо два сигнали з рівнями strong1 і strong0 конфліктують, результат також буде x.

Рівні сили особливо корисні для точного моделювання суперечностей сигналів, МОП-пристроїв, динамічних МОП та інших низькорівневих пристроїв. Тільки мережі типу trireg можуть мати рівні збереження large, medium і small.

Провідники, шини (Nets)

Провідники, шини (nets) – це з'єднання між апаратними елементами. Так само, як у реальних схемах, значення на провідниках безперервно формуються виходами пристроїв, до яких вони підключені [19].

У прикладі на рис. 2.1 провідник *a* підключений до виходу логічного елемента *and* з іменем *g1*. Провідник *a* постійно прийматиме значення, обчислене на виході елемента *g1*, тобто *b & c*.

Рисунок 2.1 – Приклад провідників, шин

Шини оголошуються переважно за допомогою ключового слова wire. За замовчуванням шини є однобітними (лише один провідник), якщо їх явно не оголошено як вектори (масив провідників). Терміни wire та net часто використовуються як взаємозамінні. Значенням мережі за замовчуванням є *z* (за винятком мережі типу *trireg*, яка за замовчуванням дорівнює *x*). Шина приймає вихідне значення своїх драйверів. Якщо шина не має драйвера, вона отримує значення *z*.

```
wire a;    // Оголосити шину a для наведеної вище схеми
wire b,c;  // Оголосити дві шини b, c для наведеної вище схеми
wire d = 1'b0; // Шина d зафіксована на логічному рівні 0 під час оголошення
```

Зверніть увагу, що net не є ключовим словом, а є представником класу типів даних, таких як wire, wand, wor, tri, triand, trior, trireg тощо. Найчастіше використовується оголошення wire [24].

Регістри (Registers)

Регістри являють собою елементи збереження даних. Регістри утримують значення доти, доки їм не буде призначене нове. Не потрібно плутати регістри у Verilog з апаратними регістрами, побудованими на тригерах із керуванням по фронту тактового сигналу в реальних схемах. У Verilog термін register означає просто змінну, здатну зберігати значення. На відміну від шин (nets), регістрам не потрібні драйвери. Регістри у Verilog не потребують тактового сигналу, як апаратні регістри. Значення регістрів можна змінювати будь-коли під час симуляції шляхом присвоєння їм нового значення.

Типи даних регістрів зазвичай оголошуються за допомогою ключового слова reg. Значення за замовчуванням для типу *reg* дорівнює *x*. Приклад використання регістрів наведено нижче.

```
Приклад використання регістра
reg reset; // оголошення змінної reset, що може зберігати своє значення

initial // цей конструктив буде розглянуто пізніше
begin
   reset = 1'b1;     // ініціалізувати reset значенням 1 для скидання цифрової схеми
   #100 reset = 1'b0; // через 100 одиниць часу сигнал reset скасовується
end
```

Регістр також можна оголосити як знакову змінну (*signed*). Такі регістри можна використовувати для арифметики зі знаком. Далі наведено приклад оголошення знакового регістра.

```
Приклад оголошення знакового регістра
reg signed [63:0] m; // 64-бітне знакове значення
integer i;           // 32-бітне знакове значення
```

Вектори (Vectors)

Шини (nets) або регістри (reg) можуть оголошуватися як вектори (кілька біт у ширину). Якщо ширина в бітах не вказана, за замовчуванням використовується скаляр (1 біт).

```
wire a;     // скалярна змінна типу net, за замовчуванням
wire [7:0] bus;          // 8-бітна шина
wire [31:0] busA,busB,busC; // три 32-бітні шини
reg clock;         // скалярний регістр, за замовчуванням
reg [0:40] virtual_addr;  // векторний регістр, віртуальна адреса завширшки 41 біт
```

Вектори можуть оголошуватися як [старший:молодший] або [молодший:старший], але ліве число у квадратних дужках завжди є найбільш значущим бітом вектора.

У наведеному вище прикладі бітом 0 є найбільш значущий біт вектора virtual_addr.

Вибір частини вектора (Vector Part Select)

Для векторних оголошень, показаних вище, можна звертатися до окремих бітів або частин вектора.

```
busA[7]          // біт №7 вектора busA
bus[2:0]         // три молодших біти вектора bus
                 // використання bus[0:2] є некоректним,
                 // оскільки найбільш значущий біт завжди
                 // має бути зліва у діапазоні
virtual_addr[0:1] // два найбільш значущі біти
                  //вектора virtual_addr
```

Variable Vector Part Select

Ще однією можливістю, передбаченою у Verilog HDL, є змінний вибір частин вектора (variable part select). Це дозволяє використовувати вибір частин вектора у циклах for для послідовного доступу до різних частин вектора.

Існують два спеціальні оператори для вибору частин вектора:

```
[<початковий_біт> +: ширина] — вибір частини вектора з
інкрементом від початкового біта.

[<початковий_біт> -: ширина] — вибір частини вектора з
декрементом від початкового біта.
```

Початковий біт для вибору частини вектора (part select) може змінюватися, але ширина вибраної частини завжди має бути сталою.

Нижче наведено приклад використання змінного вибору частин вектора (variable vector part select):

```
reg [255:0] data1;   // Позначення Little endian
reg [0:255] data2;   // Позначення Big endian
reg [7:0] byte;      // Окремий байт

// Використання змінного вибору частини вектора
byte = data1[31-:8];   // початковий біт = 31, ширина = 8
=> data1[31:24]
```

```
    byte = data1[24+:8];   // початковий біт = 24, ширина = 8
=> data1[31:24]
    byte = data2[31-:8];   // початковий біт = 31, ширина = 8
=> data2[24:31]
    byte = data2[24+:8];   // початковий біт = 24, ширина = 8
=> data2[24:31]

    // Початковий біт може бути змінною, але ширина повинна
бути сталою.
    // Тому змінний вибір частини вектора можна використати у
циклі:
    for (j = 0; j <= 31; j = j + 1)
        byte = data1[(j*8)+:8];
        // послідовність: [7:0], [15:8], …, [255:248]

    // Можна ініціалізувати частину вектора:
    data1[(byteNum*8)+:8] = 8'b0;
    // якщо byteNum = 1, очистяться біти [15:8]
```

Цілісні (Integer), Дійсні (Real) та Часові (Time) типи регістрів

У Verilog підтримуються цілісні, дійсні та часові типи регістрів.

Ціле число (***integer***) – це універсальний тип регістра, який використовується для маніпуляцій із величинами.

Оголошується за допомогою ключового слова integer [11].

Хоча можна використовувати reg як універсальну змінну, зручніше оголошувати змінну типу integer для завдань, наприклад, підрахунку.

Типове значення ширини для integer дорівнює розміру машинного слова хост-машини (залежить від реалізації, але не менше 32 бітів).

Регістр типу reg зберігає беззнакові значення, тоді як integer зберігає знакові значення.

```
    integer counter; // універсальна змінна, використовується
як лічильник
    initial counter = -1; // У лічильнику зберігається -1
```

Дійсні числа (***real***) оголошуються ключовим словом real.

Вони можуть задаватися у десятковому записі (наприклад, 3,14) або у науковій нотації (наприклад, 3e6, що означає 3×10^6). Для real не можна оголошувати діапазон значень. Типове значення – 0.0. У разі присвоєння дійсного значення змінній типу integer, відбувається округлення до найближчого цілого числа.

```
    real delta; // Оголосити дійсну змінну delta
    initial begin
    delta = 4e10; // delta отримує значення у науковій нотації
    delta = 2.13; // delta отримує значення 2.13
```

```
end
integer i; // Оголосити цілу змінну i
initial
i = delta; // i отримує значення 2 (округлене від 2.13)
```

Time. Імітація у Verilog виконується відносно симуляційного часу.

Для збереження симуляційного часу використовується спеціальний тип регістра time. Ширина time залежить від реалізації, але не менше 64 бітів. Системна функція $time використовується для отримання поточного симуляційного часу [26].

```
time save_sim_time; // Оголосити змінну часу
                    //save_sim_time initial
save_sim_time = $time;//Зберегти поточний симуляційний час
```

Симуляційний час вимірюється в секундах (s), як і реальний час.

Але співвідношення між реальним часом у цифровій схемі та симуляційним часом визначається користувачем.

Масиви (arrays)

У Verilog дозволяється використовувати масиви для типів даних reg, integer, time, real, realtime, а також для векторних регістрів. Можна оголошувати багатовимірні масиви з будь-якою кількістю вимірів. Масиви типу net також можна застосовувати для з’єднання портів у згенерованих екземплярах модулів [4].

Кожен елемент масиву може використовуватись також як скалярний або векторний провідник.

Доступ до елементів здійснюється за допомогою запису:

```
<array_name>[<індекс>]
```

Для багатовимірних масивів потрібно вказувати індекс для кожного виміру. Приклади оголошення масивів:

```
 integer count[0:7];  // масив із 8 змінних типу integer
 reg bool[31:0]; // масив із 32 однобітних регістрів (булевих)
 time chk_point[1:100];  // масив із 100 змінних типу time
 reg [4:0] port_id[0:7];// масив із 8 елементів по 5 біт кожен
 integer matrix[4:0][0:255];// двовимірний масив цілих чисел
 reg  [63:0]  array_4d  [15:0][7:0][7:0][255:0];//  4-вимірний
масив
 wire  [7:0]  w_array2  [5:0];    //  масив  8-бітних  векторних
сигналів типу wire
 wire  w_array1[7:0][5:0];   //  масив  однобітних  сигналів  типу
wire
```

Не потрібно плутати масиви з векторними мережами або регістрами. Вектор – це один елемент шириною n біт. Масив – це множина елементів, де кожен елемент може бути 1-бітним або n-бітним [16].

Приклади операцій з масивами

```
count[5] = 0;   // скидання 5-го елемента масиву count
chk_point[100] = 0; // скидання 100-го чекпоінту часу
port_id[3] = 0;           // скидання 3-го елемента (5-бітного) масиву port_id

matrix[1][0] = 33559; // присвоєння значення 33559 елементу [1][0]
array_4d[0][0][0][0][15:0] = 0;  // очищення бітів 15:0 елемента, доступного за індексами [0][0][0][0]

// неправильний синтаксис:
port_id = 0;    // спроба записати в увесь масив port_id
matrix[1] = 0;  // спроба записати одразу [1][0]..[1][255]
```

Пам’ять (Memories)

У цифровому моделюванні часто виникає потреба описати регістрові файли, RAM та ROM. У Verilog пам’ять моделюється просто як одновимірний масив регістрів. Кожен елемент такого масиву називається словом (word). Доступ до слова здійснюється за допомогою індексу масиву. Кожне слово може мати ширину від 1 біта до n бітів. Важливо розрізняти: n однобітних регістрів і один n-бітний регістр.

```
reg mem1bit[0:1023]; // пам’ять mem1bit: 1K слів по 1 біту
reg [7:0] membyte[0:1023]; // пам’ять membyte: 1K слів по 8 біт (байти)
membyte[511];   // доступ до байта за адресою 511
```

Таким чином, у Verilog пам’ять описується як простий масив, де кожен елемент є словом певної ширини, і доступ до нього виконується через індекс (адресу).

Параметри (Parameters)

У Verilog константи в модулі можна визначати за допомогою ключового слова parameter. Параметри не можна використовувати як змінні. Для кожного екземпляра модуля значення параметрів можна перевизначити окремо під час компіляції. Це дозволяє налаштовувати модулі індивідуально. Також можна задавати типи та розміри параметрів.

```
parameter port_id = 5;     // визначає константу port_id
parameter cache_line_width = 256; // константа для ширини кеш-лінії
parameter signed [15:0] WIDTH; // параметр WIDTH зі знаком і діапазоном
```

Модулі доцільно описувати через параметри. Жорстко закодованих чисел потрібно уникати. Значення параметрів можна змінювати під час інстанціювання модуля або через оператор defparam. Таким чином, параметри роблять визначення модуля гнучким: змінюючи лише їхні значення, можна змінювати поведінку модуля [26].

У мові Verilog HDL є також локальні параметри (localparam). Вони ідентичні звичайним параметрам, але:

- їх не можна змінити через defparam або через явне присвоєння значень за іменем чи порядком;

- вони використовуються тоді, коли значення параметрів мають залишатися незмінними.

Це забезпечує захист від випадкових змін параметрів. Наприклад, кодування станів автомата можна визначити так:

```
localparam state1 = 4'b0001,
           state2 = 4'b0010,
           state3 = 4'b0100,
           state4 = 4'b1000;
```

У цьому випадку кодування станів змінити не можна.

Рядки (Strings)

Рядки у Verilog можна зберігати в змінних типу reg. Ширина регістра має бути достатньою, щоб умістити увесь рядок [11].

Кожен символ у рядку займає 8 біт (1 байт). Якщо ширина регістра більша за довжину рядка, то зліва від рядка біти заповнюються нулями. Якщо ширина регістра менша за довжину рядка, то ліві символи рядка відсікаються. Тому завжди безпечніше оголошувати рядок трохи ширшим, ніж необхідно.

```
reg [8*18:1] string_value;   // змінна на 18 байтів
initial
   string_value = "Hello Verilog World";
   // рядок можна зберегти у змінній string_value
```

Під час відображення рядків у Verilog спеціальні символи відіграють особливу роль: символ нового рядка, табуляція, виведення значень аргументів тощо.

У табл. 2.6 подано спеціальні символи та їхнє призначення.

Таблиця 2.6 – Спеціальні символи рядків

Екрановані символи	Відображуваний символ
\n	newline (новий рядок)
\t	tab (табуляція)
%%	%
\\	\
\"	"
\ooo	Символ, записаний у вигляді 1–3 вісімкових цифр

Такі символи можуть відображатися у рядках лише тоді, коли перед ними ставиться escape-послідовність (зворотна коса риска \).

2.1.6 Системні задачі та директиви компілятора

Системні задачі (System Tasks)

Мова Verilog надає стандартний набір системних задач для виконання рутинних операцій.

Усі системні задачі мають форму: `$<ключове_слово>`.

Такі задачі застосовуються для відображення інформації на екрані, моніторингу значень сигналів, зупинення та завершення симуляції тощо. Розглянемо лише найбільш поширені системні задачі.

Виведення інформації (Displaying information)

$display – це основна системна задача для відображення значень змінних, рядків або виразів. Це одна з найкорисніших задач у Verilog.

Синтаксис:

```
$display(p1, p2, p3, ..., pn);
```

де p1, p2, ..., pn можуть бути рядками у лапках, змінними або виразами.

Формат $display дуже схожий на printf у мові C.

$display автоматично вставляє новий рядок наприкінці виведення.

Виклик $display без аргументів створює порожній рядок.

Рядки можна форматувати за допомогою специфікаторів, наведених у таблиці 2.7 (String Format Specifications). [16]

Для детальнішої інформації див. IEEE Standard Verilog HDL Specification.

Таблиця 2.7 – Специфікатори для форматування рядків

Специфікатор	Опис
%d or %D	Вивести змінну у десятковому форматі
%b or %B	Вивести змінну у двійковому форматі
%s or %S	Вивести рядок
%h or %H	Вивести змінну у шістнадцятковому форматі
%c or %C	Вивести символ ASCII
%m or %M	Вивести ієрархічне ім'я (аргумент не потрібен)
%v or %V	Вивести рівень сили сигналу
%o or %O	Вивести змінну у вісімковому форматі
%t or %T	Вивести у поточному форматі часу
%e or %E	Вивести дійсне число у науковому форматі (наприклад, 3e10)
%f or %F	Вивести дійсне число у десятковому форматі (наприклад, 2.13)
%g or %G	Вивести дійсне число у науковому або десятковому форматі (коротший з двох)

Приклад використання задачі $display:

```
// Вивести рядок у лапках
$display("Hello Verilog World");
// -- Hello Verilog World

// Вивести поточний час симуляції = 230
$display($time);
// -- 230

// Вивести 41-бітну віртуальну адресу на момент часу 200
reg [0:40] virtual_addr;
$display("At  time  %d  virtual  address  is  %h",  $time,
virtual_addr);
// -- At time 200 virtual address is 1fe0000001c

// Вивести значення port_id = 5 у двійковому форматі
reg [4:0] port_id;
$display("ID of the port is %b", port_id);
// -- ID of the port is 00101

// Вивести X-значення (конфлікт сигналів)
```

```
reg [3:0] bus;
$display("Bus value is %b", bus);
// -- Bus value is 10xx

// Вивести ієрархічне ім’я інстанса p1 у модулі top
$display("This string is displayed from %m level of hierarchy");
// -- This string is displayed from top.p1 level of hierarchy
```

Якщо змінні містять значення x або z, вони виводяться у вигляді символів "x" або "z".

Приклад використання спеціальних символів:

```
// Вивести спеціальні символи: новий рядок та знак %
$display("This is a \n multiline string with a %% sign");
// -- This is a
// -- multiline string with a % sign
```

Системна задача $monitor

$monitor відстежує зміни сигналів у симуляції.

Формат подібний до $display.

Виведення відбувається кожного разу, коли змінюється хоча б один із параметрів.

Достатньо викликати $monitor лише один раз [16].

Синтаксис:

```
$monitor(p1, p2, ..., pn);
```

Увімкнення/вимкнення моніторингу:

```
$monitoron;   // Увімкнути
$monitoroff;  // Вимкнути.
```

Приклад моніторингу часу та сигналів clock і reset

```
initial begin
    $monitor($time,
        " Value of signals clock = %b reset = %b", clock, reset);
end
```

// Частковий результат:

```
-- 0  Value of signals clock = 0 reset = 1
-- 5  Value of signals clock = 1 reset = 1
-- 10 Value of signals clock = 0 reset = 0
```

Зупинення та завершення симуляції

$stop; – призупиняє симуляцію та переводить її в інтерактивний режим для налагодження.

$finish; – повністю завершує симуляцію.

Приклад зупинення симуляції на часі 100 та завершення симуляції на часі 1000

```
initial begin
    clock = 0; reset = 1;
    #100 $stop;   // Призупиняє на t=100
    #900 $finish; // Завершує на t=1000
end
```

Таким чином:

$display – разове виведення інформації.

$monitor – автоматичне виведення за зміни сигналів.

$stop – зупинення симуляції для налагодження.

$finish – завершення симуляції.

Директиви компілятора (Compiler Directives)

У Verilog передбачено директиви компілятора. Усі директиви компілятора починаються з символу ` (backtick) перед ключовим словом.

Розглянемо дві найуживаніші директиви: `define та `include [4].

Директива `define використовується для визначення текстових макросів у Verilog. Під час компіляції текст макроса підставляється всюди, де зустрічається `<ім’я_макроса>. Це аналогічно конструкції #define у мові C. Використані константи або текстові макроси підставляються у код із символом ` перед ім’ям.

Приклад директиви `define

```
// Визначаємо текстовий макрос WORD_SIZE = 32
// Використовується у коді як `WORD_SIZE
`define WORD_SIZE 32

// Визначаємо псевдонім: скрізь, де зустрінеться `S,
// буде підставлено $stop;
`define S $stop;

// Визначаємо часто вживану текстову конструкцію
`define WORD_REG reg [31:0]
// тепер можна створювати 32-бітний регістр як: `WORD_REG
reg32;
```

Директива `include дозволяє підключати вміст одного файлу Verilog до іншого під час компіляції. Працює подібно до #include у мові C.

Зазвичай використовується для підключення заголовкових файлів, які містять глобальні або часто вживані оголошення [11].

Приклад директиви `include

```
// Підключення файлу header.v, який містить оголошення
// до основного файлу design.v
`include header.v
...
... // Verilog-код у файлі design.v
...
```

Інші директиви
`ifdef – умовна компіляція (аналог #ifdef у C).
`timescale – задання одиниць часу для симуляції.

Таким чином:
`define – створює макроси (константи, текстові скорочення).
`include – підключає зовнішні файли.

`ifdef і `timescale – додаткові інструменти для умовної компіляції та керування часом у симуляції.

2.2 Модулі та порти

2.2.1 Модулі

Модуль є базовим будівельним блоком. Модуль у Verilog складається з окремих частин, як показано на рис. 2.2.

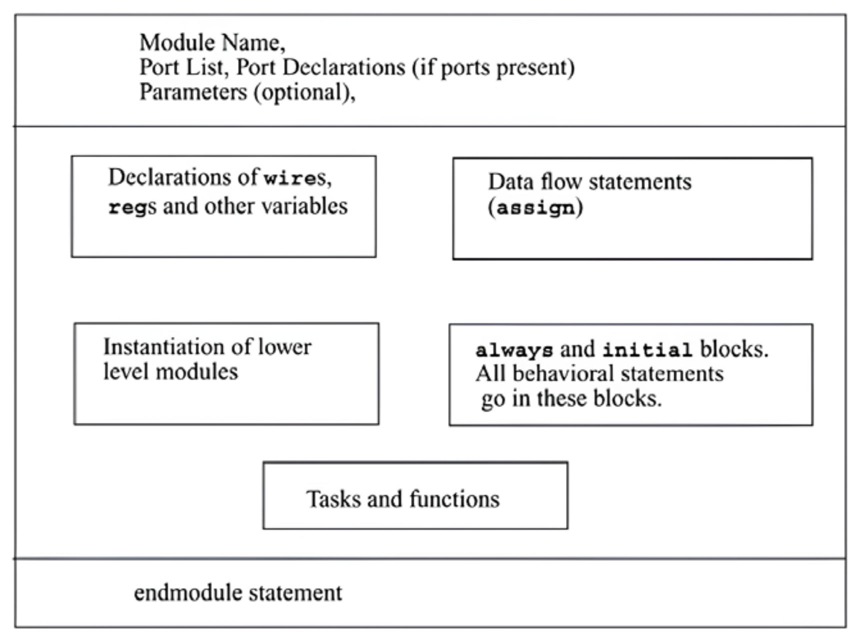


Рисунок 2.2 – Компоненти модуля

У Verilog будь-який модуль починається з ключового слова module, і закінчується завжди на endmodule. Усередині нього є чіткі обов’язкові та опціональні частини.

Обов’язкові елементи модуля:
- module – ключове слово на початку;
- назва модуля (ідентифікатор);
- список портів та їх оголошення (якщо модуль має взаємодіяти з іншими блоками);
- endmodule – закриває визначення модуля.

Опціональні елементи можуть розташовуватись у будь-якому порядку й кількості:
- оголошення змінних (reg, wire, integer, time тощо);
- операторні конструкції dataflow (наприклад, assign);
- інстанціювання інших модулів (створення підмодулів);
- поведінкові блоки (always, initial).
- задачі (tasks) та функції (functions) для повторного використання коду.

Порядок компонентів у тілі модуля не фіксований. У одному файлі можна описати кілька модулів у будь-якому порядку. Порядок у коді довільний, якщо дотримано синтаксис. Розглянемо простий приклад SR-защіпки, зображеної на рис. 2.3 [11].

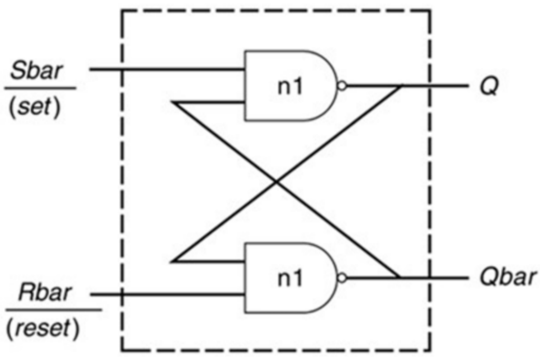


Рисунок 2.3 – SR-защіпка

SR-защіпка має S та R як вхідні порти та Q і Qbar як вихідні порти. SR-защіпка та її стимул можуть бути змодельовані, як показано у наступному прикладі:

```
// Цей приклад ілюструє різні складові модуля
// Ім’я модуля та список портів // Модуль SR_latch
module SR_latch(Q, Qbar, Sbar, Rbar);
```

```
// Оголошення портів
output Q, Qbar;
input Sbar, Rbar;

// Інстанціювання модулів нижчого рівня
// У цьому випадку інстанціюються примітиви Verilog – елементи NAND
// Зверніть увагу, як дроти з'єднані у перехресному режимі
nand n1(Q, Sbar, Qbar);
nand n2(Qbar, Rbar, Q);

// оператор endmodule
endmodule

// Ім'я модуля та список портів // Модуль Stimulus
module Top;

// Оголошення wire, reg та інших змінних
wire q, qbar;
reg set, reset;

// Інстанціювання модулів нижчого рівня
// У цьому випадку інстанціюється SR_latch
// Подаємо інвертовані сигнали set і reset на SR-защіпку
SR_latch m1(q, qbar, ~set, ~reset);

// Поведінковий блок, initial
initial begin
  $monitor($time, " set = %b, reset= %b, q= %b\n",set,reset,q);
  set = 0; reset = 0;
  #5 reset = 1;
  #5 reset = 0;
  #5 set = 1;
end

// оператор endmodule
endmodule
```

У визначенні SR-защіпки вище помітно, що всі складові, описані на рис. 2.2, не обов'язково мають бути присутні у модулі. Ми не бачимо оголошень змінних, операторів потоків даних (assign) чи поведінкових блоків (always або initial) [26].

Проте стимулювальний блок для SR-защіпки містить ім'я модуля, оголошення wire, reg та змінних, інстанціювання модулів нижчого рівня, поведінковий блок (initial) та оператор endmodule, але не містить списку портів, оголошення портів і операторів потоків даних (assign).

Таким чином, усі частини, окрім module, імені модуля та endmodule, є необов'язковими і можуть поєднуватися довільним чином відповідно до потреб проєкту.

2.2.2 Порти

Порти забезпечують інтерфейс, за допомогою якого модуль може взаємодіяти зі своїм середовищем. Наприклад, вхідні/вихідні виводи мікросхеми IC є її портами. Середовище може взаємодіяти з модулем лише через його порти. Внутрішня структура модуля є невидимою для середовища. Це надає розробнику дуже потужну гнучкість: внутрішню структуру модуля можна змінювати без впливу на середовище, доки інтерфейс залишається незмінним. Порти також називають терміналами [4].

Список портів

Визначення модуля може містити необов'язковий список портів. Якщо модуль не обмінюється сигналами із середовищем, список портів відсутній. Розглянемо 4-бітний повний суматор, який інстанціюється всередині верхнього модуля Top. Схема для вхідних/вихідних портів показана на рис. 2.4.

У проєктуванні цифрових схем через Verilog модулі спілкуються із зовнішнім середовищем за допомогою портів введення/виведення (I/O ports).

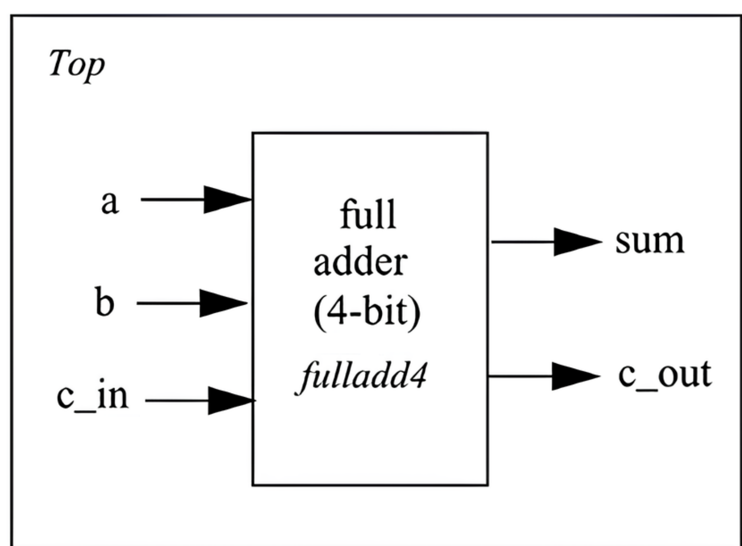


Рисунок 2.4 – Порти введення/виведення для суматора

Порти верхнього модуля (Top)

У верхньому модулі зазвичай визначаються основні сигнали, які подаються із зовнішнього середовища (наприклад, клавіші, датчики чи інші цифрові лінії) або виводяться назовні (наприклад, результат обчислень, керуючі сигнали). Верхній модуль може містити лише з'єднання й інстанціювання інших модулів, не виконуючи самостійної логіки.

Порти повного суматора (Full Adder):

Повний суматор має входи і виходи, які реалізують його функціональність. Входи: a, b (два біти, що додаються), c_in (вхідний перенос). Виходи: sum (результат додавання) та c_out (вихідний перенос).

У випадку 4-бітного повного суматора ці порти повторюються для кожного розряду, і зазвичай реалізація виконується через інстанціювання чотирьох однобітних повних суматорів [11].

Таким чином:

- Модуль Top підключає вхідні сигнали до 4-бітного суматора та отримує з нього вихідні результати.
- Модуль Full Adder виконує саму логіку додавання, використовуючи свої I/O порти як інтерфейс.

Зверніть увагу, що на рисунку вище модуль Top є верхньорівневим модулем. Модуль fulladd4 інстанціюється всередині Top.

Модуль fulladd4 приймає вхідні дані через порти a, b та c_in і формує вихід через порти sum та c_out. Таким чином, модуль fulladd4 виконує операцію додавання для свого середовища [19].

Приклад списку портів

```
module fulladd4(sum, c_out, a, b, c_in);
// Модуль зі списком портів
module Top;
// Без списку портів, верхньорівневий модуль у симуляції
```

Оголошення портів

Усі порти, зазначені у списку портів, обов’язково мають бути оголошені всередині модуля [26].

Порти можна оголошувати таким чином:

input – вхідний порт
output – вихідний порт
inout – двонаправлений порт

Кожен порт у списку портів визначається як input, output або inout, залежно від напрямку сигналу.

Отже, для прикладу fulladd4, оголошення портів виглядатиме так:

```
//Приклад оголошення портів
module fulladd4(sum, c_out, a, b, c_in);

// Початок секції оголошень портів
output [3:0] sum;
output c_cout;

input [3:0] a, b;
input c_in;
// Кінець секції оголошень портів
```

```
...
<module internals>
...

endmodule
```

В цьому випадку всі порти за замовчуванням неявно оголошуються як wire у Verilog.

Якщо порт має бути провідником (wire), достатньо оголосити його як output, input або inout.

Вхідні (input) та двонаправлені (inout) порти зазвичай оголошуються як wire.

Якщо ж вихідний порт (output) зберігає своє значення, його потрібно оголосити як reg.

Порти типу input та inout не можуть бути оголошені як reg, адже змінні reg зберігають значення, тоді як вхідні порти мають лише відображати зміни зовнішніх сигналів, до яких вони підключені.

Також модуль fulladd4 може бути оголошений у стилі ANSI C для більш компактного опису портів. У цьому випадку кожен порт оголошується одразу з його типом, іменем і розрядністю. Це дозволяє уникнути дублювання імен у визначенні модуля та у списку портів.

```
// Стиль оголошення портів у синтаксисі ANSI C
module fulladd4(output reg [3:0] sum,
                output reg c_out,
                input [3:0] a, b, // за замовчуванням wire
                input c_in);      // за замовчуванням wire
...
<module internals>
...
endmodule
```

Правила підключення портів

Уявити порт можна як такий, що складається з двох частин:

- внутрішня частина належить самому модулю і підключається до його логіки;
- зовнішня частина належить середовищу і забезпечує підключення сигналів ззовні.

Ці дві частини завжди з’єднані між собою.

Коли один модуль інстанціюється всередині іншого, існують чіткі правила з’єднання портів як показано на рис. 2.5. Якщо правила порушуються, симулятор Verilog повідомляє про помилку.

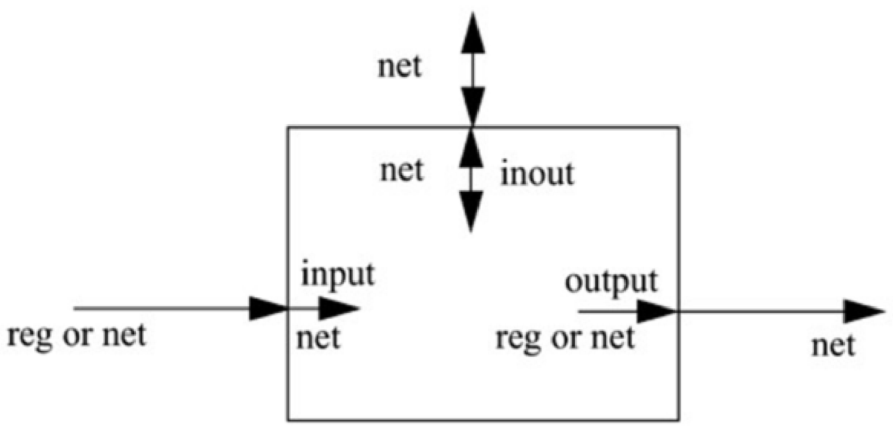


Рисунок 2.5 – Правила підключення портів

Вхідні порти (Inputs)
- Внутрішньо (inside module): завжди мають бути типу net.
- Зовнішньо (outside module): можуть підключатися як до змінних типу reg, так і до net.

Вихідні порти (Outputs)
- Внутрішньо: вихідні порти можуть бути як reg, так і net.
- Зовнішньо: завжди мають підключатися до net.
- Вихідні порти не можна підключати до reg.

Двонаправлені порти (Inouts)
- Внутрішньо: завжди мають бути типу net.
- Зовнішньо: також завжди мають підключатися до net.

Відповідність розрядності (Width matching)
- Дозволено підключати внутрішні та зовнішні сигнали різної розрядності.
- Однак симулятор зазвичай видає попередження про невідповідність ширини.

Непідключені порти (Unconnected ports)
- У Verilog дозволяється залишати порти непідключеними.
- Наприклад, вихідні порти, що використовуються лише для налагодження, можна ігнорувати [24].

```
fulladd4 fa0(SUM, , A, B, C_IN);
// Вихідний порт c_out залишено непідключеним
```

❌ Приклад неправильного підключення порту (Example 4-6)

```
module Top;

// Оголошення змінних для з'єднань
```

```
reg [3:0] A, B;
reg C_IN;
reg [3:0] SUM;   // ← помилка: SUM оголошено як reg
wire C_OUT;

// Інстанціювання fulladd4
fulladd4 fa0(SUM, C_OUT, A, B, C_IN);

// Неправильне підключення, тому що вихідний порт sum у
// fulladd4 підключений до змінної типу reg (SUM) у модулі
// Top.
...
<stimulus>
...
endmodule
```

Щоб виправити помилку, змінну SUM потрібно оголосити як wire:

```
wire [3:0] SUM;
```

Тоді підключення стане правильним і відповідатиме правилам Verilog.

Підключення портів до зовнішніх сигналів

Існує два методи встановлення з'єднань між сигналами, зазначеними в екземплярі модуля, та портами у визначенні модуля. Ці два методи не можна змішувати.

Підключення за допомогою впорядкованого списку є найбільш інтуїтивно зрозумілим методом для більшості початківців. Сигнали, що підключаються, мають з'являтися в екземплярі модуля в тому ж порядку, що й порти у списку портів у визначенні модуля. Знову ж таки, розглянемо модуль fulladd4. В ньому зовнішні сигнали SUM, C_OUT, A, B та C_IN з'являються в точно такому ж порядку, як порти sum, c_out, a, b та c_in у визначенні модуля fulladd4 [19].

```
// Приклад підключення за впорядкованим списком
module Top;

// Оголошення змінних для з'єднання
reg [3:0] A, B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;

// Інстанціювання fulladd4, екземпляр fa_ordered
```

```
    // Сигнали підключаються до портів у тому ж порядку, що й
у списку портів
    fulladd4 fa_ordered(SUM, C_OUT, A, B, C_IN);

    ...
    <stimulus>
    ...

    endmodule

    module fulladd4(sum, c_out, a, b, c_in);
      output [3:0] sum;
      output c_cout;
      input [3:0] a, b;
      input c_in;

      ...
      <module internals>
      ...

    endmodule
```

У цьому випадку порядок зовнішніх сигналів (SUM, C_OUT, A, B, C_IN) має точно відповідати порядку портів у визначенні fulladd4 (sum, c_out, a, b, c_in) [4].

Підключення портів за іменами (Named Association)

Для великих проєктів (де модулі можуть мати десятки портів) запам'ятати їхній порядок практично неможливо.

У такому випадку використовується підключення за іменами портів.

```
    // Інстанціювання fulladd4 з підключенням сигналів за
іменами
    fulladd4 fa_byname(
      .c_out(C_OUT),
      .sum(SUM),
      .b(B),
      .c_in(C_IN),
      .a(A)
    );
```

Особливості

- Порядок переліку сигналів у інстанціюванні не має значення.
- Головне – правильно вказати ім'я порту з визначення модуля.

Непідключені порти

Якщо якийсь порт не потрібен, його можна просто не вказувати під час підключення за іменами.

```
// Приклад: порт c_out залишено непідключеним
fulladd4 fa_byname(
  .sum(SUM),
  .b(B),
  .c_in(C_IN),
  .a(A)
);
```

Переваги підключення за іменами

- Не потрібно запам’ятовувати порядок портів.
- Можна вільно змінювати порядок портів у визначенні модуля (module fulladd4(...)), не змінюючи інстанціювання.
- Легше читати й супроводжувати код.
- Підходить для великих ієрархічних схем.

2.2.3 Ієрархічні імена у Verilog

Verilog підтримує ієрархічну методологію проєктування. Кожен екземпляр модуля, сигнал або змінна визначається за допомогою ідентифікатора. Кожен ідентифікатор має унікальне місце в ієрархії проєкту.

Ієрархічні імена дозволяють однозначно позначити кожен ідентифікатор у проєкті. Таке ім’я складається зі списку ідентифікаторів, розділених крапками (.), для кожного рівня ієрархії. Таким чином, будь-який ідентифікатор можна адресувати з будь-якого місця, просто вказавши повне ієрархічне ім’я [4].

Верхній рівень (top-level module) називається кореневим модулем (root module), оскільки він ніде не інстанціюється. Це початкова точка проєкту. Щоб призначити унікальне ім’я ідентифікатору, потрібно почати з верхнього рівня та простежити шлях ієрархії до потрібного ідентифікатора.

Приклад ієрархії проєкту для симуляції SR-защіпки показано на рисунку 2.6.

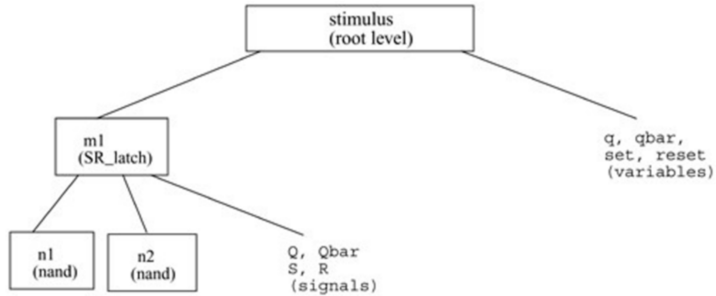


Рисунок 2.6 – Ієрархія проєкту для симуляції SR-защіпки

У симуляції SR latch, stimulus – верхній модуль (root). У ньому визначено ідентифікатори: q, qbar, set, reset. Root-модуль інстанціює m1, який є екземпляром модуля SR_latch. Усередині m1 інстанціюються елементи NAND: n1 та n2. Порти Q, Qbar, S, R належать до екземпляра m1.

Приклади ієрархічних імен:

```
stimulus
stimulus.q
stimulus.qbar
stimulus.set
stimulus.reset
stimulus.m1
stimulus.m1.Q
stimulus.m1.Qbar
stimulus.m1.S
stimulus.m1.R
stimulus.n1
stimulus.n2
```

Кожен рівень ієрархії відокремлюється крапкою.

У задачі $display для виведення рівня ієрархії використовується спеціальний формат %m.

2.3 Поведінкове моделювання

Мова Verilog надає розробникам можливість описувати функціональність у алгоритмічній формі. Іншими словами, проєктувальник описує поведінку схеми, а не її структуру на рівні вентилів [11].

Поведінкове моделювання подає схему на дуже високому рівні абстракції. Проєктування на цьому рівні більше схоже на програмування мовою С, ніж на традиційне конструювання цифрових схем.

Конструкції поведінкового рівня Verilog у багатьох аспектах нагадують конструкції мови С. Verilog багатий на поведінкові оператори, що надає розробнику велику гнучкість у описі апаратури.

2.3.1 Структуровані процедури

У Verilog існує два структуровані оператори процедур:

1) always,

2) initial.

Ці оператори є двома найосновнішими конструкціями у поведінковому моделюванні. Усі інші поведінкові оператори можуть з’являтися лише всередині цих процедур [16].

Verilog – це паралельна мова програмування, на відміну від C, яка є послідовною. У Verilog кожен блок always та initial подає окремий потік виконання. Всі ці потоки стартують одночасно за часу симуляції 0. Вкладати оператори always або initial всередину один одного заборонено.

Оператор initial

Усі оператори всередині initial становлять початковий блок (initial block). Початковий блок починає виконання за часу 0, виконується рівно один раз за всю симуляцію і більше не запускається. Якщо у модулі є кілька початкових блоків, то всі вони стартують одночасно за часу 0. Кожен блок завершується незалежно від інших [26].

Якщо в initial більше ніж один оператор, їх потрібно об’єднати за допомогою ключових слів begin ... end. Якщо оператор лише один, групування не потрібне. Це нагадує блоки begin-end у Pascal або { ... } у C.

Приклад використання initial

```
module stimulus;

  reg x, y, a, b, m;
  initial
    m = 1'b0; // один оператор; групування не потрібне
  initial
  begin
    #5  a = 1'b1; //кілька операторів; потрібне групування
    #25 b = 1'b0;
  end
  initial
  begin
    #10 x = 1'b0;
    #25 y = 1'b1;
  end
  initial
    #50 $finish;
endmodule
```

Пояснення до прикладу. Три оператори initial починають виконання одночасно за часу 0. Якщо перед оператором вказано затримку #<delay>, цей оператор виконається через <delay> тактів від поточного часу симуляції.

Послідовність виконання

```
Час  Виконуваний оператор
0    m = 1'b0;
5    a = 1'b1;
10   x = 1'b0;
30   b = 1'b0;
35   y = 1'b1;
50   $finish;
```

Початкові блоки зазвичай застосовують для: ініціалізації змінних, моніторингу, побудови часових діаграм (waveforms), процесів, які потрібно виконати лише один раз за всю симуляцію.

Існує також альтернативний скорочений синтаксис ініціалізації, який має той самий ефект, що й поєднання блока initial з оголошенням змінної.

Змінні можуть бути ініціалізовані під час їх оголошення, як наприклад ось в такому прикладі:

```
//Змінна clock оголошена першою
reg clock;
//Значення clock встановлюється у 0
initial clock = 0;

//Замість вищенаведеного методу змінну clock
//можна ініціалізувати під час оголошення
//Це дозволено тільки для змінних, оголошених
//на рівні модуля.
reg clock = 0;
```

Поєднане оголошення порту/даних також може бути поєднане з ініціалізацією як наприклад ось в такому прикладі:

```
module adder (sum, co, a, b, ci);
output reg [7:0] sum = 0; //Ініц. 8-бітного виходу sum
output reg       co  = 0; //Ініц. 1-бітного виходу co
input      [7:0] a, b;
input            ci;
-- --
endmodule
```

Оголошення порту у стилі ANSI C також може бути поєднане з ініціалізацією, як показано в наступному прикладі:

```
module adder (output reg [7:0] sum = 0,
//Ініц. 8-бітного вих
              output reg       co  = 0,
//Ініціалізація 1-бітного вих. co
              input      [7:0] a, b,
              input            ci
              );
-- --
endmodule
```

Оператор always

Усі поведінкові оператори всередині оператора always утворюють always-блок. Оператор always починається з часу 0 і виконує оператори в

always-блоці безперервно у циклічному режимі. Цей оператор використовується для моделювання блока активності, яка повторюється постійно у цифровій схемі.

Прикладом є модуль генератора тактового сигналу, який змінює стан сигналу clock кожні півперіоди. У реальних схемах генератор такту активний від часу 0 і доти, доки схема живиться. Наступний приклад ілюструє один із методів моделювання генератора тактового сигналу у Verilog.

```
// Оператор always
module clock_gen (output reg clock);
//Ініціалізація clock у момент часу нуль
initial
        clock = 1'b0;

//Зміна стану clock кожні півперіоду (період = 20)
always
        #10 clock = ~clock;

initial
        #1000 $finish;

endmodule
```

У цьому прикладі оператор always стартує у момент часу 0 і виконує оператор clock = ~clock кожні 10 одиниць часу. Варто зазначити, що ініціалізація clock має бути виконана всередині окремого оператора initial.

Якщо помістити ініціалізацію clock всередину блока always, вона буде виконуватися кожного разу під час входу в always.

Також симуляція має зупинятися всередині оператора initial. Якщо відсутні оператори $stop або $finish для зупинення симуляції, генератор такту працюватиме нескінченно [26].

Програмісти на C можуть провести аналогію між блоком always та нескінченним циклом. Але апаратні дизайнери розглядають його як безперервну діяльність у цифровій схемі, що починається від моменту подачі живлення. Активність припиняється лише за вимкнення живлення ($finish) або перериванні ($stop).

2.3.2 Процедурні присвоєння

Процедурні присвоєння оновлюють значення змінних типів reg, integer, real або time. Значення, присвоєне змінній, зберігається доти, доки інше процедурне присвоєння не оновить її новим значенням. Це відрізняється від безперервних присвоєнь, де один оператор-присвоєння змушує вираз справа постійно передаватися на ліву сторону мережі.

Синтаксис найпростішої форми процедурного присвоєння:

```
assignment::=variable_lvalue=[delay_or_event_control]expression
```

Ліва частина процедурного присвоєння <lvalue> може бути:

- змінна типу reg, integer, real, time або елемент пам'яті;
- вибір біта цих змінних (наприклад, addr[0]) ;
- вибір діапазону цих змінних (наприклад, addr[31:16]) ;
- конкатенація будь-яких із вищенаведених.

Права частина може бути будь-яким виразом, що обчислюється у значення.

Є два типи процедурних присвоєнь: блокувальні та неблокувальні.

Блокувальні процедурні присвоєння

Блокувальні оператори виконуються у тому порядку, в якому вони записані у послідовному блоці. У паралельному блоці вони не блокують інші оператори. Оператор = використовується для позначення блокувальних присвоєнь [19].

```
//Приклад блокувальні оператори
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;

//Усі поведінкові оператори мають бути всередині
//initial або always-блока
initial begin
        x = 0; y = 1; z = 1; //Присвоєння скалярів
        count = 0; //Присвоєння цілочисельної змінної
        reg_a = 16'b0; reg_b = reg_a; //Ініц. векторів

        #15 reg_a[2] = 1'b1; //Присвоєння біта з затримкою
        #10 reg_b[15:13] = {x, y, z};
          //Конкатенація в частину вектора
        count = count + 1;
          //Інкремент цілочисельної змінної end
```

Час виконання операторів:

x = 0 … reg_b = reg_a виконуються в момент часу 0;

reg_a[2] = 1 у момент часу 15;

reg_b[15:13] = {x, y, z} у момент часу 25;

count = count + 1 у момент часу 25.

Якщо у правій частині більше бітів, ніж у змінної-реєстра, зайві старші біти відкидаються.

Якщо бітів менше – старші біти доповнюються нулями.

Неблокувальні процедурні присвоєння

Неблокувальні присвоєння дозволяють планувати виконання операторів без блокування наступних у послідовному блоці.

Використовується оператор <=.

Важливо: <= може означати і «менше або дорівнює» (у виразах), і «неблокувальне присвоєння» (у контексті присвоєнь) [11].

```
//Приклад неблокувальні присвоєння
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;

//Усі поведінкові оператори мають бути всередині initial або always-блока
initial begin
        x = 0; y = 1; z = 1; //Присвоєння скалярів
        count = 0; //Присвоєння цілочисельної змінної
        reg_a = 16'b0; reg_b = reg_a; //Ініц. векторів

        reg_a[2]   <= #15 1'b1; //Присв. біта з затримкою
        reg_b[15:13] <= #10 {x, y, z};
          //Конкатенація → частина вектора
        count <= count + 1; //Інкремент end
```

У цьому прикладі:

reg_a[2] оновиться у момент часу 15

reg_b[15:13] – у момент часу 10

count оновиться одразу (у час 0)

Симулятор планує неблокувальні присвоєння і переходить до наступних операторів, не чекаючи їх завершення. Зазвичай неблокувальні оператори виконуються наприкінці поточного кроку симуляції.

Рекомендується не змішувати блокувальні й неблокувальні оператори в одному always-блоці.

Використання неблокувальних присвоєнь

Вони дозволяють моделювати одночасні передачі даних після спільної події (наприклад, фронт тактового сигналу).

Приклад:

```
always @(posedge clock) begin
    reg1 <= #1 in1;
    reg2 <= @(negedge clock) in2 ^ in3;
    reg3 <= #1 reg1; //Отримає старе значення reg1
end
```

На кожному позитивному фронті clock:
- зчитуються значення in1, in2, in3, reg1;
- плануються записи: reg1 після 1 такту, reg2 на негативному фронті, reg3 після 1 такту;
- виконуються записи у запланований час.

Таким чином, значення reg3 буде правильним (старе reg1), навіть якщо reg1 вже оновився.

Приклад гонок у блокувальних і їх усунення неблокувальними

```
//Ілюстрація 1: блокувальні оператори (гонка)
always @(posedge clock) a = b;
always @(posedge clock) b = a;

//Ілюстрація 2: неблокувальні оператори (коректний обмін)
always @(posedge clock) a <= b;
always @(posedge clock) b <= a;
```

У першому випадку порядок виконання залежить від симулятора, і a та b можуть отримати однакове значення.

У другому випадку значення гарантовано обміняються місцями.

```
Приклад емуляції неблокувальних через блокувальні
always @(posedge clock) begin
   //Операція читання
   temp_a = a;
   temp_b = b;

   //Операція записування
   a = temp_b;
   b = temp_a;
end
```

Для цифрового проєктування використання неблокувальних присвоєнь замість блокувальних дуже рекомендується у тих місцях, де після спільної події відбуваються одночасні передачі даних [4].

У таких випадках блокувальні присвоєння можуть потенційно спричинити гонки (race conditions), оскільки кінцевий результат залежатиме від порядку, в якому оператори виконуються.

Неблокувальні присвоєння можна ефективно використовувати для моделювання паралельних передач даних, адже їхній кінцевий результат не залежить від порядку виконання.

Типові області застосування неблокувальних присвоєнь:
- моделювання конвеєрів (pipeline modeling);
- моделювання кількох взаємовиключних передач даних.

Недоліком неблокувальних присвоєнь є те, що вони можуть призвести до зниження продуктивності симулятора та збільшення використання пам’яті.

2.3.3 Керування часом

У Verilog доступні різні конструкції поведінкового керування часом. Якщо у Verilog відсутні оператори керування часом, симуляційний час не рухається вперед. Керування часом дозволяє вказати момент симуляційного часу, в який буде виконано процедурний оператор.

Існує три методи керування часом:
- керування часом на основі затримки (delay-based timing control);
- керування часом на основі події (event-based timing control);
- керування часом на основі рівня сигналу (level-sensitive timing control).

Керування часом на основі затримки (Delay-Based Timing Control)

У виразі керування часом на основі затримки вказується інтервал між моментом, коли оператор був зустрінутий у коді, і моментом його виконання. Затримки задаються символом #.

Синтаксис:

```
delay3 ::= # delay_value
         | # ( delay_value [ , delay_value [ , delay_value ] ] )

delay2 ::= # delay_value
         | # ( delay_value [ , delay_value ] )

delay_value ::= unsigned_number
              | parameter_identifier
              | specparam_identifier
              | mintypmax_expression
```

Затримка може бути задана числом, ідентифікатором або mintypmax_expression.

Для процедурних присвоєнь існують три типи керування затримками:
- звичайне (regular delay control);
- внутрішньоприсвоювальне (intra-assignment delay control);
- нульова затримка (zero delay control).

Звичайне керування затримкою (Regular Delay Control) використовується, коли ненульова затримка задається зліва від оператора присвоєння. Наприклад:

```
//визначення параметрів
parameter latency = 20;
parameter delta   = 2;

//визначення регістрів
reg x, y, z, p, q;

initial begin
        x = 0; // без затримки
        #10 y = 1; // затримка з числом.
                   //Виконати y=1 через 10 тактів
         #latency z = 0; // затримка з ідентифікатором
                        //(20 тактів)
        #(latency + delta) p = 1; // затримка виразом
        #y x = x + 1; // затримка з ідентифікатором.
                   // Береться значення y
        #(4:5:6) q = 0; // мінімальне, типове та
                        // максимальне значення затримки
end
```

У цьому прикладі виконання операторів відтерміновується на вказану кількість тактів. Наприклад, y = 1 буде виконано через 10 одиниць часу після того, як симулятор дійде до цього оператора.

Внутрішньоприсвоювальне керування затримкою (Intra-assignment Delay Control)

Замість вказання затримки зліва від присвоєння, можна задати затримку справа. Така форма інакше впливає на порядок виконання. Наприклад:

```
//визначення регістрів
reg x, y, z;

//використання внутрішньоприсвоювальних затримок
initial begin
        x = 0;
        z = 0;
        y = #5 x + z; //взяти значення x i z у час=0,
        //обчислити x+z і призначити його y через 5 тактів
end

//еквівалентний метод із тимчасовою змінною та
//звичайною затримкою
initial begin
        x = 0; z = 0;
```

```
        temp_xz = x + z;
        #5 y = temp_xz; //береться x+z у поточний момент
          //часу та зберігається у temp_xz
          //навіть якщо x і z зміняться між 0 та 5,
          //y отримає значення temp_xz у момент часу 5
end
```

Різниця

Звичайна затримка відкладає виконання всього присвоєння.

Внутрішньоприсвоювальна затримка спочатку обчислює праву частину у поточний момент, а потім відкладає лише присвоєння значення.

Це еквівалентно використанню проміжної змінної для збереження результату.

Нульова затримка (Zero Delay Control)

Процедурні оператори в різних блоках always або initial можуть бути оцінені в один і той самий момент симуляційного часу. Порядок їх виконання в різних блоках не є визначеним [16].

Нульова затримка (#0) – це метод, який дозволяє гарантувати, що оператор буде виконано останнім, після того як усі інші оператори на цьому кроці симуляції вже виконані.

Це використовується для усунення гонок (race conditions).

Однак якщо є декілька операторів з нульовою затримкою, порядок їх виконання також недетермінований.

```
// Приклад нульової затримки
initial begin
    x = 0;
    y = 0;
end

initial begin
    #0 x = 1; //нульова затримка
    #0 y = 1;
end
```

У цьому прикладі чотири оператори – x = 0, y = 0, x = 1, y = 1 – мають бути виконані в момент часу 0. Але оскільки x = 1 і y = 1 мають #0, вони будуть виконані після перших двох операторів. Отже, в кінці моменту часу 0, x = 1 і y = 1. Порядок, у якому саме виконаються x = 1 і y = 1, не визначений.

Цей приклад наведений лише для ілюстрації. Використання #0 не рекомендується у практичному проєктуванні.

Керування часом на основі подій (Event-Based Timing Control)

Подія – це зміна значення регістра або шини (net).

Події можуть використовуватися для запуску виконання окремого оператора або цілого блока [16].

У Verilog є чотири типи керування часом на основі подій:

- звичайне керування подією (regular event control);
- іменоване керування подією (named event control);
- керування подіями через OR (event OR control);
- рівнево-чутливе керування часом (level-sensitive timing control).

Звичайне керування подією (Regular Event Control)

Символ @ використовується для вказання керування подією.

Оператори можна виконувати:

- за будь-якої зміни сигналу;
- у разі позитивного фронту (posedge);
- у випадку негативного фронту (negedge).

```
// Звичайне керування подією
@(clock) q = d; // q=d викон. за будь-якої зміни clock
@(posedge clock) q = d; // q = d виконується за переходу
                    // 0→1, x→1, z→1
@(negedge clock) q = d; // q = d виконується за переходу
                    // 1→0, x→0, z→0
q = @(posedge clock) d; // d обчислюється одразу
          // і присвоюється q на позитивному фронті clock
```

Іменоване керування подією (Named Event Control)

Verilog дозволяє оголошувати події й потім запускати та відстежувати їх настання.

Оголошення події: event

Запуск події: ->

Відстеження події: @

```
// Іменоване керування подією
// Буфер даних зберігає дані після отримання останнього
пакета

event received_data; // оголошення події

always @(posedge clock) begin
   if(last_data_packet)
      ->received_data; // запустити подію
end
```

```
always @(received_data)
     // коли подія спрацює, записати всі 4 пакети у буфер
data_buf={data_pk[0],data_pk[1],data_pk[2],data_pk[3]};
```

Керування подіями через OR (Event OR Control, Sensitivity List)

Іноді зміна будь-якого з кількох сигналів чи подій може запустити виконання блока [11].

Це виражається як OR сигналів у списку чутливості (sensitivity list).

```
// Керування подіями через OR
// Рівнево-чутливий тригер з асинхронним скиданням
always @(reset or clock or d) begin
     if (reset)
             q = 1'b0;   // якщо reset=1 → q=0
     else if(clock)
            q = d;      // якщо clock=1 → запам'ятати вхід
end
```

Використання коми у списках чутливості

Окрім or, можна застосовувати оператор ,.

Він також працює зі списками, де є edge-чутливі тригери.

```
// Список чутливості з комою
// Рівнево-чутливий тригер з асинхронним скиданням
always @(reset, clock, d) begin
        if (reset)
              q = 1'b0;
        else if(clock)
              q = d;
end

// D-тригер з позитивним фронтом
// та асинхронним скиданням (по спаду)
always @(posedge clk, negedge reset)
        if(!reset) q <= 0;
        else       q <= d;
```

Спрощення для комбінаційної логіки — @ і @(*)*

У випадку великої кількості входів список чутливості стає громіздким. Якщо забути хоч один вхід, блок не працюватиме як комбінаційна логіка.

Для вирішення цієї проблеми введено спеціальні символи: @* та @(*). Обидва автоматично враховують усі сигнали, що зчитуються в блоці.

```
// Приклад використання @*

// Складно писати вручну:
always @(a or b or c or d or e or f or g or h or p or m)
begin
        out1 = a ? b+c : d+e;
        out2 = f ? g+h : p+m;
end

//Замість методу вище використовуй символ @(*)
//Альтернативно можна застосовувати символ @*
//Усі вхідні змінні автоматично включаються
//до списку чутливості
// Коротко і безпечно:
always @(*) begin
        out1 = a ? b+c : d+e;
        out2 = f ? g+h : p+m;
end
```

Таблиця 2.8 – Порівняння **@(…)** та **@(*)** у Verilog

Конструкція	**Сфера застосування**	**Переваги**	**Недоліки / Ризики**
`@(…)` (список чутливості вручну)	Використовується у **процесуальних блоках** (`always`), де потрібно чітко вказати сигнали, які запускають виконання блока	- Повний контроль над переліком сигналів - Можна вказувати як **рівневі** (`a or b`), так і **фронтові** (`posedge clk, negedge rst`) чутливості	- Якщо забути включити якийсь сигнал → поведінка буде відрізнятись від очікуваної - За великої кількості сигналів список стає громіздким
`@(*)` або `@*` (автоматичний список чутливості)	Використовується у **комбінаційній логіці**, де блок має реагувати на **всі** вхідні сигнали	- Автоматично включає всі сигнали, які читаються у блоці - Захищає від помилок (неможливо «забути» сигнал) - Значно зручніше за великої кількості входів	- Можна використовувати **тільки для комбінаційної логіки** - Не підходить, якщо потрібна реакція **лише на конкретні сигнали** (наприклад, тільки на `posedge clk`)

Отже:

- використовуйте @(…) там, де потрібно точно контролювати чутливість, особливо у синхронних схемах (тригери, лічильники);
- використовуйте @(*) у комбінаційній логіці, щоб уникнути помилок зі списками сигналів і зробити код компактним [26].

Рівнево-чутливе керування часом (Level-Sensitive Timing Control)

Раніше @ дозволяв чекати зміни сигналу (edge-sensitive control).

Але Verilog також підтримує очікування рівня сигналу, тобто виконання оператора лише тоді, коли певна умова істинна.

Використовується ключове слово wait.

```
always
    wait (count_enable) #20 count = count + 1;
```

Якщо count_enable = 0, оператор не виконується.

Якщо count_enable = 1, тоді count = count + 1 виконається через 20 тактів.

Якщо count_enable лишається таким, що дорівнює 1, то лічильник зростатиме кожні 20 тактів.

2.3.4 Умовні оператори

Умовні оператори використовуються для прийняття рішень на основі певних умов. Ці умови визначають, чи буде виконано оператор [4].

Для умовних операторів застосовуються ключові слова if та else.

Є три типи умовних операторів.

Форми запису:

```
//Тип 1: без else.
//Оператор виконується або не виконується.
if (<expression>) true_statement;

//Тип 2: з одним else.
//Виконується або true_statement, або false_statement.
if (<expression>) true_statement;
else false_statement;

//Тип 3: вкладений if-else-if.
//Вибір між кількома операторами, виконується лише один.
if (<expression1>) true_statement1;
else if (<expression2>) true_statement2;
else if (<expression3>) true_statement3;
else default_statement;
```

Вираз <expression> обчислюється. Якщо він істинний (1 або будь-яке ненульове значення) → виконується true_statement. Якщо він хибний (0) або невизначений (x) → виконується false_statement.

Кожен true_statement або false_statement може бути:

- окремим оператором

або

- блоком кількох операторів (у цьому випадку блок групується словами begin і end).

```
// Умовні оператори
//Тип 1
if(!lock) buffer = data;
if(enable) out = in;

//Тип 2
if (number_queued < MAX_Q_DEPTH) begin
        data_queue = data;
        number_queued = number_queued + 1;
```

```
end else
        $display("Queue Full. Try again");

//Тип 3
//Вибір дії на основі сигналу керування ALU
if (alu_control == 0)
        y = x + z;
else if(alu_control == 1)
        y = x - z;
else if(alu_control == 2)
        y = x * z;
else
        $display("Invalid ALU control signal");
```

2.3.5 Багатоваріантне розгалуження

У попередньому пункті ми бачили приклад вкладених if-else-if. Якщо альтернатив занадто багато, такий підхід стає громіздким. Скорочений спосіб досягти того самого результату – використати оператор case [26].

Оператор case

Використовуються ключові слова case, endcase та default.
Загальна форма:

```
case (expression)
    alternative1: statement1;
    alternative2: statement2;
    alternative3: statement3;
    ...
    default: default_statement;
endcase
```

Вирази-стани statement1, statement2, default_statement можуть бути як одним оператором, так і блоком із кількох операторів (у такому випадку їх потрібно об’єднувати через begin … end).

Вираз у case порівнюється з альтернативами послідовно, і виконується перший збіг. Якщо збігів немає – виконується default. default необов’язковий, але двох default у одному case не може бути.

Оператори case можна вкладати один в один.

```
// Реалізація вибору дії для ALU
//(альтернатива до вкладених if-else):
reg [1:0] alu_control;
...
case (alu_control)
  2'd0 : y = x + z;
  2'd1 : y = x - z;
```

```
  2'd2 : y = x * z;
  default : $display("Invalid ALU control signal");
endcase
```

Використання case для мультиплексора

Оператор case зручно застосовувати для моделювання мультиплексорів.

```
// Приклад 4-до-1 мультиплексора
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Оголошення портів
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;
always @(s1 or s0 or i0 or i1 or i2 or i3)
case ({s1, s0})// Вибір за конкатенацією сигн. керування
    2'd0 : out = i0;
    2'd1 : out = i1;
    2'd2 : out = i2;
    2'd3 : out = i3;
    default: $display("Invalid control signals");
endcase
endmodule
```

Обробка x та z у case

Під час порівняння у case значення 0, 1, x, z порівнюються побітово.

Якщо ширина операндів різна – вони доповнюються нулями до більшої.

```
// Приклад демультиплексора 1-до-4
module demultiplexer1_to_4 (out0, out1, out2, out3, in, s1, s0);
output out0, out1, out2, out3;
reg  out0, out1, out2, out3;
input in;
input s1, s0;
always @(s1 or s0 or in)
case ({s1, s0})
2'b00 :
     begin
     out0 = in;  out1 = 1'bz; out2 = 1'bz; out3 = 1'bz;
     end
2'b01 :
     begin
     out0 = 1'bz; out1 = in;  out2 = 1'bz; out3 = 1'bz;
     end
2'b10 :
```

```
        begin
        out0 = 1'bz; out1 = 1'bz; out2 = in;  out3 = 1'bz;
        end
    2'b11 :
        begin
        out0 = 1'bz; out1 = 1'bz; out2 = 1'bz; out3 = in;
        end

    // Облік невизначених сигналів select
    2'bx0, 2'bx1, 2'bxz, 2'bxx, 2'b0x, 2'b1x, 2'bzx :
        begin
        out0 = 1'bx; out1 = 1'bx; out2 = 1'bx; out3 = 1'bx;
        end
    2'bz0, 2'bz1, 2'bzz, 2'b0z, 2'b1z :
        begin
        out0 = 1'bz; out1 = 1'bz; out2 = 1'bz; out3 = 1'bz;
        end

    default: $display("Unspecified control signals");
    endcase
    endmodule
```

В цьому прикладі також показано кілька альтернатив, що ведуть до одного результату, можна об'єднувати через кому.

Ключові слова casex та casez

Є дві варіації оператора case:

- casez – усі значення z у виразі чи альтернативах сприймаються як don't care. Можна також використовувати символ ? як синонім z.
- casex – усі значення x і z сприймаються як don't care. Це дозволяє порівнювати лише значущі біти, ігноруючи невизначені.

```
// Використання casex
reg [3:0] encoding;
integer state;

casex (encoding) // логічне значення x означає don't care
    4'b1xxx : next_state = 3;
    4'bx1xx : next_state = 2;
    4'bxx1x : next_state = 1;
    4'bxxx1 : next_state = 0;
    default : next_state = 0;
endcase
```

Наприклад, encoding = 4'b10xz → спрацює перший шаблон (4'b1xxx), і next_state = 3.

В табл. 2.9 наведено порівнянн використання різних варіантів розгалуджень.

Таблиця 2.9 – Порівняльна таблиця if-else vs case / casex / casez у Verilog

Конструкція	Коли застосовувати	Переваги	Недоліки
`if-else`	- Коли умови мають **нерівності**, діапазони, складні вирази - Коли потрібно зробити **послідовну перевірку** (пріоритетність умов)	- Гнучкість (будь-який логічний вираз) - Добре підходить для **пріоритетних кодувальників** та **умов з діапазонами**	- У разі великої кількості гілок стає громіздким і важко читати - Легко помилитися з пріоритетами
`case`	- Коли вибір базується на **точному значенні сигналу** (декодування, мультиплексор) - Коли потрібні багато варіантів вибору (multiway branching)	- Чистий, компактний синтаксис - Легко зрозуміти як **мультиплексор** або **дешифратор** - Автоматично не має пріоритету – всі альтернативи рівноправні	- Якщо умова складна (діапазон, нерівності) → `case` непридатний
`casez`	- Коли у виразі або альтернативах можуть бути **невідомі `z`**, які потрібно трактувати як **don't care**	- Зручно для кодування схем з «масками» (наприклад, декодери адрес) - Можна використовувати символ `?` для скорочення запису	- Ігнорує лише `z`, але не `x` - Може приховати логічні помилки
`casex`	- Коли у виразі чи альтернативах можливі як `x`, так і `z`, які потрібно трактувати як **don't care**	- Дуже зручно для FSM або складних декодерів, де важливі лише деякі біти	- Ігнорує і **`x`,** і **`z`** – це може приховати небажані стани та помилки - Не рекомендується для синтезу (може призвести до неоптимальної логіки)

2.3.6 Цикли

У Verilog є чотири типи циклів:
- while;
- for;
- repeat;
- forever.

Синтаксис цих циклів подібний до циклів у мові С. Всі оператори циклів можуть з'являтися тільки всередині initial або always блока. У циклах можна використовувати затримки (delay expressions).

Цикл while

Ключове слово while задає цей цикл. Він виконується, доки умова (while-expression) істинна. Якщо під час входу в цикл умова вже хибна, тіло циклу не виконується жодного разу. Умовний вираз може містити будь-які логічні оператори. Якщо у тілі циклу більше одного оператора, вони групуються за допомогою begin ... end [4].

```
//Приклад While Loop
// 1: Інкремент від 0 до 127. Вихід при count=128.
integer count;
initial begin
    count = 0;
    while (count < 128) begin
        $display("Count = %d", count);
        count = count + 1;
    end
end

// 2: Пошук першого біта зі значенням 1 у векторі flag
'define TRUE 1'b1
'define FALSE 1'b0
reg [15:0] flag;
integer i;
reg continue;

initial begin
    flag = 16'b0010_0000_0000_0000;
    i = 0;
    continue = 'TRUE;

    while((i < 16) && continue ) begin
        if (flag[i]) begin
            $display("Encountered a TRUE bit at element
number %d", i);
            continue = 'FALSE;
        end
        i = i + 1;
    end
end
```

Цикл for

Ключове слово for використовується для циклів із фіксованою кількістю ітерацій [11].

Структура циклу складається з:
- початкової умови,
- перевірки умови завершення,
- оновлення змінної-лічильника.

Цикл for компактніший за while, але менш універсальний.

```
// Приклад For Loop
integer count;
initial
    for (count=0; count < 128; count = count + 1)
        $display("Count = %d", count);
```

```
Ініціалізація масиву:

'define MAX_STATES 32
integer state [0:'MAX_STATES-1];
integer i;

initial begin
   for(i = 0; i < 32; i = i + 2) //всі парні індекси → 0
       state[i] = 0;
   for(i = 1; i < 32; i = i + 2) //всі непарні індекси → 1
       state[i] = 1;
end
```

Використовуйте for, коли є чіткі межі ітерацій. Для умовних повторів краще підходить while.

Цикл repeat

Ключове слово repeat вказує виконати цикл фіксовану кількість разів. На відміну від while, воно не може мати логічну умову – тільки число (константа, змінна або сигнал). Це число обчислюється лише на початку [19].

```
//Приклад Repeat Loop
// 1: рахунок від 0 до 127
integer count;
initial begin
    count = 0;
    repeat(128) begin
        $display("Count = %d", count);
        count = count + 1;
    end
end

// 2: Буфер даних
//Зберігає дані протягом 8 тактів після сигналу data_start
module data_buffer(data_start, data, clock);

parameter cycles = 8;
input data_start;
input [15:0] data;
input clock;

reg [15:0] buffer [0:7];
integer i;

always @(posedge clock) begin
  if(data_start) begin
    i = 0;
```

```
    repeat(cycles) begin
      @(posedge clock) buffer[i] = data;
      i = i + 1;
    end
  end
end

endmodule
```

Цикл forever

Ключове слово forever означає цикл без умови, який виконується нескінченно. Вихід можливий лише через disable або $finish. По суті, це еквівалент while(1). Застосовується для безперервних процесів, зазвичай з таймінгом. Якщо не вказати затримку чи подію, симулятор зациклиться і зупинить решту дизайну [24].

```
//Приклад Forever Loop
//Приклад 1: Генерація тактового сигналу
reg clock;
initial begin
    clock = 1'b0;
    forever #10 clock = ~clock; // період 20
end

//Приклад 2: Синхронізація регістрів
reg clock;
reg x, y;
initial
    forever @(posedge clock) x = y;
```

Таким чином:

- while – універсальний цикл з умовою.
- for – зручно, коли відома кількість ітерацій.
- repeat – простий спосіб повторити фіксовану кількість разів.
- forever – безкінечний цикл для генерації сигналів чи постійних процесів.

В табл. 2.10 наведено порівняння чотирьох типів циклів.

Таблиця 2.10 – Порівняльна таблиця 4-х типів циклів у Verilog

Цикл	Синтаксис	Коли застосовувати	Переваги	Обмеження
while	`while (умова) begin ... end`	Коли кількість ітерацій **не відома наперед**, цикл виконується доки умова істинна	**- Гнучкий - Можна комбінувати кілька умов - Підходить для пошуку, перевірки, зупинення за**	- Може не виконатись жодного разу - Легко зробити нескінченний цикл, якщо умова не оновлюється

Цикл	Синтаксис	Коли застосовувати	Переваги	Обмеження
			першого збігу	
`for`	`for(init; умова; update) begin ... end`	Коли є **чіткі межі ітерацій** (наприклад, від 0 до N)	- Компактний запис - Зручний для ініціалізації масивів, пам'яті, лічильників	- Менш універсальний за `while` - Не підходить, якщо умова не має фіксованого ліміту
`repeat`	`repeat(N) begin ... end`	Коли потрібно виконати блок **фіксовану кількість разів** (N відоме наперед або сигнал на початку)	- Дуже простий запис - Зручно для моделювання буферів або повторів	- Умова не перевіряється під час виконання - N обчислюється лише один раз на старті
`forever`	`forever begin ... end`	Коли процес має йти **безперервно** (наприклад, генерація тактового сигналу)	- Найпростіший для нескінченних процесів - Зручно з таймінгом або подіями	- Без затримки симулятор зациклиться і зупинить решту дизайну - Вихід можливий лише через `disable` або `$finish`

2.3.7 Послідовні та паралельні блоки

Блокові оператори використовуються, щоб згрупувати кілька операторів і виконувати їх як один.

У попередніх прикладах ми вже застосовували begin ... end для групування кількох операторів. Це – послідовні блоки, у яких оператори виконуються один за одним [19].

У Verilog існує два типи блоків:

- послідовні (sequential blocks);
- паралельні (parallel blocks).

Також доступні спеціальні можливості: іменовані блоки, зупинка іменованих блоків та вкладені блоки.

Послідовні блоки

Задаються ключовими словами begin та end. Оператори виконуються у порядку запису (один за одним). Якщо є затримка чи подія, вони відлічуються від часу завершення попереднього оператора.

```
//Приклад Sequential Blocks
//Ілюстрація 1: Без затримок
reg x, y; reg [1:0] z, w;
initial begin
    x = 1'b0;
    y = 1'b1;
    z = {x, y};
    w = {y, x};
```

```
end

//Ілюстрація 2: Затримки
reg x, y; reg [1:0] z, w;
initial begin
    x = 1'b0;          // час 0
    #5  y = 1'b1;      // час 5
    #10 z = {x, y};    // час 15
    #20 w = {y, x};    // час 35
end
```

Паралельні блоки

Вони задаються ключовими словами fork і join. Всі оператори починають виконуватись одночасно у момент входу в блок [16].

Тривалість залежить від затримок кожного оператора.

```
// Приклад Parallel Blocks
reg x, y; reg [1:0] z, w;
initial fork
    x = 1'b0;         // час 0
    #5  y = 1'b1;     // час 5
    #10 z = {x, y};   // час 10
    #20 w = {y, x};   // час 20
join
```

Результат – блок завершується на часі 20, а не 35, як у послідовному випадку.

Обережно: можливі гонки (race conditions), якщо кілька операторів одночасно впливають на одні й ті самі змінні.

Вкладені блоки

Блоки можна вкладати один у інший. Можна змішувати послідовні та паралельні.

```
// Приклад Nested Blocks
initial begin
    x = 1'b0;
    fork
        #5  y = 1'b1;
        #10 z = {x, y};
    join
    #20 w = {y, x};
end
```

Найменування блоків

Блок можна назвати (begin: name, fork: name). Всередині іменованого блока можна оголошувати локальні змінні. До змінних можна звертатися за допомогою ієрархічного імені. Іменовані блоки можна відключати (disable) [16].

```
//Приклад Named Blocks
//Модуль із двома іменованими блоками
module top;
    initial begin: block1 // послідовний блок
        integer i;
        // доступ через top.block1.i
    end

    initial fork: block2 // паралельний блок
        reg i;
        // доступ через top.block2.i
    join
endmodule
```

Відключення іменованих блоків

Ключове слово disable зупиняє виконання іменованого блока. Це схоже на break у C, але потужніше: можна зупинити будь-який іменований блок.

```
//Приклад використання disable
reg [15:0] flag;
integer i;

initial begin
  flag = 16'b0010_0000_0000_0000;
  i = 0;
  begin: block1 // названий блок усередині while
    while(i < 16) begin
        if (flag[i]) begin
            $display("Found TRUE bit at %d", i);
            disable block1; // вихід з блоку
        end
        i = i + 1;
    end
  end
end
```

В табл. 2.11 наведено порівняння для послідовних (sequential) та паралельних (parallel) блоків у Verilog.

Таблиця 2.11 – Порівняльна таблиця для послідовних (sequential) та паралельних (parallel) блоків у Verilog

Характеристика	Послідовні блоки (`begin ... end`)	Паралельні блоки (`fork ... join`)
Порядок виконання	Оператори виконуються **один за одним**, строго зверху вниз	Усі оператори запускаються **одночасно** під час входу в блок
Затримки та події	Відліковуються **від завершення попереднього оператора**	Відліковуються **від моменту входу в блок**
Час завершення блоку	Дорівнює сумі затримок усіх операторів у порядку виконання	Дорівнює часу **найдовшого** оператора у блоці
Приклад	Логіка, що має виконуватись **поетапно** (наприклад, послідовна ініціалізація)	Логіка, яка може виконуватись **паралельно** (наприклад, кілька сигналів з різними затримками)
Ризики	В основному немає, поведінка завжди визначена	Можливі **гонки (race conditions)**, якщо кілька операторів змінюють одну змінну одночасно
Зручність	Простий та передбачуваний	Гнучкий, але потребує обережності під час роботи з одними й тими самими змінними

2.3.8 Generate-оператори

Generate-оператори дозволяють формувати код Verilog динамічно ще на етапі elaboration (підготовки до симуляції) [26].

Це спрощує створення параметризованих моделей, коли:

- одна й та сама операція чи інстанс модуля повторюється для багатьох бітів вектора;

- частина коду включається умовно, залежно від параметрів.

Всередині generate ... endgenerate можна створювати:

- модулі,
- користувацькі або вбудовані примітиви,
- безперервні призначення (continuous assignments),
- блоки initial та always.

У generate дозволяється також оголошувати змінні типів: net, reg, integer, real, time, realtime, event. Усі ці сутності отримують унікальні ієрархічні імена.

В generate не можна оголошувати: parameters, localparam, порти input/output, блоки specify.

Три способи використання generate:

- Generate loop (повторення через цикл);
- Generate conditional (умовне включення);
- Generate case (вибір із кількох варіантів).

Generate цикл

Дозволяє багаторазово створювати: змінні, модулі, примітиви, асайнменти, initial/always.

Застосовується цикл for з індексом типу genvar.

Genvar існує тільки на етапі розгортання коду, а не під час симуляції.

```
// Цей модуль генерує побітовий XOR двох N-бітових шин
module bitwise_xor (out, i0, i1);

// Оголошення параметра. Може бути перевизначено
parameter N = 32; // За замовчуванням 32-бітна шина
// Оголошення портів
output [N-1:0] out;
input  [N-1:0] i0, i1;

// Тимчасова змінна для циклу. Використовується тільки
// під час розгортання generate-блоків. У симуляції
// цієї змінної не існує
genvar j;

// Генеруємо побітовий XOR через цикл
generate
  for (j=0; j<N; j=j+1) begin: xor_loop
    xor g1 (out[j], i0[j], i1[j]);
  end // кінець циклу for усередині generate
endgenerate // кінець generate-блоку


// --- Альтернативний стиль ---
// Замість логічних елементів XOR можна
// використати always-блоки
// reg [N-1:0] out;
//generate
//  for (j=0; j<N; j=j+1) begin: bit
//    always @(i0[j] or i1[j])
//      out[j] = i0[j] ^ i1[j];
//  end
//endgenerate

endmodule
```

Проведемо аналіз вищенаведеного прикладу. Перед початком симуляції симулятор виконує elaboration (розгортання). Код у generate-блоках перетворюється на плоский список звичайних операторів Verilog без циклів. Потім симуляція виконується вже над цим розгорнутим кодом.

Таким чином, generate-блоки – це зручний спосіб замінити десятки повторюваних операторів одним циклом [24].

Genvar – це спеціальне ключове слово для оголошення змінних, що застосовуються лише під час розгортання generate-блоку.

У реальній симуляції вони не існують.

Значення змінної genvar може визначатися тільки всередині generate-циклу.

Generate-цикли можна вкладати, але дві вкладені конструкції не можуть одночасно використовувати один і той самий genvar як індекс.

Ім’я xor_loop, яке ми дали блоку begin: xor_loop, використовується для ієрархічного доступу. Наприклад: xor_loop[0].g1, xor_loop[1].g1, … , xor_loop[31].g1. Тобто кожен інстанс XOR отримує унікальне ім’я у структурі дизайну. [4]

Цикли generate є досить гнучкими. Усередині них можна використовувати різні конструкції Verilog. Важливо розуміти, як виглядатиме опис Verilog після розгортання generate-циклу. Це дає більш чітке розуміння поведінки generate-циклів. Наступний приклад демонструє згенерований каскадний суматор (ripple adder) з оголошенням локальних з’єднань (net) всередині generate-циклу.

```
    // Приклад Ripple Adder з використанням generate
    // Цей модуль генерує суматор на логічних елементах
(рівень гейтів)

    module ripple_adder(co, sum, a0, a1, ci);

    // Оголошення параметра. Може бути перевизначено
    parameter N = 4; // За замовчуванням 4-бітна шина

    // Оголошення портів
    output [N-1:0] sum;
    output co;
    input  [N-1:0] a0, a1;
    input ci;

    // Локальна шина для переносів
    wire [N-1:0] carry;

    // Присвоєння: молодший перенос дорівнює вхідному ci
    assign carry[0] = ci;

    // Тимчасова змінна циклу (існує тільки
    // на етапі розгортання)
    genvar i;

    // Generate-блок: створюємо послідовність суматорів
    generate
      for (i=0; i<N; i=i+1) begin: r_loop

    // Локальні проміжні сигнали для кожного біта
```

```
wire t1, t2, t3;

// Побудова однорозрядного суматора:
xor g1 (t1, a0[i], a1[i]); // попередня сума без переносу
xor g2 (sum[i], t1, carry[i]); // результатний біт суми
and g3 (t2, a0[i], a1[i]);// частина для обчисл. переносу
and g4 (t3, t1, carry[i]);
or  g5 (carry[i+1], t2, t3);// перенос для наступного біта
end
endgenerate

// Вихідний перенос (co) дорівнює останньому carry
assign co = carry[N];
endmodule
```

В цьому прикладі кожна ітерація циклу for створює повний однорозрядний суматор (full adder). Змінна genvar i використовується тільки для розгортання. Після elaboration код стає «плоским»: ніби ми написали вручну чотири однорозрядних суматори. У кожному блоці r_loop[i] створюються проміжні сигнали t1, t2, t3 для локальних обчислень.

Ієрархічні імена інстансів виглядають так:

r_loop[0].g1, r_loop[0].g2, ...

r_loop[1].g1, r_loop[1].g2, ...

r_loop[2].g1, ...

r_loop[3].g1, ...

Те ж саме стосується і локальних сигналів: r_loop[0].t1, r_loop[1].t2 тощо. За допомогою generate for ми можемо описати N-бітний ripple-carry adder у компактній формі. Після розгортання симулятор сприймає це як звичайний набір інстансів гейтів без циклів.

Generate Conditional

Умовний generate – це конструкція на кшталт if-else-if, яка дозволяє умовно інстанціювати такі елементи в інший модуль залежно від виразу, що визначається на етапі elaboration (розгортання проєкту перед симуляцією):

- модулі;
- користувацькі примітиви, логічні гейти;
- оператори безперервних присвоєнь (assign);
- блоки initial і always.

```
// Параметризований множник із використанням Generate Conditional
// Цей модуль реалізує параметризований множник
module multiplier (product, a0, a1);
// Оголошення параметрів. Можна перевизначати під час інстанціювання
```

```
parameter a0_width = 8;   // за замовчуванням 8-бітна шина
parameter a1_width = 8;   // за замовчуванням 8-бітна шина

// Локальний параметр (не можна змінити через defparam або
#)
// Використовується тільки всередині цього модуля
localparam product_width = a0_width + a1_width;

// Оголошення портів
output [product_width -1:0] product;
input  [a0_width-1:0] a0;
input  [a1_width-1:0] a1;

// Умовна інстанціація множника
// Якщо хоча б одна шина < 8 біт → CLA-множник
// Якщо обидві ≥ 8 біт → дерево-множник
generate
if ((a0_width < 8) || (a1_width < 8))
 cla_multiplier  #(a0_width,a1_width) m0 (product,a0,a1);
else
 tree_multiplier #(a0_width,a1_width) m0 (product,a0,a1);
endgenerate
endmodule
```

В цьому прикладі використано параметри a0_width і a1_width для визначення розрядності вхідних шин. Localparam product_width задає ширину добутку, і він незмінний для зовнішніх модулів. Конструкція generate if ... else вибирає, який саме підмножник буде інстанційовано:

- якщо хоча б один із входів має ширину < 8 біт → використовується CLA multiplier (carry-lookahead);
- якщо обидва входи ≥ 8 біт → використовується tree multiplier.

Це приклад, як за допомогою умовного generate можна робити один модуль універсальним, підлаштовуючи його під різні розрядності без дублювання коду.

Generate Case

Умовний generate case дозволяє інстанціювати певні конструкції Verilog на основі виразу «вибери один із багатьох» (case). Рішення приймається під час elaboration (розгортання проєкту перед симуляцією) [4].

У generate-case можуть використовуватись:

- модулі;
- користувацькі та гейтові примітиви;
- безперервні присвоєння (assign);
- блоки initial і always.

```
// Генерація N-бітного суматора з використанням Generate
Case
// Цей модуль генерує N-бітний суматор

module adder(co, sum, a0, a1, ci);

// Оголошення параметра. Може бути перевизначено
parameter N = 4; // За замовчуванням 4-бітна шина

// Оголошення портів
output [N-1:0] sum;
output co;
input  [N-1:0] a0, a1;
input ci;

// Умовна інстанціація суматора залежно від ширини шини
// Параметр N можна змінити під час інстанціювання
generate
  case (N)
    // Спеціальні випадки для 1- та 2-бітних суматорів
    1: adder_1bit adder1(c0, sum, a0, a1, ci); // 1-бітна
реалізація
    2: adder_2bit adder2(c0, sum, a0, a1, ci); // 2-бітна
реалізація

    // Типова реалізація — N-бітний carry look ahead adder
    default: adder_cla #(N) adder3(c0, sum, a0, a1, ci);
  endcase
endgenerate

endmodule
```

В цьому прикладі параметр N визначає ширину вхідних та вихідних шин. За невеликих значень (1 або 2 біти) інстанціюються спеціалізовані модулі adder_1bit чи adder_2bit. Для будь-якого іншого значення N автоматично використовується модуль adder_cla (carry look ahead). Таким чином, один універсальний модуль adder підлаштовується під різні розрядності без дублювання коду.

Наступний приклад демонструє реалізацію N-бітного суматора з використанням блока generate case.

```
// Приклад N-бітного суматора
module adder(co, sum, a0, a1, ci);
parameter N = 4;

output [N-1:0] sum;
output co;
input [N-1:0] a0, a1;
input ci;
```

```
generate
  case (N)
    1: adder_1bit  adder1(c0, sum, a0, a1, ci);
    2: adder_2bit  adder2(c0, sum, a0, a1, ci);
    default: adder_cla #(N) adder3(c0, sum, a0, a1, ci);
  endcase
endgenerate
endmodule
```

Вибір конкретної схеми залежить від параметра N. [26]

В табл. 2.12 наведено порівняння трьох методів generate у Verilog у Verilog.

Таблиця 2.12 – Порівняльна таблиця трьох методів generate у Verilog

Метод	Синтаксис	Типові задачі	Переваги	Обмеження
Generate Loop	`generate for (genvar i=0; i<N; i=i+1) begin ... end endgenerate`	- Масове створення однакових елементів (гейти, регістри, модулі) - Побітові операції (наприклад, XOR для кожного біта) - Побудова каскадних структур (суматори, шифратори)	- Дуже компактно заміняє десятки однакових рядків коду - Гнучкість: можна створювати як модулі, так і нети/регістри	- Потрібна змінна **`genvar`** (існує тільки на етапі elaboration) - Неможна використовувати всередині generate-loop функції/задачі
Generate Conditional	`generate if (умова) ... else ... endgenerate`	- Вибір архітектури залежно від параметрів (наприклад, для малих і великих розрядностей — різні реалізації) - Умовне включення підмодуля	- Дає змогу адаптувати модуль під різні конфігурації без дублювання коду	- Умова має бути визначена **до симуляції** (на етапі elaboration)
Generate Case	`generate case (вираз) ... endcase endgenerate`	- Вибір із кількох реалізацій залежно від параметра (наприклад, 1-бітний, 2-бітний чи N-бітний суматор) - Керування підключенням складних структур	- Зручно, коли є **багато варіантів** реалізації - Легко читати й підтримувати	- Вираз у `case` також має бути відомим **на етапі elaboration**

Таким чином, на основі таблиці 2.12 можна зробити коротенькі висновки:

1. Generate Loop використовується для створення великої кількості однакових елементів чи повторюваних структур (суматори, шифратори). Це компактний і гнучкий спосіб уникнути дублювання коду, але потребує змінної genvar і не дозволяє використовувати функції чи задачі всередині циклу.

2. Generate Conditional дозволяє вибирати між двома варіантами архітектури на основі параметра. Зручно для адаптації модулів під різні конфігурації без дублювання коду. Обмеженням є те, що умова має бути відома ще до симуляції (на етапі elaboration).

3. Generate Case – найзручніший варіант, коли є багато можливих реалізацій і потрібно вибрати одну. Робить код структурованим і легким для підтримки. Проте, як і в conditional, вираз у case має бути визначений на етапі elaboration.

2.4 Таски та функції

2.4.1 Відмінності між Tasks і Functions

Під час проєктування розробнику часто доводиться багаторазово застосовувати однакову логіку в різних частинах поведінкового опису. Щоб уникнути дублювання коду, доцільно виділяти повторювані фрагменти в окремі підпрограми та викликати їх у потрібних місцях. У більшості мов програмування для цього існують процедури чи підпрограми. У Verilog поділ великих поведінкових описів на менші складові реалізується за допомогою tasks (задач) та functions (функцій). Вони дають змогу оформити часто використовуваний код у зручну форму й застосовувати його у багатьох місцях проєкту [4].

Властивості функцій:

- функція може викликати іншу функцію, але не може викликати task;
- функції завжди виконуються за 0 часу симуляції;
- функції не мають містити операторів затримки, подій чи керування часом;
- функції потрібно, щоб мали принаймні один вхідний аргумент (може бути кілька);
- функції завжди повертають одне значення і не можуть мати аргументів output чи inout.

Властивості tasks:

- tasks може викликати як інші tasks, так і functions;
- tasks можуть виконуватися за ненульовий час симуляції;
- tasks можуть містити оператори затримки, подій чи керування часом;
- tasks можуть мати нуль або більше аргументів типів input, output чи inout;
- tasks не повертають значення напряму, але можуть передавати кілька результатів через аргументи output і inout.

І tasks, і functions мають бути оголошені всередині модуля й є локальними для нього.

Tasks застосовують для спільного коду Verilog, що містить затримки, таймінг, події або кілька вихідних аргументів.

Functions використовують тоді, коли код є повністю комбінаційним, виконується за нульовий час симуляції та має рівно один вихід. Зазвичай функції застосовують для конвертацій та поширених обчислень.

Tasks можуть мати аргументи типів input, output і inout, тоді як functions – лише input. Крім того, і tasks, і functions можуть містити локальні змінні: reg, time, integer, real або event. Водночас вони не можуть мати wires [19].

Tasks і functions складаються лише з поведінкових операторів. Вони не містять блоків always чи initial, але можуть викликатися з always, initial або з інших tasks чи functions.

2.4.2 Таски

Оголошення завдань здійснюється за допомогою ключових слів task та endtask. Використовувати завдання необхідно, якщо виконується хоча б одна з таких умов:

- у процедурі присутні затримки, конструкції керування часом або подіями;
- процедура має нуль або більше одного вихідного аргументу;
- процедура не містить вхідних аргументів.

Оголошення та виклик завдання

Синтаксис для оголошення та виклику завдань виглядає таким чином:

```
task_declaration ::=
      task [ automatic ] task_identifier ;
      { task_item_declaration }
      statement
      endtask
      | task [automatic] task_identifier (task_port_list);
      { block_item_declaration }
      statement
      endtask

task_item_declaration ::=
          block_item_declaration
        | { attribute_instance } tf_input_declaration ;
        | { attribute_instance } tf_output_declaration ;
        | { attribute_instance } tf_inout_declaration ;

task_port_list ::= task_port_item { , task_port_item }
```

```
task_port_item ::=
          { attribute_instance } tf_input_declaration
        | { attribute_instance } tf_output_declaration
        | { attribute_instance } tf_inout_declaration

tf_input_declaration  ::=
input [reg] [signed] [range] list_of_port_identifiers
| input [ task_port_type ] list_of_port_identifiers

tf_output_declaration ::=
output [reg] [signed] [range] list_of_port_identifiers
| output [ task_port_type ] list_of_port_identifiers

tf_inout_declaration  ::=
inout [reg] [signed] [range] list_of_port_identifiers
| inout [ task_port_type ] list_of_port_identifiers

task_port_type ::= time | real | realtime | integer
```

Оголошення I/O аргументів у завданні відбувається за допомогою ключових слів input, output або inout, залежно від того, який тип аргументу використовується:

- input та inout передаються у завдання; вхідні аргументи обробляються у тілі завдання;
- output та inout після завершення виконання повертають значення назад у змінні, які були вказані у виклику завдання.

Завдання можуть викликати інші завдання або функції [4].

Хоча для аргументів введення/виведення в завданнях використовуються ті ж ключові слова (input, output, inout), що й при оголошенні портів у модулях, є принципова різниця:

- порти в модулях застосовуються для підключення зовнішніх сигналів;
- аргументи введення/виведення в завданні використовуються для передачі значень у завдання та отримання результатів з нього.

Приклади використання завдань (Tasks Examples)

Розглянемо два приклади завдань.

У першому показано використання вхідних та вихідних аргументів у завданні.

У другому прикладі моделюється генератор асиметричної послідовності, який формує асиметричний сигнал на тактовому імпульсі.

Використання вхідних та вихідних аргументів

Приклад демонструє, як працювати з вхідними та вихідними аргументами у завданнях [19].

Розглянемо завдання bitwise_oper, яке виконує побітові операції AND, OR та XOR для двох 16-бітних чисел. Вхідні аргументи: два 16-бітних числа a і b. Вихідні аргументи: три 16-бітних значення ab_and, ab_or, ab_xor. Також у завданні використовується параметр затримки delay.

```
// Приклад вхідні та вихідні аргументи у завданні
// Оголошуємо модуль operation, що містить завдання bitwise_oper
module operation;
...
parameter delay = 10;
reg [15:0] A, B;
reg [15:0] AB_AND, AB_OR, AB_XOR;

always @(A or B) // при зміні A або B
begin
        // Викликаємо завдання bitwise_oper.
        // Передаємо 2 вхідних аргументи: A, B
        // Очікуємо 3 вих. арг.: AB_AND, AB_OR, AB_XOR
        // Аргументи вказуються в тому ж порядку, що й
        // у визначенні завдання.
        bitwise_oper(AB_AND, AB_OR, AB_XOR, A, B);
end
...
// Оголошення завдання bitwise_oper
task bitwise_oper;
output [15:0] ab_and, ab_or, ab_xor; //вих. значення з завдання
input [15:0] a, b;     // вхідні значення для завдання
begin
        #delay ab_and = a & b;
               ab_or  = a | b;
               ab_xor = a ^ b;
end
endtask
...
endmodule
```

У цьому прикладі, у таску (завдання) передаються значення A і B. Під час входу в таску вони прив'язуються до локальних змінних a та b. Після затримки (визначеної параметром delay = 10) обчислюються вихідні значення. Коли завдання завершує роботу, значення результатів повертаються у вихідні змінні:

```
AB_AND = ab_and
AB_OR = ab_or
AB_XOR = ab_xor
```

Таким чином, таска дає можливість багаторазово виконувати побітові операції, використовуючи один і той самий опис.

Альтернативний стиль оголошення аргументів у завданнях – стиль ANSI C.

Наступний приклад демонструє, як те саме завдання bitwise_oper можна описати у скороченій формі з оголошенням аргументів у дужках при визначенні.

```
// Приклад оголошення завдання у стилі ANSI C
// Оголошення завдання bitwise_oper у стилі ANSI C
task bitwise_oper (
        output [15:0] ab_and, ab_or, ab_xor,
        input  [15:0] a, b
);
begin
        #delay ab_and = a & b;
               ab_or  = a | b;
               ab_xor = a ^ b;
end
endtask
```

У цьому варіанті аргументи описуються одразу в дужках після назви завдання. Це робить код компактнішим і більш зрозумілим, подібно до оголошення функцій у C/ANSI C стилі.

Генератор асиметричної послідовності

Завдання можуть напряму працювати зі змінними типу reg, оголошеними в модулі. Наступний приклад демонструє, як таска оперує безпосередньо зі змінною clock, формуючи асиметричну послідовність сигналу [24].

```
// Приклад роботи із змінними reg напряму
// Оголошуємо модуль з таскою asymmetric_sequence
module sequence;
...
reg clock;
...

initial
        init_sequence; // Виклик завдання ініціалізації

always begin
        asymmetric_sequence;
// Безперервний виклик завдання генерації послідовності
end
...
```

```
// Завдання ініціалізації сигналу
task init_sequence;
begin
        clock = 1'b0;
end
endtask

// Завдання для формування асиметричної послідовності
// Працює напряму зі змінною clock, оголошеною в модулі
task asymmetric_sequence;
begin
        #12 clock = 1'b0;
        #5  clock = 1'b1;
        #3  clock = 1'b0;
        #10 clock = 1'b1;
end
endtask
...
endmodule
```

В цьому прикладі, спочатку завдання init_sequence встановлює початкове значення clock = 0. Далі завдання asymmetric_sequence генерує послідовність з нерівними інтервалами між переходами сигналу:

- низький рівень тримається 12 одиниць часу,
- потім високий рівень – 5,
- знову низький – 3,
- і високий – 10.

Результат – формується асиметричний тактовий сигнал, що може бути корисним у моделюванні специфічних схем.

В табл. 2.13 наведено порівняння чим відрізняється стиль опису аргументів завдань звичайний vs ANSI C, а в табл. 2.14 показано використання reg у завданні/

Таблиця 2.13 – Порівняння стилів оголошення аргументів у завданнях

Ознака	Звичайний стиль (Classic)	Стиль ANSI C
Де описуються аргументи	Аргументи оголошуються всередині тіла завдання після ключового слова `task`.	Аргументи одразу вказуються у дужках при оголошенні завдання.
Зручність	Більш розлогий, аргументи потрібно описувати окремо.	Компактний і зрозумілий, ближчий до опису функцій у мовах C/C++.
Приклад синтаксису	`verilog task bitwise_oper; output [15:0] ab_and; input [15:0] a, b; ... endtask`	`verilog task bitwise_oper(output [15:0] ab_and, input [15:0] a, b); ... endtask`
Застосування	Використовується у старих стилях кодування або для кращої наочності за великої кількості аргументів.	Зручний у сучасному написанні коду, коли важлива компактність і читабельність.

Таблиця 2.14 – Використання reg у завданнях

Параметр	Опис
Призначення	Дозволяє завданню напряму змінювати змінні, оголошені у модулі.
Перевага	Не потрібно передавати змінні як аргументи – завдання працює безпосередньо з ними.
Недолік	Зменшується універсальність завдання, воно прив'язується до конкретних змінних модуля.
Приклад	Завдання `asymmetric_sequence` змінює змінну `clock` напряму, формуючи асиметричну послідовність.

Отже, ANSI C стиль більше підходить для опису багатофункціональних завдань, а робота напряму з reg – для специфічних випадків, де потрібна прив'язка до конкретного сигналу.

Автоматичні (Re-entrant) завдання

Звичайні таски у Verilog є статичними. Це означає, що всі змінні, оголошені всередині завдання, зберігаються у спільній області пам'яті. Якщо одне й те саме завдання викликається одночасно з різних місць коду, обидва виклики працюватимуть з одними й тими самими змінними. У такому випадку дуже ймовірно, що результат буде неправильним.

Щоб уникнути цієї проблеми, використовується ключове слово automatic. Якщо воно додається перед ключовим словом task, завдання стає re-entrant (тобто здатним до повторного виклику без конфліктів). Усі змінні всередині автоматичного завдання створюються динамічно для кожного виклику. Кожне виконання працює у власному середовищі, незалежному від інших викликів. Це дозволяє виконувати завдання паралельно без ризику пошкодження даних [4].

Рекомендується використовувати automatic tasks, якщо є ймовірність, що завдання буде викликано одночасно з кількох ділянок коду.

```
    // Приклад автоматичного (re-entrant) завдання
    // Модуль, що містить автоматичне (re-entrant) завдання
    // Показана лише частина модуля, де визначено завдання.
    // Є два тактових сигнали.
    // clk2 працює у два рази швидше за clk і синхронізований
із ним.
    module top;

    reg [15:0] cd_xor, ef_xor; // змінні у модулі top
    reg [15:0] c, d, e, f;     // змінні у модулі top

    // Автоматичне завдання для побітової операції XOR
    task automatic bitwise_xor;
        output [15:0] ab_xor;   // вихід із завдання
        input  [15:0] a, b;     // вхідні аргументи
    begin
```

```
    #delay ab_and = a & b;
           ab_or  = a | b;
           ab_xor = a ^ b;
end
endtask

// Два always-блоки викликають завдання bitwise_xor
// одночасно на різних тактових фронтах.
// Завдяки автоматичному оголошенню, виклики працюють
незалежно.
always @(posedge clk)
    bitwise_xor(ef_xor, e, f);

always @(posedge clk2) // у два рази частіше, ніж
попередній блок
    bitwise_xor(cd_xor, c, d);

endmodule
```

В цьому прикладі таска bitwise_xor визначена як automatic, тому кожен її виклик має власні локальні змінні.

У прикладі є два тактові сигнали: clk і clk2, який працює удвічі швидше.

Два блоки always викликають те саме завдання одночасно, але завдяки автоматичному типу кожен виклик працює з власними копіями змінних. У результаті – коректне паралельне виконання без перезапису даних.

2.4.3 Функції

Функції у Verilog оголошуються за допомогою ключових слів function і endfunction. Функції застосовуються лише тоді, коли для процедури виконуються такі умови:

- у процедурі немає затримок, керування часом чи подій (#delay, @, wait);
- процедура повертає тільки одне значення;
- має бути принаймні один вхідний аргумент;
- немає аргументів типу output чи inout;
- немає неблокувальних присвоєнь (<=).

Оголошення та виклик функцій

```
// Приклад синтаксису функцій
function_declaration ::=
       function [ automatic ] [ signed ] [ range_or_type ]
                 function_identifier ;
       function_item_declaration                          {
function_item_declaration }
```

```
            function_statement
            endfunction

          | function [ automatic ] [ signed ] [ range_or_type ]
                   function_identifier (function_port_list ) ;
            block_item_declaration { block_item_declaration }
            function_statement
            endfunction

     function_item_declaration ::= block_item_declaration
                                 | tf_input_declaration ;

     function_port_list    ::=    {    attribute_instance    }
tf_input_declaration
                            {,    {    attribute_instance    }
tf_input_declaration }

     range_or_type ::= range | integer | real | realtime | time
```

Коли функція оголошується, у Verilog автоматично створюється регістр із назвою function_identifier. Результат функції повертається шляхом присвоєння значення цьому регістру. Виклик функції здійснюється через її ім'я та передачу вхідних аргументів. Коли функція завершує виконання, значення регістру function_identifier автоматично підставляється у вираз, де функцію було викликано.

```
//Приклад виклику функції
y = my_function(a, b);
```

Тут результат роботи my_function буде автоматично підставлено у змінну y.

Опційно можна вказати range_or_type для визначення ширини або типу внутрішнього регістру (наприклад, integer, real, time або діапазон [7:0]). Якщо не вказати, за замовчуванням функція повертає 1-бітне значення [42].

У функції обов'язково має бути хоча б один вхідний аргумент. Функції не можуть мати output або inout аргументів, оскільки повертають результат через неявний регістр. Функції не можуть викликати завдання (tasks). Дозволяється викликати лише інші функції.

Приклади функцій (Function Examples)

Розглянемо два приклади використання функцій:

1. Обчислення парності (parity calculator) – функція повертає 1-бітне значення.

2. Регістр зсуву вліво/вправо – функція повертає 32-бітне значення після зсуву.

Обчислення парності

Розглянемо функцію, яка обчислює парність для 32-бітної адреси й повертає результат. Припустимо, що використовується парна (even) парність [26].

```
// Приклад функції calc_parity
// Модуль, що містить функцію calc_parity
module parity;
...
reg [31:0] addr;
reg parity;

// Обчислення нової парності при зміні адреси
always @(addr) begin
        parity = calc_parity(addr);
        // перше використання calc_parity - результат записується у змінну parity

        $display("Parity calculated = %b", calc_parity(addr) );
        // друге використання calc_parity - результат напряму передається у $display
end
...

// Функція обчислення парності
function calc_parity;
input [31:0] address;
begin
        // Встановлення значення виходу через неявний регістр calc_parity
        calc_parity = ^address; // XOR усіх бітів адреси
end
endfunction
...
endmodule
```

У цьому прикладі значення, яке повертає функція, може бути збережене у регістрі (parity) або одразу використане у виразі (наприклад, у $display).

```
// Приклад ANSI C стиль для calc_parity
// Функція обчислення парності у стилі ANSI C
function calc_parity (input [31:0] address);
begin
        calc_parity = ^address; // XOR усіх бітів адреси
end
endfunction
```

У цьому стилі аргументи оголошуються прямо в дужках після імені функції.

Регістр зсуву вліво/вправо

Щоб показати, як задається ширина виходу функції, розглянемо функцію, що виконує зсув 32-бітного значення на один біт вліво або вправо залежно від керувального сигналу [42].

```
// Приклад функції shift
// Модуль, що містить функцію shift
module shifter;
...

// Константи для керування напрямком зсуву
`define LEFT_SHIFT   1'b0
`define RIGHT_SHIFT  1'b1

reg [31:0] addr, left_addr, right_addr;
reg control;

// При зміні addr обчислюються зсунути значення
always @(addr) begin
        left_addr  = shift(addr, `LEFT_SHIFT);
        right_addr = shift(addr, `RIGHT_SHIFT);
end
...

// Функція shift із 32-бітним результатом
function [31:0] shift;
input [31:0] address;
input control;
begin
        // Якщо control = LEFT_SHIFT - зсув вліво, інакше
- вправо
        shift = (control == `LEFT_SHIFT) ? (address << 1)
: (address >> 1);
end
endfunction
...
endmodule
```

Таким чином, функція calc_parity демонструє повернення 1-бітного результату й те, що функції можна викликати у виразах. Функція shift показує, як вказати ширину виходу (тут [31:0]) і реалізувати умовну логіку.

На відміну від завдань (tasks), функції завжди повертають одне значення, не мають output/inout аргументів і не можуть містити затримок чи неблокувальних присвоєнь.

Автоматичні (Рекурсивні) функції

Звичайні функції у Verilog зазвичай використовуються нерекурсивно. Якщо одна і та сама функція викликається одночасно з двох місць, результати можуть бути недетермінованими, оскільки обидва виклики працюють з однією і тією самою областю пам’яті [40].

Щоб уникнути цієї проблеми, застосовується ключове слово automatic, яке робить функцію рекурсивною (або re-entrant). Для кожного рекурсивного виклику створюється власна незалежна копія змінних. Кожен виклик працює в окремому просторі пам’яті, тому рекурсія стає можливою.

Необхідно памятати, що елементи автоматичної функції не можна викликати через ієрархічні посилання, хоча саму функцію можна викликати за ієрархічним ім’ям.

```
// Рекурсивна (Automatic) функція для обчислення факторіалу
// Модуль з рекурсивною функцією factorial
module top;
...

// Визначення функції факторіала
function automatic integer factorial;
    input [31:0] oper;
    integer i;
begin
    if (oper >= 2)
        factorial = factorial(oper - 1) * oper; // рекурсивний виклик
    else
        factorial = 1;
end
endfunction

// Виклик функції
integer result;
initial begin
     result = factorial(4); // Обчисл. факторіала числа 4
     $display("Factorial of 4 is %0d", result);//Виведе:24
end
...
endmodule
```

В цьому прикладі ключове слово automatic забезпечує створення нових змінних під час кожного виклику функції, тому виклики не заважають один одному. Виклик factorial(4) запускає рекурсивні виклики factorial(3), factorial(2), factorial(1). Кожен виклик має власну копію аргументів і змінних. У результаті – factorial(4) = 4 * 3 * 2 * 1 = 24.

Отже:

- Звичайні функції не можна використовувати рекурсивно.
- Automatic-функції дозволяють рекурсію, бо кожен виклик ізольований.
- Такі функції корисні для математичних обчислень (наприклад, факторіал, рекурсивні обходи структур тощо).

Константні функції (Constant Functions)

Константна функція у Verilog – це звичайна функція, але з певними обмеженнями. Такі функції можуть використовуватися замість констант у параметрах чи виразах, коли потрібно обчислити складне значення під час елаборації (на етапі компіляції моделі). Це дозволяє, наприклад, автоматично визначати розрядність шин чи інші параметри на основі заданих значень [41].

```
// Використання константної функції
// Модель RAM
module ram (...);

parameter RAM_DEPTH = 256;

// Ширина адресної шини визначається константною функцією
clogb2
input [clogb2(RAM_DEPTH)-1:0] addr_bus;

// clogb2(256) = 8, тому шина матиме вигляд:
// input [7:0] addr_bus;

--
// Константна функція
function integer clogb2(input integer depth);
begin
   for(clogb2=0; depth > 0; clogb2 = clogb2 + 1)
       depth = depth >> 1;
end
endfunction
--

endmodule
```

У цьому прикладі функція clogb2 обчислює кількість бітів, потрібних для адресації пам'яті заданої глибини (RAM_DEPTH).

Для RAM_DEPTH = 256 результат дорівнює 8, отже addr_bus буде 8-бітним.

Signed Functions (Функції з підписаним результатом)

Signed functions дозволяють виконувати операції зі знаковими значеннями (signed) для повернeного результату функції.

Це корисно, коли потрібно працювати з від'ємними числами або арифметикою зі знаком.

```
// Приклад Signed Function
module top;
--
// Оголошення signed-функції
// Повертає 64-бітне signed значення
function signed [63:0] compute_signed(input [63:0]
vector);
    // ... реалізація функції ...
endfunction
--
// Виклик signed-функції з іншого модуля
if (compute_signed(vector) < -3) begin
    // ... дії, якщо результат менший за -3 ...
end
endmodule
```

В цьому прикладі функція compute_signed визначена як signed [63:0], тому результат можна порівнювати із від'ємними числами (наприклад, -3).

В табл. 2.15 наведено порівняння функцій та тасок (завдань).

Таблиця 2.15 – Порівняння функцій та тасок (завдань) у Verilog

Ознака / Властивість	Функція (function)	Завдання (task)
Кількість результатів	Повертає **лише одне значення** (через неявний регістр з іменем функції).	Може мати **кілька виходів** через аргументи `output` та `inout`.
Тип аргументів	Обов'язково хоча б **один `input`**. Немає `output` або `inout`.	Дозволяє `input`, `output`, `inout`.
Затримки та синхронізація	❌ **Не допускається** використання `#delay`, `@`, `wait`.	✅ Можна використовувати затримки, події, керування часом.
Присвоєння	❌ Немає неблокувальних присвоєнь (`<=`). Лише блокуючі (`=`).	✅ Можна використовувати як блокувальні (`=`), так і неблокувальні (`<=`).
Виклики	Може викликати **інші функції**, але ❌ **не може викликати завдання**.	Може викликати як **інші завдання**, так і функції.
Використання у виразах	✅ Можна безпосередньо підставляти у вирази: `y = my_func(a, b);`	❌ Завдання не можна напряму використовувати у виразах, тільки викликати: `my_task(out, in1, in2);`
Призначення	Для **простих обчислень**, які повертають одне значення (арифметика, логіка, перевірки).	Для **складних процедур**, які можуть мати кілька результатів і керувати часом.
Схожість	Подібні до функцій у C, FORTRAN.	Подібні до підпрограм / процедур у мовах програмування.

Висновки щодо функцій у Verilog

1. *Основна роль функцій*

Функції у Verilog дозволяють інкапсулювати логіку обчислень у компактному, повторно використовуваному блоці коду. Вони забезпечують більш чистий і зрозумілий опис моделей, полегшують налагодження та підвищують читабельність коду.

Ключова відмінність від завдань (tasks) полягає в тому, що функції завжди повертають одне значення і не мають вихідних аргументів.

2. *Основні обмеження функцій*

У функції завжди має бути хоча б один вхідний аргумент.

Вони не можуть мати output чи inout аргументів, оскільки результат повертається через неявний регістр із назвою функції.

Усередині функцій не можна використовувати: затримки (#delay), керування подіями (@), очікування (wait), неблокувальні присвоєння (<=).

Функції можуть викликати інші функції, але не завдання.

Таким чином, функції підходять для швидких комбінаційних обчислень, де немає часових залежностей.

3. *Різновиди функцій*

У Verilog передбачено кілька варіантів використання функцій: звичайні функції (виконуються у статичному контексті; застосовуються для простих арифметичних і логічних обчислень), automatic (рекурсивні) функції (оголошуються з ключовим словом automatic; для кожного виклику створюється власний набір змінних, що дозволяє рекурсію; використовуються для математичних задач (наприклад, факторіал, обходи структур), constant functions (використовуються на етапі компіляції моделі; дозволяють обчислювати параметри та ширину шин автоматично), signed functions (результат функції визначається як signed; дозволяють виконувати арифметику зі знаком і працювати з від'ємними числами) [24].

4. *Переваги функцій*

* Модульність і повторне використання: складні вирази можна винести у функцію і викликати багаторазово.

* Читабельність коду: скорочує дублювання, робить опис системи більш структурованим.

* Виразність: функції можна викликати у виразах (наприклад, у присвоєннях або умовах).

* Компактність: підтримка ANSI C стилю оголошення робить синтаксис більш зручним.

* Підтримка різних типів: можна вказати ширину (наприклад, [31:0]) чи тип (integer, real).

5. *Області застосування*

* Обчислення парності, контрольних сум, кодів перевірки.

* Арифметичні операції (зсув, додавання, віднімання, факторіал тощо).

* Побітові операції з векторами.

* Генерація параметрів і визначення розрядності шин (через constant functions).

* Виконання математичних перетворень зі знаком (signed functions).

6. *Порівняння з завданнями*

Якщо потрібно одне значення без часових затримок → використовується функція.

Якщо потрібно кілька результатів, управління часом чи синхронізацією → використовується завдання (task).

Таким чином, функції і завдання не замінюють одне одного, а доповнюють.

7. *Загальний висновок*

Функції у Verilog – це потужний механізм для опису комбінаційної логіки та параметризованих моделей. Вони забезпечують: високу гнучкість; можливість рекурсії та роботи зі знаком; зручність під час опису складних виразів і параметрів; за грамотного використання функції роблять код чистим, масштабованим і легко підтримуваним.

2.5 Контрольні питання до розділу 2

1. Що таке мова Verilog і для чого вона використовується?
2. У чому відмінність HDL від традиційних мов програмування?
3. Які основні області застосування Verilog?
4. Які рівні опису апаратури підтримує Verilog?
5. У чому різниця між структурним і поведінковим описом?
6. Які елементи входять до лексики Verilog?
7. Які правила запису числових констант?
8. Як задаються двійкові, десяткові, шістнадцяткові числа?
9. Що означає запис 8'b10101010?
10. Як у Verilog записуються коментарі?
11. Які обмеження щодо регістру символів у Verilog?
12. Чи є Verilog чутливим до регістру?
13. Які типи операторів підтримує Verilog?
14. Які існують арифметичні оператори?
15. Які логічні оператори використовуються?
16. У чому різниця між логічними та побітовими операціями?
17. Як працює оператор конкатенації?

18. Що таке оператор реплікації?
19. Який порядок пріоритету операторів?
20. Які типи операторів існують у Verilog?
21. Що таке оператор безперервного присвоєння?
22. Як працює оператор assign?
23. Які оператори належать до процедурних?
24. Яка різниця між = та <=?
25. Які правила створення ідентифікаторів?
26. Чи можна використовувати ключові слова як ідентифікатори?
27. Які ключові слова найчастіше використовуються?
28. Що таке escaped-ідентифікатор?
29. Які базові типи даних існують у Verilog?
30. У чому різниця між wire і reg?
31. Що таке векторні сигнали?
32. Як оголошується масив?
33. Що таке параметр (parameter)?
34. Які значення може мати логічний сигнал (0,1,X,Z)?
35. Що таке системні задачі у Verilog?
36. Для чого використовується $display?
37. Яке призначення $monitor?
38. Що таке директиви компілятора?
39. Для чого використовується define?
40. Яке призначення include?
41. Що таке модуль у Verilog?
42. Яка загальна структура модуля?
43. Що таке параметризований модуль?
44. Як створюється екземпляр модуля?
45. Що таке ієрархія модулів?
46. Які типи портів існують?
47. У чому різниця між input, output, inout?
48. Як оголошуються векторні порти?
49. Які способи підключення портів існують?
50. Що таке позиційне та іменоване підключення?
51. Що таке ієрархічне ім'я?
52. Як звернутися до сигналу у вкладеному модулі?
53. Які ризики використання ієрархічних посилань?
54. У яких випадках застосовують ієрархічні імена?
55. Що таке поведінковий опис?
56. Яка роль блока always?
57. Що таке процедурний блок?
58. Яка різниця між initial і always?
59. Як задається чутливість у always?

60. Яка різниця між блокувальним та неблокувальним присвоєнням?
61. У яких випадках застосовують <=?
62. Які помилки виникають у разі неправильного використання присвоєнь?
63. Що таке затримка в Verilog?
64. Як задається затримка часу?
65. Що таке подвійне керування?
66. Як працює оператор if?
67. Яка різниця між if та тернарним оператором ?:?
68. Які особливості вкладених умов?
69. Як працює оператор case?
70. У чому різниця між case, casex, casez?
71. Які помилки можуть виникати у разі неповного опису case?
72. Які типи циклів підтримує Verilog?
73. Як працює цикл for?
74. Яке призначення while?
75. Чим цикл у Verilog відрізняється від циклу в C?
76. У чому різниця між begin-end та fork-join?
77. Як працює паралельний блок?
78. Що таке generate-конструкція?
79. Для чого використовується genvar?
80. У чому різниця між task і function?
81. Чи може function містити затримки?
82. Як передаються параметри в task?
83. Які обмеження має function?
84. У яких випадках доцільно використовувати task?
85. У яких випадках краще застосовувати function?

3 ПРОГРАМУВАННЯ ПЛІС НА ПРАКТИЦІ

3.1 Опис та склад Altera-Cyclone-IV-board-V3.0

Altera-Cyclone-IV-board-V3.0 – це відлагоджувальна/навчальна плата на ПЛІС Intel (Altera) Cyclone IV, призначена як універсальний майданчик для створення й тестування цифрових систем на FPGA. Зовнішній вигляд плати з усіх сторін наведено на рис. 3.1 [43].

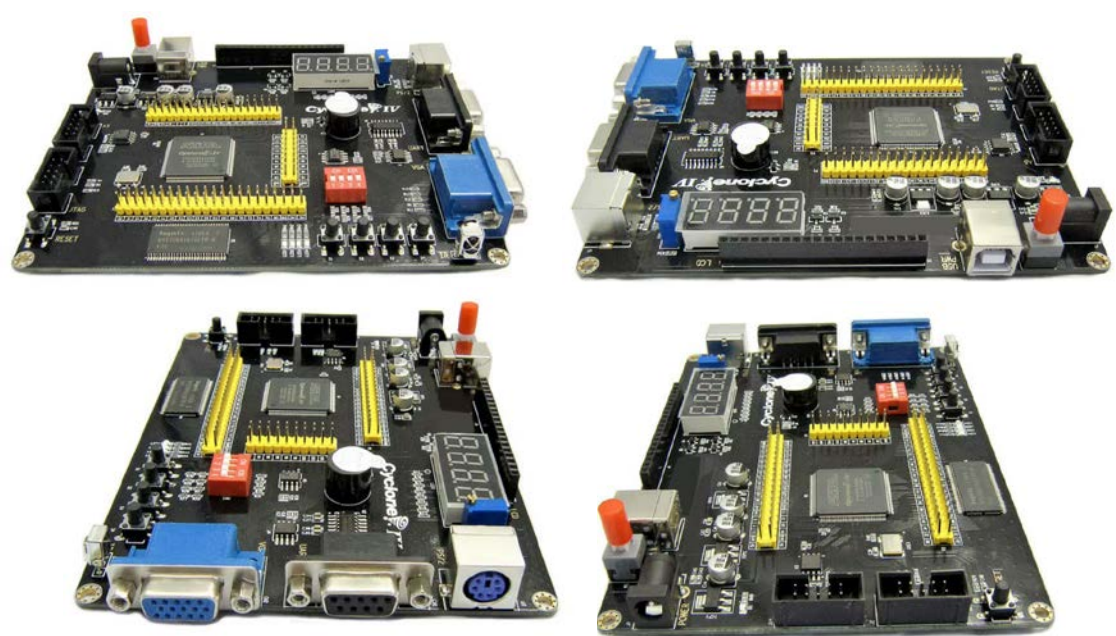


Рисунок 3.1 – Зовнішній вигляд плати Altera-Cyclone-IV-board-V3.0

Основні сценарії використання плати Altera-Cyclone-IV-board-V3.0:

- освоєння Verilog/VHDL: від елементарної логіки та автоматів станів до конвеєрів, шин і контролерів пам'яті;
- швидке прототипування цифрових пристроїв: лічильники, PWM, інтерфейси UART/SPI/I^2C, базова обробка сигналів, прості аудіо/графічні проєкти;
- Soft-CPU: побудова процесора типу Nios II і запуск прикладних програм;
- робота з периферією через доступні GPIO/PMOD та стандартні елементи вводу-виводу (світлодіоди, кнопки, індикатори – конкретний набір залежить від ревізії).

Головним елементом плати є ПЛІС Altera EP4CE6 (рис. 3.2). Вона має підведені типові інтерфейси – LCD, VGA, UART та інші для підключення периферії.

Сама мікросхема виконана у корпусі QFP: на відміну від BGA, такий формат простіше паяти, тестувати й використовувати під час розробки та налагодження. Усі контакти основної мікросхеми виведено на роз'єми з

кроком 2,54 мм, що спрощує підключення модулів і дає широкі можливості для розширення.

Плату оснащено прозорим екранувальним кожухом, який захищає чип і мінімізує ризик ESD-пошкоджень. Також у комплект входить USB-програматор для прошивки FPGA.

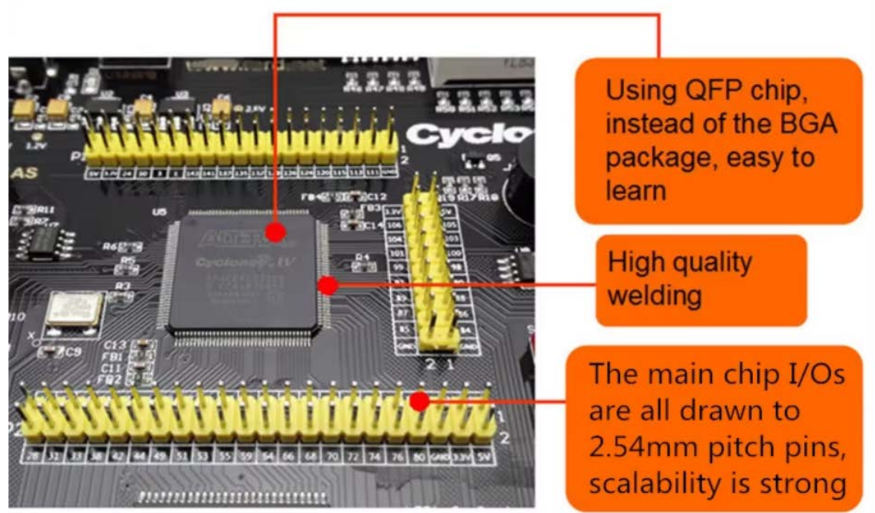


Рисунок 3.2 – Головний елемент плати ПЛІС Altera EP4CE6

Параметри плати Altera-Cyclone-IV-board-V3.0:

1. FPGA: серцем плати є ALTERA Cyclone IV EP4CE6E22C8N.
2. Пам’ять і конфігурація:
   - встановлено 16-Мбіт послідовний конфігураційний чип EPCS16N;
   - прошивання та налагодження можливі через JTAG або AS.
   - наявна 64 Мбіт SDRAM, що підходить для проєктів SOPC і процесора Nios II.
3. Живлення та стабілізатори:
   - лінія 3,3 В на мікросхемі регулятора напруги 1117-3.3V;
   - напруга ядра 1,2 В на мікросхемі регулятора напруги 1117-1.2V;
   - Лінія 2,5 В для PLL на мікросхемі регулятора напруги 1117-2.5V.
4. Вхідне живлення: DC 5 В; можливе живлення також через USB.
5. Інтерфейси прошивки/налагодження:
   - JTAG: завантаження файлів .SOF, висока швидкість;
   - AS: завантаження файлів .POF, збереження конфігурації після вимкнення живлення.
6. Розширення: усі виводи FPGA виведено на роз’єми з кроком 2,54 мм для зручного підключення модулів.
7. Габарити плати: 136 × 106 мм.

Примітка: у типовій практиці конфігурація .SOF через JTAG є тимчасовою (втрачається після вимкнення живлення), тоді як .POF через AS – постійна.

Altera плата дозволяє загружати файл конфігурації в пристрій двома шляхами: з допомогою JTAG режиму і з допомогою AS режиму.

Плату можна конфігурувати двома способами: через JTAG або через AS (Active Serial) [43].

Конфігураційні дані передаються з комп’ютера, на якому запущено Quartus II, по USB-кабелю: його підключають до ПК і до лівого USB-порту плати. Для коректної роботи встановіть драйвер USB-Blaster. Перед початком переконайтеся, що кабель під’єднано правильно і плата живиться.

JTAG-режим: конфігурація записується безпосередньо у мікросхему FPGA. JTAG (Joint Test Action Group) – це стандарт IEEE для тестування й програмування цифрових пристроїв. У цьому режимі прошивка зберігається лише доти, доки подано живлення; після вимкнення вона втрачається.

AS-режим: використовується окремий конфігураційний пристрій із флеш-пам’яттю, де зберігаються дані конфігурації. Quartus II спершу записує ці дані у конфігураційний чип на платі, а під час вмикання живлення або ініціації реконфігурації вони автоматично завантажуються в FPGA.

Периферійні інтерфейси та ресурси плати Altera-Cyclone-IV-board-V3.0 зображено на рис. 3.3.

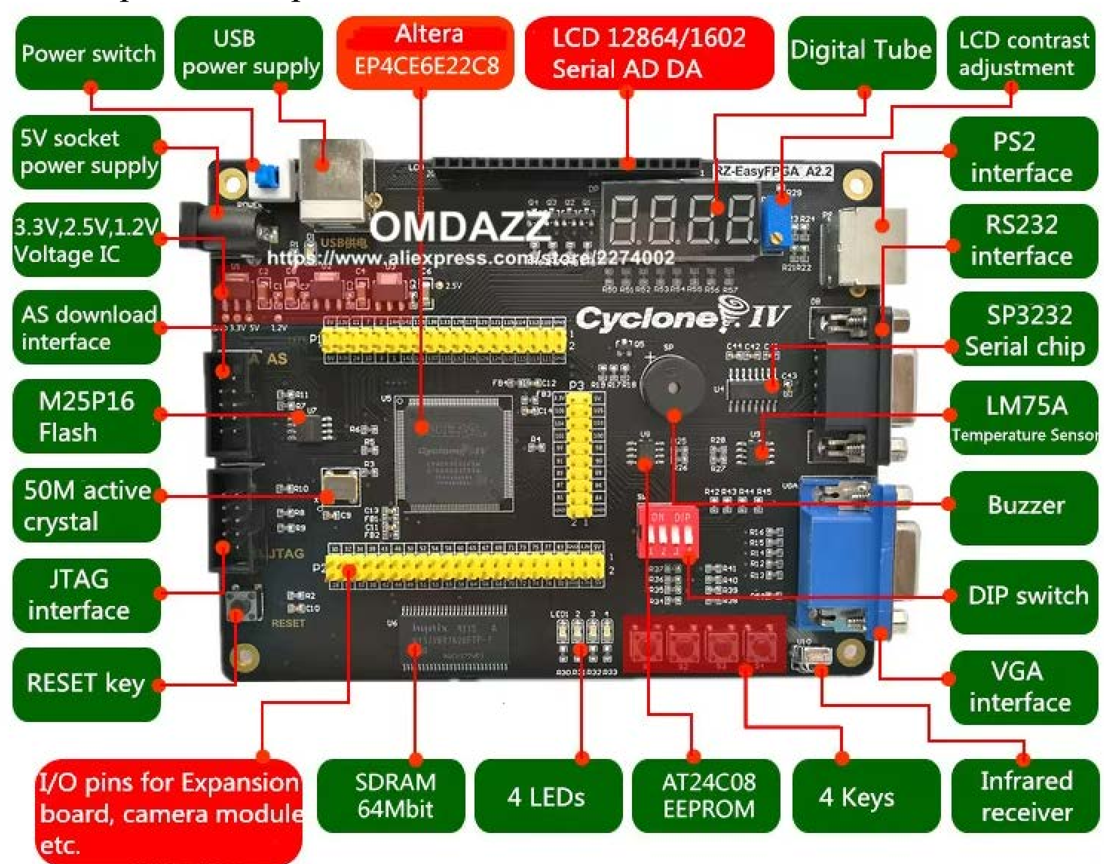


Рисунок 3.3 – Головний елемент плати ПЛІС Altera EP4CE6

Відповідно до рис. 3.3, периферійними інтерфейсами та ресурсами плати Altera-Cyclone-IV-board-V3.0 є:

1. 1 вимикач живлення з кнопкою самоблокування, 1 клавіша скидання, 4 клавіші користувача;
2. 4 світлодіодних діоди;
3. 4-розрядний семисегментний індикатор;
4. 4-розрядний DIP-перемикач;
5. 1 мінігучномовець (пищалка) ;
6. Інтерфейс PS2;
7. Послідовний порт RS232;
8. 1*20-контактний роз'єм для РК-дисплеїв, сумісний з LCD1602, LCD12864, TFT LCD;
9. Точне регулювання опору, регульоване підсвічування РК-дисплея;
10. Мікросхема датчика температури LM754A;
11. 8-кольоровий інтерфейс VGA;
12. EEPROMAT24C08 Послідовний порт I2C для експериментів з шиною IIC;
13. Модуль інфрачервоного приймача;
14. Послідовний порт RS232;
15. Модуль для підключення до всіх виводів основної мікросхеми, з відстанню між ними 2,54 мм.

Додатковими елементами є:

- з'єднувальна пластина (рис. 3.4);
- модуль камери OV7670 (рис. 3.5);
- VGA-модуль зі слотом для SD-карти (рис. 3.6), який містить 16-бітний True Color, підтримує 65536 кольорів;
- 2,4-дюймовий TFT-модуль з сенсорним інтерфейсом та зі слотом для SD-карти (рис. 3.7);
- модуль Ethernet (рис. 3.8);
- USB-модуль CH376S для зв'язку між USB та ПК, читання та запису USB-флеш-накопичувачів, читання та запису SD-карт (рис. 3.9).

Рисунок 3.4 – З'єднувальна пластина

Рисунок 3.5 – Модуль камери OV7670

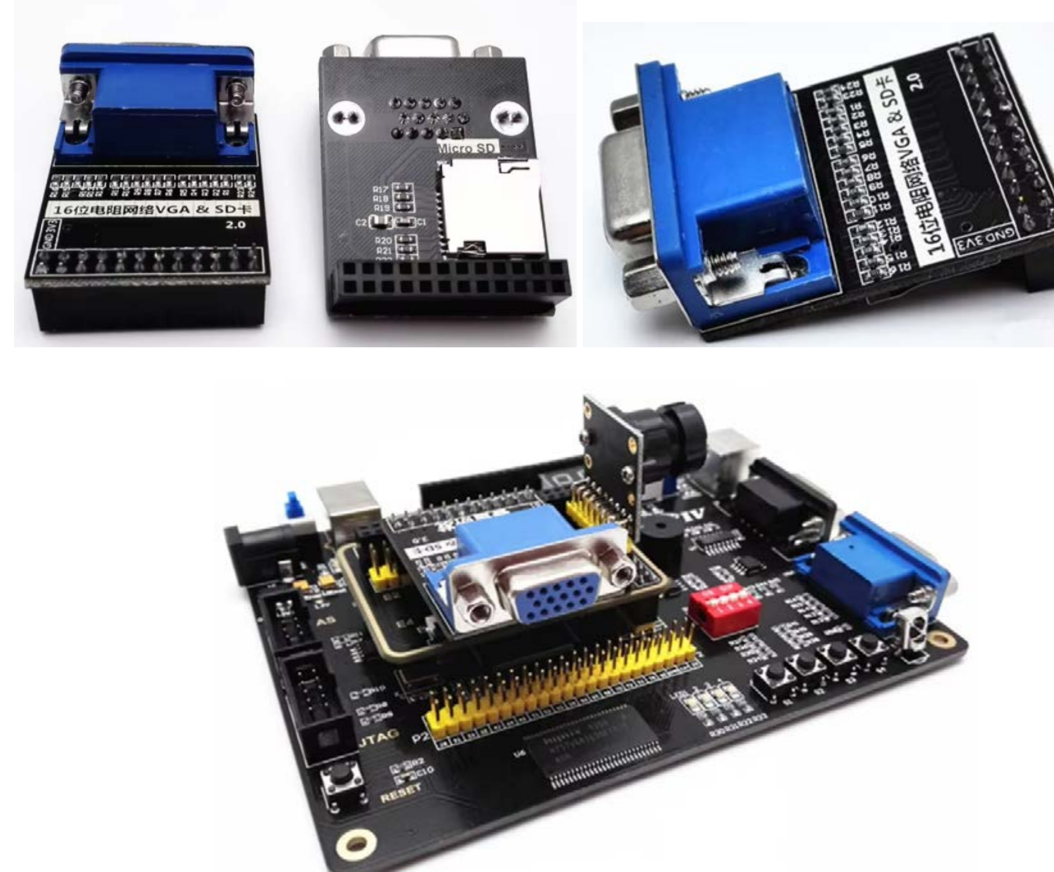


Рисунок 3.6 – VGA-модуль зі слотом для SD-карти
та приклад його підключення

Рисунок 3.7 – 2,4-дюймовий TFT-модуль з сенсорним інтерфейсом та зі слотом для SD-карти

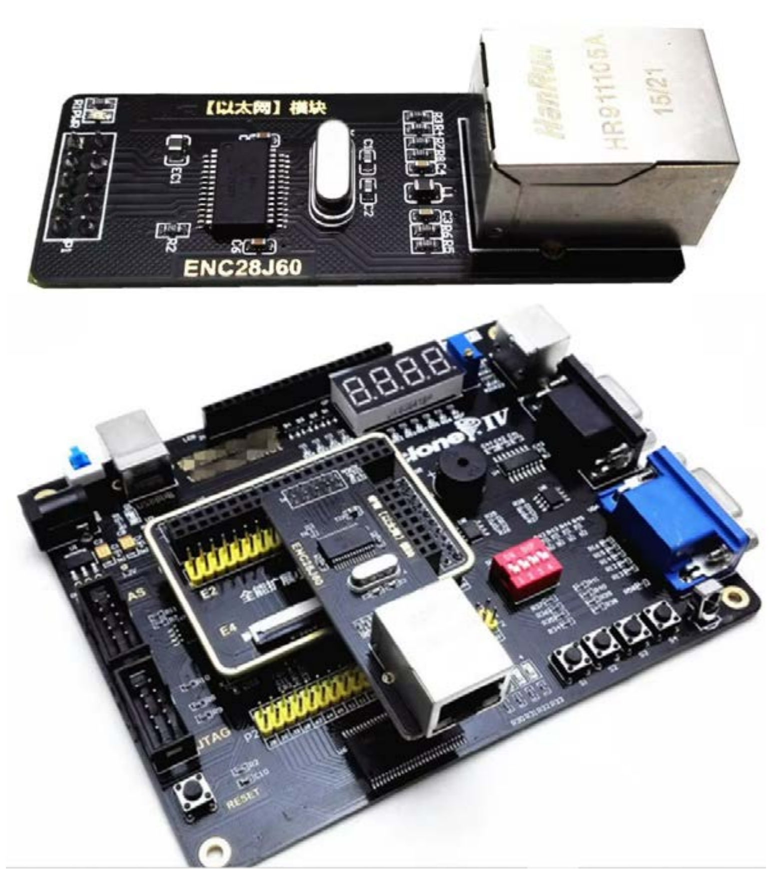


Рисунок 3.8 – Модуль Ethernet та приклад його підключення

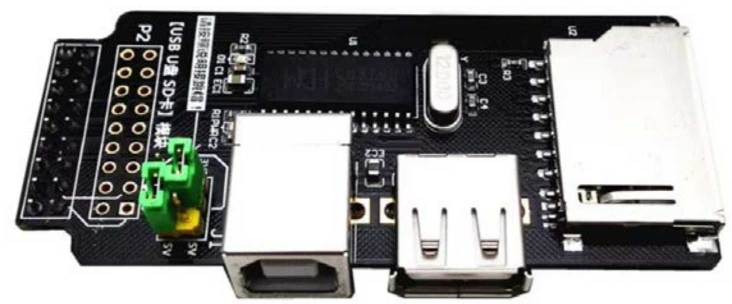


Рисунок 3.9 – USB-модуль CH376S

Таким чином, можна відзначити переваги використання плати Altera-Cyclone-IV-board-V3.0:

1. Гарна база для навчання та прототипування. Cyclone IV підтримується безкоштовним Quartus Lite, багато прикладів/гайдів.

2. Зручний корпус QFP. Легше оглядати, щупати осцилографом і, за потреби, ремонтувати, ніж BGA.

3. Повне виведення сигналів на 2,54 мм хедери. Просто під’єднувати модулі, логічні аналізатори, макетні рішення.

4. Дві опції прошивки. Швидке тимчасове завантаження через JTAG (.SOF) і постійна конфігурація через AS/EPCS (.POF).

5. Наявна зовнішня SDRAM (≈64 Мбіт). Достатньо для курсів із SOPC/NIOS II, буферів, простих графічних/аудіозавдань.

6. Базова периферія на борту. LCD, VGA, UART – можна демонструвати результати «з коробки».

7. Просте живлення. 5 В або від USB, LDO 1117 дають типові лінії 3,3 В / 2,5 В / 1,2 В.

8. ESD-захист. Прозора захисна пластина зменшує ризик статичних ушкоджень.

Недоліки плати Altera-Cyclone-IV-board-V3.0:

1. Обмежені ресурси FPGA. EP4CE6 – невелика за місткістю ПЛІС; для великих DSP/ML-проєктів чи широких відеоконвеєрів може не вистачати.

2. Без високошвидкісних інтерфейсів. У версії E немає гігабітних SerDes/PCIe; під HDMI/камера-лінки потрібні зовнішні рішення.

3. SDRAM скромна за обсягом і швидкістю. Підійде для Nios II й буферів, але не для «важкого» відео/обробки потоків.

4. LDO-стабілізатори 1117 – невисока ефективність. За великих струмів гріються та марнують енергію; від USB доступний струм обмежений.

5. Хедери 2,54 мм – не для високих частот. Довгі дроти/макетні з’єднання погіршують цілісність сигналу на десятках-сотнях мегагерц.

6. Архітектура не найновіша. Менше сучасних «плюшок» (наприклад, апаратні блоки під нові протоколи) порівняно з новішими сім’ями.

Якщо назріває потреба у швидкісних інтерфейсах чи більших ресурсах – необхідно вибирати плати з новішими FPGA (або Cyclone IV GX/далі) та з імпульсними перетворювачами замість LDO.

3.2 Встановлення драйвера USB-Bluster

Для роботи з макетом, необхідно встановити драйвер USB-Bluster. Спочатку під’єднайте плату до живлення – через блок живлення або USB-кабель. Далі з’єднайте комп’ютер (USB-порт) із лівим USB-роз’ємом на платі тим самим кабелем і увімкніть пристрій червоною кнопкою. Операційна система визначить новий USB-пристрій, але працювати з ним можна лише після встановлення потрібного драйвера. Плата DE2 прошивається через інтерфейс Altera USB-Blaster; якщо драйвер USB-Blaster ще не інстальовано, з’явиться вікно майстра додавання нового обладнання (New Hardware Wizard), як показано на рис. 3.10 [38].

Рисунок 3.10 – Вікно Found New Hardware Wizard

В вікні виберіть **No, not this time** і натисніть **Next**. В вікні, що з’явилось, зображеному на рис. 3.11, виберіть **Install from a specific location** і натисніть **Next.**

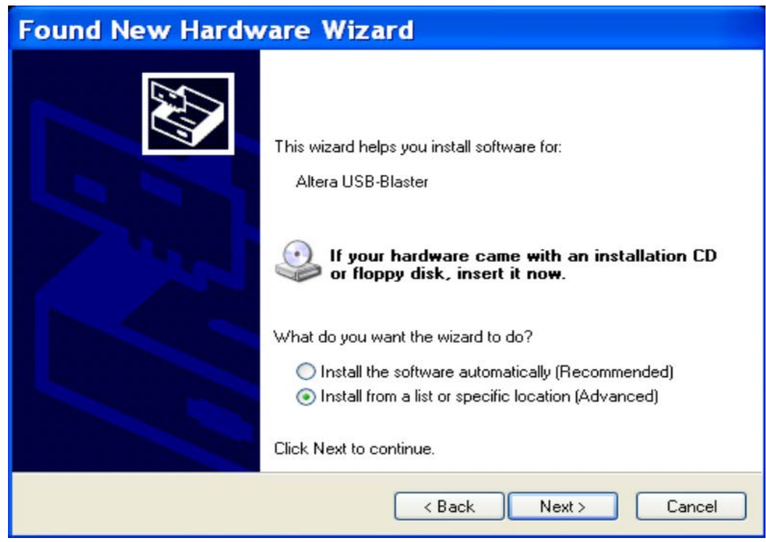


Рисунок 3.11 – Драйвер найдено в визначеному місці.

В вікні, що з'явилося, зображеному на рис. 3.12, виберіть Search fro the best driver in these locations і натисніть Browse для того, щоб добратися до вікна, зображеного на рис. 3.13.

Рисунок 3.12 - Визначення місцезнаходження драйвера

Рисунок 3.13 - Вибір місцезнаходження в дереві каталогів

Найдіть необхідний драйвер, який знаходиться за шляхом *altera\quartus72\drivers\usb-blaster*. Натисніть **OK**, далі **Next**. Після цього з'явиться вікно, зображене на рис. 3.14.

Рисунок 3.14 – Нема необхідності тестувати драйвер

Натисніть **Continue Anyway**.

Тепер драйвер установлено, як показано на рис. 3.15. Натисніть **Finish** і можете починати роботу з платою.

Рисунок 3.15 – Драйвер установлено

Встановлення драйвера USB-Blaster потрібне, щоб ПК «умів» спілкуватися з кабелем-програматором і самою платою по JTAG/AS. Без драйвера Quartus просто не бачить обладнання.

Що це дає на практиці?

- Прошивка FPGA (.sof) через JTAG (тимчасове завантаження в ОЗП ПЛІС).
- Програмування конфігураційної флеш (.pof) через AS (постійна конфігурація EPCS/Max10 тощо).
- Виявлення кола JTAG (ID пристроїв, Boundary-Scan).
- Налагодження і сервісні інструменти: SignalTap II (вбудований логічний аналізатор), In-System Memory/Source & Probe, System Console.
- Зв’язок із Nios II: завантаження ELF, налагодження, JTAG-UART.

Як зрозуміти, що все ок?

- У Quartus Programmer в списку Hardware з’являється USB-Blaster [USB-x].
- Кнопка Auto Detect бачить ваше FPGA у колі.

Якщо драйвер не встановлено, пристрій визначається як «Unknown device»/непізнаний та у Quartus список Hardware порожній, прошивка неможлива.

Таким чином, драйвер – це міст між Quartus і платою. Без нього ні прошивання, ні налагодження за JTAG/AS не працюватимуть.

3.3 Приклад створення логічного елемента 4І

Для створення логічного елемента І на ПЛІС використаємо ПЗ Quartus 13. Спочатку нам потрібно створити проєкт як показано на рис. 3.16 та рухатися по процесу, дотримуючись інструкцій рис. 3.17.

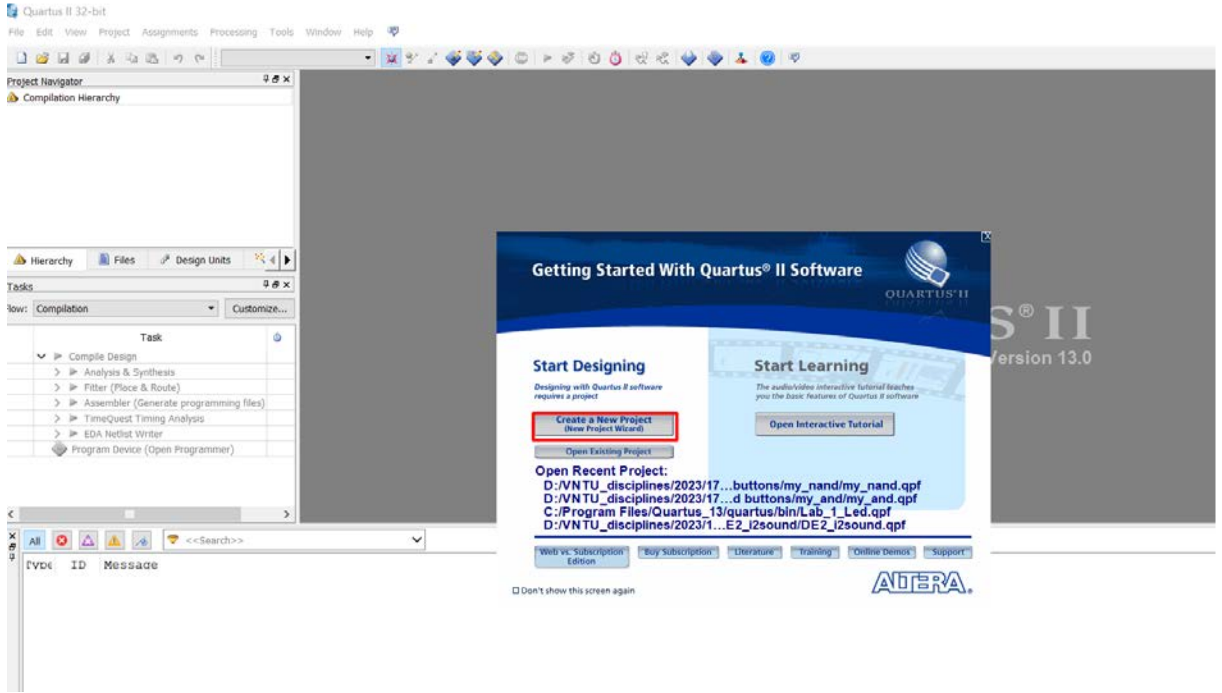


Рисунок 3.16 – Створення проекту

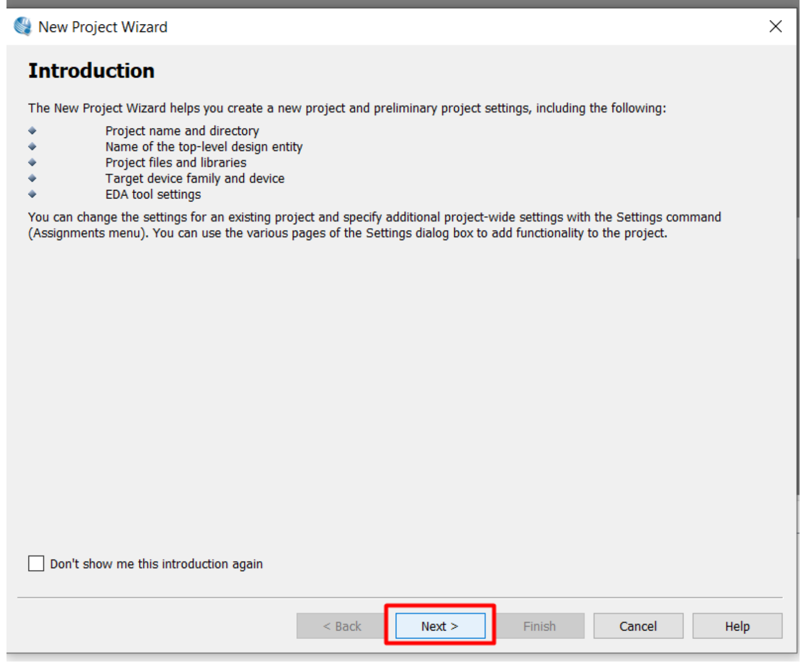


Рисунок 3.17 – Інструкції створення проєкту

Наступним кроком є задання шляху до файлів проєкту та назви проєкту як показано на рис. 3.18.

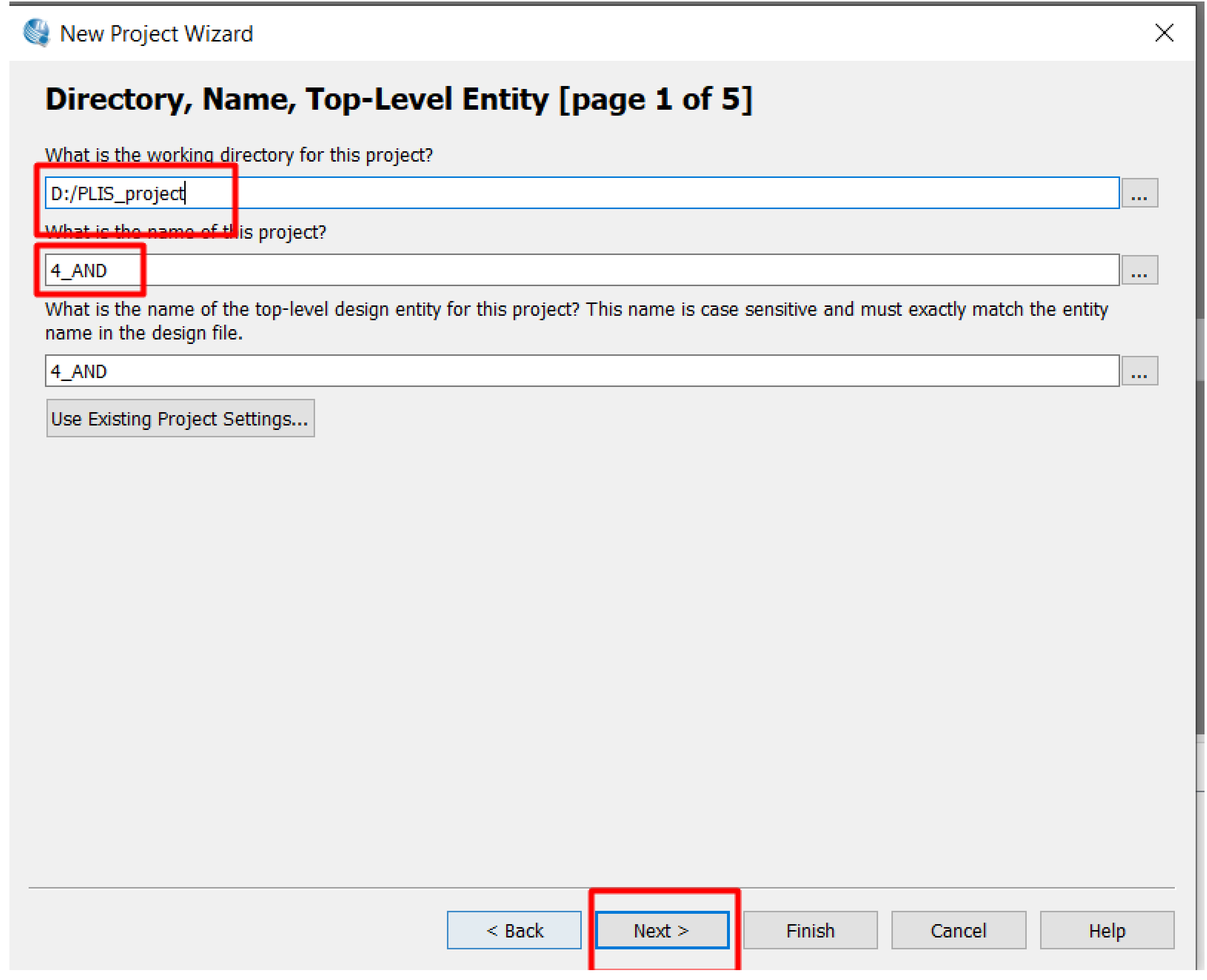


Рисунок 3.18 – Задання шляху до проєкту та його назви

Далі необхідно добавити до проєкту новий файл як показано на рис. 3.19 та вибрати його тип (рис. 3.20).

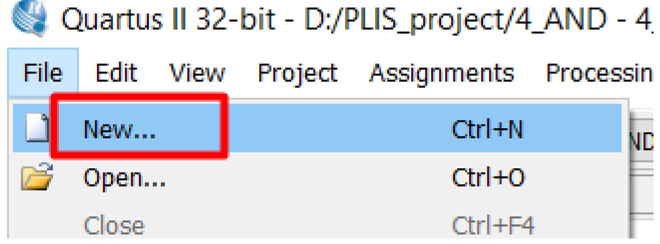


Рисунок 3.19 – Додавання файлу до проєкту

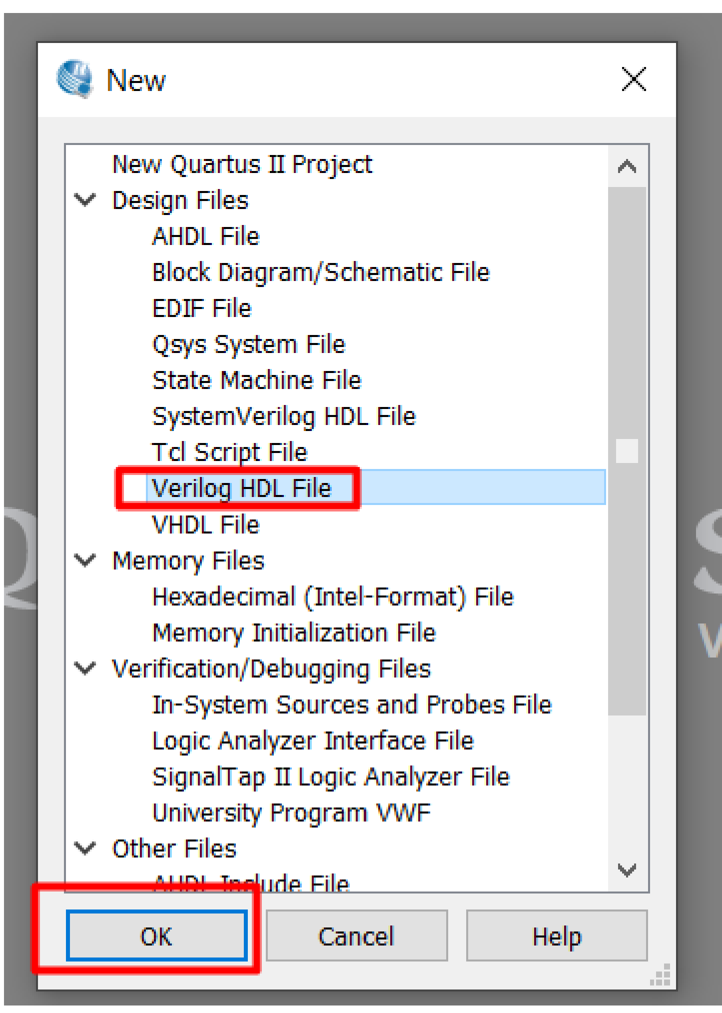


Рисунок 3.20 – Вибір типу файлу

Наступним кроком є додавання Verilog-коду для реалізації логічного елемента 4І як показано на рис. 3.21.

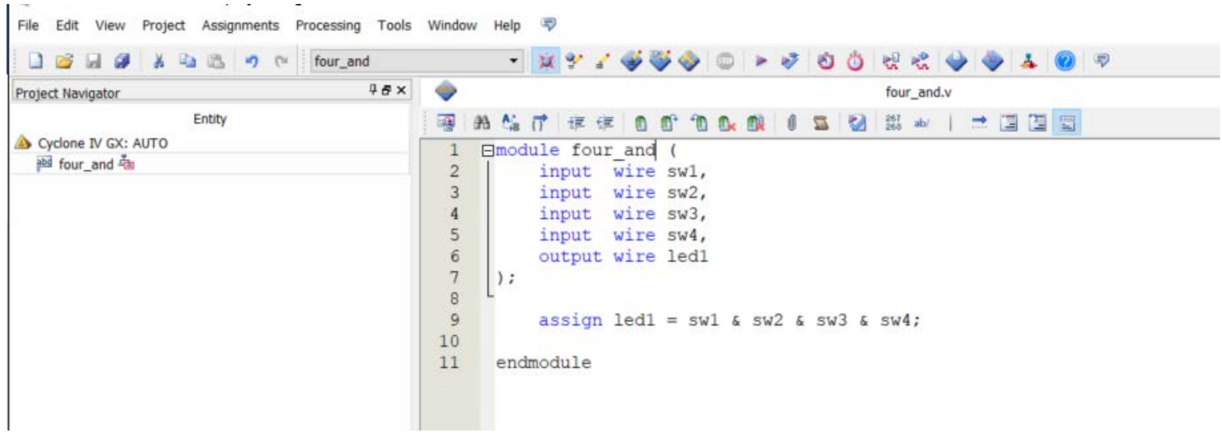


Рисунок 3.21 – Додавання Verilog коду

Наступним кроком є компіляція для перевірки на наявність помилок у коді як показано на рис. 3.22.

Рисунок 3.22 – Компіляція коду

Далі необхідно вибрати конкретну модель ПЛІС мікросхеми, на якій реалізовуватимемо наш пристрій. Для цього необхідно натиснути правою кнопкою мишки на на проєкті і вибрати пункт налаштування – як показано на рис. 3.23, потім вибрати необхідну ПЛІС як показано на рис. 3.24.

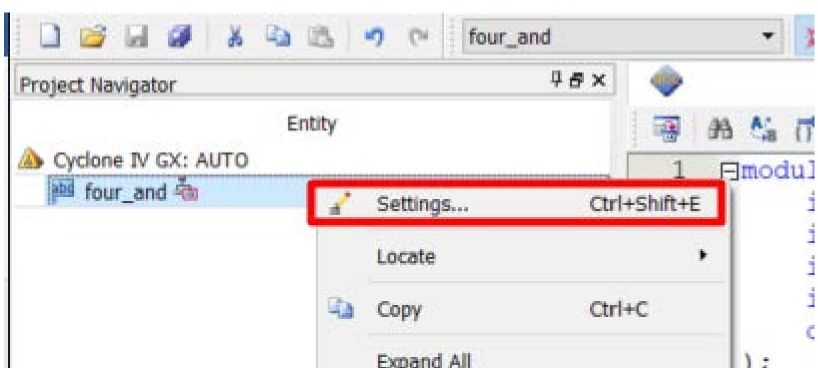


Рисунок 3.23 – Перехід в налаштування для вибору ПЛІС мікросхеми

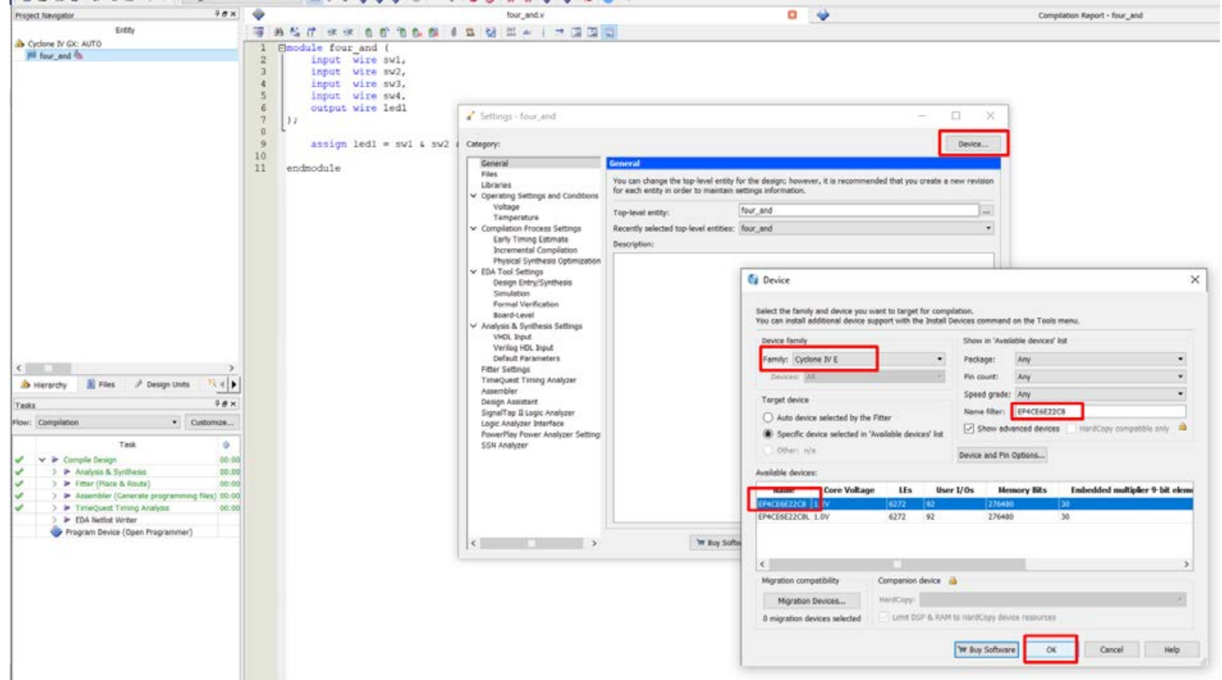


Рисунок 3.24 – Вибір ПЛІС мікросхеми

Далі необхідно призначити необхідні входи та виходи для нашого логічного елемента 4І. Для цього необхідно зайти в assignments->Pin Planner як показано на рис. 3.25. І потім потрібно задати відповідно до документації на цю ПЛІС мікросхему необхідні піни як показано на рис. 3.26.

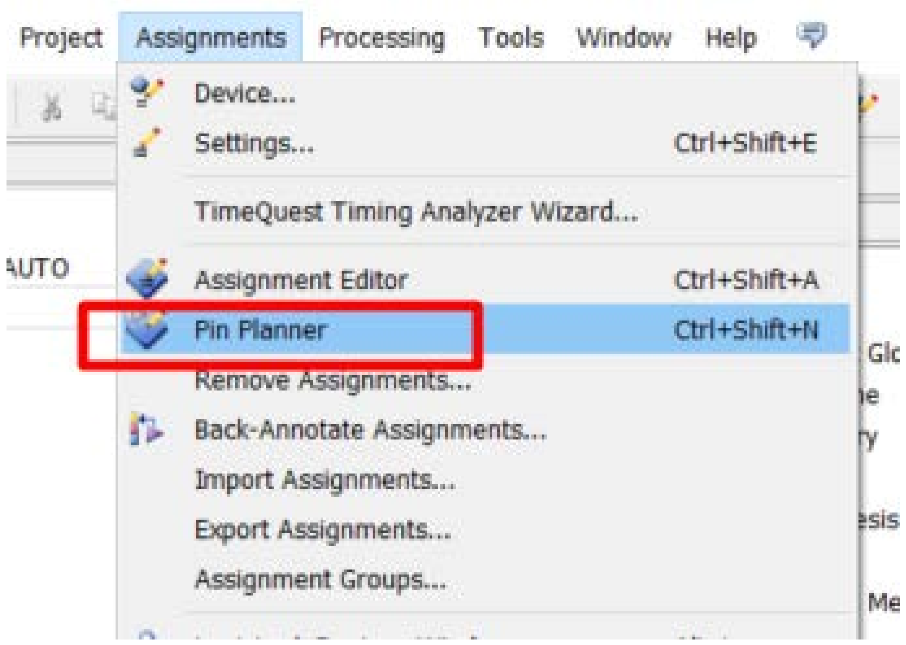


Рисунок 3.25 – Перехід в Pin Planner

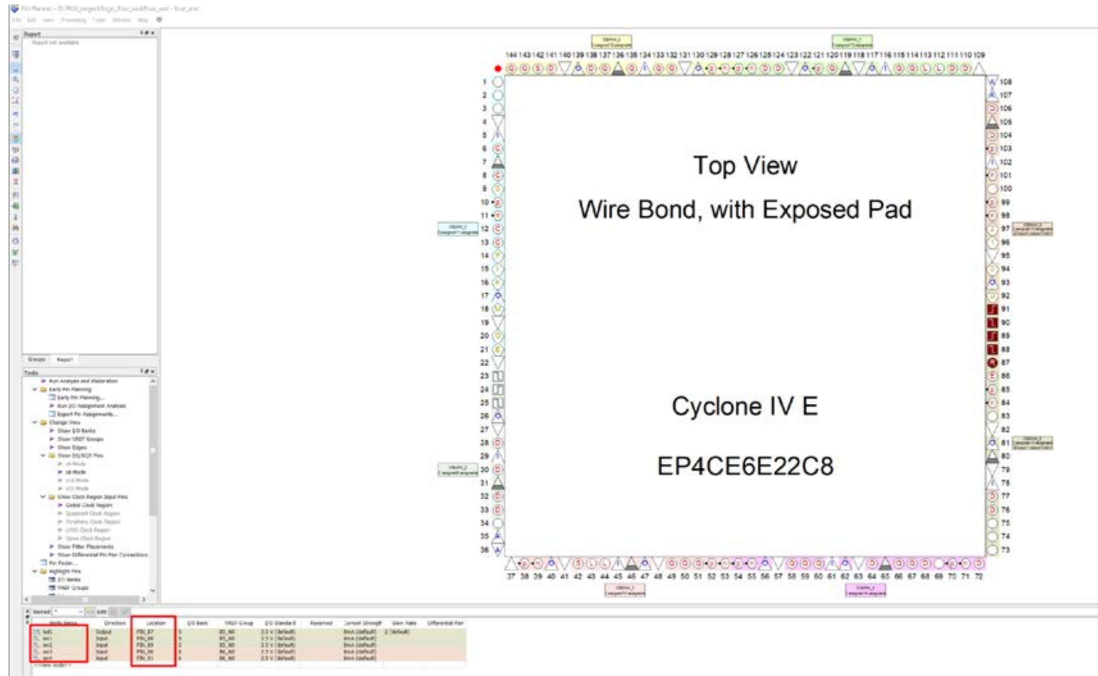


Рисунок 3.26 – Задання пінів входу та виходу

Наступний крок – це перехід до програмування (прошивки) нашої ПЛІС мікросхеми. Для цього необхідно вибрати у лівому вікні пункт, пов'язаний з програмуванням як показано на рис. 3.27 та вибрати два параметри програматора.

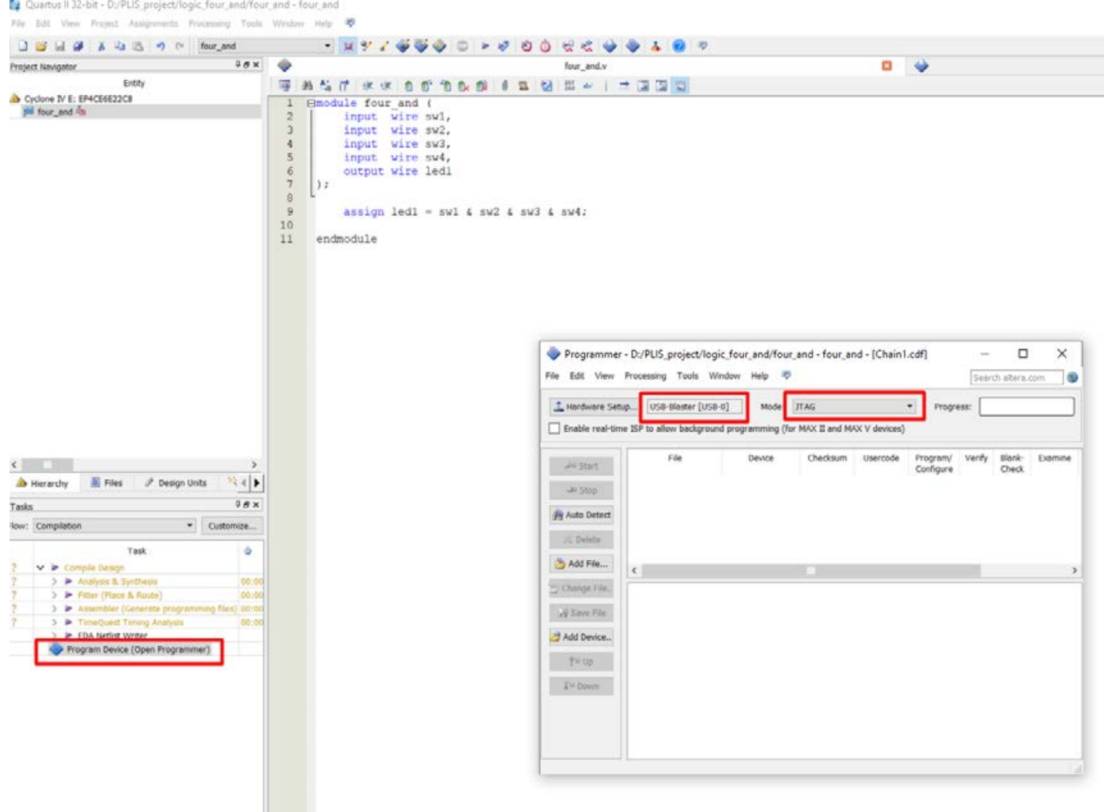


Рисунок 3.27 – Програмування ПЛІС

Далі необхідно вибрати наш файл з Verilog кодом та виставити необхідні налаштування як показано на рис. 3.28 і потім натиснути кнопку Start.

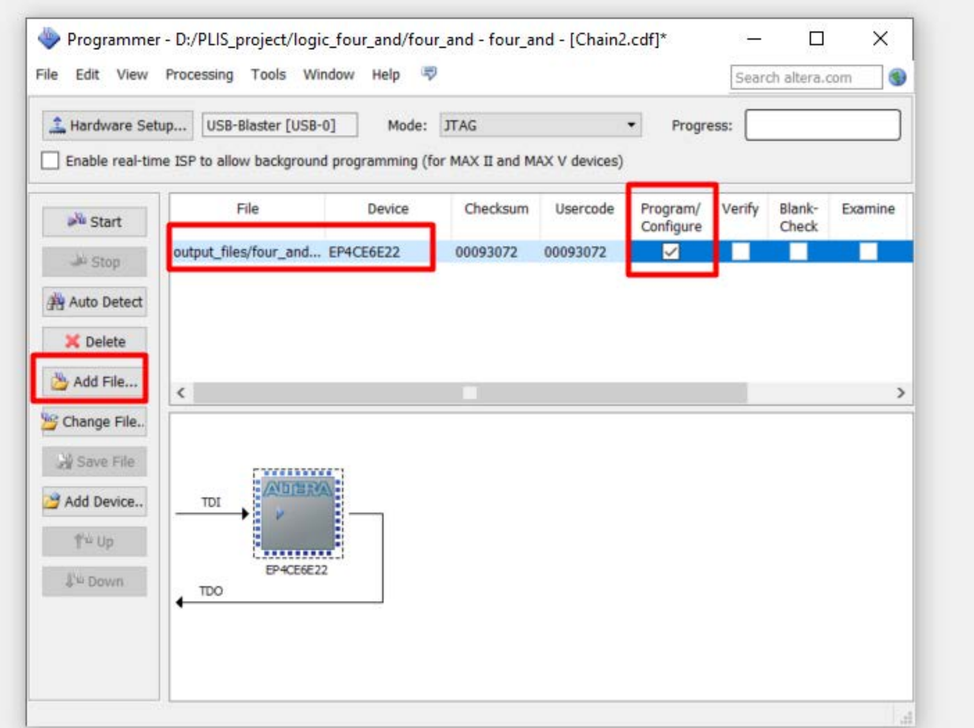


Рисунок 3.28 – Додавання файлу програми

Далі необхідно дочекатись успішного запису прошивки в мікросхему як показано на рис. 3.29.

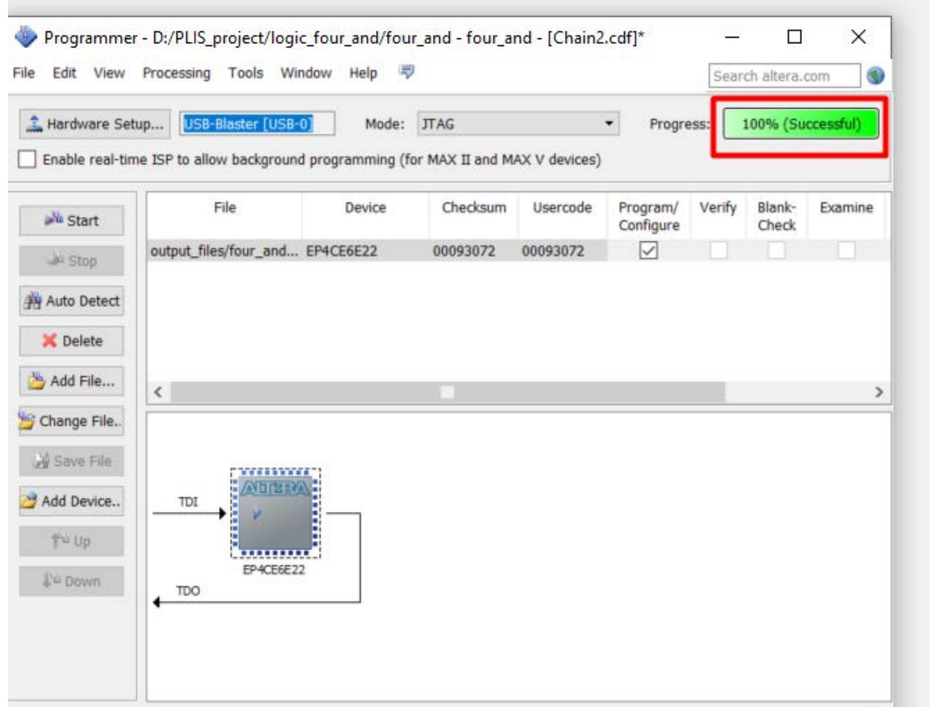


Рисунок 3.29 – Результат програмування ПЛІС мікросхеми

На цьому процес розробки, програмування та прошивки завершено. Переходимо до тестування нашого функціоналу. Як показано на рис. 3.30, перемикачі в нижньому положенні – це логічна 1, у верхньому – це 0. Світлодіод потухне лише в тому випадку коли всі перемикачі будуть встановлені в нижнє положення. На рис. 3.30 та рис. 3.31 є різні комбінації і вихідний діод горить, тобто елемент І дає нуль (еквівалент, що діод горить).

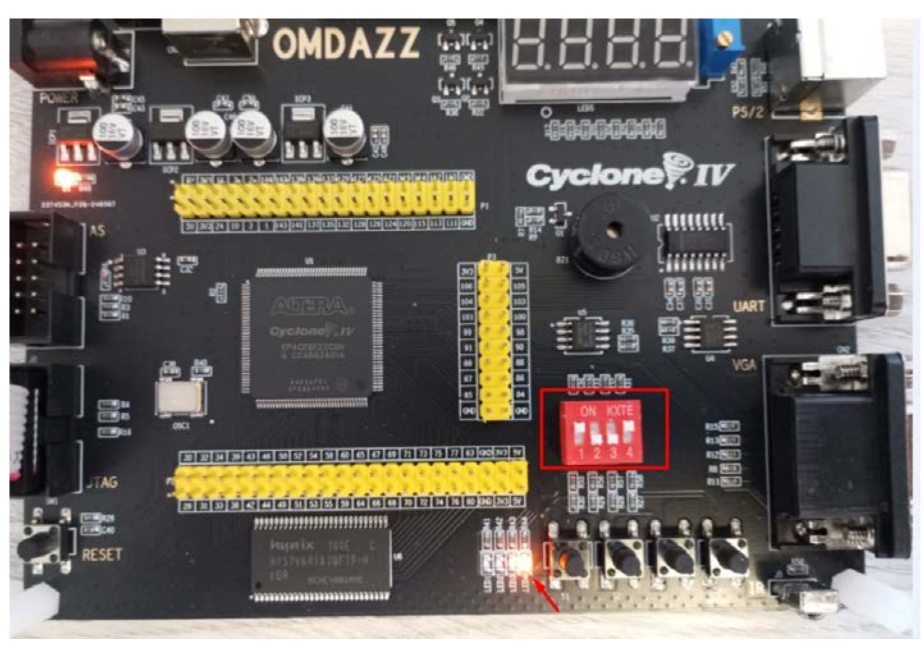


Рисунок 3.30 – Комбінація 1 і 4 входи нуль, 2 і 3 – логічна одиниця

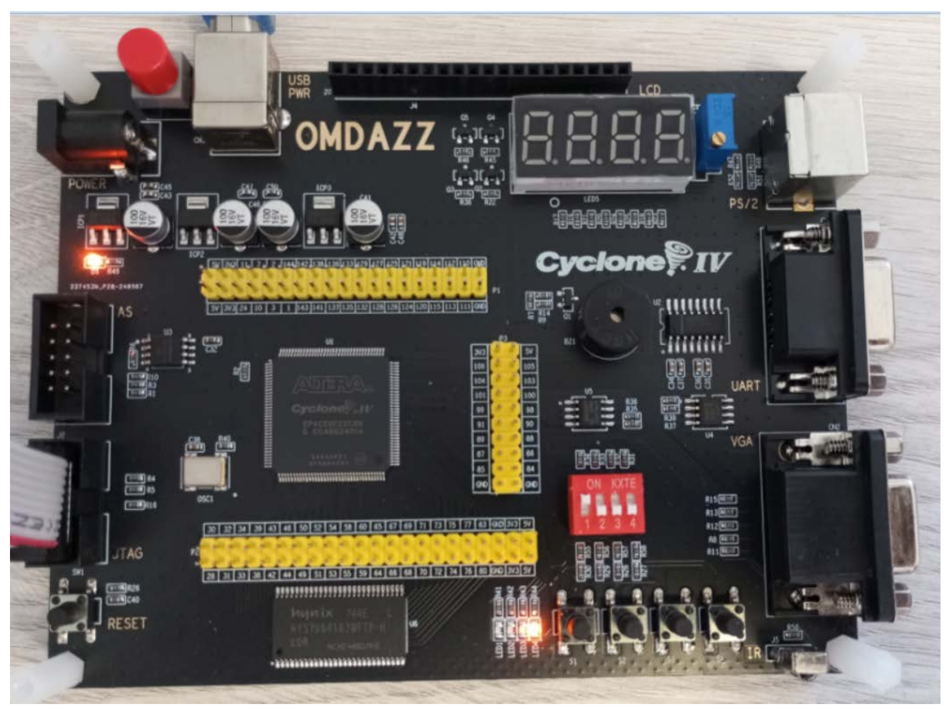


Рисунок 3.31 – Комбінація 1 вхід нуль, 2-4 – логічна одиниця

На рис. 3.32 показано випадок, коли всі входи мають логічну одиницю і відповідно діод на виході не світиться.

Рисунок 3.32 – Комбінація 1-4 – логічна одиниця

Таким чином, було здійснено проєктування логічного елемента 4І (чотиривходового елемента AND) із використанням програмного середовища Quartus II версії 13 та мови опису апаратури Verilog.

Під час роботи було створено опис логічного елемента мовою Verilog, який реалізує логічну операцію кон’юнкції для чотирьох вхідних сигналів. Також реалізовано логічний елемент на апаратному рівні за допомогою навчальної плати з ПЛІС Cyclone IV від Intel (раніше Altera). Як вхідні сигнали було використано перемикачі DIP-switch, розташовані на платі, а результат роботи логічного елемента відображався за допомогою світлодіода LED1.

Під час практичної перевірки було підтверджено коректність роботи схеми: світлодіод вимикався лише у випадку, коли всі чотири перемикачі перебували у стані логічної одиниці. Це підтвердило відповідність апаратної реалізації результатам моделювання.

ПЕРЕЛІК РЕКОМЕНДОВАНИХ ДЖЕРЕЛ

1. Каплун В. М., Кондратенко Ю. П. Цифрова схемотехніка та програмовані логічні інтегральні схеми : навч. посіб. Київ : КПІ ім. Ігоря Сікорського, 2021. 320 с.
2. Мельник О. М., Довгий С. О. Проєктування цифрових систем на основі FPGA : навч. посіб. Львів : Видавництво Львівської політехніки, 2022. 280 с.
3. Бондаренко О. В., Савченко І. П. Сучасні програмовані логічні інтегральні схеми : навч. посіб. Харків : ХНУРЕ, 2020. 250 с.
4. Головко В. А., Петренко О. В. Мови опису апаратури Verilog та VHDL у проєктуванні цифрових систем : навч. посіб. Київ : НТУУ «КПІ», 2023. 300 с.
5. Сидоренко В. Г., Пилипенко О. В. Проєктування цифрових пристроїв на ПЛІС. Київ : НАУ, 2021. 260 с.
6. Шевченко В. П., Романюк О. В. Основи цифрової електроніки та логічного проєктування. Київ : Ліра-К, 2022. 340 с.
7. Дудник В. М., Кравченко С. О. Програмовані логічні інтегральні схеми в цифрових системах. Дніпро : НМетАУ, 2020. 210 с.
8. Іванов І. О., Паламарчук В. М. Архітектура та застосування FPGA у сучасних електронних системах // *Вісник НТУУ «КПІ». Серія Радіотехніка.* 2021. № 86. С. 45–52.
9. Кузьменко О. О., Бойко А. В. Дослідження можливостей використання FPGA у вбудованих системах // *Сучасні інформаційні технології.* 2022. № 4. С. 73–79.
10. Романюк О. В., Мельник М. В. Методи оптимізації цифрових схем у програмованих логічних пристроях // *Комп'ютерні системи та мережі*. 2021. № 5. С. 62–69.
11. Бондар І. В., Черненко А. С. Використання мови Verilog для моделювання цифрових систем // *Вісник ХНУРЕ*. 2023. № 2. С. 58–65.
12. Мельник О. М., Кравець О. С. Проєктування цифрових пристроїв на основі FPGA із застосуванням мов опису апаратури // *Радіоелектронні і комп'ютерні системи*. 2022. № 3. С. 80–87.
13. Соловйов С. І., Ткаченко О. В. Аналіз архітектур програмованих логічних інтегральних схем // *Інформаційні технології та комп'ютерна інженерія*. 2020. № 49. С. 33–40.
14. Кравчук Ю. В., Петренко М. О. Моделювання цифрових систем на мовах опису апаратури : навч. посіб. Одеса : ОНПУ, 2021. 240 с.
15. Поліщук А. М., Савченко І. П. Проєктування та дослідження цифрових систем на ПЛІС. Київ : НУБіП України, 2023. 270 с.
16. Brown S., Vranesic Z. Fundamentals of Digital Logic with Verilog Design. 4th ed. New York : McGraw-Hill Education, 2021. 960 p.
17. Harris D., Harris S. Digital Design and Computer Architecture. 3rd ed. Morgan Kaufmann, 2021. 720 p.

18. Palnitkar S. Verilog HDL: A Guide to Digital Design and Synthesis. Updated ed. SunSoft Press, 2020. 370 p.
19. Chu P. FPGA Prototyping by Verilog Examples. 2nd ed. Hoboken : John Wiley & Sons, 2021. 672 p.
20. Kilts S. Advanced FPGA Design: Architecture, Implementation and Optimization. Hoboken : Wiley-IEEE Press, 2020. 544 p.
21. Wolf W. FPGA-Based System Design. Boston : Pearson Education, 2021. 528 p.
22. LaForest C. FPGA Design Elements. Toronto : Self-published, 2020. 430 p.
23. Trimberger S. Three Ages of FPGAs: A Retrospective on the First Thirty Years. IEEE Press, 2021. 420 p.
24. Bergeron J. Writing Testbenches Using SystemVerilog. Springer, 2020. 510 p.
25. Bhasker J., Chadha R. Static Timing Analysis for Nanometer Designs. Springer, 2021. 380 p.
26. Ashenden P., Lewis J. The Designer's Guide to SystemVerilog. Springer, 2020. 650 p.
27. Smith M. Application-Specific Integrated Circuits. Addison-Wesley Professional, 2021. 1024 p.
28. O'Flynn C. The Hardware Hacking Handbook. No Starch Press, 2021. 420 p.
29. Intel Corporation. Intel FPGA Cyclone V Device Handbook. Intel, 2022. 1300 p.
30. Intel Corporation. Intel Quartus Prime Handbook. Intel, 2023. 900 p.
31. AMD (Xilinx). Vivado Design Suite User Guide. AMD Documentation, 2023. 870 p.
32. AMD (Xilinx). 7 Series FPGA Overview. AMD Documentation, 2022. 220 p.
33. Microchip Technology. PolarFire FPGA Architecture User Guide. Microchip, 2022. 430 p.
34. Gisselquist D. ZipCPU FPGA Design Tutorial. Gisselquist Technology, 2021. 350 p.
35. Wakerly J. Digital Design: Principles and Practices. 5th ed. Pearson, 2021. 910 p.
36. Brown S., Rose J. FPGA and CPLD Architectures: A Tutorial. IEEE Journal, 2020. 45 p.
37. Tessier R., Burleson W. Reconfigurable Computing for Digital Signal Processing. Springer, 2020. 500 p.
38. Hauck S., DeHon A. Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation. Morgan Kaufmann, 2020. 780 p.
39. Patterson D., Hennessy J. Computer Organization and Design RISC-V Edition. Morgan Kaufmann, 2021. 800 p.

40. Mano M., Ciletti M. Digital Design with RTL Design, Verilog and VHDL. Pearson, 2021. 880 p.
41. Pedroni V. Circuit Design and Simulation with Verilog HDL. MIT Press, 2021. 520 p.
42. Chu P. RTL Hardware Design Using Verilog. Wiley, 2022. 610 p.
43. Altera (Intel). Cyclone IV FPGA Device Handbook. Intel Corporation, 2021. 980 p.
44. Xilinx. UltraScale Architecture and Product Data Sheet. AMD, 2023. 150 p.

Електронне навчальне видання

Максим Олександрович Притула
Ігор Андрійович Дудатьєв
Ярослав Олександрович Осадчук

ПРОГРАМОВАНІ ЛОГІЧНІ ІНТЕГРАЛЬНІ СХЕМИ ТА ЇХ ПРОГРАМУВАННЯ

Навчальний посібник

Рукопис оформив: *М. Притула*

Редактор *Т. Старічек*

Оригінал-макет виготовила *Т. Старічек*

Підписано до видання 26.05.2026 р.
Гарнітура Times New Roman.
Зам. № Р2026-067.

Видавець та виготовлювач
Вінницький національний технічний університет,
Редакційно-видавничий відділ.
ВНТУ, ГНК, к. 114.
Хмельницьке шосе, 95, м. Вінниця, 21021.
press.vntu.edu.ua;
E-mail: irvc.ed.vntu@gmail.com.
Свідоцтво суб'єкта видавничої справи
серія ДК № 3516 від 01.07.2009 р.