КОМП’ЮТЕРНА ЛОГІКА

ЛЕКЦІЯ 14

FPGA та сучасний цикл розробки цифрової апаратури

LUT, тригери, BRAM, DSP, синтез, place-and-route, timing та bitstream

Державний університет «Житомирська політехніка»
2026–2027 навчальний рік

Про що цей розділ

Розділ пояснює, як перевірений код SystemVerilog перетворюється на фізичну цифрову систему в FPGA. Матеріал починається з будови кристала, продовжується сучасним design flow і завершується часовим аналізом та наскрізним прикладом 4-бітного ALU.

Після опрацювання матеріалу

- визначати призначення LUT, FF, BRAM, DSP, clock та I/O ресурсів
- розрізняти синтез, placement, routing і bitstream
- обчислювати період clock і тлумачити slack
- прогнозувати апаратну структуру за RTL-кодом
- пояснювати роль testbench і constraints

1 FPGA та її ресурси

Від перевіреного RTL до апаратури

У симуляції	У кристалі
логічні значення сигналів; ідеальний clock; умовні затримки; повна видимість	фізичні LUT і тригери; реальні маршрути; затримки та skew; обмежена видимість

Симулятор працює з моделлю, а FPGA — з фізичними електричними ресурсами. У функціональній симуляції логіка може бути правильною, хоча реальний шлях від одного регістра до іншого надто довгий для обраної частоти. Тактовий сигнал у testbench зазвичай ідеальний, тоді як на платі він проходить через спеціальну мережу з власними затримками. Внутрішні сигнали в симуляції можна додати на waveform, але на платі їх не видно без спеціального аналізатора. Тому перевірений testbench є необхідною, але не остаточною умовою успіху: після нього потрібні синтез, аналіз ресурсів, часовий аналіз і перевірка на цільовій платі.

Що таке FPGA

- програмована структура логічних блоків
- конфігурація визначає функції та з’єднання
- паралельне виконання апаратних операцій
- можливість багаторазового перепрограмування

FPGA — інтегральна схема, внутрішню структуру якої користувач налаштовує після виготовлення. Конфігураційна пам’ять визначає, яку функцію виконує кожна таблиця відповідності, де використовуються тригери і як програмовані маршрутизатори з’єднують блоки. Після конфігурації схема працює як спеціалізована апаратура: суматори, регістри й автомати діють паралельно, а не виконуються процесором по черзі. Більшість навчальних FPGA можна перепрограмовувати багато разів. Це робить їх зручними для прототипування, навчання, цифрової обробки сигналів, мережевих пристроїв, керування та систем, де потрібна висока паралельність.

FPGA, мікроконтролер і ASIC

Ознака	FPGA	Мікроконтролер	ASIC
Що змінює розробник	апаратну структуру	програму	структуру кристала

Ознака	FPGA	Мікроконтролер	ASIC
Паралельність	висока	обмежена ядром	найвища
Початкова вартість	середня	низька	дуже висока
Перепрограмування	так	так	ні
Типове застосування	прототип, прискорювач	керування	масове виробництво

FPGA, мікроконтролер і ASIC розв'язують різні задачі. Мікроконтролер виконує програму на готовому процесорному ядрі, тому зручний для послідовних алгоритмів і керування. FPGA дає змогу створити власні паралельні апаратні шляхи та змінювати їх конфігурацією. ASIC (Application-Specific Integrated Circuit, спеціалізована інтегральна схема) проєктується для конкретної функції й після виготовлення не змінюється, зате може мати найкращі швидкість, енергоспоживання та собівартість у великій серії. Вибір залежить від обсягу виробництва, потрібної продуктивності, термінів розробки та допустимого ризику.

Основні ресурси FPGA

Послідовність або складові
LUT + FF → BRAM → DSP → Clock → I/O

Сучасна FPGA складається не з окремих універсальних вентилів, а з повторюваних спеціалізованих ресурсів. LUT реалізують довільні комбінаційні функції обмеженої кількості входів. Тригери зберігають стан. BRAM (Block Random Access Memory, блокова оперативна пам'ять) зберігає масиви даних. DSP-блоки прискорюють множення, додавання та накопичення. Тактові ресурси розподіляють clock по кристалу з малим перекосом, а I/O-блоки з'єднують внутрішню логіку з контактами корпусу. Програмована мережа між цими ресурсами часто визначає не меншу частину затримки, ніж сама логіка.

Конфігурований логічний блок

Послідовність або складові
Входи → LUT → MUX → D-тригер → Вихід

Типовий конфігурований логічний блок поєднує LUT, тригер і мультиплексор. LUT обчислює комбінаційну функцію. Результат можна подати безпосередньо на вихід блока або записати у D-тригер на активному фронті clock. Мультиплексор вибирає потрібний шлях. Саме тому один і той самий фізичний блок може реалізувати як чисту комбінаційну логіку, так і регістровий етап. Конкретні виробники об'єднують такі елементи у slices, logic elements або adaptive logic modules, але базова ідея залишається однаковою: невелика програмована функція плюс локальна пам'ять стану.

LUT як таблиця істинності

- LUT з k входами зберігає 2^k бітів
- адреса формується вхідним набором
- збережений біт стає виходом
- функція задається конфігурацією

LUT (Look-Up Table, таблиця відповідності) можна уявляти як дуже малу пам'ять. Для кожної комбінації k входів у конфігурації зберігається один вихідний біт. Вхідний вектор працює як

адреса, а вибраний біт стає значенням функції. LUT із чотирма входами має 16 комірок і може реалізувати будь-яку булеву функцію чотирьох змінних. З погляду RTL не потрібно вручну записувати таблицю: синтезатор сам обчислює її вміст із виразу, case або if. Важливо розуміти лише, що складніша функція може потребувати кількох LUT і додаткового рівня з'єднань.

Приклад LUT для XOR

A	B	Адреса AB	F=A⊕B
0	0	00	0
0	1	01	1
1	0	10	1
1	1	11	0

Для функції XOR два входи утворюють чотири адреси. У конфігураційній пам'яті LUT записується послідовність 0, 1, 1, 0. Коли A і B змінюються, вони вибирають одну з цих комірок. Фізично LUT не перемикає ланцюжок окремих XOR-вентилів; вона видає попередньо запрограмоване значення для адреси. Такий погляд пояснює, чому різні булеві вирази однакової кількості входів можуть займати однаковий ресурс. Синтезатор мінімізує й розкладає логіку так, щоб використати доступні LUT максимально ефективно.

Місткість LUT і декомпозиція

- функція вкладається в одну LUT, якщо має допустиму кількість входів
- ширші функції діляться на частини
- додатковий рівень збільшує затримку
- спільні підвирази можуть економити ресурси

Кількість входів LUT обмежена архітектурою кристала. Якщо функція залежить від більшої кількості незалежних сигналів, синтезатор розкладає її на кілька LUT і з'єднує їх. Кожний додатковий логічний рівень та маршрут збільшує затримку. Водночас оптимізатор може знайти спільні підвирази й використати один проміжний результат для кількох виходів. Тому число операторів у SystemVerilog не дорівнює прямо числу LUT. Остаточний результат залежить від ширини сигналів, структури умов, цільової FPGA та параметрів оптимізації. Звіт синтезу показує фактичне використання ресурсів.

Тригер у логічному блоці

```
always_ff @(posedge clk) begin
   if (rst) q <= '0;
   else     q <= d;
end
```

Конструкція always_ff із позитивним фронтом описує групу D-тригерів. Кожний біт q потребує окремого тригера. На фронті clock значення d запам'ятовується, а між фронтами q зберігається. Синхронний reset реалізується логікою на вході D або спеціальною керувальною можливістю ресурсу — це залежить від архітектури та стилю коду. Неблокувальне присвоєння <= відображає одночасне оновлення всіх тригерів. Якщо регістр використовується між комбінаційними етапами, він скорочує критичний шлях, але додає один такт затримки.

Швидкі ланцюги перенесення

Послідовність або складові
A_0+B_0 → carry → A_1+B_1 → carry → A_2+B_2 → carry → A_3+B_3

Звичайне з’єднання перенесення через програмовану мережу було б повільним, тому FPGA мають спеціальні швидкі carry chains. Синтезатор розпізнає додавання, віднімання, інкремент і частину порівнянь та відображає їх на цей ланцюг. Перенесення проходить між сусідніми бітами коротким виділеним маршрутом. Саме тому вираз y=a+b зазвичай кращий за ручне складання суматора з вентилів: інструмент бачить арифметичний намір і використовує оптимізований ресурс. Для широких операцій критичним може стати саме час проходження перенесення, хоча він значно менший, ніж через загальну маршрутизацію.

Програмовані з’єднання

- локальні маршрути між сусідніми блоками
- довгі лінії для віддалених сигналів
- програмовані комутатори
- затримка залежить від маршруту

Логічні блоки корисні лише тоді, коли їх можна з’єднати. FPGA містить ієрархічну мережу провідників і програмованих комутаторів. Короткі локальні маршрути мають малу затримку, довгі лінії переносять сигнал через більшу частину кристала, а комутатори обирають напрямок. Після синтезу ще невідомо, де саме буде розміщено кожний елемент, тому точні затримки з’являються лише після place-and-route. Невдале розміщення може зробити короткий логічний вираз повільним через довгий маршрут. Саме тому часовий аналіз після реалізації важливіший за оцінку лише за кількістю логічних рівнів.

Блоки введення та виведення

- зв’язок внутрішніх сигналів із контактами корпусу
- напрямок input, output або bidirectional
- вибір електричного стандарту
- обмеження напруги й швидкості

I/O-блоки з’єднують логіку FPGA з кнопками, світлодіодами, пам’яттю та іншими мікросхемами. У файлі обмежень порт RTL пов’язується з фізичним контактом плати й електричним стандартом. Помилкове призначення контакту може дати неправильну поведінку, а несумісний стандарт напруги — створити електричний ризик. Напрямок порту визначається кодом, але фізичні параметри задаються окремо. Для навчального проєкту важливо користуватися офіційним файлом плати або її схемою, а не вгадувати номери pin. Перед конфігурацією завжди перевіряють банк вводу-виводу та допустиму напругу.

Блокова пам’ять BRAM

Регістри	BRAM
малі обсяги; довільний доступ; дорогі для великих масивів	великі вбудовані блоки; синхронний доступ; ефективні FIFO і буфери

Великий масив неефективно будувати з окремих тригерів, тому FPGA мають блокову пам’ять BRAM. Вона організовується з потрібною шириною та глибиною, може мати один або два порти й

зазвичай працює синхронно: адреса подається до фронту, дані з'являються після фронту. Синтезатор може вивести BRAM із правильно написаного масиву SystemVerilog. У BRAM реалізують таблиці, буфери кадрів, FIFO, пам'ять інструкцій і даних. Якщо код має асинхронне читання або незвичну схему reset, інструмент може не розпізнати шаблон і побудувати пам'ять із LUT або регістрів.

Арифметичні блоки DSP

- апаратне множення
- додавання та накопичення
- конвеєрні регістри
- цифрові фільтри та обробка сигналів

DSP-блоки є спеціалізованими арифметичними вузлами. Вони виконують множення, додавання, віднімання та операцію multiply-accumulate швидше й економніше, ніж еквівалентна логіка з LUT. Блок часто містить внутрішні конвеєрні регістри, які дають високу частоту за ціною кількох тактів затримки. Синтезатор розпізнає відповідні арифметичні вирази, але ширина та знаковість операндів мають бути задані точно. DSP використовують у фільтрах, перетвореннях, машинному навчанні, керуванні двигунами й обробці зображень. Для простого 4-бітного ALU окремий DSP зазвичай не потрібний: додавання реалізує carry chain.

Тактові ресурси

Послідовність або складові
Clock pin → PLL/MMCM → Global buffer → Clock regions → Registers

Тактовий сигнал повинен приходити до великої кількості регістрів майже одночасно. Для цього FPGA має виділені глобальні та регіональні clock-мережі з малим skew — різницею часу прибуття фронту. PLL або MMCM можуть множити, ділити й фазово зміщувати частоту. Clock не слід створювати звичайною логікою або розводити як довільний сигнал, якщо архітектура не передбачає такого режиму. Для дозволу оновлення зазвичай використовують enable усередині always_ff, а не логічно вимикають clock. Правильне використання тактових ресурсів робить часову поведінку передбачуваною.

Ресурси та utilization

Ресурс	Що реалізує	Що перевіряти
LUT	комбінаційну логіку	кількість і рівні
FF	стан і конвеєр	кількість регістрів
BRAM	масиви даних	ширину та глибину
DSP	множення і MAC	ширини операндів
I/O	порти плати	pin та стандарт

Звіт utilization показує, скільки фізичних ресурсів використано. Високий відсоток не є автоматично помилкою, але ускладнює трасування й залишає менше запасу для розвитку проєкту. Нульове використання очікуваного ресурсу може бути підказкою: наприклад, масив не виведено в BRAM або множення не потрапило в DSP. Надто велика кількість LUT іноді означає невдалу ширину сигналу, неповну оптимізацію або реалізацію великого мультиплексора. Порівнювати слід не лише загальний відсоток, а й ресурси конкретних модулів та критичні шляхи.

2 Сучасний цикл розробки

Повний цикл розробки

Послідовність або складові
Специфікація → RTL → Симуляція → Синтез → Place & Route → Bitstream → Плата

Цикл починається зі специфікації: функцій, інтерфейсів, частоти й допустимих затримок. RTL-код реалізує цю поведінку. Функціональна симуляція перевіряє логіку до прив'язки до конкретного кристала. Синтез перетворює RTL на мережу ресурсів. Place-and-route розміщує елементи й прокладає фізичні маршрути. Статичний часовий аналіз перевіряє, чи всі шляхи вкладаються в обмеження. Після успіху генерується bitstream і завантажується в плату. Апаратна перевірка може виявити помилки pin, reset, clock або взаємодії із зовнішнім світом, після чого цикл повторюється.

Етап 1 Специфікація

- функції та режими
- порти й розрядності
- clock і reset
- цільова частота
- граничні випадки

Специфікація повинна бути достатньо точною, щоб за нею можна було написати і RTL, і testbench. Для ALU потрібно визначити список операцій, коди op, розрядність a, b та y, значення прапорців і поведінку за переповнення. Для послідовнісного вузла додаються активний фронт, тип reset і затримка у тактах. Цільова частота належить до специфікації так само, як логічна функція: схема, що правильно рахує лише на нижчій частоті, не відповідає вимозі. Граничні випадки мають бути записані до початку кодування, а не вигадані після появи помилки.

Етап 2 RTL-опис

```
module block (...);
  always_comb begin
    // наступні значення
  end
  always_ff @(posedge clk) begin
    // регістри стану
  end
endmodule
```

RTL (Register-Transfer Level, рівень регістрових передач) розділяє стан і комбінаційну логіку між регістрами. always_comb формує виходи або наступні значення, always_ff записує стан на фронті. Синтезований код не повинен містити затримки #, системні завдання testbench або конструкції, які не мають апаратного змісту. Розрядність, signed/unsigned і пріоритет умов мають бути явними. Один регістр повинен мати одне джерело запису. Добрий RTL не прив'язується без потреби до конкретного виробника, але враховує типові шаблони виведення пам'яті, DSP та clock enable.

Етап 3 Функціональна симуляція

- DUT у testbench

- звичайні та граничні стимули
- автоматичні expected results
- waveform для налагодження
- PASS або FAIL

Функціональна симуляція виконується до синтезу і перевіряє логічну поведінку без точних фізичних затримок. Testbench подає стимули, обчислює або містить очікувані результати й автоматично порівнює їх із DUT. Для комбінаційного вузла потрібно перевірити всі операції, нуль, максимум, перенос і позику. Для послідовнісного — reset, утримання, переходи та переповнення. Waveform допомагає знайти причину, але не замінює автоматичні перевірки. Після кожної зміни запускається весь набір тестів, щоб нове виправлення не зламало раніше правильну поведінку.

Етап 4 Синтез

Послідовність або складові
RTL → Elaboration → Optimization → Technology mapping → Netlist

Синтез читає RTL, розгортає параметри та ієрархію, оптимізує логічні рівняння й відображає їх на ресурси цільової FPGA. Результатом є netlist — мережа логічних елементів та з'єднань. На цьому етапі інструмент може видалити невикористану логіку, об'єднати однакові вирази, перетворити арифметику на carry chains і вивести пам'ять або DSP. Попередження синтезу важливі: обрізання бітів, невикористані порти, неповні case, кілька драйверів або виведена latch часто вказують на логічну проблему. Успішне завершення синтезу не означає, що часові вимоги виконано.

Netlist і технологічне відображення

RTL	FPGA netlist
y = a + b; if / case; регістри; масиви	LUT; carry chain; FF; BRAM або DSP

Технологічне відображення пов'язує абстрактні операції з конкретними примітивами. Вираз додавання стає набором LUT і швидким carry chain; умова case — мережею мультиплексорів; always_ff — тригерами; правильно описаний масив — BRAM. Netlist усе ще не містить остаточних координат і маршрутів, але вже дозволяє оцінити ресурси. Перегляд схеми після синтезу корисний для навчання: можна побачити, чи відповідає апаратна структура задуму. Якщо замість очікуваного регістра з'явилася latch або пам'ять реалізовано сотнями LUT, потрібно повернутися до RTL і перевірити шаблон коду.

3 Обмеження та часовий аналіз

Файл обмежень

```
# приклад ідеї обмежень
set_property PACKAGE_PIN <PIN> [get_ports clk]
create_clock -period 10.000 [get_ports clk]
set_property IOSTANDARD <STANDARD> [get_ports clk]
```

RTL описує логіку, але не знає, до якого контакту плати під'єднаний clock або якою має бути його частота. Це задає файл обмежень. Синтаксис залежить від інструмента: у середовищі AMD/Xilinx використовують XDC, в Intel — SDC і файли призначень. PACKAGE_PIN пов'язує порт із фізичним контактом, IOSTANDARD задає електричний стандарт, create_clock — період тактового

сигналу. Значення не можна копіювати з випадкового прикладу: pin і напруга беруться з документації конкретної плати. Без clock constraint часовий аналіз не знає, до якої швидкості має прагнути.

Обмеження тактового сигналу

Частота	Період	Формула
50 МГц	20 нс	T=1/f
100 МГц	10 нс	T=1/f
125 МГц	8 нс	T=1/f

Період і частота пов’язані формулою T=1/f. Для 100 МГц один такт триває 10 нс. Це не означає, що вся комбінаційна логіка може використовувати всі 10 нс: частина бюджету потрібна на затримку clock-to-Q вихідного регістра, setup time вхідного регістра, skew і невизначеності clock. create_clock повідомляє інструменту період, після чого статичний часовий аналіз перевіряє всі регістр-регістр шляхи. Якщо constraint помилково занадто слабкий, звіт може бути зеленим, але реальна вимога не виконуватиметься.

Часовий шлях між регістрами

Послідовність або складові
FF source → clock-to-Q → Combinational logic → Routing → FF destination

Синхронний часовий шлях починається у тригері-джерелі після активного фронту. Через clock-to-Q його вихід змінюється, потім сигнал проходить комбінаційну логіку та маршрутизацію і повинен стабілізуватися на D-вході тригера-приймача до наступного фронту. Статичний часовий аналіз додає затримки всіх елементів і порівнює час прибуття з допустимим часом. Він не перебирає конкретні дані, а математично аналізує всі можливі шляхи. Критичним називається шлях із найменшим запасом. Оптимізація часто спрямована не на весь проєкт, а саме на кілька критичних шляхів.

Setup time

- дані мають стабілізуватися до фронту
- перевіряється максимальна затримка
- порушення означає надто повільний шлях
- виправлення: конвеєр, простіша логіка, нижча частота

Setup time — мінімальний час, протягом якого D-вхід має бути стабільним перед активним фронтом. Для setup-перевірки важливий найповільніший можливий шлях. Якщо дані прибувають запізно, запас slack стає від’ємним. Типові способи виправлення: додати конвеєрний регістр і розділити обчислення на два такти; спростити або переписати логіку; покращити розміщення; використати спеціалізований ресурс; зменшити частоту. Просте збільшення затримки в testbench нічого не змінює, бо проблема існує у фізичному шляху між реальними регістрами.

Hold time

- дані мають не змінюватися одразу після фронту
- перевіряється мінімальна затримка
- надто швидкий шлях теж може бути проблемою

- маршрутизатор зазвичай додає затримку автоматично

Hold time — інтервал після активного фронту, протягом якого D-вхід повинен залишатися стабільним. На відміну від setup, тут небезпечний надто швидкий шлях: нові дані можуть досягти приймача до завершення утримання старих. Реалізатор FPGA зазвичай виправляє hold-порушення, змінюючи маршрут або додаючи затримку, не змінюючи логіку. Зменшення частоти майже не допомагає, бо hold перевіряється біля того самого фронту, а не до наступного. У навчальних проєктах setup-порушення трапляються частіше, але розуміння hold потрібне для повної часової моделі.

Slack і timing closure

Slack	Тлумачення	Результат
> 0	дані прибувають із запасом	вимога виконана
= 0	запасу немає	граничний стан
< 0	дані запізнюються	timing violation

Slack — різниця між допустимим і фактичним часом прибуття даних. Додатний slack означає запас, нульовий — роботу на межі, від'ємний — порушення. Timing closure називають процес досягнення виконання всіх часових обмежень. Він може вимагати кількох ітерацій RTL, синтезу та place-and-route. Важливо дивитися не лише на найгірший шлях, а й на кількість порушених шляхів, правильність constraints та тактові домени. Позитивний slack за неправильних або відсутніх обмежень не доводить, що проєкт відповідає реальним вимогам.

4 Реалізація та конфігурація

Етап 5 Place and Route

Placement	Routing
вибір фізичних LUT і FF; групування пов'язаних ресурсів; урахування clock regions	вибір провідників; програмування комутаторів; оцінка точних затримок

Placement призначає кожному елементу netlist конкретне місце в кристалі. Близьке розміщення пов'язаних блоків скорочує маршрути, але інструмент одночасно враховує доступність carry chains, BRAM, DSP, clock regions та I/O. Routing прокладає з'єднання доступними провідниками й комутаторами. Після цього відомі значно точніші затримки, тому виконується остаточний статичний часовий аналіз. Якщо ресурсів забагато або мережа перевантажена, реалізація може не знайти маршрут. Зміна RTL, конвеєризація або вибір більшого кристала можуть бути потрібні навіть за правильній логіці.

Етап 6 Генерація bitstream

- кодує функції LUT
- задає маршрути та параметри I/O
- налаштовує BRAM, DSP і clock
- створюється після успішної реалізації

Bitstream — конфігураційний файл, який задає стан програмованих ресурсів FPGA. У ньому закодовано вміст LUT, з'єднання комутаторів, режими блоків пам'яті, DSP, clock і I/O. Це не машинний код для послідовного виконання процесором: після завантаження конфігурації виникає

апаратна структура. Bitstream генерують лише для точно визначеної моделі кристала. Перед генерацією інструмент перевіряє правила проєктування й часові вимоги. Файл не можна переносити на іншу FPGA, навіть якщо RTL однаковий. Для безпеки реальних систем можуть застосовувати шифрування та автентифікацію конфігурації.

Конфігурація FPGA

Послідовність або складові
Bitstream → JTAG або Flash → Configuration memory → Startup → User logic

Під час налагодження bitstream часто завантажують через JTAG безпосередньо в конфігураційну пам’ять FPGA. У багатьох SRAM-based FPGA ця пам’ять енергозалежна, тому після вимкнення живлення конфігурація зникає. Для автономного запуску bitstream записують у зовнішню flash-пам’ять, звідки кристал завантажує його під час старту. Після конфігурації проходить startup-послідовність, і користувацька логіка починає працювати. Конкретний спосіб залежить від плати та сімейства. Важливо відрізняти тимчасове програмування через JTAG від постійного запису у flash.

Перевірка на платі

- перевірити clock і reset
- зіставити pin constraints зі схемою плати
- подавати відомі тестові набори
- спостерігати LED, UART або логічний аналізатор
- порівнювати з testbench

Апаратна перевірка починається з найпростіших ознак життя: правильний clock, керований reset і відомий вихід. Потрібно подавати ті самі тестові набори, що були перевірені у симуляції, і порівнювати результати. Якщо логіка в симуляції правильна, а плата не працює, першими перевіряють pin constraints, активний рівень кнопок і LED, частоту clock, debounce механічних кнопок та reset. Внутрішні сигнали недоступні безпосередньо, тому корисно тимчасово вивести стан через LED, UART або вбудований логічний аналізатор. Зміни для діагностики мають бути контрольованими й не маскувати помилку.

Вбудований логічний аналізатор

Послідовність або складові
Internal signals → Probe core → BRAM capture → JTAG → Waveform viewer

Вбудований логічний аналізатор вставляється у проєкт як додаткове ядро. Його probes під’єднуються до вибраних внутрішніх сигналів, trigger визначає момент захоплення, а BRAM зберігає вибірку. Дані передаються на комп’ютер через JTAG і відображаються як waveform. Це дає видимість, подібну до симуляції, але обмежену кількістю probes, глибиною пам’яті й частотою захоплення. Аналізатор споживає ресурси й може змінити розміщення, тому його слід використовувати усвідомлено. Найкраще спочатку локалізувати проблему симуляцією, а на платі перевіряти конкретну гіпотезу.

5 Наскрізний приклад 4 бітного ALU

Приклад проєкту 4-бітний ALU

op	Операція	Результат
00	AND	y=a&b
01	OR	y=a|b
10	ADD	y=a+b
11	SUB	y=a-b

Приклад поєднує всі етапи. Специфікація задає чотири операції й двобітний код op. Входи a та b і вихід y мають по чотири біти. Для ADD прапорець flag означає перенос, для SUB — позику, для логічних операцій він дорівнює нулю. zero активний, якщо результат дорівнює нулю. Testbench перевіряє кожну операцію, а також 15+1 і 0-1 як граничні випадки. Після синтезу логічні операції відображаються на LUT, додавання й віднімання використовують carry chain, а case створює мультиплексування результату.

RTL-код 4-бітного ALU

```
always_comb begin
   {flag, y} = '0;
   case (op)
     2'b00: y = a & b;
     2'b01: y = a | b;
     2'b10: {flag,y} = {1'b0,a}+{1'b0,b};
     2'b11: begin y=a-b; flag=(a<b); end
   endcase
end
assign zero = (y == '0);
```

На початку блока задаються значення за замовчуванням, тому всі виходи визначені для кожного шляху. Для додавання a і b розширюються до п’яти бітів, щоб старший біт результату став переносом. Для віднімання y автоматично обрізається до чотирьох бітів, а flag окремо повідомляє про позику. Логічні операції не формують перенесення. zero є комбінаційною функцією готового y. Код не містить clock і стану, тому весь ALU є комбінаційним. Після зміни входів виходи змінюються після фізичної затримки LUT і маршрутів, але не чекають тактового фронту.

Очікувана структура після синтезу

Послідовність або складові
AND/OR LUT → Adder/Sub carry → Operation MUX → flag logic → zero reduction

Синтезатор не створює чотири повністю незалежні пристрої буквально. Він оптимізує спільну логіку й використовує ресурси цільового кристала. AND та OR реалізуються в LUT. Додавання й віднімання можуть спільно використовувати carry chain із керованою інверсією та початковим переносом. Код op керує мультиплексором результату. zero утворюється редукційною перевіркою всіх бітів y. Точне число LUT залежить від архітектури FPGA й оптимізації, тому правильна відповідь до синтезу — прогноз типів ресурсів, а не фіксована кількість. Фактичні значення беруться зі звіту utilization.

Призначення портів плати

Входи	Виходи
a[3:0] → switches; b[3:0] → switches; op[1:0] → switches	y[3:0] → LEDs; flag → LED; zero → LED

Для демонстрації ALU на навчальній платі входи можна під’єднати до перемикачів, а виходи — до світлодіодів. Це лише логічне зіставлення; конкретні номери pin залежать від плати. У constraints кожний біт порту отримує окремий фізичний контакт і I/O standard. Потрібно врахувати активний рівень: іноді LED світиться за логічного нуля, а кнопка натиснута за нульового рівня. Комбінаційний ALU не потребує clock. Якщо перемикачі змінюються, y реагує без такту. Для механічних кнопок можливий bounce, але для статичних перемикачів він зазвичай не заважає демонстрації.

Симуляція і плата доповнюють одна одну

Перевірка	Симуляція	FPGA-плата
Усі комбінації	легко автоматизувати	незручно вручну
Внутрішні сигнали	повністю видимі	потрібні probes
Фізичні pin	не перевіряються	перевіряються
Timing	модель або без затримок	реальна робота
Повторюваність	висока	залежить від оточення

Симуляція та апаратна перевірка не замінюють одна одну. Testbench швидко й повторювано перевіряє багато наборів і показує внутрішні сигнали. Плата підтверджує фізичні pin, електричні рівні, справжній clock і взаємодію з периферією. Ручне перемикання не може замінити самоперевірний testbench: людина легко пропускає комбінації й помилки. З іншого боку, ідеальна симуляція не виявить переплутаний контакт або неправильну напругу I/O. Надійний процес спочатку доводить логічну правильність автоматично, потім перевіряє timing і лише після цього переходить до контрольованого випробування на платі.

6 Практичні правила та підсумок

Типові помилки RTL для FPGA

- неповне присвоєння в always_comb
- блокувальне = у always_ff
- кілька драйверів одного сигналу
- неявні ширини та signed/unsigned
- clock, створений звичайною логікою
- великі асинхронні масиви

Частина помилок компілюється, але створює неочікувану апаратуру. Неповне присвоєння в always_comb виводить latch. Блокувальне присвоєння в always_ff може дати симуляційну поведінку, що не відповідає паралельному оновленню тригерів. Кілька процесів, які записують один сигнал, створюють конфлікт драйверів. Неявне розширення signed/unsigned змінює арифметику. Clock, сформований LUT, має поганий skew і glitches. Великий масив з асинхронним читанням може не перейти в BRAM. Усі warnings синтезу потрібно прочитати, а не вважати успішну генерацію bitstream доказом правильності.

Reset, початковий стан і запуск

У симуляції	На FPGA
регістр до reset часто X; testbench явно формує reset; легко повторити старт	є startup-послідовність; можливі initial values; зовнішній reset має бути надійним

У симуляції неініціалізований регістр зазвичай має X, тому testbench повинен явно активувати reset. FPGA після конфігурації може підтримувати початкові значення регістрів, але поведінка залежить від архітектури й інструмента. Для переносимого дизайну важливо мати визначену стратегію запуску: синхронний або асинхронний reset, тривалість сигналу та синхронне зняття в кожному clock domain. Зовнішня кнопка reset є асинхронною до clock і може потребувати синхронізатора та debounce. Невизначений старт часто проявляється лише на платі, хоча в окремій симуляції все виглядало правильно через випадкову ініціалізацію.

Метастабільність і синхронізатор

Послідовність або складові
Async input → FF1 → можлива metastability → FF2 → Synchronized signal

Асинхронний вхід може змінитися близько до фронту й порушити setup або hold першого тригера. Тоді він тимчасово потрапляє у метастабільний аналоговий стан, який не моделюється звичайним 0/1 RTL. Для однобітного рівневого сигналу використовують два послідовні тригери в цільовому clock domain: перший приймає ризик, другий значно зменшує ймовірність поширення нестабільності. Такий синхронізатор не підходить для довільної багатобітної шини, бо біти можуть перейти в різні такти; для неї потрібні handshake, FIFO або спеціальний протокол переходу між clock domains.

Інструменти розробки

Група	Приклади	Призначення
Вендорські	Vivado, Quartus	повний flow для відповідних FPGA
Симулятори	Icarus, Verilator	компіляція і симуляція
Перегляд waveform	EPWave, GTKWave	аналіз VCD/FST
Відкритий flow	Yosys та сумісні P&R	синтез і реалізація підтримуваних сімейств

Вендорські середовища мають повний доступ до архітектури відповідних FPGA, генерації bitstream і вбудованих аналізаторів. Безкоштовні симулятори зручні для навчання та автоматичних тестів. Переглядачі waveform відкривають VCD або інші формати. Відкриті інструменти підтримують частину сімейств і корисні для розуміння flow та автоматизації. Синтаксис constraints і доступні примітиви відрізняються, але основні етапи однакові: HDL, симуляція, синтез, place-and-route, timing, bitstream. Навички читання звітів і правильного RTL переносяться між інструментами краще, ніж запам’ятовування розташування кнопок.

Контрольний список перед bitstream

- усі автоматичні тести PASS
- warnings синтезу переглянуто
- utilization відповідає очікуванню

- clock і pin constraints задано
- setup та hold без порушень
- reset і clock-domain crossings перевірено

Генерація bitstream повинна бути завершенням перевіреного процесу, а не способом випадково побачити, чи запрацює плата. Спочатку всі функціональні тести мають пройти. Потім читаються warnings і звіт ресурсів: чи немає latch, обрізаних бітів або неочікуваної пам'яті з LUT. Перевіряються clock, pin та I/O constraints. Статичний часовий аналіз повинен показати не від'ємний slack для setup і hold. Для асинхронних входів та переходів між clock domains потрібні відповідні синхронізатори або протоколи. Лише після цього bitstream має зміст як реалізація специфікації.

Підсумок

- FPGA реалізує апаратну структуру, а не послідовну програму
- RTL відображається на LUT, FF, BRAM, DSP та routing
- синтез і place-and-route є різними етапами
- constraints роблять часові та фізичні вимоги перевірними
- успішний bitstream не замінює testbench і timing analysis

FPGA перетворює перевірений RTL на реальну паралельну структуру. LUT реалізують комбінаційні функції, FF зберігають стан, BRAM і DSP виконують спеціалізовані задачі, а програмовані маршрути з'єднують усе разом. Синтез визначає, які ресурси потрібні; place-and-route вибирає їх фізичне розташування й маршрути; timing analysis перевіряє швидкість; bitstream задає конфігурацію кристала. Специфікація, testbench і constraints є трьома різними формами вимог: функціональною, перевірочною та фізично-часовою. Надійний результат отримують лише тоді, коли всі три узгоджені.

7 Запитання для самоперевірки

1. Чим FPGA відрізняється від мікроконтролера?
2. Як LUT реалізує булеву функцію?
3. Що роблять синтез і place-and-route?
4. Для чого потрібен clock constraint?
5. Чим setup відрізняється від hold?
6. Що означає від'ємний slack?
7. Які ресурси очікуються для 4-бітного ALU?
8. Чому bitstream не є машинним кодом?
9. Коли потрібний двотригерний синхронізатор?
10. Чому перевірка на платі не замінює testbench?

8 Словник термінів

Термін	Значення
FPGA	програмована користувачем вентильна матриця
LUT	таблиця відповідності для комбінаційної функції

Термін	Значення
FF	тригер, що зберігає один біт
BRAM	вбудована блокова пам’ять
DSP	спеціалізований арифметичний блок
RTL	рівень регістрових передач
Netlist	мережа елементів і з’єднань після синтезу
Placement	фізичне розміщення елементів
Routing	прокладання фізичних з’єднань
Bitstream	файл конфігурації FPGA
Setup time	час стабільності даних до фронту
Hold time	час стабільності даних після фронту
Slack	запас між допустимим і фактичним часом
Timing closure	досягнення виконання всіх часових обмежень
Metastability	нестабільний аналоговий стан тригера через порушення часових вимог