КОМП’ЮТЕРНА ЛОГІКА

## ЛЕКЦІЯ 13

# Симуляція та верифікація цифрових схем

DUT, testbench, стимули, генерація clock і reset, часові діаграми, граничні випадки, assertions та налагодження

# Про що ця лекція

Верифікація (verification) — це перевірка того, що цифрова схема поводиться саме так, як вимагає її опис, тобто специфікація. Симуляція (simulation) — основний інструмент верифікації: програма-симулятор обчислює, які значення матимуть сигнали схеми в кожний момент часу, ще до того, як схему буде реалізовано в апаратурі.

На лекціях 11 і 12 було написано модулі мовою SystemVerilog: суматор, мультиплексор, регістр, лічильник, скінченний автомат. Сьогодні відповідаємо на наступне запитання: як довести, що написаний модуль працює правильно. Для цього потрібні три речі: тестове оточення (testbench), яке подає на модуль вхідні сигнали; часова діаграма (waveform), на якій видно поведінку сигналів; автоматичні перевірки, які самі повідомляють про помилку.

Матеріал безпосередньо потрібний для лабораторної роботи 7 «Моделювання, testbench та реалізація скінченного автомата» і лабораторної роботи 8, де перевіряється 4-бітний арифметико-логічний пристрій (Arithmetic Logic Unit, ALU). Усі приклади лекції використовують уже знайомі модулі: adder із лекції 11, counter і автомат IDLE → LOAD → RUN → DONE із лекції 12.

## Після лекції ви зможете

- пояснити, що таке пристрій, що перевіряється (DUT), і тестове оточення (testbench);
- написати testbench зі стимулами, тактовим сигналом clock і сигналом скидання reset;
- прочитати часову діаграму та знайти на ній помилку;
- дібрати граничні випадки для перевірки модуля;
- застосувати самоперевірку й твердження assert.

**Як користуватися конспектом.** Усі скорочення розшифровано під час першого вживання і зібрано в таблиці наприкінці. Приклади з обчисленнями розв’язано покроково. Повні тексти програм — у додатку та в архіві прикладів до лекції.

# 1. Навіщо потрібна верифікація

## Навіщо перевіряти цифрову схему

Компілятор перевіряє лише синтаксис: чи правильно розставлено дужки, оголошено сигнали, узгоджено типи. Він не знає, що саме мала робити схема. Лічильник, який рахує до 10 замість 9, компілюється без жодного попередження.

| Де знайдено помилку | Ціна виправлення |
|---|---|
| Симуляція | Хвилини: змінити рядок і запустити знову |
| Плата FPGA | Години: повторний синтез і пошук без повної видимості сигналів |
| Виготовлена мікросхема ASIC | Новий виробничий цикл: місяці та значні кошти |

В апаратурі ціна помилки зростає з кожним етапом. У симуляції виправлення займає хвилини. На платі з програмованою користувачем вентильною матрицею (Field-Programmable Gate Array, FPGA) потрібен повторний синтез, а внутрішні сигнали видно лише частково. У спеціалізованій інтегральній схемі (Application-Specific Integrated Circuit, ASIC) помилку після виготовлення вже не виправити: потрібна нова партія кристалів.

Тому головне правило проєктування цифрової апаратури: помилку слід знайти якомога раніше, тобто в симуляції.

## Верифікація, симуляція, тестування

Специфікація (specification) — це опис потрібної поведінки: словами, таблицею істинності, таблицею переходів або часовою діаграмою. Наприклад: «чотирибітний лічильник із синхронним скиданням і дозволом рахунку».

| Поняття | Запитання, на яке воно відповідає |
|---|---|
| Специфікація | Що схема повинна робити? |
| Симуляція | Що схема робить насправді? |
| Верифікація | Чи збігається одне з другим? |
| Налагодження | Чому не збігається і де помилка? |

Симуляція (simulation) — програмне моделювання роботи схеми. Симулятор читає опис мовою SystemVerilog і обчислює значення всіх сигналів у часі. Верифікація (verification) — ширше поняття: це весь процес, який доводить відповідність схеми специфікації. Симуляція є її основним, але не єдиним методом.

Налагодження (debugging) починається тоді, коли верифікація виявила розбіжність: потрібно знайти причину помилки, виправити її і повторити перевірку.

## Цикл Design → Testbench → Simulation → Debugging

Робота над цифровим модулем є циклом із чотирьох кроків. Design — проєктування: пишеться опис модуля. Testbench — створюється тестове оточення, яке подає вхідні сигнали й перевіряє виходи. Simulation — симулятор виконує testbench разом із модулем. Debugging — якщо результат не збігається з очікуваним, шукають і виправляють причину.

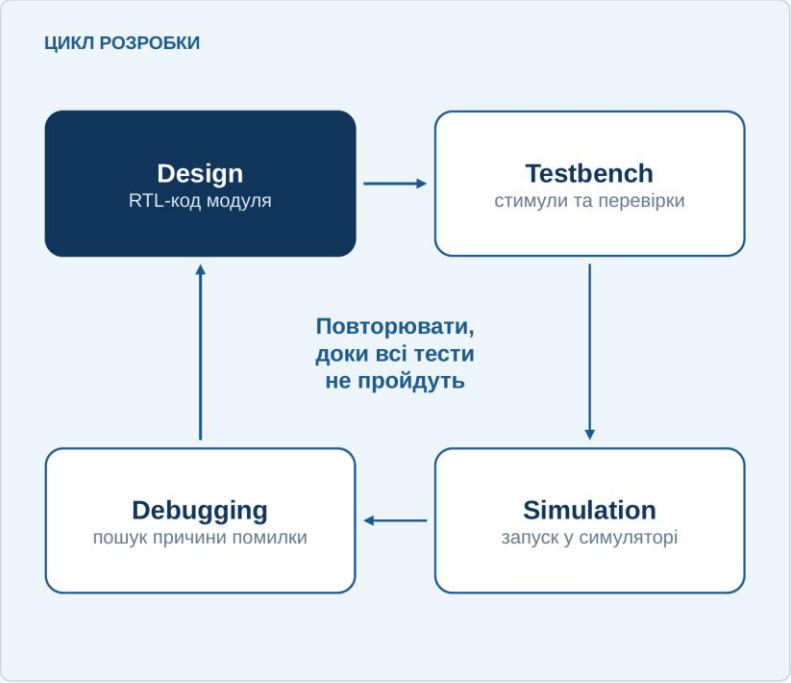


Рис. 1. Цикл розробки цифрового модуля

Після виправлення цикл запускається знову, причому повторюються всі тести, а не лише той, що виявив помилку: виправлення в одному місці може зламати інше. Повторний запуск усіх раніше пройдених тестів називають регресійним тестуванням (regression testing).

# 2. DUT і testbench

## DUT — пристрій, що перевіряється

DUT (Device Under Test) — пристрій, що перевіряється. Це той самий модуль, який потім буде синтезовано: adder, counter, fsm_ctrl. У літературі трапляється також скорочення UUT (Unit Under Test) з тим самим значенням.

Головне правило: текст DUT під час перевірки не змінюється. Якщо для зручності тесту додати в модуль рядки, які потім доведеться вилучити, то перевірено буде вже інший модуль. Testbench взаємодіє з DUT так, як це робитиме реальне оточення, — через вхідні та вихідні порти.

**Запам'ятайте:** входи → [ DUT ] → виходи

## Testbench — тестове оточення

Testbench (тестове оточення, скорочено TB) — це модуль SystemVerilog, написаний спеціально для перевірки. Він не має власних портів, бо є найвищим рівнем симуляції: усередині нього створюється екземпляр DUT.




Рис. 2. Структура тестового оточення

Testbench виконує три ролі. Перша — генерувати стимули (stimulus), тобто значення, які подаються на входи DUT. Друга — спостерігати виходи. Третя — перевіряти, чи збігаються виходи з очікуваними значеннями. Для послідовнісних схем додається четверта роль: формування тактового сигналу clock і сигналу скидання reset.

Testbench ніколи не синтезується, тому в ньому дозволені конструкції, які не мають апаратного відповідника: затримки, друк повідомлень, цикли з очікуванням часу.

## Код проєкту і код testbench

Код модуля, який буде синтезовано, називають RTL-кодом. RTL (Register-Transfer Level) — рівень регістрових передач: опис схеми як регістрів і комбінаційної логіки між ними. Саме так було написано всі модулі лекцій 11 і 12.

| Конструкція | У проєкті (RTL) | У testbench |
|---|---|---|
| assign, always_comb, always_ff | так | так |
| initial | ні | так |
| затримка #10 | ні | так |
| $display, $finish | ні | так |

Код testbench підпорядковується іншим правилам. Він може містити блок initial, затримки на зразок #10 і системні завдання $display або $finish. Жодна з цих конструкцій не має апаратного відповідника, тому в RTL-коді їх не вживають.

Ще одна відмінність: оператори всередині блоку initial виконуються по черзі, згори донизу, як у звичайній програмі. Це не суперечить паралельній природі апаратури, бо testbench описує не схему, а сценарій її випробування.

# 3. Перший testbench

## Каркас testbench

Будь-який testbench має однакову будову. Спочатку директива `timescale задає одиниці часу. Далі оголошується модуль без списку портів. Усередині оголошуються сигнали типу logic: по одному на кожний порт DUT. Потім створюється екземпляр DUT, порти якого з'єднуються із цими сигналами. Наприкінці розташовано блок initial зі сценарієм.

```
`timescale 1ns/1ps

module tb_adder;          // портів немає
    logic [3:0] a, b;     // входи DUT
    logic       cin;
    logic [3:0] sum;      // виходи DUT
    logic       cout;

    adder #(.W(4)) dut (
        .a(a), .b(b), .cin(cin),
        .sum(sum), .cout(cout)
    );

    initial begin
        // стимули та перевірки
    end
endmodule
```

У прикладі перевіряється параметризований суматор adder із лекції 11. Сигнали a, b і cin testbench записує сам, тому вони стають входами DUT. Сигнали sum і cout testbench лише читає: їх формує DUT.

## Підключення DUT

Створення екземпляра модуля називають інстанціюванням (instantiation). Запис adder #(.W(4)) dut (...) читається так: створити екземпляр модуля adder, задати параметру W значення 4, назвати екземпляр dut.

| Фрагмент | Значення |
|---|---|
| `adder` | ім’я модуля, який перевіряється |
| `#(.W(4))` | значення параметра: розрядність 4 біти |
| `dut` | ім’я екземпляра |
| `.a(a)` | порт a з’єднано із сигналом a |
| `.sum(sum)` | вихід sum потрапляє в сигнал sum |

У дужках перелічено з’єднання за іменами (named port connection). Запис .a(a) означає: порт a модуля з’єднати із сигналом a, оголошеним у testbench. Ліворуч, після крапки, стоїть ім’я порту; у дужках — ім’я сигналу. З’єднання за іменами не залежить від порядку портів в оголошенні модуля, тому його вважають надійнішим за позиційне.

## Блок initial і затримка #

Блок initial починає виконуватися в момент часу 0 і виконується один раз. Оператори між begin та end ідуть по черзі. Запис #10 зупиняє виконання блоку на 10 одиниць модельного часу; з директивою `timescale 1ns/1ps це 10 наносекунд.

```
initial begin
    a = 4'd3;  b = 4'd5;  cin = 1'b0;
    #10;
    $display("%0d + %0d + %0d = %0d (%b)",
             a, b, cin, {cout, sum}, {cout, sum});

    a = 4'd9;  b = 4'd8;  cin = 1'b0;
    #10;
    $display("%0d + %0d + %0d = %0d (%b)",
             a, b, cin, {cout, sum}, {cout, sum});

    a = 4'd15; b = 4'd15; cin = 1'b1;
    #10;
    $display("%0d + %0d + %0d = %0d (%b)",
             a, b, cin, {cout, sum}, {cout, sum});

    $finish;
end
```

Кожний тест складається з трьох дій. Спочатку входам присвоюються значення. Потім затримка #10 дає комбінаційній схемі час сформувати результат. Після цього системне завдання $display друкує значення. Запис {cout, sum} є конкатенацією: перенесення і сума об’єднуються в одне п’ятибітне число.

Системне завдання $finish завершує симуляцію. Без нього симулятор зупиниться сам лише тоді, коли не залишиться жодної запланованої події.

## Одиниці часу: `timescale

Директива `timescale 1ns/1ps складається з двох частин. Перша задає одиницю часу: число після символу # вимірюється в наносекундах. Друга задає точність: час у симуляції округлюється до одного пікосекундного кроку.

| Запис | Значення |
|---|---|
| `` `timescale 1ns/1ps `` | одиниця 1 нс, точність 1 пс |
| `#10` | 10 нс |
| `#2.5` | 2,5 нс = 2500 пс |
| `$time` | поточний час в одиницях, тобто в нс |

Наносекунда (нс, ns) дорівнює $10^{-9}$ секунди, пікосекунда (пс, ps) — $10^{-12}$ секунди, тобто 1 нс = 1000 пс. Системна функція $time повертає поточний модельний час в одиницях, заданих директивою.

Символ ` на початку — це зворотний апостроф (backtick), а не звичайний апостроф. Ним починаються директиви компілятора.

## Системні завдання симулятора

Системні завдання (system tasks) — вбудовані команди симулятора. Їх легко впізнати за знаком $ на початку імені. $display друкує один рядок у момент виклику. $monitor викликають один раз, після чого симулятор сам друкує рядок щоразу, коли змінюється будь-який із перелічених сигналів.

| Завдання | Призначення |
|---|---|
| `$display` | надрукувати рядок один раз |
| `$monitor` | друкувати за кожної зміни сигналів |
| `$time` | поточний модельний час |
| `$error` | повідомити про помилку |
| `$finish` | завершити симуляцію |
| `$dumpfile, $dumpvars` | записати часову діаграму у файл |

$error друкує повідомлення про помилку разом з іменем файла, номером рядка і модельним часом. $finish завершує симуляцію. Пара $dumpfile і $dumpvars вмикає запис значень сигналів у файл для подальшого перегляду часової діаграми.

## Покроковий розбір першого тесту

Розберемо кожний набір окремо. Перший: 3 + 5 + 0 = 8. Число 8 у п’яти бітах записується як 01000. Старший біт є перенесенням cout = 0, чотири молодші — сумою sum = 1000.

| Час | a + b + cin | Результат | cout | sum |
|---|---|---|---|---|
| 10 нс | 3 + 5 + 0 | $8 = 01000_2$ | 0 | 1000 |
| 20 нс | 9 + 8 + 0 | $17 = 10001_2$ | 1 | 0001 |
| 30 нс | 15 + 15 + 1 | $31 = 11111_2$ | 1 | 1111 |

Другий: 9 + 8 + 0 = 17. Чотирибітна сума вміщує лише значення від 0 до 15, тому 17 у неї не поміщається. Розкладаємо: 17 = 16 + 1. Доданок $16 = 2^4$ відповідає перенесенню cout = 1, а залишок 1 записується в sum = 0001. Разом {cout, sum} = 10001.

Третій: 15 + 15 + 1 = 31 — найбільший можливий результат. 31 = 16 + 15, отже cout = 1, sum = 1111. Симулятор друкує: «3 + 5 + 0 = 8 (01000)», «9 + 8 + 0 = 17 (10001)», «15 + 15 + 1 = 31 (11111)».

# 4. Самоперевірка та граничні випадки

## Самоперевірний testbench

У першому тестовому оточенні результат оцінювала людина, читаючи консоль. Для трьох рядків це можливо, для трьохсот — ні. Самоперевірний testbench (self-checking testbench) містить очікуване значення і сам порівнює його з фактичним.

```
logic [4:0] expected;

initial begin
    a = 4'd9;  b = 4'd8;  cin = 1'b0;
    #10;
    expected = 5'd17;
    if ({cout, sum} !== expected)
        $error("got %0d, expected %0d",
               {cout, sum}, expected);
    else
        $display("PASS");
    $finish;
end
```

Очікуване значення записано у змінну expected. Умова if порівнює з нею {cout, sum}. Якщо значення різні, системне завдання $error друкує повідомлення з обома числами. Слова PASS і FAIL — усталені позначення: тест пройдено або не пройдено.

Оператор !== порівнює всі чотири можливі стани біта: 0, 1, X (невідоме значення) і Z (високий імпеданс, тобто вихід ні до чого не підключено). Звичайний оператор != у разі значення X дає результат X, умова if його не вважає істинною, і помилка залишається непоміченою.

## Завдання task: одна перевірка — один виклик

Щоб не повторювати ті самі рядки для кожного набору, їх об’єднують у завдання (task). Завдання check отримує три аргументи, подає їх на входи DUT, чекає 10 нс, обчислює очікуваний результат і порівнює.

```
task automatic check(input logic [3:0] ta, tb,
                     input logic       tcin);
    logic [4:0] expected;
    a = ta;  b = tb;  cin = tcin;
    #10;
    expected = ta + tb + tcin;
    if ({cout, sum} !== expected) begin
        errors++;
        $error("%0d + %0d + %0d: got %0d, expected %0d",
               ta, tb, tcin, {cout, sum}, expected);
    end
endtask
```

Рядок expected = ta + tb + tcin є еталонною моделлю (reference model): це найпростіший запис того, що має обчислювати суматор. Змінна expected має п’ять бітів, тому перенесення в ній не втрачається. Лічильник errors збільшується на кожній розбіжності.

Ключове слово automatic означає, що кожний виклик завдання отримує власні копії локальних змінних.

## Граничні випадки

Граничний випадок (corner case) — це набір входів, у якому значення перебувають на межі допустимого діапазону або одночасно збігаються кілька рідкісних умов. Типові значення схема зазвичай обробляє правильно, бо саме про них думав автор. Помилки ховаються на межах.

| Модуль | Що обов’язково перевірити |
|---|---|
| Суматор | 0 + 0; максимум + максимум; перенесення через усі розряди |
| Лічильник | перехід 15 → 0; пауза en = 0; rst і en одночасно |
| Регістр зсуву | усі нулі; усі одиниці; перший і останній біт |
| FSM | кожний перехід; утримання стану; reset із кожного стану |

Для суматора це найменші й найбільші операнди та випадки, коли перенесення проходить через усі розряди. Для лічильника — переповнення, пауза рахунку та одночасна активність керувальних сигналів. Для скінченного автомата — кожний перехід, утримання в стані та скидання з будь-якого стану.

## Граничні випадки суматора

Завдяки завданню check кожний граничний випадок займає один рядок. Перший виклик перевіряє найменші значення: 0 + 0 + 0 = 0. Другий — найбільші: 15 + 15 + 1 = 31. Третій — 15 + 1 = 16: у двійковому записі 1111 + 0001 перенесення виникає в кожному розряді й доходить до cout. Четвертий — 8 + 8 = 16: перенесення виникає одразу в старшому значущому біті (Most Significant Bit, MSB).

```
initial begin
    check(4'd0,  4'd0,  1'b0);  // найменші значення
    check(4'd15, 4'd15, 1'b1);  // найбільші значення
    check(4'd15, 4'd1,  1'b0);  // 1111 + 1
    check(4'd8,  4'd8,  1'b0);  // перенесення з MSB

    if (errors == 0) $display("PASS");
    else $display("FAIL: %0d errors", errors);
    $finish;
end
```

Після всіх викликів testbench друкує підсумок. Якщо лічильник errors дорівнює нулю, виводиться PASS; інакше — FAIL і кількість помилок. Людині достатньо прочитати один останній рядок.

## Вичерпна перевірка

Вичерпна перевірка (exhaustive testing) перебирає всі можливі набори входів. Для чотирибітного суматора кількість наборів обчислюється так: вхід a має $2^4 = 16$ значень, вхід b — 16 значень, вхід cin — 2 значення. Разом $16 \cdot 16 \cdot 2 = 512$ наборів.

```
initial begin
    for (int i = 0; i < 16; i++)
        for (int j = 0; j < 16; j++)
            for (int k = 0; k < 2; k++) begin
                a = i[3:0];  b = j[3:0];  cin = k[0];
                #10;
                expected = i + j + k;
                tests++;
                if ({cout, sum} !== expected) errors++;
            end

    $display("tests=%0d errors=%0d", tests, errors);
    $finish;
end
```

Три вкладені цикли for перебирають i від 0 до 15, j від 0 до 15 і k від 0 до 1. Кожний набір триває 10 нс, тому вся симуляція займає $512 \cdot 10 = 5120$ нс модельного часу. Результат запуску: tests=512 errors=0.

Для 32-бітного суматора наборів було б $2^{32} \cdot 2^{32} \cdot 2 = 2^{65} \approx 3{,}7 \cdot 10^{19}$. Навіть за мільярда перевірок на секунду це понад тисячу років. Тому для широких шин вичерпну перевірку замінюють граничними випадками та випадковими наборами.

### Тест знаходить помилку

Щоб переконатися, що testbench справді здатний виявити помилку, у суматор навмисно внесено дефект: із виразу вилучено доданок cin. Такий модуль компілюється без попереджень.

```
// adder_bug.sv: у виразі втрачено cin
assign {cout, sum} = {1'b0, a} + {1'b0, b};

--- tb_adder_check: 4 граничні набори ---
ERROR: tb_adder_check.sv:24:
       15 + 15 + 1: got 30, expected 31
FAIL: 1 errors

--- tb_adder_full: усі 512 наборів ---
tests=512 errors=256
```

Testbench із чотирма граничними наборами повідомляє про одну помилку: для 15 + 15 + 1 отримано 30 замість 31. Три інші набори мають cin = 0 і тому дефекту не помічають. Вичерпна перевірка показує повну картину: 256 помилок із 512.

Число 256 можна передбачити. Помилка проявляється лише тоді, коли cin = 1, а таких наборів рівно половина: 16 значень a, помножені на 16 значень b, дають 256.

## 5. Перевірка послідовнісних схем

### Перевірка послідовнісних схем

Комбінаційну схему перевіряти просто: подали входи, зачекали, порівняли вихід. Послідовнісна схема має стан, який змінюється лише на активному фронті тактового сигналу. Тому testbench повинен сам формувати тактовий сигнал clk (clock) і сигнал скидання rst (reset).

Кожна перевірка складається з трьох кроків, які задає ключова модель: до фронту встановити входи, дочекатися активного фронту, після фронту порівняти стан з очікуваним. Саме цю послідовність лекція 12 сформулювала на слайді «Як перевіряти послідовнісну схему».

**Запам’ятайте:** ПІДГОТУВАТИ ВХОДИ → ФРОНТ → ПЕРЕВІРИТИ

### Генерація тактового сигналу

Тактовий сигнал у testbench формують два рядки: logic clk = 0; задає початкове значення, а always #5 clk = ~clk; кожні 5 нс замінює значення на протилежне. Блок always без списку подій повторюється нескінченно.

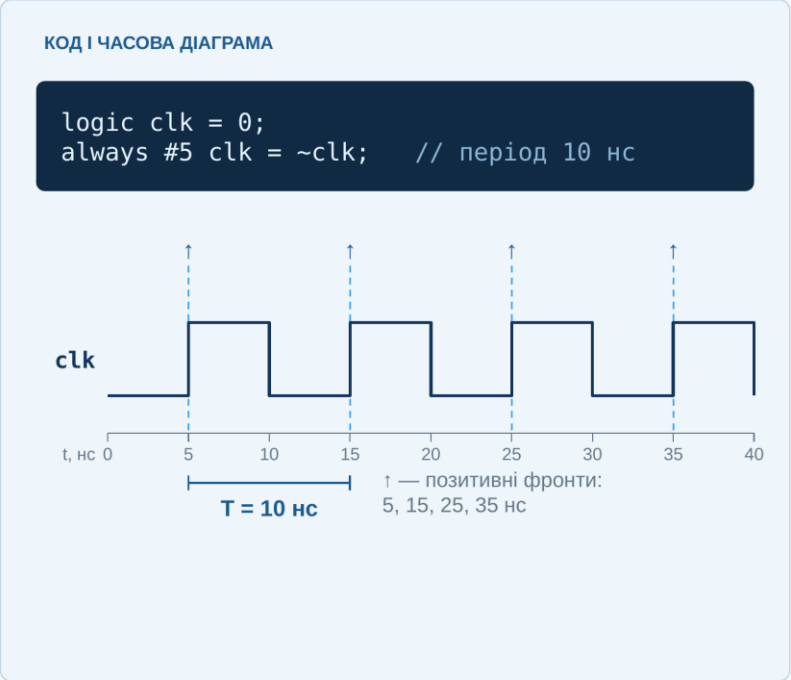


Рис. 3. Генератор тактового сигналу та його часова діаграма

Обчислимо параметри покроково. Півперіод дорівнює 5 нс. Повний період T складається з двох півперіодів: T = 2 · 5 = 10 нс. Частота є оберненою до періоду: f = 1 / T = 1 / (10 · $10^{-9}$ с) = $10^{8}$ Гц = 100 МГц (мегагерц, мільйон герців).

Оскільки clk починається з нуля, перший позитивний фронт настає в момент 5 нс, а наступні — через кожні 10 нс: 5, 15, 25, 35 нс. Негативні фронти припадають на 10, 20, 30 нс.

## Генерація reset

На початку симуляції регістри DUT не мають визначеного значення: симулятор показує X. Тому сценарій завжди починається зі скидання. У момент 0 встановлюється rst = 1. На першому позитивному фронті (5 нс) лічильник записує нуль, і завдання expect_count перевіряє це.

```
initial begin
    rst = 1;  en = 0;      // 1. скидання
    expect_count(4'd0);

    @(negedge clk);
    rst = 0;  en = 1;      // 2. рахунок
    expect_count(4'd1);
end
```

Після цього testbench чекає негативного фронту (10 нс), знімає скидання і вмикає дозвіл рахунку. До наступного позитивного фронту (15 нс) нові значення входів уже стабільні, тому лічильник упевнено переходить до значення 1.

## Керування часом у testbench

Testbench має два способи чекати. Затримка # зупиняє виконання на заданий інтервал. Керування подією @ зупиняє виконання до моменту, коли на сигналі відбудеться вказана зміна: @(posedge clk) чекає перехід clk з 0 в 1, @(negedge clk) — перехід з 1 в 0.

| Запис | Що робить |
|---|---|
| `#10;` | чекає 10 одиниць часу |
| `@(posedge clk);` | чекає наступний позитивний фронт |
| `@(negedge clk);` | чекає наступний негативний фронт |
| `repeat (3) @(posedge clk);` | чекає три позитивні фронти |

Конструкція repeat (N) повторює наступний оператор N разів. Запис repeat (3) @(posedge clk); пропускає три такти. Для синхронних схем очікування фронту надійніше за числові затримки: якщо змінити період тактового сигналу, сценарій із @ залишиться правильним.

## Коли змінювати входи і коли перевіряти

DUT читає свої входи в момент позитивного фронту. Якщо testbench змінить вхід саме в цей момент, результат залежатиме від того, яку з двох одночасних подій симулятор опрацює першою. Така ситуація називається гонкою (race condition): один і той самий код може дати різні результати.

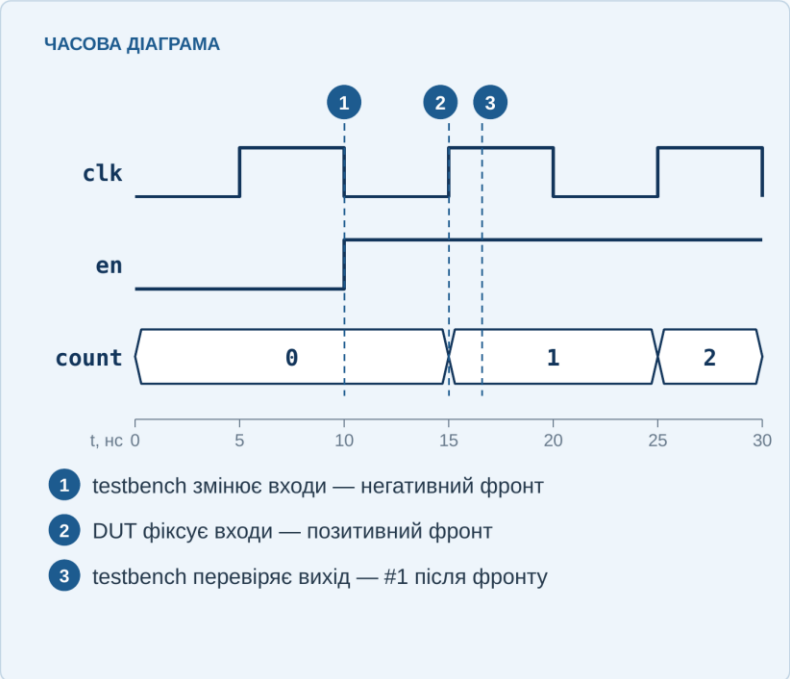


Рис. 4. Моменти зміни входів, фіксації та перевірки

Просте правило усуває гонку: testbench змінює входи на негативному фронті, тобто посередині між двома позитивними. До наступного позитивного фронту входи вже пів періоду стабільні.

Перевіряти вихід потрібно після того, як регістр оновився. Тому завдання expect_count чекає позитивний фронт, потім ще #1 — одну наносекунду — і лише тоді порівнює значення.

## Testbench лічильника: каркас

Каркас складається з уже знайомих частин. Сигнали rst, en, count і лічильник помилок errors оголошено так само, як у testbench суматора; на слайді показано лише clk із початковим значенням 0. Створено екземпляр чотирибітного лічильника counter із лекції 12. Один рядок always формує тактовий сигнал.

```
logic clk = 0;

counter #(.W(4)) dut (
    .clk(clk), .rst(rst), .en(en), .count(count)
);

always #5 clk = ~clk;      // період 10 нс

task automatic expect_count(input logic [3:0] expected);
    @(posedge clk);        // активний фронт
    #1;                    // регістр уже оновився
    if (count !== expected) begin
        errors++;
        $error("t=%0d: count=%0d, expected %0d",
               $time, count, expected);
    end
endtask
```

Завдання expect_count є аналогом check для послідовнісної схеми. Воно чекає позитивний фронт, робить паузу #1, щоб регістр устиг оновитися, і порівнює count з очікуваним значенням. У разі розбіжності друкується момент часу, фактичне та очікуване значення.

## Сценарій лічильника: кроки 1–3

Сценарій перевіряє шість ситуацій зі списку граничних випадків. На цьому слайді — перші три. Перша — скидання: у момент 0 встановлено rst = 1, і після першого позитивного фронту count має дорівнювати 0.

```
rst = 1;  en = 0;      // 1. скидання
expect_count(4'd0);

@(negedge clk);
rst = 0;  en = 1;      // 2. рахунок
expect_count(4'd1);
expect_count(4'd2);
expect_count(4'd3);

@(negedge clk);
en = 0;                // 3. утримання
expect_count(4'd3);
expect_count(4'd3);
```

Друга — рахунок: на негативному фронті знято скидання й увімкнено дозвіл, тому три наступні позитивні фронти дають значення 1, 2 і 3. Третя — утримання: на негативному фронті en стає нулем, і два наступні фронти не змінюють значення 3.

## Сценарій лічильника: кроки 4–6

Четверта ситуація — рахунок до максимуму. Після ввімкнення en лічильник має значення 3. Конструкція repeat (11) пропускає одинадцять фронтів: 3 + 11 = 14. Наступний виклик expect_count чекає ще один фронт і перевіряє значення 15.

```
@(negedge clk);
en = 1;                 // 4. рахунок до 15
repeat (11) @(posedge clk);
expect_count(4'd15);
expect_count(4'd0);     // 5. переповнення
expect_count(4'd1);

@(negedge clk);
rst = 1;                // 6. rst і en одночасно
expect_count(4'd0);

if (errors == 0) $display("PASS");
else $display("FAIL: %0d errors", errors);
$finish;
```

П’ята — переповнення: після 15 чотирибітний лічильник переходить у 0, далі в 1. Шоста — одночасно rst = 1 та en = 1: за специфікацією скидання має вищий пріоритет, тому очікується 0.

Наприкінці testbench друкує підсумок і викликає $finish. Результат запуску — один рядок PASS.

# 6. Часові діаграми

## Часова діаграма лічильника

Часова діаграма (waveform) показує значення сигналів у часі. Горизонтальна вісь — модельний час, кожний рядок — один сигнал. Однобітний сигнал зображують лінією на двох рівнях. Багаторозрядний сигнал, або шину, зображують смугою, усередині якої записано значення.

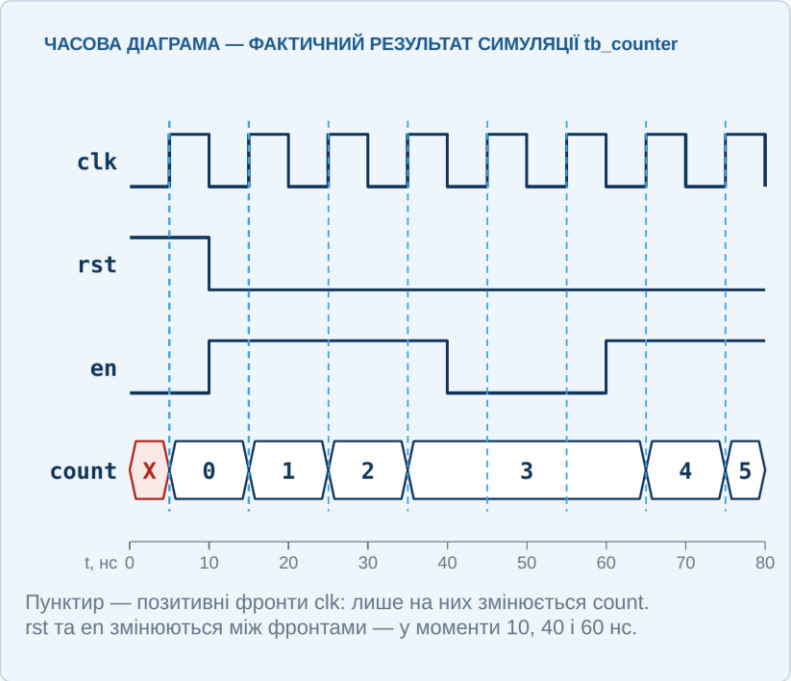


Рис. 5. Часова діаграма лічильника: фактичний результат симуляції tb_counter

На діаграмі показано перші 80 нс роботи testbench лічильника. До першого позитивного фронту (5 нс) count має невідоме значення X. На фронті 5 нс діє скидання, і count стає 0. У момент 10 нс testbench знімає rst і вмикає en. На фронтах 15, 25 і 35 нс лічильник набуває значень 1, 2 і 3.

У момент 40 нс en стає нулем, тому фронти 45 і 55 нс значення не змінюють. Після 60 нс рахунок продовжується: 4 на фронті 65 нс і 5 на фронті 75 нс.

## Як читати часову діаграму

Діаграму читають у три кроки. Спочатку знаходять активний фронт тактового сигналу. Потім дивляться, які значення мали входи безпосередньо перед цим фронтом. Нарешті перевіряють, чи відповідає новий стан правилу роботи схеми.

| Позначення | Значення | Типова причина |
|---|---|---|
| 0, 1 | логічні рівні | штатна робота |
| X | невідоме значення | регістр без скидання; непідключений вхід |
| Z | високий імпеданс | сигнал ніхто не формує |
| шина 0…F | значення в шістнадцятковому коді | вигляд за замовчуванням |

Крім рівнів 0 і 1, симулятор показує два особливі значення. X (unknown) — невідоме: симулятор не може визначити, 0 це чи 1. Найчастіше це регістр, який ще не отримав скидання, або вхід, якому testbench нічого не присвоїв. Z (high impedance, високий імпеданс) — сигнал, до якого не підключено жодного джерела.

Значення шин програми перегляду зазвичай показують у шістнадцятковій системі: число 10 буде записано як A, число 15 — як F.

## Покроковий розбір діаграми

Таблиця переводить діаграму в послідовність кроків. Правило лічильника з лекції 12: якщо rst = 1, наступне значення дорівнює 0; інакше якщо en = 1, значення збільшується на 1; інакше зберігається.

| Фронт | rst | en | count до | count після |
|---|---|---|---|---|
| 5 нс | 1 | 0 | X | 0 |
| 15 нс | 0 | 1 | 0 | 1 |
| 25 нс | 0 | 1 | 1 | 2 |
| 35 нс | 0 | 1 | 2 | 3 |
| 45 нс | 0 | 0 | 3 | 3 |
| 55 нс | 0 | 0 | 3 | 3 |
| 65 нс | 0 | 1 | 3 | 4 |
| 75 нс | 0 | 1 | 4 | 5 |

Фронт 5 нс: rst = 1, тому з невідомого X отримуємо 0. Фронти 15, 25 і 35 нс: rst = 0, en = 1, тому 0 → 1 → 2 → 3. Фронти 45 і 55 нс: en = 0, значення 3 зберігається. Фронти 65 і 75 нс: en знову 1, тому 3 → 4 → 5.

Кожний рядок таблиці відповідає одному виклику expect_count у сценарії. Якщо хоча б в одному рядку стовпець «count після» відрізнявся б від очікуваного, testbench надрукував би повідомлення з відповідним моментом часу.

## Запис часової діаграми у файл

Щоб симулятор зберіг часову діаграму, на початку сценарію викликають два системні завдання. $dumpfile("dump.vcd") задає ім’я файла. $dumpvars(0, tb_counter) вказує, що записувати: модуль tb_counter і всі вкладені в нього модулі. Перший аргумент 0 означає «на всю глибину ієрархії».

```
initial begin
    $dumpfile("dump.vcd");
    $dumpvars(0, tb_counter);
end
```

Файл має формат VCD (Value Change Dump) — «дамп змін значень». У ньому записано не значення в кожний момент, а лише моменти, коли сигнал змінився. Це текстовий формат, який розуміють усі програми перегляду.

На комп'ютері файл відкривають програмою GTKWave. В онлайн-середовищі EDA Playground достатньо ввімкнути параметр «Open EPWave after run»: після запуску діаграма відкриється у вбудованому переглядачі EPWave.

# 7. Перевірка скінченного автомата

## Testbench скінченного автомата

Скінченний автомат (Finite-State Machine, FSM) перевіряють за тим самим шаблоном, що й лічильник. Модуль fsm_ctrl містить автомат із лекції 12 зі станами IDLE, LOAD, RUN і DONE, входами start і finish та виходами busy і done.

```
rst = 1;  start = 0;  finish = 0;
expect_out(0, 0);      // IDLE

@(negedge clk);
rst = 0;  start = 1;
expect_out(1, 0);      // LOAD
expect_out(1, 0);      // RUN
expect_out(1, 0);      // RUN: finish = 0

@(negedge clk);
finish = 1;
expect_out(0, 1);      // DONE

@(negedge clk);
finish = 0;
expect_out(0, 1);      // DONE: start = 1

@(negedge clk);
start = 0;
expect_out(0, 0);      // IDLE
```

Завдання expect_out(busy, done) чекає позитивний фронт, паузу #1 і порівнює обидва виходи з очікуваними. Сценарій веде автомат усіма переходами: скидання в IDLE; start = 1 переводить у LOAD; далі безумовно RUN; RUN утримується, доки finish = 0; finish = 1 переводить у DONE; DONE утримується, доки start = 1; start = 0 повертає в IDLE.

## Траєкторія FSM у симуляції

Таблиця відтворює фактичний вивід симулятора. Перевірки відбуваються в моменти 6, 16, 26 нс і далі, тобто через 1 нс після кожного позитивного фронту.

| Час | Входи | state | busy | done |
|---|---|---|---|---|
| 6 нс | rst = 1 | IDLE | 0 | 0 |
| 16 нс | start = 1 | LOAD | 1 | 0 |
| 26 нс | — | RUN | 1 | 0 |
| 36 нс | finish = 0 | RUN | 1 | 0 |
| 46 нс | finish = 1 | DONE | 0 | 1 |
| 56 нс | start = 1 | DONE | 0 | 1 |
| 66 нс | start = 0 | IDLE | 0 | 0 |

Вихід busy дорівнює 1 у станах LOAD і RUN, вихід done — лише в стані DONE. Обидва виходи залежать тільки від стану, тому це автомат Мура. У момент 36 нс автомат залишається в RUN, бо finish ще дорівнює 0; у момент 56 нс залишається в DONE, бо start ще дорівнює 1.

Назву стану testbench друкує через ієрархічне ім'я dut.state: спочатку ім'я екземпляра, потім ім'я внутрішнього сигналу. Метод name() повертає текстову назву значення перелічуваного типу.

# 8. Assertions і налагодження

## Assertions: твердження

Твердження (assertion) — це умова, яка за специфікацією повинна бути істинною. Запис assert (умова) else $error(...) означає: перевірити умову; якщо вона хибна, надрукувати повідомлення.

```
// твердження: count завжди в межах 0…9
always @(negedge clk)
    assert (count < 10)
        else $error("t=%0d: count=%0d",
                    $time, count);
```

У прикладі перевіряється лічильник за модулем 10 із лекції 12: його значення завжди має бути в межах від 0 до 9. Блок always @(negedge clk) виконує перевірку на кожному негативному фронті, коли значення вже стабільне. На відміну від expect_count, твердження не потребує окремого виклику в сценарії: воно стежить за властивістю впродовж усієї симуляції.

Повідомлення мають чотири рівні серйозності: $info — інформація, $warning — попередження, $error — помилка, $fatal — критична помилка, після якої симуляція зупиняється.

## Негайні та паралельні твердження

Твердження з попереднього слайда називається негайним (immediate assertion): умова обчислюється в ту мить, коли виконання доходить до рядка assert.

```
// після такту з rst=1 лічильник дорівнює нулю
assert property (@(posedge clk)
    rst |=> count == 4'd0)
    else $error("reset failed");
```

Паралельне твердження (concurrent assertion) записують зі словом property. Воно прив'язане до тактового сигналу й може описувати поведінку впродовж кількох тактів. Запис rst |=> count == 4'd0 читається так: якщо на фронті rst = 1, то на наступному фронті count має дорівнювати нулю.

Набір таких конструкцій називають SVA (SystemVerilog Assertions). На цій лекції паралельні твердження розглядаються лише оглядово. Їх підтримують не всі безкоштовні симулятори: Icarus Verilog 12 цей приклад не компілює, Verilator 5 — виконує.

## Налагодження: приклад помилки

У лічильник за модулем 10 внесено типову помилку: поточне значення порівнюється з N, а не з N−1. Модуль компілюється, і перші дев’ять тактів поводиться правильно.

```
// counter_mod_bug.sv, N = 10
if (count == N) count <= '0;   // помилка
else count <= count + 1'b1;

--- консоль симулятора ---
ERROR: tb_counter_mod.sv:18: t=110: count=10
ERROR: tb_counter_mod.sv:18: t=220: count=10

// виправлення: останній дозволений стан N-1
if (count == N-1) count <= '0;
```

Твердження з попереднього слайда повідомляє про порушення в момент 110 нс: count = 10. Момент можна обчислити: скидання знято у 10 нс, лічильник набуває значення k на фронті 5 + 10 · k нс, отже значення 10 з’являється на фронті 105 нс, а найближчий негативний фронт, на якому працює перевірка, — 110 нс.

Причина: умова count == N спрацьовує лише тоді, коли count уже дорівнює 10, тому лічильник проходить одинадцять станів 0…10. Правильна умова — count == N-1: останній дозволений стан 9. Після виправлення симуляція завершується без повідомлень.

## Алгоритм налагодження

Налагодження (debugging) — це пошук причини, а не вгадування. Алгоритм має п’ять кроків. Перший: уважно прочитати повідомлення — воно містить момент часу і фактичне значення. Другий: відкрити часову діаграму саме в цей момент.

Третій: рухаючись назад у часі, знайти останній такт, на якому все ще було правильно. У прикладі це фронт 95 нс зі значенням 9. Четвертий: з’ясувати, що мало статися на наступному фронті і яка умова в коді за це відповідає. Мав відбутися перехід 9 → 0, за нього відповідає порівняння count == N.

П’ятий: виправити причину і запустити всі тести знову, а не лише той, що виявив помилку.

**Запам’ятайте:** ПОВІДОМЛЕННЯ → ЧАС → СИГНАЛ → ПРИЧИНА → ВИПРАВЛЕННЯ

# 9. Практичні правила

## Типові помилки testbench

Testbench — теж код, і в ньому теж бувають помилки. П’ять найчастіших. Перша: тактовий сигнал оголошено без початкового значення; інверсія X дає X, і тактів немає. Друга: забули $finish; через нескінченний генератор clk симуляція не зупиняється.

Третя: входи змінюються одночасно з активним фронтом, виникає гонка. Четверта: перевірка стоїть одразу після фронту без паузи й читає ще старе значення. П’ята, найпідступніша: тест лише друкує значення через $display, але ні з чим їх не порівнює — такий тест завжди «проходить».

**Запам’ятайте:** ТЕСТ ТАКОЖ ПОТРІБНО ПЕРЕВІРЯТИ

## Інструменти симуляції

Для запуску прикладів достатньо безкоштовних інструментів. Icarus Verilog складається з двох програм: iverilog компілює файли, vvp виконує симуляцію. Ключ -g2012 вмикає підтримку

SystemVerilog за стандартом 2012 року; без нього конструкції logic, always_ff і always_comb не розпізнаються. Ключ -o задає ім'я результату компіляції.

```
# 1. компіляція (Icarus Verilog)
iverilog -g2012 -o sim tb_counter.sv counter.sv

# 2. запуск симуляції
vvp sim

# 3. перегляд часової діаграми
gtkwave dump.vcd
```

Після запуску з'являється файл dump.vcd, який відкривають програмою GTKWave. Першим у командному рядку вказано файл testbench: тоді директива `timescale з нього поширюється і на файл модуля.

Якщо встановлювати програми немає змоги, той самий код запускається у вебсервісі EDA Playground: testbench вставляють у ліве вікно, модуль — у праве, вибирають симулятор і натискають Run.

### Контрольний список testbench

Перед здачею лабораторної роботи testbench варто перевірити за п'ятьма запитаннями. Чи починається сценарій зі скидання? Чи змінюються входи на негативному фронті, а не одночасно з активним? Чи кожна перевірка порівнює фактичне значення з очікуваним через !==?

Чи охоплено всі граничні випадки зі списку, складеного за специфікацією? Чи друкує testbench однозначний підсумок PASS або FAIL і чи завершується викликом $finish?

**Запам'ятайте:** СКИДАННЯ → СТИМУЛИ → ПЕРЕВІРКИ → ПІДСУМОК

### Підсумок

Головна думка лекції: написаний модуль стає надійним лише після перевірки. Пристрій, що перевіряється (DUT), розміщують у тестовому оточенні (testbench), яке подає стимули, спостерігає виходи і порівнює їх з очікуваними значеннями.

Для комбінаційної схеми достатньо подати входи й зачекати. Для послідовнісної потрібні тактовий сигнал і скидання, а входи змінюють між активними фронтами. Часова діаграма дає змогу побачити поведінку кожного сигналу, а твердження автоматично повідомляють про порушення правил.

Цикл Design → Testbench → Simulation → Debugging повторюється, доки всі тести не пройдуть. На наступній лекції перевірений модуль переноситься в реальну апаратуру: тема 14 — FPGA та сучасний цикл розробки цифрової апаратури.

**Запам'ятайте:** DESIGN → TESTBENCH → SIMULATION → DEBUGGING

## 10. Запитання для самоперевірки

Повна відповідь містить визначення, правило і приклад.

1. Чим testbench відрізняється від DUT?
2. Чому входи змінюють на негативному фронті?
3. Що означає X на часовій діаграмі?
4. Чим !== відрізняється від !=?
5. Скільки наборів потребує вичерпна перевірка 4-бітного суматора з cin?

6. Які період і частота задає always #5 clk = ~clk; за `timescale 1ns/1ps?
7. Навіщо в expect_count пауза #1 після @(posedge clk)?
8. Що робить $dumpvars(0, tb_counter)?
9. Чим твердження відрізняється від перевірки в сценарії?
10. Чому після виправлення помилки запускають усі тести?

# 11. Вправи для самостійної роботи

Вправи відповідають завданню 13 самостійної роботи «Розробка тестових сценаріїв і аналіз waveform». Розв'язання записуйте покроково, як у прикладах лекції.

**Вправа 1.** Скільки наборів потрібно для вичерпної перевірки компаратора comparator із лекції 11, якщо W = 3?

**Вправа 2.** У testbench записано `timescale 1ns/1ps і always #4 clk = ~clk; причому clk = 0 у момент 0. Знайдіть період, частоту і моменти перших трьох позитивних фронтів.

**Вправа 3.** Чотирибітний лічильник counter отримує на послідовних позитивних фронтах такі входи: rst = 1; далі rst = 0 і чотири такти en = 1, один такт en = 0, два такти en = 1. Запишіть значення count після кожного фронту.

**Вправа 4.** Напишіть самоперевірний testbench для мультиплексора mux2 із лекції 11 з W = 4. Передбачте щонайменше чотири набори і підсумок PASS / FAIL.

**Вправа 5.** У testbench оголошено logic clk; і записано always #5 clk = ~clk;. Симуляція запускається, але лічильник не рахує, а clk на діаграмі має значення X. Поясніть причину і виправте.

**Вправа 6.** Автомат fsm_ctrl перебуває в IDLE. Подано start = 1 і водночас finish = 1, обидва сигнали утримуються. Запишіть стани після чотирьох наступних фронтів.

# 12. Словник термінів

| Термін | Значення |
|---|---|
| Верифікація (verification) | Перевірка відповідності схеми специфікації. |
| Симуляція (simulation) | Програмне обчислення значень сигналів схеми в модельному часі. |
| Специфікація (specification) | Опис потрібної поведінки схеми. |
| Пристрій, що перевіряється (Device Under Test, DUT) | Модуль, поведінку якого перевіряє testbench. |
| Тестове оточення (testbench) | Несинтезований модуль без портів, який містить DUT, подає стимули та перевіряє виходи. |
| Стимул (stimulus) | Значення, яке testbench подає на вхід DUT. |
| Самоперевірний testbench (self-checking testbench) | Testbench, який сам порівнює фактичні значення з очікуваними. |
| Еталонна модель (reference model) | Простий опис правильної поведінки, з яким порівнюють DUT. |
| Граничний випадок (corner case) | Набір входів на межі діапазону або рідкісний збіг умов. |
| Вичерпна перевірка (exhaustive testing) | Перебір усіх можливих наборів входів. |
| Часова діаграма (waveform) | Графічне подання значень сигналів у часі. |
| Гонка (race condition) | Залежність результату від порядку опрацювання одночасних подій. |

| Термін | Значення |
|---|---|
| Твердження (assertion) | Умова, яка має бути істинною; її порушення симулятор фіксує автоматично. |
| Налагодження (debugging) | Пошук і усунення причини помилки. |
| Регресійне тестування (regression testing) | Повторний запуск усіх тестів після кожної зміни. |

# 13. Розшифрування скорочень

| Скорочення | Повна назва | Значення |
|---|---|---|
| DUT | Device Under Test | пристрій (модуль), що перевіряється |
| UUT | Unit Under Test | те саме, що DUT |
| TB | testbench | тестове оточення |
| RTL | Register-Transfer Level | рівень регістрових передач; синтезований опис схеми |
| HDL | Hardware Description Language | мова опису апаратури |
| FSM | Finite-State Machine | скінченний автомат |
| VCD | Value Change Dump | файл зі змінами значень сигналів |
| SVA | SystemVerilog Assertions | твердження мови SystemVerilog |
| EDA | Electronic Design Automation | автоматизація проєктування електроніки |
| FPGA | Field-Programmable Gate Array | програмована користувачем вентильна матриця |
| ASIC | Application-Specific Integrated Circuit | спеціалізована інтегральна схема |
| ALU | Arithmetic Logic Unit | арифметико-логічний пристрій (АЛП) |
| MSB | Most Significant Bit | старший значущий біт |
| clk | clock | тактовий сигнал |
| rst | reset | сигнал скидання |
| en | enable | сигнал дозволу |
| cin / cout | carry in / carry out | вхідне / вихідне перенесення |
| X | unknown | невідоме значення сигналу |
| Z | high impedance | високий імпеданс: сигнал ніхто не формує |
| нс (ns) | nanosecond | наносекунда, $10^{-9}$ с |
| пс (ps) | picosecond | пікосекунда, $10^{-12}$ с |
| МГц (MHz) | megahertz | мегагерц, $10^{6}$ Гц |
| PASS / FAIL | pass / fail | тест пройдено / тест не пройдено |
| IDLE, LOAD, RUN, DONE | назви станів автомата | очікування, завантаження, виконання, завершення |
| EPWave | EDA Playground Wave | переглядач часових діаграм у сервісі EDA Playground |

# Додаток. Повні тексти прикладів

Ці файли потрібні для лабораторної роботи 7. Запуск у Icarus Verilog: iverilog -g2012 -o sim tb_counter.sv counter.sv, далі vvp sim. Для автомата: iverilog -g2012 -o sim tb_fsm.sv fsm_ctrl.sv.

### counter.sv

```
// Лекція 12: лічильник із синхронним reset та enable
module counter #(parameter int W = 4) (
    input  logic         clk, rst, en,
    output logic [W-1:0] count
);
    always_ff @(posedge clk) begin
        if (rst)      count <= '0;
        else if (en)  count <= count + 1'b1;
    end
endmodule
```

### tb_counter.sv

```
// Лекція 13: testbench лічильника — clock, reset, сценарій, waveform
`timescale 1ns/1ps

module tb_counter;
    logic       clk = 0;
    logic       rst, en;
    logic [3:0] count;
    int         errors = 0;

    counter #(.W(4)) dut (
        .clk(clk), .rst(rst), .en(en), .count(count)
    );

    always #5 clk = ~clk;      // період 10 нс

    task automatic expect_count(input logic [3:0] expected);
        @(posedge clk);        // активний фронт
        #1;                    // регістр уже оновився
        if (count !== expected) begin
            errors++;
            $error("t=%0d: count=%0d, expected %0d",
                   $time, count, expected);
        end
    endtask

    initial begin
        $dumpfile("dump.vcd");
        $dumpvars(0, tb_counter);

        rst = 1;  en = 0;      // 1. скидання
        expect_count(4'd0);

        @(negedge clk);
        rst = 0;  en = 1;      // 2. рахунок
        expect_count(4'd1);
        expect_count(4'd2);
        expect_count(4'd3);

        @(negedge clk);
        en = 0;                // 3. утримання
        expect_count(4'd3);
        expect_count(4'd3);

        @(negedge clk);
        en = 1;                // 4. рахунок до 15
        repeat (11) @(posedge clk);
```

```
        expect_count(4'd15);
        expect_count(4'd0);    // 5. переповнення
        expect_count(4'd1);

        @(negedge clk);
        rst = 1;               // 6. rst і en одночасно
        expect_count(4'd0);

        if (errors == 0) $display("PASS");
        else $display("FAIL: %0d errors", errors);
        $finish;
    end
endmodule
```

### fsm_ctrl.sv

```
// Лекція 12: скінченний автомат IDLE → LOAD → RUN → DONE
module fsm_ctrl (
    input  logic clk, rst, start, finish,
    output logic busy, done
);
    typedef enum logic [1:0] {IDLE, LOAD, RUN, DONE} state_t;
    state_t state, next_state;

    always_ff @(posedge clk) begin
        if (rst) state <= IDLE;
        else     state <= next_state;
    end

    always_comb begin
        next_state = state;
        unique case (state)
            IDLE: if (start)  next_state = LOAD;
            LOAD:             next_state = RUN;
            RUN:  if (finish) next_state = DONE;
            DONE: if (!start) next_state = IDLE;
            default:          next_state = IDLE;
        endcase
    end

    assign busy = (state == LOAD) || (state == RUN);
    assign done = (state == DONE);
endmodule
```

### tb_fsm.sv

```
// Лекція 13: testbench скінченного автомата
`timescale 1ns/1ps

module tb_fsm;
    logic clk = 0;
    logic rst, start, finish;
    logic busy, done;
    int   errors = 0;

    fsm_ctrl dut (
        .clk(clk), .rst(rst),
        .start(start), .finish(finish),
        .busy(busy), .done(done)
    );

    always #5 clk = ~clk;

    task automatic expect_out(input logic exp_busy, exp_done);
        @(posedge clk);
        #1;
        $display("t=%0d state=%s busy=%b done=%b",
                 $time, dut.state.name(), busy, done);
```

```
        if (busy !== exp_busy || done !== exp_done) begin
            errors++;
            $error("expected busy=%b done=%b",
                   exp_busy, exp_done);
        end
    endtask

    initial begin
        $dumpfile("dump.vcd");
        $dumpvars(0, tb_fsm);

        rst = 1;  start = 0;  finish = 0;
        expect_out(0, 0);      // IDLE

        @(negedge clk);
        rst = 0;  start = 1;
        expect_out(1, 0);      // LOAD
        expect_out(1, 0);      // RUN
        expect_out(1, 0);      // RUN: finish = 0

        @(negedge clk);
        finish = 1;
        expect_out(0, 1);      // DONE

        @(negedge clk);
        finish = 0;
        expect_out(0, 1);      // DONE: start = 1

        @(negedge clk);
        start = 0;
        expect_out(0, 0);      // IDLE

        if (errors == 0) $display("PASS");
        else $display("FAIL: %0d errors", errors);
        $finish;
    end
endmodule
```