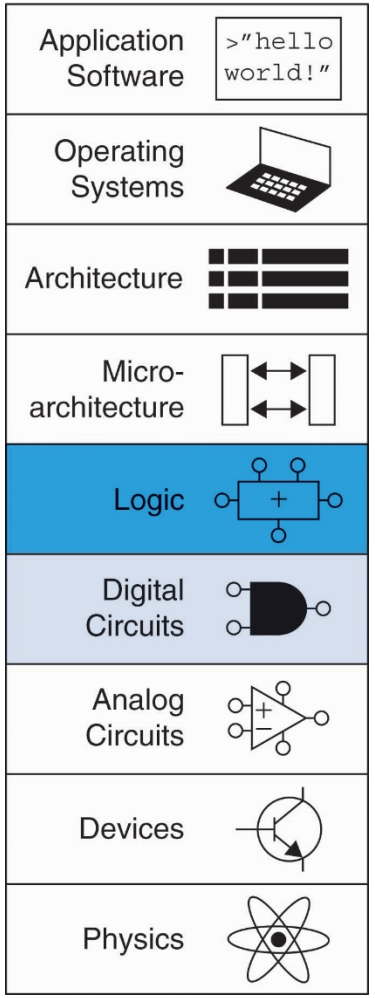
Розділ 4. Теми:

- Вступ
- Комбінаційна логіка
- Структурне моделювання
- Послідовнісна логіка
- Знову комбінаційна логіка
- Скінченні автомати
- Параметризовані модулі
- Середовище тестування



Вступ

- Мови опису і верифікації апаратури (HDL):
  - Визначають функціональність проектованого пристрою
  - Засоби САПР синтезують оптимізовану схему пристрою, яка складається з логічних елементів
- Більшість комерційних проектів побудована з використанням мов HDL
- Дві лідируючі мови HDL:

  - **SystemVerilog**
    - Розроблений в 1984році компанією Gateway Design Automation
    - Стандарт IEEE standard (1364) – в 1995
    - Розширений стандарт – в 2005 (IEEE STD 1800-2009)
    - Останній стандарт IEEE STD 1800-2012
  - **VHDL 2008**
    - Розроблений в 1981 міністерством оборони США
    - Стандарт IEEE standard (1076) – в 1987
    - Обновлений в 2008 (IEEE STD 1076-2008)

Від HDL описання до логічних елементів

- **Моделювання**
  - Тестові впливи подаються на входи
  - Аналіз виходів – для перевірки коректності роботи
  - Мільйони доларів, зекономлених при налагодженні в процесі моделювання, – замість тестування апаратури
- **Синтез**
  - Перетворення HDL коду у список з’єднань (*netlist)* апаратного модуля (список елементів і зв’язків між ними)

Важливо:

При використанні HDL потрібно думати про апаратну реалізацію HDL коду

Нові конструкції SystemVerilog

- С-синтаксис
- ООП
- Структури, черги, динамічні масиви, перечислення
- Перетворення типів
- Контроль поведінки програми за допомогою foreach, return, break, continue
- Semaphores, mailboxes
- Assertions

Універсальний тип logic (замість reg і wire)

Покращена ініціалізація типів

```
Reg [63:0] data = ‘1;  // data = ‘0;  data = ‘bz
```

Перераховні типи

```
enum {ONE,TWO, THREE} state;  // int за замовчуванням
enum {ONE  1, FIVE = 5, TEN = 10 } state;
$display(\n”Current state id %dc(%b)”, State.name);// друк значень
```

Можна задати тип явно

```
enum bit (TRUE, FALSE} Boolean;
enum logic [1:0] {WAIT, LOAD, READY} state;
```

Описання машини станів

```
enum {WAIT, LOAD, STORE} state, NextState;
always_ff @(posedge clock, negedge, resetN)
      if (!resetN) State <=WAITE;
      else State <= NextState;
always_comb begin
      case (State)
            WAITE: NextState = LOAD;
            LOAD: NextState = Store;
            Store: NextState = Wait;
      endcase
end
```

Структури

```
struct {
        int a,b;
        logic [23:0] address;
         bit error;
        } Instruction_Worg;
```

Об’єднаня

```
union {
        int i;
        int unsigned u;
        } data;
```

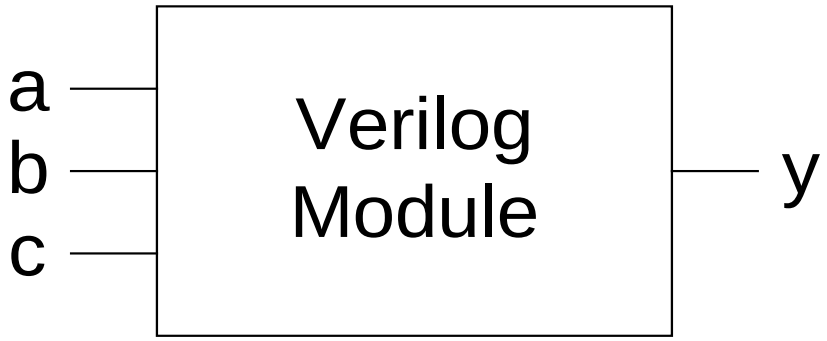
Багатовимірні масиви (зберігаються в пам’яті як вектор)

```
logic [3:0][7:0] data;
logic [1:0][3:0][7:0] data1;
```

Доступ до вектора

```
logic [1:0][1:0][7:0] a;
a[1][1][0]= 1’b0;
a=32’h1A3C5E7;
a[1][0][3:0] = 4’hF;
a[0]=16’hACE;
```

Модулі System Verilog



Два типа модулів:

- **Поведінковий:** описує, що робить модуль
- **Структурний:** визначає модуль як сукупність взаємозв’язаних більш простих модулів

Поведінкове описання на System Verilog

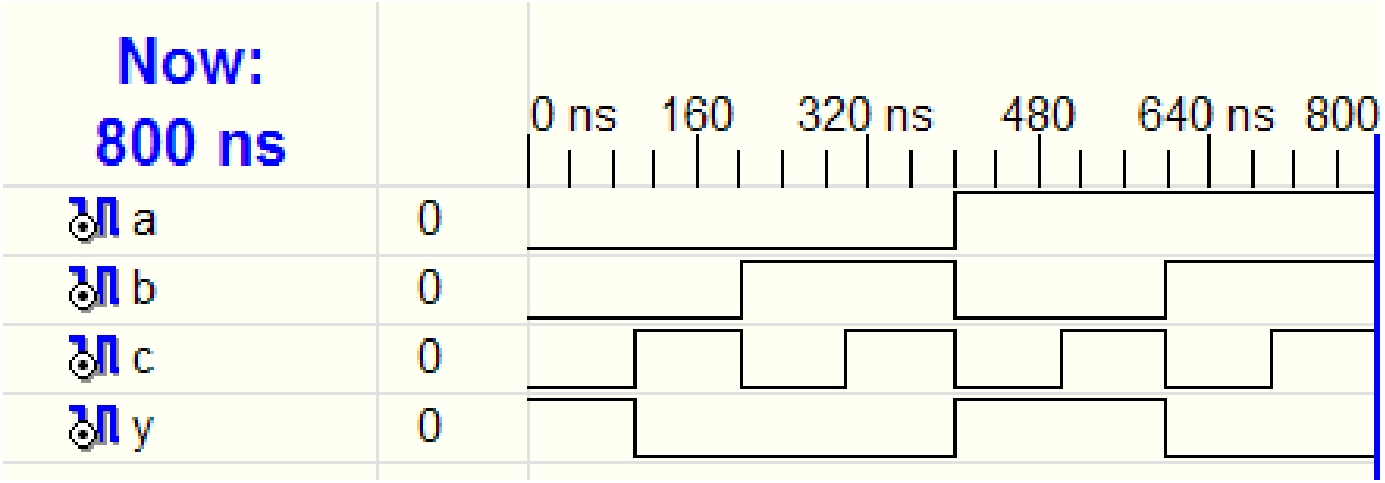
SystemVerilog:

```
module example(input  logic a, b, c,
               output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

HDL моделювання

SystemVerilog:

```
module example(input  logic a, b, c,
               output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;
endmodule
```

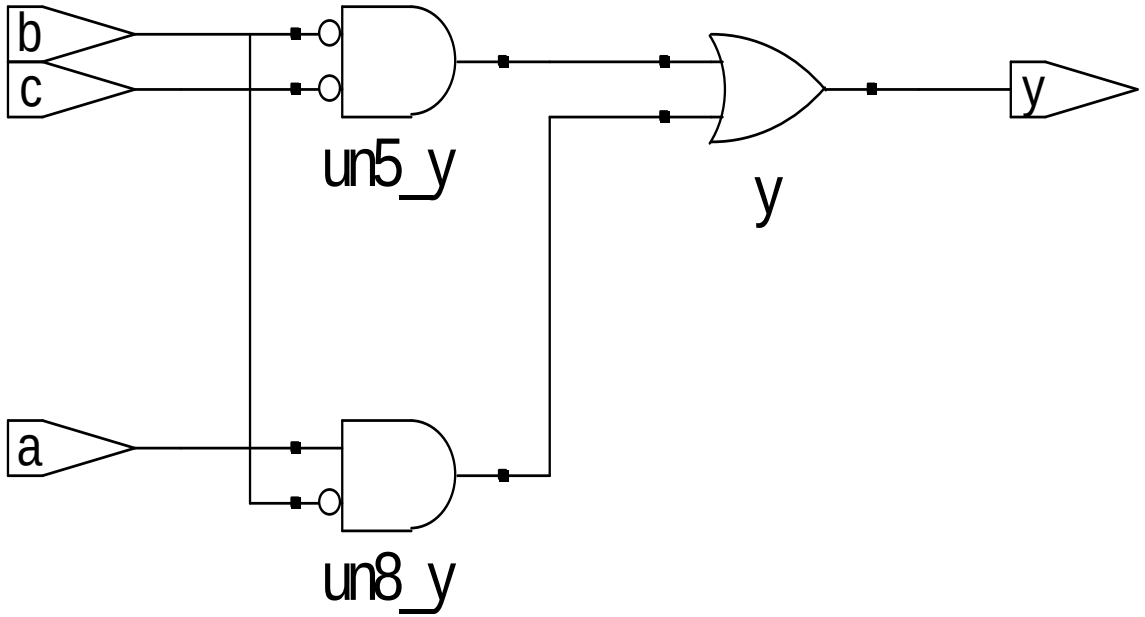


HDL синтез

SystemVerilog:

```
module example(input  logic a, b, c,
               output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

Синтез:



Синтаксис Verilog

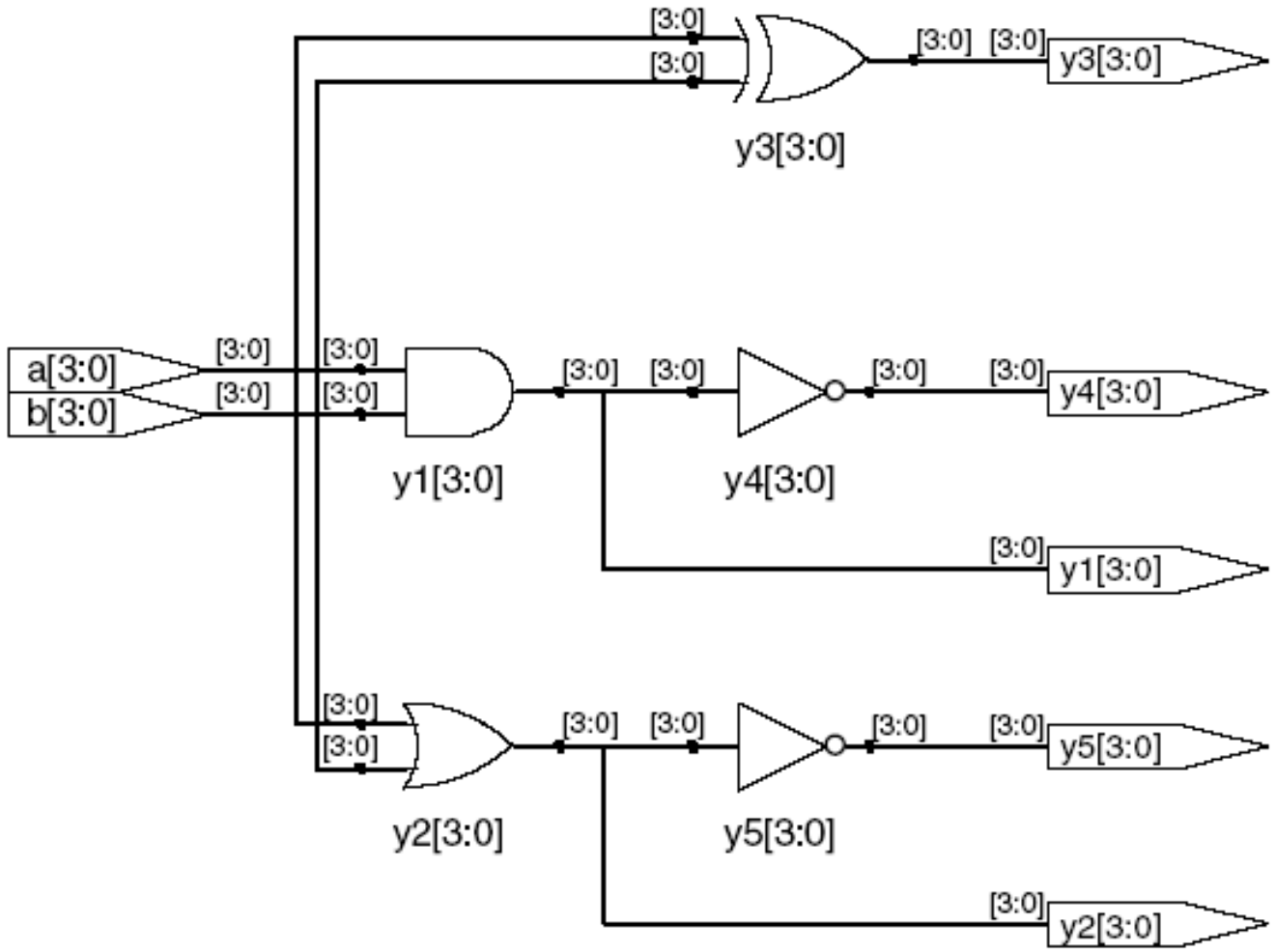
- Чутливий до регістру символів
  - Приклад: `reset` і `Reset` не одне і те ж.
- Імена не можуть починатися з цифри. Приклад: `2mux` – некоректне ім’я
- Пропуски ігноруються
- Коментарі:
  - `// однорядковий`
  - `/* багаторядковий`
    `коментар */`

Синтез структурних модулів

```
module and3(input  logic a, b, c,
            output logic y);
  assign y = a & b & c;
endmodule

module inv(input  logic a,
           output logic y);
  assign y = ~a;
endmodule

module nand3(input  logic a, b, c
             output logic y);
  logic n1;                      // внутрішній сигнал

  and3 andgate(a, b, c, n1);  // екземпляр and3
  inv  inverter(n1, y);       // екземпляр inverter
endmodule
```

Порозрядні операції

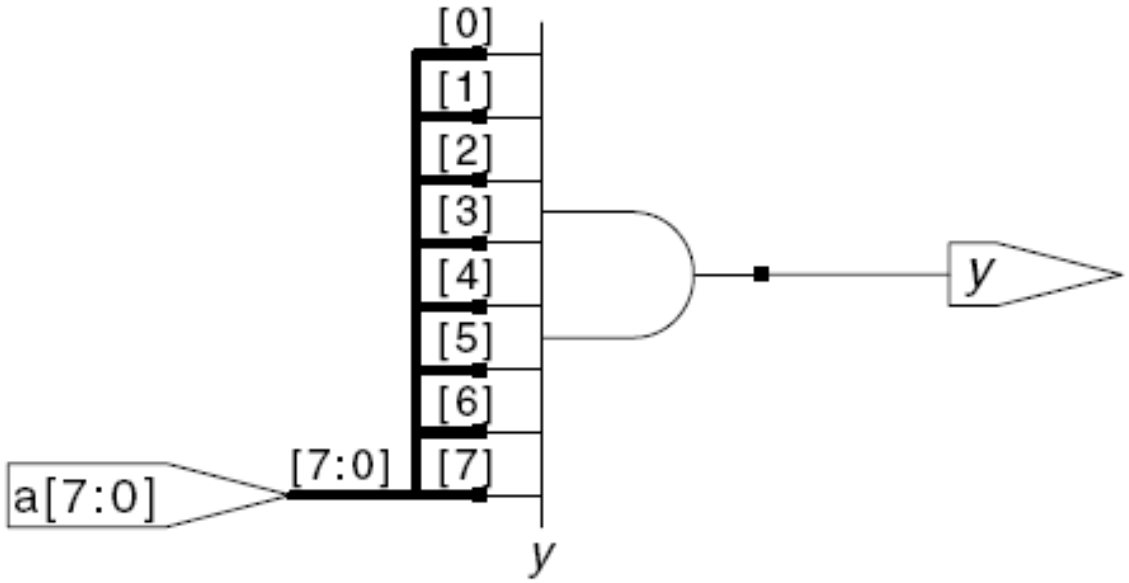
```
module gates(input  logic [3:0]  a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
   /* Чотири різних 2-входових логічних елементів
   під'єднаних до 4-розрядних шин */
   assign y1 = a & b;    // AND
   assign y2 = a | b;    // OR
   assign y3 = a ^ b;    // XOR
   assign y4 = ~(a & b); // NAND
   assign y5 = ~(a | b); // NOR
endmodule
```



~ операція інверсії

Операції скорочення

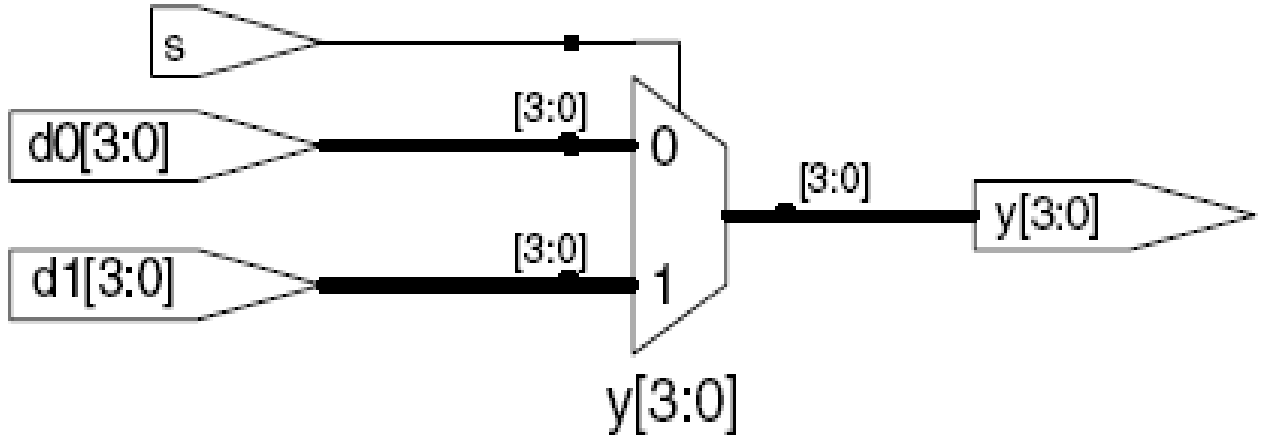
```
module and8(input  logic [7:0] a,
            output logic       y);
   assign y = &a;
   // &a легше записати ніж
   // assign y = a[7] & a[6] & a[5] & a[4] &
   //            a[3] & a[2] & a[1] & a[0];
endmodule
```



Умовне присвоєння

Модуль 2-входового мультиплексора:

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);
   assign y = s ? d1 : d0;
endmodule
```

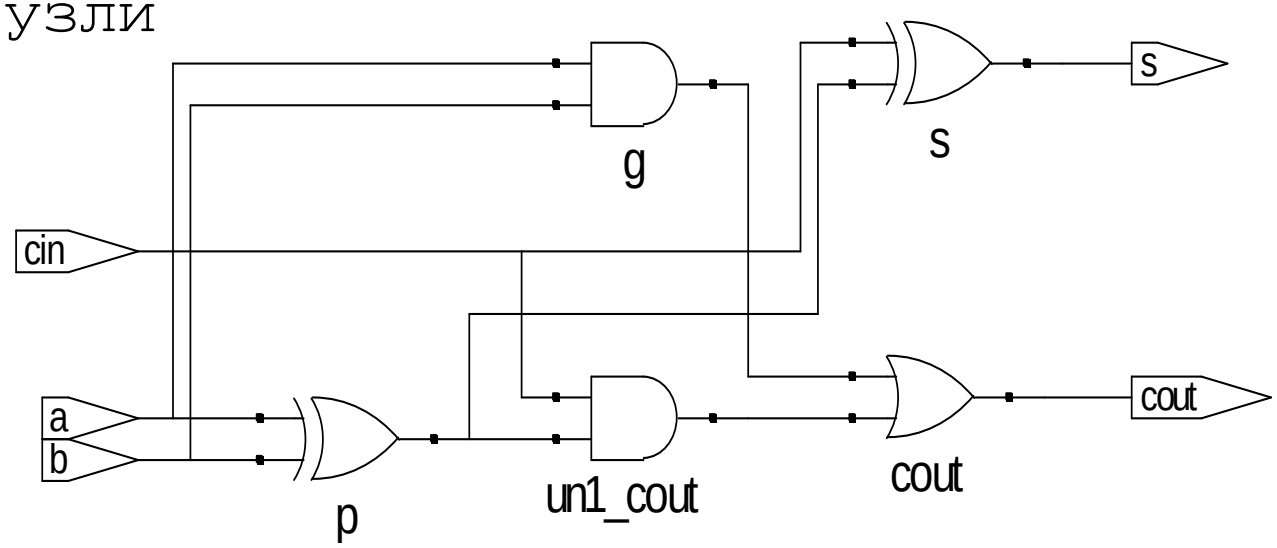


?: також називається тернарним оператором тому, що має 3 входи: `s`, `d1` і `d0`.

Внутрішні сигнали

Повний 1-розрядний суматор:

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;   // внутріщні вузли

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```



Пріоритет операцій

Порядок операцій

Вищий

~	Заперечення
*, /, %	Множення, ділення, залишок
+, -	Додавання, віднімання
<<, >>	Зсув
<<<, >>>	Арифметичний зсув
<, <=, >, >=	Порівняння (більше-менше)
==, !=	Порівняння на рівність
&, ~&	І, І-НЕ
^, ~^	Виключальне АБО, виключальне АБО-НЕ
|, ~|	АБО, АБО-НЕ
?:	Тернарний оператор

Нищий

Форми подання чисел

Формат: N'B value

N = кількість розрядів, **B** = основа

N'B не є обов'язковим, але рекомендується (за замовчуванням використовується десяткова система)

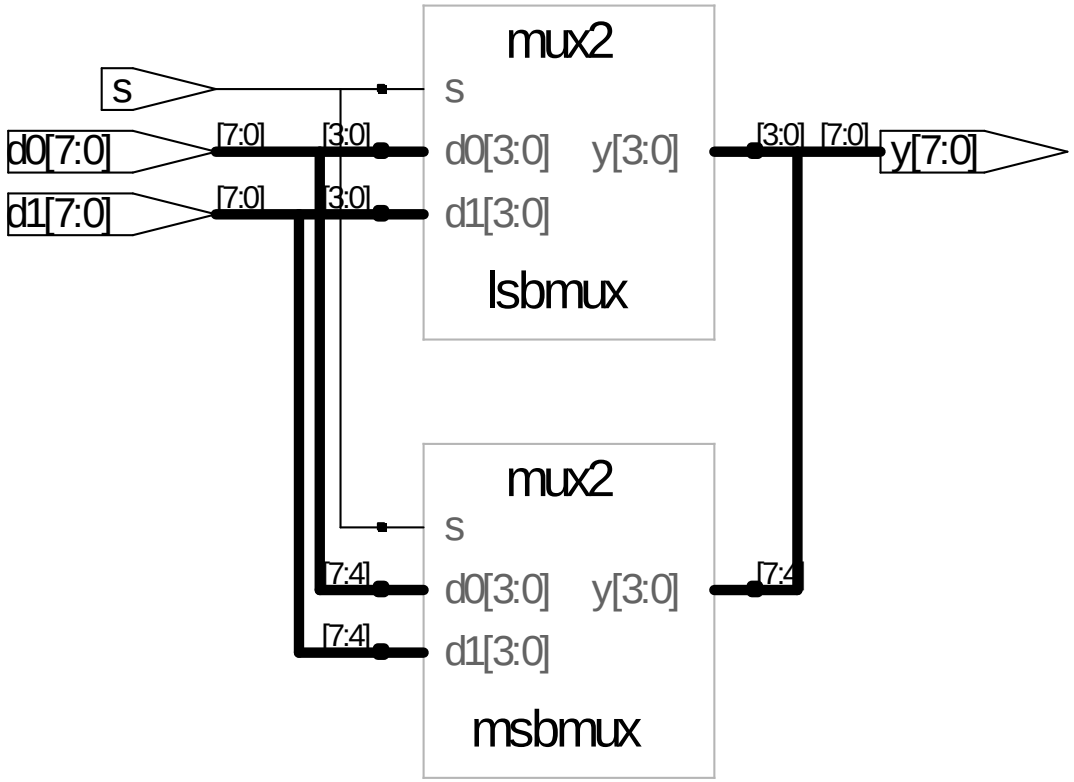
Число	Кількість розрядів	Основа	Десятковий еквівалент	Число в пам'яті
3'b101	3	Двійкова	5	101
'b11	Не визначено	Двійкова	3	00…0011
8'b11	8	Двійкова	3	00000011
8'b1010_1011	8	Двійкова	171	10101011
3'd6	3	Десяткова	6	110
6'o42	6	Вісімкова	34	100010
8'hAB	8	Шістнадцяткова	171	10101011
42	Не визначено	Десяткова	42	00…0101010

Робота з бітами

```
assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100_010};

// якщо y - 12-бітовий сигнал, то оператор вище сформує:
y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0

// подкреслення (_) використовується тільки для
// зручності читання. SystemVerilog його ігнорує.
```

Робота з бітами. Приклад 2

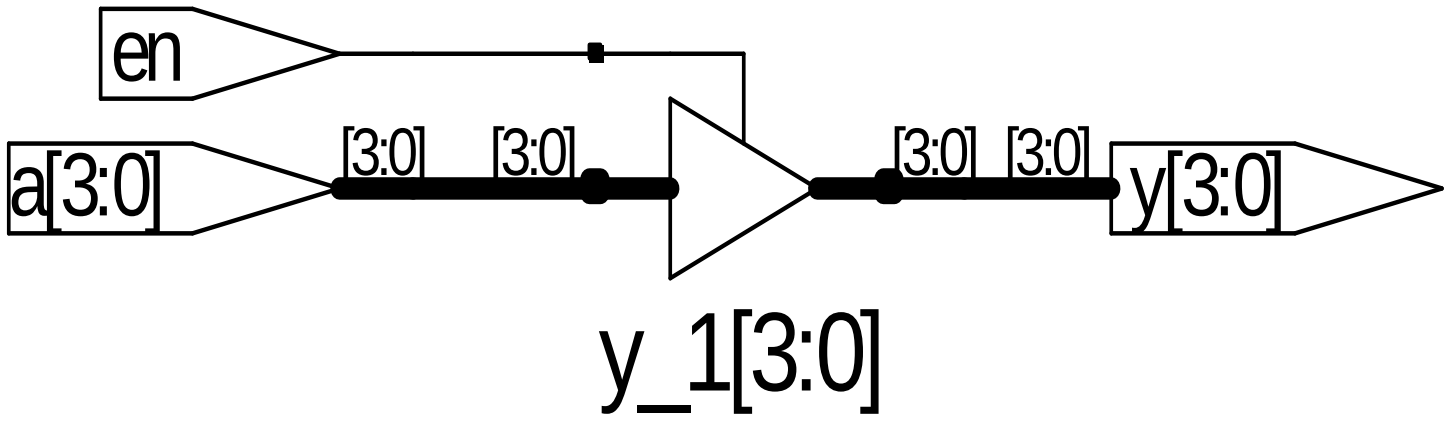
```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```



Непідключений високоімпедансний стан

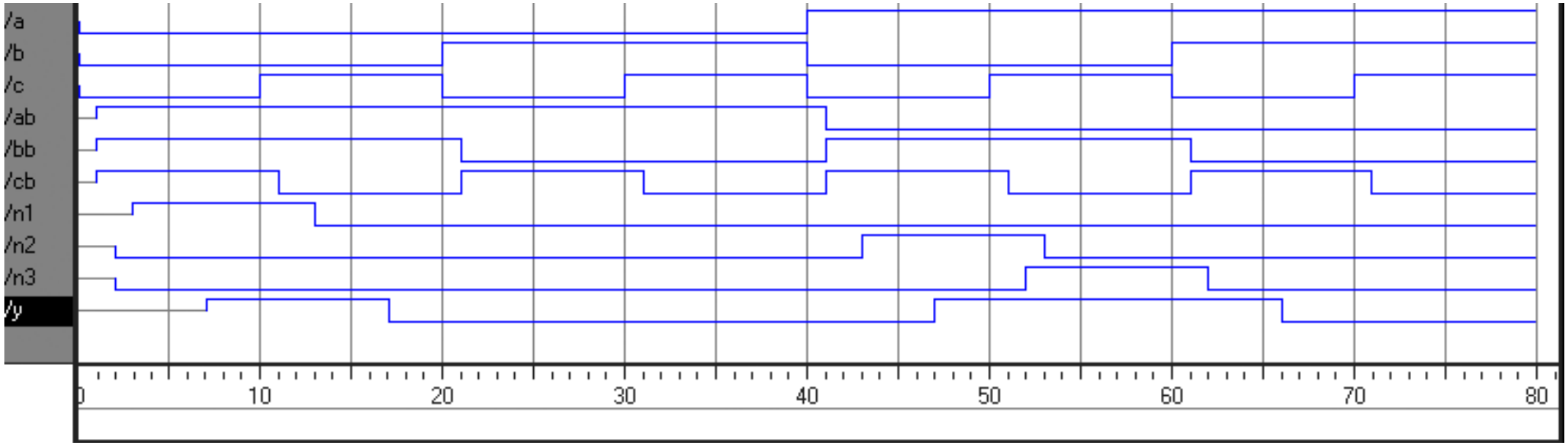
SystemVerilog:

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output logic [3:0] y);
   assign y = en ? a : 4'bz;
endmodule
```



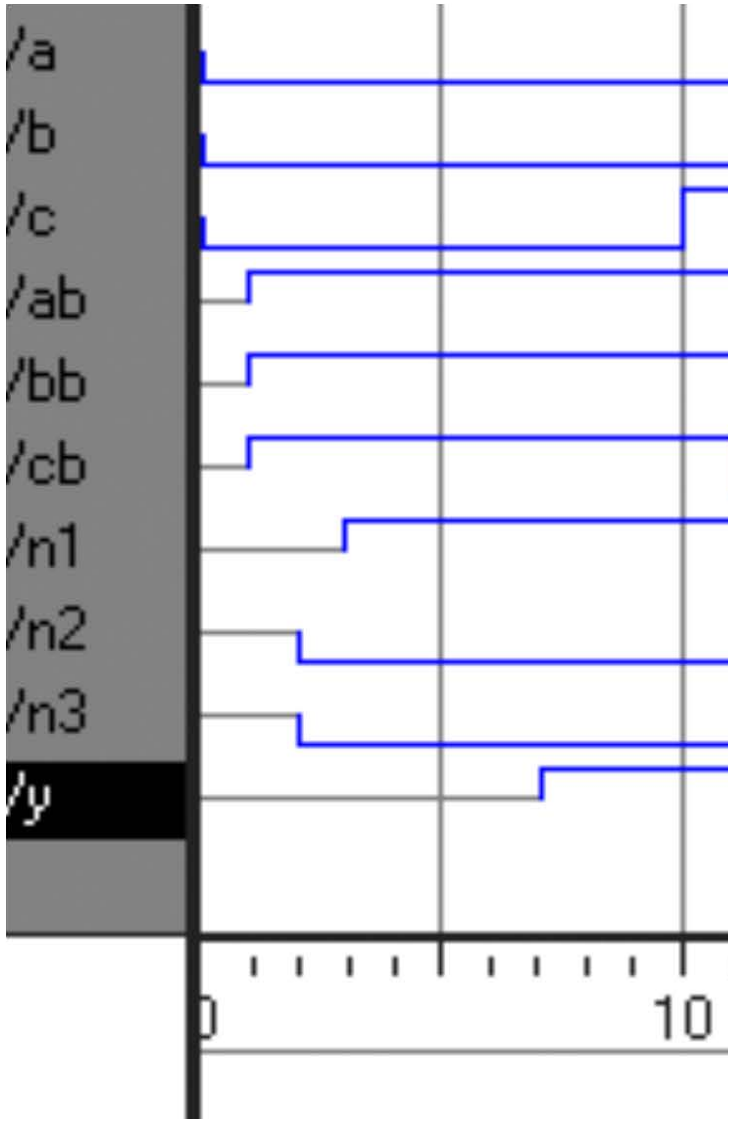
Затримки

```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



Затримки

```
module example(input  logic a,
  b, c,
                output logic y);
   logic ab, bb, cb, n1, n2, n3;
   assign #1 {ab, bb, cb} =
                    ~{a, b, c};
   assign #2 n1 = ab & bb & cb;
   assign #2 n2 = a & bb & cb;
   assign #2 n3 = a & bb & c;
   assign #4 y = n1 | n2 | n3;
endmodule
```



Послідовнісна логіка

- SystemVerilog використовує **ідіоми** для описання заклацок, тригерів і скінченних автоматів

- Довільні стилі HDL кодування можуть моделюватися правильно, але результат синтезу може не відповідати ні результатам моделювання, ні бажанням розробника

Оператор always

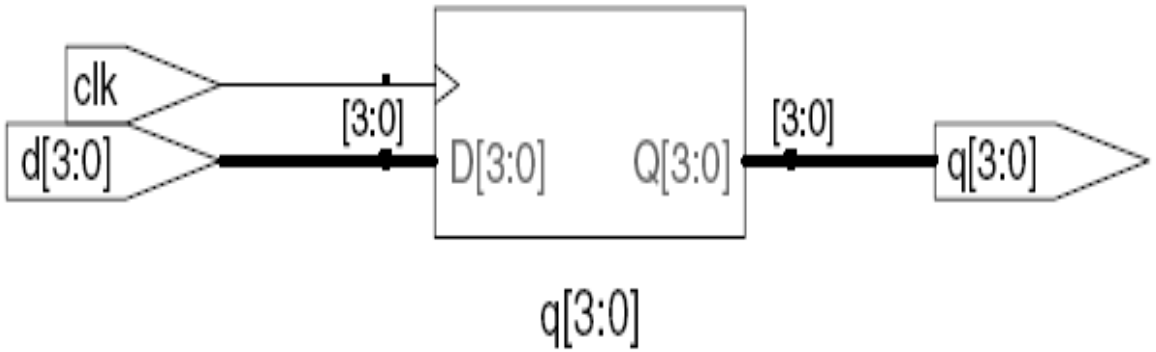
Загальна структура:

```
always @(sensitivity list)
  statement;
```

Кожний раз, коли відбувається подія із списку `sensitivity list`, виконується оператор `statement`

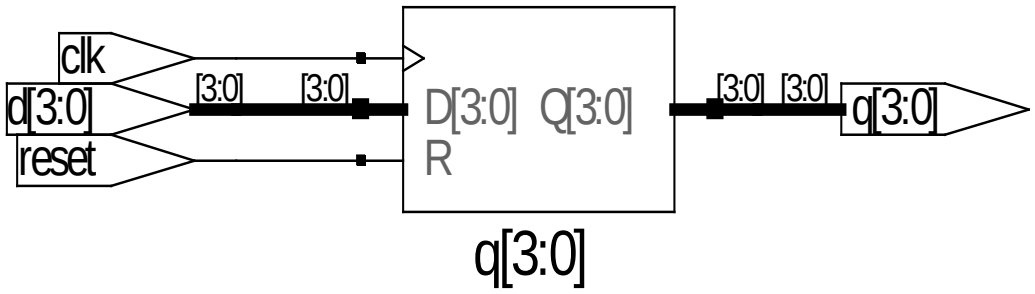
D-тригер

```
module flop(input  logic       clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff @(posedge clk)
    q <= d;    //вимовляється "q отримує d"

endmodule
```



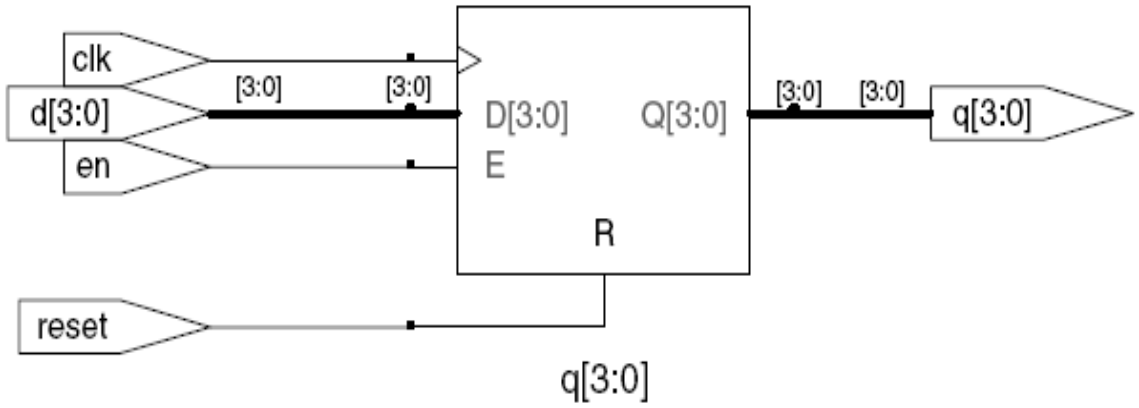
D-тригер із сигналом скидання

```
module flopr(input  logic        clk,
             input  logic        reset,
             input  logic [3:0]  d,
             output logic [3:0]  q);

  // синхронне скидання
  always_ff @(posedge clk)
    if (reset) q <= 4'b0;
    else       q <= d;

endmodule
```



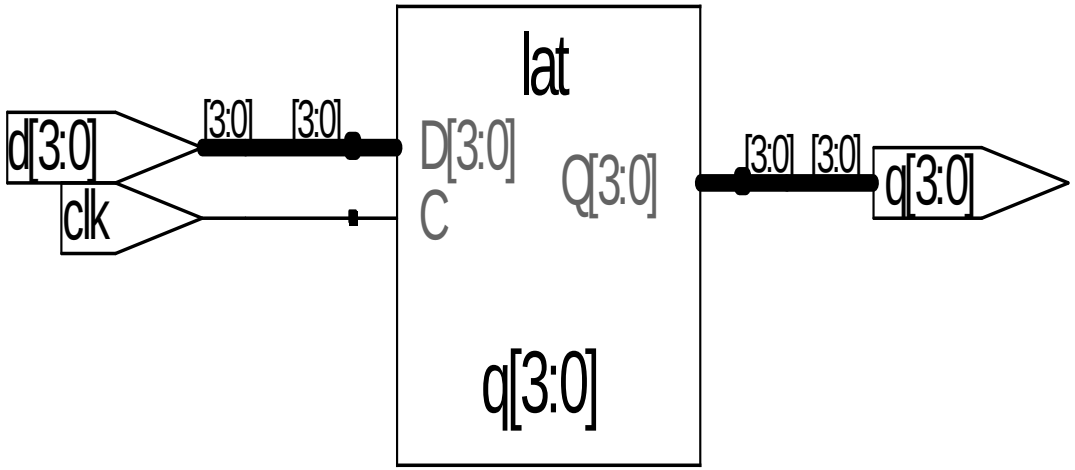
D-тригер із сигналом дозволу

```
module flopren(input  logic       clk,
               input  logic       reset,
               input  logic       en,
               input  logic [3:0] d,
               output logic [3:0] q);

  // асинхроне скидання і дозвіл
  always_ff @(posedge clk, posedge reset)
    if      (reset) q <= 4'b0;
    else if (en)    q <= d;

endmodule
```



Заклацки

```
module latch(input  logic       clk,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_latch
    if (clk) q <= d;

endmodule
```



Увага! В даному курсі не використовуються заклацки.
Але при написанні коду можна ненавмисно реалізувати заклацку.
Перевірити синтезований апаратний модуль – якщо він має заклацку то в коді допущена помилка.

Інші поведінкові оператори

- Оператори, які повинні бути розміщені всередині оператору `always` :
  - `if` / `else`
  - `case`, `casez`

Комбінаційна логіка з оператором always

```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
  always_comb
  // потрібні begin/end так як є більш ніж один вираp в операторі always
   begin
      y1 = a & b;    // AND
      y2 = a | b;    // OR
      y3 = a ^ b;    // XOR
      y4 = ~(a & b); // NAND
      y5 = ~(a | b); // NOR
    end
endmodule
```

Цей апаратний модуль може бути описаний за допомогою оператора неперервного присвоєння `assign` з меншою кількістю рядків коду.

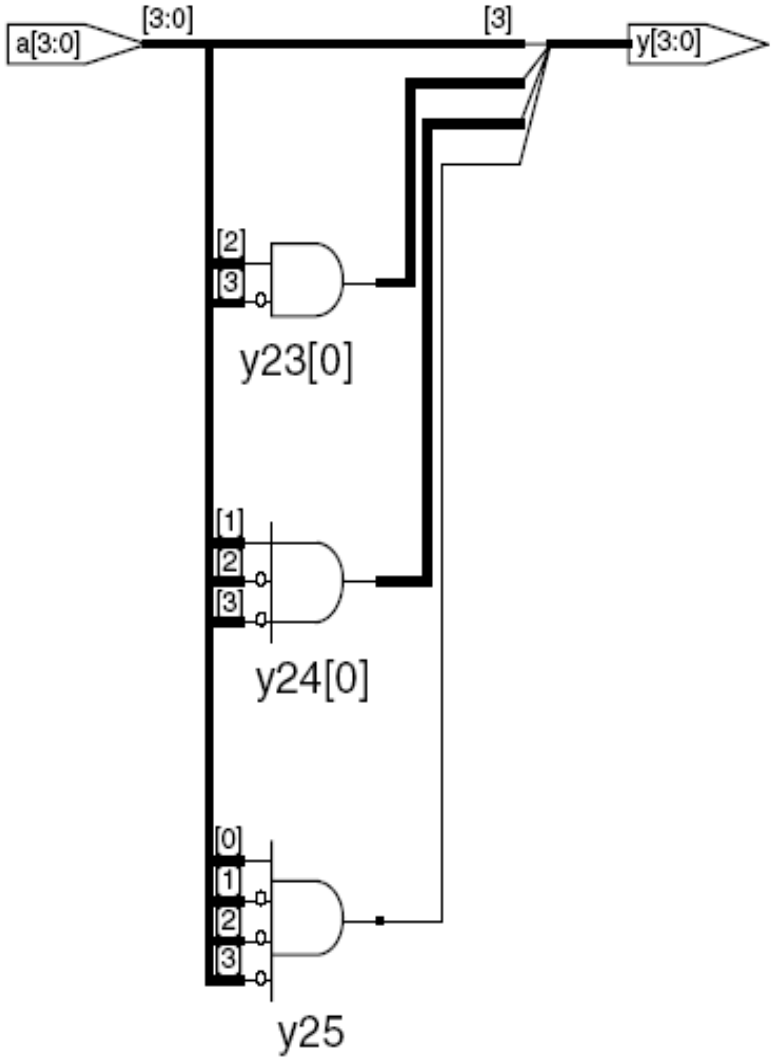
Комбінаційна логіка з case

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);
  always_comb
    case (data)
      //                    abc_defg
      0: segments =       7'b111_1110;
      1: segments =       7'b011_0000;
      2: segments =       7'b110_1101;
      3: segments =       7'b111_1001;
      4: segments =       7'b011_0011;
      5: segments =       7'b101_1011;
      6: segments =       7'b101_1111;
      7: segments =       7'b111_0000;
      8: segments =       7'b111_1111;
      9: segments =       7'b111_0011;
      default: segments = 7'b000_0000; // потрібно
    endcase
endmodule
```

- Оператор `Case` реалізує комбінаційну логіку, **тільки якщо у** всіх гілках перераховані всі можливі вхідні комбінації
- **`default`** (вибір за замовчуванням)

Комбінаційна логіка з casez

```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);
  always_comb
    casez(a)
      4'b1???: y = 4'b1000;
  // ? = не визначений стан
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
   endcase
endmodule
```



Блокуючі і неблокуючі присвоєння

<= Неблокуюче присвоєння

- виконується одночасно з іншими

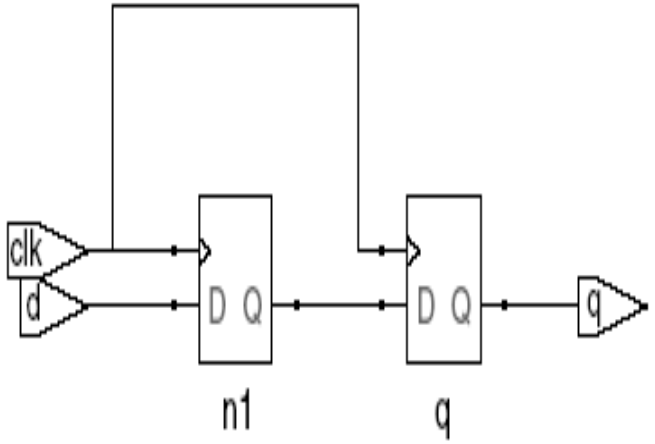
= Блокуюче присвоєння

- виконується у порядку, описаному у файлі

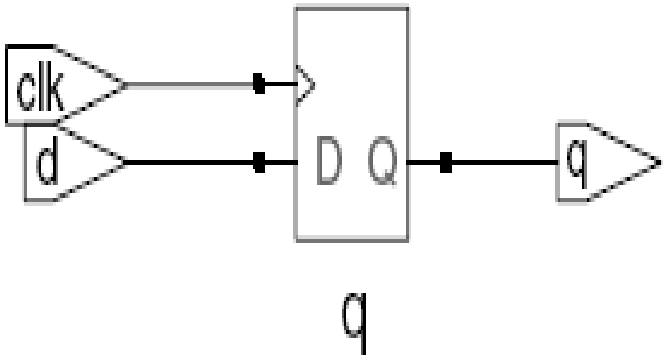
```
// A - добрий синхронізатор з
   неблокуючим привоєнням

module syncgood(input  logic
   clk,
                input  logic d,
                output logic
   q);
  logic n1;
  always_ff @(posedge clk)
    begin
      n1 <= d;  // nonblocking
      q  <= n1; // nonblocking
    end
endmodule
```

```
// Б - поганий синхронізатор з
// блокуючим присвоєнням

module syncbad(input logic  clk,
               input  logic d,
               output logic q);
  logic n1;
  always_ff @(posedge clk)
    begin
      n1 = d;  // blocking
      q  = n1; // blocking
    end
endmodule
```

А



Б



Правила присвоєння сигналів

- **Синхронна послідовнісна логіка:** використовує **always_ff @(posedge clk)** і неблокуюче присвоєння (**<=**)

```
always_ff @ (posedge clk)
  q <= d; // nonblocking
```

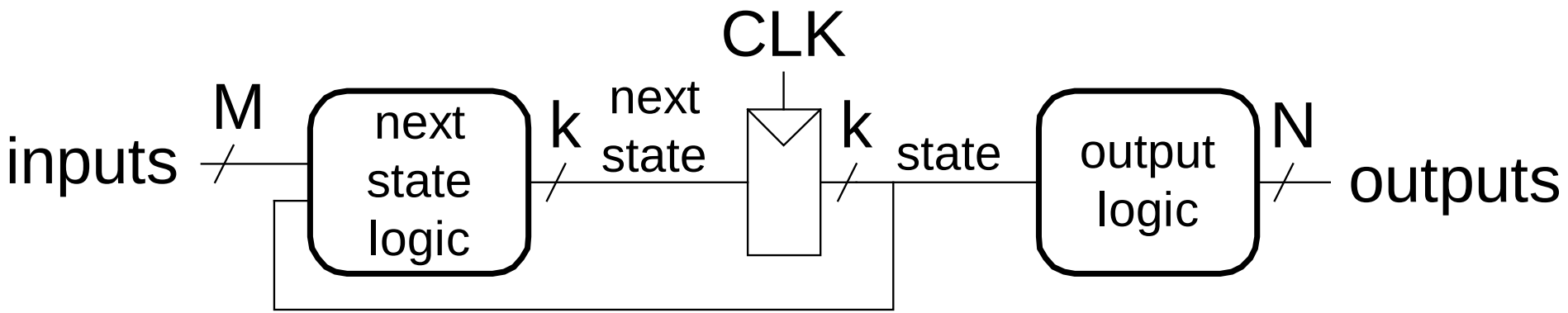
- **Проста комбінаційна логіка:** використовує неперевне присвоєння (**assign…**)

```
assign y = a & b;
```

- **Більш складна комбінаційна логіка:** використовує **always_comb** і блокуюче присвоєння (**=**)
- Сигнал змінюється **тільки одним** оператором `always` або оператором неперервного присвоєння (спроба змінити сигнал декількома операторами `always` або `assign` без використання відключеного стану спричинить конфлікт і помилку синтезу).

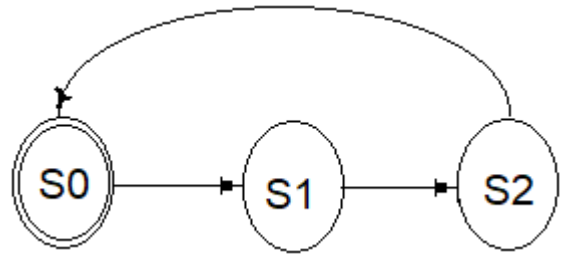
Скінченний автомат

- **Три блоки:**
  - Логіка наступного стану
  - Регістр станів
  - Логіка виходів



Приклад скінченного автомату на SystemVerilog: Дільник на 3

```
module divideby3FSM (input  logic clk,
                     input  logic reset,
                     output logic q);
   typedef enum logic [1:0] {S0, S1, S2} statetype;
   statetype [1:0] state, nextstate;
    // регістр станів
   always_ff @ (posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;
   // логіка наступного стану
   always_comb
      case (state)
         S0:      nextstate = S1;
         S1:      nextstate = S2;
         S2:      nextstate = S0;
         default: nextstate = S0;
      endcase
```




```
// логіка вихідних сигналів
   assign q = (state == S0);
endmodule
```

Параметризовані модулі

2:1 мультиплексор:

```
module mux2
  #(parameter width = 8)  // ім’я і значення за замовчуванням
   (input  logic [width-1:0] d0, d1,
    input  logic             s,
    output logic [width-1:0] y);
  assign y = s ? d1 : d0;
endmodule
```

Приклад з 8-бітною шиною (використовується за замовчуванням):

```
  mux2 mux1(d0, d1, s, out);
```

Приклад з 12-бітною шиною :

```
 mux2 #(12) lowmux(d0, d1, s, out);
```

Середовище тестування (TestBenches)

- HDL модуль, який тестує інший модуль: тестований пристрій (device under test - DUT)
- TestBenches **не** призначений для синтезу
- Типи тестування:
  - просте
  - з самоперевіркою
  - з самоперевіркою і тестовими векторами

Приклад середовища тестування

- Написати System Verilog код для апаратної реалізації наступних функцій:

$$y = a'b'c' + ab'c' + ab'c'$$

```
module sillyfunction(input  logic a, b, c,
                     output logic y);
  assign y = ~a & ~b & ~c |
             a & ~b & ~c |
             a & ~b & c;
endmodule
```

Просте середовище тестування

```
module testbench1();
  logic a, b, c;
  logic y;
  // екземпляр перевірюваного пристрою
  sillyfunction dut(a, b, c, y);
  // послідовно формуються значення сигналів на входах
initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
    a = 1; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
  end
endmodule
```

Середовище тестування з самоперевіркою

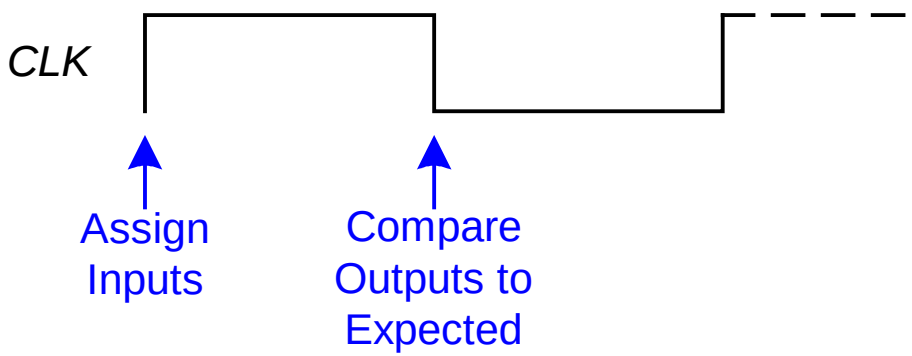
```
module testbench2();
  logic  a, b, c;
  logic y;
// створення екземпляру dut
  sillyfunction dut(a, b, c, y);
// послідовно беруться входи і
  тестуються виходи
  initial); begin
  a = 0; b = 0; c = 0; #10;
    if (y !== 1) $display("000
  failed.");
    c = 1; #10;
    if (y !== 0) $display("001
  failed.");
    b = 1; c = 0; #10;
 if (y !== 0) $display("010 failed.");
    c = 1; #10;
if (y !== 0) $display("011 failed.");
    a = 1; b = 0; c = 0; #10;
    if (y !== 1) $display("100 failed.");
    c = 1; #10;
    if (y !== 1) $display("101 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("110 failed.");
    c = 1; #10;
    if (y !== 0) $display("111 failed.");
  end
endmodule
```

Середовище тестування з тестовими векторами

- Файл тестових векторів: вхідні сигнали і очікувані стани виходів
- Середовище тестування:
  1. Формування тактового сигналу для зміни входів, зчитування вихідних сигналів
  2. Зчитування тестових векторів з файлу в буферний масив для наступної подачі їх на входи
  3. Присвоєння значень вхідним сигналам, визначення очікуваних значень вихідних сигналів
  4. Порівняння реальних значень вихідних сигналів з очікуваними і формування списку помилок

Середовище тестування з тестовими векторами

- Середовище тестування, тактовий сигнал:
  - Зміна вхідних сигналів за переднім фронтом тактового сигналу
  - Порівняння станів виходів з очікуваними значеннями за заднім фронтом тактового сигналу



- Тактовий сигнал середовища тестування також використовується для синхронізації послідовнісних схем

Файли тестових векторів

- Файл: `example.tv`
- Містить вектори abc_yexpected

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

1. Генерація тактового сигналу

```
module testbench3();
  logic        clk, reset;
  logic        a, b, c, yexpected;
  logic        y;
  logic [31:0] vectornum, errors;    // bookkeeping
 variables
  logic [3:0]  testvectors[10000:0]; // array of
 testvectors


  // створення екземпляру тестованого пристрою
  sillyfunction dut(a, b, c, y);


  // генерація тактового сигналу
  always     // без списку чутливості нескінченний цикл
    begin
      clk = 1; #5; clk = 0; #5;
    end
```

2. Зчитування тестових векторів у масив

```
// при запуску тесту завантажуються вектори і генерується сигнал скидання
initial
    begin
      $readmemb("example.tv", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #27; reset = 0;
    end


// Зауваження: $readmemb зчитує файл тестових векторів,
// записаних у шістнадцятковому поданні
```

3. Призначення входів і очікувані стани виходів

```
// подача тестових векторів за переднім фронтом
// синхросигналу
always @(posedge clk)
  begin
    #1; {a, b, c, yexpected} = testvectors[vectornum];
  end
```

3. Порівняння вихідних сигналів з очікуваними

```
// перевірка результатів за заднім фронтом синхросигналу
   always @(negedge clk)
    if (~reset) begin // skip during reset
      if (y !== yexpected) begin
        $display("Error: inputs = %b", {a, b, c});
        $display("  outputs = %b (%b expected)",y,yexpected);
        errors = errors + 1;
      end

// Зауваження: для виведення на друк в шістнадцятковому коді
// (hexadecimal), використовується %h. Наприклад,
//       $display(”Error: inputs = %h”, {a, b, c});
```

4. Порівняння вихідних сигналів з очікуваними

```
// інкремент індексу масиву і зчитування чергового тестового вектору
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 4'bx) begin
          $display("%d tests completed with %d errors",
                vectornum, errors);
        $finish;
      end
    end
endmodule


// === and !== може порівнювати значення 1, 0, x, or z.
```