

2.1. Khái niệm trong mô phỏng mạch số (Mô phỏng dựa trên sự kiện; logic 4 giá trị; thời gian trong mô phỏng) – 1 LT

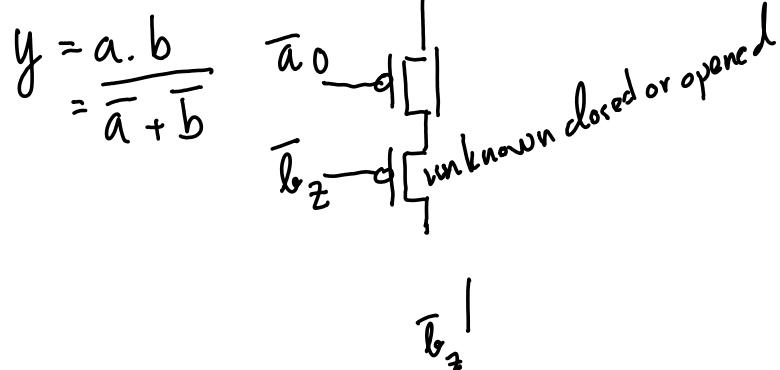
2.1.1. Định nghĩa mô phỏng mạch số: Là quá trình phân tích mô hình mô tả mạch để tính toán giá trị đầu ra từ đầu vào

2.1.2. Nguyên tắc mô phỏng mạch số

2.1.4. Logic 4 trạng thái trong mô phỏng

Các tín hiệu trong mạch được mô phỏng có thể nhận 4 giá trị: 0, 1, x(unknown), z (high-impedance) và các phần tử trong mạch sẽ được tính toán dựa trên bảng sự thật mở rộng cho logic 4 trạng thái

a	y=NOT(a)
0	1
1	0
x	x
z	x



Bài tập ví dụ: Viết module test các phép toán logic 4 giá trị trong Verilog

Chữa:

```
module test_logical_operators;
```

```
reg a,b;
// Cách 1: Dùng hàm $monitor và gán giá trị các biến
initial $monitor("%b & %b = %b",a, b, a&b);
initial $monitor("%b | %b = %b",a, b, a|b);
initial
begin
    a=0;b=0#5;
    a=0;b=1'b1#5;
    a=0;b=1'bx#5;
    a=0;b=1'bz#5;
    ....
end
```

// Cách 2: Dùng hàm \$display và các phép toán trực tiếp các hằng số

```
$display("%b & %b = %b",1'b0,1'b0,1'b0 & 1'b0);
$display("%b & %b = %b",1'b0,1'b1,1'b0 & 1'b1);
$display("%b & %b = %b",1'b0,1'bx,1'b0 & 1'bx);

endmodule
```

2.2. Testbench (Khái niệm; Chức năng; Các thành phần cơ bản của testbench; Phương pháp tạo kích thích đầu vào; Phương pháp kiểm tra đầu ra) – 1LT

2.2.1. Khái niệm và chức năng của testbench

- Testbench là mô hình mô trường hoạt động của vi mạch được sử dụng để kiểm tra thiết kế mạch bằng phương pháp mô phỏng
- Testbench có 3 chức năng chủ yếu
 - **Tạo ra các giá trị đầu vào** (input stimulus, input patterns) và đưa các tín hiệu đầu vào tới cổng vào của thiết kế (theo đúng kịch bản hoạt động, theo đúng giao thức vào yêu cầu). Ví dụ: testbench sẽ tạo ra các chuỗi lệnh được mã hóa đúng theo MIPS ISA và đưa vào đầu vào Instr của thiết kế khi được yêu cầu.
 - **Quan sát giá trị đầu ra, tự động tạo ra giá trị đầu ra đúng và kiểm tra giá trị đầu ra của thiết kế** (so sánh với giá trị đầu ra đúng)
 - **Giám sát quá trình mô phỏng và đo lường mức độ hoàn thành** (chất lượng của quá trình mô phỏng)

2.2.2. Các thành phần cơ bản của testbench

Testbench tối thiểu sẽ gồm các thành phần cơ bản sau:

- 1) Tạo ra thiết kế cần được kiểm tra (Design Under Test, Design Under Verification) và kết nối với các biến được khai báo bên trong testbench
- 2) Quan sát và kiểm tra các giá trị đầu ra
- 3) Tạo các giá trị đầu vào
- 4) [Tùy chọn]-Đo lường mức độ hoàn thành của quá trình mô phỏng

```
module full_adder_testbench; // khai bao module testbench khong co dau vao/dau ra
```

```
reg a_sim, b_sim, ci_sim; // khai báo các biến sử dụng trong testbench
wire s_sim, co_sim; // khai báo các biến sử dụng trong testbench
```

```
// tạo ra một thành phần có kiểu là thiết kế full_adder_structure tên là fa_duv trong
testbench
```

```
// module instantiation
```

```
// nối tín hiệu tên a_sim, b_sim, ci_sim, s_sim, co_sim vào các đầu vào a, b, ci, s, co tương
ứng của thiết kế
full_adder_structure fa_duv(.a(a_sim), .b(b_sim), .ci(ci_sim), .s(s_sim), .co(co_sim));
```

```
// đưa ra giá trị đầu ra để kiểm tra
```

```
initial $monitor("%t: a=%b,b=%b,ci=%b,s=%b,co=%b",$time,
a_sim,b_sim,ci_sim,s_sim,co_sim);
```

```
// tạo ra mẫu giá trị đầu vào
```

```
initial
```

```
begin
```

```
// sau 5 đơn vị thời gian mô phỏng thì thay đổi 1 mẫu giá trị đầu vào
#5 a_sim = 0; b_sim = 0; ci_sim = 0;
#5 a_sim = 1;
```

```
end
```

```
endmodule // full_adder_testbench
```

2.2.3. Các phương pháp tạo kích thích đầu vào (mẫu đầu vào)

- a) Tạo ra giá trị đầu vào trực tiếp (Direct test): Tạo ra một vài giá trị tổ hợp đầu vào và đưa vào mạch.
- Dễ hiểu,

- Dễ tạo ra giá trị đầu vào, nhanh tạo được testbench
- Có khả năng không phát hiện được lỗi ở góc (corner cases)
- Nhược điểm:
 - Không bao quát hết các trường hợp (miss corner cases)
 - Không mô phỏng được toàn bộ các tổ hợp đầu vào cho mạch lớn
 - Khó duy trì testbench

b) Tạo ra tất cả các tổ hợp giá trị đầu vào (Full test-Complete test/simulation)

- Sử dụng vòng **for** trong Verilog.
- Chậm
- Không khả thi
- Sử dụng cho các mạch nhỏ
- Đảm bảo chắc chắn phát hiện được các lỗi của mạch

c) Tạo ra các tổ hợp giá trị đầu vào ngẫu nhiên (Random test)

Sử dụng hàm hệ thống **\$random** trong Verilog

- Dễ tạo ra testbench
 - Có thể tạo ra các tín hiệu đầu vào không hợp lệ
 - Có khả năng không phát hiện được lỗi ở góc
- d) Kết hợp phương pháp đầu vào ngẫu nhiên và đầu vào trực tiếp cùng với quá trình đo lường chất lượng mô phỏng

B1: Mô phỏng ngẫu nhiên mạch

B2: Đo chất lượng mô phỏng

B3: Phân tích các kịch bản hoạt động của mạch để tạo ra các trường hợp đầu vào trực tiếp tương ứng với các kịch bản hoạt động của mạch để đạt được chất lượng mô phỏng tốt hơn

2.2.4. Các phương pháp quan sát và kiểm tra giá trị đầu ra

a) Phương pháp thủ công

- Giá trị tín hiệu đầu vào và đầu ra được hiển thị dưới dạng đồ thị (waveform) hoặc dạng text và được so sánh đối chiếu thủ công bởi kỹ sư kiểm tra
- Không mất thời gian công sức để xây dựng testbench
- Mất nhiều công sức để so sánh đối chiếu
- Chỉ có thể áp dụng cho số lượng rất ít tổ hợp giá trị đầu vào
- Chỉ áp dụng để gỡ lỗi
- Cần được thay thế bằng phương pháp tự động và teschbench tự kiểm tra (self-test)

b) Phương pháp tự động

- Giá trị tín hiệu đầu ra chuẩn được tính toán bởi testbench và sẽ được so sánh với đầu ra của thiết kế. Khi có sai khác, testbench sẽ cảnh báo kỹ sư kiểm tra để gỡ lỗi

- Mất nhiều công sức xây dựng testbench
- Quá trình kiểm tra sẽ được thực hiện tự động
- Giá trị đầu ra chuẩn có thể được tính toán bằng

- C1: Tạo ra một mô hình khác của thiết kế bằng ngôn ngữ Verilog ở testbench. Mô hình này ở mức độ trừu tượng cao hơn thiết kế và không cần tối ưu như thiết kế để làm mô hình chuẩn (golden model). Mô hình chuẩn và thiết kế được

- cùng mô phỏng và đầu ra sẽ được so sánh với nhau

- C2: Mô hình phần mềm của thiết kế (thường được tạo ra ở bước Architecture Design) sẽ được sử dụng làm mô hình chuẩn

- Testbench sẽ giao tiếp với mô hình phần mềm chuẩn thông qua giao tiếp tập tin (đọc kết quả chuẩn được ghi bởi mô hình phần mềm) hoặc thông qua giao diện lập trình PLI của Verilog (gọi các hàm tín toán C/C++) hoặc thông qua giao tiếp Hardware In Loop (đồng mô phỏng các hàm Matlab/Simulink)

$$y = a \cdot b$$

2.2.5. Kiểm soát chất lượng mô phỏng

Chất lượng mô phỏng được đo lường/đánh giá thông qua tham số mức độ bao phủ (coverage; tính bằng %). Mục tiêu cơ bản của quá trình mô phỏng là đạt được mức độ bao phủ 100%. Mức độ bao phủ không đạt 100% có nghĩa là có những khu vực của thiết kế chưa được mô phỏng, và cần được mô phỏng bằng các tổ hợp đầu vào trực tiếp hoặc phải được giải thích bằng các giá trị đầu vào cấm. **Chú ý, mức độ bao phủ 100% không đảm bảo là thiết kế không có lỗi.**

Độ bao phủ chỉ có thể chỉ ra rằng mô phỏng chưa hoàn thành, chứ không chỉ ra mô phỏng đã hoàn thành.

Các loại độ bao phủ

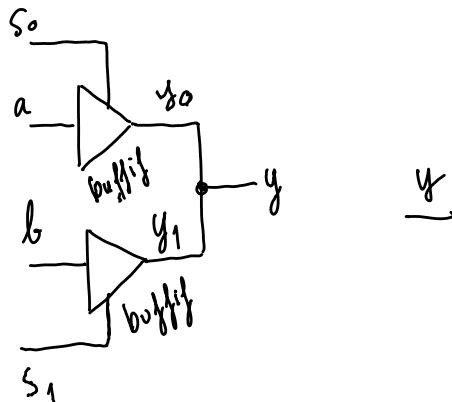
- 1) Độ bao phủ mã: Được tính toán dựa trên các thành phần trong mã thiết kế và được tính toán tự động bởi chương trình mô phỏng

CHƯƠNG 2: Kiểm tra chức năng IC số, hệ thống số (4 LT+1BT)
 2.1. Khái niệm trong mô phỏng mạch số (Mô phỏng dựa trên sự kiện; logic 4 giá trị; thời gian trong mô phỏng) – 1 LT
 2.1.4. Logic 4 trạng thái trong mô phỏng

Các tín hiệu trong mạch được mô phỏng có thể nhận 4 giá trị: 0, 1, x(unknown), z (high-impedance).

Ví dụ logic 4 trạng thái

s0	0	0	1	1	1	1
a	0,1,x,z	0,1,x,z	0	1	x	z
s1	0	1	1	1	1	1
b	0,1,x,z	0,1,x,z	0	0	0	0
y0	z	z	0	1	x	x
y1	z	0,1,x,x	0	0	0	0
y	z	0,1,x,x	0	x	x	x



2.2. Testbench (Khái niệm; Chức năng; Các thành phần cơ bản của testbench; Phương pháp tạo kích thích đầu vào; Phương pháp kiểm tra đầu ra) – 1LT

2.2.1. Khái niệm và chức năng testbench

- Testbench là mô hình môi trường hoạt động của thiết kế vi mạch.
- Testbench được kết nối với thiết kế và được mô phỏng để kiểm tra thiết kế
- Testbench nó có 3 chức năng chính:
 - Tạo tổ hợp các giá trị đầu vào (input patterns, input stimulus) và đưa giá trị đầu vào tới thiết kế theo đúng định dạng và thời gian yêu cầu.
 - Ví dụ: Khi cần kiểm tra thiết kế MIPS thực hiện đúng lệnh cộng, testbench cần tạo ra lệnh cộng dạng mã máy theo đúng quy định trong MIPS ISA và đưa lệnh đó vào đầu vào Instr của thiết kế tại thời điểm được yêu cầu.
 - Quan sát giá trị đầu ra, tự động tạo ra giá trị đầu ra đúng và kiểm tra giá trị đầu ra của thiết kế (so sánh với giá trị đầu ra đúng)
 - Giám sát quá trình mô phỏng và đo lường mức độ hoàn thành (chất lượng của quá trình mô phỏng)

2.2.2. Các thành phần cơ bản của testbench

- a) Khai báo biến được sử dụng trong testbench
 Các biến trong testbench được sử dụng để kết nối với cổng vào/ra của thiết kế hoặc sử dụng để kiểm soát quá trình đưa giá trị đầu vào tới thiết kế, kiểm soát quá trình mô phỏng

Biến trong Verilog được khai báo là **reg** hoặc **wire** theo cú pháp sau:

```
reg ten_bien;
wire ten_bien;
```

Các biến kiểu **reg** sẽ giữ nguyên giá trị cho đến khi nào nó được gán giá trị mới. Các biến kiểu reg sẽ được sử dụng trong các câu lệnh thủ tục nằm bên trong các khối lệnh **initial** và **always**.

Các biến kiểu **wire** dùng để mô tả dây dẫn trong thiết kế, nó sẽ thay đổi giá trị khi biến điều khiển nó thay đổi giá trị. Các biến kiểu wire được sử dụng trong các câu lệnh gán liên tục **assign** hoặc để kết nối các thành phần mạch.

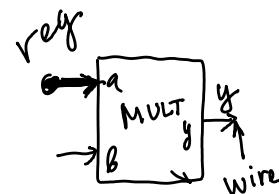
Các biến trong Verilog có thể được khai báo là dạng bit vector hoặc dạng bộ nhớ theo cú pháp

```
reg [7:0] a,b;
wire [15:0] y;
```

```
reg [31:0] patterns [0:255]; // khai báo một mảng bộ nhớ 256 ô nhớ 32 bit
```

- b) Tạo ra thiết kế như một thành phần bên trong testbench (Module instantiation)

- c) Tạo mẫu đầu vào (kích thích đầu vào)



```
module multiplier (
    input [7:0] a,b;
    output [15:0] y;
)
module multiplier_test;
    // khai báo các biến kết nối
    reg [7:0] a_sim;
    reg [7:0] b_sim;
    wire [15:0] y_sim;
    // khai báo bộ nhớ lưu trữ
    // 256 mẫu đầu vào ra và biến địa chỉ cho bộ nhớ
    reg [31:0] patterns [0:255];
    reg [7:0] addr;
    // tạo ra thiết kế nằm trong testbench
    // DUT

```

b) Tao ra thiết kế như một thành phần bên trong testbench
(Module instantiation)

c) Tạo mẫu đầu vào (kích thích đầu vào)

Giá trị đầu vào được tạo ra và đưa tới thiết kế bằng các câu lệnh thủ tục
(được thực hiện tuần tự) nằm trong khối lệnh **initial** hoặc **always**.

Khối lệnh **initial** có cú pháp như sau:

```
initial
begin
... các lệnh thủ tục...
end
```

Khối lệnh **initial** được thực hiện duy nhất một lần từ lúc bắt đầu mô phỏng.
Các lệnh trong khối **initial** được thực hiện tuần tự.

Khối lệnh **always** có cú pháp như sau:

```
always @(danh_sách_tín_hiệu)
begin
... các lệnh thủ tục
end
```

Trong đó danh_sách_tín_hiệu là **ten_bien_1 or ten_bien_2**

or ...

Khối lệnh **always** được thực hiện bất cứ khi nào có sự kiện
xuất hiện ứng với các tín hiệu trong danh_sách_tín_hiệu.

Các lệnh trong khối **always** sẽ được thực hiện tuần tự
ngoại trừ lệnh gán **non-blocking**.

Mẫu tín hiệu đầu vào có thể được tạo ra theo các cách
như sau:

- 1) Trực tiếp: một số tổ hợp giá trị đầu vào được gán trực tiếp
cho các biến nối với đầu vào của thiết kế (direct
test/simulation)

Thời gian đưa giá trị đầu vào tới thiết kế được điều khiển
qua cú pháp tạo trễ lan truyền như sau:

```
#giá_trị_trễ_theo đơn vị thời_gian_mô_phỏng
```

Tạo đầu vào trực tiếp cho phép xây dựng testbench nhanh,
ít tốn công sức, thời gian tuy nhiên dễ bỏ qua các kịch bản
hoạt động khó trong mạch => không phát hiện hết các lỗi
trong thiết kế.

- 2) Ngẫu nhiên: nhiều tổ hợp giá trị đầu vào ngẫu nhiên được
tạo ra và đưa tới đầu vào của thiết kế.

Thời gian đưa giá trị đầu vào có thể được điều khiển thông
qua cú pháp tạo trên lan truyền

Số tổ hợp đầu vào được tạo ra bằng lệnh lặp có điều
kiện (Lệnh repeat, while, for)

Số ngẫu nhiên được tạo bởi hàm hệ thống **\$random()**

Tạo đầu vào ngẫu nhiên cho phép xây dựng testbench
nhanh, tạo được nhiều mẫu đầu vào, ít bị thiên lệch so với
tạo đầu vào trực tiếp => xác suất xuất hiện lỗi cao hơn. Tuy
nhiên: dễ tạo ra các đầu vào không hợp lệ, hoặc chỉ đi theo
một hướng hoạt động, vẫn có khả năng để sót lỗi, bỏ qua
các trường hợp hoạt động góc (corner cases)

- 3) Toàn bộ: tạo ra tất cả các tổ hợp đầu vào cho thiết kế.
Sử dụng vòng lặp for với biến lặp chính là các biến kết nối
với đầu vào

// tạo ra thiết kế

```
// tạo ra thiết kế nằm trong testbench
// DUT
```

```
multiplier mult_dut(.a(a_sim), .b(b_sim), .y(y_sim));
```

// tạo kích thích đầu vào

// trực tiếp

```
initial
begin
#5 a_sim = 0; b_sim=0;
#5 a_sim = 10; b_sim = 10;
end
```

// tạo kích thích đầu vào

// ngẫu nhiên

```
initial
begin
repeat (20)
begin
#5 a_sim = $random();
b_sim=$random();
end
end
```

// tạo ra tất cả các kích thích đầu vào

initial

begin

```
for (a_sim=0;a_sim<255;a_sim=a_sim+1)
  for (b_sim=0;b_sim<255;b_sim=b_sim+1)
    #5;
```

Sử dụng vòng lặp for với biến lặp chính là các biến kết nối với đầu vào

Tạo đầu vào toàn bộ chỉ được sử dụng cho các mạch có kích thước nhỏ. Không khả thi cho mạch kích thước lớn. Tuy nhiên, đảm bảo chắc chắn phát hiện lỗi

- 4) Kết hợp ngẫu nhiên và trực tiếp theo kịch bản hoạt động
B1: Mô phỏng ngẫu nhiên mạch

B2: Đo chất lượng mô phỏng

B3: Phân tích các kịch bản hoạt động của mạch để tạo ra các trường hợp đầu vào trực tiếp tương ứng với các kịch bản hoạt động của mạch để đạt được chất lượng mô phỏng tốt hơn

Các kịch bản hoạt động của mạch cần được liệt kê trong bước Verification Plan (thực hiện đồng thời với bước Architecture design) bằng cách:

- Phân chia chức năng của thiết kế: Ứng với mỗi chức năng sẽ là một trường hợp hoạt động
- Phân chia cấu hình thiết kế: Ứng với mỗi cấu hình là một trường hợp hoạt động
- Phân chia sơ đồ khối của thiết kế: các trường hợp hoạt động cho phép kích hoạt tất cả các tín hiệu kết nối giữa các khối

- d) Quan sát và kiểm tra đầu ra

- Quan sát và kiểm tra thủ công: Giá trị đầu ra của thiết kế được hiển thị lên màn hình dưới dạng biểu đồ dạng sóng hoặc dạng văn bản. Kỹ sư thiết kế sẽ so sánh thủ công giữa đầu ra của thiết kế và giá trị đúng (do anh ta tự tính toán) và quyết định sự đúng đắn của mạch.

Các giá trị tín hiệu được đưa ra bằng hàm hệ thống

\$monitor(chuỗi định dạng, danh sách các biến cần quan sát). Hàm \$monitor sẽ được thực hiện bất cứ khi nào có sự kiện xảy ra với các biến trong danh sách quan sát.

- Không mất thời gian công sức để xây dựng testbench
 - Mất rất nhiều công sức để so sánh đổi chiều
 - Chỉ có thể áp dụng cho số lượng rất ít tổ hợp giá trị đầu vào
 - Chỉ áp dụng để gỡ lỗi
 - Cần được thay thế bằng phương pháp tự động và teschbench tự kiểm tra (self-test)
- Quan sát và kiểm tra tự động

Testbench cần có cơ chế tạo ra đầu ra đúng và so sánh đầu ra đúng với đầu ra của thiết kế, cảnh báo khi có sự sai khác

- Đầu ra đúng có thể được tạo ra bằng mã Verilog ngay bên trong testbench. Testbench bao gồm triển khai thay thế (thứ 2) của thiết kế. Tuy nhiên bản triển khai thiết kế trong testbench không cần tối ưu, không cần tổng hợp và có thể ở mức độ trừu tượng cao hơn thiết kế thật. (Mô hình chuẩn trong testbench-golden model)

Mô hình chuẩn và việc so sánh đầu ra được thực hiện trong khối lệnh always được kích hoạt khi các biến đầu vào thiết kế thay đổi giá trị.

- Đầu ra đúng có thể được tạo ra bằng mô hình chuẩn ở mức phần mềm (mô hình C/C++, Matlab/Simulink).

```

for (a_sim=0;a_sim<255;a_sim=a_sim+1)
    for (b_sim=0;b_sim<255;b_sim=b_sim+1)
        #5;
// Chú ý: Do a_sim, b_sim là 2 số 8 bit,
// vòng lặp chỉ test được đến 254 để tránh bị lặp
// vô hạn lần
// để test được đến 255 cần khai báo a_sim
// b_sim là 2 số 9 bit
end

```

// Tạo đầu vào trực tiếp dựa trên kịch bản hoạt động

```

initial
begin
    #5 a_sim = 0; b_sim=0;
    #5 a_sim = 255; b_sim = 255;
    #5 a_sim = 0; b_sim = $random();
    #5 a_sim = 8'h55; b_sim = 8'h55;
end

```

// đưa ra giá trị mô phỏng
// để so sánh thủ công

```

initial
$monitor("%t: %d%d=%d",
    $time, a_sim, b_sim, y_sim);

```

```

always @(a_sim or b_sim)
begin
    if (a_sim*b_sim != y_sim)
        $display("%t: ERROR %d%d!=%d",
            $time, a_sim, b_sim, y_sim);
end

```

đầu vào thiết kế thay đổi giá trị.

- Đầu ra đúng có thể được tạo ra bằng mô hình chuẩn ở mức phần mềm (mô hình C/C++, Matlab/Simulink). Mô hình chuẩn thường được tạo ra ở bước Architecture Design.

Testbench giao tiếp với mô hình chuẩn để lấy đầu ra đúng thông qua:

- Giao diện tập tin: Mô hình chuẩn sẽ ghi giá trị đầu vào và đầu ra vào tập tin và testbench sẽ đọc các giá trị này từ tập tin.
- Giao diện lập trình: Testbench sẽ sử dụng giao diện lập trình PLI của Verilog để gọi các hàm C/C++ trong mô hình chuẩn nhằm yêu cầu tính giá trị đầu ra đúng
- Giao diện đồng mô phỏng phần cứng (Hardware-In-the-Loop): Testbench và mô hình chuẩn Matlab/Simulink được đồng thời mô phỏng bằng kỹ thuật HIL trong môi trường Matlab/Simulink.

Với giao diện tập tin, đầu vào và đầu ra sẽ được đọc vào mảng bộ nhớ bằng các lệnh **\$readmemh**, **\$readmemb**

Bài tập thuyết trình số 2: Cài đặt phần mềm mô phỏng Modelsim và mô phỏng kiểm tra chức năng của bộ nhân 8 bit song song.
Cần kiểm tra thiết kế của giáo viên đưa ra.

display /> t. LRRURR you you:-you ,

\$time, a_sim, b_sim, y_sim);

end

```
// đọc giá trị đầu vào, đầu ra từ một file text
// định dạng file text được cho như sau
// mỗi dòng là một số 32 bit ở hệ 16 trong đó
// 8 bit cao nhất là giá trị của a,
// 8 bit tiếp theo là giá trị của b,
// 16 bit thấp nhất là giá trị của y
// ví dụ:
// 00000000
// FFFFFE01
// 0OFF0000
// 01FF00FF
// 55551C39
initial
begin
    // đọc đầu vào, đầu ra từ file correct_outputs.txt
    $readmemh("correct-outputs.txt", patterns);
    // đưa kích thích tới thiết kế
    for (addr = 0; addr < 5; addr=addr+1)
        begin
            a_sim = patterns[addr][31:24];
            b_sim = patterns[addr][23:16];
            #5;
        end
    end

    // so sánh đầu ra của thiết kế và đầu ra từ file
    always @(a_sim or b_sim)
        begin
            if (patterns[addr][15:0] != y_sim)
                $display("%t: ERROR %d*%d!=%d",
                         $time, a_sim, b_sim, y_sim);
        end

```

endmodule

Chú ý: Các khối lệnh always, initial trong cùng một module có thể coi là hoạt động song song/dồng thời. Vì vậy, thời gian thực hiện các khối và mối quan hệ thời gian giữa chúng cần được xem xét cẩn thận. Ví dụ với module multiplier_test ở trên có thể xảy ra trường hợp a_sim được gán 2 giá trị khác nhau ở cùng một trường hợp mô phỏng (xung đột gán giá trị-contention). Để tránh xung đột chỉ gán giá trị cho một biến trong một khối initial/always duy nhất.

17/09/2013

Tuesday, September 17, 2013 8:03 AM

2.3. Một số câu lệnh có thể dùng trong testbench (Các lệnh lặp; Khối lệnh và điều khiển khối lệnh; Vào ra file; Đặt giá trị tín hiệu trong mạch) – 1, 5 LT

2.3.1. Khối lệnh initial

2.3.2. Khối lệnh always

Bổ xung:

- Ta có thể đặt tên cho các khối lệnh như sau

```
initial  
begin : ten_khoi_len  
    ... các lệnh thực hiện tuần tự...  
end
```

- Sự thực hiện các khối lệnh có thể được điều khiển khi khối lệnh có tên như sau

```
'timescale 1ns/1ns  
module test_diable();  
    integer a, b;  
    integer i;  
    reg clk;
```

```
initial begin : break  
    for (i = 0; i < 20; i = i+1) begin : continue  
        @(posedge clk)  
        if (a == 0) // "continue" loop  
            disable continue; // Dừng không thực hiện lệnh bên trong khối continue  
        if (a == b) // "break" from loop  
            disable break; // Dừng không thực hiện lệnh bên trong khối break  
        $display("%t:Inside continue block a=%d,b=%d,i=%d",$time,a,b,i);  
    end  
    $display("%t:Inside break block a=%d,b=%d,i=%d",$time,a,b,i);  
end
```

```
initial begin  
    a = 2; b = 1;  
    #20 a = 0;  
    #20 a = 3;  
    #20 a = 1;  
end
```

```

initial begin
    clk = 0;
    forever #5 clk = ~clk;
end
initial $monitor ("%t:a=%d,b=%d,i=%d",$time,a,b,i);
endmodule

```

Kết quả mô phỏng

- run 19

```

#          0:a=      2,b=      1,i=      0
#          5:Inside continue block a=      2,b=      1,i=      0
#          5:a=      2,b=      1,i=      1
#          15:Inside continue block a=      2,b=      1,i=      1
#          15:a=      2,b=      1,i=      2

```

- run 6

```
#          20:a=      0,b=      1,i=      2
```

- run 1

```
#          25:a=      0,b=      1,i=      3
```

- run 16

```
#          35:a=      0,b=      1,i=      4
```

```
#          40:a=      3,b=      1,i=      4
```

- run 5

```
#          45:Inside continue block a=      3,b=      1,i=      4
```

```
#          45:a=      3,b=      1,i=      5
```

- run 15

```
#          55:Inside continue block a=      3,b=      1,i=      5
```

```
#          55:a=      3,b=      1,i=      6
```

```
#          60:a=      1,b=      1,i=      6
```

run 5

run 1

- Khi khối lệnh được bao bởi **begin...end**, các lệnh trong khối được thực hiện tuân tự. Khi khối lệnh được bao bởi **fork...join**, các lệnh trong khối được thực hiện song song

2.3.3. Thời gian mô phỏng

- Đơn vị thời gian trong mô phỏng được quy định bởi lệnh **'timescale**. Trong đó quy định đơn vị gốc và sai số của thời gian mô phỏng
'timescale 1ns/0.1ns
- Để điều khiển sự kiện về thời gian ta dùng 2 lệnh

- #: Điều khiển độ trễ thực hiện lệnh tiếp theo. Ví dụ: #5, #10
- @: Điều khiển sự dừng chờ một sự kiện với tín hiệu. Ví dụ: @(a or b), @(posedge clk)

2.3.4. Các lệnh lặp

- repeat: Dùng để lặp một số lần cố định
- for: Dùng lặp với chỉ số
- forever: Dùng lặp vô hạn lần
- while: Dùng lặp khi điều kiện lặp có thể thay đổi

2.4. Một số hàm hệ thống trong Verilog – 0,5 LT

CHƯƠNG 3: Mô tả IC số, hệ thống số dùng ngôn ngữ Verilog (12LT+3BT)

3.1. Các thành phần cơ bản của ngôn ngữ Verilog (Nhắc lại chương 1: Khái niệm module, port, chú thích; Khai báo tín hiệu, biến; hằng trong mạch; Các mô hình mô tả mạch: mô hình cấu trúc, mô hình dòng dữ liệu, mô hình hành vi) – 1LT

3.1.1. Khái niệm module và port

a) Module

- Thiết kế mạch bằng Verilog được khai báo bằng cú pháp module
- Cú pháp

```
module tên_module (
    khai_báo_cổng_vào_ra
);
    /* Khai báo biến, tham số sử dụng trong module; */
    /* Mô tả cấu trúc, hoạt động của module bằng các
       cú pháp Verilog;*/

endmodule
```

b) Cổng vào ra

- Cổng vào ra của thiết kế được khai báo cùng với module
- Cú pháp
input/output/inout [khoảng_chỉ_số] tên_cổng,

Trong đó:

- **input, output, inout** được sử dụng để chỉ ra hướng tín hiệu (vào, ra, hoặc vào/ra) của cổng
- **[khoảng_chỉ_số]** là tùy chọn dùng để khai báo một cổng là bit vector (gồm nhiều tín hiệu).
khoảng_chỉ_số có cú pháp
[chỉ_số_lớn:chỉ_số_nhỏ]
- các cổng trong khai báo được phân cách nhau bởi dấu ,; cổng cuối cùng trong khai báo kết thúc bằng dấu) của khai báo module
- Chú ý: Có thể sử dụng cú pháp như sau để khai báo module và cổng
Trong khai báo module chỉ liệt kê tên các cổng. Hướng và kích thước của cổng được chỉ rõ trong thân module.

```
module tên_module (danh_sách_cổng_vào_ra);
```

```
    /* Khai báo tham số sử dụng trong module */
```

```
    /* Khai báo cổng vào ra */
```

Ví dụ: Khai báo module và tham số

module mux 21

```

/* Khai báo tham số sử dụng trong module */

/* Khai báo cổng vào ra */
// [khoảng_chỉ_số] tên_port;
...
/* Khai báo biến, hằng sử dụng trong module */
/* Mô tả module */
endmodule

```

3.1.2. Tham số, hằng số, biến, tín hiệu

a) Tham số

- Tham số là một hằng số trong module được khởi tạo giá trị khi tạo ra/sử dụng module trong một thiết kế khác. Tham số được dùng để tái cấu hình lại module.
Ví dụ: Bộ cộng n bit
- Cú pháp: Khai báo cùng module và cổng vào ra
module tên_module #(**parameter** tên_tham_số = giá_trị_mặc định, tên_tham_số2=giá_trị_mặc định,...)
(khai_báo_cổng_vào_ra);
- Cú pháp bên trong thân module, trước khai báo cổng
parameter tên_tham_số = giá_trị_mặc định;

b) Hằng số

- Cú pháp
 - localparam** tên_hằng = giá_trị;
 - define** tên_hằng = giá_trị;
- localparam** thường được dùng để khai báo tên và giá trị mã hóa trạng thái của FSM
- define** thường được dùng để khai báo các giá trị mã hóa đầu vào hoặc đầu ra

tham số

```

module mux_21
#(parameter n=8)
(
  input [n-1:0] a,b,
  input s,
  output [n-1:0] y
);

```

...

```
endmodule
```

```
module mux_21 (a, b, s, y);
```

```

parameter n = 8;
[n-1:0] a,b;
s;
output y;

```

...

```
endmodule
```

Ví dụ: Khai báo hằng số

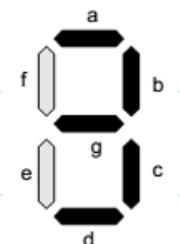
- Khai báo 3 trạng thái của mạch IDLE, SEND, RECV:

```

localparam IDLE = 2'b00;
localparam SEND = 2'b01;
localparam RECV = 2'b10;

```

- Khai báo hằng số giá trị đầu ra điều khiển LED 7 thanh



```

`define BLANK = 7'b1111111;
`define ZERO = 7'b0000001;
`define ONE = 7'b1001111;
`define TWO = 7'b0010010;

```

c) Biến và tín hiệu

- Cú pháp

kiểu_biến [khoảng_chỉ_số] tên_biến;

Trong đó:

- **kiểu biến:** có thể là **wire, reg, real, integer, time, wor, wand, tri, trireg, trior, triand, supply0, supply1**
- **khoảng chỉ số:** là khai báo tùy chọn để chỉ ra kích thước theo bit của biến (không áp dụng cho các kiểu integer, time, real)
- Các kiểu biến
 - Các kiểu biến net (**wire, wor, wand, tri, trior, triand, supply0, supply1**) được dùng để lan truyền giá trị, mô tả các dây dẫn kết nối thành phần mạch
 - Các kiểu biến reg (**reg, trireg, real, integer, time**) được dùng để lưu trữ giá trị. Trong đó **reg** thì được dùng để mô tả các cổng logic hoặc flip-flop, latch trong mạch. Các kiểu **real** (64 bit), **integer** (32 bit), **time** (64 bit) được dùng trong testbench khi mô phỏng-không có thành phần tương ứng trong phần cứng.
 - Các thành phần trong thiết kế phần cứng số đồng bộ thường được mô tả thông qua 2 loại biến chính là **wire** và **reg**
 - Phân biệt **wire** và **reg**

wire	reg
dùng để mô tả dây dẫn	dùng để mô tả cổng logic, flip-flop, latch
Không lưu trữ giá trị	lưu trữ giá trị cho đến khi được gán giá trị mới
Cần được nối với cổng, Nhận giá trị của cổng điều khiển nó	Không nhất thiết là thanh ghi hay flip flop
Kết nối đầu ra và đầu vào khi thực thể hóa các module	Có thể kết nối với 1 cổng vào của 1 module con Không thể kết nối với 1 cổng ra của 1 module con
Có thể khai báo tín hiệu input, output của module là wire	Có thể khai báo tín hiệu output là reg Không thể khai báo tín hiệu input là reg
Là kiểu duy nhất có thể nằm bên trái phép gán dùng assign	Không thể nằm bên trái phép gán assign
Không thể nằm bên trái phép gán = và <= trong khối always@	Là kiểu duy nhất nằm bên trái phép gán trong khối always và khối initial
Chỉ có thể dùng để mô tả logic tổ hợp logic	Dùng để mô tả logic tổ hợp và logic tuần tự

- Giá trị biến
 - Với các biến 1 bit: Nhận logic 4 trạng thái: 1'b0, 1'b1, 1'bX, 1'bZ
 - Với các biến bit vector: Nhận chuỗi bit, mỗi bit có

Ví dụ phân biệt wire và reg

```
module mux_21 (a, b, s, y);
  module test_mux;
    // Q: a_sim, b_sim, s_sim, y_sim có thể khai báo là kiểu gì?
    // A: a_sim, b_sim, s_sim có thể là kiểu wire, reg
    // y_sim phải là wire
    // nếu a_sim, b_sim, s_sim được gán nhiều giá trị trong initial/always => phải là kiểu reg
  endmodule
```

```
mux_21 dut(.a(a_sim),
            .b(b_sim),
            .s(s_sim),
            .y(y_sim));

```

endmodule

giá trị là logic 4 trạng thái

- Cú pháp: <số_bit>'<cơ_số>
 <chuỗi_chữ_số_giá_trị>
 Trong đó: <số_bit> xác định kích thước của
 giá trị
 <cơ_số> nhận giá trị: b, d, o, h tương ứng với
 cơ số 2, 10, 8, 16
 • Ví dụ: 2'b01, 2'b0X, 16'hF0A5, -16'd10

• Mảng

- Cú pháp: kiểu [khoảng_chỉ_số_1] tên_mảng [khoảng_chỉ_số_2];
- Trong đó: kiểu là wire, reg, integer, ...
- khoảng_chỉ_số_1 chỉ ra kích thước mỗi phần tử trong mảng
- khoảng_chỉ_số_2 chỉ ra kích thước mảng

3.1.3. Các mô hình mô tả mạch

a) Mô hình cấu trúc:

- Mô hình cấu trúc Verilog dùng để mô tả mạng (đồ thị) các phần tử của mạch. Trong đó các phần tử có thể là các cổng logic nguyên tố hoặc các module con. Mạng (đồ thị) các phần tử là cách kết nối đầu ra phần tử này tới đầu vào phần tử khác.
- Cổng logic nguyên tố (primitive gates): Là các cổng logic **and**, **or**, **xor**, **not**, **buff** được khai báo sẵn trong ngôn ngữ Verilog

n-Input	n-Output, 3-state
and	buf
nand	not
or	bufif0
nor	bufif1
xor	notif0
xnor	notif1
Cổng logic có 1 đầu ra (cổng thứ nhất) và n đầu vào	Cổng logic có n đầu ra và 1 đầu vào

- Các cổng logic nguyên tố sẽ được sử dụng trong mạch bằng cú pháp
 kiểu_cổng #giá_trị_trễ tên_cổng (danh_sách_tín_hiệu);

Trong đó:

- kiểu_cổng là các từ khóa **and**, **nand**, **or**, ...
- #giá_trị_trễ: là tùy chọn chỉ ra độ trễ lan truyền khi mô phỏng của cổng. Có nghĩa là đầu ra sẽ thay đổi giá trị tại thời điểm = thời điểm hiện tại + giá_trị_trễ
- tên_cổng: là tùy chọn tuân theo quy tắc đặt tên biến
- danh_sách_tín_hiệu là tên các biến bắt đầu bằng các biến được nối với đầu ra và theo sau bởi các biến nối với đầu vào
- Các thành phần của mạch có thể là các module thiết kế trước. Các module con được sử dụng trong mạch theo cách:

Ví dụ: Mô hình cấu trúc

```

module mux_21_1bit (
    input a, b, s,
    output y);
    wire y0, y1, not_s;
    not #1 g_not_s(not_s, s);
    and #1 g_y0 (y0, a, not_s);
    and #1 g_y1 (y1, b, s);
    or #1 g_y (y, y0, y1);

endmodule

module mux_21_nbit
    (parameter n=8)
    (
        input [n-1:0] a,b;
        input s;
        output [n-1:0] y;
    );
    mux_21_1bit
        mux_21_1bit_inst[n-1:0]
        (
            /* đầu vào a[0] của mux_21_nbit được
            kết nối với đầu vào a của đối tượng mux_
            21_1bit_inst[0] */
            .a(a),
            .b(b),
            /* tạo ra n-bit vector gồm n bits từ đầu
            vào s của mux_21_nbit để kết nối với đầu
            vào s của n đối tượng mux_21_
            1bit_inst[0] */
        );

```

- Các thành phần của mạch có thể là các module thiết kế trước. Các module con được sử dụng trong mạch theo cú pháp

tên_module tên_instance (danh_sách_kết_nối_vào_ra);

Trong đó:

- tên_module là tên khi khai báo module con
- tên_instance là tùy chọn tên của đối tượng module con được sử dụng trong thiết kế
- danh_sách_kết_nối_vào_ra được chỉ ra theo
 - cách ẩn: gồm danh sách tên các biến được kết nối theo **thứ tự** tới các cổng khai báo trong module: (tên_biến_1, tên_biến_2, ...)
 - cách rõ ràng: chỉ ra tên biến và tên cổng được kết nối với nhau
 .tên_cổng_1 (tên_biến_1),
 .tên_cổng_2 (tên_biến_2),
 ...

- Mảng các thành phần của mạch được tạo ra như sau

- Cú pháp
 tên_module/kiểu_cổng tên đối_tượng [khoảng_chỉ_số] (danh_sách_kết_nối_vào_ra);
- Với mảng thành phần thì
 danh_sách_kết_nối_vào_ra cần chỉ ra sự kết nối của các bit vector đại diện cho từng cổng vào ra của module được sử dụng.
- Cú pháp
 .tên_cổng_1 (tên_bit_vector_1),
 .tên_cổng_2 (tên_bit_vector_2),

trong đó tên_bit_vector_1, tên_bit_vector_2 cần có kích thước bằng số thành phần được tạo ra nhân với kích thước cổng. Khi đó thành phần thứ nhất của bit_vector_1[0] sẽ được kết nối với cổng_1 của đối tượng[0] được tạo ra.

```

vao s cua mux_21_nb1 de ket noi voi dau
vao s cua n doi tuong mux_21_
1bit_inst[0] */
.s({n{s}}),
.y(y)
);
endmodule

module test_mux;
wire [15:0] y;
reg [15:0] a,b;
reg s;
mux_21_nb1 #(n(16))
duv(.a(a), .b(b), .s(s), .y(y));

initial
begin
$monitor ("%t: a=%d, b=%d, s=%b, y=%d", $time, a, b, s, y);
#5;
repeat (10)
begin
a=$random();
b=$random();
s=$random();
#5;
end
end // initial begin
endmodule // test_mux

```

3.1.3 Các mô hình mô tả mạch

a) Mô hình cấu trúc

- Mô hình cấu trúc sẽ miêu tả mạng các thành phần cấu trúc của mạch. Tức là mô tả các thành phần của mạch và sự kết nối giữa chúng.
- Các thành phần của mạch có thể là
 - Cổng logic nguyên tố (primitive gates): Là các cổng logic **and**, **or**, **xor**, **inv**, **buf** được khai báo sẵn trong ngôn ngữ Verilog

n-Input	n-Output, 3-state
and	buf
nand	not
or	bufif0
nor	bufif1
xor	notif0
xnor	notif1
Cổng logic có 1 đầu ra (cổng thứ nhất) và n đầu vào	Cổng logic có n đầu ra và 1 đầu vào

- Các cổng logic nguyên tố sẽ được sử dụng trong mạch bằng cú pháp
`kiểu_cổng #giá_trị_trễ tên_cổng (danh_sách_tín_hiệu);`

Trong đó:

- kiểu_cổng là các từ khóa **and**, **nand**, **or**, ...
- #giá_trị_trễ: là tùy chọn chỉ ra độ trễ lan truyền khi mô phỏng của cổng. Có nghĩa là đầu ra sẽ thay đổi giá trị tại thời điểm = thời điểm hiện tại + giá_trị_trễ
- tên_cổng: là tùy chọn tuân theo quy tắc đặt tên biến
- danh_sách_tín_hiệu là tên các biến bắt đầu bằng các biến được nối với đầu ra và theo sau bởi các biến nối với đầu vào

Ví dụ:

```
and #5 g1(a, b, c, d); // Tạo ra một cổng AND 3 đầu vào nối với b, c, d; đầu ra nối với a. Cổng này có độ trễ 5 đơn vị mô phỏng.
```

- Các thành phần của mạch có thể là các module thiết kế trước. Các module con được sử dụng trong mạch theo cú pháp

tên_module tên_instance (danh_sách_kết_nối_vào_ra);

Trong đó:

- tên_module là tên khi khai báo module con
- tên_instance là tùy chọn tên của đối tượng module con được sử dụng trong thiết kế
- danh_sách_kết_nối_vào_ra được chỉ ra theo
 - cách ẩn: gồm danh sách tên các biến được kết nối theo thứ tự tới các cổng khai báo trong module: (tên_biến_1, tên_biến_2, ...): tín hiệu tên_biến_1 sẽ được kết nối với cổng vào/ra thứ nhất trong module.
 - cách rõ ràng: chỉ ra tên biến và tên cổng được kết nối với nhau
 - .tên_cổng_1 (tên_biến_1),
 - .tên_cổng_2 (tên_biến_2),
 - ...
 - Lời khuyên: nên sử dụng theo cách rõ ràng để tránh lỗi.

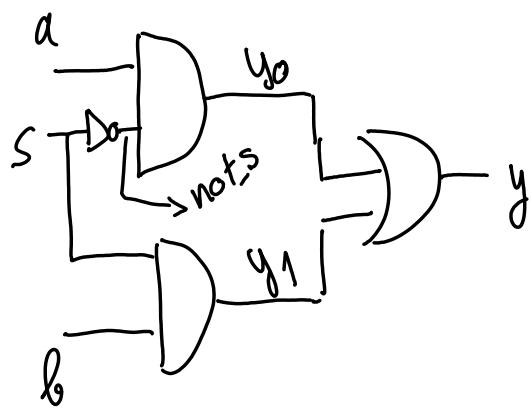
- Mảng các thành phần của mạch được tạo ra như sau

- Cú pháp
 - tên_module/kiểu_cổng tên_đối_tượng [khoảng_chỉ_số] (danh_sách_kết_nối_vào_ra);
- Với mảng thành phần thì danh_sách_kết_nối_vào_ra cần chỉ ra sự kết nối của các bit vector đại diện cho từng cổng vào ra của module được sử dụng.
- Cú pháp
 - .tên_cổng_1 (tên_bit_vector_1),
 - .tên_cổng_2 (tên_bit_vector_2),

trong đó tên_bit_vector_1, tên_bit_vector_2 cần có kích thước bằng số thành phần được tạo ra nhân với kích thước cổng. Khi đó thành phần thứ nhất của bit_vector_1[0] sẽ được kết nối với cổng_1 của đối tượng[0] được tạo ra.

- Ví dụ: MUX 21 n-bit

```
module mux_21_1bit (
    input a, b, s,
    output y);
    wire y0, y1, not_s;
    not #1 g_not_s(not_s, s);
    and #1 g_y0 (y0, a, not_s);
    and #1 g_y1 (y1, b, s);
    or #1 g_y (y, y0, y1);
endmodule
```



```

or #1 g_y (y, y0, y1);

endmodule

module mux_21_nbit
#(parameter n=8)
(
    input [n-1:0] a,b,
    input s,
    output [n-1:0] y
);

mux_21_1bit
    mux_21_1bit_inst[n-1:0]
    (
        /* đầu vào a[0] của mux_21_nbit được kết nối với đầu vào a của đối tượng
        mux_21_1bit_inst[0] */
        .a(a),
        .b(b),
        /* tạo ra n-bit vector gồm n bits từ đầu vào s của mux_21_nbit để kết nối với
        đầu vào s của n đối tượng mux_21_1bit_inst[n-1:0] */
        .s({n{s}}),
        .y(y)
    );
endmodule
|
module test_mux;

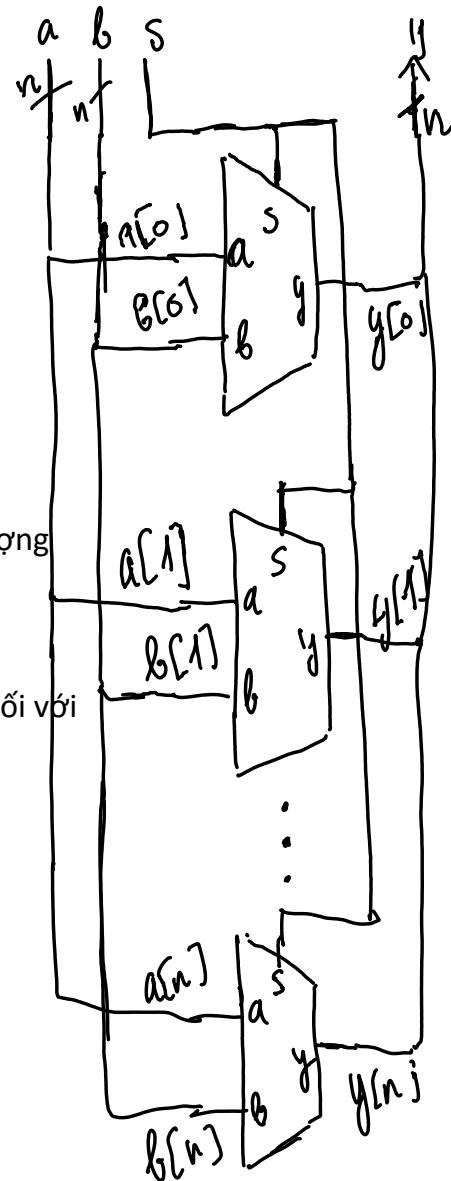
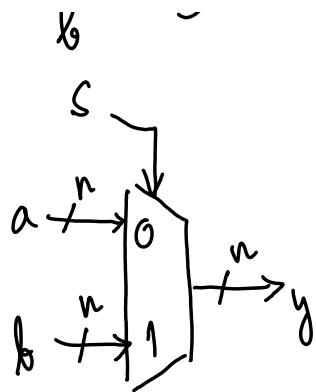
wire [15:0] y;
reg [15:0] a,b;
reg s;

mux_21_nbits #(n(16))
duv(.a(a), .b(b), .s(s), .y(y));

initial
begin
$monitor ("%t: a=%d, b=%d, s=%b, y=%d", $time, a, b, s, y);
repeat (10)
begin
a=$random();
b=$random();
s=$random();

#5;
end

```



```
    end // initial begin  
endmodule // test_mux
```

b) Mô hình dòng dữ liệu

- Mô hình dòng dữ liệu trong Verilog mô tả các biểu thức Bool của hàm truyền đạt của mạch.
- Mô hình dòng dữ liệu được dùng để mô tả logic tổ hợp
- Mô hình dòng dữ liệu sử dụng câu lệnh gán liên tục **assign** để gán biểu thức Bool cho 1 biến wire
- Hay được dùng để mô tả đường dữ liệu
- Mô hình dòng dữ liệu sẽ được phần mềm tổng hợp chuyển đổi thành mạch tự động
- Mô hình dòng dữ liệu sẽ đơn giản, dễ hiểu hơn mô hình cấu trúc. Tuy nhiên trong các trường hợp cần mạch tốc độ rất cao, mô hình cấu trúc vẫn được sử dụng

Ví dụ: Bộ mux n-bit

```
module mux_21_nbit  
#(parameter n=8)  
(  
    input [n-1:0] a,b,  
    input s,  
    output [n-1:0] y  
,  
  
    assign y = (s==0)?a:b;  
endmodule
```

c)

3.1.3. Các mô hình mô tả mạch

- a) Mô hình cấu trúc:
- b) Mô hình dòng dữ liệu

- Mô hình dòng dữ liệu trong Verilog mô tả các biểu thức Bool của hàm truyền đạt của mạch.
- Mô hình dòng dữ liệu sử dụng câu lệnh gán liên tục **assign** để gán biểu thức Bool cho 1 biến wire
- Thông thường, mô hình dòng dữ liệu được dùng để mô tả logic tổ hợp.
(Có thể dùng mô hình dòng dữ liệu để mô tả phần tử chốt - latch)
- Hay được dùng để mô tả đường dữ liệu
- Mô hình dòng dữ liệu sẽ được phần mềm tổng hợp chuyển đổi thành mạch tự động
- Mô hình dòng dữ liệu sẽ đơn giản, dễ hiểu hơn mô hình cấu trúc. Tuy nhiên trong các trường hợp cần mạch tốc độ rất cao, mô hình cấu trúc vẫn được sử dụng
- Ví dụ: bộ mux_21_nbits

```
module mux_21_nbits #(parameter n=8) (
    input [n-1:0] a,b,
    input s,
    output [n-1:0] y);
```

```
    assign y = (s==0)?a:b;
endmodule
```

Ví dụ: bộ nhân 8 bit

```
module mult(
    input [7:0] a,b,
    output [15:0] y
)
```

```
    assign y = a*b;
endmodule
```

- c) Mô hình hành vi

- Mô hình hành vi mô tả hành động tính toán đầu ra của mạch khi đầu vào thay

đổi giá trị.

- Sử dụng khối lệnh always và các câu lệnh thủ tục để xây dựng mô hình hành vi.
Câu lệnh thủ tục: if else, case, for, repeat, do...while, phép gán biểu thức
- Mô hình hành vi có thể dùng để mô tả logic tổ hợp (always không có xung nhịp) và logic tuần tự (always có xung nhịp).
- Mô hình hành vi đơn giản gần với ngôn ngữ bậc cao tương tự như ngôn ngữ lập trình.
- Chú ý: Mô hình hành vi sẽ được chuyển đổi (tổng hợp) thành phần cứng nhờ phần mềm tự động; nhưng không phải bất cứ mô hình hành vi nào cũng có thể tổng hợp thành phần cứng. Luôn cần liên kết giữa mô hình hành vi và cấu trúc phần cứng tương ứng.
- Ví dụ: Mô hình hành vi của bộ mux n bit

```
module mux_21_nbts #(parameter n=8) (
    input [n-1:0] a,b,
    input s,
    output [n-1:0] y);

    always @(a or b or s)
    begin
        if (s==0) y = a;
        else y = b;
    end
endmodule
```

3.2. Mô tả mạch logic tổ hợp (Mô hình dòng dữ liệu: Biểu thức Bool và phép gán liên tục; Trễ lan truyền; Các phép toán cơ bản; Mô hình hành vi: Cấu trúc always; Khái niệm về sự kiện; Phép gán blocking) – 3LT

Mạch logic tổ hợp có thể mô tả bằng mô hình cấu trúc, mô hình dòng dữ liệu và mô hình hành vi.

3.2.1. Mô hình dòng dữ liệu

a) Phép gán liên tục

- Cú pháp:

assign tên_biến = biểu_thức;

- tên_biến là một biến kiểu **wire** hoặc tên cỗng đầu ra của module
- biểu_thức: biểu thức trên các biến sử dụng các toán tử Boolean hoặc số học

b) Biểu thức Verilog - phép toán

- Phép toán số học

Nhân (*), Mũ(**), Chia(/), Phần dư(%), Cộng(+), Trừ (-)

Chú ý:

- Kích thước biểu thức
- Có thể tổng hợp

- Phép toán Boolean

- Bit-wise: and (&), or (|), xor (|^), ~

Chú ý: Kích thước biểu thức, mở rộng kích thước toán hạng ngắn

- Bit-reduction (rút gọn bit): and(&), nand(~&), or (|), nor (~|), xor(^), xnor (^^)

- Phép dịch: Dịch trái logic (<<), dịch phải logic (>>), dịch trái số học (<<<), dịch phải số học (>>>)

Chú ý: Kết quả tổng hợp phụ thuộc số bít dịch là hằng số hay không

- Phép toán điều kiện (phép toán logic)

- Phép toán logic: and (&&), or (||), not (!)

Chú ý: Nếu một toán hạng chứa x/z => kết quả là x. Nếu không kết quả là 0 (false), 1 (true).

- Phép so sánh

lớn hơn (>), lớn hơn bằng (>=), nhỏ hơn (<), nhỏ hơn bằng (<=) bằng (==), khác (!=)

bằng 4 giá trị (==), khác 4 giá trị (!==)

Chú ý: Phép so sánh sẽ cho kết quả x nếu toán hạng chứa x

Phép so sánh 4 giá trị sẽ so sánh sự giống nhau (khác nhau) của từng bit trong toán hạng kết cả x và z.

Ví dụ: Phép toán điều kiện

```
module relational_operators();
```

```
initial begin
```

```
    $display (" 5  <= 10 = %b", (5  <= 10));
    $display (" 5  >= 10 = %b", (5  >= 10));
    $display (" 1'bx <= 10 = %b", (1'bx <= 10));
    $display (" 1'bz <= 10 = %b", (1'bz <= 10));
```

```
// Case Equality
```

```
    $display (" 4'bx001 === 4'bx001 = %b", (4'bx001 === 4'bx001));
    $display (" 4'bx0x1 === 4'bx001 = %b", (4'bx0x1 === 4'bx001));
    $display (" 4'bx001 == 4'bx001 = %b", (4'bx001 == 4'bx001));
    $display (" 4'bz001 == 4'bz001 = %b", (4'bz001 == 4'bz001));
    $display (" 4'bz0x1 === 4'bz0x1 = %b", (4'bz0x1 === 4'bz0x1));
    $display (" 4'bz0x1 === 4'bz001 = %b", (4'bz0x1 === 4'bz001));
```

```
// Case Inequality
```

```
    $display (" 4'bx0x1 != 4'bx001 = %b", (4'bx0x1 != 4'bx001));
    $display (" 4'bz0x1 != 4'bz001 = %b", (4'bz0x1 != 4'bz001));
```

```
// Logical Equality
```

```
    $display (" 5  == 10  = %b", (5  == 10));
    $display (" 5  == 5  = %b", (5  == 5));
```

```

// Logical Inequality
$display ("5 != 5 = %b", (5 != 5));
$display ("5 != 6 = %b", (5 != 6));

if (4'bx001 == 4'bx001) // khi so sánh thông thường, kết quả biểu thức điều kiện là x tương đương với false => nhánh else được thực hiện
    $display ("4'bx001 == 4'bx001");
else
    $display ("4'bx001 != 4'bx001");

if (4'bx001 === 4'bx001) / khi so sánh 4 giá trị, kết quả biểu thức điều kiện là true => nhánh if được thực hiện
    $display ("4'bx001 === 4'bx001");
else
    $display ("4'bx001 !== 4'bx001");

end
endmodule

run
# 5 <= 10 = 1
# 5 >= 10 = 0
# 1'bx <= 10 = x
# 1'bz <= 10 = x
# 4'bx001 === 4'bx001 = 1
# 4'bx0x1 === 4'bx001 = 0
# 4'bx001 == 4'bx001 = x
# 4'bz001 == 4'bz001 = x
# 4'bz0x1 === 4'bz0x1 = 1
# 4'bz0x1 === 4'bz001 = 0
# 4'bx0x1 !== 4'bx001 = 1
# 4'bz0x1 !== 4'bz001 = 1
# 5 == 10 = 0
# 5 == 5 = 1
# 5 != 5 = 0
# 5 != 6 = 1
# 4'bx001 != 4'bx001
# 4'bx001 === 4'bx001

```

- Phép toán trên bit vector
 - Slice: Sử dụng để lấy ra bit vector con của 1 toán hạng bit vector.
Cú pháp: biến[index1:index2];
 - Nối bit vector: Sử dụng để nối các bit vector
Cú pháp: {biến_1, biến_2, ...};
 - Sao chép bit vector: Sử dụng để ghép n bản sao của một bit vector
Cú pháp: n{biến}

Víður

assign out = {a[1:0] b c d a[?]};

```
assign out = {a[1:0],b,c,d,a[2]};  
assign imm32 = {16{imm16[15]}, imm16}; // mở rộng dấu imm16
```

$$\text{imm } 16 = 16'b1010111100001010$$

imm32 = 32'b1010111100001010
imm16 = imm16[15:0]

thành số 32 bit

- Phép toán điều kiện

Cú pháp (biểu_thức_điều_kiện)?biểu_thức_1:biểu_thức_2;

assign y = (s==0)?a:b;

3.2.2. Mô hình hành vi

a) Khối lệnh always không có xung nhịp

- Cú pháp:

always @(danh_sách_kích_hoạt)

begin

...

danh_sách_lệnh_thủ_tục

end

Trong đó

danh_sách_kích_hoạt là danh sách các biến/tín hiệu phân cách bởi từ khóa **or**

- Hoạt động:

- Khối always sẽ được thực hiện bất cứ khi nào có sự thay đổi giá trị của các biến trong *danh_sách_kích_hoạt*
- Các khối always sẽ hoạt động song song cùng các khối initial, phép gán liên tục. *Thứ tự thực hiện các khối always, initial, phép gán liên tục là không xác định.*

- Tổng hợp

- Khối always khi không có xung nhịp được tổng hợp thành mạch tổ hợp hoặc mạch tuần tự sử dụng mạch chốt.
- Chú ý: Cần hạn chế sử dụng khối always không xung nhịp để mô tả mạch tuần tự.

b) Phép gán blocking (phép gán tuần tự)

- Cú pháp

biến_LHS = biểu_thức; (biến_LHS = #trễ biểu_thức)

Trong đó: biến_LHS là biến khai báo kiểu **reg**; biểu_thức là biểu thức

Verilog tương tự trong phép gán liên tục

- Hoạt động

- B1: Tính toán biểu_thức
- B2: Gán giá trị biểu_thức cho biến_LHS sau thời gian trễ
- B3: Kết thúc phép gán tuần tự (thực hiện lệnh tiếp theo)
- Chú ý: Lệnh tiếp theo của phép gán tuần tự chỉ được thực hiện sau khi phép gán tuần tự kết thúc

- Tổng hợp:

- Thông thường, phép gán blocking sẽ được tổng hợp thành mạch logic

tổ hợp

- Nếu biến_LHS không được gán giá trị trong một số đường thực hiện của khối lệnh always thì phép gán blocking sẽ được tổng hợp thành mạch tuần tự sử dụng mạch chốt
- Một số phép toán trong Verilog sẽ không thể tổng hợp thành mạch hoặc tổng hợp thành mạch có kích thước lớn hoặc/và chậm.

c) Câu lệnh điều kiện if/else

• Cú pháp

if (biểu_thức_điều_kiện)

lệnh/khối_lệnh;

else

lệnh/khối_lệnh;

Trong đó: khối_lệnh gồm nhiều lệnh thủ tục bao bì từ khóa

begin...end

- Hoạt động: Nếu biểu_thức_điều_kiện có giá trị khác 0 hoặc x, khối lệnh nhánh if sẽ được thực hiện, nếu biểu thức điều kiện có giá trị 0, x thì khối lệnh nhánh else sẽ được thực hiện
- Tổng hợp:

- Các câu lệnh trong cả nhánh if và else sẽ được tổng hợp và chúng được ghép nối thông qua bộ mux
- Nếu một biến không được gán giá trị trong cả 2 nhánh if, else thì sẽ được tổng hợp thành mạch chốt
- Chú ý: Luôn luôn viết đủ nhánh cho câu lệnh điều khiển
- Các lệnh if lồng nhau sẽ tạo ra mạch lựa chọn có ưu tiên trong đó điều kiện của lệnh if phía trước sẽ có độ ưu tiên cao hơn (bộ mux tương ứng sẽ đứng gần đầu ra hơn)
- Với các lệnh if song song, lệnh if phía sau sẽ có độ ưu tiên cao hơn
- Khi điều kiện của các nhánh if lồng nhau không loại trừ nhau (overlapped conditions - có thể cùng đúng) thì kết quả mạch có thể khác với specification

• Ví dụ: câu lệnh điều kiện if/else

Mạch encoder 8-3

```
module encoder83(
    input [7:0]      data,
    output [2:0]     code
);
    reg [2:0] code;

    always @(data)
        begin
            if (data == 8'b0000_0001) code = 0; else
                if (data == 8'b0000_0010) code = 1; else
                    if (data == 8'b0000_0100) code = 2; else
```

```

        if (data == 8'b0000_1000) code = 3; else
            if (data == 8'b0001_0000) code = 4; else
                if (data == 8'b0010_0000) code = 5; else
                    if (data == 8'b0100_0000) code =
6; else
                        if (data == 8'b1000_0000)
code = 7; else code = 3'bx;
    end
endmodule

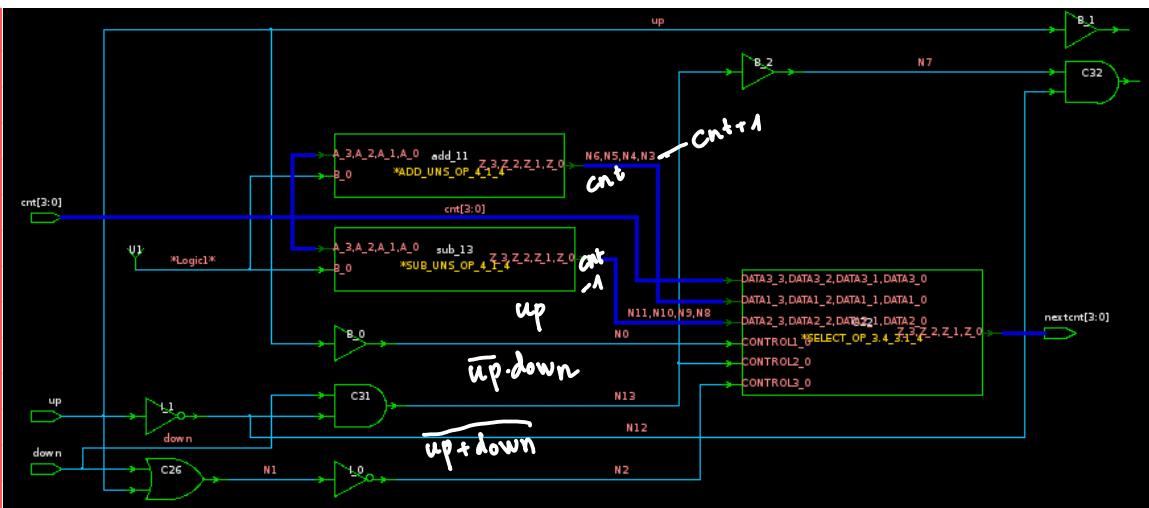
module encoder83_latch(
    input [7:0]      data,
    output [2:0] code
);
reg [2:0]                                     code;

always @(data)
begin
    if (data == 8'b0000_0001) code = 0; else
        if (data == 8'b0000_0010) code = 1; else
            if (data == 8'b0000_0100) code = 2; else
                if (data == 8'b0000_1000) code = 3; else
                    if (data == 8'b0001_0000) code = 4; else
                        if (data == 8'b0010_0000) code =
5; else
                            if (data == 8'b0100_0000) code =
6; else
                                if (data == 8'b1000_0000)
code = 7;
    end
endmodule

module encoder83_priority(
    input [7:0]      data,
    output [2:0] code
);
reg [2:0]                                     code;

always @(data)
begin
    if (data[0]) code = 0; else
        if (data[1]) code = 1; else
            if (data[2]) code = 2; else
                if (data[3]) code = 3; else
                    if (data[4]) code = 4; else
                        if (data[5]) code = 5; else
                            if (data[6]) code = 6; else
                                if (data[7]) code = 7; else
code = 3'bx;
    end
endmodule

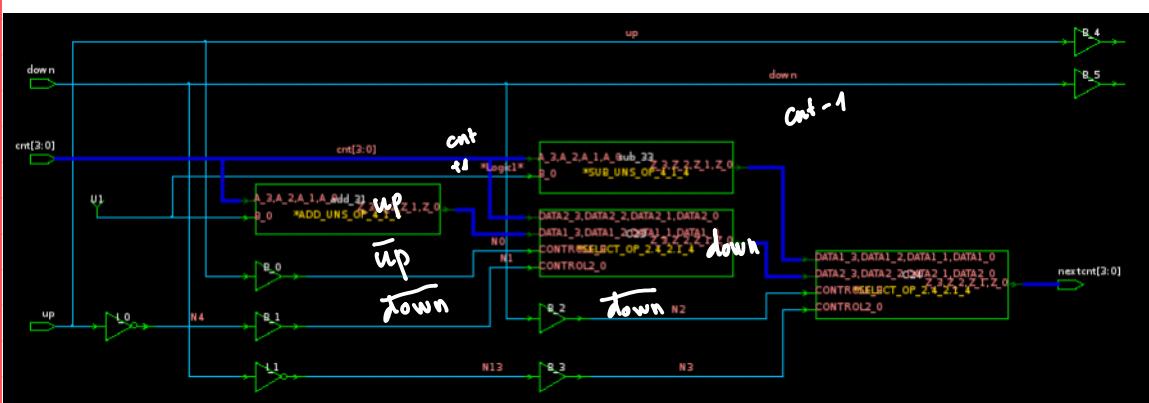
```



V

Ví dụ: updown

nested-if:



parallel-if

d) Câu lệnh lựa chọn case

- Cú pháp

case/casex/casez (biểu_thức)

giá_trị_1: khối_lệnh_1

giá_trị_2: khối_lệnh_2

...

default: khối_lệnh_n;

endcase

- Hoạt động

- Giá trị của biểu_thức sẽ được so sánh với các giá trị lựa chọn giá_trị_1, giá_trị_2, ... tùy từng loại case:

- case: so sánh sử dụng phép toán == (phân biệt giữa các giá trị 0, 1, x, z); khối lệnh có giá trị lựa chọn bằng giá trị biểu thức sẽ được thực hiện. Nếu không có giá trị lựa chọn bằng giá trị biểu

- thức, khối lệnh n tương ứng với nhánh default sẽ được thực hiện
- casex: các giá trị x, z, ? trong giá trị lựa chọn sẽ được coi là bằng với bít giá trị 0 và 1. Nếu có nhiều hơn 1 khối lệnh có giá trị lựa chọn bằng giá trị biểu thức thì khối lệnh đứng trước sẽ được thực hiện (có mức ưu tiên cao hơn)
 - casez: các giá trị z, ? trong giá trị lựa chọn sẽ được coi là bằng với bit giá trị 0 và 1. Nếu có nhiều hơn 1 khối lệnh có giá trị lựa chọn bằng giá trị biểu thức thì khối lệnh đứng trước sẽ được thực hiện (có mức ưu tiên cao hơn)
 - Tổng hợp:
 - Các khối lệnh của tất cả các nhánh lựa chọn đều được tổng hợp và được ghép nối qua khối mux
 - casex, casez sẽ được tổng hợp thành mạch có mức ưu tiên
 - nếu không có default và có biến không được gán giá trị trong mọi nhánh giá trị thì sẽ tổng hợp thành mạch tuần tự sử dụng chốt
 - Ví dụ
 - Bộ giải mã
 - Bộ tính toán số học logic ALU

ALUOp	Funct	ALUControl
00	X	010 (add)
X1	X	110 (subtract)
1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)
1X	100100 (and)	000 (and)
1X	100101 (or)	001 (or)
1X	101010 (slt)	111 (set less than)

- Tham khảo thêm: Clifford Cummings: "full_case parallel_case, the Evil Twins of Verilog Synthesis"
- Lời khuyên: chỉ sử dụng case (hạn chế dùng casex và casez), các giá trị lựa chọn trong các nhánh loại trừ nhau.
- Bài tập: Thiết kế lại bộ encoder83, encoder83_priority sử dụng case

e) Lệnh lặp tĩnh (Số lần lặp cố định không phụ thuộc biến số)

- Cú pháp
- Hoạt động

- Tổng hợp
-

3.2.2. Mô hình hành vi

- a) Khối lệnh always
- b) Câu lệnh gán tuần tự (blocking)
- c) Câu lệnh điều kiện if/else
- d) Câu lệnh lựa chọn case

- Cú pháp

case/casez/casez (biểu_thức)

giá_trị_1: khối_lệnh_1

giá_trị_2: khối_lệnh_2

...

default: khối_lệnh_n;

endcase

- Hoạt động

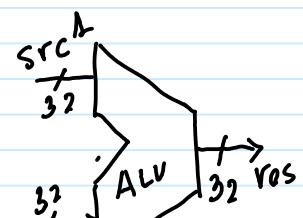
- Giá trị của biểu_thức sẽ được so sánh với các giá trị lựa chọn giá_trị_1, giá_trị_2, ... tùy từng loại case:

- case: so sánh sử dụng phép toán === (phân biệt giữa các giá trị 0, 1, x, z);
- casex: các giá trị x, z, ? trong giá trị lựa chọn sẽ được coi là bằng với bít giá trị 0 và 1. Ví dụ: 0x? có thể coi là bằng 000, 001, 010, 011
- casez: các giá trị z, ? trong giá trị lựa chọn sẽ được coi là bằng với bit giá trị 0 và 1.
- Khối lệnh có giá trị lựa chọn bằng giá trị biểu thức sẽ được thực hiện.
- Nếu không có giá trị lựa chọn bằng giá trị biểu thức, khối lệnh n tương ứng với nhánh default sẽ được thực hiện
- Nếu có nhiều hơn 1 khối lệnh có giá trị lựa chọn bằng giá trị biểu thức thì khối lệnh đứng trước sẽ được thực hiện (có mức ưu tiên cao hơn)

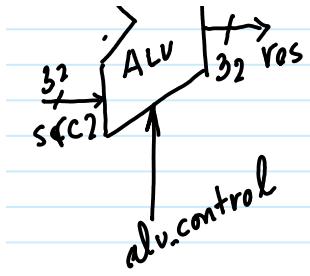
- Tổng hợp:

- Các khối lệnh của tất cả các nhánh lựa chọn đều được tổng hợp và được ghép nối qua khối mux
- casex, casez sẽ được tổng hợp thành mạch có mức ưu tiên khi có 2 nhánh lựa chọn cùng đúng
- nếu không có default và có biến không được gán giá trị trong mọi nhánh giá trị thì sẽ tổng hợp thành mạch tuần tự sử dụng chốt
- Ví dụ: Bộ tính toán số học và logic cho MIPS32

ALUOp	Funct	ALUControl
00	X	010 (add)
X1	X	110 (subtract)
1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)



1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)
1X	100100 (and)	000 (and)
1X	100101 (or)	001 (or)
1X	101010 (slt)	111 (set less than)



- Ví dụ (Bài tập về nhà)
 - Bộ decoder38, encoder83, encoder83_priority: Xây dựng bằng case
- Chú ý:
 - Tham khảo thêm: Clifford Cummings: "full_case parallel_case, the Evil Twins of Verilog Synthesis"
 - Lời khuyên: chỉ sử dụng case (hạn chế dùng casex và casez), các giá trị lựa chọn trong các nhánh loại trừ nhau
- e) Lệnh lặp tĩnh (Số lần lặp cố định không phụ thuộc biến số)
 - Cú pháp
 - for


```
for (gán_khởi_tạo;biểu_thức_điều_kiện;gán_chỉ_số)
                  lệnh/khổi_lệnh;
```

 trong đó: gán_khởi_tạo là phép gán giá trị đầu cho chỉ số; biểu_thức_điều_kiện sử dụng để kiểm tra điểm kết thúc của lệnh gán; gán_chỉ_số là phép gán thay đổi giá trị chỉ số
 - repeat


```
repeat (số_lần_lặp)
                  lệnh/khổi_lệnh;
```
 - Hoạt động
 - Với lệnh for:
 - Lệnh gán_khởi_tạo sẽ được thực hiện 1 lần ở bắt đầu vòng lặp
 - Biểu_thức_điều_kiện sẽ được kiểm tra mỗi lần lặp
 - Nếu biểu_thức_điều_kiện đúng thì các lệnh trong khối_lệnh sẽ được thực hiện
 - Lệnh gán_chỉ_số sẽ được thực hiện sau mỗi lần lặp
 - Với lệnh repeat: các lệnh trong khối_lệnh sẽ được thực hiện số_lần_lặp lần
 - Tổng hợp
 - Các lệnh lặp sẽ được trải (unroll) số lần bằng số_lần_lặp và sau đó được tổng hợp như một khối lệnh thủ tục thông thường
 - Ví dụ: mạch majority

3.2.2. Mô hình hành vi

- a) Khối lệnh always
- b) Câu lệnh gán tuần tự (blocking)
- c) Câu lệnh điều kiện if/else
- d) Câu lệnh lựa chọn case
- e) Câu lệnh lặp

- Cú pháp
 - **for** (lệnh_khởi_tạo; điều_kiện;lệnh_chỉ_số) **begin ... end**
 - **repeat** (số_lần_lặp) **begin ... end**
 - **while** (điều_kiện) **begin ... end**

- Hoạt động

- for:
 - Khi bắt đầu vòng lặp, lệnh_khởi_tạo được thực hiện để gán giá trị bắt đầu cho biến chỉ số điều khiển vòng lặp
 - Thực hiện lặp các lệnh:
 - Kiểm tra biểu thức điều_kiện, nếu biểu thức sai thì kết thúc lặp
 - Nếu biểu thức điều_kiện đúng thì thực hiện các lệnh trong vòng lặp bao bởi begin ... end
 - Thực hiện lệnh_chỉ_số để thay đổi giá trị biến chỉ số
- repeat: Thực hiện lặp số_lần_lặp lần các lệnh bao giữa begin...end
- while: Thực hiện các lệnh bao giữa begin...end đến khi nào biểu thức điều_kiện sai.

- Tổng hợp

- Với các lệnh lặp có số lần lặp cố định (vòng lặp tĩnh - static loop), phần mềm tổng hợp sẽ trải vòng lặp thành khối lệnh thủ tục thông thường và thực hiện tổng hợp
- Với các lệnh lặp có số lần lặp phụ thuộc biến (vòng lặp động non-static loop): phần mềm tổng hợp sẽ thường không tổng hợp thành mạch được. Tuy nhiên có một số trường hợp đặc biệt với phần mềm tổng hợp mức cao (high level synthesis), vòng lặp động có thể tổng hợp được.

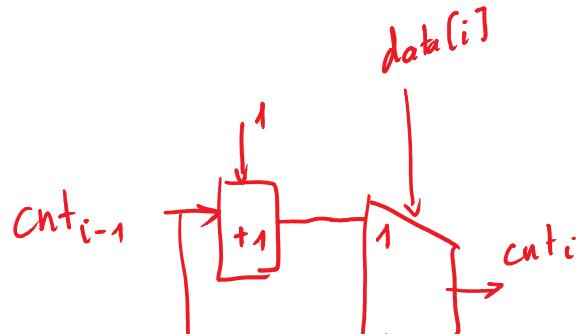
- Ví dụ: Mạch majority

```

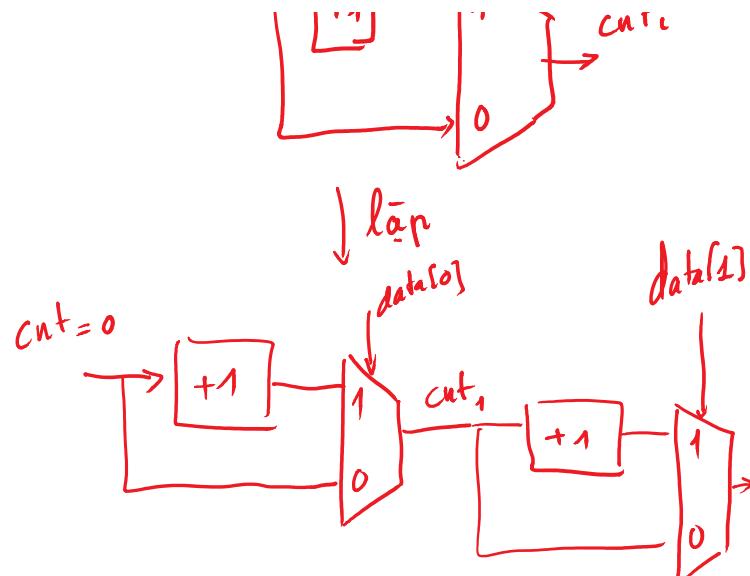
module majority
  # (parameter data_width = 8,
    cnt_width = 4,
    majority_value = 4
  )
  (
    input [data_width-1:0] data,
    output reg y
  );

  reg [cnt_width-1:0] cnt;
  integer i;

  always @(data)
  begin
    cnt = 0;
    for (i = 0;i<data_width;i=i+1)
    begin
      if (data[i]) cnt = cnt+1;
    
```



```
...  
for (i = 0;i<data_width;i=i+1)  
begin  
    if (data[i]) cnt = cnt+1;  
end  
if (cnt > majority_value)  
    y = 1;  
else  
    y = 0;  
end  
endmodule
```

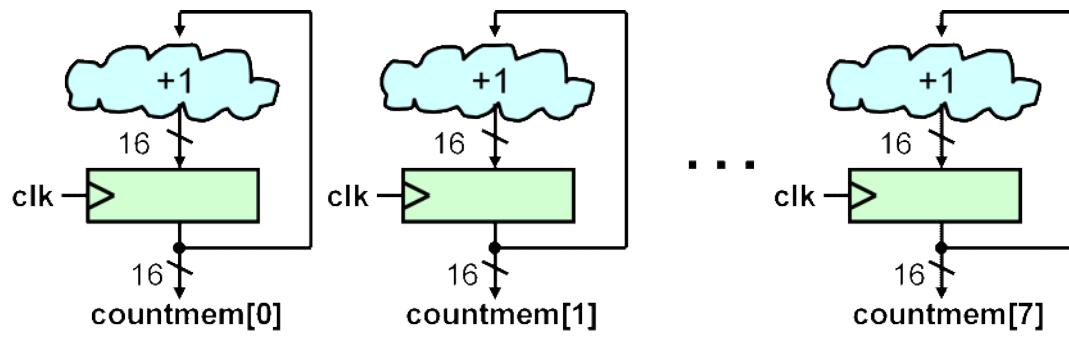


```
module test();  
  
    reg [7:0] data;  
    wire          y;  
  
    majority_for duv(.data(data), .y(y));
```

```
initial
begin
    $monitor ("%t: data=%b, y=%b", $time, data, y);
repeat (20)
begin
    data = $random();
    #5;
end
end
endmodule // test
```

Ví dụ vòng for trong khối always có đồng hồ

```
reg [15:0] countmem [0:7];  
integer x;  
always @(posedge clk) begin  
    for (x = 0; x < 8; x = x + 1) begin  
        countmem[x] <= countmem[x] +1;  
    end  
end
```



a) Chương trình con

- Chương trình con được sử dụng để đóng gói một đoạn mã và sử dụng lại đoạn mã đó nhiều lần.

3.2.2. Mô hình hành vi

- a) Khối lệnh always
- b) Câu lệnh gán tuần tự (blocking)
- c) Câu lệnh điều kiện if/else
- d) Câu lệnh lựa chọn case
- e) Câu lệnh lặp
- f) Chương trình con
 - Chương trình con được sử dụng để đóng gói một đoạn mã và sử dụng lại đoạn mã đó nhiều lần.
 - 2 loại chương trình con trong Verilog là task và function
 - task không trả về giá trị và được sử dụng như một câu lệnh thủ tục trong khối always hoặc initial
 - function trả về 1 giá trị vô hướng và được sử dụng trong biểu thức tính toán
 - Cú pháp
 - task:

```
task tên_task;
    khai_báo_output;
    khai_báo_input;
    khai_báo_biến;
    begin
        lệnh_thủ_tục;
    end
endtask
```
 - function:

```
function kiểu_giá_trị_trả_về tên_function;
    khai_báo_input;
    khai_báo_biến;
    begin
        lệnh_thủ_tục;
        tên_function = giá_trị_trả_về;
    end
endfunction
```
 - Tổng hợp: Task và Function sẽ được tổng hợp thành khối logic tổ

hợp với các đầu vào được chỉ ra trong khai_báo_input và đầu ra chỉ ra trong khai_báo_output (trường hợp task) và 1 đầu ra có kiểu_giá_trị_trả_về (trường hợp function). Để có thể tổng hợp được, task và function không được chứa các câu lệnh điều khiển thời gian và sự kiện (@, wait)

- Ví dụ:

- alu

3.3. Mô tả các phần tử nhớ Latch, Flip-flop theo sườn và theo mức (Mô tả phần tử chốt theo mức dùng phép gán liên tục; Mô tả phần tử Flip-flop theo sườn dùng cấu trúc always và phép gán non-blocking; Khái niệm về sườn đồng hồ; Mô tả tín hiệu khởi tạo Flip-flop không đồng bộ; So sánh phép gán non-blocking và phép gán blocking)-2 LT

3.3.1. Mô tả latch

- Khi biến ở bên tay trái phép gán được sử dụng ở phép toán bên tay phải phép gán liên tục thì phần tử chốt (latch) sẽ được tạo ra.
- Ví dụ:
assign d_latch = (en==1)?d_in:d_latch;

3.3.2. Mô tả flip-flop

- Sử dụng cấu trúc always với sự kiện là sườn dương hoặc sườn âm của đồng hồ.
- Cú pháp

```
always @(posedge clk)
begin
    d_ff <= d_in;
end
```

```
always @(negedge clk)
begin
    d_ff <= d_in;
end
```

- Chú ý: Trong khối always có đồng hồ/xung nhịp thường sử dụng phép gán song song (non-blocking <=). Phép gán song song trong cùng khối always được thực hiện song song:
 - Khi thực hiện mô phỏng, phép toán bên phải phép gán sẽ được tính toán trước; sau đó giá trị tính toán được gán cùng lúc cho các biến bên trái phép gán
 - Khi tổng hợp, các biến ở bên phải phép gán nhận giá trị hiện tại (giá trị tại đầu ra Q của Flip-flop)
- Ví dụ so sánh phép gán non-blocking và phép gán blocking
- Tham khảo: Clifford E. Cummings, "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!"
- Chú ý: Các phép gán non-blocking và blocking trong khối mô tả mạch tổ hợp

có thể hoạt động khác nhau khi mô phỏng, khi tổng hợp ở mức cồng và tổng hợp tới mức layout

Để đảm bảo mạch hoạt động đúng, phải mô phỏng sau khi layout-mô phỏng mạch ở lớp cồng với các thông tin đầy đủ về độ trễ của các cồng logic và độ trễ dây dẫn.

Ngoài ra có thể hạn chế sự sai khác (dẫn tới lỗi) khi mô phỏng, thiết kế mạch bằng cách sử dụng mạch đồng bộ

3.3.3 Mô tả tín hiệu reset của flip flop

a) Reset đồng bộ (FPGA)

```
always @(posedge clk)
begin
    if (reset)
        q <= 1'b0;
    else
        q <= d;
end
```

b) Reset không đồng bộ (ASIC)

```
always @(posedge clk or posedge reset)
begin
    if (reset)
        q <= 1'b0;
    else
        q <= d;
end
```

Bài tập về nhà: Cài đặt synopsys design compiler. Mô tả các loại flip-flop điều khiển bằng sườn dương, âm của đồng hồ, có reset đồng bộ, không đồng bộ. So sánh kích thước sau khi tổng hợp.

Kiểm tra: làm theo nhóm, gọi random theo người, tính điểm theo nhóm

	posedge clk	negedge clk
reset đồng bộ		
reset không đồng bộ, sườn âm		
reset không đồng bộ, sườn dương		

3.2.2. Mô hình hành vi

- a) Khối lệnh always
- b) Câu lệnh gán tuần tự (blocking)
- c) Câu lệnh điều kiện if/else
- d) Câu lệnh lựa chọn case
- e) Câu lệnh lặp
- f) Chương trình con

- Chương trình con được dùng để đóng gói 1 đoạn mã và tái sử dụng nó
- Chương trình con được dùng để mô tả một khối mạch => chương trình con sẽ có đầu vào, đầu ra, và có thể là mạch tổ hợp hoặc tuần tự.
- Trong Verilog có 2 loại chương trình con là
 - Thủ tục (Task)
 - Được sử dụng như một câu lệnh trong khối always hoặc khối initial
 - Không trả về giá trị, có thể có nhiều đầu ra, nhiều đầu vào
 - Trong thủ tục dùng để mô tả mạch tổ hợp, có thể sử dụng tất cả các câu lệnh như trong khối always trừ các câu lệnh điều khiển thời gian (@, wait)
 - Cú pháp:

```
task tên_thủ_tục;
    khai_báo_đầu_ra;
    khai_báo_đầu_vào;
    khai_báo_biến;
```



```
begin
    ....các lệnh....
```



```
end
endtask
```
 - Ví dụ:

```
module adder4bit (
    input [3:0] a, b,
    input ci,
    output [3:0] s,
    output co);
```

```
task full_adder;
```

```

input a, b, ci;
output s, co;

begin
    {co, s} = a+b+ci;
end
endtask

reg [2:0] c;

always @(a, b, ci)
begin
    full_adder (a[0], b[0], ci, s[0], c[0]);
    full_adder (a[1], b[1], c[0], s[1], c[1]);
    full_adder (a[2], b[2], c[1], s[2], c[2]);
    full_adder (a[3], b[3], c[2], s[3], co);
end
endmodule

```

- Hàm (Function)

- Được sử dụng như một biến trong các biểu thức
- Trả về 1 giá trị: có 1 đầu ra, nhiều đầu vào
- Hàm thì chỉ được sử dụng để mô tả mạch tổ hợp, có thể sử dụng tất cả các câu lệnh như trong khối always trừ các câu lệnh điều khiển thời gian (@, wait)
- Cú pháp


```

function [msb:lsb] tên_hàm;
    khai_báo_đầu_vào;
    khai_báo_bien;
```

```

begin
    ....các_lệnh....
    tên_hàm = giá_trị_trả_về;
end
endfunction

```

- Ví dụ

```

module adder4bit (
    input [3:0] a, b,
    input ci,
    output [3:0] s,
    output co);

function [1:0] full_adder;

input a, b, ci;

```

```

begin
    full_adder = a+b+ci;
end
endfunction

reg [2:0] c;

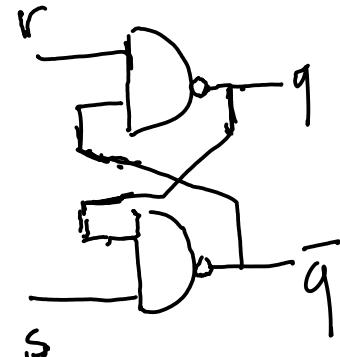
always @(a, b, ci)
begin
    {c[0], s[0]} = full_adder (a[0], b[0], ci);
    {c[1], s[1]} = full_adder (a[1], b[1], c[0]);
    {c[2], s[2]} = full_adder (a[2], b[2], c[1]);
    {co, s[3]} = full_adder (a[3], b[3], c[2]);
end
endmodule

```

3.3. Mô tả các phần tử nhớ Latch, Flip-flop theo sườn và theo mức (Mô tả phần tử chốt theo mức dùng phép gán liên tục; Mô tả phần tử Flip-flop theo sườn dùng cấu trúc always và phép gán non-blocking; Khái niệm về sườn đồng hồ; Mô tả tín hiệu khởi tạo Flip-flop không đồng bộ; So sánh phép gán non-blocking và phép gán blocking)-2 LT

3.3.1. Mô tả phần tử chốt (latch)

- Sử dụng phép gán liên tục với biến được gán giá trị ở phía trái phép gán được sử dụng ở biểu thức phía phải phép gán
 - Ví dụ: assign q_latch = (en==1)?d_in:q_latch;
 - Chú ý: Sự phụ thuộc vòng giữa các biến trong phép gán liên tục cũng tạo ra latch, nhưng nên hạn chế dùng
 - Ví dụ
assign q_rs = ~(r & q_bar_rs);
assign q_bar_rs = ~(s & q_rs);



3.3.2. Mô tả phần tử flip-flop

- Sử dụng khối always với danh sách độ nhạy là sườn xung nhịp
- Cú pháp


```

always @(posedge clk)
begin
    ....các_lệnh...
end

```

```
always @(negedge clk)
```

```
begin
```

```
....các_lệnh...
```

```
end
```

- Hoạt động: các_lệnh sẽ được thực hiện khi có sườn lên (xuống) của xung nhịp đồng hồ
- Tổng hợp: thành các flip-flop điều khiển bởi sườn lên (xuống) của xung nhịp
- Các phép gán trong khối always điều khiển bằng sườn đồng hồ thường là các phép gán song song (<=) (non-blocking)
- Các câu lệnh lựa chọn (if, case) trong khối always điều khiển bằng sườn đồng hồ không cần có đầy đủ các nhánh vì luôn tạo ra flip-flop
- Ví dụ:

```
module shifter (
```

```
    input d_in,
```

```
    input clk,
```

```
    output d_out);
```

```
    reg a, b, c;
```

```
always @(posedge clk)
```

```
begin
```

```
    a <= d_in;
```

```
    b <= a;
```

```
    c <= b;
```

```
    d_out <= c;
```

```
end
```

```
endmodule
```

- Phân biệt giữa phép gán song song và phép gán tuần tự

Tham khảo: Clifford E. Cummings, "Nonblocking Assignments in Verilog

Synthesis, Coding Styles That Kill!"

3.3.3. Mô tả phần tử flip-flop có tín hiệu reset

- a) Reset đồng bộ: tín hiệu reset không nằm trong danh sách điều khiển khối

```
always
```

```
always @(posedge clk)
```

```
begin
```

```
    if (reset)
```

```
        q <= 1'b0;
```

```
    else
```

```
        q <= d;
```

```
end
```

- b) Reset không đồng bộ: tín hiệu reset (sườn của nó) nằm trong danh sách điều khiển khối always

```
always @(posedge clk or posedge reset)
begin
    if (reset)
        q <= 1'b0;
    else
        q <= d;
end
```

	posedge clk	negedge clk
reset đồng bộ mức 0		
reset đồng bộ mức 1		
reset không đồng bộ, sườn âm		
reset không đồng bộ, sườn dương		

- Chú ý: Khi mô tả flip-flop cần chú ý tới các phần tử có trong thư viện chuẩn. Phần mềm tổng hợp sử dụng các phần tử có sẵn => một số loại flip-flop sẽ cần 2 phần tử (hoặc nhiều hơn) từ thư viện chuẩn => có kích thước lớn hơn. Nên sử dụng loại flip-flop có hỗ trợ trong thư viện chuẩn (kích thước mạch sẽ nhỏ hơn).
- Bài tập về nhà:

Cài đặt synopsys design compiler. Mô tả các loại flip-flop d điều khiển bằng sườn dương, âm của đồng hồ, có reset đồng bộ, không đồng bộ. So sánh kích thước sau khi tổng hợp.

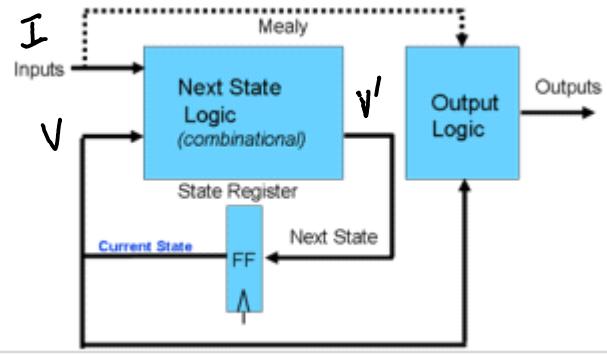
Kiểm tra: làm theo nhóm, gọi random theo người, tính điểm theo nhóm
 3.4. Mô tả máy trạng thái hữu hạn (Định nghĩa trạng thái và mã hóa trạng thái; Khai báo biến trạng thái hiện tại và biến trạng thái kế tiếp; Mô tả hàm chuyển trạng thái và hàm đầu ra bằng câu lệnh if/case; Mô tả phân tử nhớ biến trạng thái; Mô tả sự khởi tạo FSM; Một số chú ý về sự đầy đủ các nhánh trong câu lệnh if/case) – 2 LT

3.4.1. Khái niệm về máy trạng thái hữu hạn

Máy trạng thái hữu hạn FSM gồm các tập hợp

- S: tập các trạng thái được mã hóa bởi các biến trạng thái v
- X: tập các giá trị đầu vào được mã hóa bởi các biến đầu vào i
- Y: tập các giá trị đầu ra được mã hóa bởi các biến đầu ra o
- So: tập các trạng thái khởi tạo
- Hàm chuyển trạng thái: $s' = \Delta(s, x)$: $V' = \Delta(V, I)$
- Hàm đầu ra: $y = \Lambda(s, x)$: $O = \Lambda(V, I)$





FSM được triển khai bởi mạch tuần tự gồm các thành phần sau:

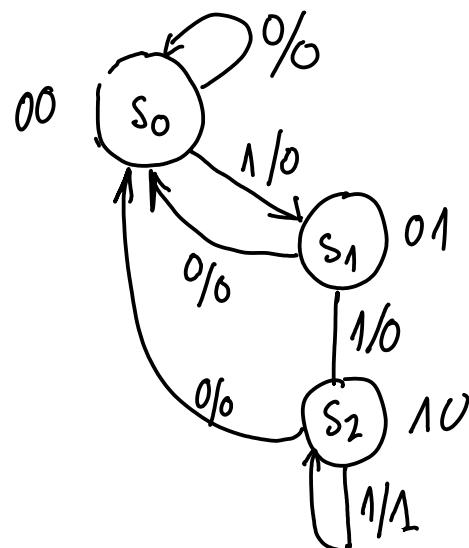
- Mạch tổ hợp triển khai hàm chuyển trạng thái Delta có đầu vào là đầu vào của mạch I và biến trạng thái hiện tại V, đầu ra là biến trạng thái kế tiếp của mạch
- Các phần tử nhớ (flip-flop) lưu biến trạng thái của mạch (Đầu vào D nối với V', đầu ra Q nối với V)
- Mạch tổ hợp triển khai hàm đầu ra Lamda có đầu vào là biến trạng thái hiện tại V (và đầu vào của mạch I) nếu mà máy Moore (Mealy), đầu ra là đầu ra của mạch

4.4.2. Mô tả FSM bằng Verilog

a) Nguyên tắc chung

- Mô tả tập các trạng thái và sự mã hóa các trạng thái bằng từ khóa localparam
- Biến trạng thái kết tiếp và trạng thái hiện tại cần được khai báo với kích thước phù hợp
- Mô tả đồ thị chuyển trạng thái bằng cấu trúc case
- Mô tả hàm đầu ra bằng cấu trúc case và if
- Mô tả flip-flop lưu trạng thái bằng cấu trúc always điều khiển bằng đồng hồ

b) Ví dụ: Thiết kế mạch nhận dạng chuỗi 3 bit 1 liên tiếp



```

module pattern111 (
    input di, clk, rst,
    output reg do
);

// khai báo tập trạng thái và mã hóa trạng thái
localparam s0 = 2'b00,
           s1 = 2'b01,
           s2 = 2'b10;

// khai báo biến trạng thái hiện tại, và kế tiếp
reg [1:0] state, nextstate;

// mô tả đồ thị chuyển trạng thái
always @(di or state)
begin
    case (state)
        s0: if (di) nextstate = s1; else nextstate = s0;
        s1: if (di) nextstate = s2; else nextstate = s0;
        s2: if (di) nextstate = s2; else nextstate = s0;
        default: nextstate = s0;
    endcase
end

// mô tả flip-flop
always @(posedge clk or negedge rst)
begin
    if (!rst)
        state <= s0;
    else
        state <= nextstate;
end

always @(posedge clk or negedge rst)
begin
    if (!rst)
        state <= s0;
    else
        case (state)

```

```

    s0: if (di) state <= s1;
    s1: if (di) state <= s2; else state <= s0;
    s2: if (di) state <= s2; else state <= s0;
    default: state <= s0;
    endcase
  end
end

// mô tả hàm đầu ra
always @(di or state)
begin
  case (state)
    s0: if (di) do=0; else do=0;
    s1: if (di) do=0; else do=0
    s2: if (di) do=1; else do=0;
    default: do=0;
    endcase
  end
endmodule

```

Bài tập về nhà: 1) Viết testbench cho fsm ở trên, xuất báo cáo coverage
 2) Tổng hợp bằng Design Compiler
 3) Tìm cách tối ưu diện tích / code

3.4. Mô tả máy trạng thái hữu hạn tường minh (Định nghĩa trạng thái và mã hóa trạng thái; Khai báo biến trạng thái hiện tại và biến trạng thái kế tiếp; Mô tả hàm chuyển trạng thái và hàm đầu ra bằng câu lệnh if/case; Mô tả phân tử nhớ biến trạng thái; Mô tả sự khởi tạo FSM; Một số chú ý về sự đầy đủ các nhánh trong câu lệnh if/case) – 2 LT

3.4.1. Khái niệm về máy trạng thái hữu hạn (nhắc lại)

3.4.2. Mô tả FSM bằng Verilog HDL

3.5. Thiết kế mô tả hệ thống số dùng phương pháp ASMD (Khái niệm nút trạng thái, nút đầu ra có điều kiện, nút hoạt động thanh ghi và nút quyết định trong đồ thị ASMD; Chuyển đổi đồ thị ASMD thành mô tả Verilog; Ví dụ minh họa:) – 2 LT.

3.6. Thiết kế, mô tả hệ thống số dùng phương pháp FSMD (Khái niệm về đường dữ liệu và khối điều khiển; Giao tiếp giữa đường dữ liệu và khối điều khiển; Mô tả đường dữ liệu; Mô tả khối điều khiển; Khái niệm về đường dữ liệu pipeline; Ví dụ minh họa: Mạch đếm tăng giảm) – 2 LT

3.5.1. Máy trạng thái tương tác - Interactive FSM

- Các hệ thống số phức tạp được thiết kế gồm nhiều máy trạng thái FSM tương tác với nhau
 - Đầu vào của một FSM có thể là đầu vào của toàn bộ hệ thống hoặc là đầu ra trạng thái của một FSM khác
 - Thông thường, khi 1 FSM ở một trạng thái nhất định, thì các FSM còn lại sẽ thực hiện một chuỗi chuyển trạng thái và khi chúng kết thúc sự chuyển trạng thái, thì FSM ban đầu sẽ thực hiện chuyển trạng thái.
 - Trong các hệ thống phức tạp, sẽ có một FSM chính (main-FSM) làm nhiệm vụ điều khiển hoạt động của toàn bộ hệ thống

• Ví dụ: Mạch đếm tăng, giảm, dừng

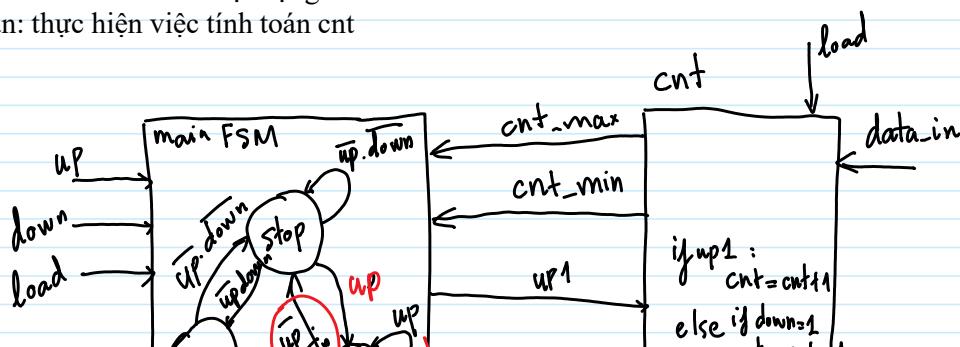
- module counter

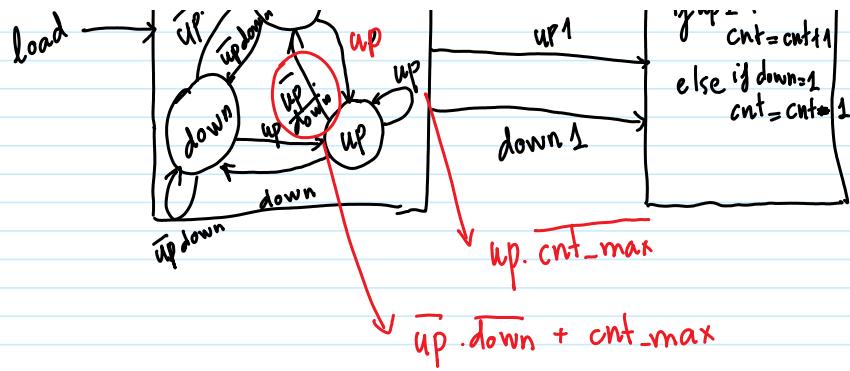
```
input clk, rst_n,  
input load, up, down,  
input [3:0] data_in,  
output reg [3:0] cnt);
```

```
// khi load = 1 thì cnt = data_in  
// khi up = 1 thì cnt = cnt+1 cho đến khi cnt đạt giá trị max = 15, thì  
// dừng  
// khi down = 1 thì cnt = cnt-1 cho đến khi cnt đạt min = 0  
// ngoài ra cnt = cnt
```

- mạch đếm có thể thiết kế gồm 2 FSM:

- FSM chính: điều khiển hoạt động
- FSM ẩn: thực hiện việc tính toán cnt





- Khi khôi điều khiển ở trạng thái UP, bộ đếm sẽ thay đổi trạng thái: 0->1->2->3..., Khi khôi điều khiển ở trạng thái DOWN, bộ đếm sẽ thay đổi trạng thái 4->3->2->...

Ví dụ: Mạch điều khiển đèn giao thông

- Để thiết kế hệ thống gồm nhiều FSM tương tác nhau người ta sử dụng mô hình ASMD (Algorithmic State Machine Datapath) hoặc FSMD (Finite State Machine Datapath)

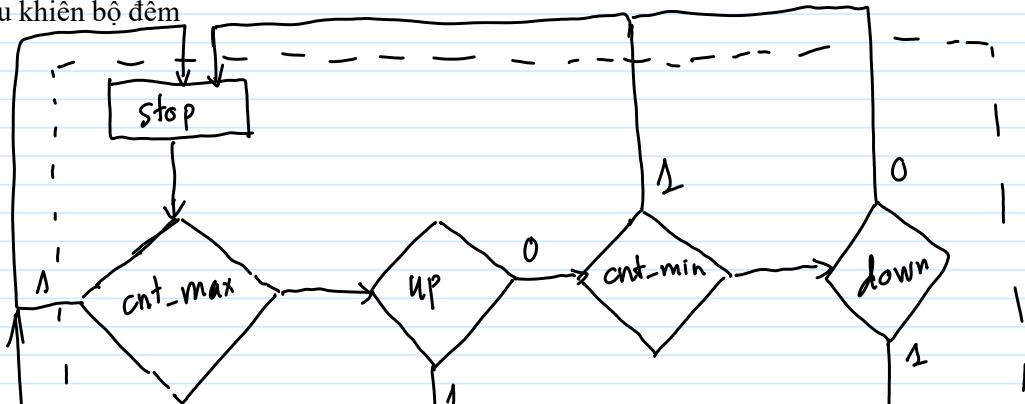
3.5.2. Mô hình thiết kế ASMD

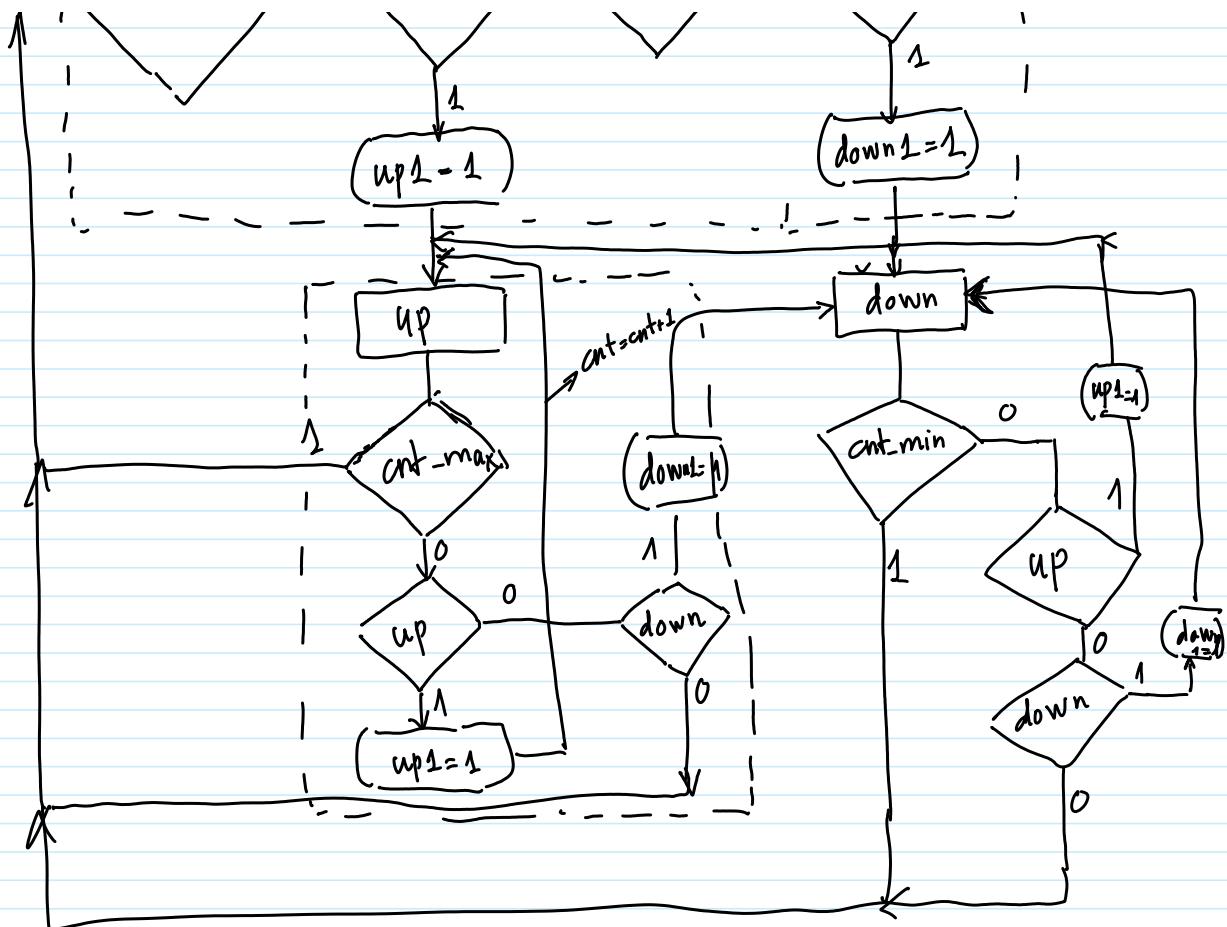
- ASM: Là một đồ thị biểu diễn thuật toán sử dụng trong phần cứng gồm các nốt thuộc 3 loại
 - Node trạng thái - state box (hình chữ nhật)
 - Node quyết định - decision box (hình thoi)
 - Đầu ra điều kiện - Conditional output box (round box)



- Các nốt trong ASM sẽ được nhóm thành các khôi. Mỗi khôi gồm
 - 1 nốt trạng thái
 - Nhiều nốt quyết định, nhiều nốt điều kiện
 - Một khôi có nhiều cạnh đi vào nốt trạng thái của khôi
 - Một khôi có thể có nhiều cạnh đi ra nối với các khôi khác
- ASM hoạt động như sau:
 - Đầu vào của ASM là các biến được kiểm tra trong nốt quyết định
 - Đầu ra của ASM là các biến được gán giá trị trong nốt đầu ra có điều kiện
 - Trạng thái của ASM là tập hợp các nốt trạng thái
 - Mỗi khôi trong ASM tương ứng với 1 trạng thái của máy FSM và hoạt động trong 1 chu kỳ xung nhịp
 - Các cạnh giữa các khôi tương ứng với 1 sự chuyển trạng thái của máy FSM và được thực hiện khi có sườn xung nhịp

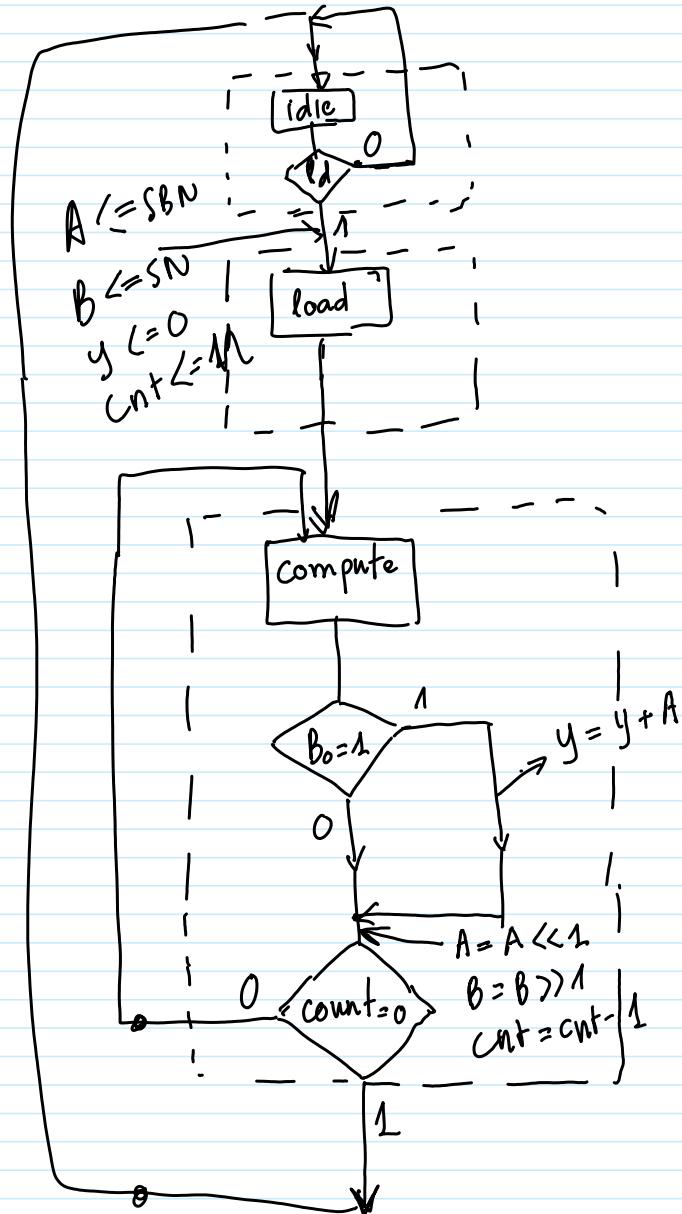
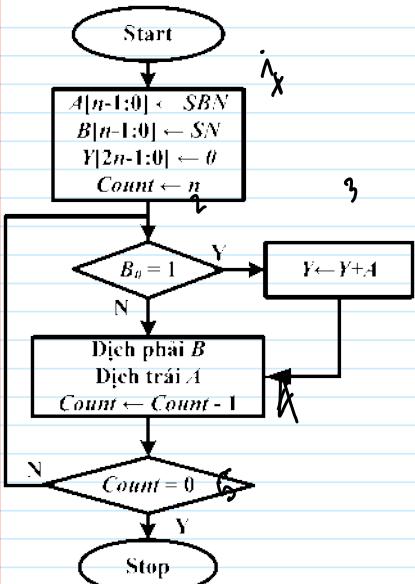
- Ví dụ: ASM điều khiển bộ đếm





- o ASMD: Sử dụng khi mô tả phần cứng gồm khối điều khiển và đường tính toán dữ liệu gồm
 - o Đồ thị ASM
 - o Các cạnh của đồ thị ASM sẽ được đánh dấu (annotate) bằng các phép toán thao tác trên dữ liệu.
 - o Hoạt động của ASMD: Khi hệ thống chuyển từ khối này sang khối khác thông qua 1 cạnh (khi có sườn xung nhịp) thì phép toán tương ứng trên cạnh đó sẽ được thực hiện
- o Các bước thiết kế dùng ASMD
 - o B1: Tìm hiểu thuật toán và vẽ lưu đồ thuật toán
 - o B2: Chia lưu đồ thuật toán thành các khối. Chú ý là độ phức tạp tính toán (số phép toán) trong mỗi khối nên tương đương nhau
 - o B3: Chuyển đổi lưu đồ thuật toán thành ASM.
 - Thêm nốt trạng thái cho mỗi khối
 - Tách riêng phép toán thao tác trên dữ liệu ra đánh dấu các cạnh chuyển giữa các khối
 - o B4: Xác định đầu vào của phần điều khiển ASM là các biến đầu vào trong nốt điều kiện và kết quả các phép so sánh dữ liệu (các biến đầu ra trạng thái của phần tính toán)
 - o B5: Xác định đầu ra của phần điều khiển là các biến điều khiển thực hiện tính toán tại các cạnh chuyển giữa các khối. Thêm nốt đầu ra vào các khối trong ASMD
 - o B6: Thực hiện mô tả bằng Verilog
- o Ví dụ: Thực hiện bộ nhân nối tiếp n bit.

Start



B1: Tìm hiểu về lưu đồ thuật toán

B2: Chia lưu đồ thuật toán thành các khối

B3: Chuyển đổi lưu đồ thuật toán thành ASMD

B3-1: Thêm nốt trạng thái vào mỗi khối

B3-2: Xác định các cạnh nối các khối

B3-3: Đánh dấu các cạnh bằng thao tác trên dữ liệu

B4: Xác định đầu vào, đầu ra của ASM

B4-1: Đầu vào

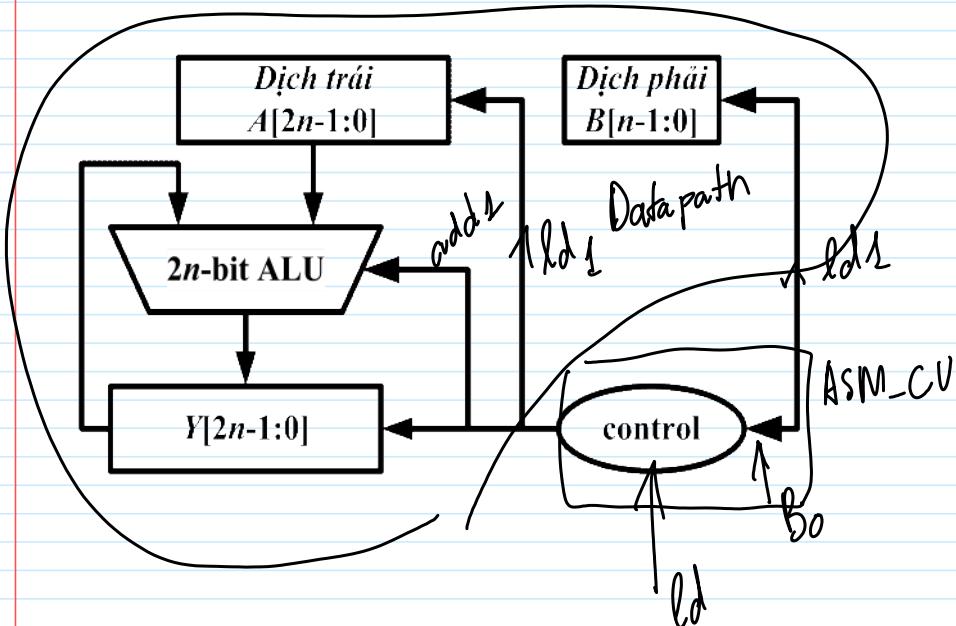
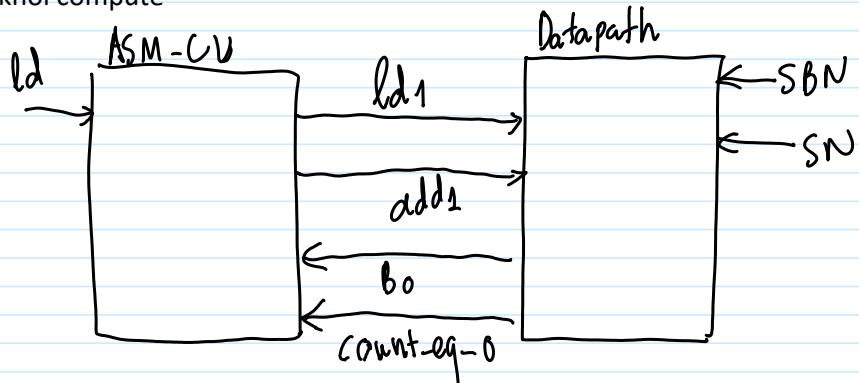
- Đầu vào Id
- Đầu vào là trạng thái của đường dữ liệu: B0, count_eq_0

B4-2: Đầu ra

- Đầu ra điều khiển quá trình nạp dữ liệu: Id1
- Đầu ra điều khiển quá trình cộng $Y = Y + A$: add1
- Đầu ra điều khiển quá trình dịch $A = A << 1$; $B = B >> 1$; $count = count - 1$ có thể bỏ qua, vì hành động tính toán này luôn được thực hiện ở cạnh ra khỏi khối compute

Author: *[Signature]*

thể bỏ qua, vì hành động tính toán này luôn được thực hiện ở cạnh ra khỏi khối compute



B5: Mô tả bằng Verilog

CHƯƠNG 4: Tổng hợp IC số, hệ thống số (12LT+3BT)

4.1. Khái niệm về tổng hợp logic (Lược đồ Y các biểu diễn mô hình biểu diễn mạch số; Đầu vào, đầu ra, các thành phần và các bước thực hiện cơ bản của phần mềm tổng hợp logic; Thư viện đích và thư viện liên kết trong tổng hợp; Giới thiệu sơ lược về một số kỹ thuật, thuật toán tổng hợp, tối ưu logic) – 1,5 LT

4.2. Tổng hợp mạch logic tổ hợp (Tổng hợp phép gán liên tục; Tổng hợp cấu trúc always và phép gán blocking; Tổng hợp cấu trúc if/case-Khai niêm cấu trúc ưu tiên priority structure; Tổng hợp sử dụng don't care; Chia sẻ tài nguyên trong tổng hợp và cấu trúc Verilog tương ứng; Tổng hợp mạch 3 trạng thái và bus) - 2,5 LT

4.3. Tổng hợp mạch dãy (Tổng hợp phần tử chốt: các lõi thường gấp với phần tử chốt; Tổng hợp flip-flop; Tổng hợp FSM và các mạch dãy có hoạt động được mô tả bằng khối always được kích hoạt bằng 1 sườn đồng hồ; Tổng hợp mạch dãy có hoạt động được mô tả trong nhiều chu kỳ đồng hồ; Tổng hợp thanh ghi; Tín hiệu reset; Điều khiển tín hiệu đồng hồ) – 4LT

4.4. Mô tả và tổng hợp mạch bằng cấu trúc lặp trong Verilog (Vòng lặp tĩnh không có điều khiển thời gian – logic tổ hợp; Vòng lặp tĩnh có điều khiển thời gian – mạch dãy một chu kỳ, đa chu kỳ; Vòng lặp động có điều

khiển thời gian – mạch dãy một chu kỳ, đa chu kỳ; Vòng lặp động không có điều khiển thời gian – không tổng hợp; Cấu trúc generate) – 3 LT
4.5. Thiết kế và tổng hợp mạch phức tạp (Kỹ thuật chia để trị - thiết kế sơ đồ khối của hệ thống; Tái sử dụng thiết kế có sẵn và yêu cầu cần thiết; Khái niệm về nhân sở hữu trí tuệ; Tham số hóa module Verilog; Hàm và thủ tục trong Verilog) – 1 LT

Bài tập thực hành số 1: Thiết kế mạch điều khiển thang máy

- Specification

- Input

- Nút tại các tầng

- input [5:1] up, down

- Nút trong buồng thang

- input keep_close, keep_open;
 - input [5:1] input_floor;

- Output

- output door; // = 1 mở, = 0 đóng

- output go_up, go_down; // = 10 đi lên, = 01 đi xuống, 11, 00 dừng - điều khiển đèn led hiển thị hướng đi ở từng tầng

- output [5:1] led_up, led_down; // đèn led nút bấm ngoài từng tầng

- output [2:0] current_floor; // điều khiển đèn led hiện thị số tầng

- output [5:1] led_floor; // đèn led nút bấm floor trong buồng thang

- Bài tập về nhà: Vẽ lưu đồ thuật toán mô tả hoạt động của thang máy

3.5. Thiết kế mô tả hệ thống số dùng phương pháp ASMD (Khái niệm nút trạng thái, nút đầu ra có điều kiện, nút hoạt động thanh ghi và nút quyết định trong đồ thị ASMD; Chuyển đổi đồ thị ASMD thành mô tả Verilog; Ví dụ minh họa:) – 2 LT.

3.6. Thiết kế, mô tả hệ thống số dùng phương pháp FSMD (Khái niệm về đường dữ liệu và khối điều khiển; Giao tiếp giữa đường dữ liệu và khối điều khiển; Mô tả đường dữ liệu; Mô tả khối điều khiển; Khái niệm về đường dữ liệu pipeline; Ví dụ minh họa: Mạch đếm tăng giảm) – 2 LT

3.5.1. Máy trạng thái tương tác - Interactive FSM

3.5.2. Mô hình thiết kế ASMD

B1: Tìm hiểu về lưu đồ thuật toán

B2: Chia lưu đồ thuật toán thành các khối

Khối 1: Box 2 trong lưu đồ thuật toán

Khối 2: Box 3, 4, 5, 6 trong lưu đồ thuật toán

Khối 3: Box 1 và 7 trong lưu đồ thuật toán

Q: Tại sao không ghép box 2 vào trong cùng khối 2?

Q: Lúc nào thì nên chia các box 3, 4, 5, 6 ra thành 2 khối?

Nguyên tắc để ghép các box trong lưu đồ thuật toán vào cùng 1 khối:

- Các thao tác ở các box có thể cùng được thực hiện song song (bằng phần cứng) trong 1 chu kỳ xung nhịp.
- Các box có số lượng tính toán tương đương

B3: Chuyển đổi lưu đồ thuật toán thành ASMD

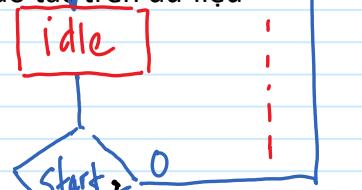
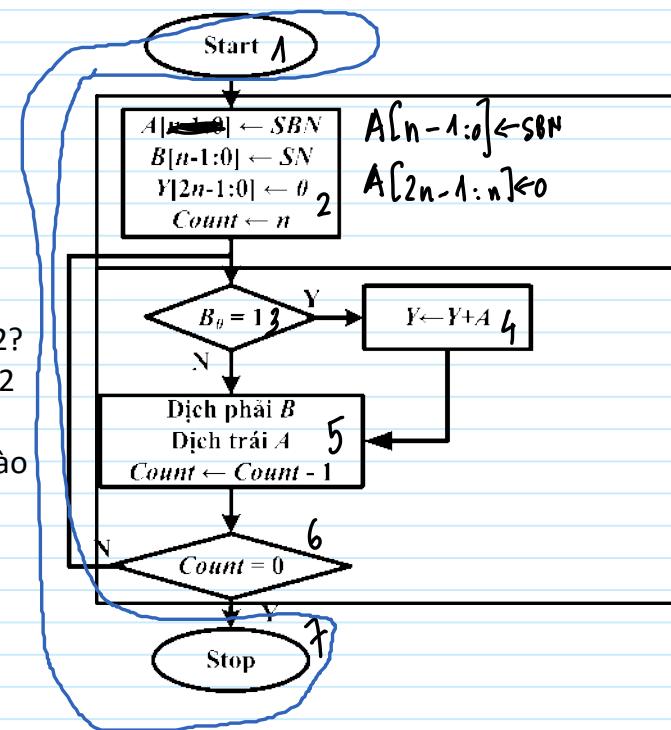
B3-1: Thêm nốt trạng thái vào mỗi khối

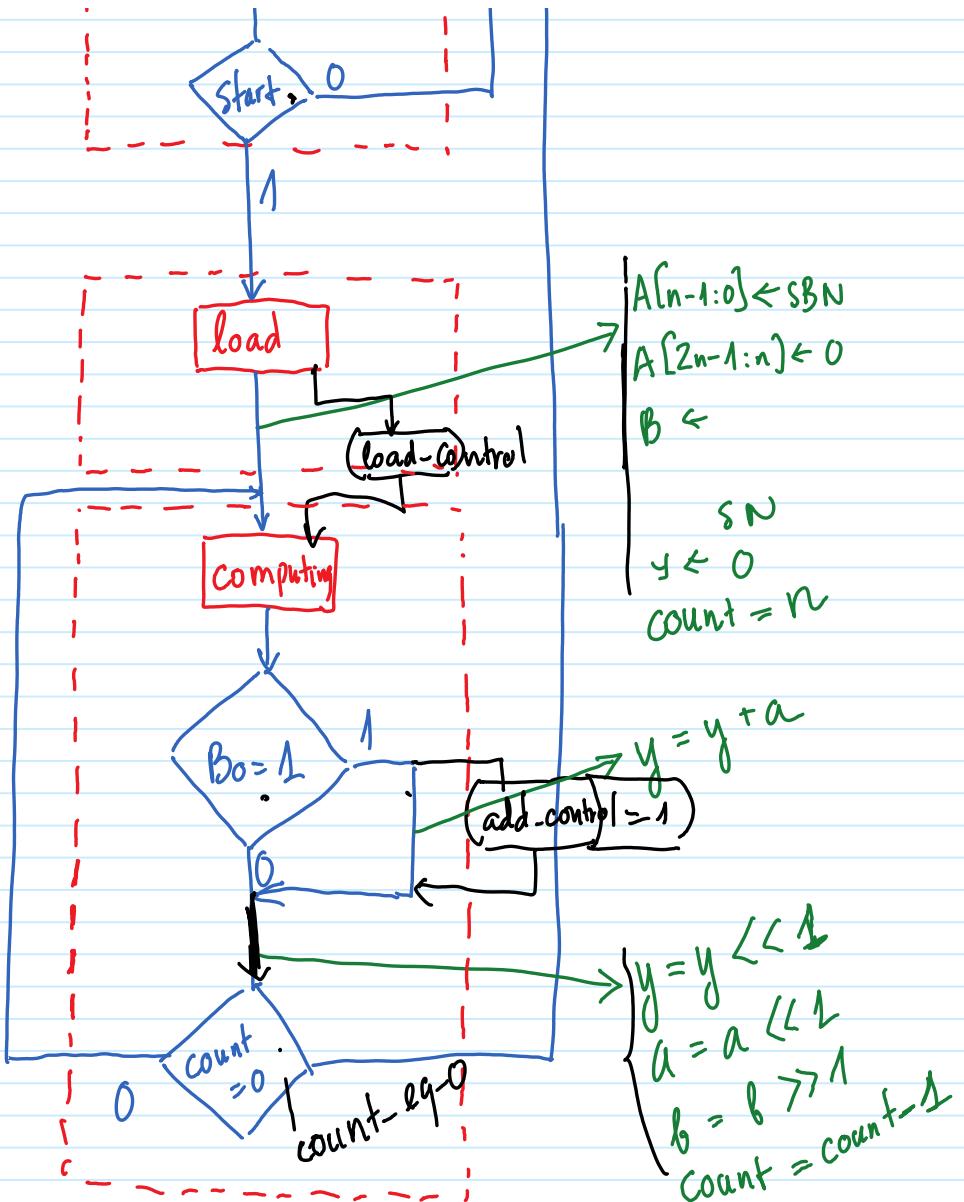
- Khối 1: Trạng thái load
- Khối 2: Trạng thái compute
- Khối 3: Trạng thái idle

B3-2: Xác định các cạnh nối các khối

- Chú ý: Khi nối khối idle và khối load: Khác với phần mềm tự động chạy, phần cứng cần 1 tín hiệu vào để ra lệnh: input start
- Từ khối load sang khối computing: không cần tín hiệu ra lệnh

B3-3: Đánh dấu các cạnh bằng thao tác trên dữ liệu





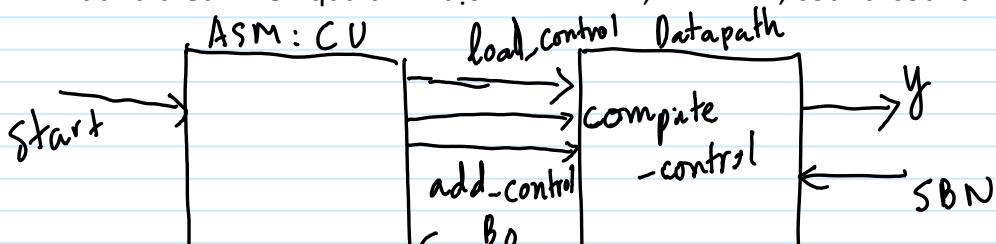
B4: Xác định đầu vào, đầu ra của ASM

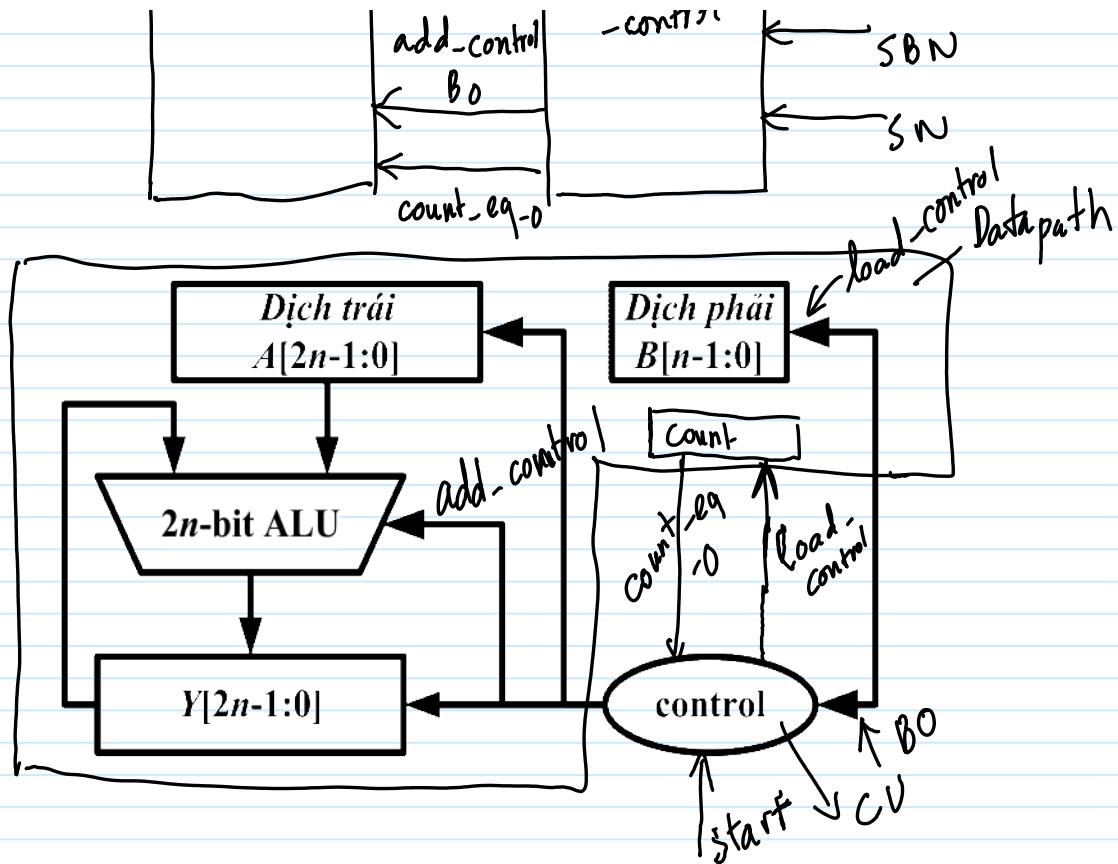
B4-1: Đầu vào

- Đầu vào từ môi trường: start
- Đầu vào là trạng thái của đường dữ liệu: $B0$, $count_eq_0$

B4-2: Đầu ra

- Đầu ra điều khiển quá trình nạp dữ liệu: $load_control$
- Đầu ra điều khiển quá trình cộng $Y = Y + A$: $add_control$
- Đầu ra điều khiển quá trình dịch $A = A \ll 1$; $B = B \gg 1$; $count = count - 1$: $compute_control$





B5. Mô tả mạch bằng Verilog

```

module serial_mult #(parameter n=8)
  (input clk, rst_n,
   input [n-1:0]           SBN,
   input [n-1:0]           SN,
   output [2*n-1:0]        y,
   output done);

  wire adder_control, load_control, compute_control;
  wire b0, count_eq_0;

  data_unit #(.n(n)) datapath
    (.clk(clk),
     .rst_n(rst_n),
     .SBN(SBN), .SN(SN),
     .y(y),
     .load_control(load_control),
     .add_control(add_control),
     .shift_control(shift_control),
     .count_eq_0(count_eq_0),
     .b0(b0)
    );

```

```

control_unit #(.n(n)) control
    (.clk(clk), .rst_n(rst_n),
     .start(start),
     .b0(b0),
     .count_eq_0(count_eq_0), // status signal from datapath
     .load_control(load_control), // control signal to datapath
     .add_control(adder_control),
     .shift_control(compute_control),
     .done(done)
    );
endmodule

module data_unit #(parameter n = 8)
(
    input clk, rst_n,
    input [n-1:0] SBN, SN,
    output reg [2*n-1:0] y,
    input load_control, add_control, shift_control,
    output count_eq_0, b0
);
    reg [n-1:0] b;
    reg [2*n-1:0] a;
    reg [n-1:0] count;

    assign b0 = b[0];
    assign count_eq_0 = (count==0);

    always @ (posedge clk or negedge rst_n)
    begin
        if (~rst_n)
            begin
                a <= 0;
                b <= 0;
                count <= n;
                y <= 0;
            end
        else
            begin
                if (load_control)
                    begin
                        a(n-1:0) <= SBN;
                        b <= SN;
                        count <= n;
                        y <= 0;
                    end
                end
            if (add_control)

```

```

y <= y+a;

if (shift_control)
begin
    b <= b >> 1;
    a <= a << 1;
    count = count -1;
end

end
end
endmodule

```

CHƯƠNG 4: Tổng hợp IC số, hệ thống số (12LT+3BT)

4.1. Khái niệm về tổng hợp logic (Lược đồ Y các biểu diễn mô hình biểu diễn mạch số; Đầu vào, đầu ra, các thành phần và các bước thực hiện cơ bản của phần mềm tổng hợp logic; Thư viện đích và thư viện liên kết trong tổng hợp; Giới thiệu sơ lược về một số kỹ thuật, thuật toán tổng hợp, tối ưu logic) – 1,5 LT

4.2. Tổng hợp mạch logic tổ hợp (Tổng hợp phép gán liên tục; Tổng hợp cấu trúc always và phép gán blocking; Tổng hợp cấu trúc if/case-Khai niệm cấu trúc ưu tiên priority structure; Tổng hợp sử dụng don't care; Chia sẻ tài nguyên trong tổng hợp và cấu trúc Verilog tương ứng; Tổng hợp mạch 3 trạng thái và bus) - 2,5 LT

4.3. Tổng hợp mạch dãy (Tổng hợp phần tử chốt: các lỗi thường gặp với phần tử chốt; Tổng hợp flip-flop; Tổng hợp FSM và các mạch dãy có hoạt động được mô tả bằng khối always được kích hoạt bằng 1 sườn đồng hồ; Tổng hợp mạch dãy có hoạt động được mô tả trong nhiều chu kỳ đồng hồ; Tổng hợp thanh ghi; Tín hiệu reset; Điều khiển tín hiệu đồng hồ) – 4LT

4.4. Mô tả và tổng hợp mạch bằng cấu trúc lặp trong Verilog (Vòng lặp tĩnh không có điều khiển thời gian – logic tổ hợp; Vòng lặp tĩnh có điều khiển thời gian – mạch dãy một chu kỳ, đa chu kỳ; Vòng lặp động có điều khiển thời gian – mạch dãy một chu kỳ, đa chu kỳ; Vòng lặp động không có điều khiển thời gian – không tổng hợp; Cấu trúc generate) – 3 LT

4.5. Thiết kế và tổng hợp mạch phức tạp (Kỹ thuật chia để trị - thiết kế sơ đồ khối của hệ thống; Tái sử dụng thiết kế có sẵn và yêu cầu cần thiết; Khái niệm về nhân sở hữu trí tuệ; Tham số hóa module Verilog; Hàm và thủ tục trong Verilog) – 1 LT

Bài tập thực hành số 1: Thiết kế mạch điều khiển thang máy

Specification:

- Input:
 - Nút tại các tầng
 - input [5:1] up, down
 - Nút trong buồng thang
 - input keep_close, keep_open;
 - input [5:1] input_floor;
- Output

- output door; // = 1 mở, = 0 đóng
- output go_up, go_down; // = 10 đi lên, = 01 đi xuống, 11, 00 dừng - điều khiển đèn led hiển thị hướng đi ở từng tầng
- output [5:1] led_up, led_down; // đèn led nút bấm ngoài từng tầng
- output [2:0] current_floor; // điều khiển đèn led hiện thị số tầng
- output [5:1] led_floor; // đèn led nút bấm floor trong buồng thang
- Bài tập: Vẽ lưu đồ thuật toán mô tả hoạt động của thang máy. Chuyển đổi lưu đồ thuật toán thành ASMD. Xây dựng kiến trúc mạch gồm Control_Unit và Datapath

Bài tập tự chọn (+10): Tối ưu bộ nhân nối tiếp sao cho kích thước là nhỏ nhất.