29/10/2013

Tuesday, October 29, 2013 9:22 AM

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_biến;

    begin
        .....các_lệnh....

        tên_hàm = giá_trị_trả_về;
    end
endfunction
```

  - Ví dụ

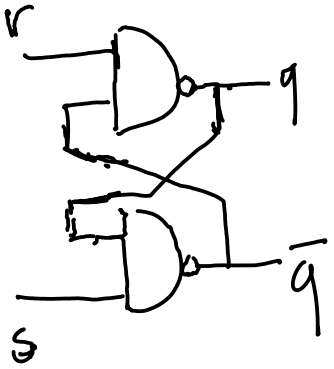
```
module adder4bit (
    input [3:0] a, b,
    input ci,
    output [3:0] s,
    output co);

    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 nhậy 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, sường dương		

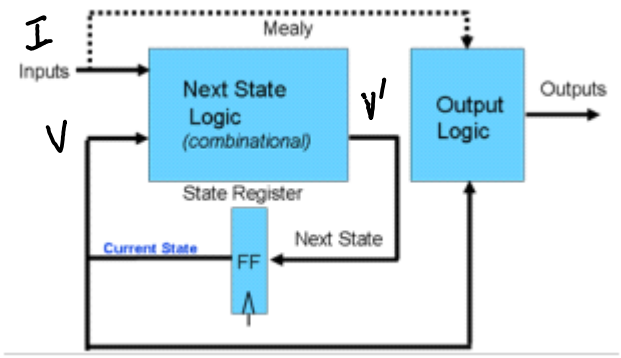
- Chú ý: Khi mô tả flip-flop cần chú ý tới các phần tử có trong thư viện chuẩn. Phần mềm tổng hợp sử dụng các phần tử có sẵn => một số loại flip-flop sẽ cần 2 phần tử (hoặc nhiều hơn) từ thư viện chuẩn => có kích thước lớn hơn. Nên sử dụng loại flip-flop có hỗ trợ trong thư viện chuẩn (kích thước mạch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

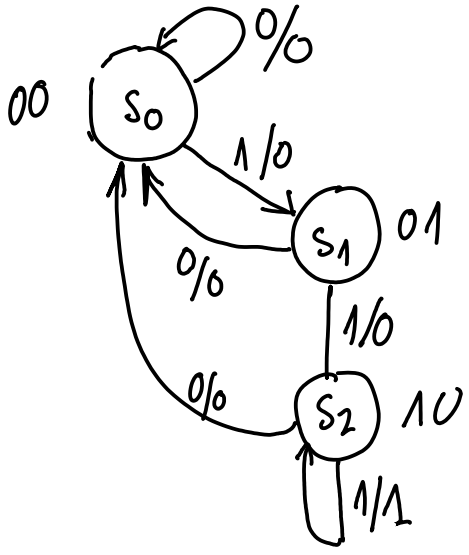
- Mạch tổ hợp triển khai hàm chuyển trạng thái Delta có đầu vào là đầu vào của mạch I và biến trạng thái hiện tại V, đầu ra là biến trạng thái kế tiếp của mạch
- Các phần tử nhớ (flip-flop) lưu biến trạng thái của mạch (Đầu vào D nối với V', đầu ra Q nối với V)
- Mạch tổ hợp triển khai hàm đầu ra Lamda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