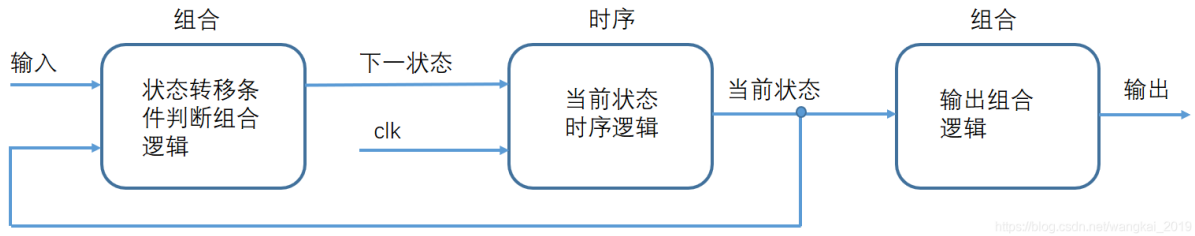
FSM Design

Basic Knowledge of FSM

> FROM: https://blog.csdn.net/wangkai_2019/article/details/1076921

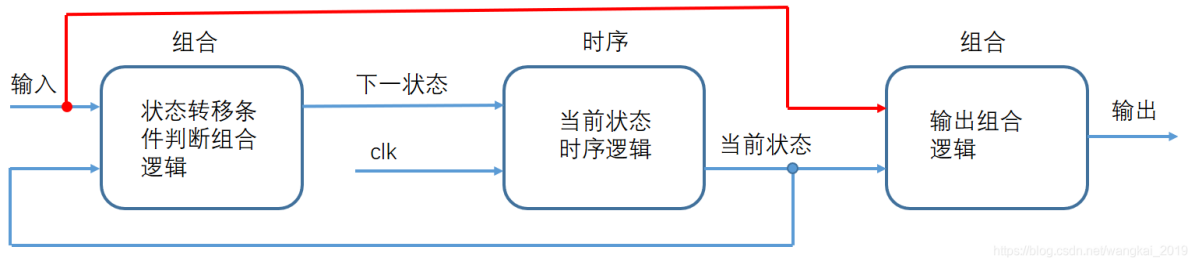
Moore FSM

状态机的输出只与当前的状态有关



Mealy FSM

状态机的输出不仅与当前的状态有关，还与当前的输入有关



Comparison

From UCB EECS150 Tutorial:

Moore machines are very useful because their output signals are **synchronized with the clock**. No matter when input signals reach the Moore Machine, its output signals will not change until the rising edge of the next clock cycle. This is very important to **avoid setup timing violations.**

For example, if a **Mealy machine**'s input signal(s) changes sometime in the middle of a clock cycle, one or more of its outputs and next state signals may change some time later. "Some time later" might come after the setup time threshold for the next rising edge.

However, this means, sometimes the Moore machine will **require more states** to specify its function than the Mealy machine

Transformation between Mealy and Moore

> https://blog.csdn.net/gordon_77/article/details/79423176
>
> **http://catonblack.cn/2019-01-18/mealy2moore/**

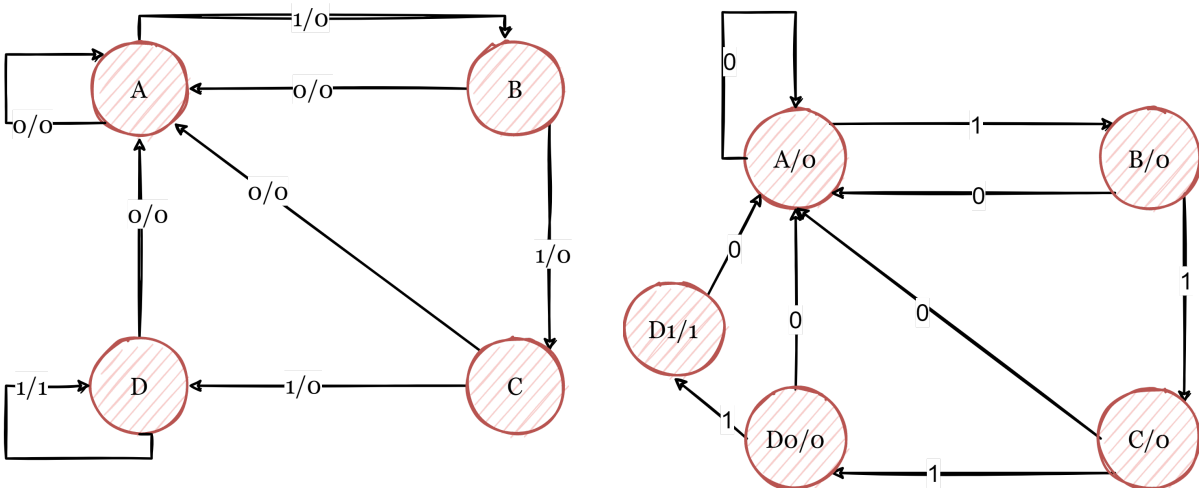
Moore to Mealy

- 把次态的输出修改为对应现态的输出,
- 同时合并一些具有等价性能的状态

Mealy to Moore

- 把当前态的输出修改为对应次态的输出,
- 同时添加一些状态

Example



Here is a example of transformation. The upper one is Mealy and the bottom one is Moore. We will show how to transform from Mealy to Moore in detail.

1. **In Mealy, for example state `A`, the possible output at state `A` are all zero (because input transition with output 0 and output transition with output 0) , so state `A` do not need to be divided. we just keep it, move output into circle and delete all output in all transitions to and from A.**
2. **In Mealy, for example state `D`, the possible output at state `D` are 0 and 1. So we need to divide it into two states `D0` and `D1`**

Design FSM in Hardware Description Language

We can divide design and implement FSM in a HDL into four steps (Moore FSM):

1. **A state encoding for each state**
2. **A mechanism for keeping track of the current state**
3. **Transitions from state to state**
4. **Output values based on the current state**

Design FSM in Verilog

State Encoding

Use `parameter` or `localparam`.

Program 2 The state encoding (in binary)

```
localparam STATE_Initial = 3'b000,
           STATE_1 = 3'b001,
           STATE_2 = 3'b010,
           STATE_3 = 3'b011,
           STATE_4 = 3'b100;
```

Encoding Bits not fully used

As 3 bits can specify a total of 8 states (0-7), our encoding specifies **3 potential states not specified** as being actual states. There are several ways of dealing with this problem:

1. **Ignore it, and always press `Reset` as a way of initializing the FSM**
2. **Specify these states, and make non-conditional transitions from them to the `STATE_Initial`**

Keeping Track of the Current State

create a `reg` element of the appropriate width and use its value as our current state

Program 4 Storing the current state in a `reg`

```
reg [2:0] CurrentState;
```

Transition from State to State

Store current state and next state information in `registers`

Program 19 Storing the current state and next state in `reg` elements

```
reg [2:0] CurrentState;
reg [2:0] NextState;
```

we will assign the next state in an `always@( * )` block.

Program 20 The `always@(posedge Clock)` block

```
always@(posedge Clock) begin
    if (Reset) CurrentState <= STATE_Initial;
    else CurrentState <= NextState;
end
```

Specify State Transition Behaviors: notice the `NextState=CurrentState` line

Program 21 The `always@( * )` block

```
always@( * ) begin
    NextState = CurrentState;
    case (CurrentState)
        STATE_Initial: begin
            NextState = STATE_1;
        end
        STATE_1: begin
            if (A & B) NextState = STATE_2;
        end
        STATE_2: begin
            if (A) NextState = STATE_3;
        end
        STATE_3: begin
            if (!A & B) NextState = STATE_Initial;
            else if (A & !B) NextState = STATE_4;
        end
        STATE_4: begin
        end
        // ---------------------------------------
        // Place-holder transitions
        // ---------------------------------------
        STATE_5_PlaceHolder: begin
            NextState = STATE_Initial;
        end
        STATE_6_PlaceHolder: begin
            NextState = STATE_Initial;
        end
        STATE_7_PlaceHolder: begin
            NextState = STATE_Initial;
        end
        // ---------------------------------------
    endcase
end
```

About `always@(*)` block

- **`always@( * )` blocks are used to describe Combinational Logic, or Logic Gates.**
- **Only `=` (blocking) assignments should be used in an `always@( * )` block. Never use `< =` (non-blocking) assignments in `always@( * )` blocks.**
- **Only use `always@( * )` block when you want to infer an element(s) that changes its value as soon as one or more of its inputs change.**
- **Always use `*` (star) for your sensitivity list in `always@( * )` blocks**
- **A very common bug is to introduce an incomplete sensitivity list.**

About Latch Generation

If you don't assign every element that can be assigned inside an `always@( * )` block every time that `always@( * )` block is executed, a **latch** (similar to a register but much harder to work with in FPGAs) will be inferred for that element.

Program 17 An `always@( * )` block that will generate a latch for `C`

```
wire Trigger, Pass;
reg A, C;

always @( * ) begin
    A = 1'b0;
    if (Trigger) begin
        A = Pass;
        C = Pass;
    end
end
```

To fix the problem, **we must make sure that `C` gets set every time the `always@` block is executed**.

- **Default values are an easy way to avoid latch generation, however, will sometimes break the logic in a design.**
- **Know that setting a `reg` to itself is not an acceptable way to ensure that the `reg` always gets set**
- **Each value that is assigned in at least one place must be assigned to a non-trivial value during every 'execution' of the always@(*) block.**

Outputting Values Based on the Current State

Program 22 The outputs from Figure 1

```
wire      Output1, Output2;
reg[2:0] Status;

assign Output1 = (CurrentState == STATE_1) | (CurrentState == STATE_2);
assign Output2 = (CurrentState == STATE_2);

always@( * ) begin
    Status = 3'b000;
    case (CurrentState)
        STATE_2: begin
            Status = 3'b010;
        end
        STATE_3: begin
            Status = 3'b011;
        end
    endcase
end
```

Design FSM in VHDL