Finite State Machine Styles

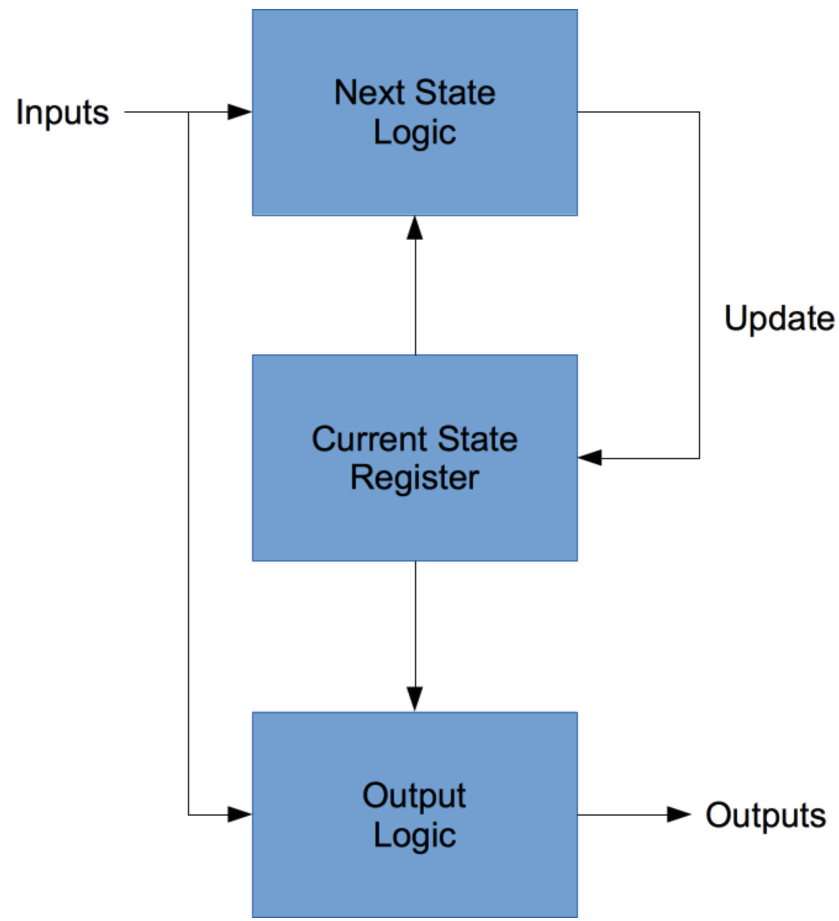
CS152A – 2015 Fall

What’s in an FSM?

- These slides assume that you already know what an FSM is in the general sense.
- In digital design, an FSM has 3 components:
  - Inputs, can be events (pulses) or states (levels)
  - Outputs, can be events (pulses) or states (levels)
  - State transition logic that depend on only inputs and current states

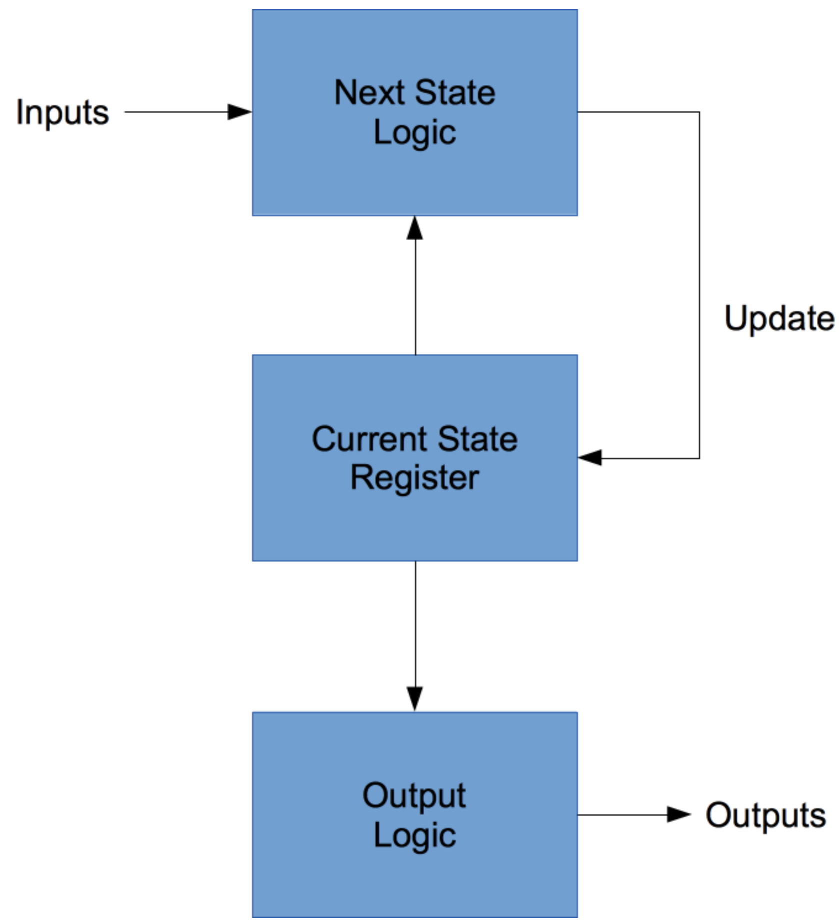
Mealy Machine

- In the conventional sense, a Mealy machine is a state machine whose outputs depend on both the current state and the input states
- A change in input may be immediately reflected in the output



Moore Machine

- In the conventional sense, a Moore machine is a state machine whose outputs only depend on the current state
- Outputs always change one cycle behind input change



Mealy vs Moore

- Advantages of Mealy-style machines
  - Low-latency output
  - Fewer states than equivalent Moore-style machines
- Advantages of Moore-style machines
  - More clean separation of inputs and outputs
  - Can guarantee stability of output
  - Better for static timing

Mealy vs. Moore, cont.

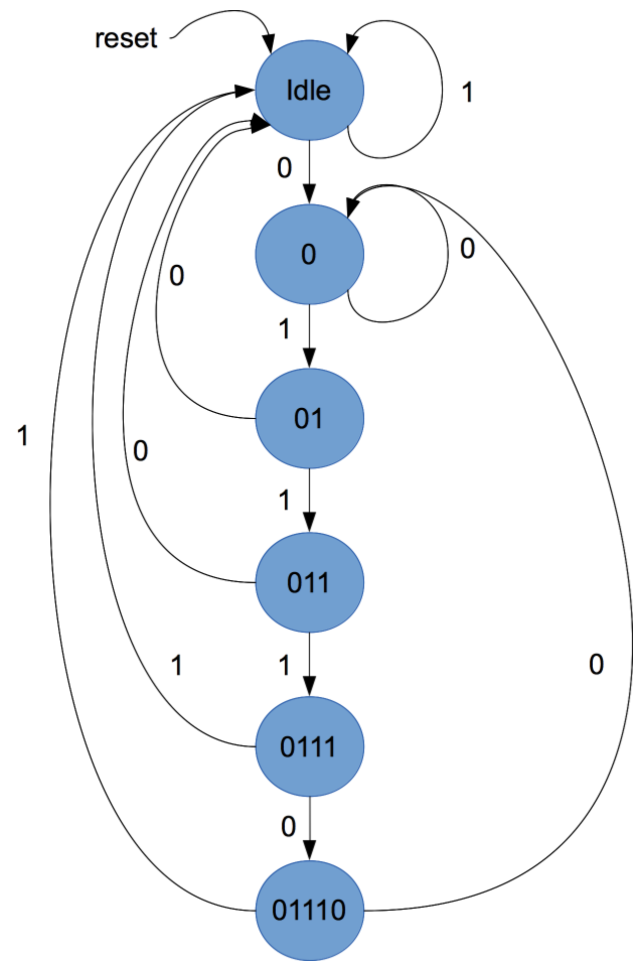
- So when do you use Mealy- or Moore-style FSM?
- Mealy and Moore are mere conceptual boxes that we (engineers) draw on designs
- In reality, few really think about FSMs in terms of Mealy or Moore
- Both will get the job done; in practice, most of our designs are Moore machines

FSM Design Tips

- Start your FSM design by first drawing the STD (State Transition Diagram) on paper
- One you are convinced that your STD achieves yours design goal, translate it to HDL
- Typical FSM has three conceptual blocks:
  – A combinational block that computes the next state
  – A sequential block that builds the state register
  – A combinational/sequential block that generates the output
- In reality many designers choose to combine these blocks to one or two blocks

Verilog Example

- Design a state machine that recognize the following input sequence: 01110
- Inputs: i_data
- Output: o_valid (pattern detected)
- States: idle, 0, 01, 011, 0111, 01110
- Is this a Mealy or Moore machine?



Part I – declarations

```
output o_valid;
input i_data;
input clk;
input srst;

parameter stIdle  = 0;
parameter st0     = 1;
parameter st01    = 2;
parameter st011   = 3;
parameter st0111  = 4;
parameter st01110 = 5;

reg [2:0] cur_state;
reg [2:0] nxt_state;
```

- Why are the width of cur_state and nxt_state 3 bits wide?
- cur_state is a sequential logic element that stores the current state
- nxt_state is a combinational variable, not registerd

Part II – State Register Block

Synchronous reset example:

```
always @ (posedge clk)
 if (srst)
  cur_state <= stIdle;
 else
  cur_state <= nxt_state;
```

Asynchronous reset example:

```
always @ (posedge clk or posedge arst)
 if (arst)
  cur_state <= stIdle;
 else
  cur_state <= nxt_state;
```

Part III – Next State Logic

```
always @*
 case (cur_state)
  stIdle  : nxt_state = i_data ? stIdle : st0;
  st0     : nxt_state = i_data ? st01 : st0;
  st01    : nxt_state = i_data ? st011 : stIdle;
  st011   : nxt_state = i_data ? st0111 : stIdle;
  st0111  : nxt_state = i_data ? stIdle : st01110;
  st01110 : nxt_state = i_data ? stIdle : st0;
  default : nxt_state = cur_state;
 endcase
```

Think: what would happen if the default clause was left out?

Part III – Next State Logic

```
always @*
 case (cur_state)
  stIdle  : nxt_state = i_data ? stIdle : st0;
  st0     : nxt_state = i_data ? st01 : st0;
  st01    : nxt_state = i_data ? st011 : stIdle;
  st011   : nxt_state = i_data ? st0111 : stIdle;
  st0111  : nxt_state = i_data ? stIdle : st01110;
  st01110 : nxt_state = i_data ? stIdle : st0;
  default : nxt_state = cur_state;
 endcase
```

Think: what would happen if the default clause was left out?

Answer: a nxt_state latch will be created!

Part IV – Output Logic Block

```
assign o_valid = (cur_state==st01110);
```

That's it!

What would a Mealy machine's output look like?

```
assign o_valid = (cur_state==st0111) & ~data_i;
```

Combining Current and Next State Block

```
always @ (posedge clk)
 if (srst)
  cur_state <= stIdle;
 else
  case (cur_state)
   stIdle  : cur_state <= i_data ? stIdle : st0;
   st0     : cur_state <= i_data ? st01   : st0;
   st01    : cur_state <= i_data ? st011  : stIdle;
   st011   : cur_state <= i_data ? st0111 : stIdle;
   st0111  : cur_state <= i_data ? stIdle : st01110;
   st01110 : cur_state <= i_data ? stIdle : st0;
  endcase
```

Think: why don’t we need the default clause any more?

Combining Current and Next State Block

```
always @ (posedge clk)
 if (srst)
  cur_state <= stIdle;
 else
  case (cur_state)
   stIdle  : cur_state <= i_data ? stIdle : st0;
   st0     : cur_state <= i_data ? st01   : st0;
   st01    : cur_state <= i_data ? st011  : stIdle;
   st011   : cur_state <= i_data ? st0111 : stIdle;
   st0111  : cur_state <= i_data ? stIdle : st01110;
   st01110 : cur_state <= i_data ? stIdle : st0;
  endcase
```

Think: why don’t we need the default clause any more?

Answer: cur_state is already a sequential register

State Register Encoding

Suppose the state encoding was the following:

```
parameter stIdle  = 4'b0000;
parameter st0     = 4'b0001;
parameter st01    = 4'b0010;
parameter st011   = 4'b0011;
parameter st0111  = 4'b0100;
parameter st01110 = 4'b1000;
```

If we assume all other states are illegal states (not possible to obtain), how would this encoding simplify the output logic?

How would this encoding impact the next state logic?

State Register Encoding

Suppose the state encoding was the following:

```
parameter stIdle  = 4'b0000;
parameter st0     = 4'b0001;
parameter st01    = 4'b0010;
parameter st011   = 4'b0011;
parameter st0111  = 4'b0100;
parameter st01110 = 4'b1000;
```

If we assume all other states are illegal states (not possible to obtain), how would this encoding simplify the output logic?

assign o_valid = cur_state[3]; // output is simple!

How would this encoding impact the next state logic?

Need to decode 4 bits at a time instead of 3 bits (more gates)

Commonly Used Encoding Styles

- Binary (00,01,10,11, …), most commonly used
- Gray (00,01,11,10, …), only one bit is changed between two adjacent values
- One-hot (0001, 0010, 0100, 1000), only one bit is "hot" for each state
- Parity (001, 010, 100, 111, …), parity of all legal states is always 1
- Each encoding style has its own merits, discussion of the merits are however outside of this lecture