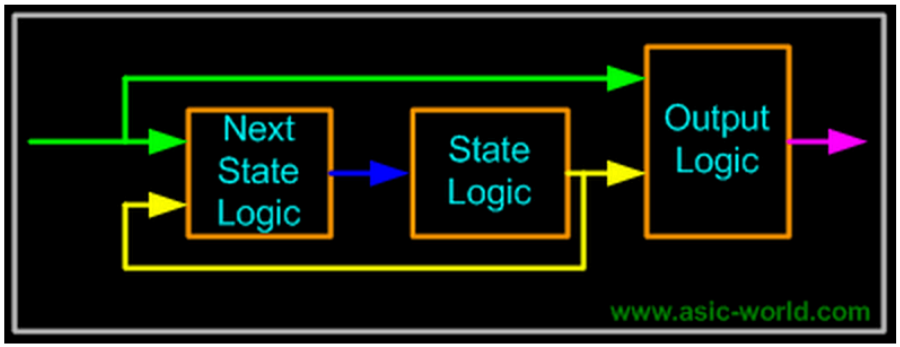
EE141 – Discussion #3

Nathan Narevsky

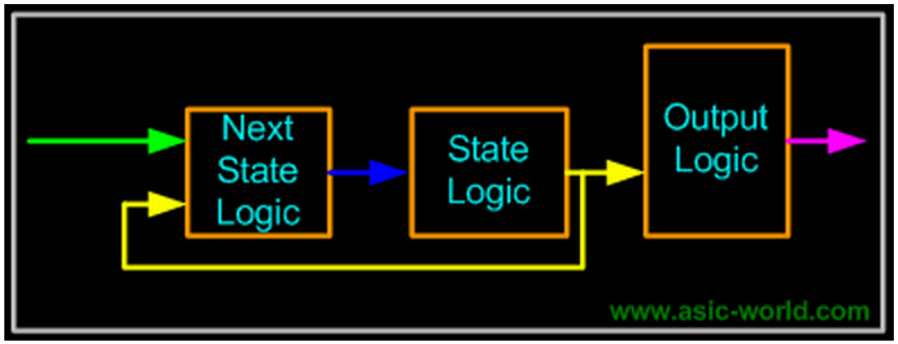
9/19/2014 and 9/22/2014

Mealy vs. Moore

Mealy Model



Moore Model



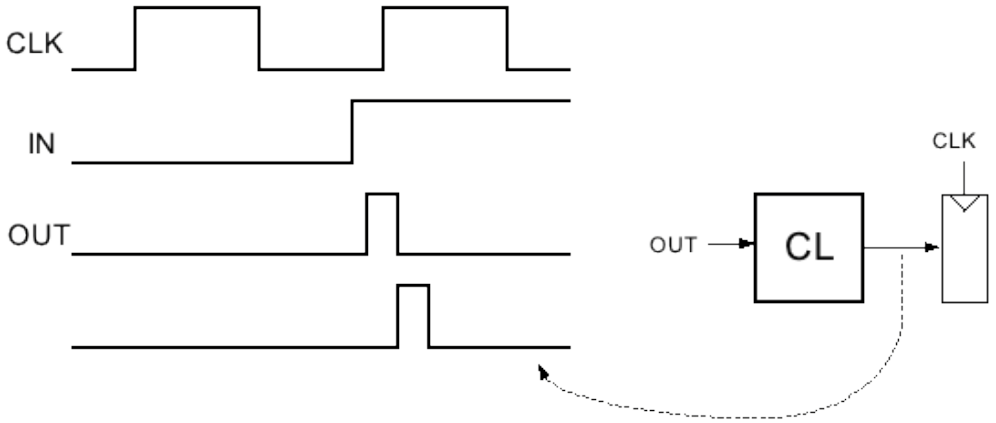
FSM Comparison

Solution A
Moore Machine

- output function only of PS
- maybe more states (why?)
- synchronous outputs
  - no glitches
  - one cycle "delay"
  - full cycle of stable output

Solution B
Mealy Machine

- output function of both PS & input
- maybe fewer states
- asynchronous outputs
  - if input glitches, so does output
  - output immediately available
  - output may not be stable long enough to be useful (below):



If output of Mealy FSM goes through combinational logic before being registered, the CL might delay the signal and it could be missed by the clock edge.

More on Generate Loop

Permits variable declarations, modules, user defined primitives, gate primitives, continuous assignments, initial blocks and always blocks to be instantiated multiple times using a for-loop.

```
// Gray-code to binary-code converter
module gray2bin1 (bin, gray);
    parameter SIZE = 8;
    output [SIZE-1:0] bin;
    input  [SIZE-1:0] gray;

    genvar i;

    generate for (i=0; i<SIZE; i=i+1) begin:bit
        assign bin[i] = ^gray[SIZE-1:i];
    end endgenerate
endmodule
```

variable exists only in the specification - not in the final circuit.

Keywords that denotes synthesis-time operations

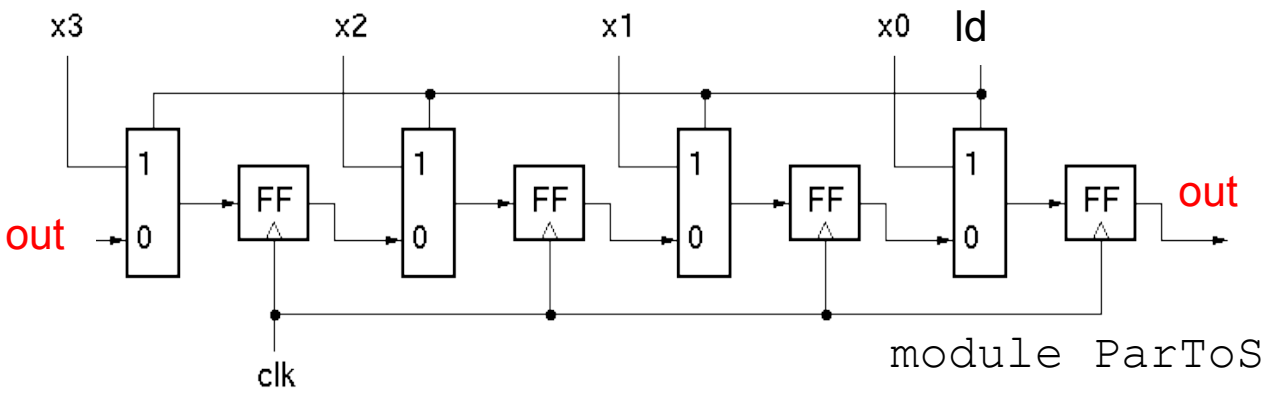
For-loop creates instances of assignments

Loop must have constant bounds

generate if-else-if based on an expression that is deterministic at the time the design is synthesized.

generate case : selecting case expression must be deterministic at the time the design is synthesized.

Example - Parallel to Serial Converter




```
module ParToSer(ld, X, out, clk);
   input [3:0] X;
   input ld, clk;
   output out;

   reg [3:0] Q;
   wire [3:0] NS;

   assign NS =
     (ld) ? X : {Q[0], Q[3:1]};

   always @ (posedge clk)
      Q <= NS;

   assign out = Q[0];
endmodule
```

Specifies the muxing with "rotation"

forces Q register (flip-flops) to be rewritten every cycle

connect output

Parameterized Version

Parameters give us a way to generalize our designs. A module becomes a "generator" for different variations. Enables design/module reuse. Can simplify testing.

```
parameter N = 4;
```

Declare a parameter with default value.

Note: this is not a port. Acts like a "synthesis-time" constant.

```
ParToSer #(.N(8))
   ps8 ( ... );

ParToSer #(.N(64))
ps64 ( ... );
```

Overwrite parameter N at instantiation.

```
module ParToSer(ld, X, out, CLK);
  input [N-1:0] X;
  input ld, clk;
  output out;
  reg out;
  reg [N-1:0] Q;
  wire [N-1:0] NS;

  assign NS =
    (ld) ? X : {Q[0], Q[N-1:1]};

  always @ (posedge clk)
     Q <= NS;

  assign out = Q[0];
endmodule
```

Replace all occurrences of "3" with "N-1".

Incomplete Triggers

Leaving out an input trigger usually results in latch generation for the missing trigger.

```
module and_gate (out, in1, in2);
  input    in1, in2;
  output       out;
  reg          out;

  always @(in1) begin
     out = in1 & in2;
  end

endmodule
```

in2 not in always sensitivity list.

A latched version of in2 is synthesized and used as input to the and-gate, so that the and-gate output is not always sensitive to in2.

Easy way to avoid incomplete triggers for combinational logic is with: `always @*`

Example: Ripple Carry Adder

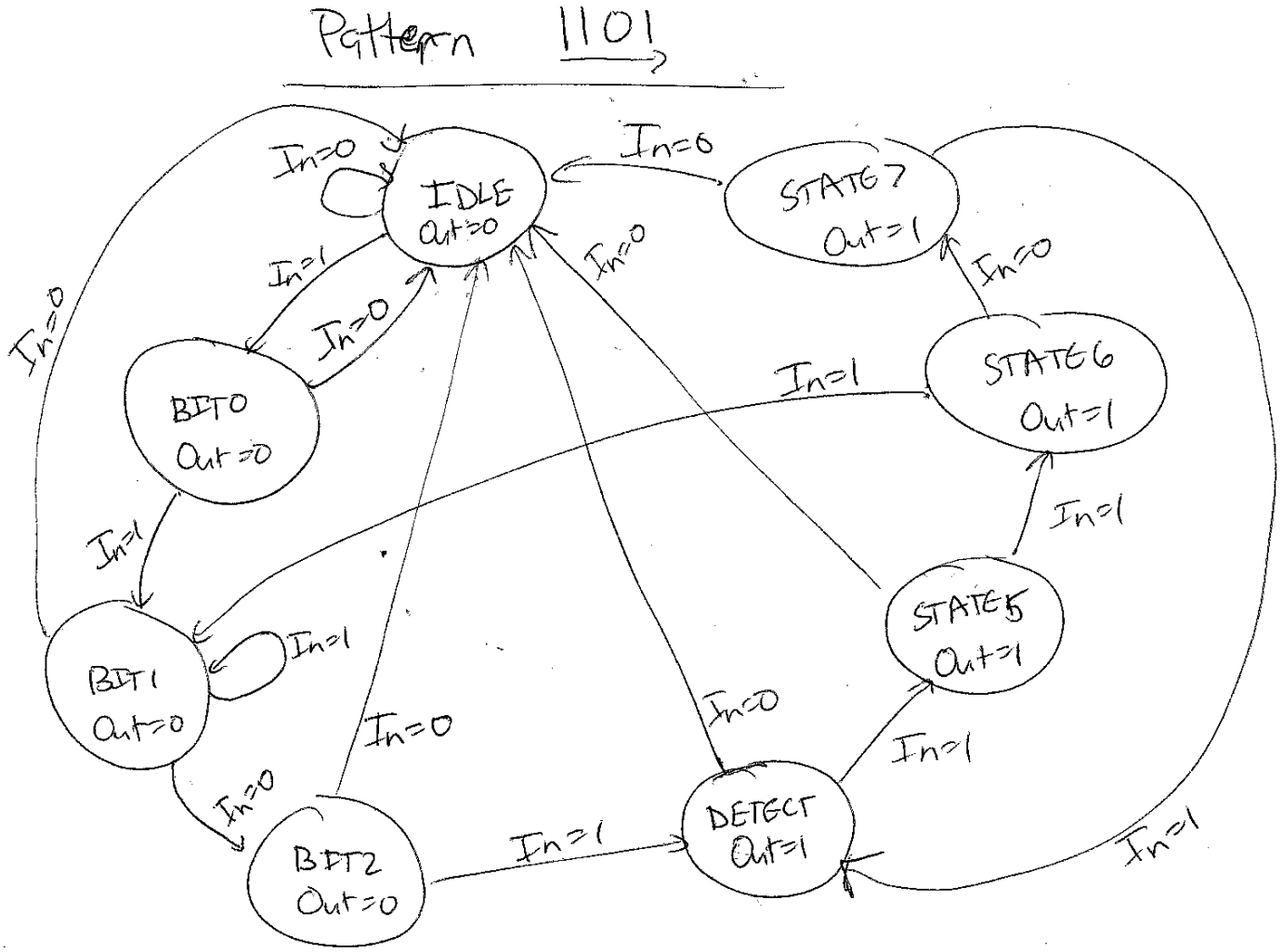
```
module ripple_adder (
   input [15:0] A,
   input [15:0] B,
   output [16:0] Sum
);

wire [16:0] carry;

genvar i;
generate for (i=0; i < 16; i=i+1) begin:foo
   full_adder u0 (.A(A[i]), .B(B[i]), .Cin(carry[i]), .Sum(Sum[i]), .Cout(carry[i+1]));
end endgenerate

assign Sum[16] = carry[16];

endmodule
```

Parameterized Ripple Carry Adder

Parametrized version:

```
module ripple_adder #(parameter NUM_BITS = 16 ) (
   input [NUM_BITS-1:0] A,
   input [NUM_BITS-1:0] B,
   output [NUM_BITS:0] Sum
);

wire [NUM_BITS:0] carry;

genvar i;
generate for (i=0; i < NUM_BITS; i=i+1) begin:foo
   full_adder u0 (.A(A[i]), .B(B[i]), .Cin(carry[i]), .Sum(Sum[i]), .Cout(carry[i+1]));
end endgenerate

assign Sum[NUM_BITS] = carry[NUM_BITS];

endmodule
```

Pattern Detection - 1101



Verilog Implementation

1101 Pattern Detector:

```
module pattern_detector (
   input In, clk,
   output Out
);

parameter   IDLE = 3'b000,
          BIT0 = 3'b001,
          BIT1 = 3'b010,
          BIT2 = 3'b011,
          DETECT = 3'b100,
          STATE5 = 3'b101,
          STATE6 = 3'b110,
          STATE7 = 3'b111;

reg [2:0] state, next_state;

always @(posedge clk) begin
   state <= next_state;
end

always @(*) begin
   case(state)
   IDLE: begin
      Out <= 0;
      next_state <= In ? BIT0 : IDLE;
   end
   BIT0: begin
      Out <= 0;
      next_state <= In ? BIT1 : IDLE;
   end
   BIT1: begin
      Out <= 0;
      next_state <= In ? BIT1 : BIT2;
   end
   BIT2: begin
      Out <= 0;
      next_state <= In ? DETECT : IDLE;
   end
   DETECT: begin
      Out <= 1;
      next_state <= In ? STATE5 : IDLE;
   end
   STATE5: begin
      Out <= 1;
      next_state <= In ? STATE6 : IDLE;
   end
   STATE6: begin
      Out <= 1;
      next_state <= In ? BIT1 : STATE7;
   end
   STATE7: begin
      Out <= 1;
      next_state <= In ? DETECT : IDLE;
   end
   endcase
end
endmodule
```

References

- http://www.sunburst-design.com/papers/CummingsSNUG2003SJ_SystemVerilogFSM.pdf
- http://www.asic-world.com/