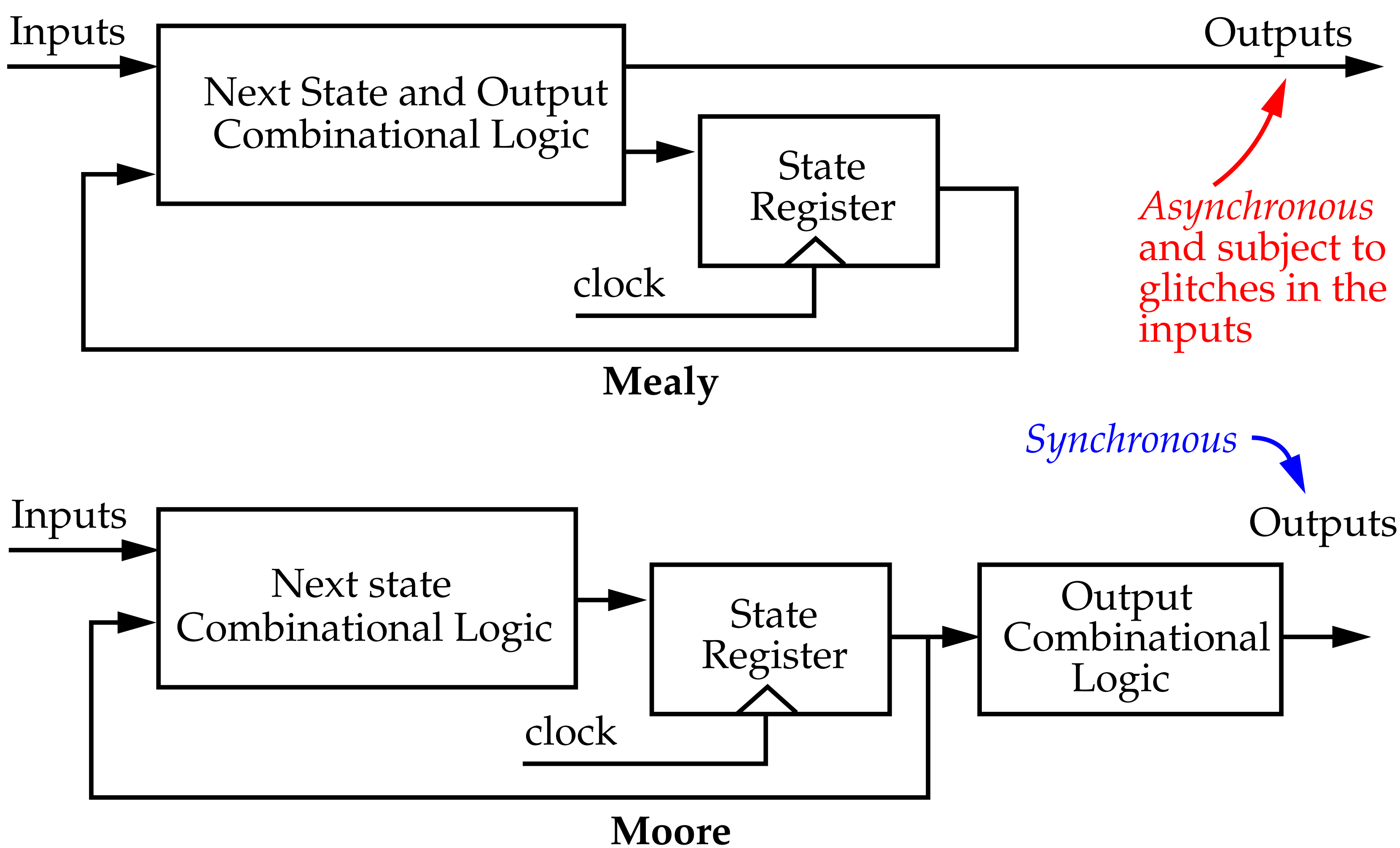
Behavioral Models of FSMs

Two basic forms of Finite State Machines



Behavioral Models of FSMs

There are two descriptive styles of FSMs.

- *Explicit*: declares a state register to encode the machine's state. A behavior **explicitly** assigns values to the state register to govern the state transitions.
- *Implicit*: uses multiple event controls within a cyclic behavior to implicitly describe an evolution of states.

Explicit FSMs, several styles are possible:

```
module FSM_style1 (...);
   input ...;
   output ...;
   parameter size = ...;
   reg [size-1 : 0] state, next_state;

   assign the_outputs = ... // a function of state and inputs
   assign next_state = ...  // a function of state and inputs.

   always @ (negedge reset or posedge clk)
      if (reset == 1'b0) state <= start_state;
      else state <= next_state;
endmodule
```

FSMs

A second style replaces the continuous assignment generating the *next_state* with asynchronous (combinational) behavior:

```
module FSM_style2 (...);
   input ...;
   output ...;
   parameter size = ...;
   reg [size-1 : 0] state, next_state;

   assign the_outputs = ... // a function of state and inputs

   always @ ( state or the_inputs )
      // decode next_state with case or if stmt

   always @ (negedge reset or posedge clk)
      if (reset == 1'b0) state <= start_state;
      else state <= next_state; //Non-blocking or procedural assignment
endmodule
```

This latter style can exploit the **case** stmt and other procedural constructs for descriptions that are complex.

Note that in both styles, the outputs are *asynchronous*.

FSMs

It may be desired to *register* the outputs, and make them *synchronous*:

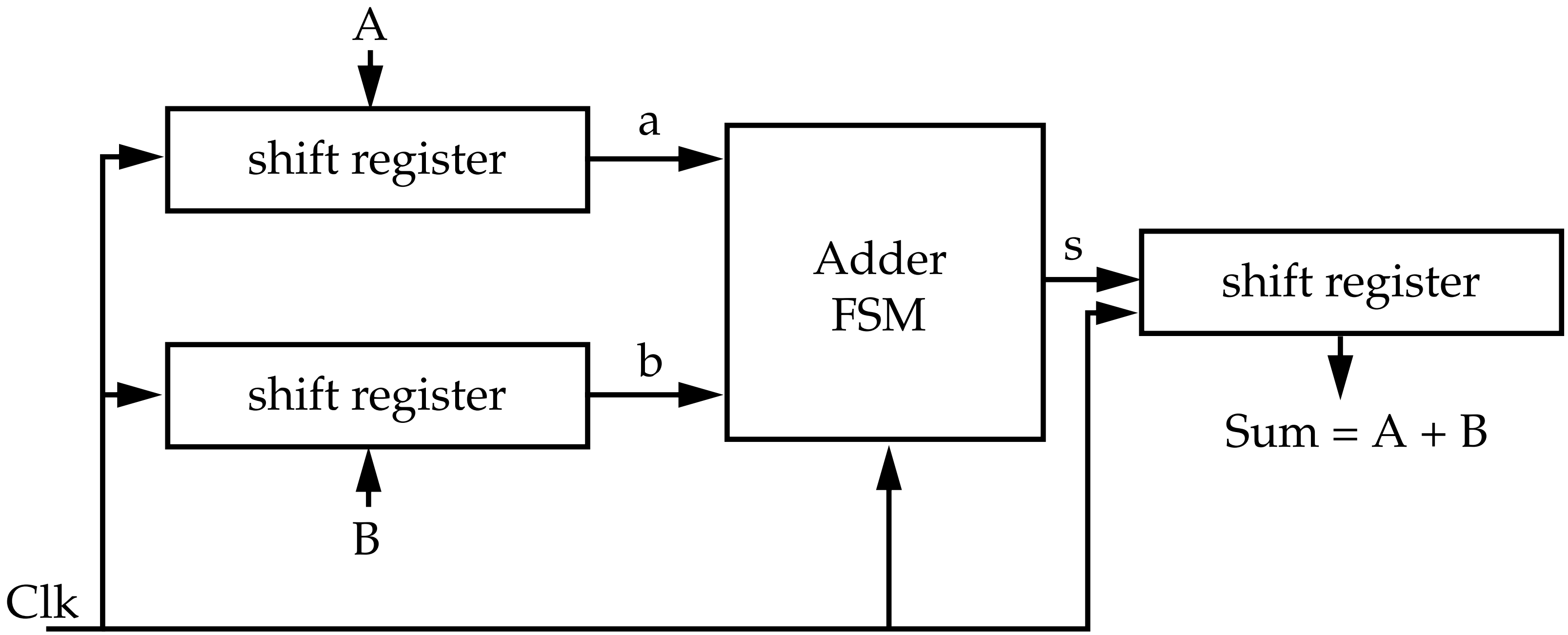
```
module FSM_style3 (...);
   input ...;
   output ...;
   parameter size = ...;
   reg [size-1 : 0] state, next_state;

   always @ ( state or the_inputs )
      // decode next_state with case or if stmt

   always @ (negedge reset or posedge clk)
      if (reset == 1'b0) state <= start_state;
      else begin
         state <= next_state;
         outputs <= some_value (inputs, next_state);
      end
endmodule
```

State machines can be represented in

- Tabular format (state transition table)
- Graphical format (state transition graph)
- Algorithmic state machine (ASM) chart

FSMs: Serial Adder

Adds operands $A = a_{n-1}a_{n-2}...a_0$ and $B = b_{n-1}b_{n-2}...b_0$, one bit pair at a time.



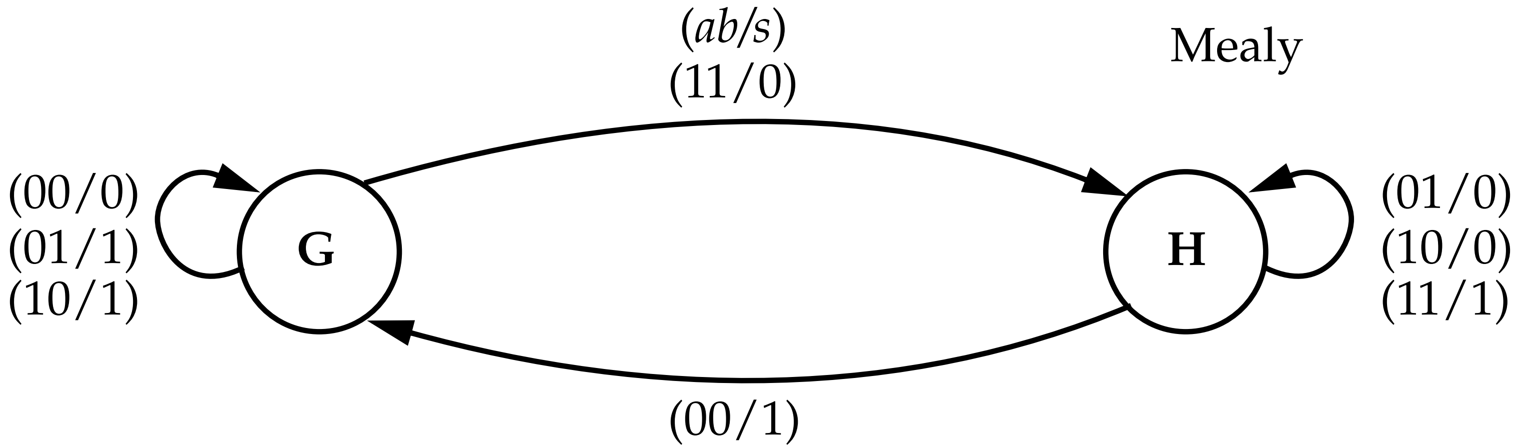
The values of *A* and *B* are loaded in parallel mode into the shift registers.

At each rising edge, the contents of all shift registers are shifted to the right one bit.

This saves the current bit-pair sum, *s*, and fetches the next pair of bits for the adder.

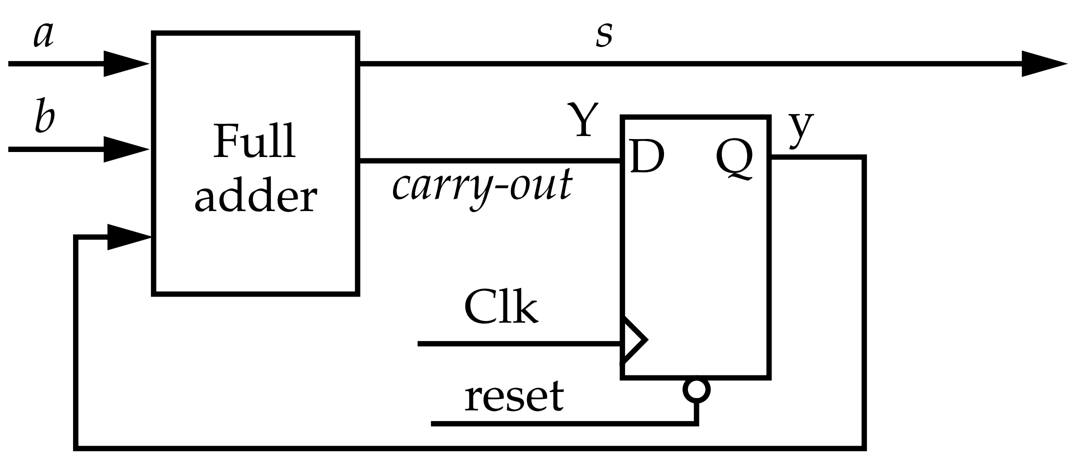
FSMs: Serial Adder: Mealy version

Two states will be used, *G* and *H*, to handle the *carry* bit alternatives.



Only one FF needed.

Output depends on both the state and present value of *a* and *b*.



FSMs: Serial Adder: Mealy version

Shift register with enable:

```
module shift_reg(in_reg, par_load, enable, in_bit, Clk, out_reg);
  parameter n = 8;
  input [n-1:0] in_reg;
  input par_load, enable, in_bit, Clk;
  output reg [n-1:0] out_reg;

  integer k;

  always @ (posedge Clk)
    if (par_load)                    // Parallel load
      out_reg <= in_reg;
    else if (enable)                 // Shift when enabled
    begin
      for (k = n-1; k > 0; k = k - 1)
        out_reg[k-1] <= out_reg[k];
      out_reg[n-1] <= in_bit;

    end
endmodule
```

FSMs: Serial Adder: Mealy version

Serial Adder:

```
module serial_adder(A, B, Reset, Clk, Sum);
   input [7:0] A, B;
   input Reset, Clk
   output wire [7:0] Sum;
   reg [3:0] Cnt;
   reg sbit, cur_state, next_state;
   wire [7:0] QA, QB;
   wire Run;
   parameter G = 1'b0, H = 1'b1;

   shift_reg shift_A(A, Reset, 1'b1, 1'b0, Clk, QA);
   shift_reg shift_B(B, Reset, 1'b1, 1'b0, Clk, QB);
   shift_reg shift_sum(8'b0, Reset, Run, sbit, Clk, Sum);
```

Instantiates three shift registers -- are loaded in parallel when *Reset* asserted.

The *sum* (third) shift_reg shifts when *Run* == 1 (drives *enable* in shift_reg), which happens on the first Clk AFTER *Reset* == 0.

This allows output combo logic (next slide) time to compute *s*.

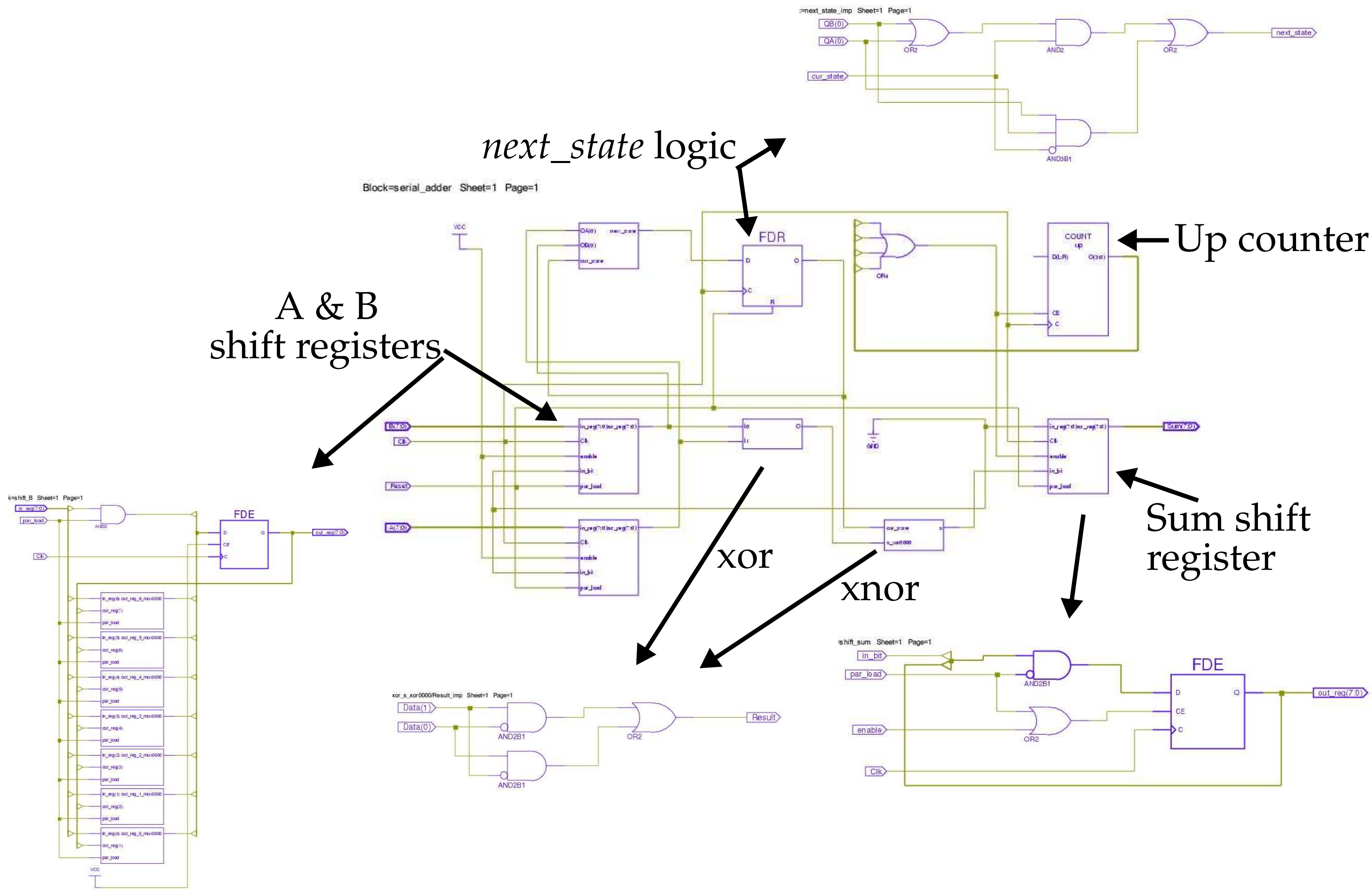
FSMs: Serial Adder: Mealy version

Serial Adder:

```
always @(QA, QB, cur_state)  // Output and next state combo logic
  case (cur_state)                              // carry == 0
    G:  begin
          sbit = QA[0] ^ QB[0];                 // Compute sum: a xor b
          if (QA[0] & QB[0]) next_state = H;    // carry = a and b
          else next_state = G;
        end
    H:  begin                                   // carry == 1
          sbit = QA[0] ~^ QB[0];      // s is 1 for ab = 00 or 11 (xnor)
          if (~QA[0] & ~QB[0]) next_state = G;  // carry is 0 again
          else next_state = H;                  // only if ab = 00
        end
    default: begin sbit = 0; next_state = G; end
  endcase
always @(posedge Clk)       // Flip-flop y
  if (Reset) cur_state <= G;
  else cur_state <= next_state;
always @(posedge Clk)       // Count down from 8 to 1, once for each bit
  if (Reset) Cnt <= 8;      // Synchronous Reset
  else if (Run) Cnt <= Cnt - 1;
assign Run = |Cnt;          // Run = 1 immediately AFTER first Clk
endmodule                   // Run = 0 after 8 more cycles (reduction or)
```

FSMs: Serial Adder: Mealy version

Schematics:



FSMs: Arbiter Circuit: Moore version

The function of a arbiter is to control access by devices to a shared resource. One one device can use the resource at a time.

All signals change only on the positive edge of Clk.

Each device has one input to the FSM, called a *request*, and the FSM produces a separate output for each device called a *grant*.

Devices request service by asserting its *request* signal, and indicates completion by deasserting the *request* signal.

The FSM grants access according to a **priority** scheme (assuming the shared resource is not already allocated to another device).

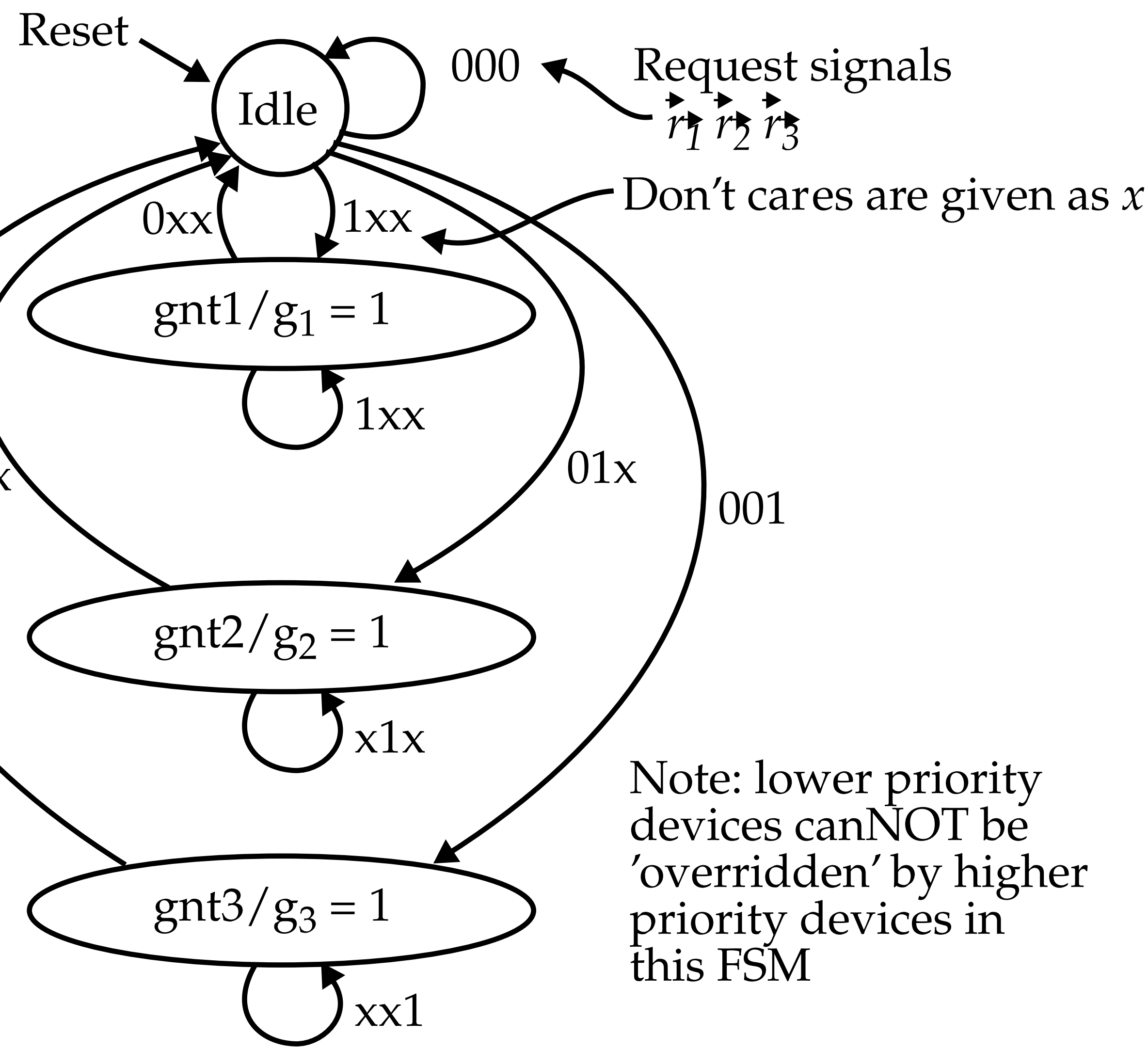
Consider a system designed to handle 3 devices, dev_1, dev_2 and dev_3, in order of decreasing priority, i.e., dev_1 has highest priority.

Let r_x represent the request signals and g_x represent the grant signals

FSMs: Arbiter Circuit: Moore version

State diagram

4 states:
Idle, gnt1, gnt2, gnt3



Note: lower priority devices canNOT be 'overridden' by higher priority devices in this FSM

FSMs: Arbiter Circuit: Moore version

```
module arbiter_moore(r, Resetn, Clk, g);
   input [1:3] r;
   input Resetn, Clk
   output wire [1:3] g;
   reg [2:1] y, Y;
   parameter Idel = 2'b00, gnt1 = 2'b01, gnt2 = 2'b10, gnt3 = 2'b1 1;
   always @(r, y)
      case (y)
         Idle: casex (r)                    // Nested casex which uses x to
                  3'b000: Y = Idle;         // indicate don't care.
                  3'b1xx: Y = gnt1;         // Order of cases listed defines
                  3'b01x: Y = gnt2;         // priority
                  3'b001: Y = gnt3;
                  default: Y = Idle;
               endcase
         gnt1: if (r[1]) Y = gnt1;
               else Y = Idle;
         gnt2: if (r[2]) Y = gnt2;
               else Y = Idle;
         gnt3: if (r[3]) Y = gnt3;
               else Y = Idle;
         default: Y = Idle;
      endcase
```

FSMs: Arbiter Circuit: Moore version

Note this specification allows potential 'starvation' of lower priority devices.

```
always @(posedge Clk)
   if (Resetn == 0) y <= Idle;
   else y <= Y;

always g[1] = (y == gnt1);
assign g[2] = (y == gnt2);
assign g[3] = (y == gnt3);
endmodule
```

Algorithmic state machines (ASMs) are more convenient for complex state machines.

They use a *flow chart* style to show the evolution of states on the application of input data over time.

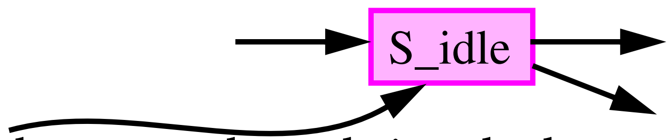
ASMs use three elements:

- *State box*: rectangular boxes that represent the state of the machine between clk events.

FSMs: Algorithmic state machines

- *State box* (cont.):

  State name appears inside the box:

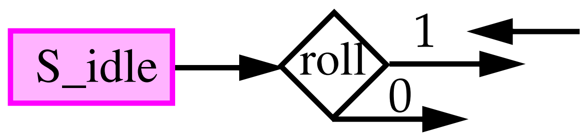


To one or more *decision* boxes

Listed: state code and signals that are asserted when state box is entered.

For Moore machines, the state name appears on the outside in the upper left and the asserted outputs are listed inside the box.

- *Decision boxes*: Determine the exit paths from the state boxes.



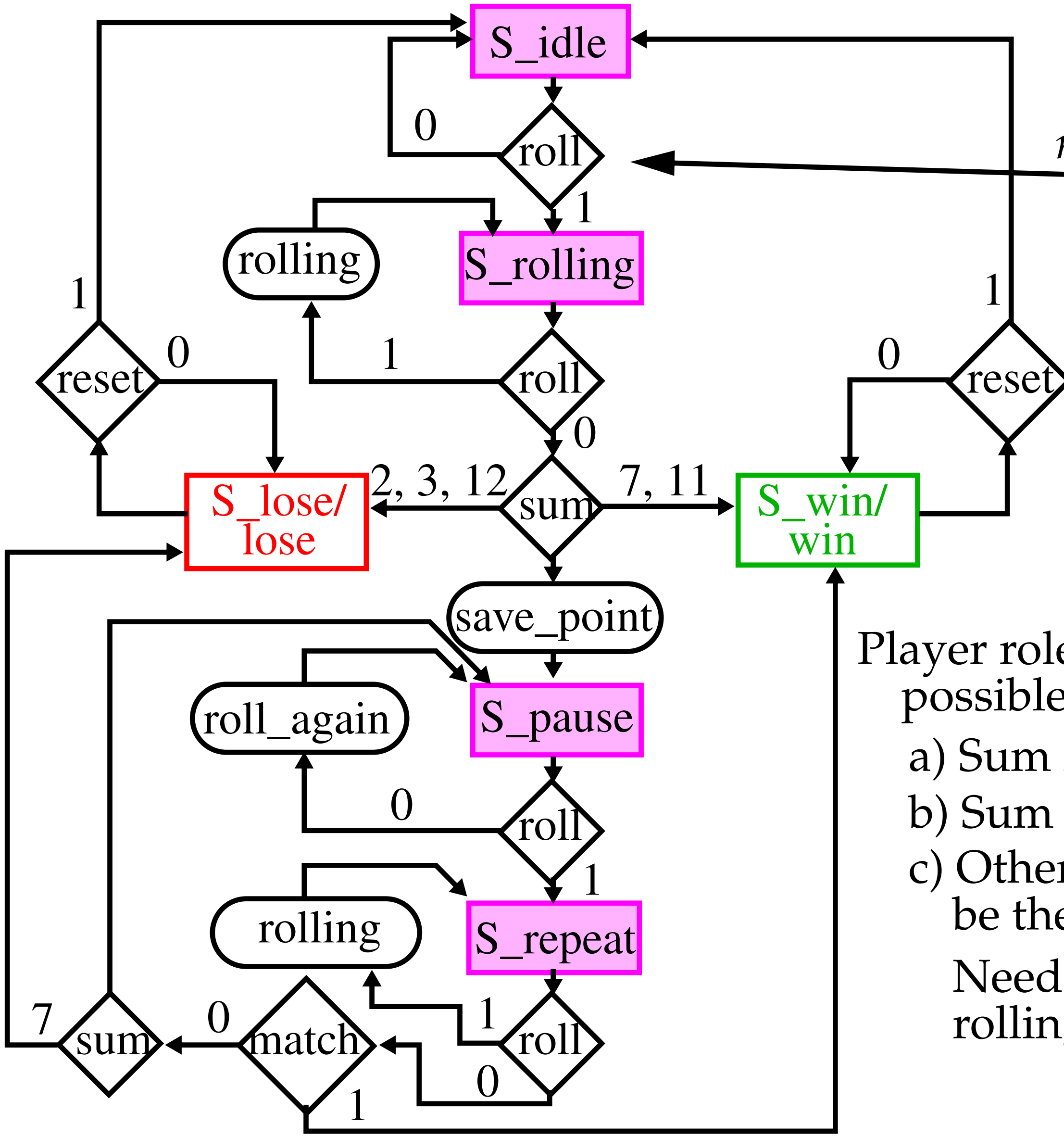
Listed: inputs that are or become asserted

- *Conditional output boxes*: Output signals that are asserted conditionally (Mealy machines only).

The assertion of the output signal depends on the state and inputs

Here, the output depends on the state AND the value of the inputs.

FSMs: Craps example



A version of craps

roll is an asynchronous input

States are entered on rising edge of clk

Player roles the die with 3 possible outcomes:

a) Sum is 7 or 11, player wins

b) Sum is 2, 3, or 12, player loses

c) Otherwise, sum is declared to be the player's *point*.

Needs to roll it again before rolling a 7 to win

FSMs: Craps example

The *rolling unit* generates (?random?) values for *D_left* and *D_right* synchronously with *clk*.

The *sum* of *D_left* and *D_right* is computed immediately (synchronously) with combinational logic *but* does NOT effect state transitions until the NEXT rising edge of clk.

Output signals, *win*, *lose*, *match*, *roll_again* are asserted synchronously or asynchronously depending on whether they depend on the input *roll*.

Pay particular attention to **how** signals are asserted and de-asserted!

next_state is computed by combinational logic using an **always** behavior

- The sensitivity list includes the signals evaluated in the *decision* blocks, e.g., *roll*, *sum* and *match*.

The description also includes a signal, *save_point*, that serves as a clk to a register that saves the value of *sum* -- it is asynchronous.

FSMs: Craps example

```
module Crap_shoot
  (clk, reset, point, roll, win, match, lose, roll_again, rolling, blank, D_left, D_right,
     sum);
   input clk, reset, roll;
   output win, lose, match, roll_again, rolling, blank;
   output [3:0] point;
   output [2:0] D_left, D_right;
   output [3:0] sum;
   parameter S_idle = 0;
   parameter S_rolling = 1;
   parameter S_pause = 2;
   parameter S_repeat = 3;
   parameter S_lose = 4;
   parameter S_win = 5;

   wire match, rolling, roll_again, win, lose, save_point;
   reg [2:0] D_left, D_right;
   wire [3:0] sum = D_left + D_right;
   reg [2:0] state, next_state;
   reg [3:0] point;
```

FSMs: Craps example

```
// Rolling Unit
   always @( posedge clk or posedge reset )
      if (reset) begin D_left <= 1; D_right <= 1; end
      else begin
         if (D_left < 6) D_left <= D_left + 1; else D_left <= 1;
         if (D_left == 6 && D_right < 6) D_right <= D_right + 1; else
            if (D_left == 6 && D_right == 6) D_right <= 1;
      end

// Scoring Unit
   assign match = (sum == point);
   assign roll_again = (state == S_pause && !roll);   // synchronously set but asynchronously reset
   assign rolling = ((state == S_rolling && roll) || (state == S_repeat && roll));
   assign save_point = ((state == S_rolling) && !roll &&
      sum != 2 &&
      sum != 3 &&
      sum != 12 &&
      sum != 7 &&
      sum != 11);
   // save_point set asynchronously
   // because it depends on roll
   assign win = (state == S_win);
   assign lose = (state == S_lose);
   assign blank = (point < 2);
```

FSMs: Craps example

```
// Control Unit
  always @( posedge save_point or posedge reset )    // save_point serves as
    if (reset) point <= 0;                            // 'clk' for point register
    else point <= sum;                                // -- asynchronous

  always @(posedge clk or posedge reset)
    if (reset) state <= S_idle;         // match can go low on rising edge of clk,
    else state <= next_state;           // changing next_state but this okay

                                               // synchronous changes to these
  always @(state or sum or roll or match)      // signals are NOT immediately
    case (state)                               // realized in next_state
      S_idle: if (roll) next_state <= S_rolling; else next_state <= S_idle;
      S_rolling: if (roll) next_state <= S_rolling; else
                   if (sum == 2 || sum == 3 || sum == 12) next_state <= S_lose;
                   else
                     if (sum == 7 || sum == 11) next_state <= S_win;
                     else next_state <= S_pause;
      S_pause: if (roll) next_state <= S_repeat; else next_state <= S_pause;
      S_repeat: if (roll) next_state <= S_repeat; else
                  if (match) next_state <= S_win; else
                    if (sum == 7) next_state <= S_lose; else next_state <= S_pause;
      S_win: next_state <= S_win;
      S_lose: next_state <= S_lose;
    endcase endmodule
```