Datapaths and Control

- Digital systems perform sequences of operations on encoded data
- *Datapath*
  - Combinational circuits for operations
  - Registers for storing intermediate results
- *Control section*: control sequencing
  - Generates *control signals*
    - Selecting operations to perform
    - Enabling registers at the right times
  - Uses *status signals* from datapath

Example: Complex Multiplier

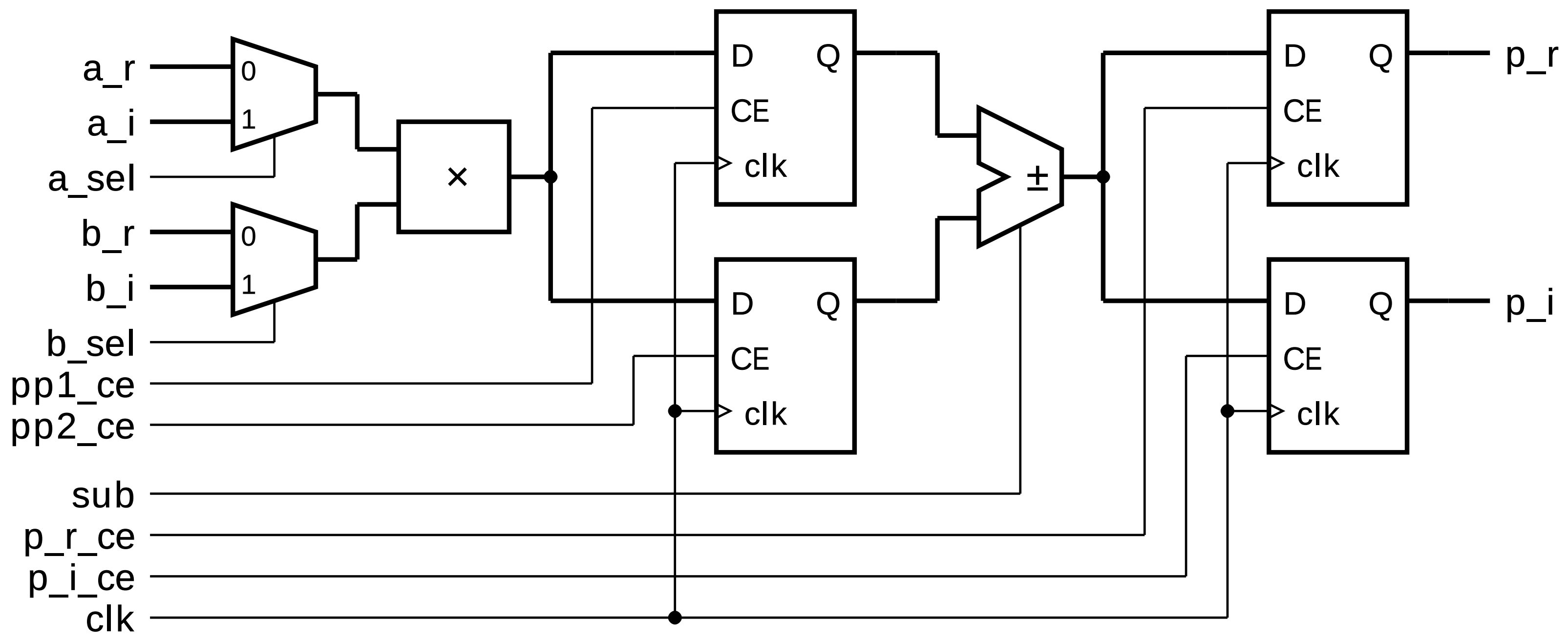
- Cartesian form, fixed-point
  - operands: 4 integer, 12 fraction bits
  - result: 8 pre-, 24 post-binary-point bits
- Subject to tight area constraints

$$a = a_r + ja_i \qquad b = b_r + jb_i$$

$$p = ab = p_r + jp_i = (a_r b_r - a_i b_i) + j(a_r b_i + a_i b_r)$$

- 4 multiplies, 1 add, 1 subtract
  - Perform sequentially using 1 multiplier, 1 adder/subtracter

Complex Multiplier Datapath



Complex Multiplier in Verilog

```
module multiplier
  ( output reg signed [7:-24] p_r, p_i,
    input      signed [3:-12] a_r, a_i, b_r, b_i,
    input                     clk, reset, input_rdy );

  reg a_sel, b_sel, pp1_ce, pp2_ce, sub, p_r_ce, p_i_ce;

  wire signed [3:-12] a_operand, b_operand;
  wire signed [7:-24] pp, sum
  reg  signed [7:-24] pp1, pp2;
  ...
```

Complex Multiplier in Verilog

```
  assign a_operand = ~a_sel ? a_r : a_i;
  assign b_operand = ~b_sel ? b_r : b_i;

  assign pp = {{4{a_operand[3]}}, a_operand, 12'b0} *
              {{4{b_operand[3]}}, b_operand, 12'b0};

  always @(posedge clk)  // Partial product 1 register
    if (pp1_ce) pp1 <= pp;

  always @(posedge clk)  // Partial product 2 register
    if (pp2_ce) pp2 <= pp;

  assign sum = ~sub ? pp1 + pp2 : pp1 - pp2;

  always @(posedge clk)  // Product real-part register
    if (p_r_ce) p_r <= sum;

  always @(posedge clk)  // Product imaginary-part register
    if (p_i_ce) p_i <= sum;

  ...
endmodule
```

Multiplier Control Sequence

- Avoid resource conflict
- First attempt
  1. a_r * b_r → pp1_reg
  2. a_i * b_i → pp2_reg
  3. pp1 – pp2 → p_r_reg
  4. a_r * b_i → pp1_reg
  5. a_i * b_r → pp2_reg
  6. pp1 + pp2 → p_i_reg
- Takes 6 clock cycles

Multiplier Control Sequence

- Merge steps where no resource conflict
- Revised attempt
  1. a_r * b_r → pp1_reg
  2. a_i * b_i → pp2_reg
  3. pp1 – pp2 → p_r_reg
     a_r * b_i → pp1_reg
  4. a_i * b_r → pp2_reg
  5. pp1 + pp2 → p_i_reg
- Takes 5 clock cycles

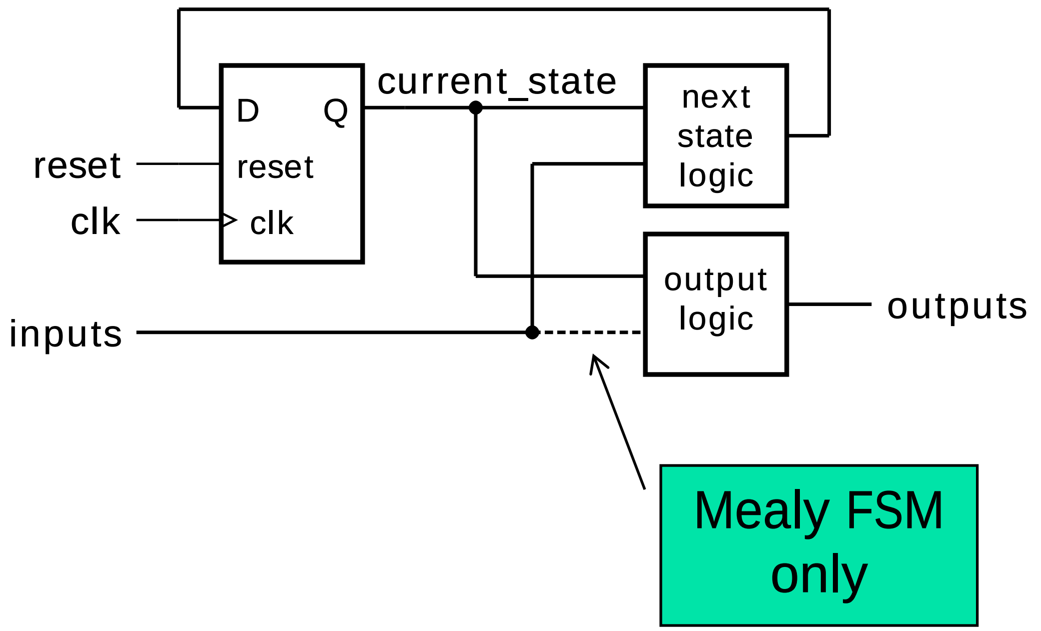
Multiplier Control Signals

Step	a_sel	b_sel	pp1_ce	pp2_ce	sub	p_r_ce	p_i_ce
1	0	0	1	0	–	0	0
2	1	1	0	1	–	0	0
3	0	1	1	0	1	1	0
4	1	0	0	1	–	0	0
5	–	–	0	0	0	0	1

Finite-State Machines

- Used the implement control sequencing
- A FSM is defined by
  - set of inputs
  - set of outputs
  - set of states
  - initial state
  - transition function
  - output function
- States are steps in a sequence of transitions
  - There are “Finite” number of states.

FSM in Hardware



- Mealy FSM: outputs depend on state and inputs
- Moore FSM: outputs depend on state only (no dash)
- Mealy and Moore FSM can convert to each other

FSM Example: Multiplier Control

- One state per step
- Separate idle state?
  - Wait for input_rdy = 1
  - Then proceed to steps 1, 2, …
  - But this wastes a cycle!
- Use step 1 as idle state
  - Repeat step 1 if input_rdy ≠ 1
  - Proceed to step 2 otherwise
- Output function
  - Defined by table on slide 43
  - Moore or Mealy?

Transition function

current_ state	input_ rdy	next_ state
step1	0	step1
step1	1	step2
step2	–	step3
step3	–	step4
step4	–	step5
step5	–	step1

State Encoding

- Encoded in binary
  - N states: use at least $\lceil \log_2 N \rceil$ bits
- Encoded value used in circuits for transition and output function
  - encoding affects circuit complexity
- Optimal encoding is hard to find
  - CAD tools can do this well
- One-hot works well in FPGAs
- Often use 000...0 for idle state
  - reset state register to idle

FSMs in Verilog

- Use parameters for state values
  - Synthesis tool can choose an alternative encoding

```
parameter [2:0] step1 = 3'b000, step2 = 3'b001,
                step3 = 3'b010, step4 = 3'b011,
                step5 = 3'b100;
reg [2:0] current_state, next_state ;
...
```

Multiplier Control in Verilog

```
always @(posedge clk or posedge reset)  // State register
  if (reset) current_state <= step1;
  else       current_state <= next_state;

always @*  // Next-state logic
  case (current_state)
    step1: if (!input_rdy) next_state = step1;
           else            next_state = step2;
    step2:                 next_state = step3;
    step3:                 next_state = step4;
    step4:                 next_state = step5;
    step5:                 next_state = step1;
  endcase
```

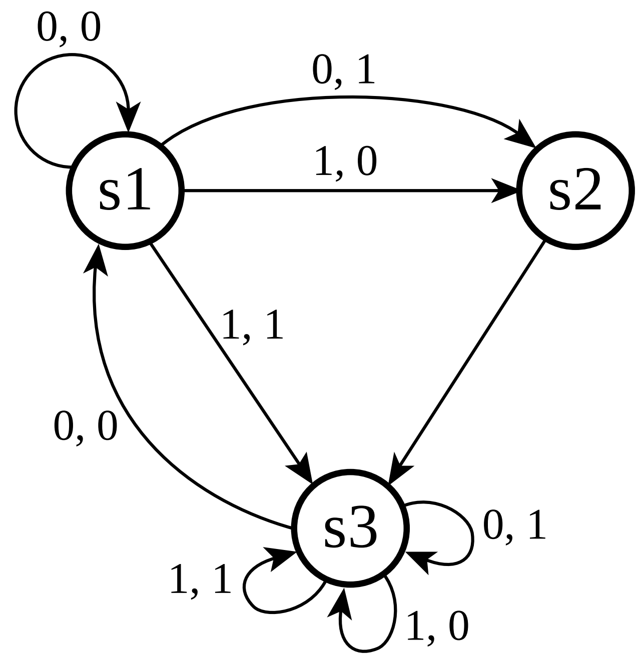
Multiplier Control in Verilog

```
always @* begin  // Output_logic
  a_sel = 1'b0; b_sel = 1'b0;  pp1_ce = 1'b0; pp2_ce = 1'b0;
  sub = 1'b0;   p_r_ce = 1'b0; p_i_ce = 1'b0;
  case (current_state)
    step1: begin
             pp1_ce = 1'b1;
           end
    step2: begin
             a_sel = 1'b1; b_sel = 1'b1; pp2_ce = 1'b1;
           end
    step3: begin
             b_sel = 1'b1; pp1_ce = 1'b1;
             sub = 1'b1;   p_r_ce = 1'b1;
           end
    step4: begin
             a_sel = 1'b1; pp2_ce = 1'b1;
           end
    step5: begin
             p_i_ce = 1'b1;
           end
  endcase
end
```

State Transition Diagrams

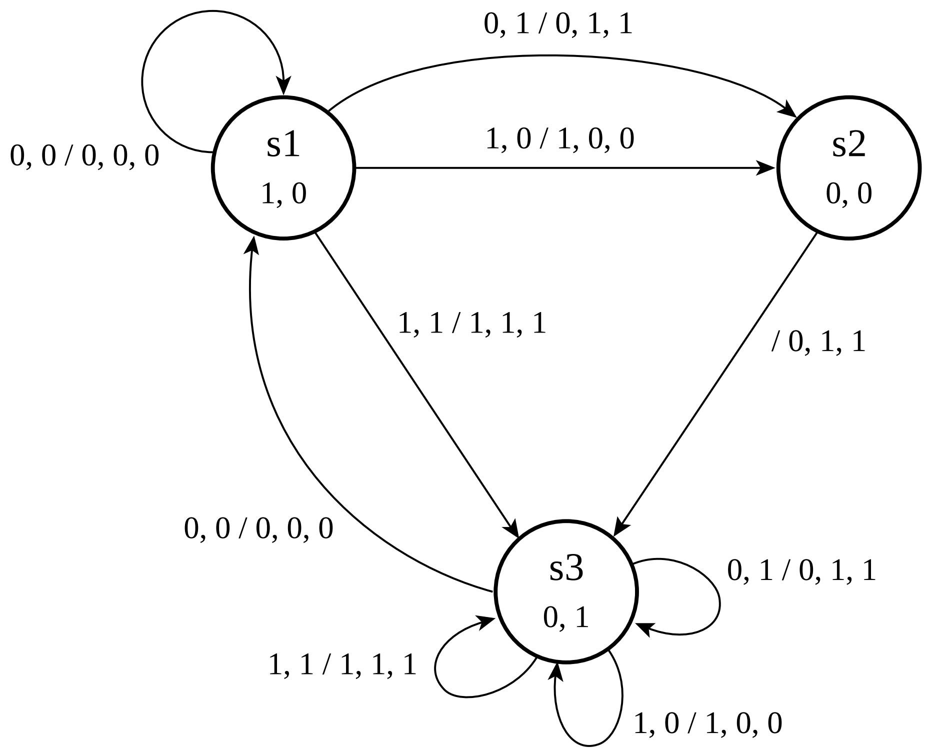
- Bubbles to represent states
- Arcs to represent transitions

- Example
  - S = {s1, s2, s3}
  - Inputs (a1, a2):
    Σ = {(0,0), (0,1), (1,0), (1,1)}
  - δ defined by diagram



State Transition Diagrams

- Annotate diagram to define output function
  - Annotate states for Moore-style outputs
  - Annotate arcs for Mealy-style outputs
- Example
  - x_1, x_2: Moore-style
  - y_1, y_2, y_3: Mealy-style


s1
1, 0
s2
0, 0
s3
0, 1
0, 0 / 0, 0, 0
0, 1 / 0, 1, 1
1, 0 / 1, 0, 0
1, 1 / 1, 1, 1
/ 0, 1, 1
0, 0 / 0, 0, 0
0, 1 / 0, 1, 1
1, 1 / 1, 1, 1
1, 0 / 1, 0, 0