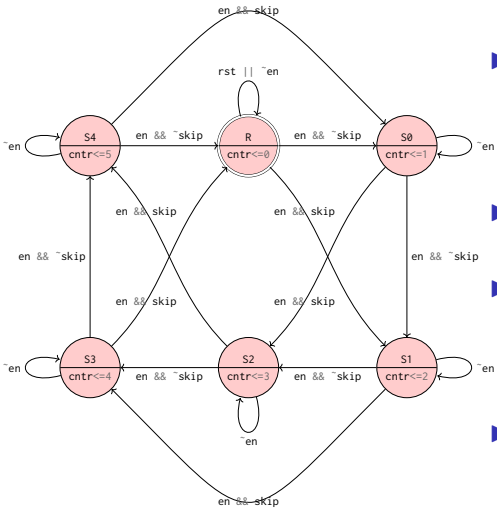
State Machine Examples

Nachiket Kapre
nachiket@uwaterloo.ca

Outline

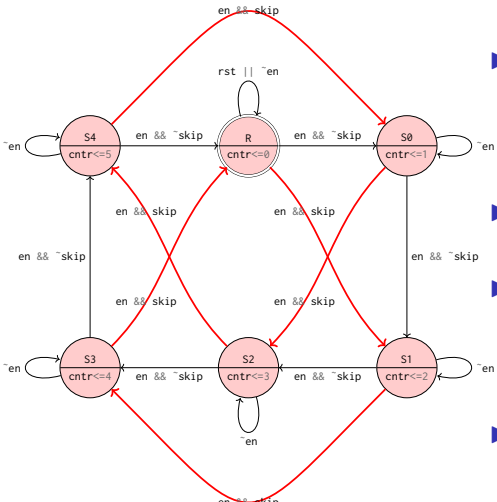
- Complex control flow in state machines
- FSMD Pattern $\rightarrow$ Finite State Machine + Datapath

Example 4: Counter with skip option



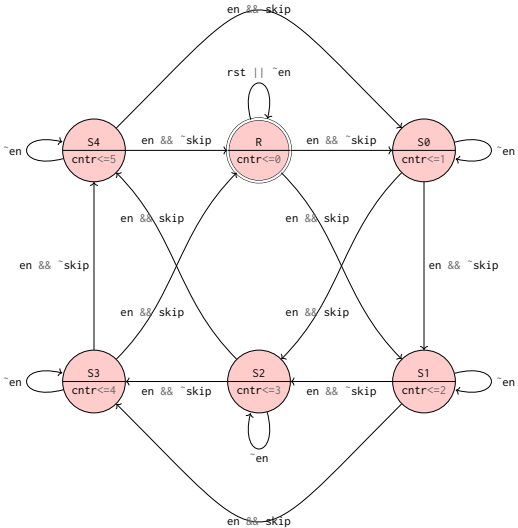
- **Problem**: design a counter that counts from 0 to 5, and restarts from 0. Counter updates only on enable, and skips by 2 if skip input is enabled!
- Number of states = 6, 1-bit enable input, 1-bit skip input, Output = 3-bit count
- Three kinds of state transitions
  - Normal advancement if en **and** ~skip
  - Jump if en && skip
  - Self loop if ~en
- Each state has a simple action to assign a constant value to cntr output

Example 4: Counter with skip option

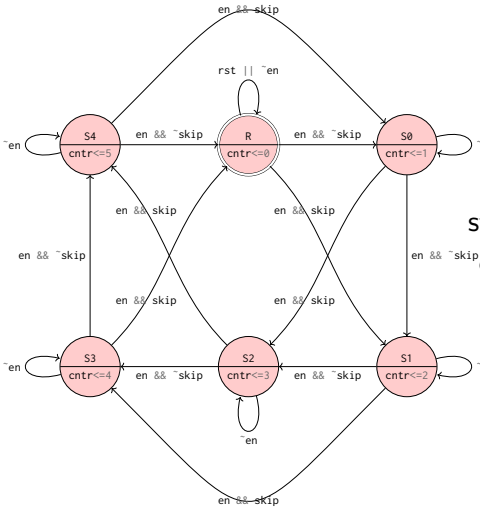


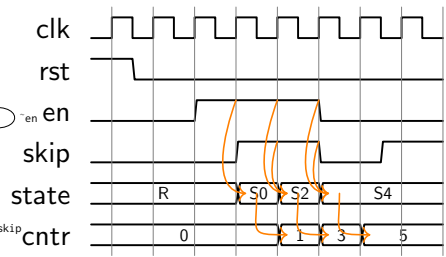
- **Problem**: design a counter that counts from 0 to 5, and restarts from 0. Counter updates only on enable, and skips by 2 if skip input is enabled!
- Number of states = 6, 1-bit enable input, 1-bit skip input, Output = 3-bit count
- Three kinds of state transitions
  - Normal advancement if en **and** ~skip
  - Jump if en && skip
  - Self loop if ~en
- Each state has a simple action to assign a constant value to cntr output

Example 4: FSM Diagram



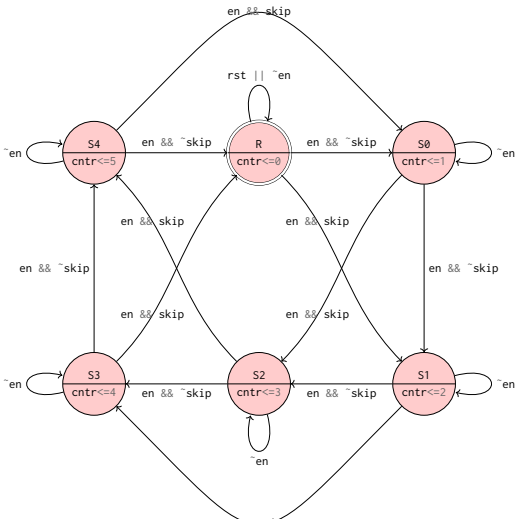
Example 4: Counter with skip option





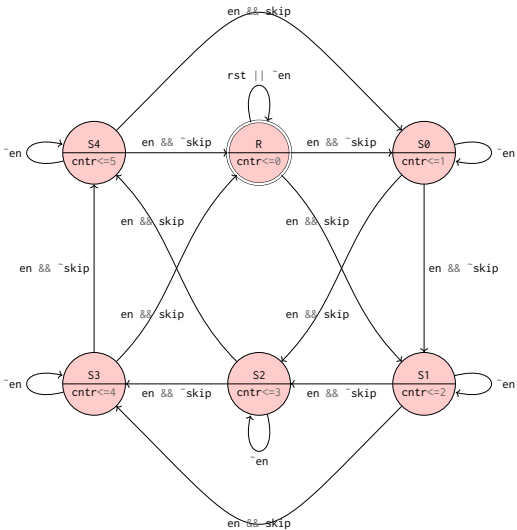
- Now have inputs en and skip in the timing diagram
- Each State has three outgoing edges $\rightarrow$
  - Either, only one condition must be true
  - Or use nested if to infer priority

Example 4: Counter with skip option




```
module cntr5enskip(
  input wire clk,
  input wire rst,
  input wire en,
  input wire skip,
  output reg [2:0] cntr
 );
```

Example 4: Counter with skip option

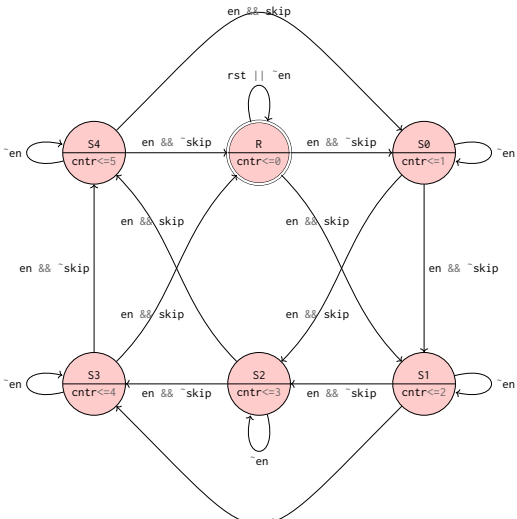


Nested if
but mutually exclusive

```
R : begin //
 if(en && ~skip) begin
  state <= S0;
 end
 else if(en && skip) begin
  state <= S1;
 end //
end
```

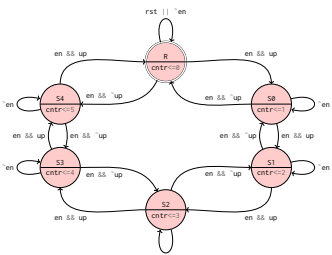
Missing Else
Stay in same state

Example 4: Counter with skip option



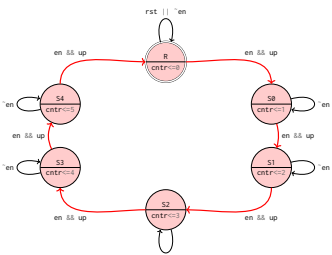

```
always @(posedge clk) begin
 if(rst) begin
  cntr <= 0;
 end else begin
  case(state)
   R  : cntr <= 0;
   S0  : cntr <= 1;
   S1  : cntr <= 2;
   S2  : cntr <= 3;
   S3  : cntr <= 4;
   S4  : cntr <= 5;
  endcase
 end
end
```

Example 5: Counter with up/down option



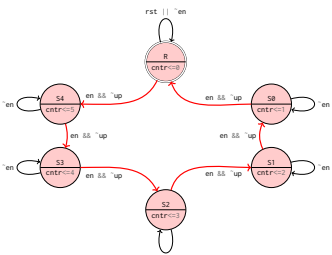
- **Problem**: design a counter that counts from 0 to 5, and restarts from 0. Counter updates only on enable, and supports reversing count.
- Number of states = 6, 1-bit enable input, 1-bit up/down input, Output = 3-bit count
- Three kinds of state transitions
  - Normal advancement if en && up
  - Reverse flow if en && ~up
  - Self loop if ~en
- Each state has a simple action to assign a constant value to cntr output

Example 5: Counter with up/down option



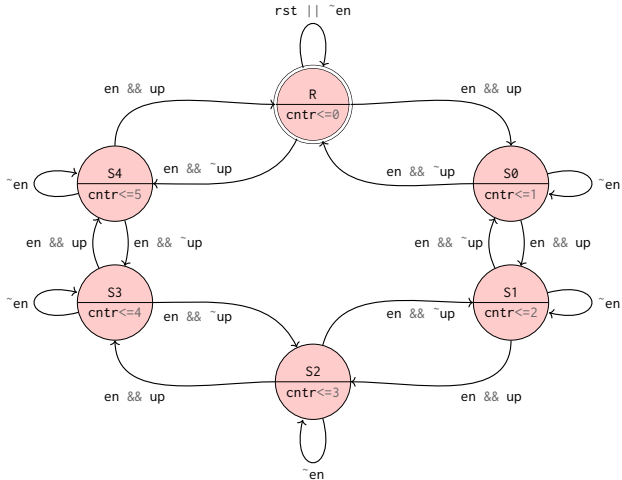
- **Problem**: design a counter that counts from 0 to 5, and restarts from 0. Counter updates only on enable, and supports reversing count.
- Number of states = 6, 1-bit enable input, 1-bit up/down input, Output = 3-bit count
- Three kinds of state transitions
  - Normal advancement if en && up
  - Reverse flow if en && ~up
  - Self loop if ~en
- Each state has a simple action to assign a constant value to cntr output

Example 5: Counter with up/down option

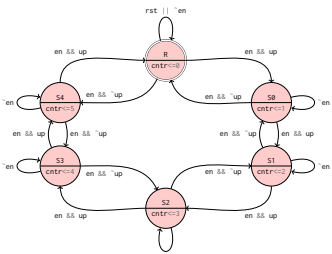


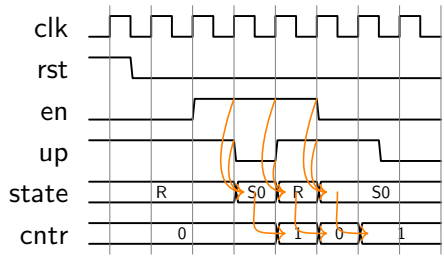
- **Problem**: design a counter that counts from 0 to 5, and restarts from 0. Counter updates only on enable, and supports reversing count.
- Number of states = 6, 1-bit enable input, 1-bit up/down input, Output = 3-bit count
- Three kinds of state transitions
  - Normal advancement if en && up
  - Reverse flow if en && ~up
  - Self loop if ~en
- Each state has a simple action to assign a constant value to cntr output

Example 5: FSM Diagram



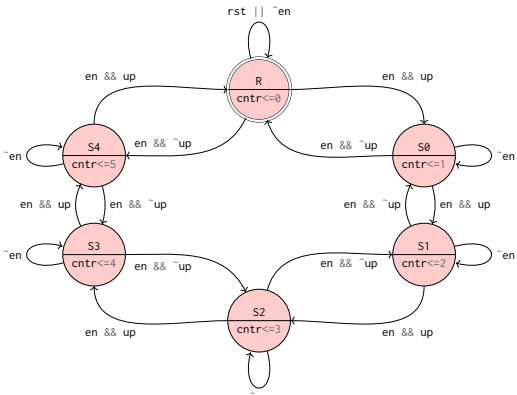
Example 5: Counter with up/down option





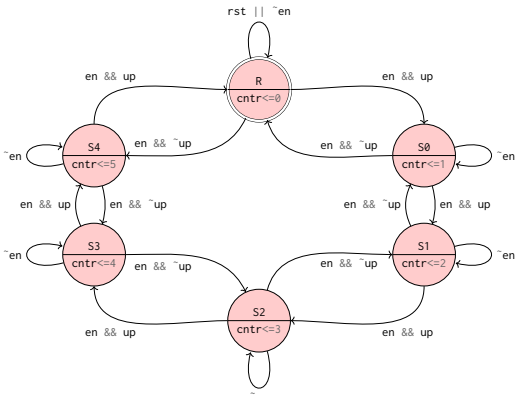
- Now have inputs en and up in the timing diagram
- Each State has three outgoing edges $\rightarrow$
  - Again, mutual exclusivity or priority must be enforced

Example 5: Counter with up/down option




```
module cntr5updown(
  input wire clk,
  input wire rst,
  input wire en,
  input wire up,
  output reg [2:0] cntr
 );
```

Example 5: Counter with up/down option

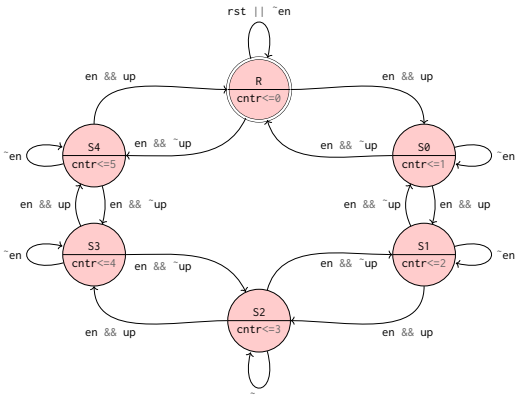


Nested if
but mutually exclusive

```
R : begin //
 if(en && up) begin
  state <= S0;
 end else if(en && ~up) begin
  state <= S4;
 end //
end
```

Missing Else
Stay in same state

Example 5: Counter with up/down option

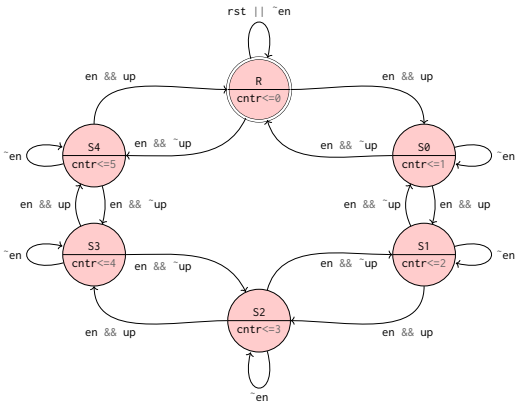


Nested if
but mutually exclusive

```
R : begin //
 if(en && up) begin
  state <= S0;
 end else if(en && ~up) begin
  state <= S4;
 end //
end
```

Missing Else
Stay in same state

Example 5: Counter with up/down option




```
always @(posedge clk) begin
 if(rst) begin
  cntr <= 0;
 end else begin
  case(state)
   R  : cntr <= 0;
   S0  : cntr <= 1;
   S1  : cntr <= 2;
   S2  : cntr <= 3;
   S3  : cntr <= 4;
   S4  : cntr <= 5;
  endcase
 end
end
```

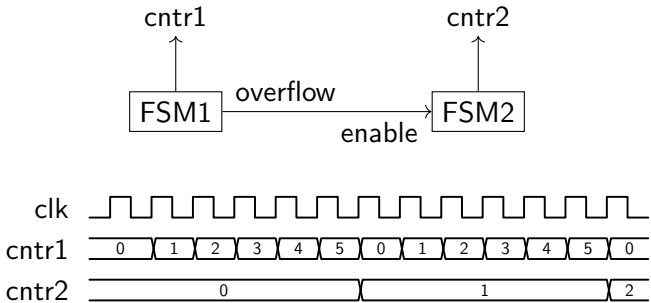
RTL Coding Impact of skip/up input

- Each state now has multiple outgoing edges
  - The condition to activate each edge should be mutually exclusive
  - If that is not possible, there should be a priority order
    - This may be denoted with a separate priority label on each edge
    - Makes the FSM diagram complicated, so generally avoided
- Diagrams can get messy $\rightarrow$ anticipate congestion, space things apart

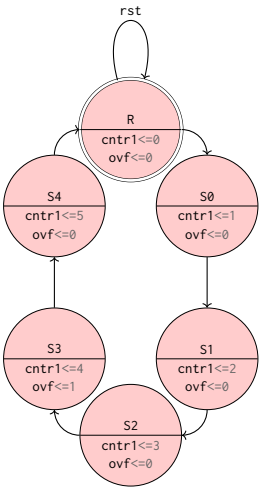
Two interacting state machines

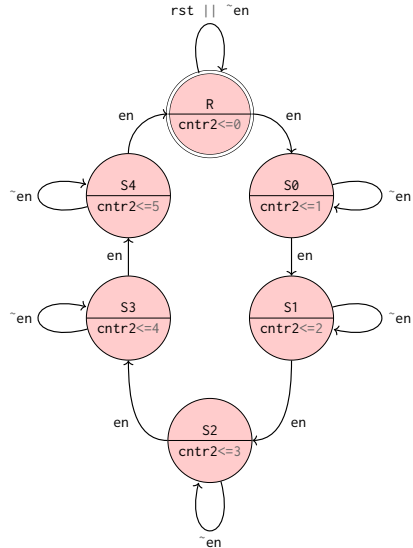
- **Problem**: Design a *cascaded* counter which contains a pair of upstream and downstream counters.
  - The upstream counter counts from 0 to 5 in a free-running manner. When it overflows, it restarts, and generates an enable signal for the downstream counter for one cycle. This tells the downstream counter to increment by 1.
  - The downstream counter also counts from 0 to 5 in a conditional manner based on an enable signal generated by the upstream counter.
- Number of FSMs=2
  - FSM1: Number of states = 6, no input, Output = 3-bit count + 1-bit overflow
  - FSM2: Number of states = 6, 1-bit enable input, Output = 3-bit count
- Interacting state machines with unidirectional flow of data

Rough sketch of design

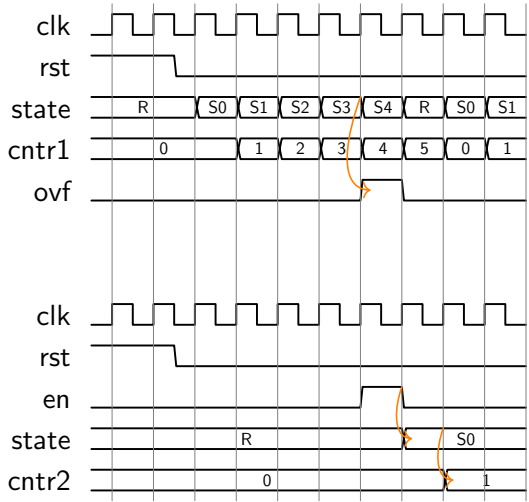


Two interacting state machines





Timing Diagram



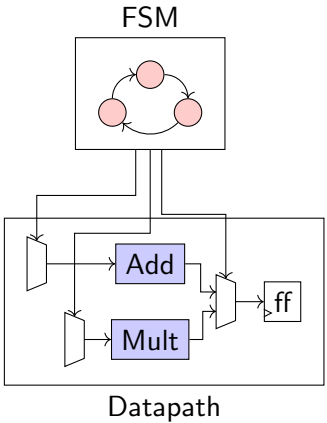
Explanation

- Generate overflow a cycle sooner to allow `cntr` values to align
- Direction of dataflow is from FSM1 to FSM2
  - Output `ovf` from FSM1 is input `en` for FSM2
- FSM1 is just a free-running counter from before, with special signal to indicate overflow
  - Note that overflow indication will be generated a cycle after the state, hence assigned as state action in `S4`.
- FSM2 will see one-cycle enable signals, so state machine will advance at a much slower rate

FSMD Design Pattern

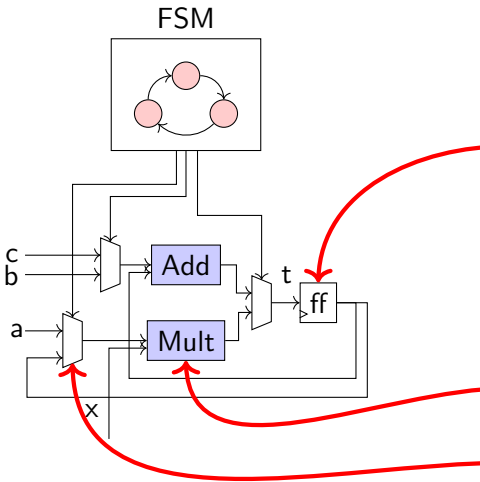
- FSMs can drive Datapath logic
- Datapaths contain arithmetic units, logic blocks, muxes, stitched together in dataflow fashion
- FSMs can control and co-ordinate data movement in datapaths
- FSMs can decide when inputs enter, and when outputs leave a Datapath

FSMD: Finite State Machine + Datapath



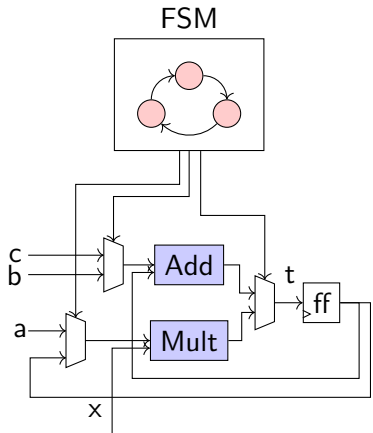
- Typically **FSMs** drive a Time-multiplexed **Datapath**
- FSM generates control signals to orchestrate use of datapath blocks in each cycle
- *e.g.* In a processor
  - The instruction decoder + program counter can be an **FSM**
  - ALU with multiplexers and bypassing logic is the **Datapath**
- FSM generates a multiplexer control table

Datapath

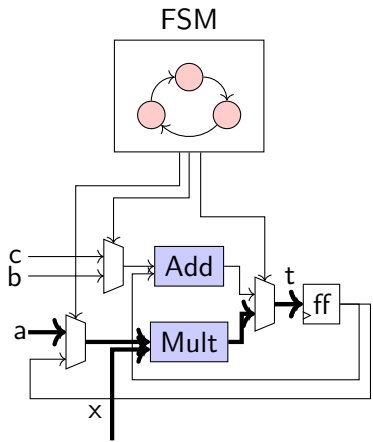



```
wire [7:0] add_mux_c;
wire [15:0] mult_mux_c;
wire [23:0] add_c;
wire [23:0] mult_c;
reg [23:0] temp1;

// output register and mux
always @(posedge clk) begin
  if(rst) begin
    temp1 <= {24{1'b0}};
  end else begin
    if((op == 1'b 0)) begin
      temp1 <= add_c;
    end else begin
      temp1 <= mult_c;
    end
  end
end

// arithmetic ops
assign add_c = add_mux_c + temp1;
assign mult_c = mult_mux_c * x;
assign y = temp1;
// input muxes
assign add_mux_c = add_in == 1'b 0 ? b : c;
assign mult_mux_c = mult_in == 1'b 0 ?
  {8'b 00000000,a} : temp1[15:0];
```

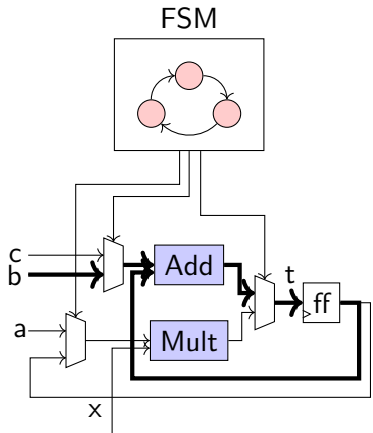
Cycle-by-cycle operation



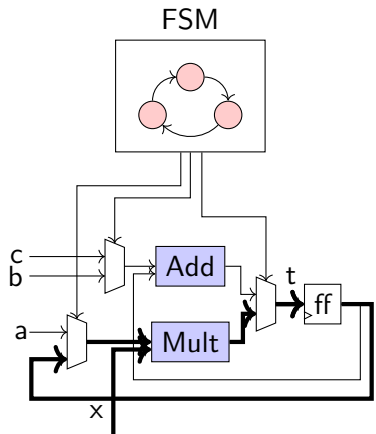
Cycle-by-cycle operation



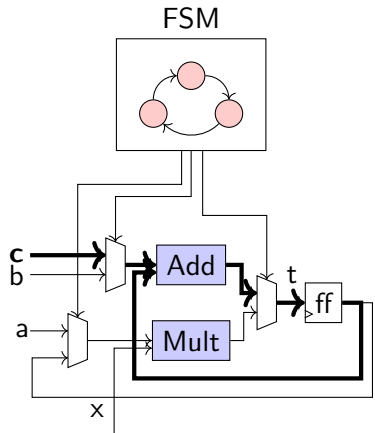
Cycle-by-cycle operation



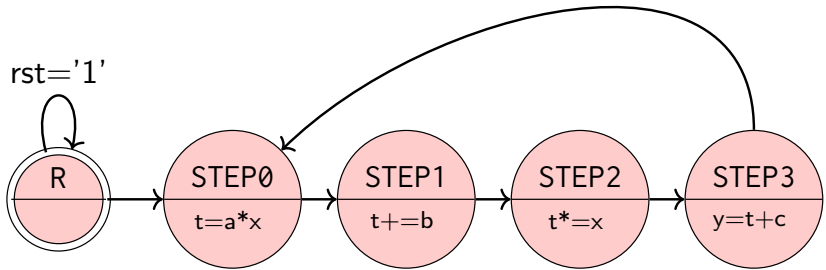
Cycle-by-cycle operation



Cycle-by-cycle operation



Poly State Machine



- Datapath will compute $a \cdot x^2 + b \cdot x + c$
- State machine will control order of operation and generation of output
- Approximately a simple CPU structure (state = program counter)
- Internal state t separate from the state variable

Poly FSMD in Verilog (only FSM)

```
always @(posedge clk) begin
 if(rst) begin
  state <= STEP0;
 end else begin
  case(state)
   STEP0 : begin
    state <= STEP1;
   end
   STEP1 : begin
    state <= STEP2;
   end
   STEP2 : begin
    state <= STEP3;
   end
   STEP3 : begin
    state <= STEP0;
   end
  endcase
  y_valid <= y_valid_c;
 end
end
```

```
always @(*) begin
 mult_in <= 1'b 0;
 add_in <= 1'b 0;
 y_valid_c <= 1'b 0;
 case(state)
  STEP0 : begin
   op <= 1'b 1;  // *
   mult_in <= 1'b 0; // A
  end
  STEP1 : begin
   op <= 1'b 0;  // +
   add_in <= 1'b 0; // B
  end
  STEP2 : begin
   op <= 1'b 1;  // *
   mult_in <= 1'b 1; // TEMP
  end
  STEP3 : begin
   op <= 1'b 0;  // +
   add_in <= 1'b 1; // C
   y_valid_c <= 1'b 1;
  end
 endcase
end
```

Parenthesis Matching Problem

- `. . . ( . . . () () ( () ) . . . )`
- Check if the expression is balanced
  - Simply counting `(` and `)` won't do
  - Can we do this with a state machine?

Wrapup

- Finite State Machines can have complex control flow with multiple branches from each state $\rightarrow$ must consider priority or mutual exclusivity
- FSMs can interact with each other $\rightarrow$ must pay attention to cycle/timing of interacting signals
- FSMs key component of the FSMD design pattern!
- FSMs are everywhere: communication chips, crypto, games, ...