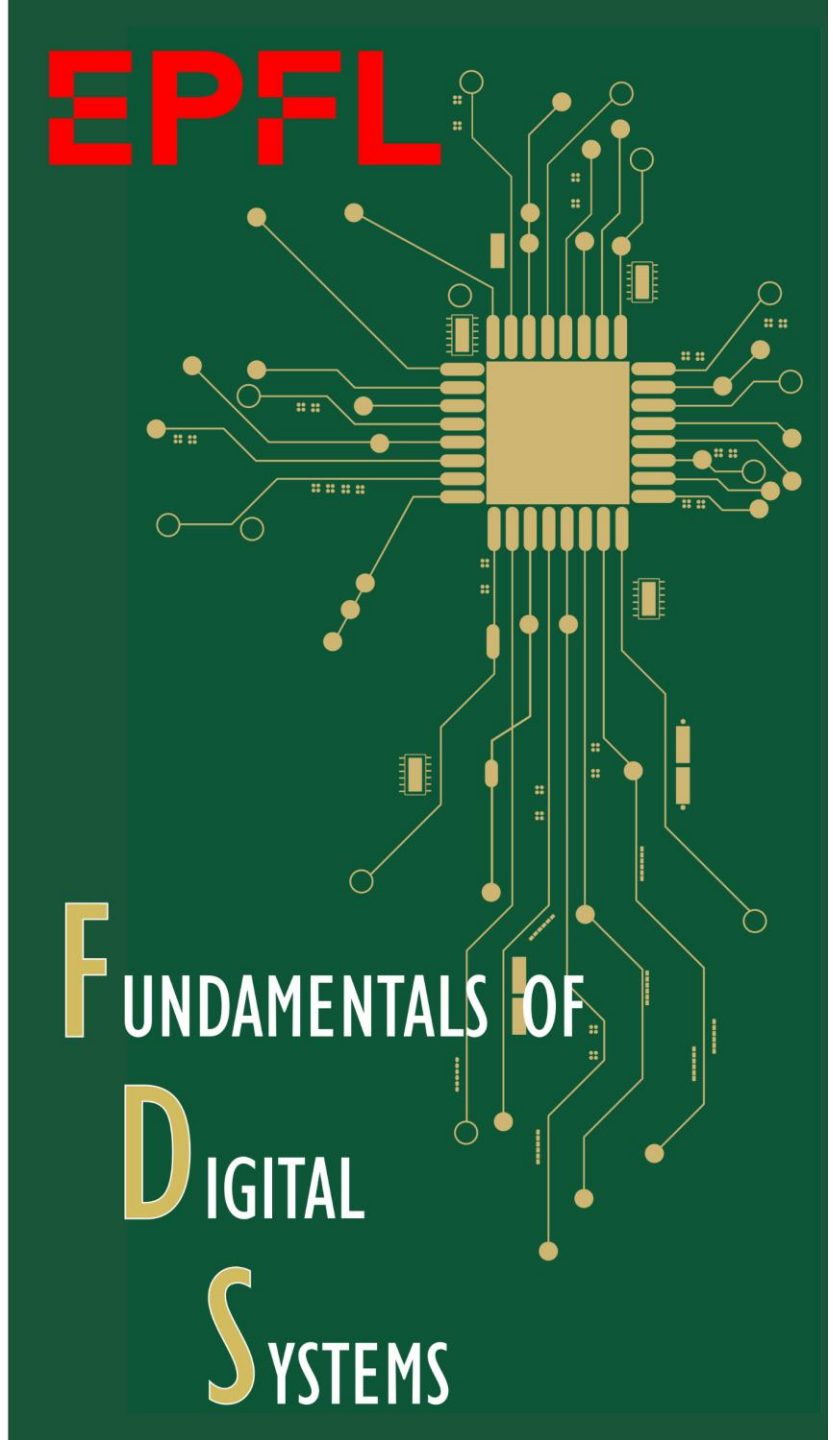


Digital Logic Circuits

Examples of FSMs in Verilog

CS-173 Fundamentals of Digital Systems

Mirjana Stojilović

Spring 2025

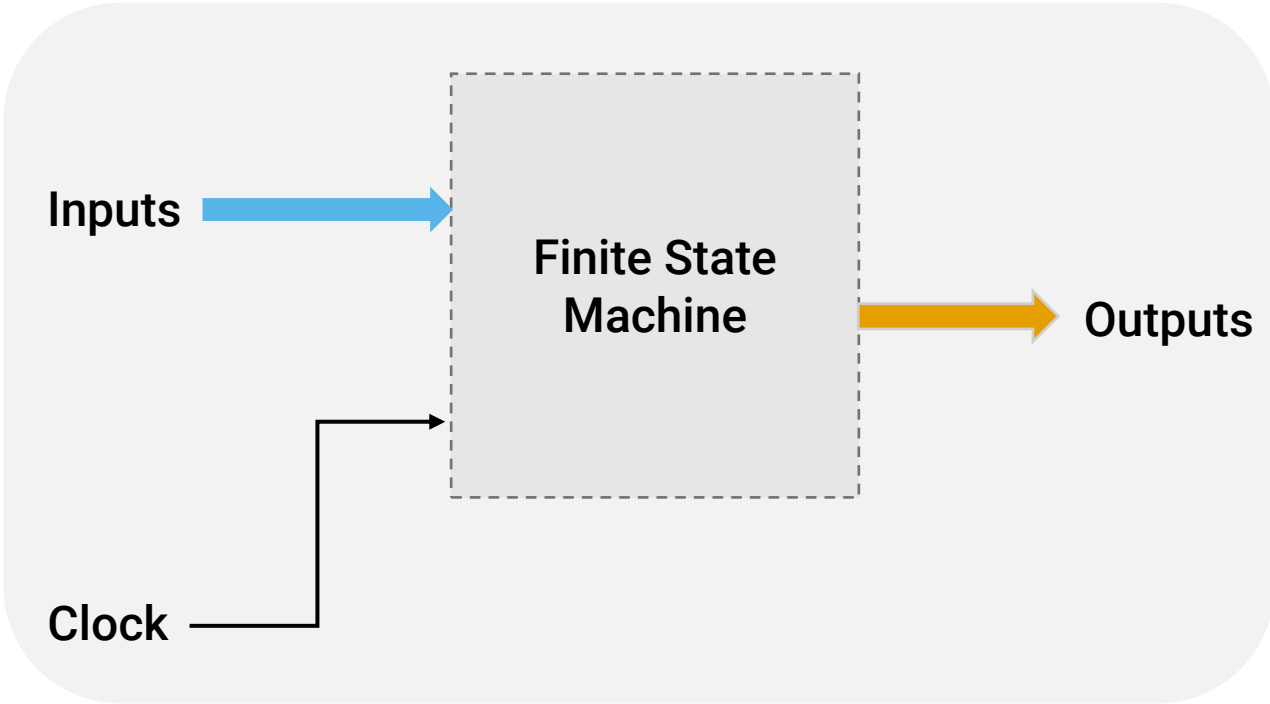
Examples of FSMs

Quick Outline

- Mealy FSM
- Moore FSM
- Serial Adder FSM

Recall: Finite State Machines

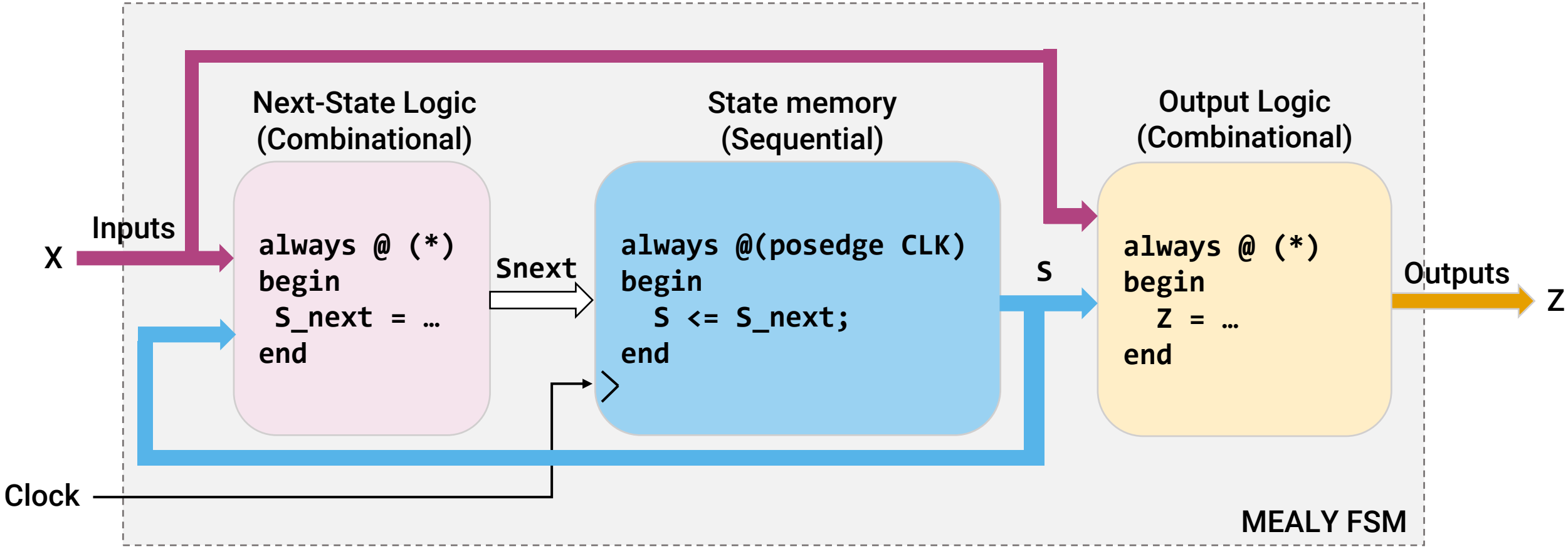
- Two types of FSMs exist
- Mealy state machines
  - Named after George H. Mealy
- Moore state machines
  - Named after Edward F. Moore



- State transitions occur in sync with the clock edges (e.g., rising)

Recall: Mealy FSM Modeling

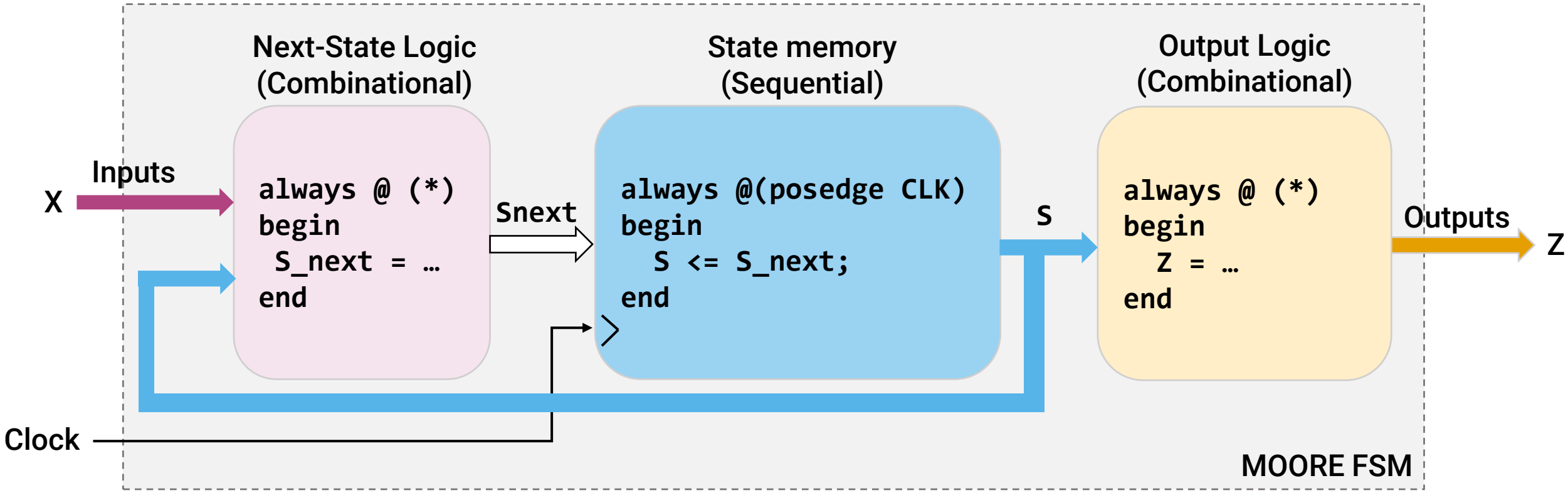
In Verilog



Note: Adjust the sensitivity list of the state memory, as required

Recall: Moore FSM Modeling

In Verilog



Note: Adjust the sensitivity list of the state memory, as required

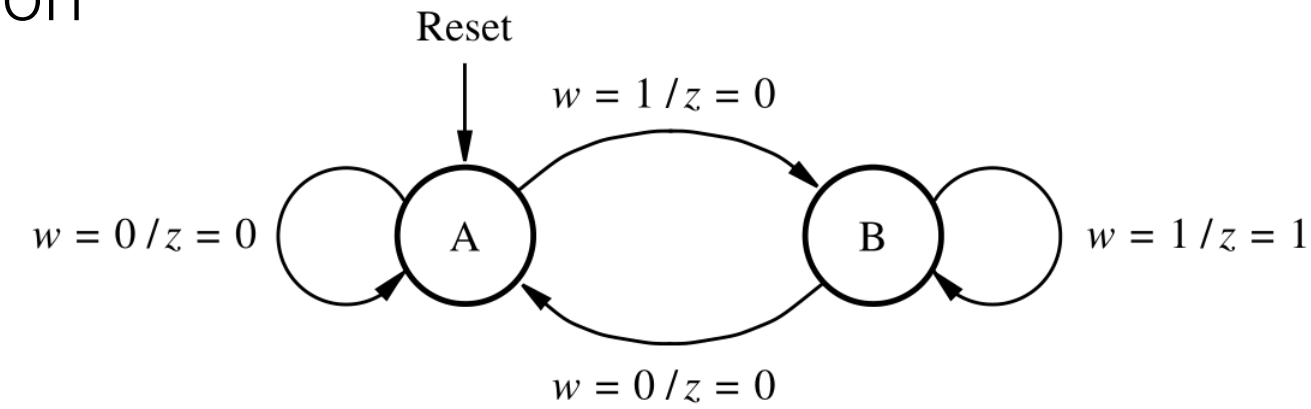
Mealy FSM

Example

Mealy FSM

From state diagram to Verilog

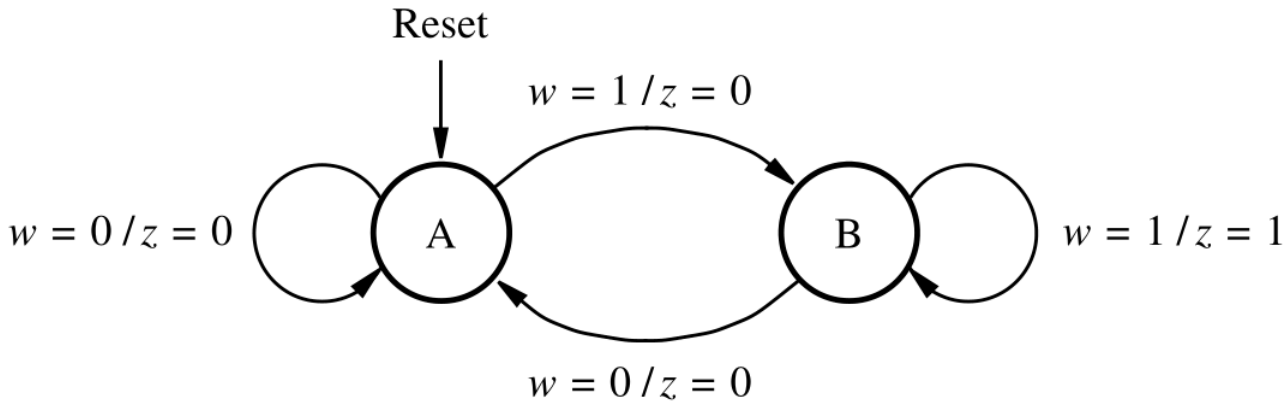
- Figure shows the state diagram of…
  - Mealy FSM
- Write the Verilog description



Mealy FSM

From state diagram to Verilog, Contd.

- States are…
  - A, B;
  - At least 1 bit is required to present them
- Inputs are…
  - 1-bit input w
- Outputs are…
  - 1-bit z
- Reset input is…
  - Asynchronous, active high
    - Strictly speaking, an edge from every state to A, in case of active Reset, could also be shown



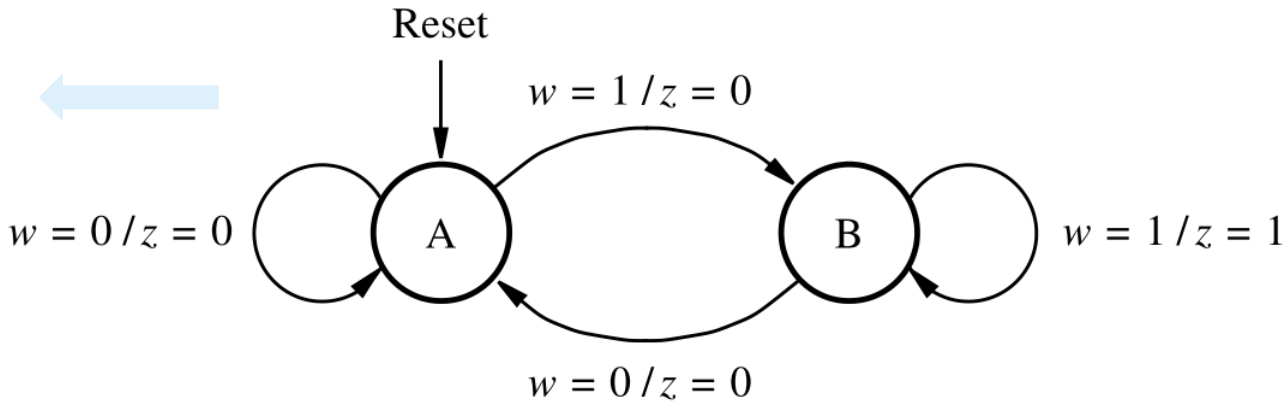
Mealy FSM – State/Output Table

From state diagram to Verilog, Contd.

- Let us fill in the state/output table

Current state S	Input	Next state	Output
S	w	S_next	z
A	0	A	0
A	1	B	0
B	0	A	0
B	1	B	1

Note: We cannot "compress" the table here, as every combination of (S, w) may lead to a different output z



Mealy FSM – Verilog

From state diagram to Verilog, Contd.

Current state S	Input	Next state	Output
S	w	S_next	z
A	0	A	0
A	1	B	0
B	0	A	0
B	1	B	1

```
module fsm_Mealy (
  input w, CLK, Reset,
  output reg z
);
reg S_next, S;
parameter A = 1'b0, B = 1'b1; // states
// Next-state logic
always @ (*) begin
  S_next = A;  // chosen as default
  case (S)
    A:  if (w == 0) S_next = A;
        else        S_next = B;
    B:  if (w == 0) S_next = A;
        else        S_next = B;
    default:        S_next = A;  // chosen as default
  endcase
end
// State memory
always @ (posedge CLK or posedge Reset) begin
  if (Reset == 1) S <= A;
  else S <= S_next;
end
// Output logic
always @ (*) begin
  if ((S == B) && (w == 1)) z = 1'b1;
  else z = 1'b0;
end
endmodule
```

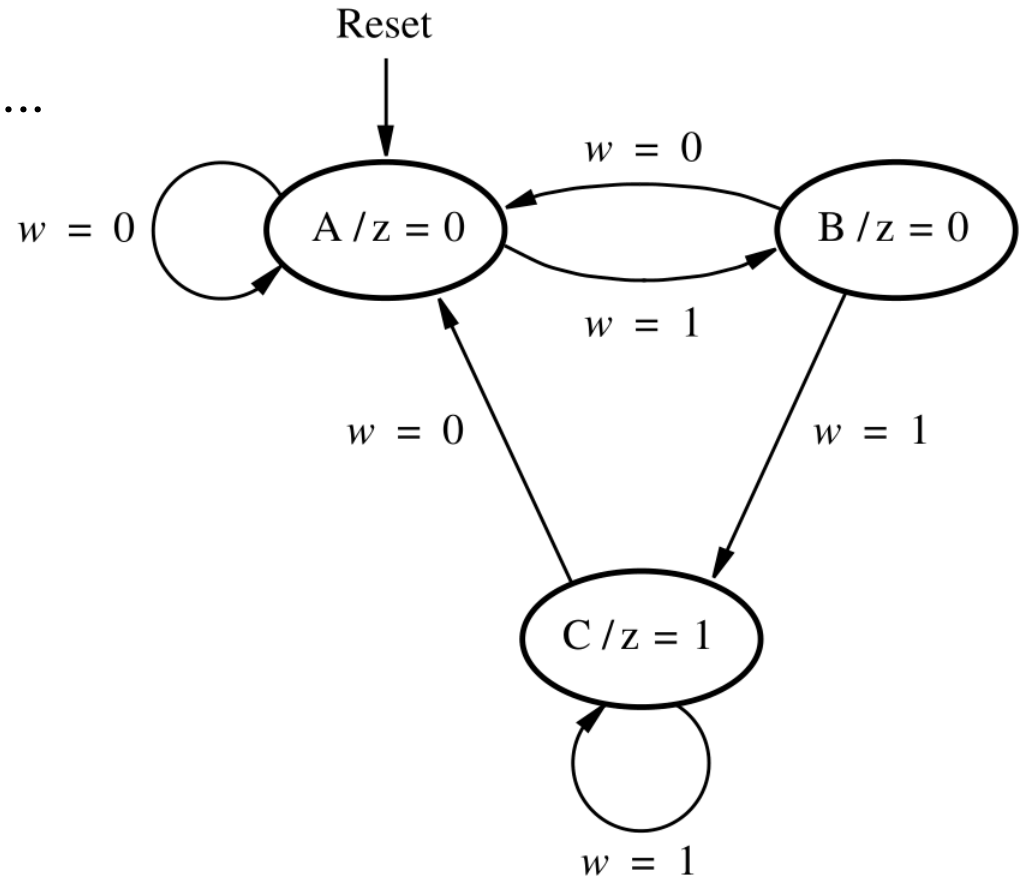
Moore FSM

Example

Moore FSM

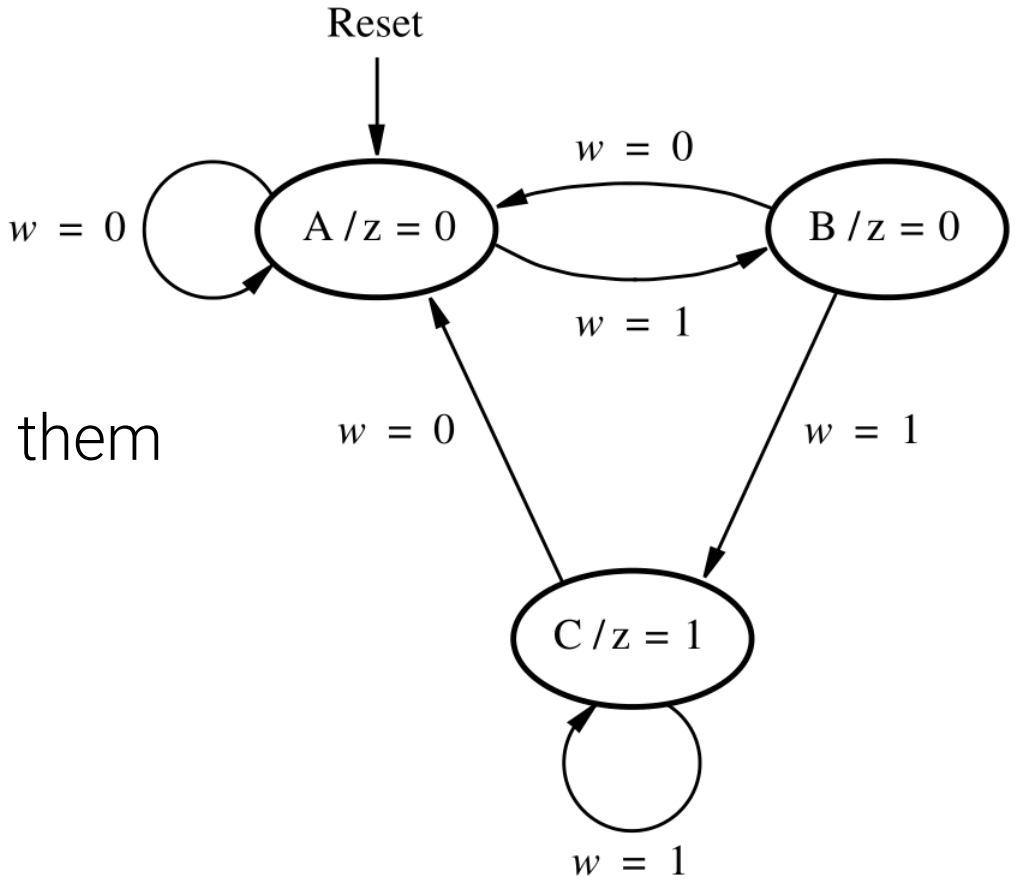
From state diagram to Verilog

- Figure shows the state diagram of…
  - Moore FSM
- Write the Verilog description



Moore FSM

From state diagram to Verilog, Contd.



- States are…
  - A, B, C;
  - At least 2 bits are required to present them
- Inputs are…
  - 1-bit input w
- Outputs are…
  - 1-bit z
- Reset input is…
  - Synchronous, active high
    - Strictly speaking, an edge from every state to A, in case of active Reset, could also be shown

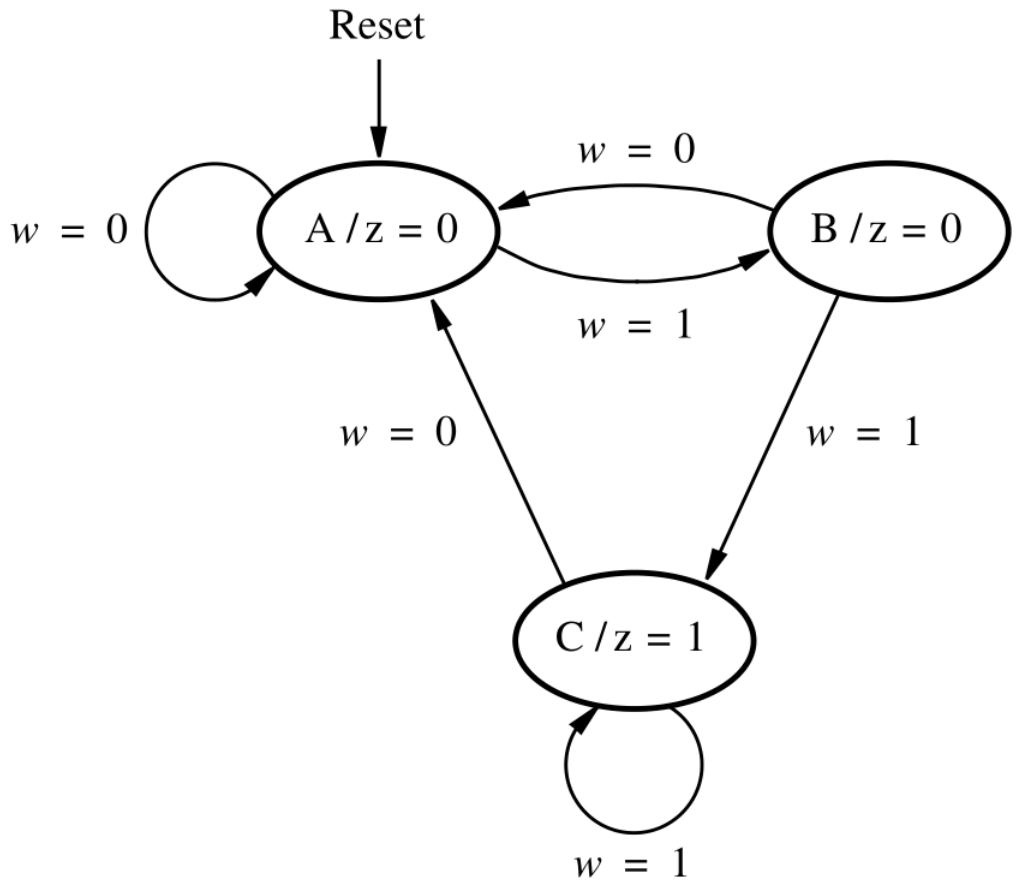
Moore FSM – State/Output Table

From state diagram to Verilog, Contd.

- Let us fill in the state/output table

Current state S	Next state S_next		Output
S	Input w		z
	0	1	
A	A	B	0
B	A	C	0
C	A	C	1

Note: We are able to "compress" the table here as the output z does not depend on the input w

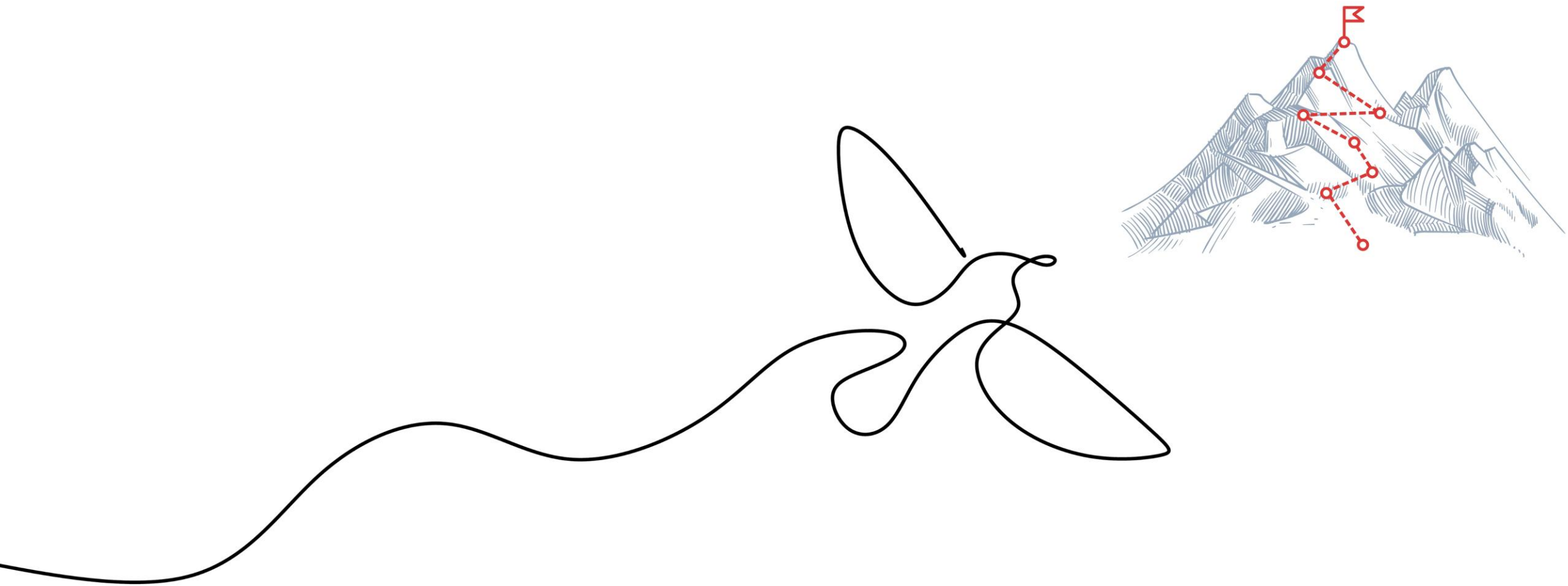


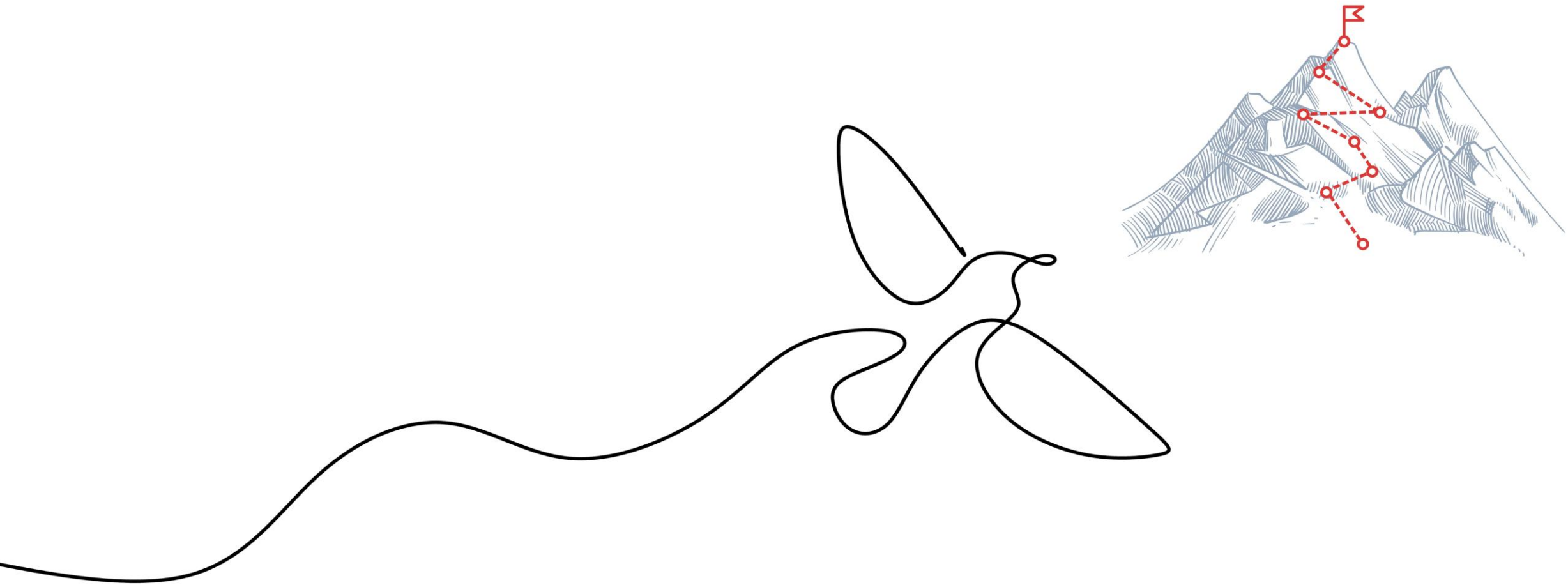
Moore FSM – Verilog

From state diagram to Verilog, Contd.

Current state S	Next state S_next		Output
S	Input w		z
	0	1	
A	A	B	0
B	A	C	0
C	A	C	1

```
module fsm_Moore (
  input w, CLK, Reset,
  output reg z);

  reg [1:0] S_next, S;
  parameter A = 2'b00, B = 2'b01, C = 2'b10; // states
  // Next-state logic
  always @ (*) begin
    S_next = A;  // chosen as default
    case (S)
      A: if (w == 0) S_next = A;
         else        S_next = B;
      B: if (w == 0) S_next = A;
         else        S_next = C;
      C: if (w == 0) S_next = A;
         else        S_next = C;
      default:       S_next = A;  // chosen as default
    endcase
  end
  // State memory
  always @ (posedge CLK) begin
    if (Reset == 1) S <= A; // effect of Reset active
    else S <= S_next;
  end
  // Output logic
  always @ (*) begin
    z = (S == C);
  end
endmodule
```





Serial Adder FSM

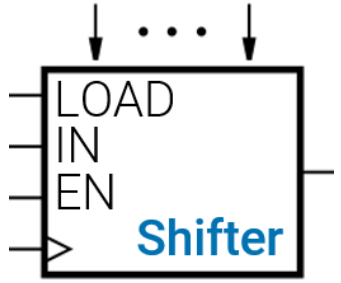
Example

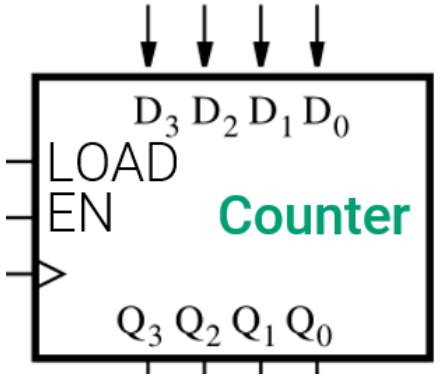
Serial Adders

- Fast adders are complex and expensive
- If speed is not a priority, one could consider a cost-effective alternative of adding a pair of bits at a time using a **serial adder**
  - **Serial adder sums one bit of each of the two operands and produces one bit of the sum per clock cycle**

Designing Serial Adder

- Suggested building blocks
  - Shift right register
    - Serial input: 1-bit IN
    - Parallel 8-bit input, taken when LOAD is active
    - Serial output: 1-bit
    - Enable input: EN, enables shifting
  - Down counter (**timer**)
    - Parallel 4-bit input, taken when LOAD is active
    - Enable input: EN, enables counting
    - Output **decreases** by one in every clock cycle

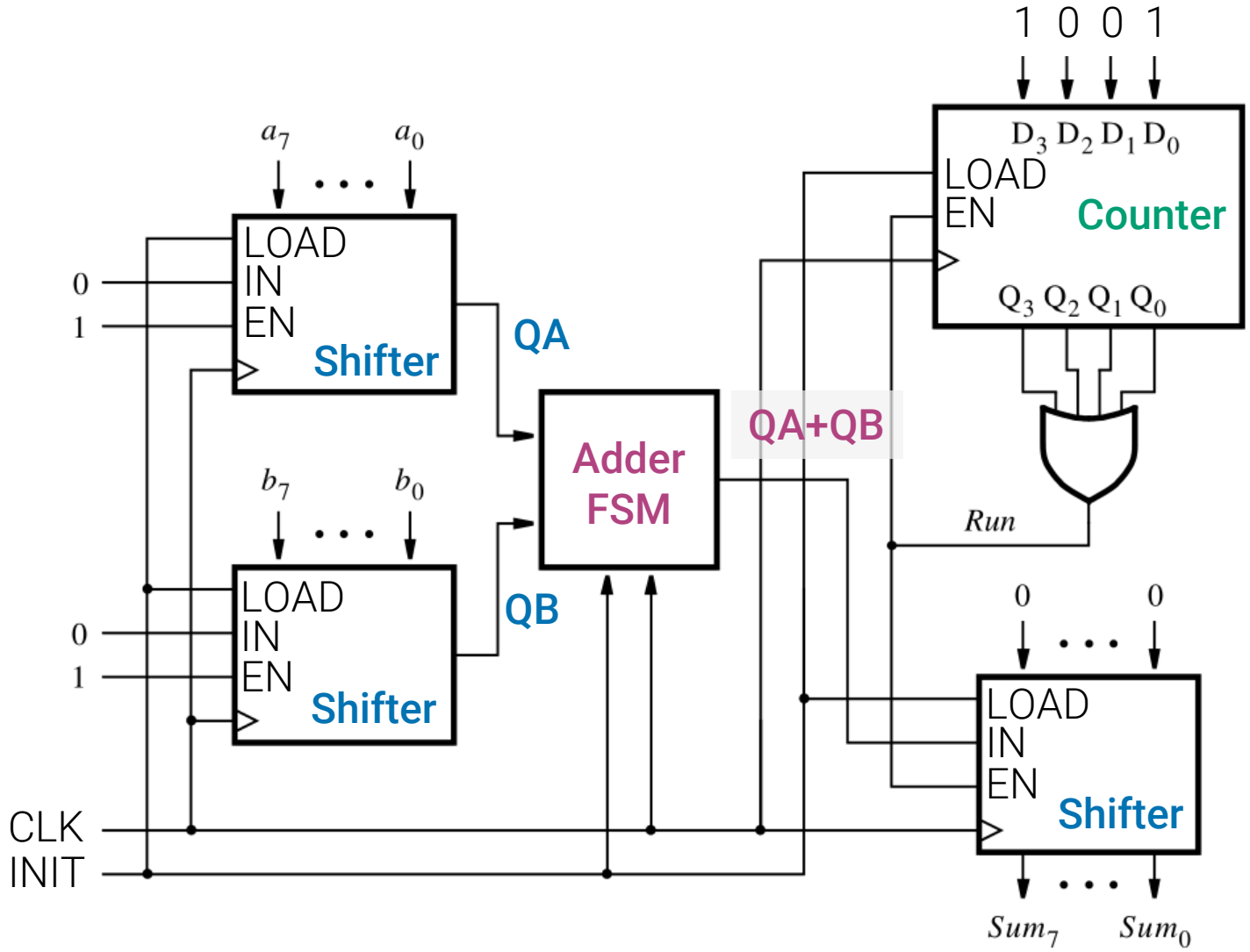




Serial Adder

Block Diagram

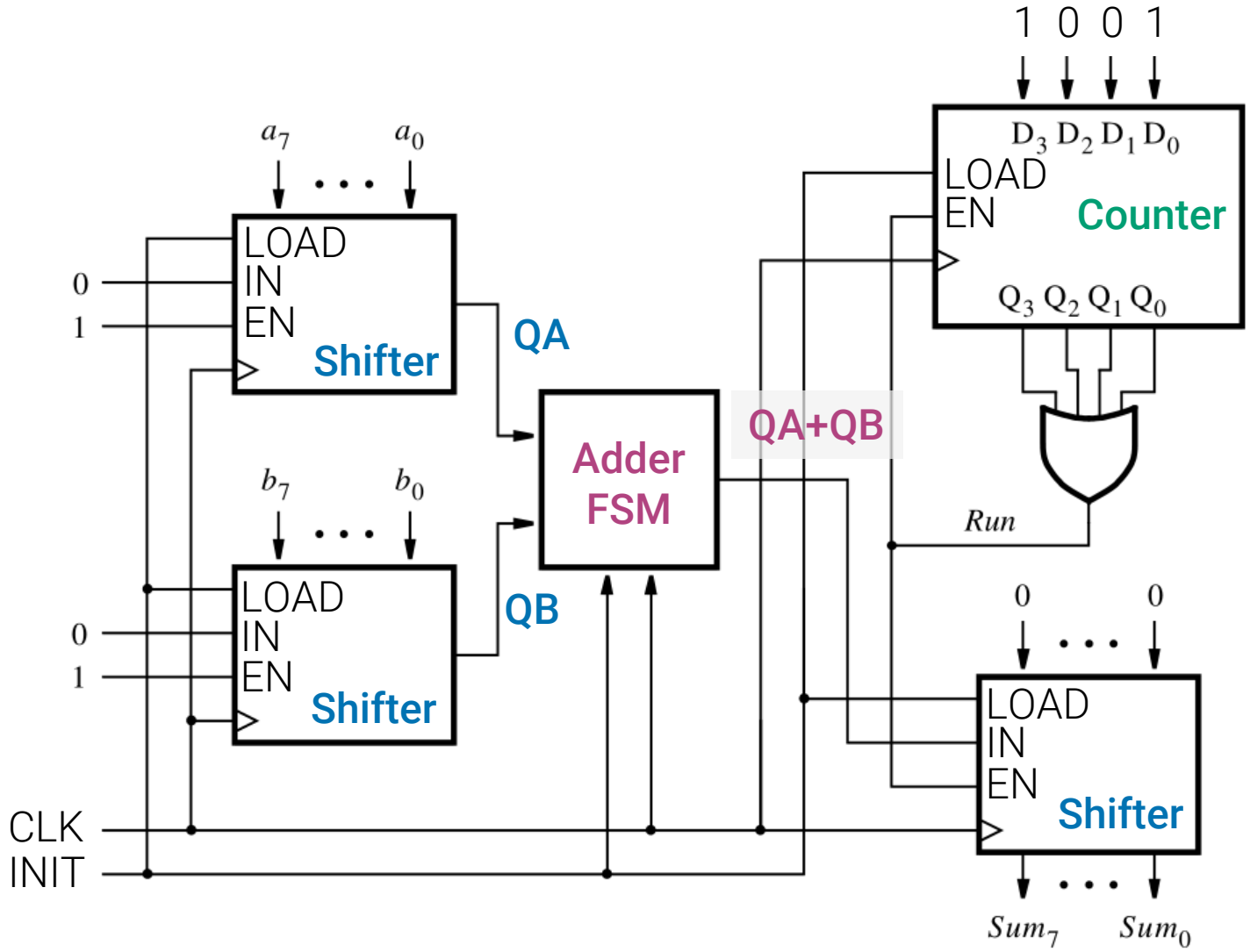
- Shift-right registers output one bit of operands **A** and **B** in every clock cycle
- Down-counter (timer) counts the number of cycles for addition (the number of pairs of bits to sum to obtain **S**)



Serial Adder

INIT

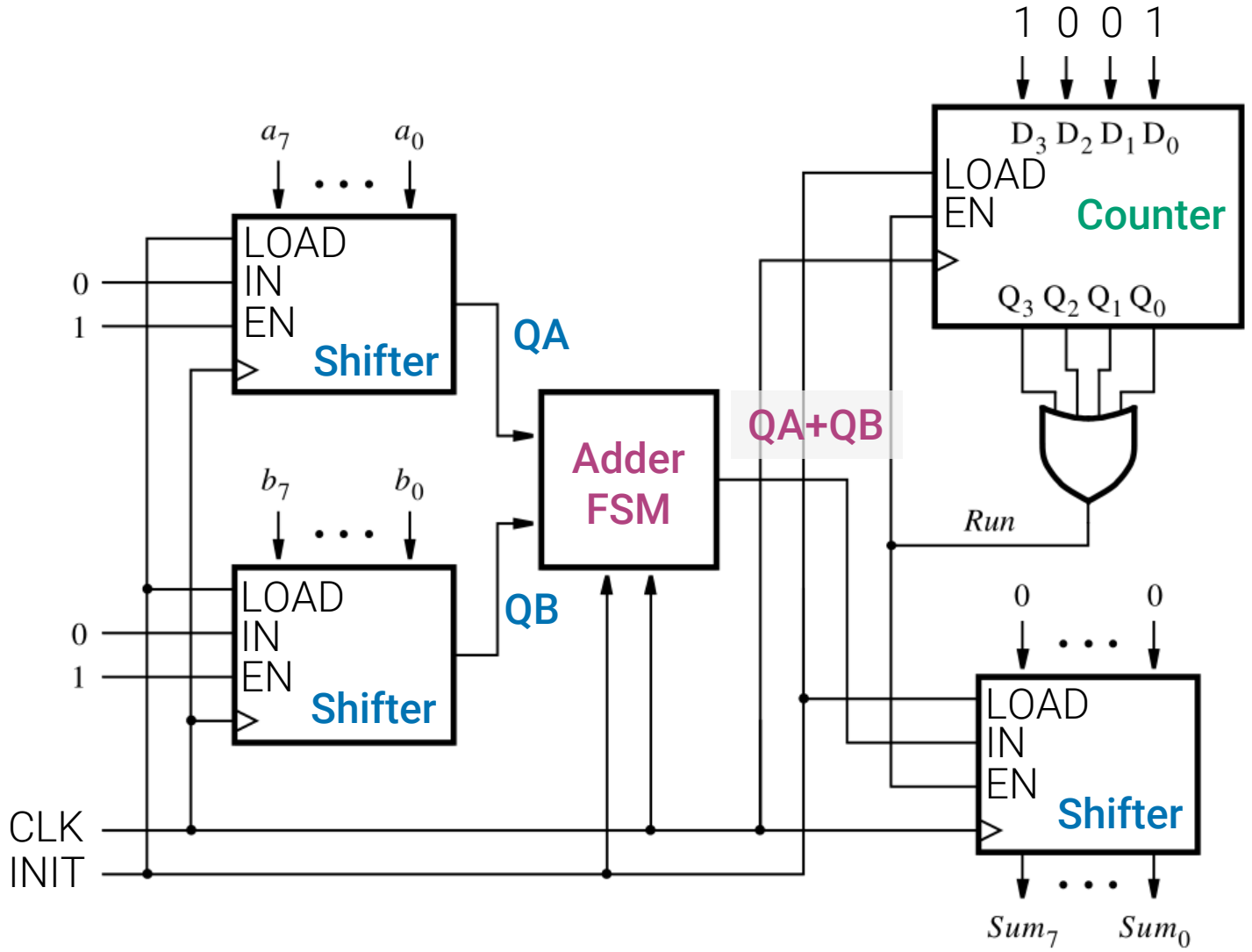
- **INIT** loads new operands and reinitializes the sum and the timer
  - Operands A and B initialized to new values
  - Sum S initialized to **zero**
  - Timer initialized to nine (the number of operand bits + 1), as that many cycles will be required to compute the sum



Serial Adder

Count Down

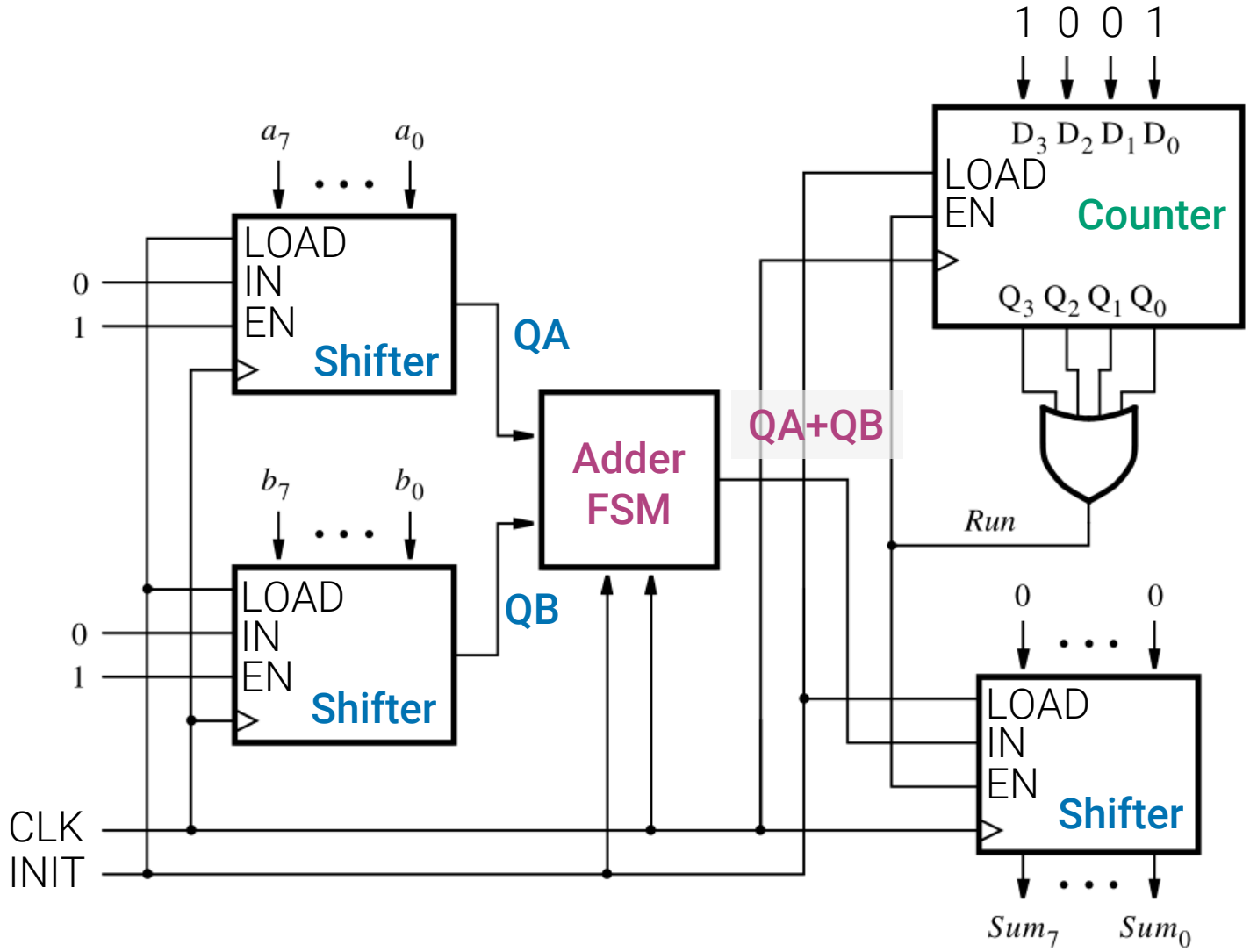
- Time's up when the output of the timer (down counter) drops to all zeros
- Sequence at the output of the counter
  - 1001 -> 1000 -> 0111 -> 0110 -> -> 0101 -> 0100 -> -> 0011 -> 0010 -> -> 0001 -> 0000
  - **Run** signal becomes '0' after nine clock cycles



Serial Adder

Count Down, Contd.

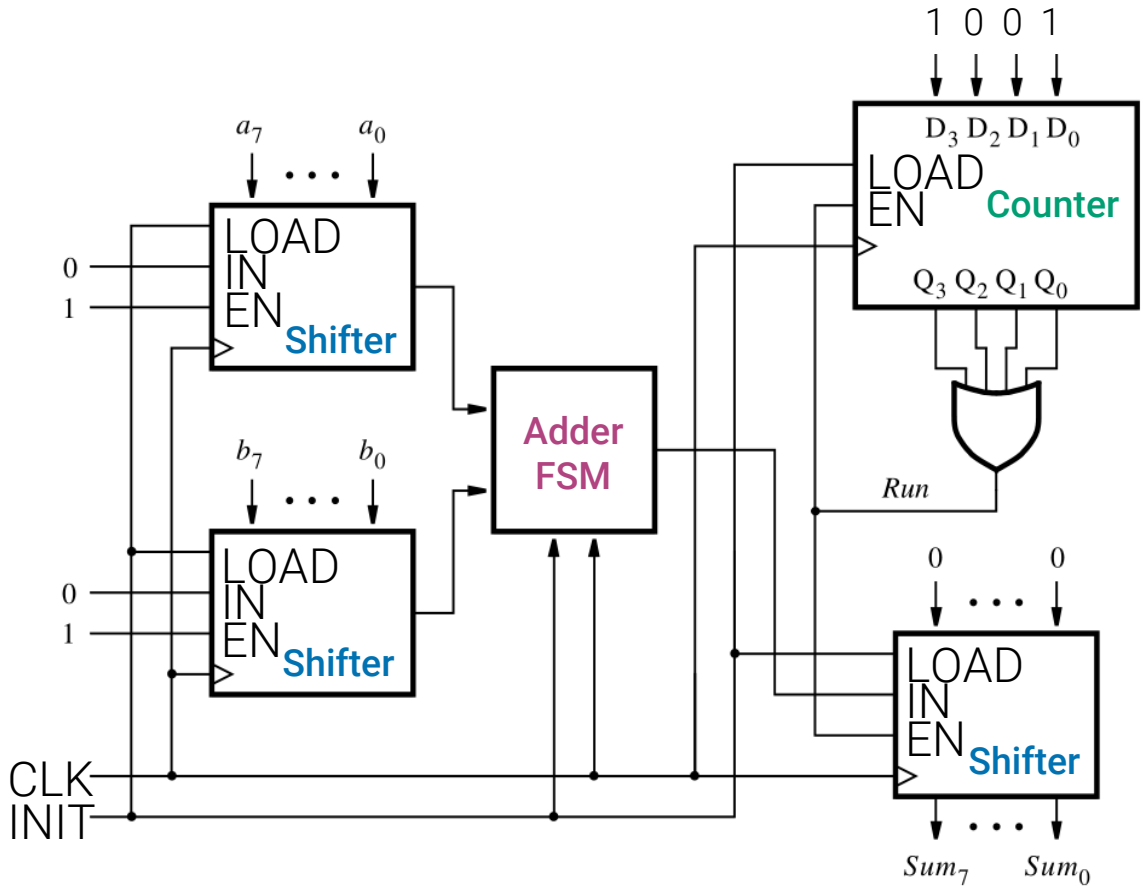
- While **Run** = ‘1’
  - The timer and the sum register are enabled (EN = '1')
  - The counter counts (time passes, clock cycles elapse) and the sum register receives one new bit every clock cycle
- While **Run** = ‘0’
  - The timer and the sum register are disabled (EN = '0')
  - The counter does not count
  - The sum **S** does not change



Serial Adder

Operation, Summary

- Initialize the shift registers and the counter
  - Load (in parallel) vectors A and B in their respective shift registers
  - Load the initial value in the timer
  - Clear the contents of the sum register
- In each clock cycle
  - Add a pair of bits using the **Adder FSM**
  - Shift the resulting sum bit into the output register
  - Shift the input vectors A and B to release the next pair of bits to be added
  - Decrement the counter (count down…)
- Stop when the timer finishes counting
  - The number of clock cycles should be chosen so that all eight bits of the sum get stored in the corresponding register, one per cycle



Serial Adder

Example Sum

- Consider A = $(0010\ 1011)_2$ and B = $(0011\ 0010)_2$
- The expected sum S is $(0101\ 1101)_2$
- Sequence at the output of the sum register
  - 0000 0000 -> **1**000 0000
    -> **0**100 0000 -> **1**010 0000
    -> **1**101 0000 -> **1**110 1000
    -> **0**111 0100 -> **1**011 1010
    -> **0**101 1101
  - The initial value is all zeros
  - Eight clock cycles are required to shift in the eight bits of the sum

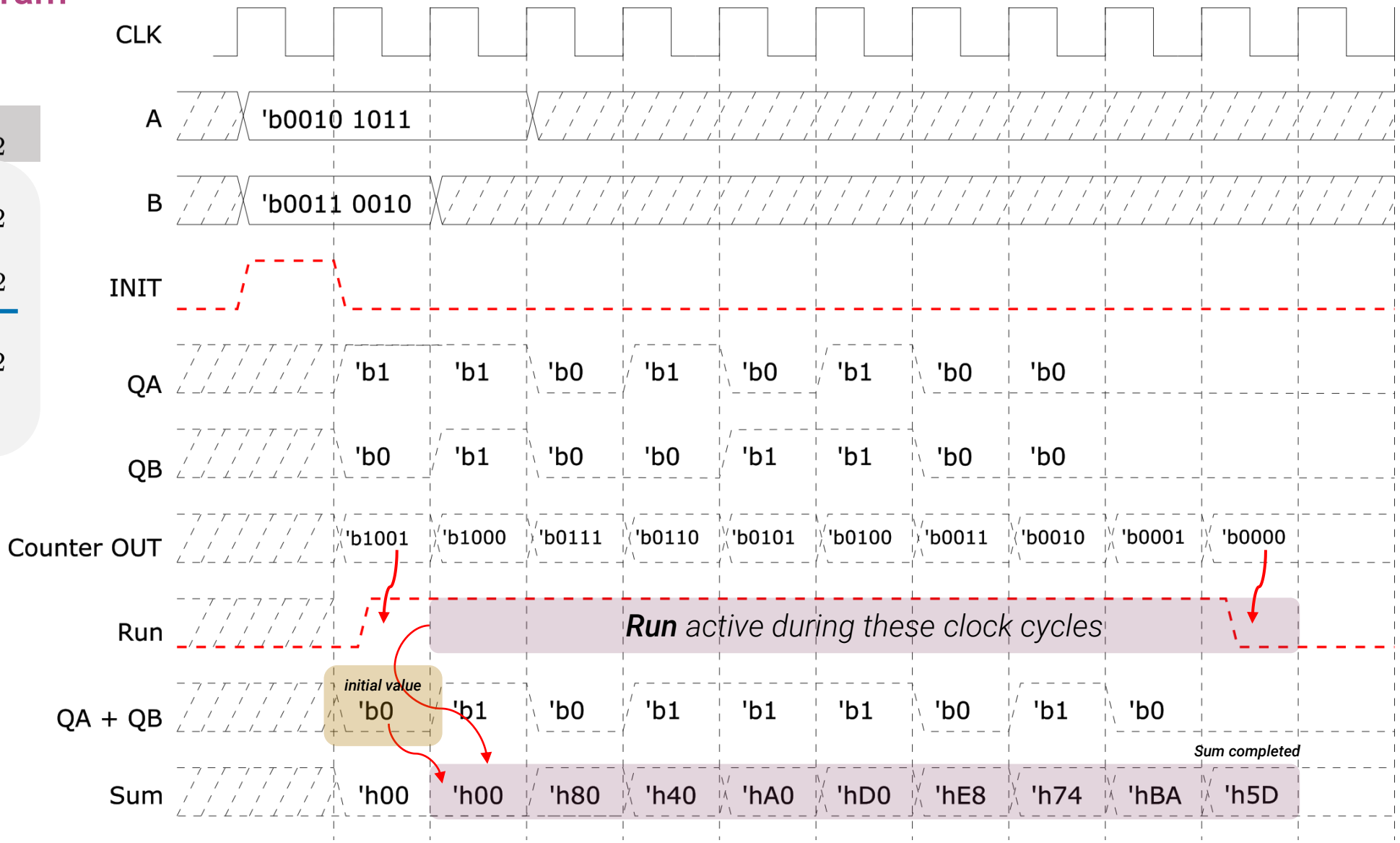
$$\begin{array}{r} (0100\ 0100)_2 \quad \text{Carry} \\ (0010\ 1011)_2 \\ +\ (0011\ 0010)_2 \\ \hline (0101\ 1101)_2 \\ = (5\mathrm{D})_{16} \end{array}$$

Serial Adder

Timing Diagram

Carry $(0100\ 0100)_2$

$$\begin{array}{r} (0010\ 1011)_2 \\ +\ (0011\ 0010)_2 \\ \hline (0101\ 1101)_2 \\ = (5D)_{16} \end{array}$$

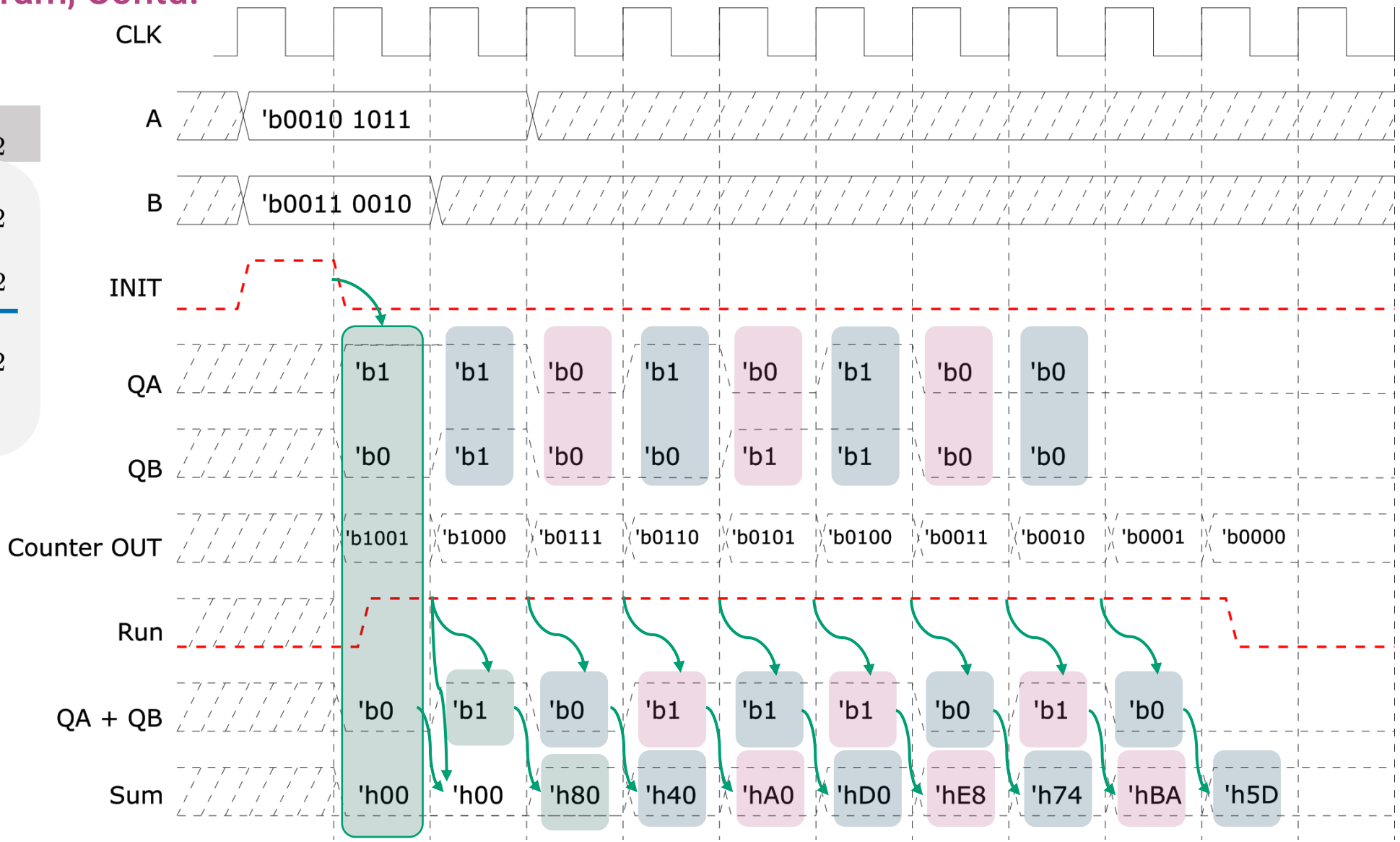


Serial Adder

Timing Diagram, Contd.

Carry $(0100\ 0100)_2$

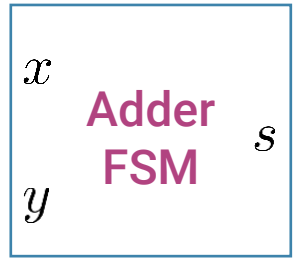
$$\begin{array}{r} (0010\ 1011)_2 \\ +\ (0011\ 0010)_2 \\ \hline (0101\ 1101)_2 \\ = (5\mathrm{D})_{16} \end{array}$$



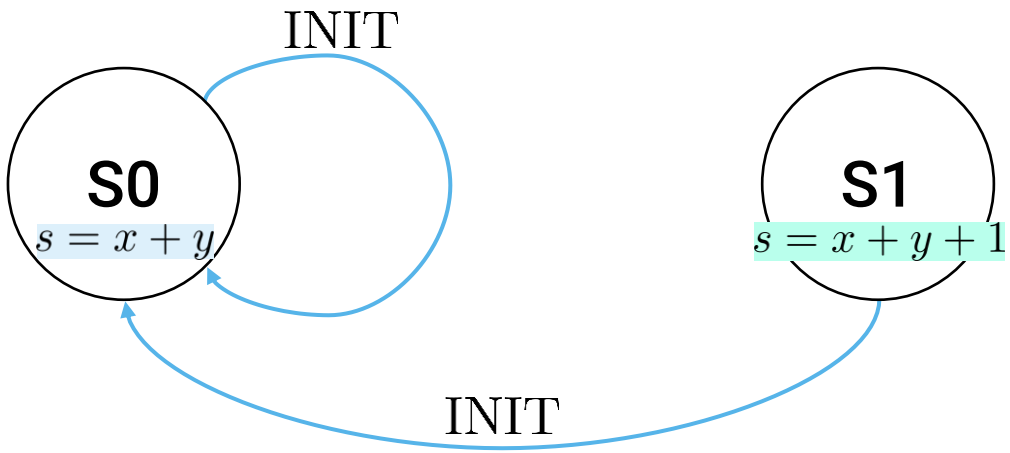
Serial Adder FSM

When INIT is Active

- Adder FSM needs only two states
  - S0: when carry is '0'
  - S1: when carry is '1'



$$s = x + y + carry$$

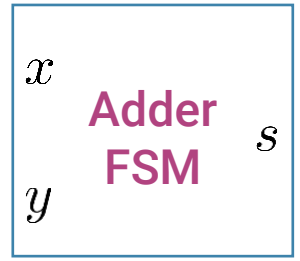


Note: INIT active means that new operands are loaded and new addition can commence; carry-in is therefore 0, and so the state machine must remain in or return to the S0 state (in which carry-in is '0');

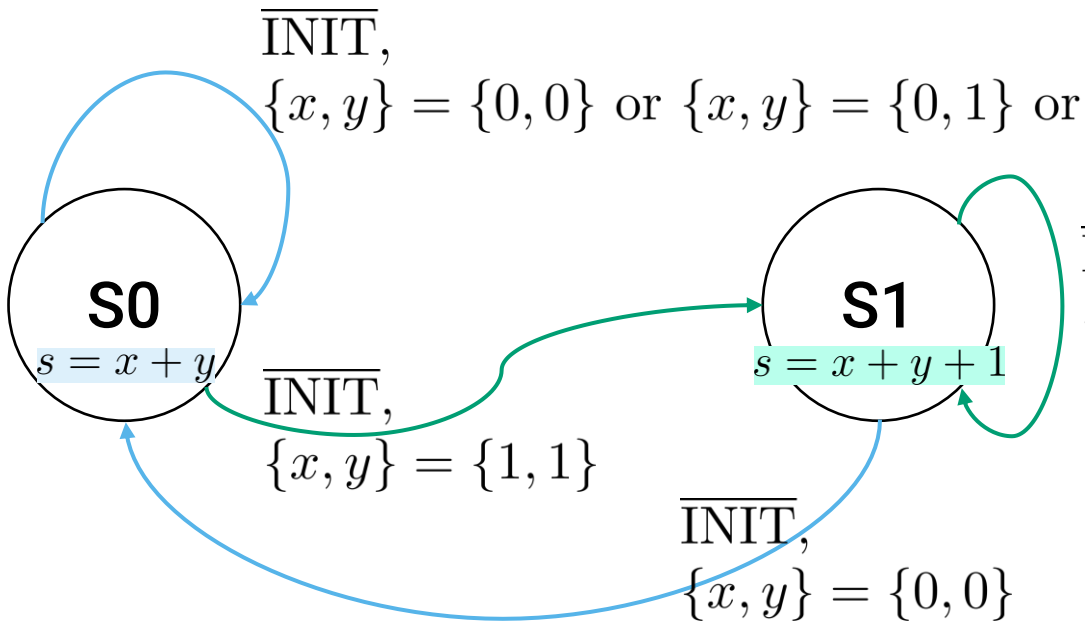
Serial Adder FSM

When INIT is Inactive

- Adder FSM needs only two states
  - S0: when carry is '0'
  - S1: when carry is '1'



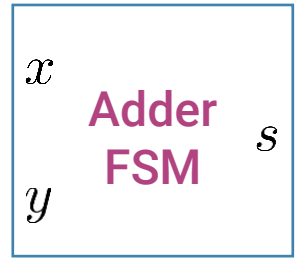
$$s = x + y + carry$$



Note: INIT inactive means the addition is in process

Serial Adder FSM, Contd.

- Adder FSM with two states
  - S0: when carry is '0'
  - S1: when carry is '1'
- State/output tables



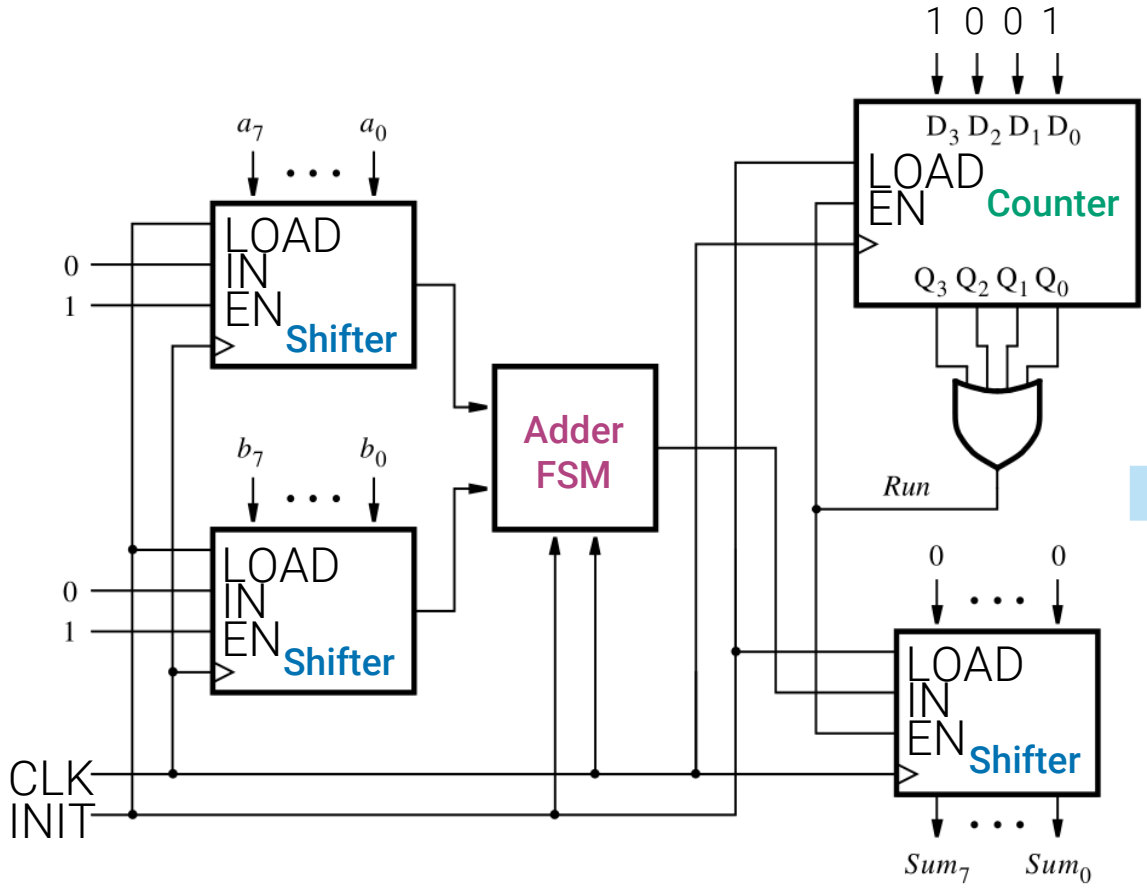
$$s = x + y + carry$$

INIT = 0								
Present State **S**	Next state **S_next**				Output **s**			
	One-bit Inputs **xy**				One-bit Inputs **xy**			
	00	01	10	11	00	01	10	11
S0	S0	S0	S0	S1	0	1	1	0
S1	S0	S1	S1	S1	1	0	0	1

INIT = 1								
Present State **S**	Next state **S_next**				Output **s**			
	One-bit Inputs **xy**				One-bit Inputs **xy**			
	00	01	10	11	00	01	10	11
S0	S0	S0	S0	S0	0	1	1	0
S1	S0	S0	S0	S0	1	0	0	1

Serial Adder FSM

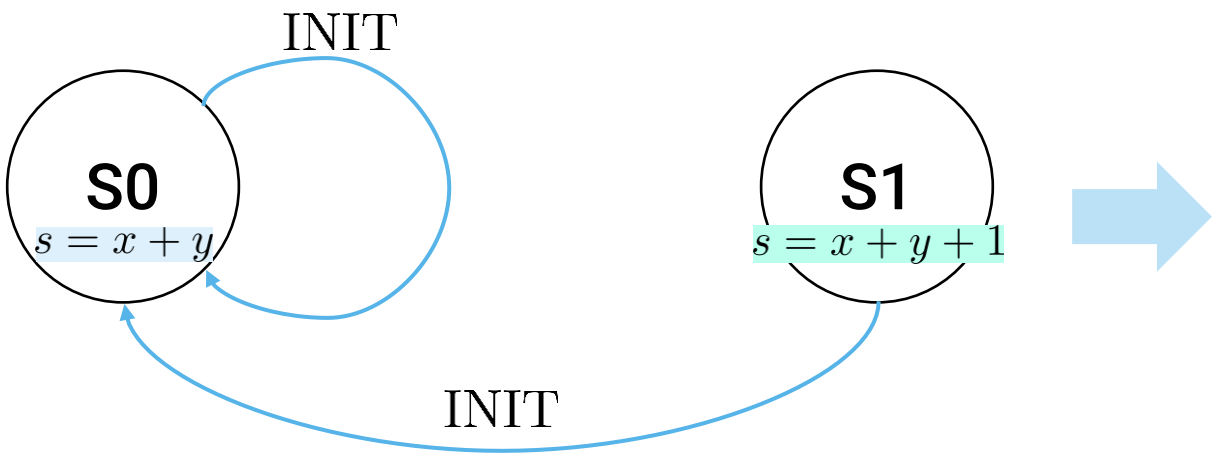
In Verilog




```
module serialadderfsm (a, b, init, clk, sum);
  parameter n = 8;
  input a, b, init, clk;
  output reg sum;
  parameter S0 = 1'b0, S1 = 1'b1;
  reg S_next, S;
  // Next-state logic
  always @(*) begin




  end
```

Serial Adder FSM

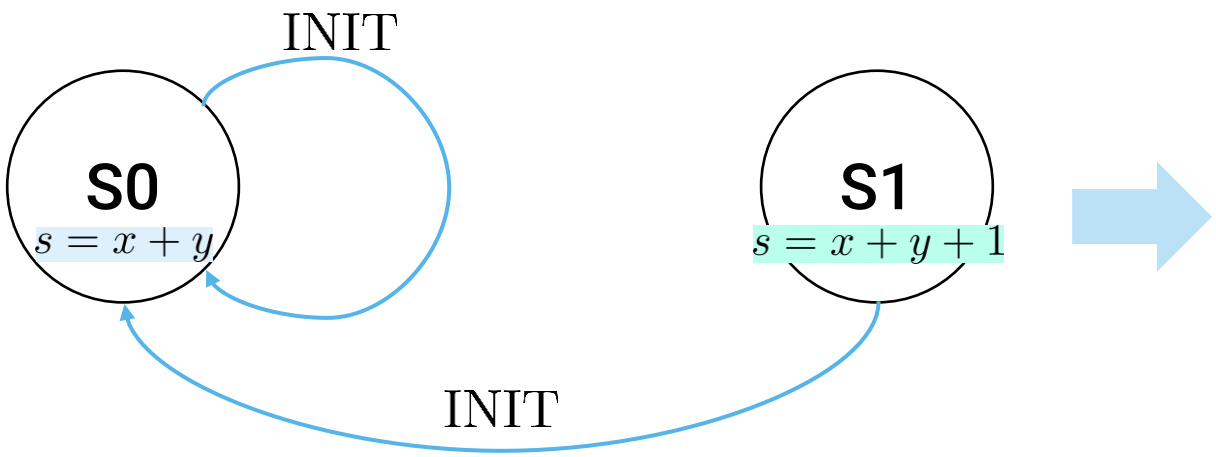
In Verilog




```
module serialadderfsm (a, b, init, clk, sum);
  parameter n = 8;
  input a, b, init, clk;
  output reg sum;
  parameter S0 = 1'b0, S1 = 1'b1;
  reg S_next, S;
  // Next-state logic
  always @(*) begin




  end
```

Serial Adder FSM

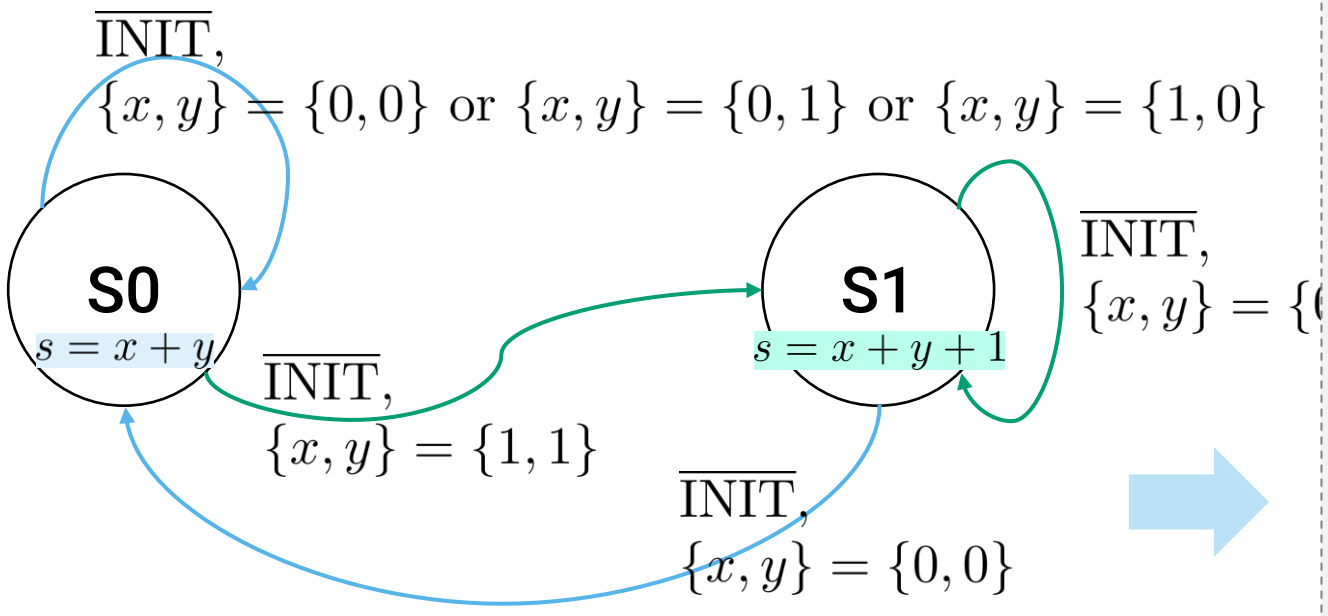
In Verilog




```
module serialadderfsm (a, b, init, clk, sum);
  parameter n = 8;
  input a, b, init, clk;
  output reg sum;
  parameter S0 = 1'b0, S1 = 1'b1;
  reg S_next, S;
  // Next-state logic
  always @(*) begin
    S_next = S0;
    case (S)
      S0: begin
        if (init)        S_next = S0;
        else if (a & b) S_next = S1;
        else             S_next = S0;
      end
      S1: begin
        if (init)          S_next = S0;
        else if (~a & ~b) S_next = S0;
        else               S_next = S1;
      end
      default: S_next = S0;
    endcase
  end
```

Serial Adder FSM

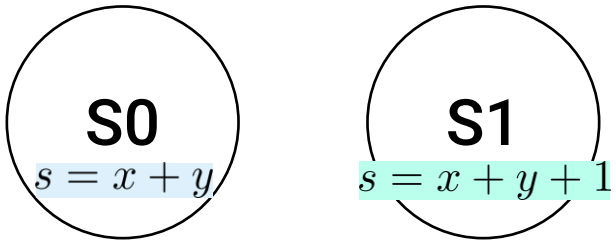
In Verilog



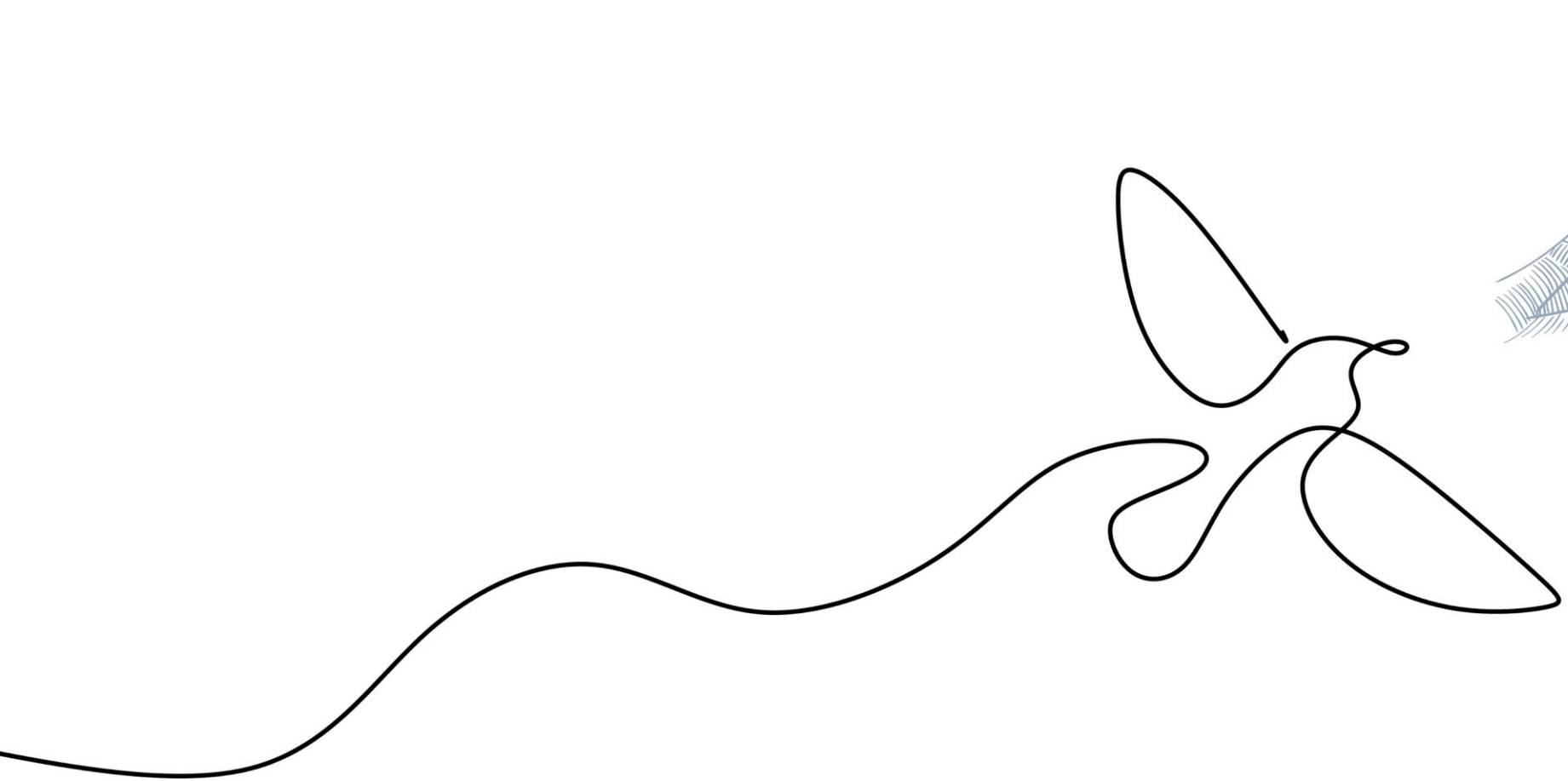

```
module serialadderfsm (a, b, init, clk, sum);
  parameter n = 8;
  input a, b, init, clk;
  output reg sum;
  parameter S0 = 1'b0, S1 = 1'b1;
  reg S_next, S;
  // Next-state logic
  always @(*) begin
    S_next = S0;
    case (S)
      S0: begin
        if (init)          S_next = S0;
        else if (a & b)    S_next = S1;
        else               S_next = S0;
      end
      S1: begin
        if (init)          S_next = S0;
        else if (~a & ~b)  S_next = S0;
        else               S_next = S1;
      end
      default: S_next = S0;
    endcase
  end
```

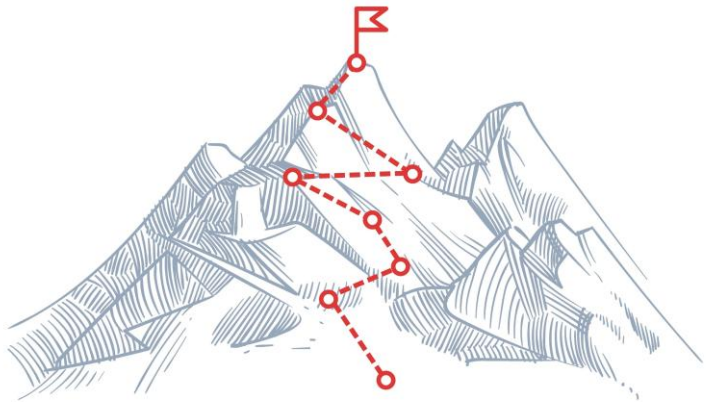
Serial Adder FSM

In Verilog

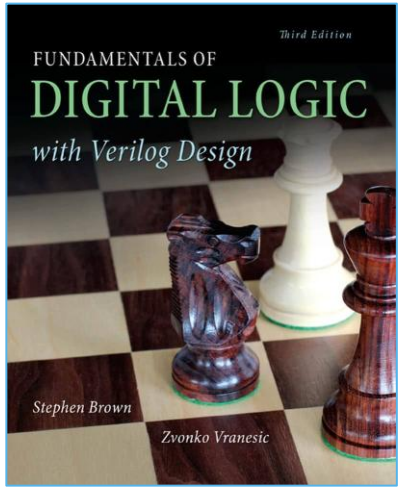



```
// State memory
// Synchronous reset
always @ (posedge clk) begin
  if (init)  S <= S0;
  else       S <= S_next;
end

// Output logic
always @(*) begin
  if (S == S0) sum = a ^ b;
  else         sum = ~(a ^ b);
end

endmodule
```





Literature



- Appendix A: Verilog Reference
  - A.14.8, A.14.9
- Chapter 6: Synchronous Sequential Circuits
  - 6.5 Serial Adder Example