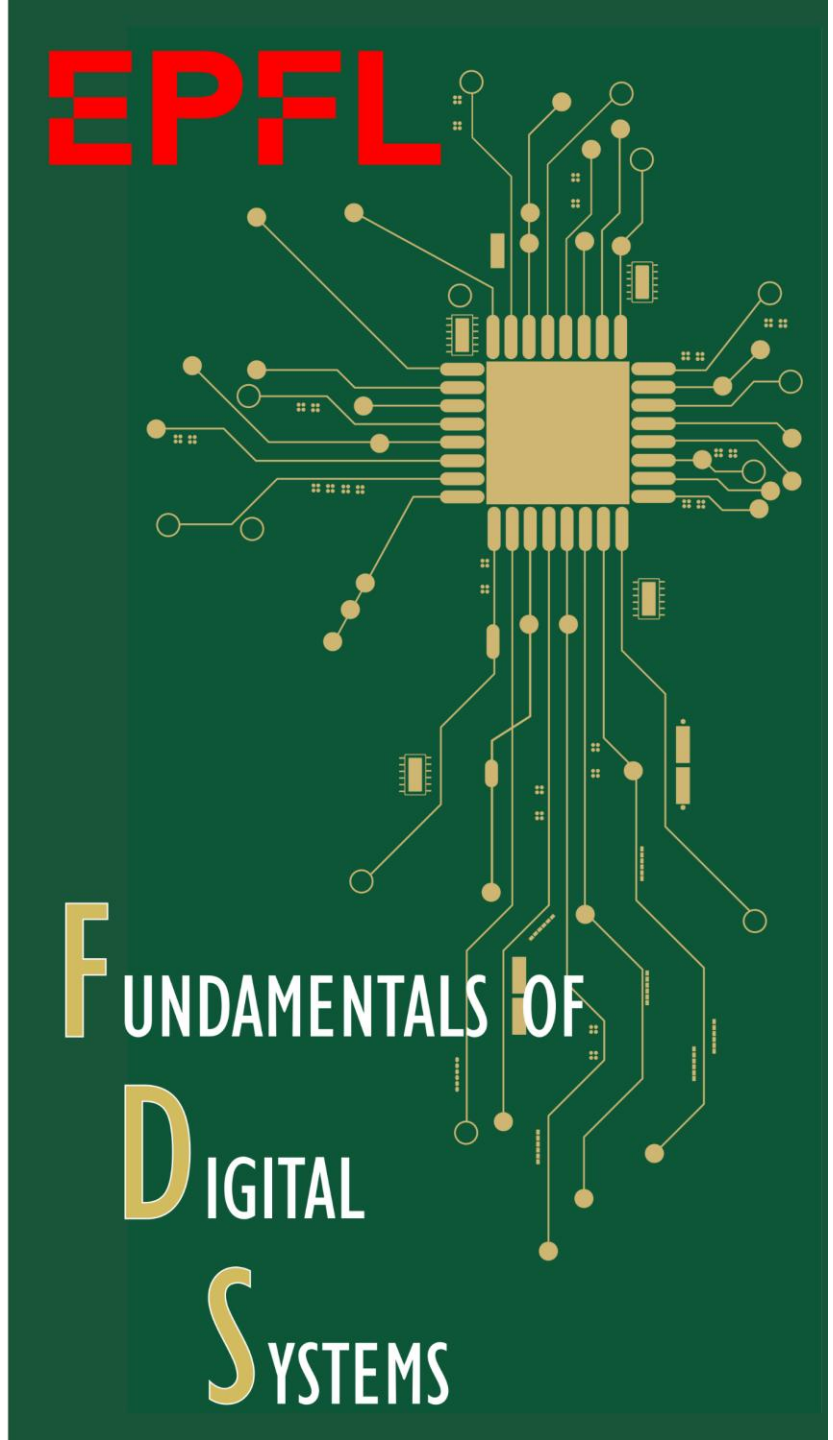


Digital Logic Circuits

Digital System Design

CS-173 Fundamentals of Digital Systems

Mirjana Stojilović

Spring 2025

Previously on FDS

Examples of FSMs

Let’s Talk About…

Some More Digital Circuit Design Examples

Quick Outline

- Digital systems with buses
  Example: Swapping two registers
  - Bus with tri-state drivers
  - Bus with a MUX
- Verilog:
  - Reduction operators
  - Generate construct
    - RCA with a `for` loop
    - RCA with a `generate` construct

Recall: Bus

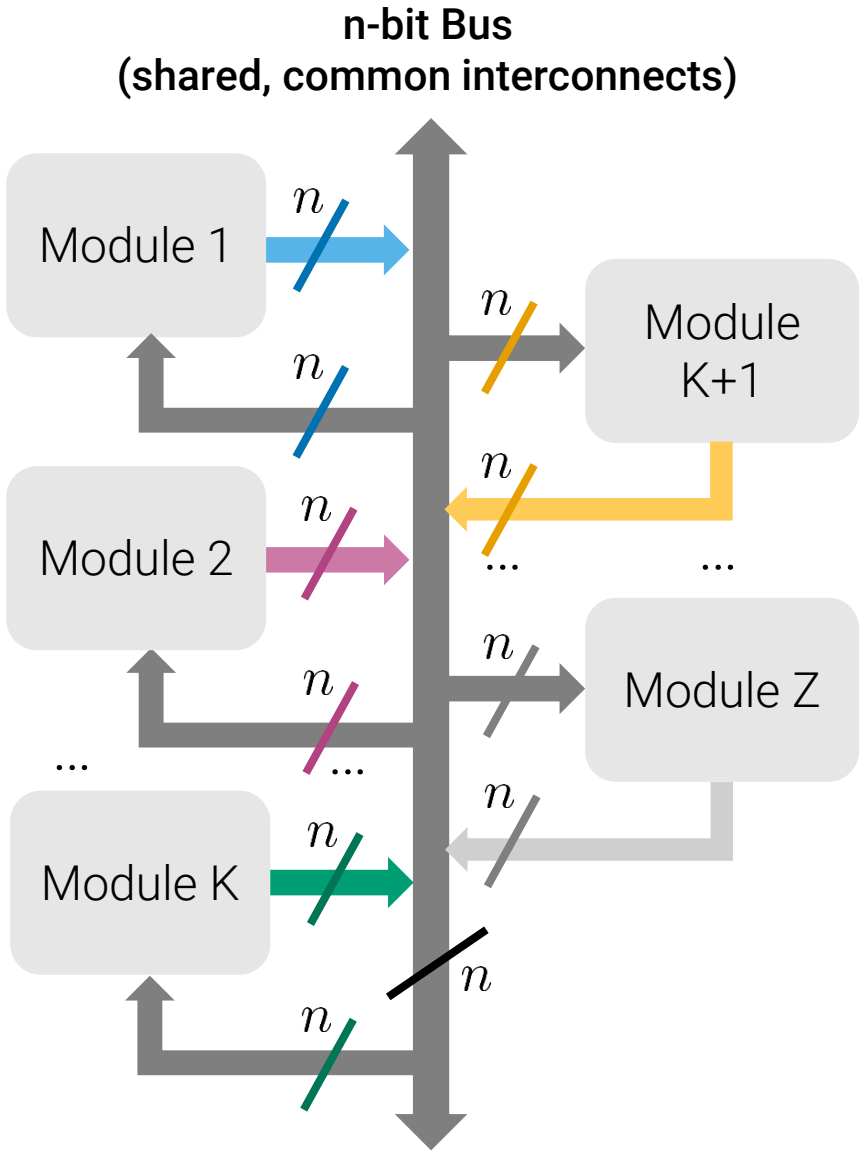
In Verilog

- with a MUX
- with tri-state drivers

Recall: Bus

Previously on FDS

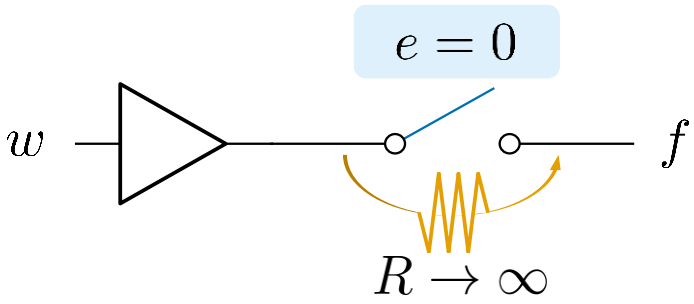
- Digital systems are commonly composed of several modules exchanging data by means of a **common** set of wires
- This shared set of wires is referred to as a **bus**
- Bus receives data from one or more modules—one at a time—and brings it to the inputs of one or more modules



Note: Optional feedback paths

Recall: Tri-State Drivers

Previously on FDS



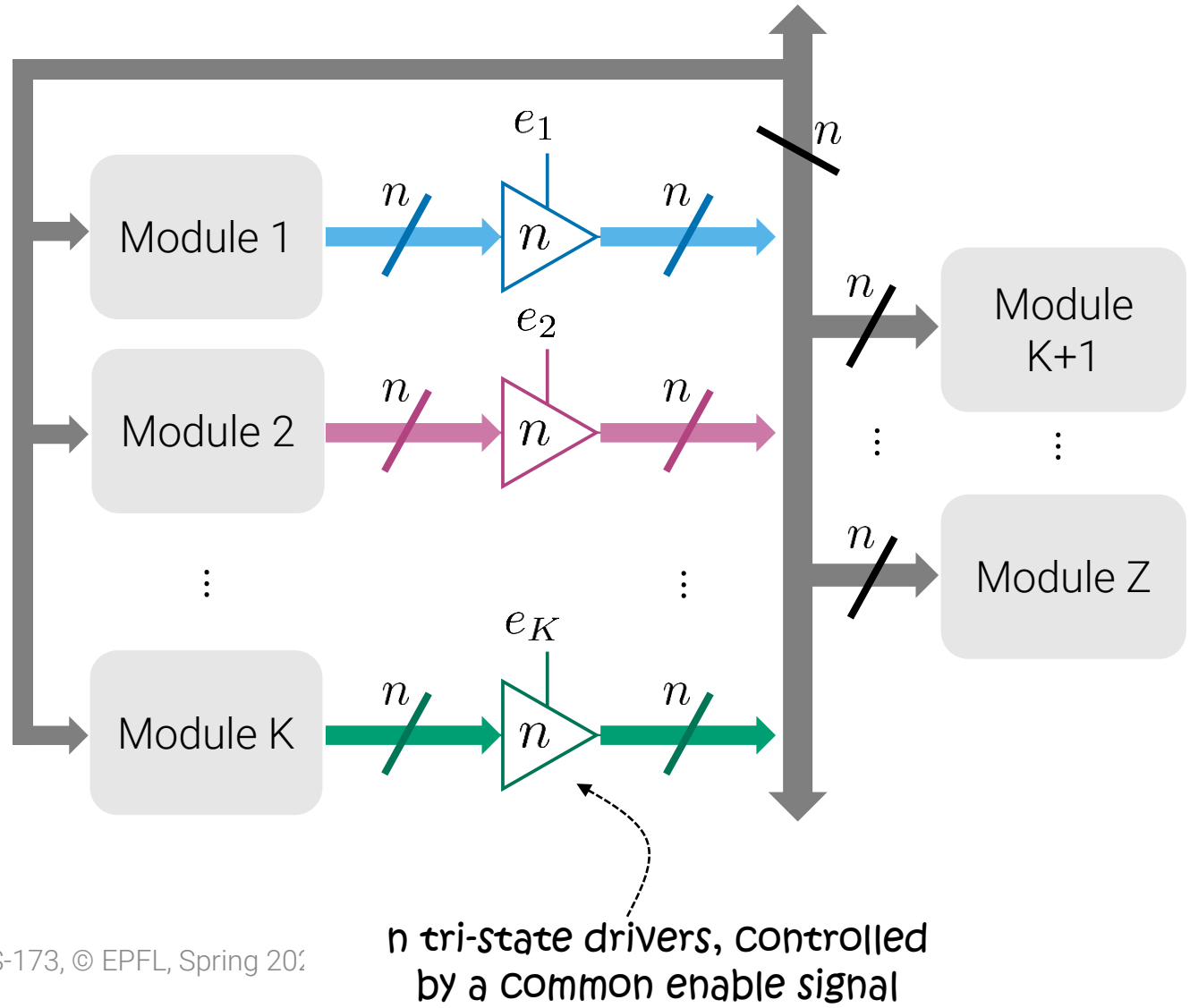
e	w	f
0	0	**Z**
0	1	**Z**
1	0	**0**
1	1	**1**

- When the enable input is inactive, the output is electrically disconnected from the data input; disconnected state is referred to as **high-impedance** state and usually denoted as **Z** (or **z**)
  - Three states of a tri-state driver are logical 0, logical 1, and Z

- *In electrical engineering, **impedance** is the opposition to alternating current presented by the combined effect of resistance and reactance in a circuit.*
- *https://en.wikipedia.org/wiki/Electrical_impedance*

Recall: Bus With Tri-State Drivers

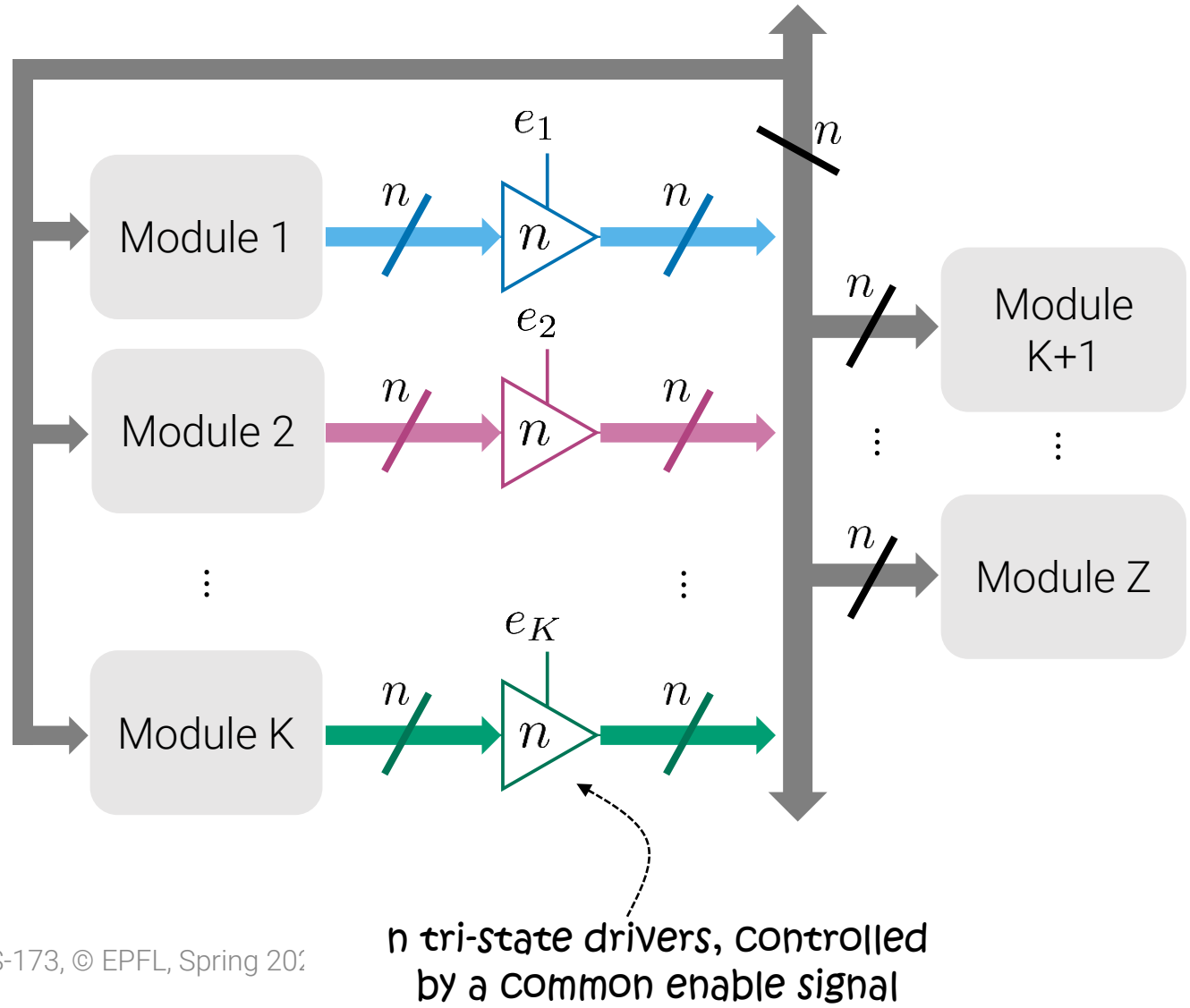
Previously on FDS



- Only **one** of the enable signals is active **at a time** so that short circuits are avoided
- There is another module, a controller FSM, responsible for the activation of the select signals (not shown)

Recall: Bus With Tri-State Drivers

Previously on FDS



- Bus implemented with tri-state drivers is less common today; it is used when one expects additional modules will be added to the system in the future

Bus with Tri-State Drivers

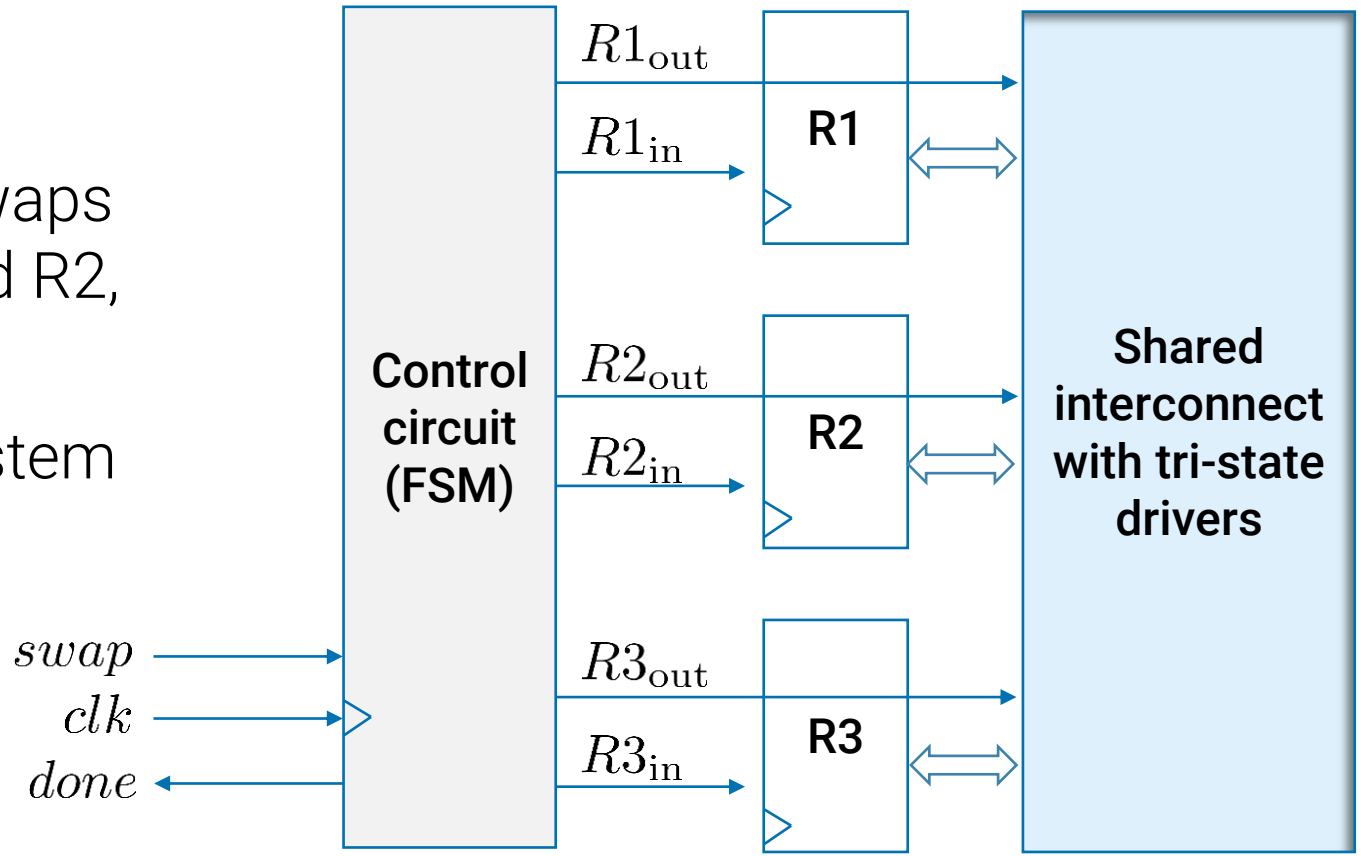
Example: Swapping Two Registers

Swapping Two Registers

Bus with Tri-State Drivers

Consider a system with three registers: R1, R2, and R3

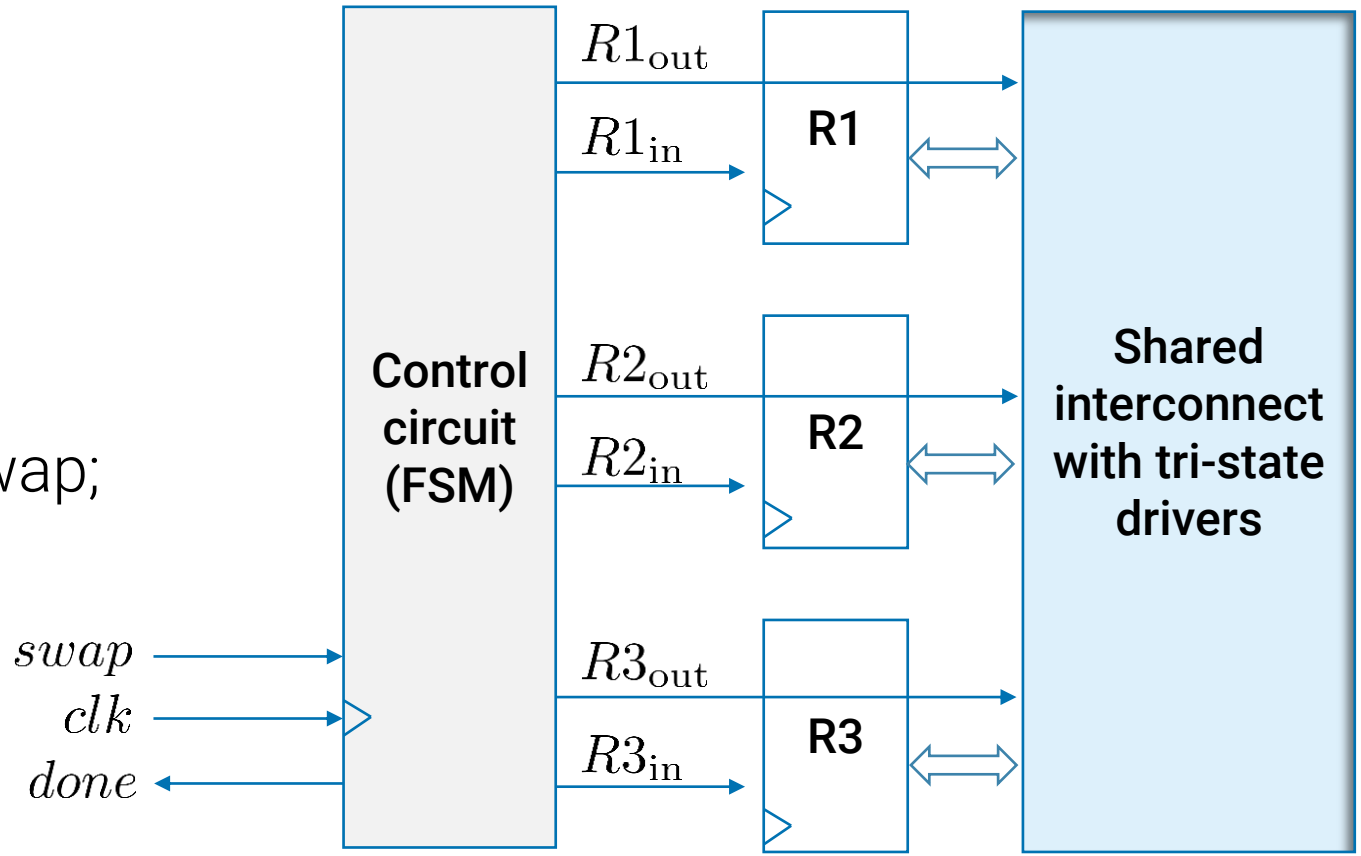
- Design a controller FSM that swaps the contents of registers R1 and R2, using R3 for temporary storage
- Write a Verilog model of the system



Swapping Two Registers

FSM Ports

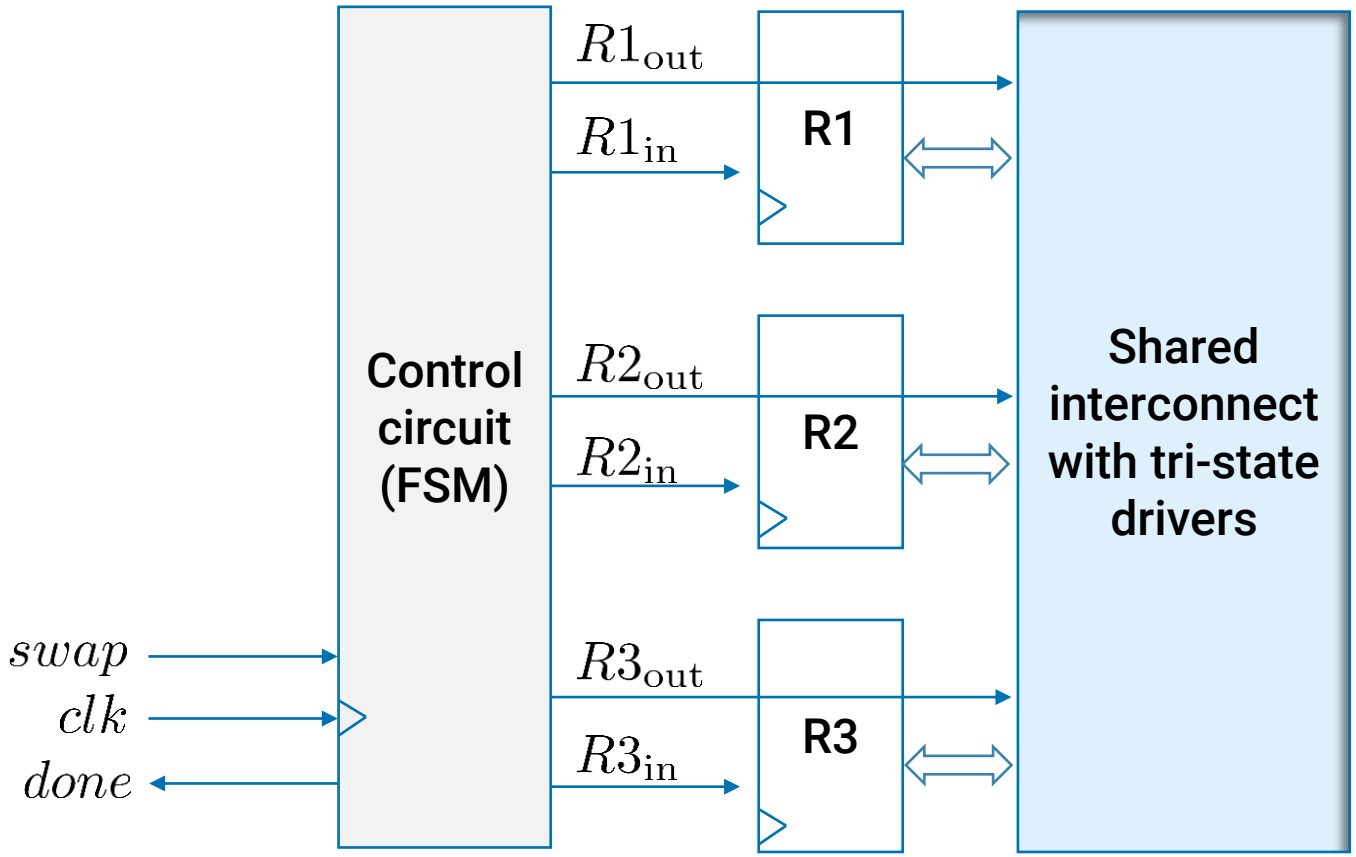
- **swap**
  - Control signal triggering (initiating) the swap
  - Input
- **done**
  - '1' (high) at the end of the swap; '0' (low), otherwise
  - Output
- Synchronous power-on reset
  - Input, not shown



Swapping Two Registers

FSM Ports

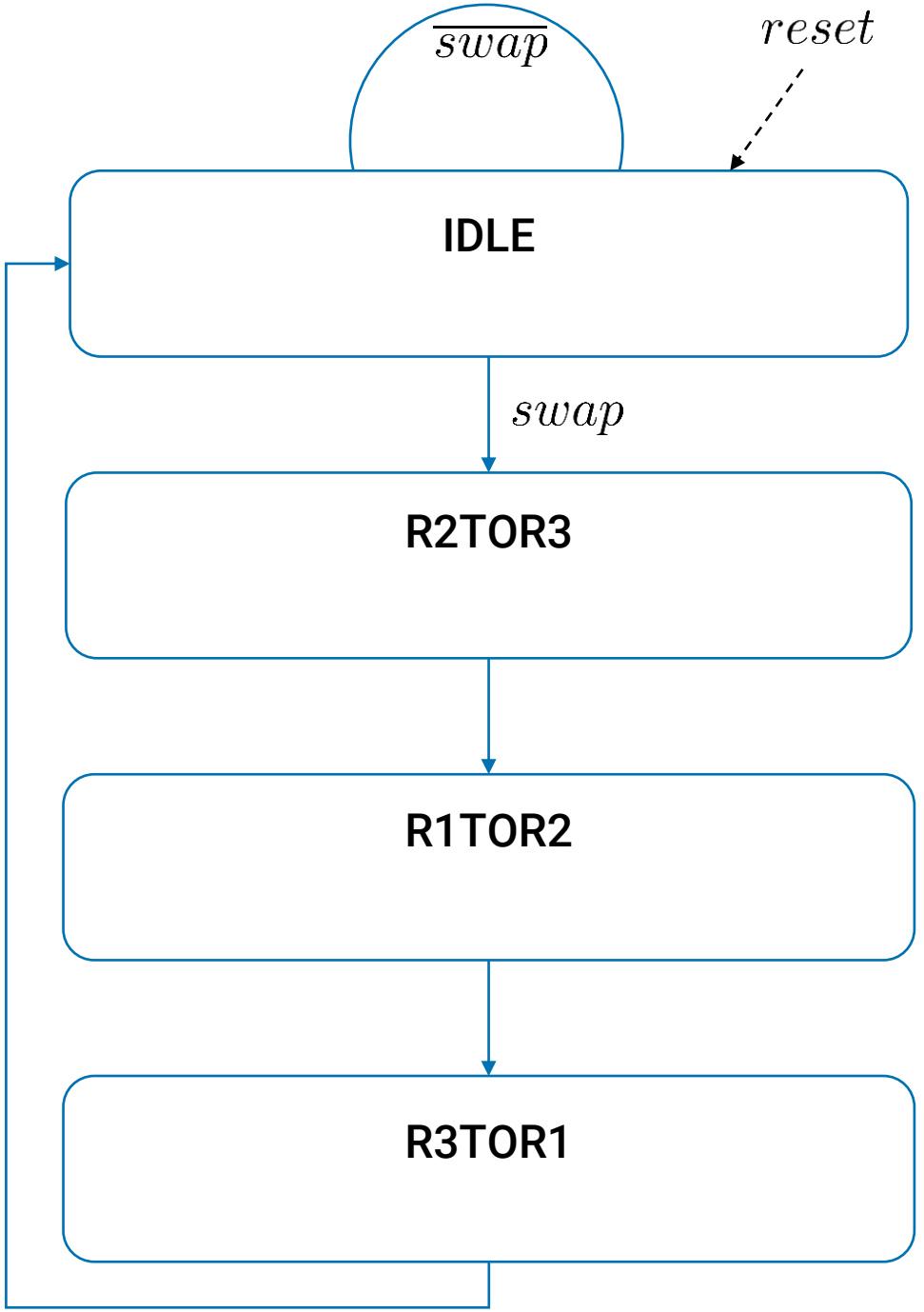
- **R1out, R2out, R3out**
  - **Write to the bus**
  - If active, places the value from a register to the bus
  - Output
- **R1in, R2in, R3in**
  - **Write to the register**
  - If active, places the value from the bus to a register
  - Output



Swapping Two Registers

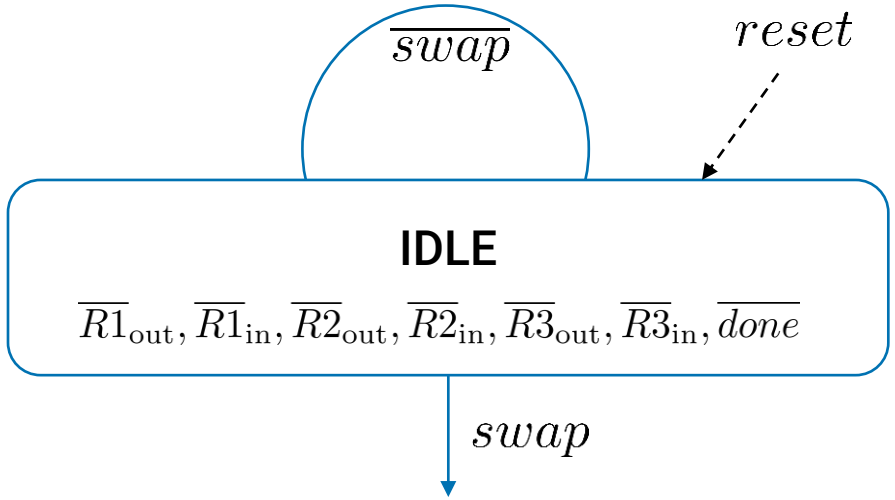
FSM

- Algorithm for swapping:
  - Swap starts → Copy data from R2 to R3
  - Copy data from R1 to R2
  - Copy contents of R3 to R1 → Swap ends
- States of the FSM
  - **IDLE**: No swapping
  - **R2TOR3**: First copy
  - **R1TOR2**: Second copy
  - **R3TOR1**: Third copy
- Synchronous power-on reset

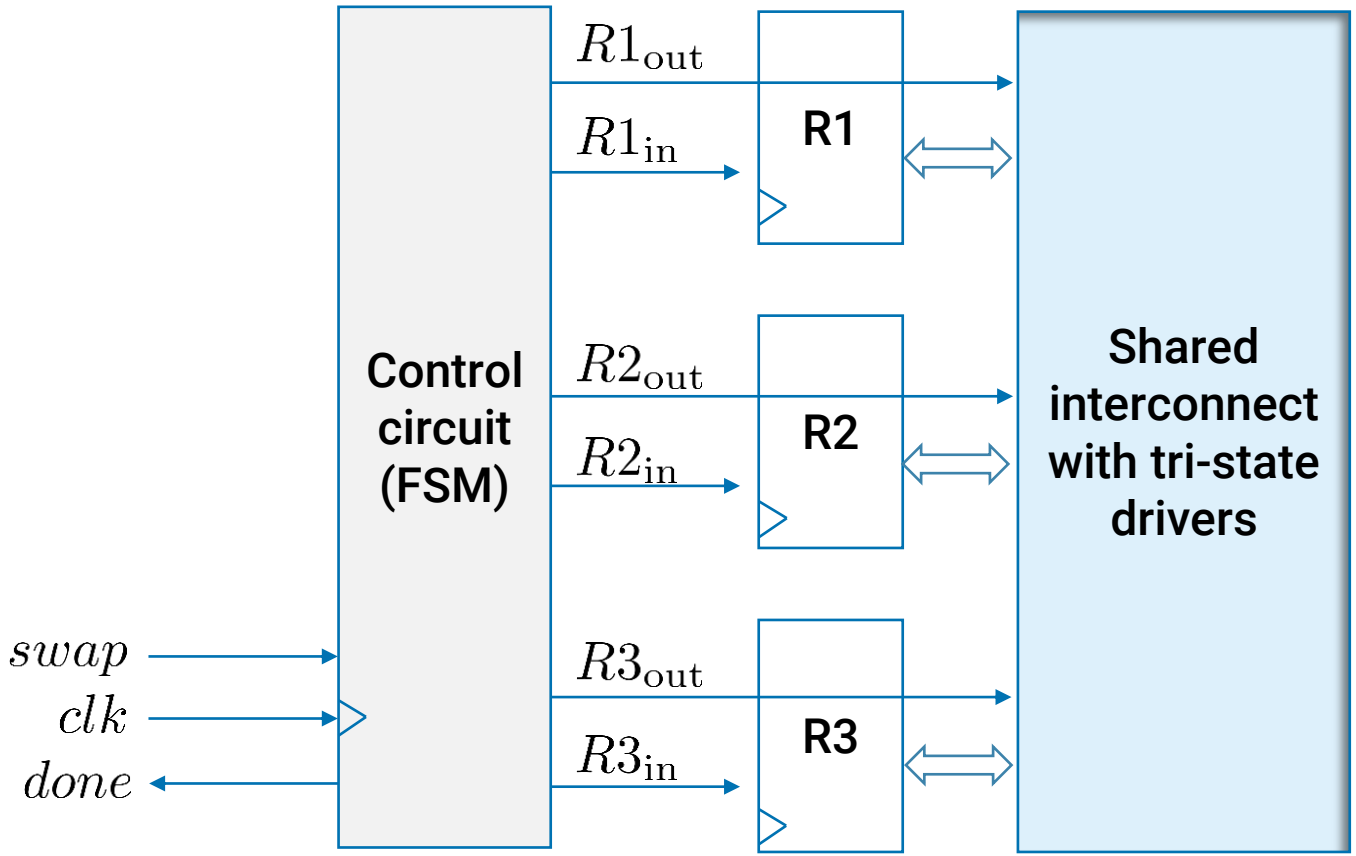


Swapping Two Registers

IDLE State

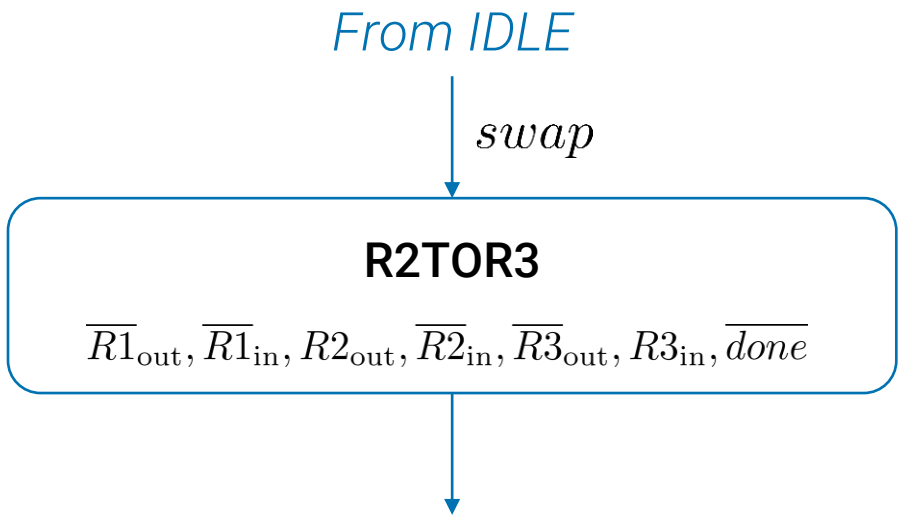


No writing to the bus
No writing to the registers
Active "swap" starts the data movement
Default state after reset

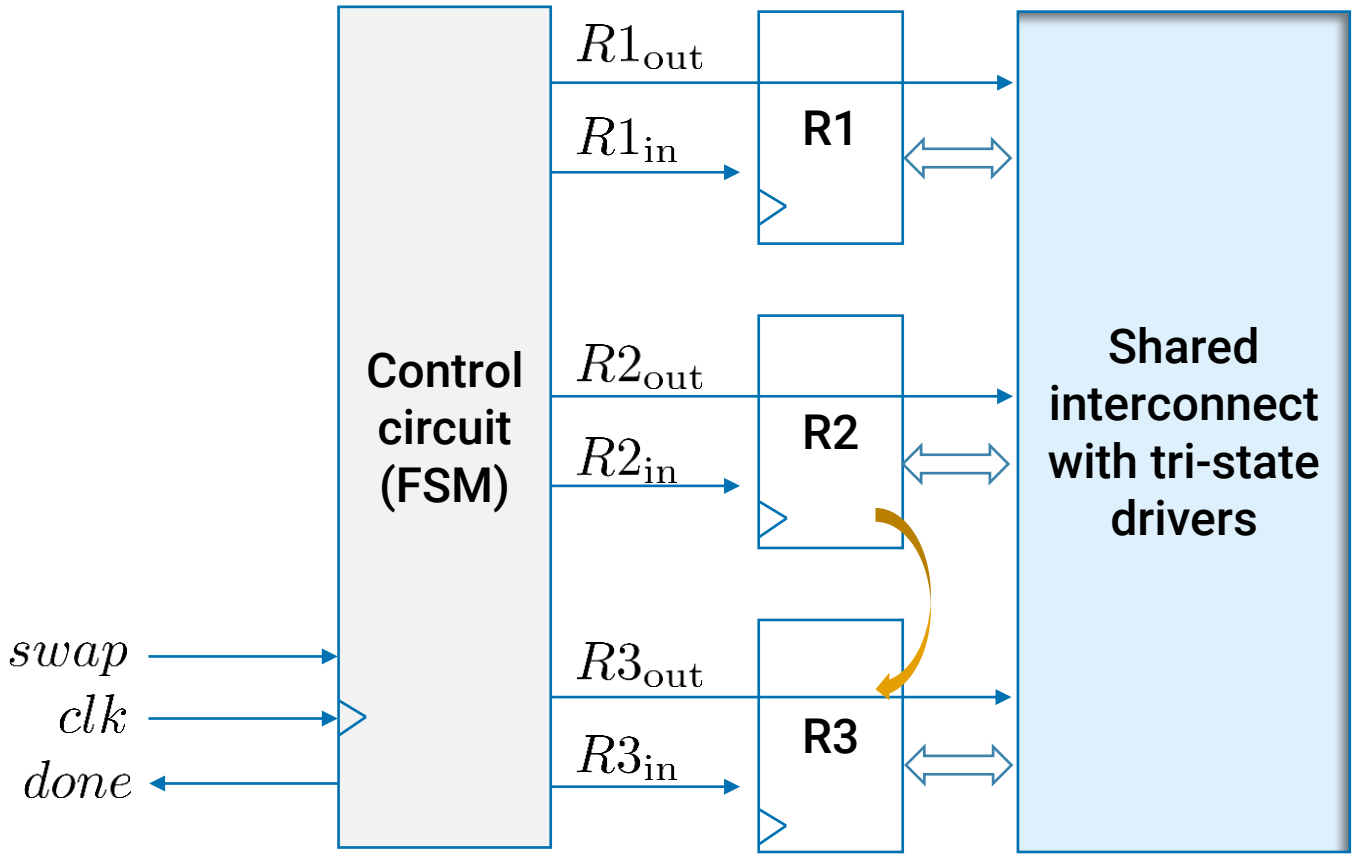


Swapping Two Registers

R2TOR3 State



Writing from register R2 to the bus
Writing from the bus to register R3



Swapping Two Registers

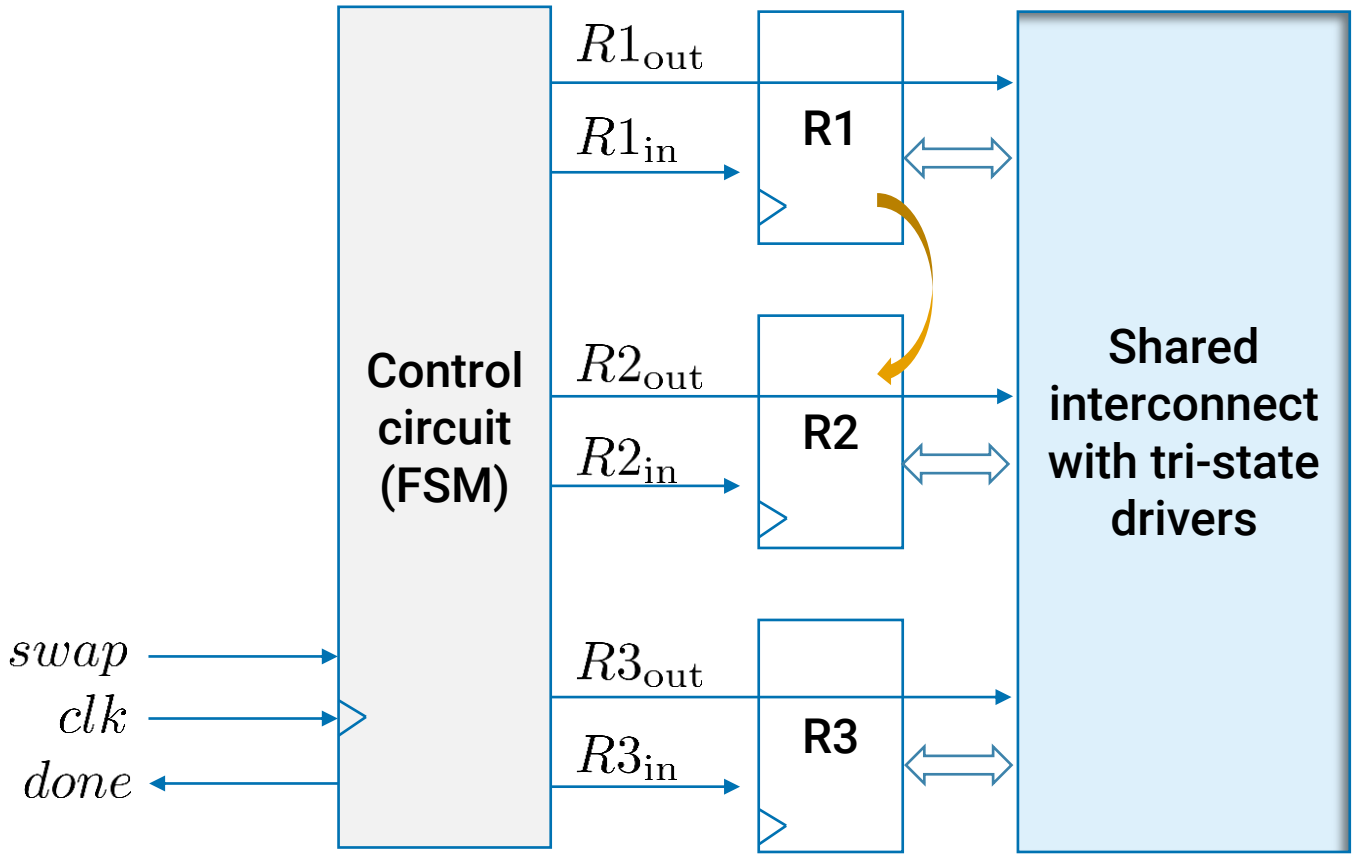
R1TOR2 State

From R2TOR3

R1TOR2

$R1_{\text{out}}, \overline{R1}_{\text{in}}, \overline{R2}_{\text{out}}, R2_{\text{in}}, \overline{R3}_{\text{out}}, \overline{R3}_{\text{in}}, \overline{done}$

Writing from register R1 to the bus
Writing from the bus to register R2



Swapping Two Registers

R3TOR1 State

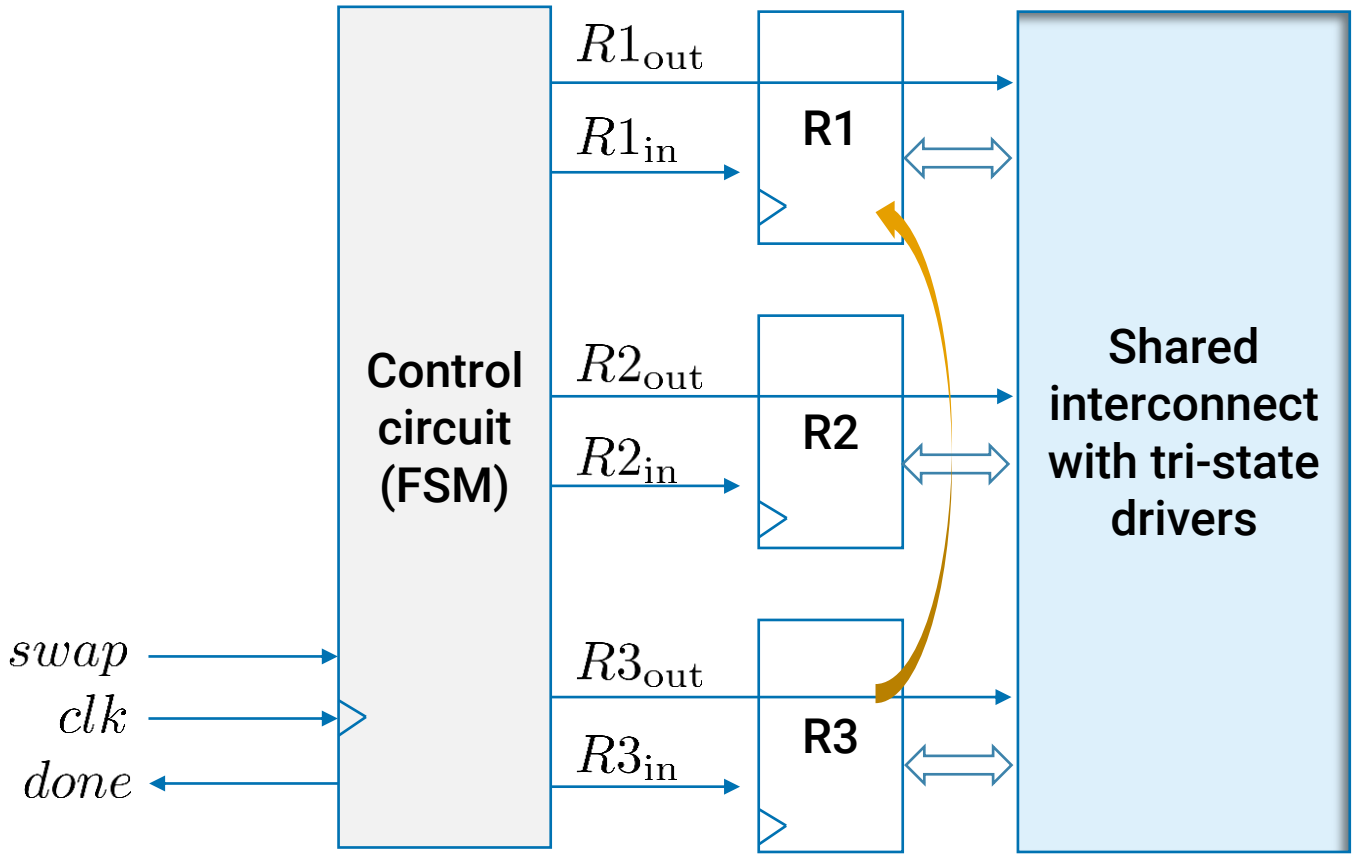
From R1TOR2

R3TOR1

$\overline{R1}_{\text{out}}, R1_{\text{in}}, \overline{R2}_{\text{out}}, \overline{R2}_{\text{in}}, R3_{\text{out}}, \overline{R3}_{\text{in}}, done$

To IDLE

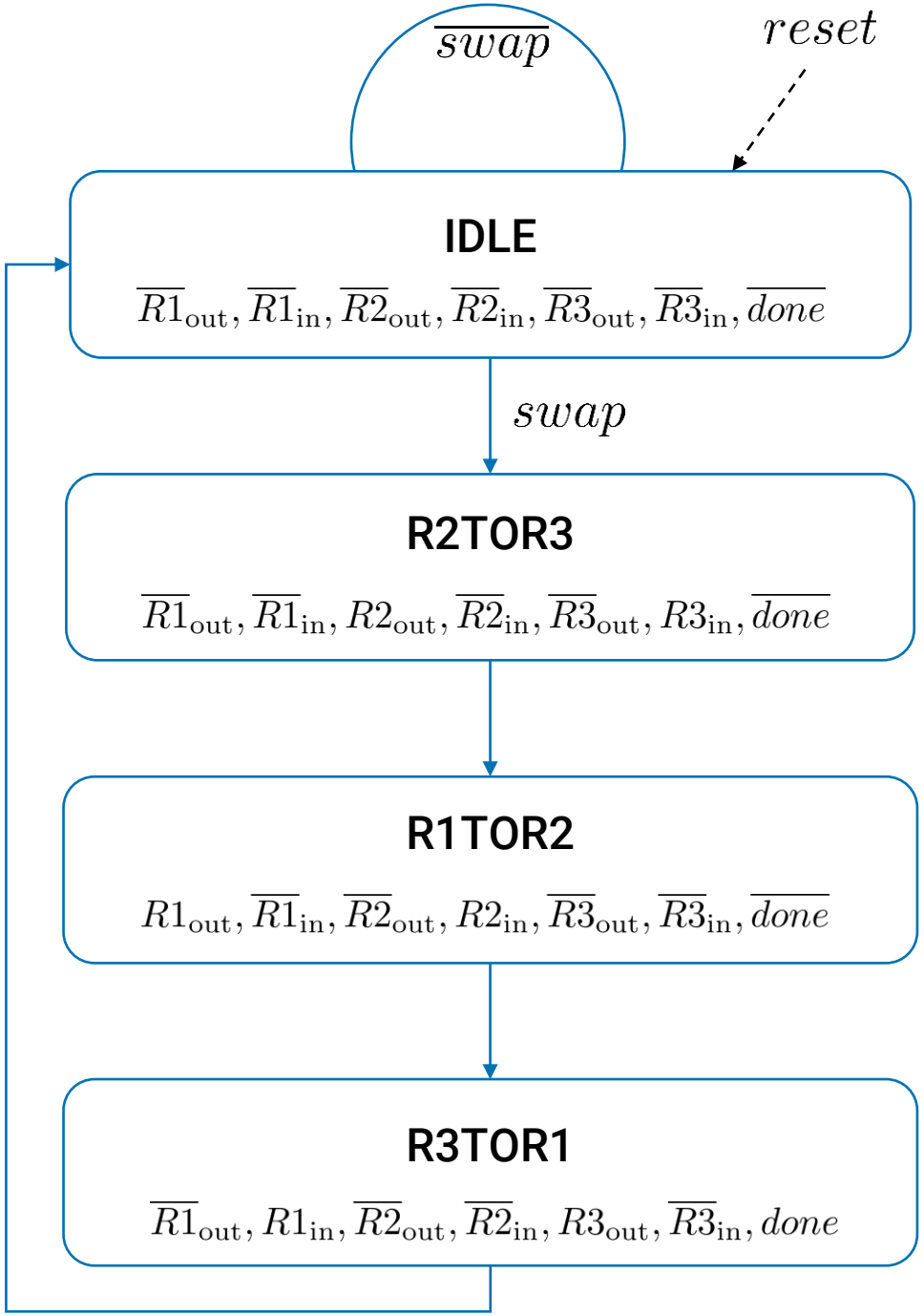
Writing from register R3 to the bus
Writing from the bus to register R1
"Done" becomes active



Swapping Two Registers

Summary

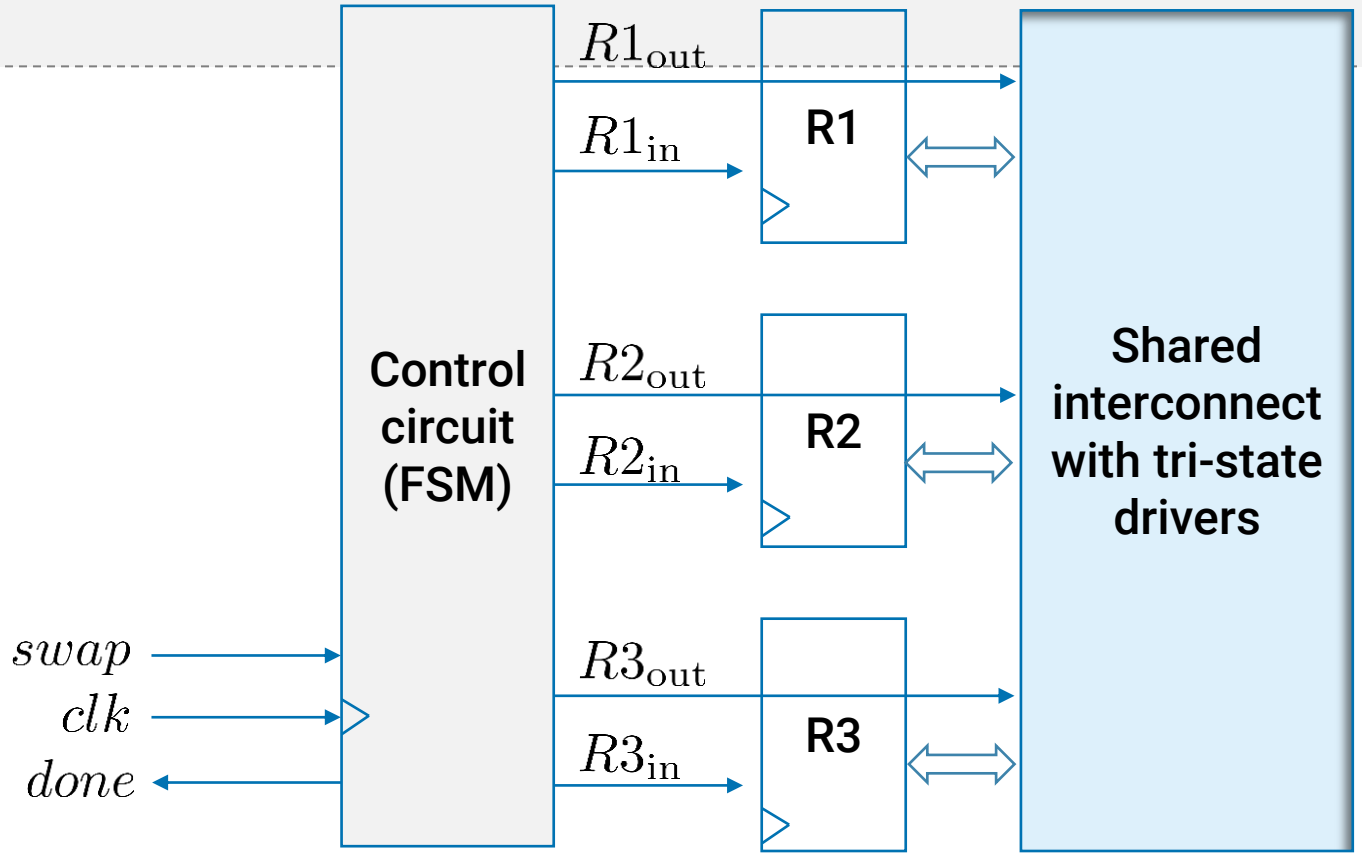
- Algorithm for swapping:
  - Swap starts → Copy data from R2 to R3
  - Copy data from R1 to R2
  - Copy contents of R3 to R1 → Swap ends
- States of the FSM
  - **IDLE**: No swapping
  - **R2TOR3**: First copy
  - **R1TOR2**: Second copy
  - **R3TOR1**: Third copy
- Synchronous power-on reset



Swapping Two Registers

FSM in Verilog

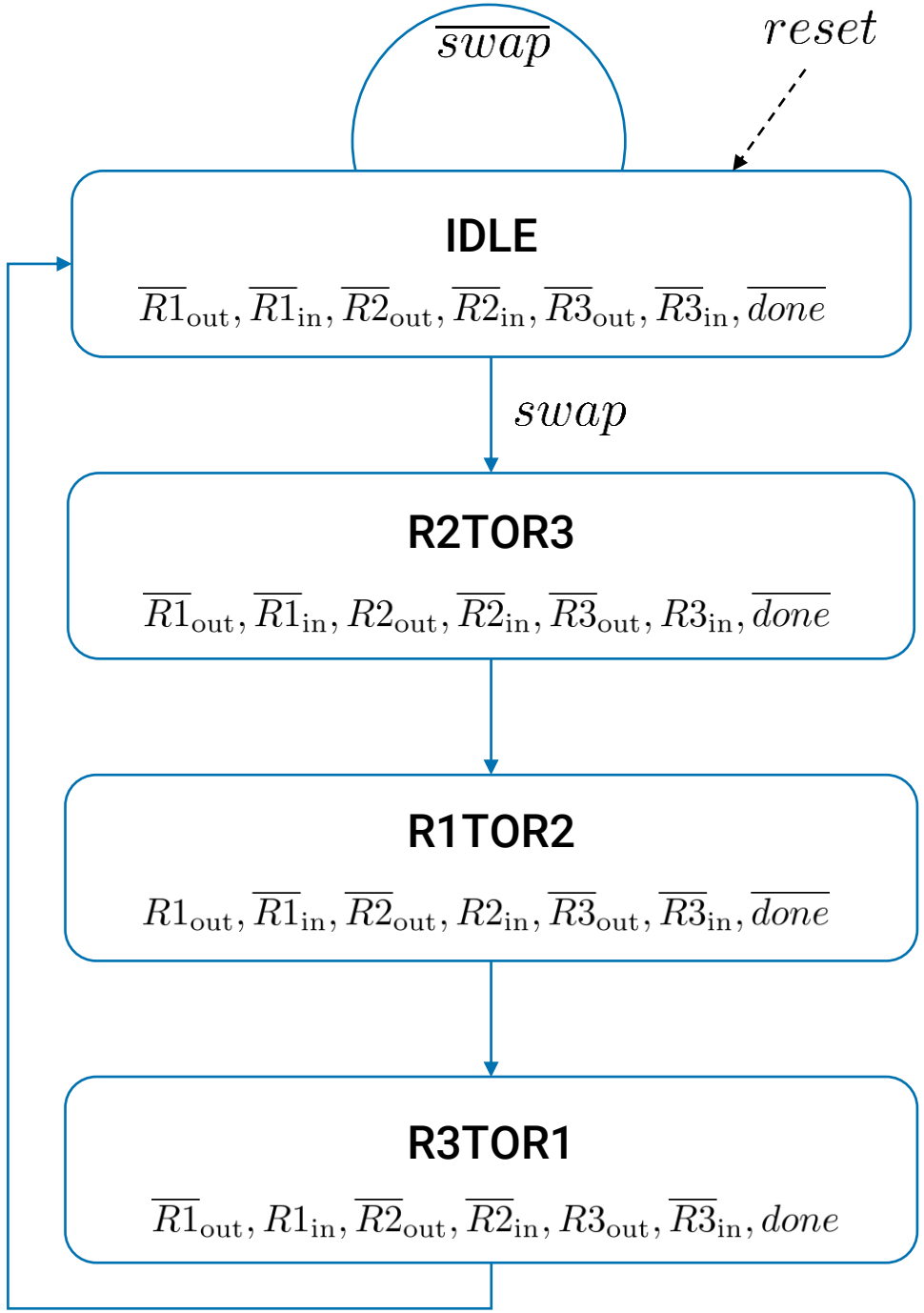
```
module control (clk, reset, swap, R1in, R1out, R2in, R2out, R3in, R3out, done);
  input       clk, reset, swap;
  output reg R1in, R1out, R2in, R2out, R3in, R3out, done;
  parameter  IDLE = 2'b00, R2TOR3 = 2'b01, R1TOR2 = 2'b10, R3TOR1 = 2'b11;
  reg [1:0]  S_next, S;
```



Swapping Two Registers

FSM in Verilog

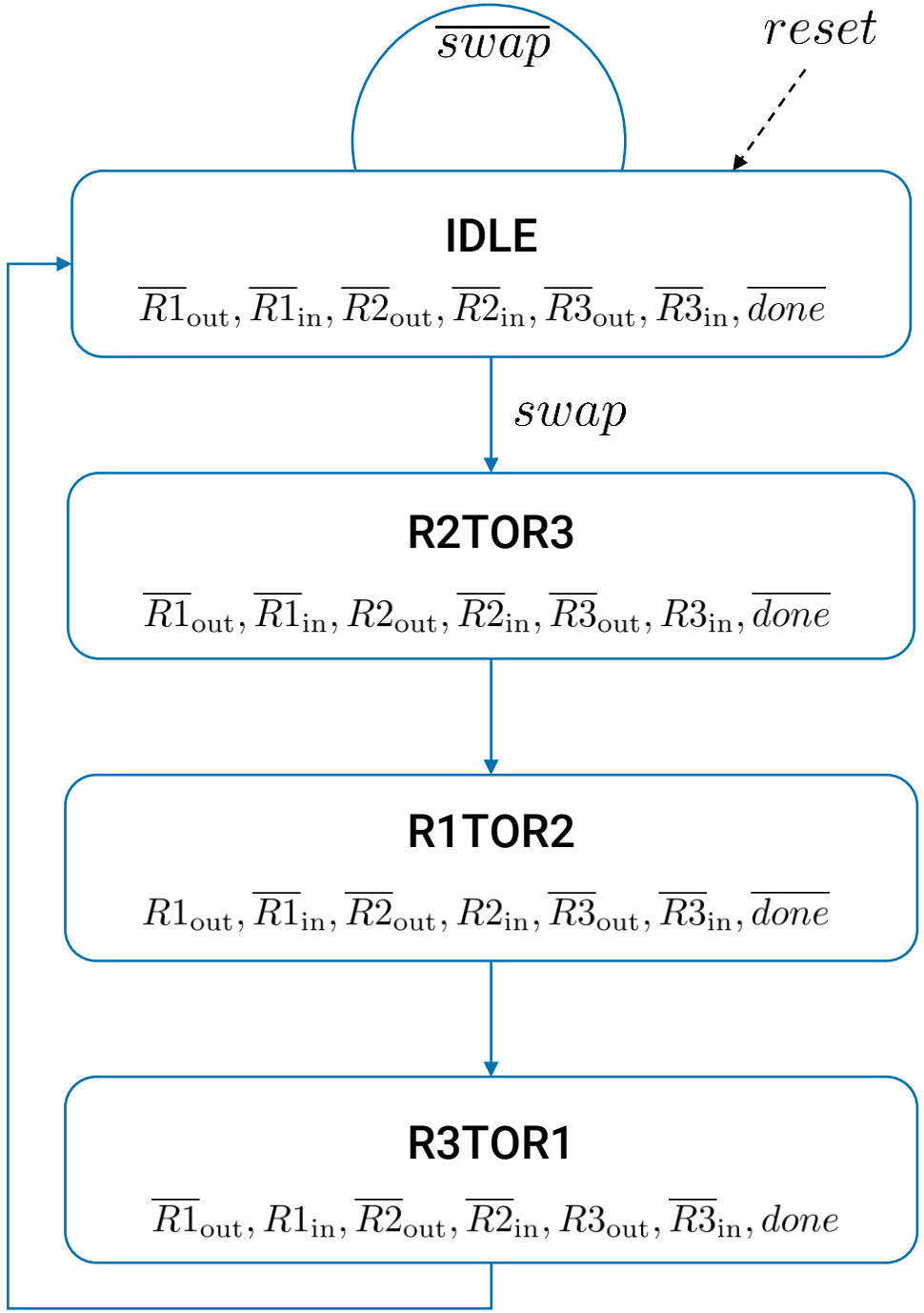
```
  // Next-state logic
  always @ (*) begin
    S_next = IDLE;  // default, idle state
    case (S)
      IDLE: if (swap) S_next = R2TOR3;
            else      S_next = IDLE;
      R2TOR3:         S_next = R1TOR2;
      R1TOR2:         S_next = R3TOR1;
      R3TOR1:         S_next = IDLE;
      default:        S_next = IDLE; // default
endcase
  end
```



Swapping Two Registers

FSM in Verilog

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

  // Output logic
  always @ (*) begin
    R1in  = (S == R3TOR1);
    R1out = (S == R1TOR2);
    R2in  = (S == R1TOR2);
    R2out = (S == R2TOR3);
    R3in  = (S == R2TOR3);
    R3out = (S == R3TOR1);
    done  = (S == R3TOR1);
  end
endmodule
```



Swapping Two Registers

Verilog, Contd.

- Missing pieces:
  - Tri-state drivers for the bus

```
module bustri (w, en, f);
  parameter           n = 8; // default size
  input  [n-1:0]      w;
  input               en;
  output [n-1:0]      f;

  assign f = en ? w : 'bz;
  // either assign a new value or high-impedance
endmodule
```

Swapping Two Registers

Verilog, Contd.

- Missing pieces:
  - Register module for registers R1, R2, R3

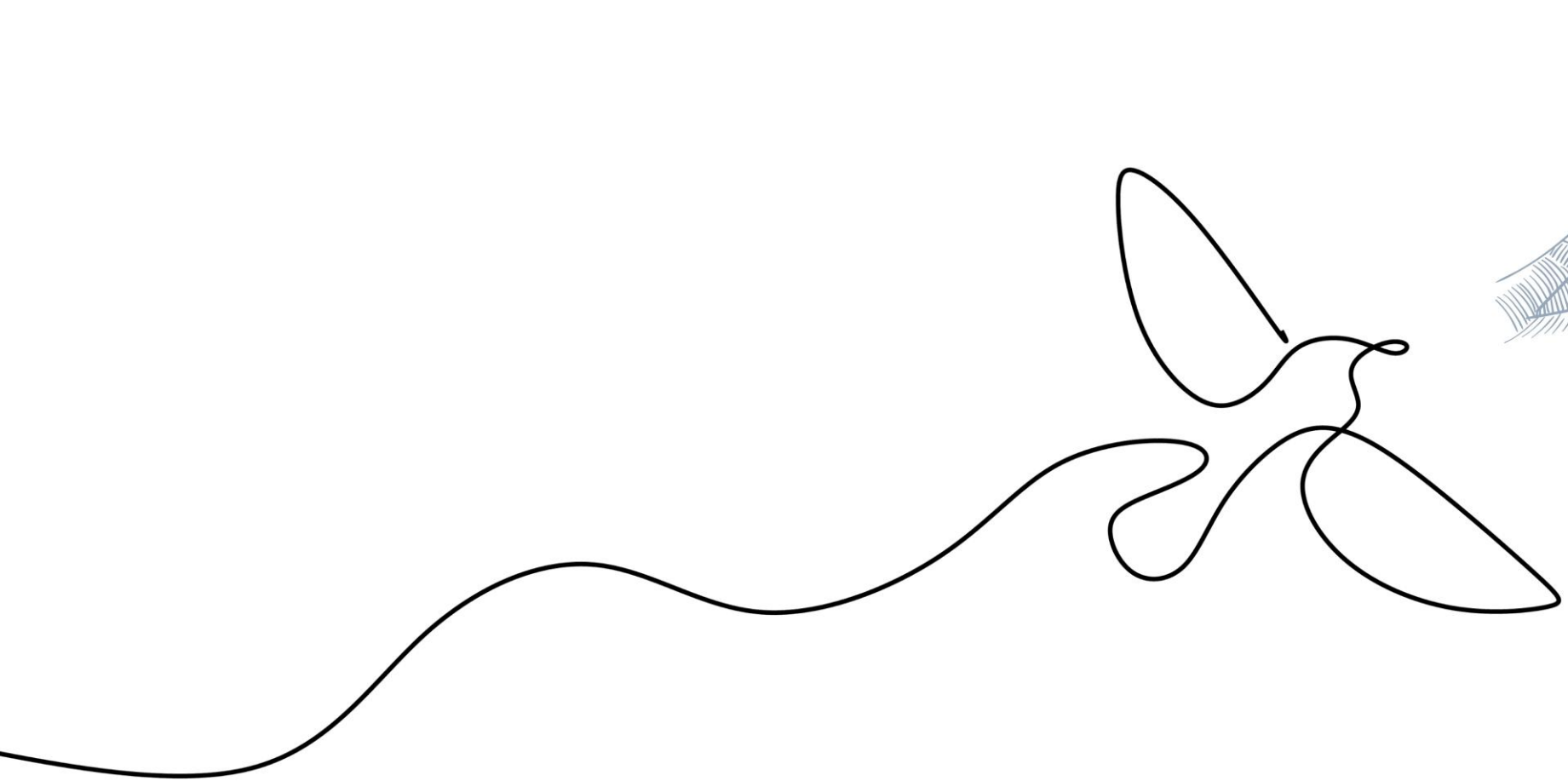
```
module regn (D, clk, reset, en, Q);
  parameter           n = 8;  // default size
  input [n-1:0]       D;
  input               clk, reset, en;
  output reg [n-1:0] Q;

  always @ (posedge clk) begin
    if (reset)   Q <= 0;
    else if (en) Q <= D;  // write enable
  end
endmodule
```

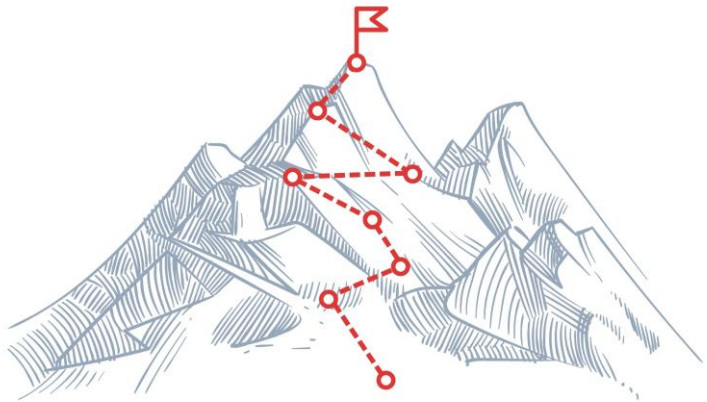
Swapping Two Registers

Putting It All Together

```
module regswap (clk, reset, swap);
  parameter width = 8; // width is "n", the number of wires on the bus
  input     clk, reset, swap;
  wire      wR1in, wR1out, wR2in, wR2out, wR3in, wR3out, wdone;
  wire [width-1:0] wR1, wR2, wR3, wBus;
  // Instantiate controller module
  control controller_module ( .clk (clk), .reset (reset), .swap (swap),
                              .R1in (wR1in), .R1out (wR1out), .R2in (wR2in), .R2out (wR2out),
                              .R3in (wR3in), .R3out (wR3out), .done (wdone));

  // Instantiate registers
  regn #(.n (width)) R1 (.D (wBus), .clk (clk), .reset (reset), .en (wR1in), .Q (wR1));
  regn #(.n (width)) R2 (.D (wBus), .clk (clk), .reset (reset), .en (wR2in), .Q (wR2));
  regn #(.n (width)) R3 (.D (wBus), .clk (clk), .reset (reset), .en (wR3in), .Q (wR3));

  // Bus with tri-state drivers
  bustri #(.n (width)) bustri1 (.w (wR1), .en (wR1out), .f (wBus));
  bustri #(.n (width)) bustri2 (.w (wR2), .en (wR2out), .f (wBus));
  bustri #(.n (width)) bustri3 (.w (wR3), .en (wR3out), .f (wBus));
endmodule
```

- *Registers drive the inputs of tri-state drivers*
- *Tri-state buffers drive the bus*
- *Bus drives the register inputs*



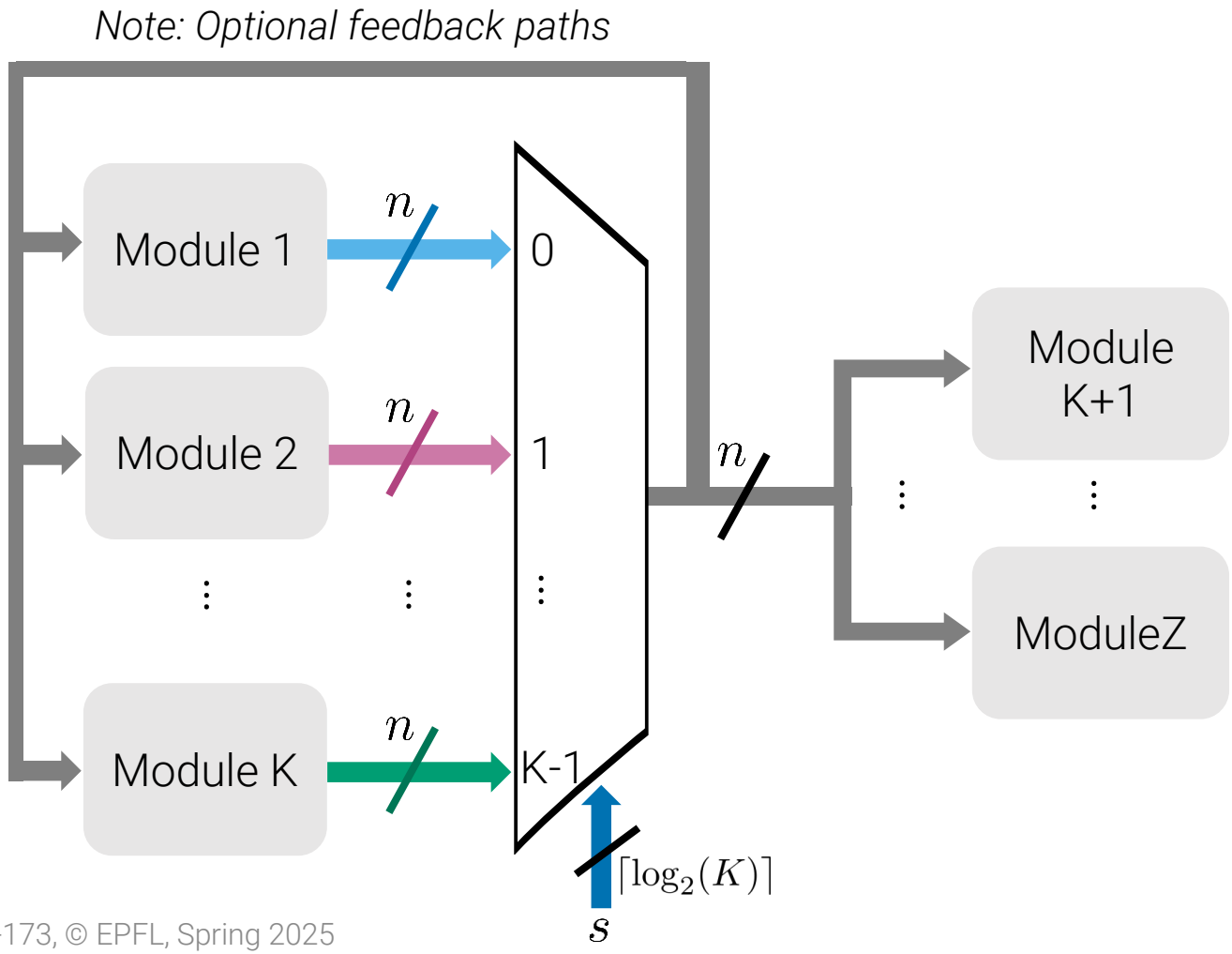


Bus with a MUX

Example: Swapping Two Registers

Recall: Bus With MUXes

Previously on FDS



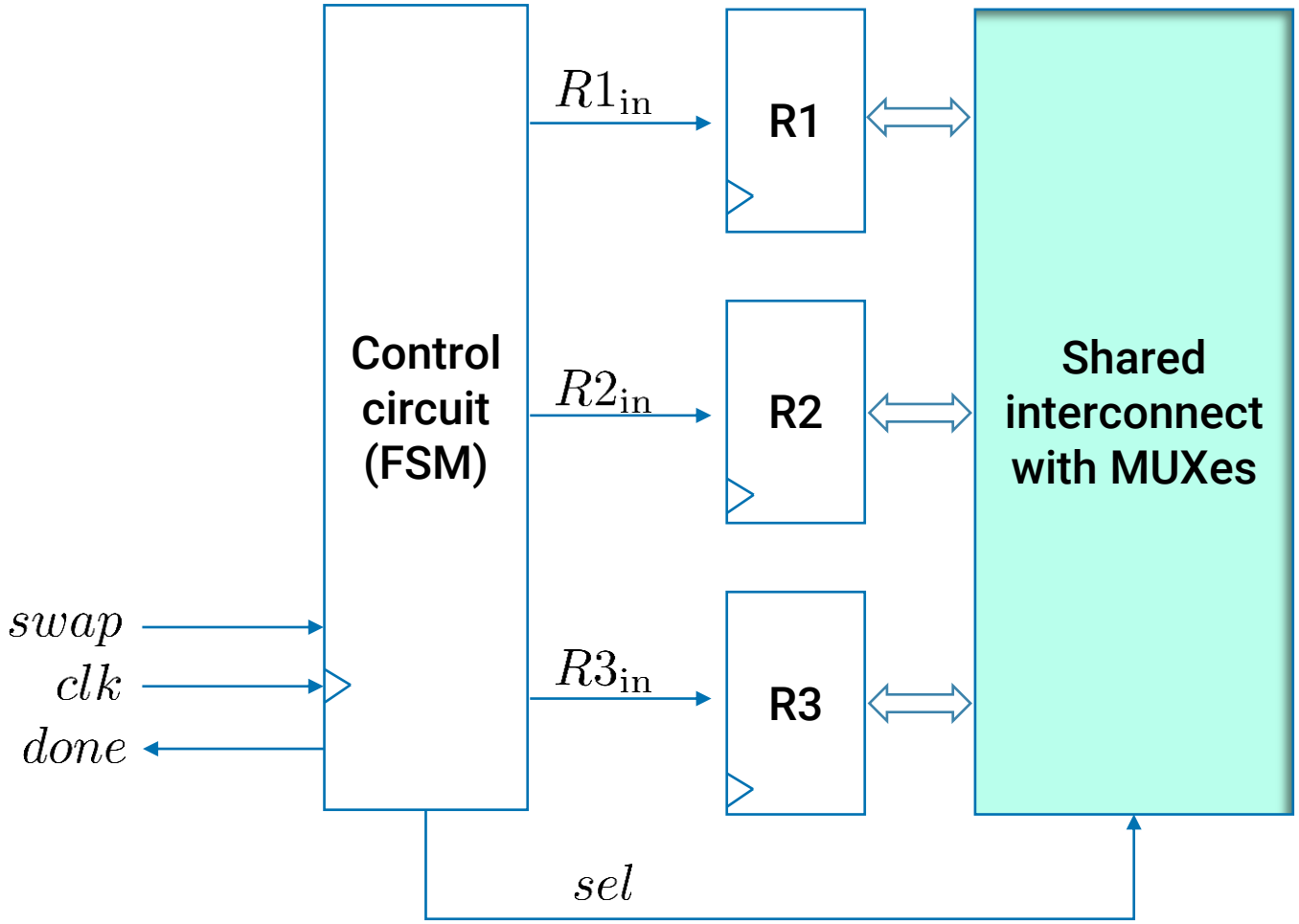
- Bus implemented with MUXes is more common
- The MUX takes K (K ≥ 2) n-bit inputs and an $\lceil \log_2(K) \rceil$-bit select signal s to select which of the inputs to pass to the output
- There is another module, typically a controller FSM, responsible for the activation of the select signals (not shown)

Swapping Two Registers

Bus with MUXes

Consider a system with three registers: R1, R2, and R3

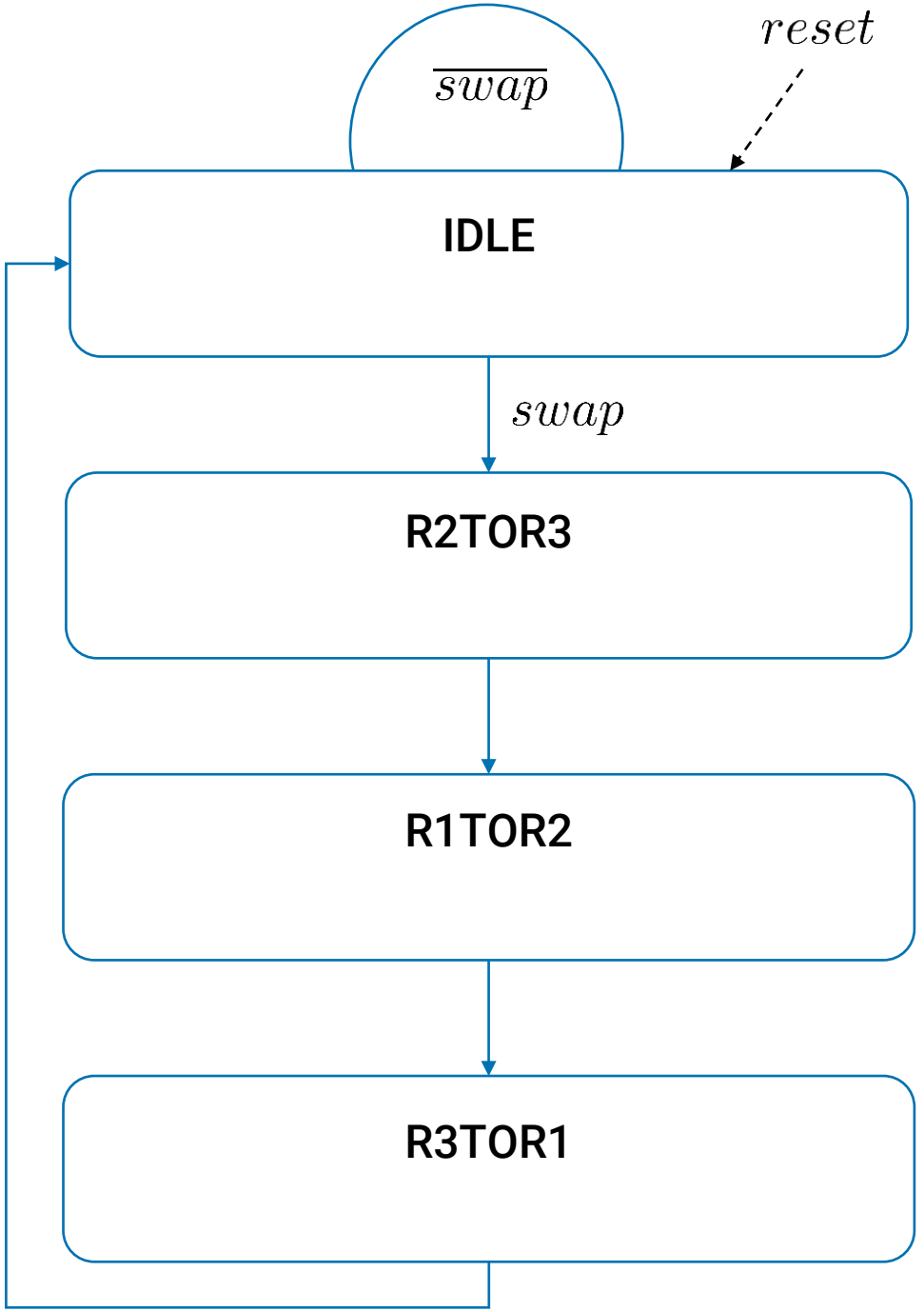
- Design a controller FSM that swaps the contents of registers R1 and R2, using R3 for temporary storage
- Write a Verilog model of the system



Swapping Two Registers

FSM

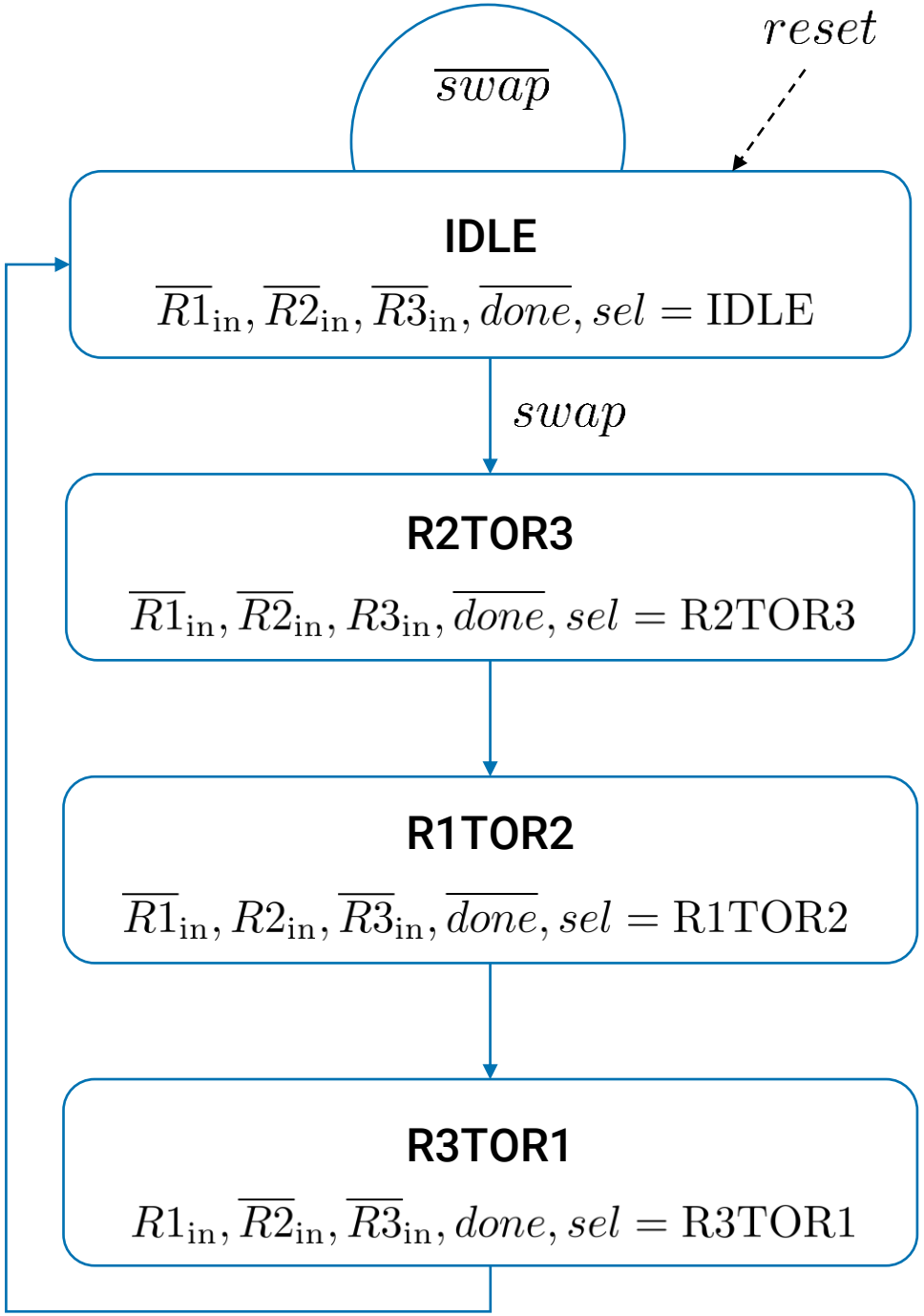
- Algorithm for swapping:
  - Swap starts → Copy data from R2 to R3
  - Copy data from R1 to R2
  - Copy contents of R3 to R1 → Swap ends
- States of the FSM
  - IDLE: No swapping
  - R2TOR3: First copy
  - R1TOR2: Second copy
  - R3TOR1: Third copy
- Synchronous power-on reset



Swapping Two Registers

FSM

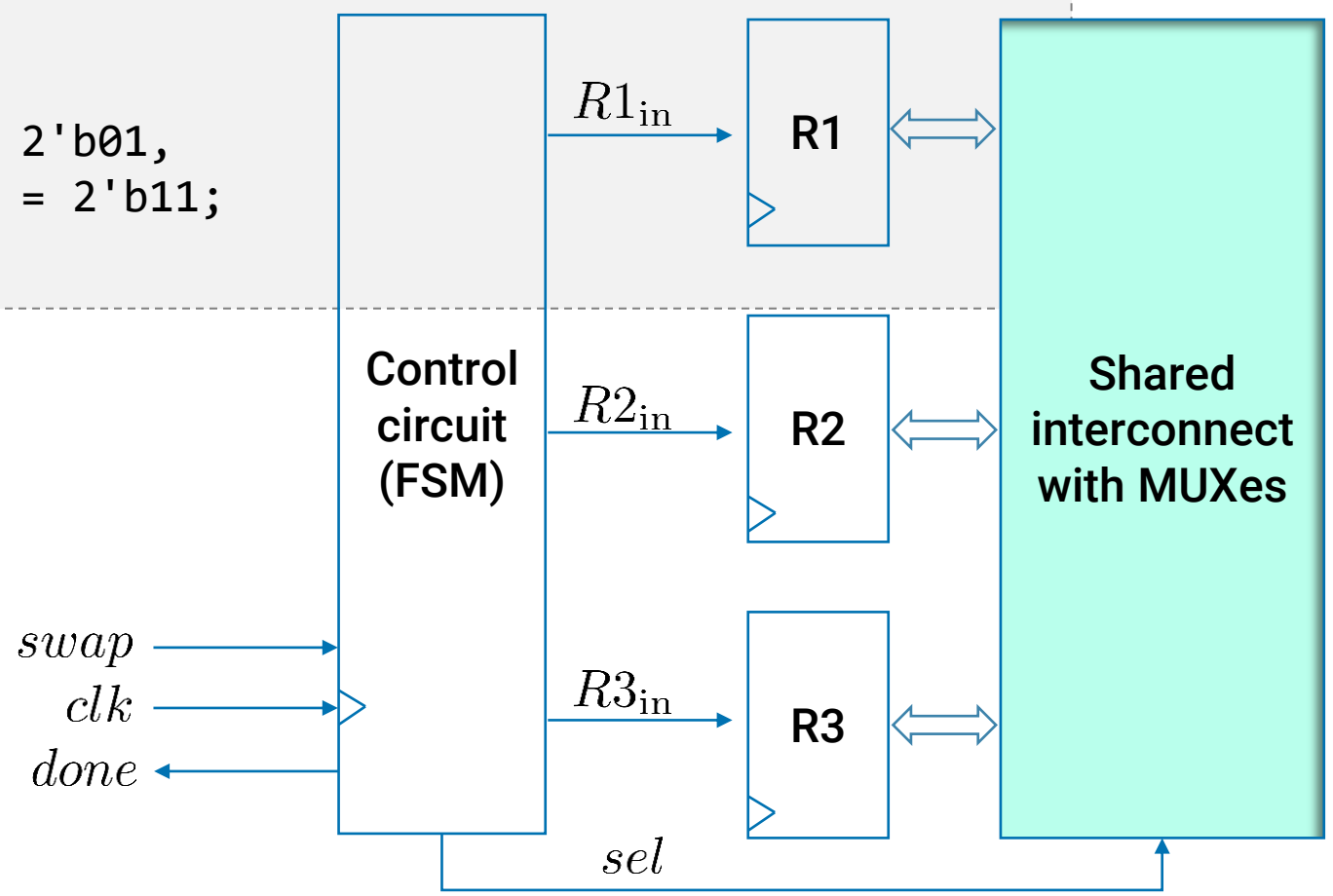
- No longer needing R1out/R2out/R3out
  - MUX select signal **sel** fulfills the role
- MUX select signal is simply the state of the FSM
- In every state (every select signal value), something is sent to the bus



Swapping Two Registers

FSM in Verilog

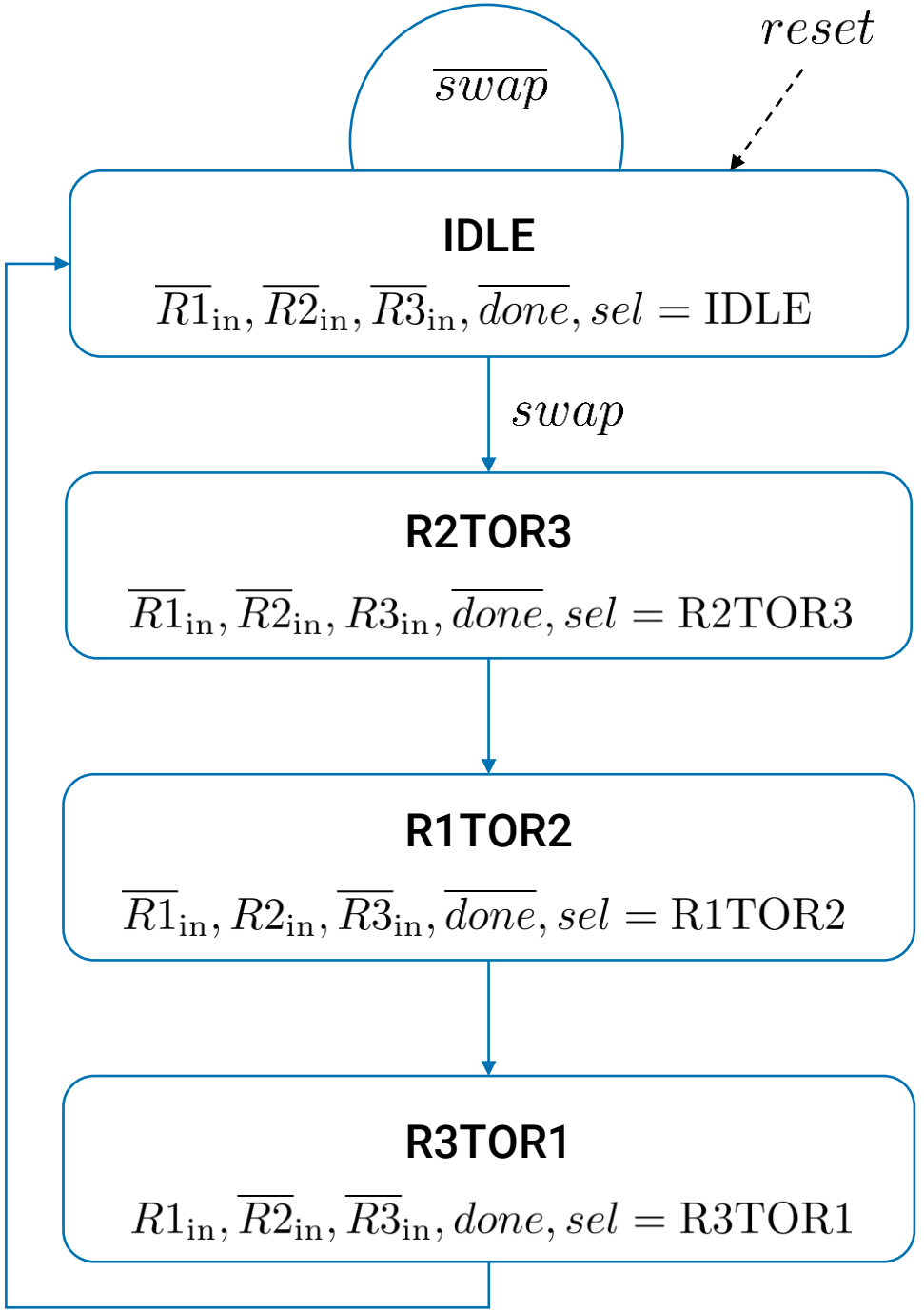
```
module controlbusmux (clk, reset, swap, R1in, R2in, R3in, done, sel);
  input       clk, reset, swap;
  output reg R1in, R2in, R3in, done;
  output reg [1:0] sel;

  parameter IDLE = 2'b00, R2TOR3 = 2'b01,
            R1TOR2 = 2'b10, R3TOR1 = 2'b11;
  reg [1:0] S_next, S;
```



Swapping Two Registers

FSM in Verilog

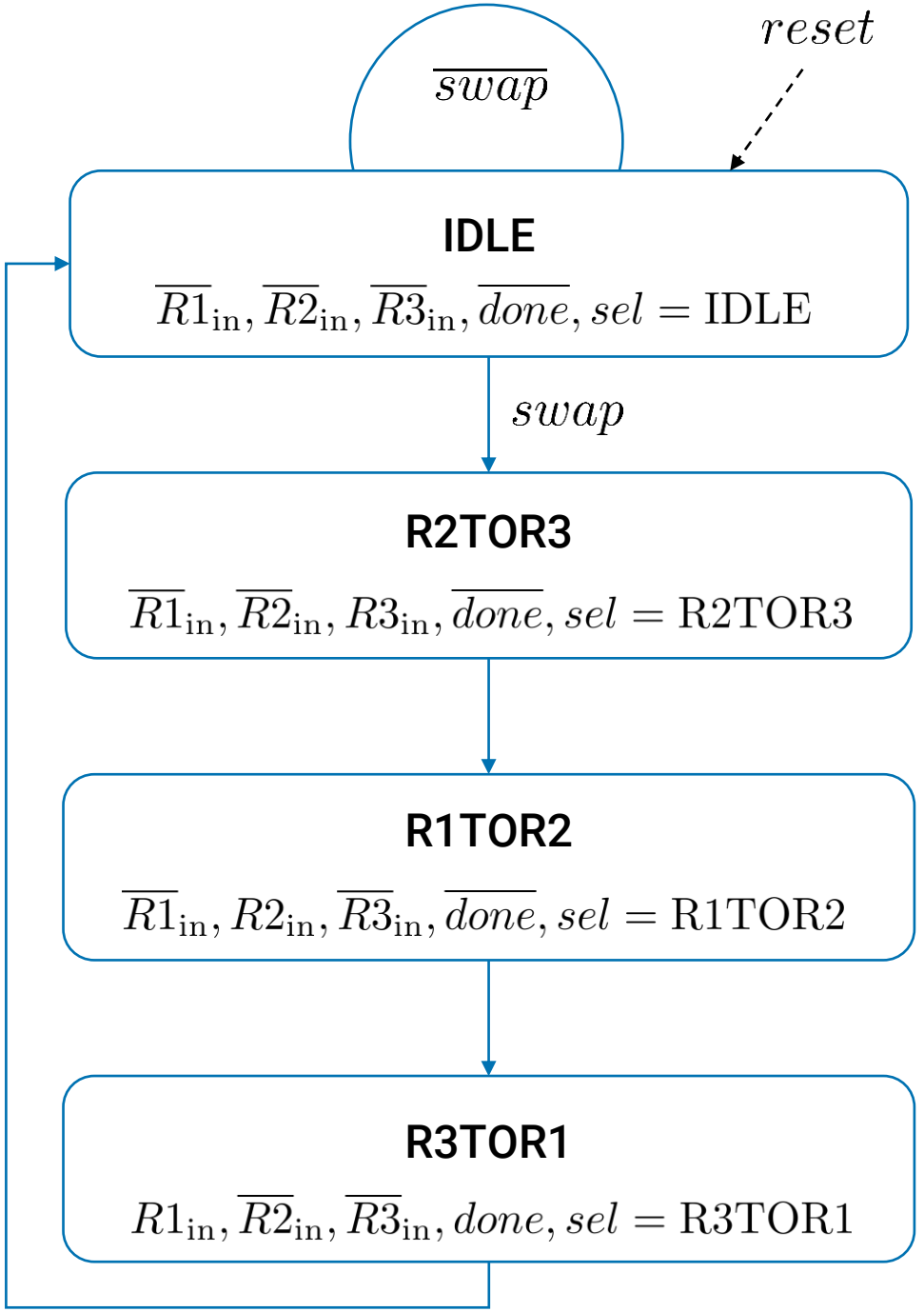
```
// Next-state logic
always @ (*) begin
  S_next = IDLE;
  case (S)
    IDLE: if (swap) S_next = R2TOR3;
          else      S_next = IDLE;
    R2TOR3:         S_next = R1TOR2;
    R1TOR2:         S_next = R3TOR1;
    R3TOR1:         S_next = IDLE;
    default:        S_next = IDLE;
  endcase
end
```



Swapping Two Registers

FSM in Verilog

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

  // Output logic
  always @ (*) begin
    R1in  = (S == R3TOR1);
    R2in  = (S == R1TOR2);
    R3in  = (S == R2TOR3);
    done  = (S == R3TOR1);
    sel   =  S;
  end
endmodule
```



Swapping Two Registers

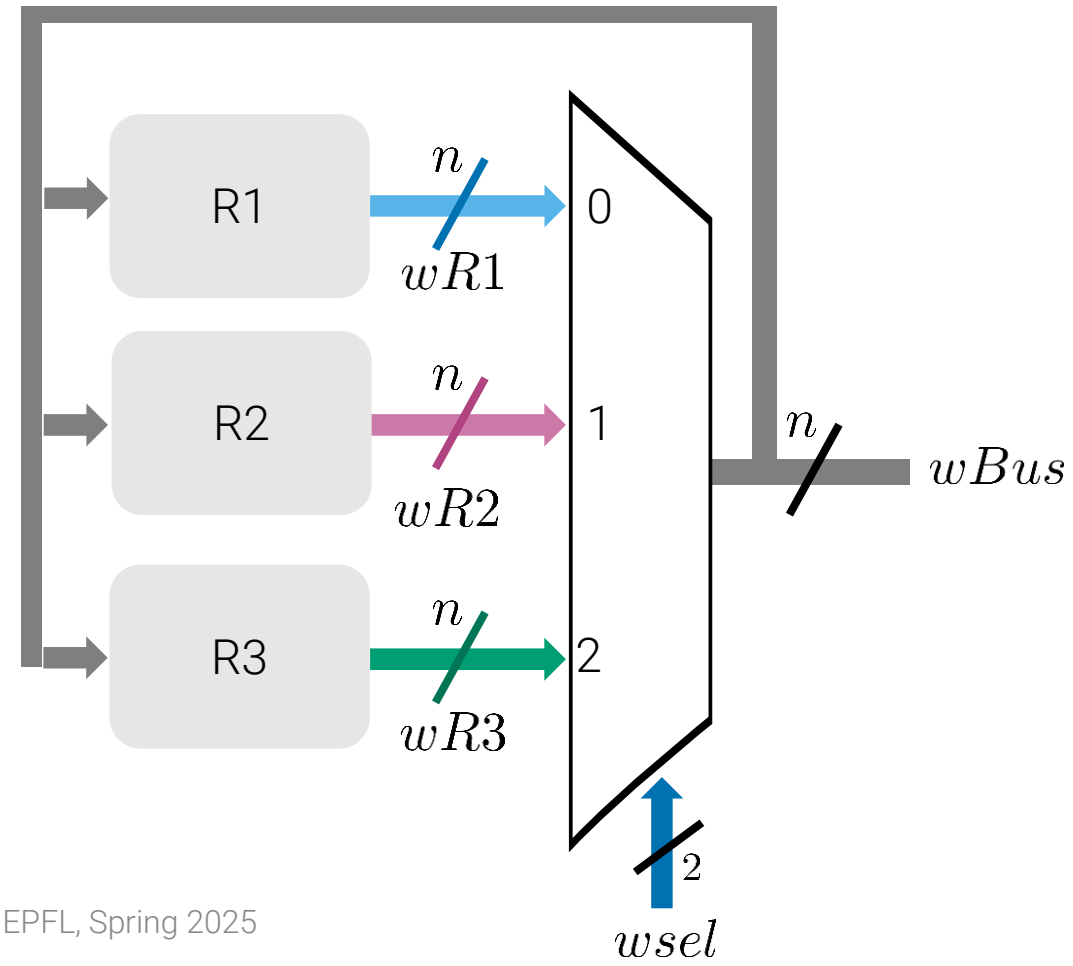
Putting it All Together

```
module regswapbusmux (clk, reset, swap);
  parameter width = 8; // width is "n", the number of wires on the bus
  parameter IDLE = 2'b00, R2TOR3 = 2'b01;
  parameter R1TOR2 = 2'b10, R3TOR1 = 2'b11;
  input     clk, reset, swap;

  wire wR1in, wR2in, wR3in, wdone;
  wire [width-1:0] wR1, wR2, wR3;
  wire [1:0]       wsel;  // for the bus with MUXes
  reg  [width-1:0] wBus;  // for the bus with MUXes
  // Instantiate controller module
  controlbusmux controller_module ( .clk (clk), .reset (reset), .swap (swap),
                                    .R1in (wR1in), .R2in (wR2in), .R3in (wR3in),
                                    .done (wdone), .sel (wsel));
  // Instantiate registers
  regn #(.n (width)) R1 (.D (wBus), .clk (clk), .reset (reset), .en (wR1in), .Q (wR1));
  regn #(.n (width)) R2 (.D (wBus), .clk (clk), .reset (reset), .en (wR2in), .Q (wR2));
  regn #(.n (width)) R3 (.D (wBus), .clk (clk), .reset (reset), .en (wR3in), .Q (wR3));
```


Complete the module by inserting the multiplexer block…

Swapping Two Registers

Verilog, Contd.

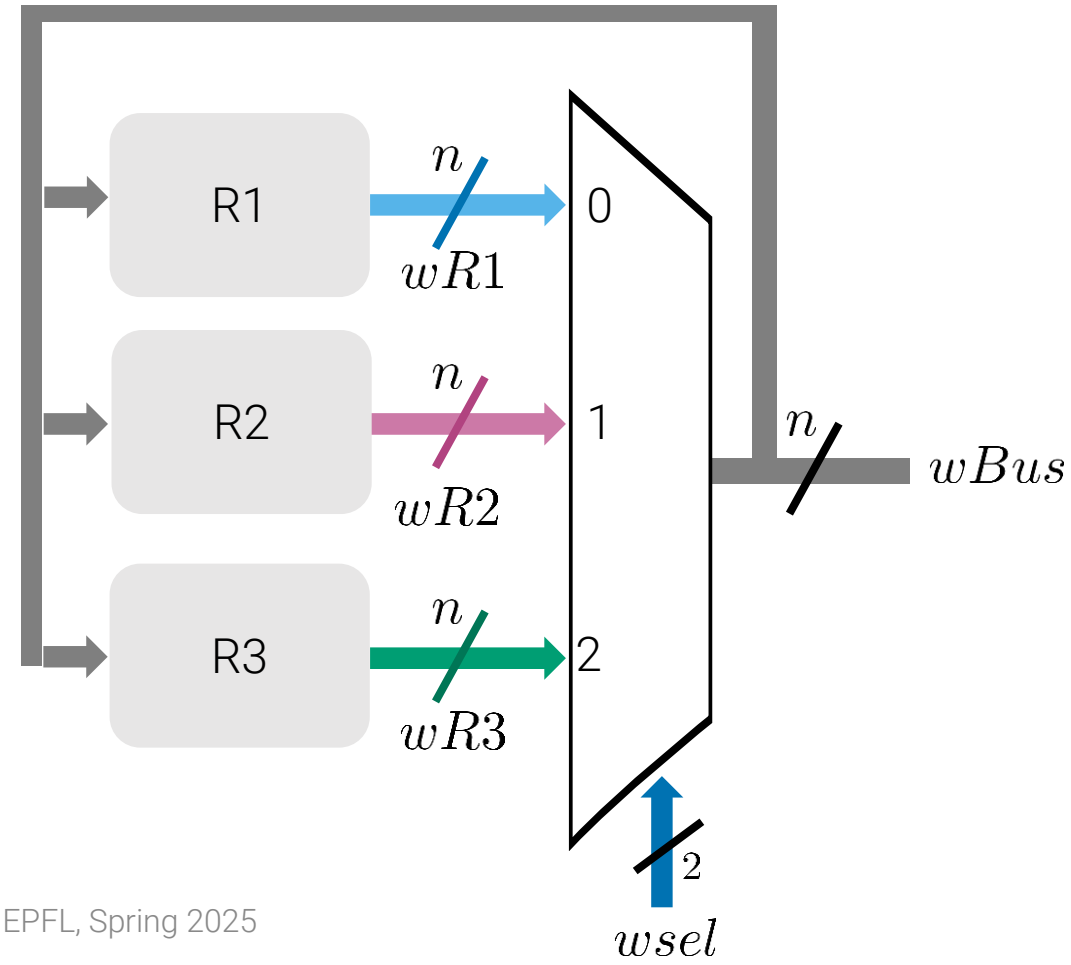


- Missing pieces
  - A bus with a MUX

wsel: select signal for the MUX
wBus: value on the bus (MUX output)
wR1: default value on the bus (output of register R1)

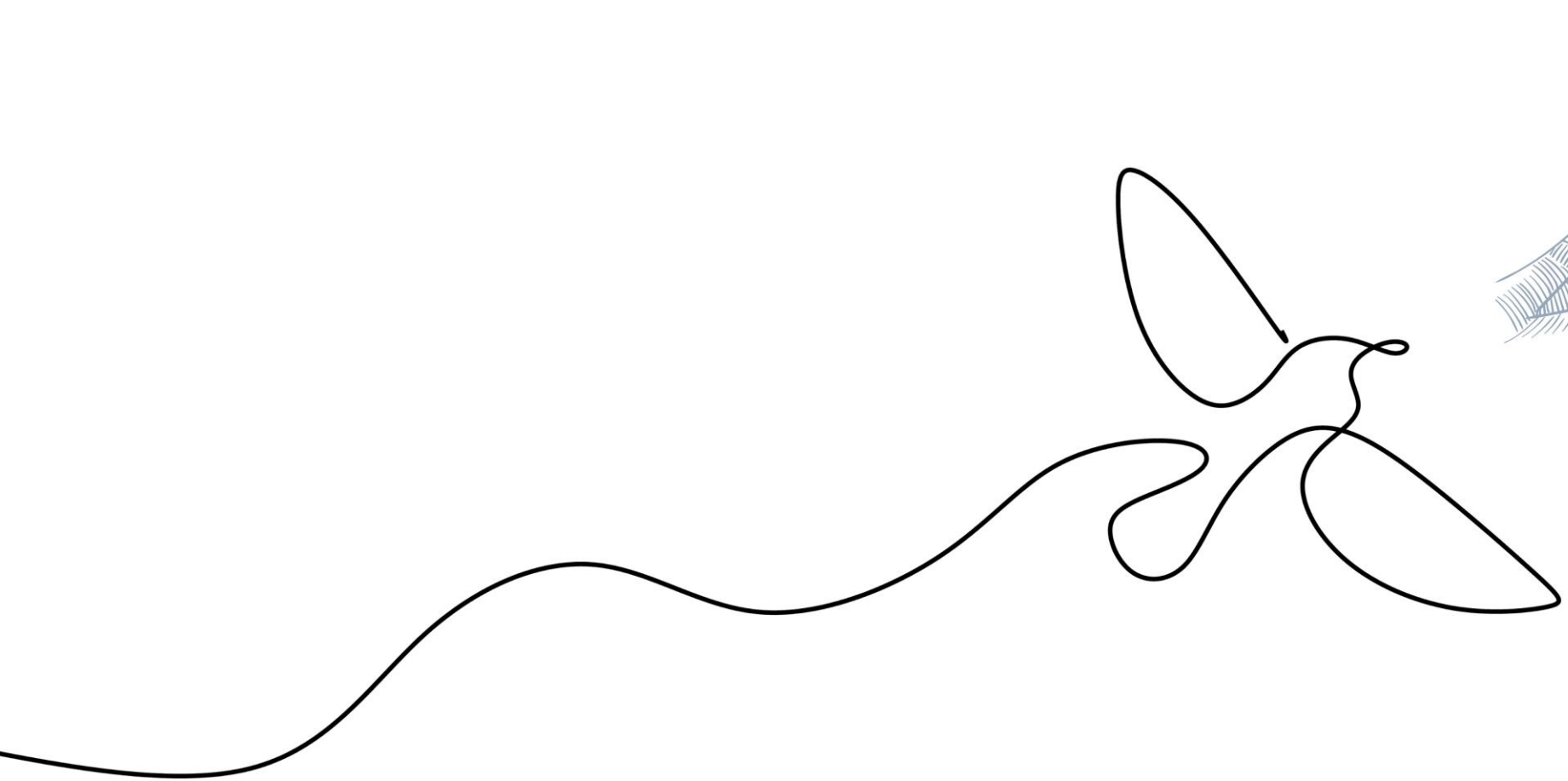
Swapping Two Registers

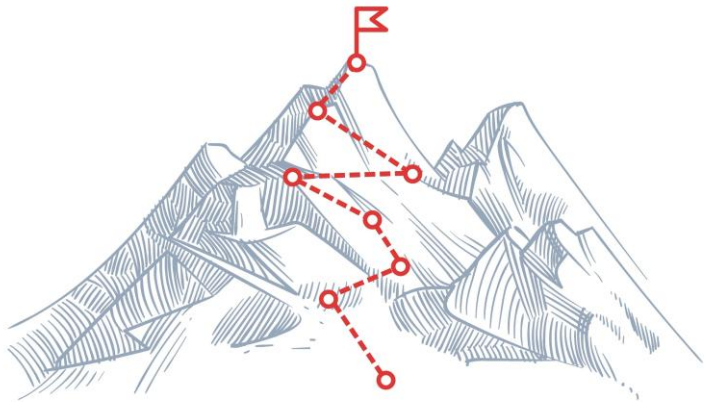
Verilog, Contd.




```
// Bus with a multiplexer
always @ (*) begin
  wBus = wR1;
  case (wsel)
    IDLE:     wBus = wR1;
    R2TOR3:   wBus = wR2;
    R1TOR2:   wBus = wR1;
    R3TOR1:   wBus = wR3;
    default:  wBus = wR1;
  endcase
end
```

Complete the module by inserting the multiplexer block…





Verilog

- Reduction operators

Verilog Reduction Operators

- Reduction operators are unary operators (i.e., take **one operand**) that perform bitwise operations on all bits of the vector operand to produce a single-bit result

&	**reduction and**
~&	**reduction nand**
|	**reduction or**
~|	**reduction nor**
^	**reduction xor**
~^	**reduction xnor**

Verilog Reduction Operators

Contd.

- Reduction **and**, **or**, **xor**
  - The first step of the operation applies the operator between the first bit of the operand and the second
  - The second and subsequent steps apply the operator between the **one-bit result of the prior step** and the next bit of the operand
- Reduction **nand**, **nor**, **xnor**
  - The result is computed by **inverting the result** of the reduction **and**, **or**, and **xor**, respectively

Verilog Reduction Operators

Examples

- A= 8'b10101111
  - Example: z = &A

    z = ((((((1 & 1) & 1) & 1) & 0) & 1) & 0) & 1)

    From least to most significant bit →

    Result: z = 0

  - Example: z = ~^A

    z = ~((((((1 ^ 1) ^ 1) ^ 1) ^ 0) ^ 1) ^ 0) ^ 1)

    Result: z = 1

&	reduction and
~&	reduction nand
|	reduction or
~|	reduction nor
^	reduction xor
~^	reduction xnor

Verilog Reduction Operators

Contd.

- Examples of practical scenarios
  - Zero detection – check if any bit of a bus vector is '1' without a loop
    ```
    wire zero = ~| my_bus;  // 1 if all bits are 0
    ```
  - Parity checking – simple error detection (for memory, communication buses) wire
    ```
    parity = ^data;  // XOR all bits for parity
    ```
  - Fast flag setting—a quick way to set a "done" flag if any of several modules have completed
    ```
    wire done = | done_signals;
    ```
  - Checking "all ones" quickly (e.g., completion/done flags, timers, etc.)
    ```
    wire all_ones = & status_bits;
    ```

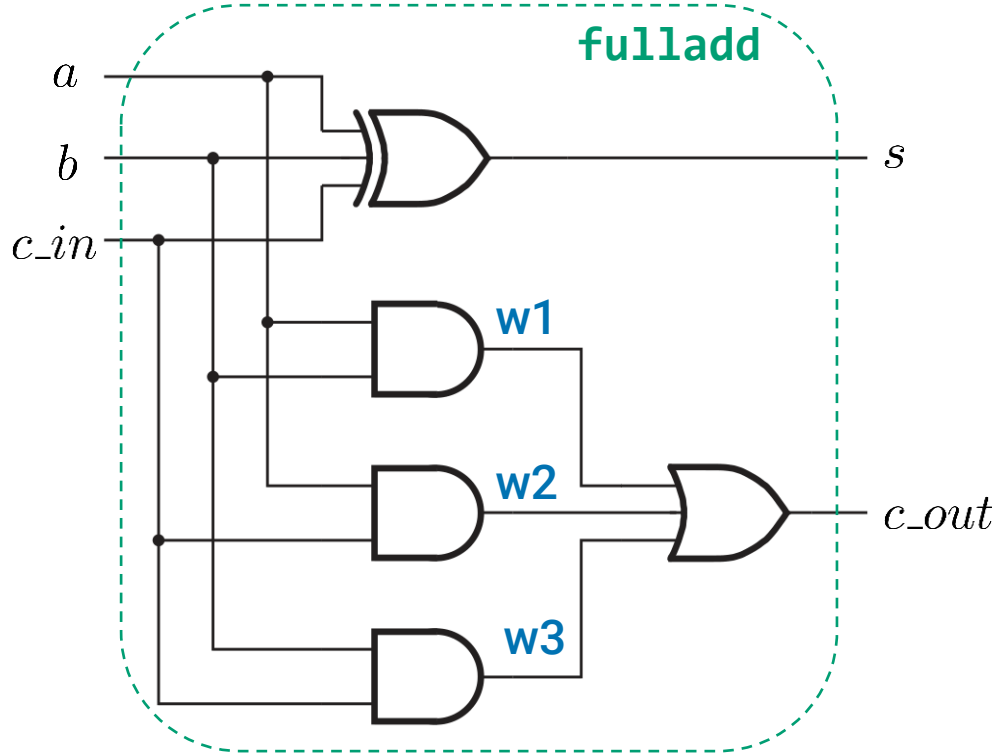
Verilog

- Generate constructs
- Example: Ripple-Carry Adder

Recall: Full Adder

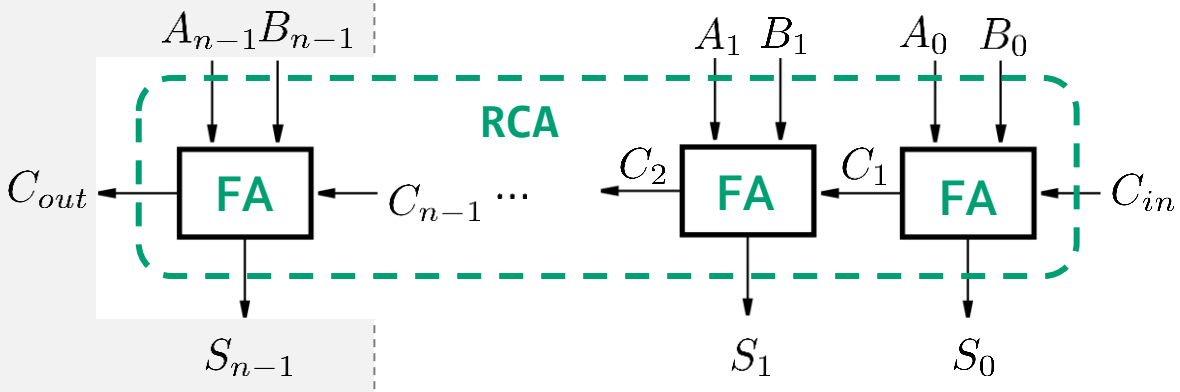
- Behavioral Verilog model

```
module fulladd (a, b, c_in, s, c_out);

  input  a, b, c_in;
  output s, c_out;

  assign s = a ^ b ^ c_in;
  assign c_out = (a & b) | (a & c_in) | (b & c_in);

endmodule
```



Ripple-Carry Adder with a `for` Loop

```
module ripplecarry (Cin, A, B, S, Cout);
  parameter           n = 32;
  input               Cin;
  input [n-1:0]       A, B;
  output reg [n-1:0] S;
  output reg          Cout;

  reg [n:0] C;  // internal wires
  integer   k;  // loop iterator, an integer

  always @(*) begin
    C[0] = Cin;
    for (k = 0; k < n; k = k + 1) begin
      S[k] = A[k] ^ B[k] ^ C[k];
      C[k+1] = (A[k] & B[k]) | (A[k] & C[k]) | (B[k] & C[k]);
    end
    Cout = C[n];
  end
endmodule
```



Verilog Generate Construct

- Can we combine `for` loops with module instantiations? **Yes!**
- Verilog `generate` construct allows module instantiation to be included inside `for` loops and `if-else` statements
  - `generate` construct lets us create multiple instances (loops, or conditionally included) of hardware at compile time—cleanly and systematically—without manually copying code (repetitive, harder maintenance) and introducing errors (esp. when scaling the design)
  - `generate` provides a way to create multiple pieces of hardware based on parameters or iterations

Verilog Generate Construct

- If a `for` loop is included in the `generate` block, the loop index variable has to be of type `genvar`

- `genvar` is an integer variable that can only have values ≥ 0, (it would not make sense to instantiate a negative number of modules) and can only be used inside `generate` blocks

Ripple-Carry Adder with a `generate` Construct

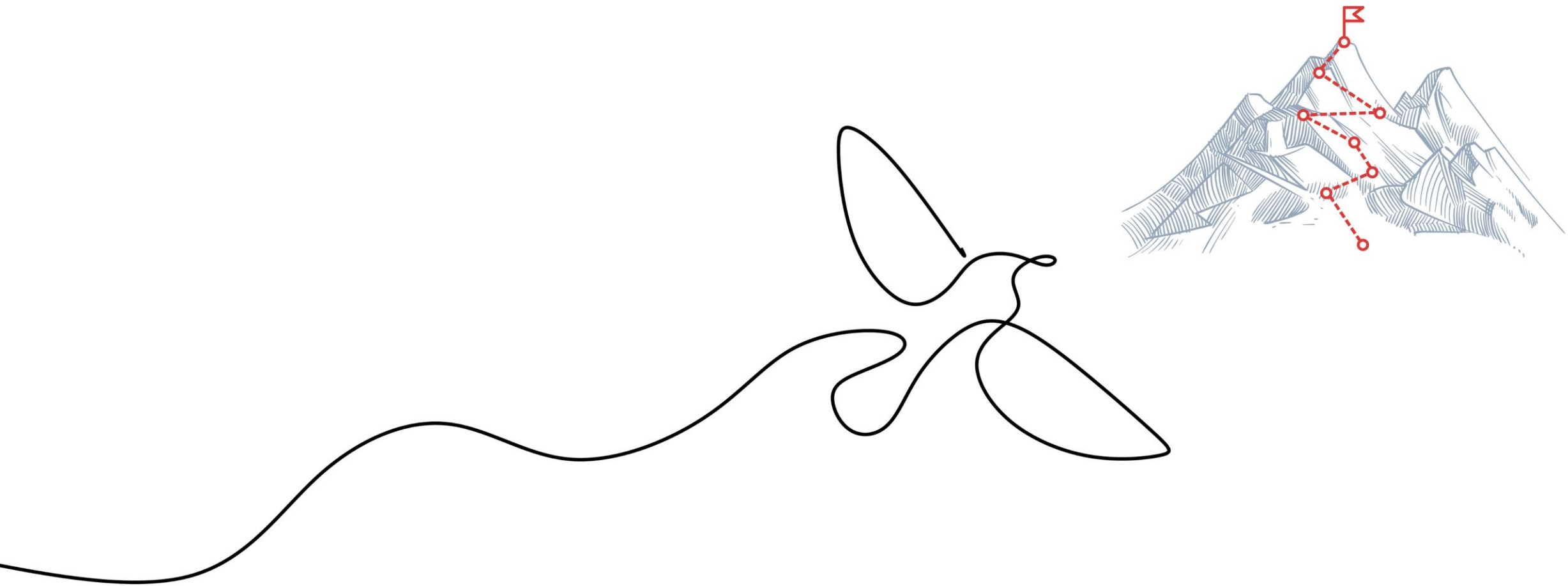
```
module ripplecarrygenerate (Cin, A, B, S, Cout);
  parameter n = 32;
  input  Cin;
  input  [n-1:0] A, B;
  output [n-1:0] S; // must match the type of fulladd port .s
  output Cout;      // must match the type of fulladd port c_out
  genvar g;         // generate loop iterator, must have genvar type
  wire [n:0] C;     // must match the type of fulladd ports .c_in, c_out

  assign C[0] = Cin; // first carry

  // generate block here



  assign Cout = C[n]; // last carry
endmodule
```

Ripple-Carry Adder with a `generate` Construct

- Full adder ports **a**, **b**, **c_in**, **s**, and **c_out** connect to vectors **A**, **B**, carry **C**, and sum **S**, respectively

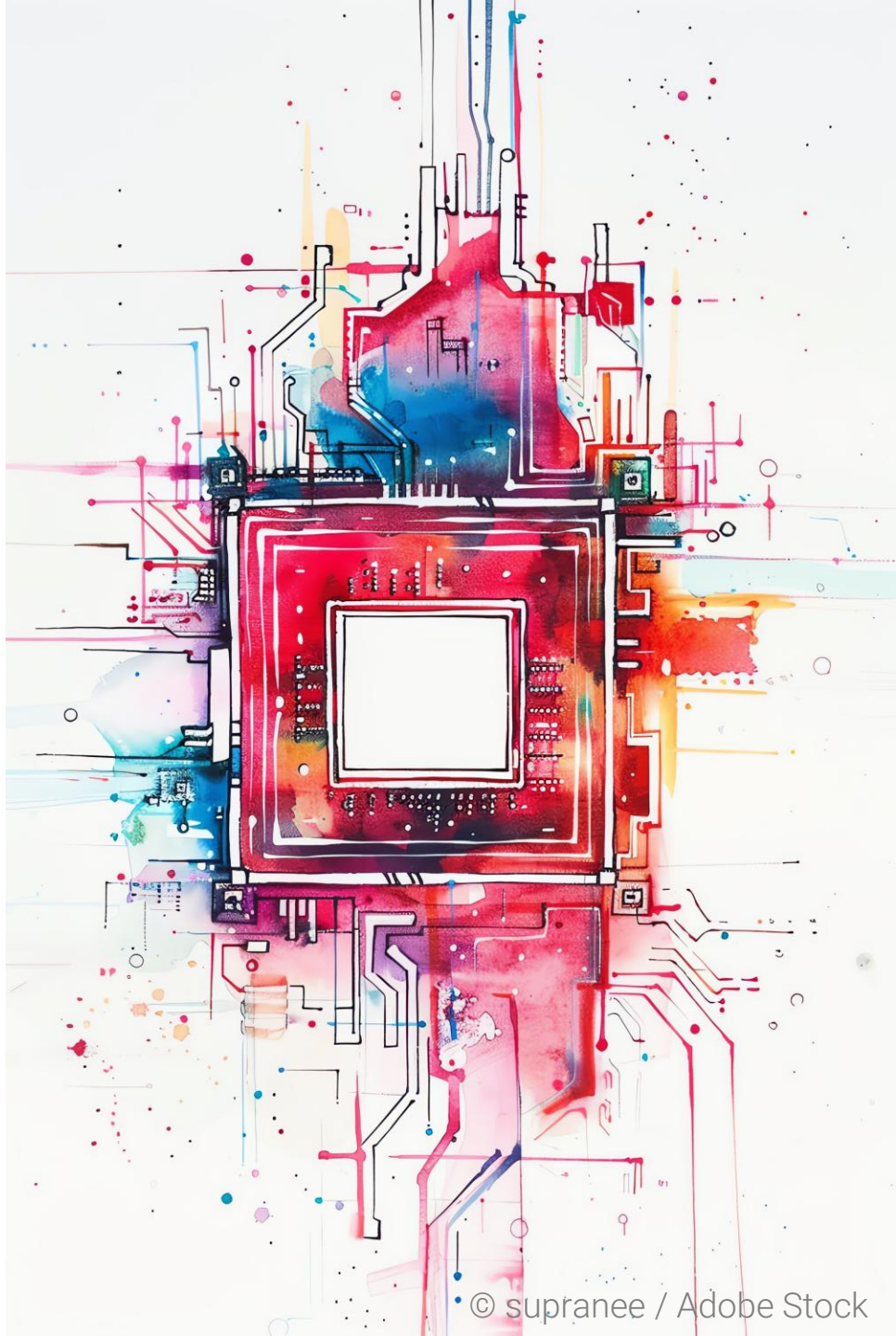
```
// generate block
generate            // optional keyword, helps readability
  for (g = 0; g < n; g = g + 1) begin
    fulladd stage (.a (A[g]), .b (B[g]), .c_in (C[g]), .s (S[g]), .c_out (C[g + 1]));
  end
endgenerate         // optional keyword, helps readability
```

- `fulladd` is the name of the module being instantiated multiple times
- `stage` is the name of one instance of a module; it is user-defined, so choose an appropriate one

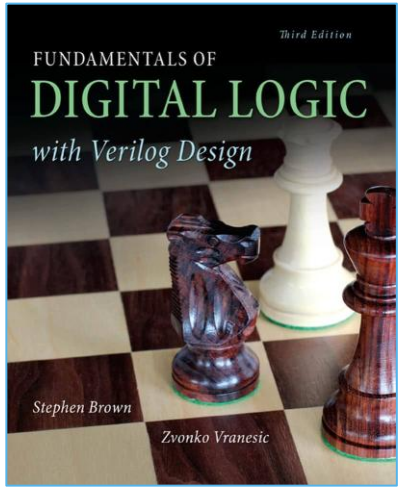


Next on FDS

…Designing a Simple Processor



Literature



- Chapter 3: Number Representation and Arithmetic Circuits
  - 3.5.3, 3.5.4
- Chapter 7: Digital System Design
  - 7.1 Bus Structure