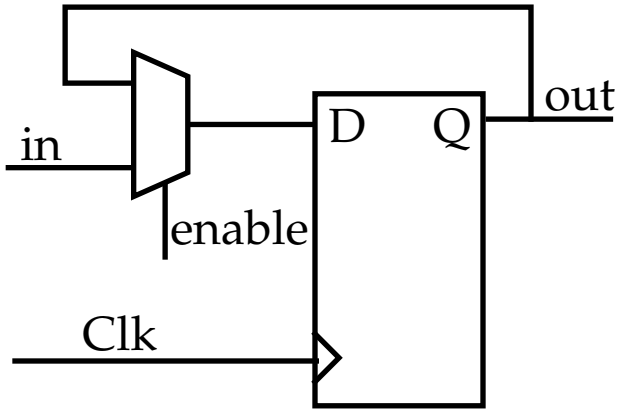
Building Blocks

Digital systems consist of 2 main parts: the **datapath** and **control** circuits.

Datapath: stores and manipulates data and includes components such as registers, shift registers, counters, multiplexers, decoders, adders, etc.

Control: an FSM that controls the datapath elements.

We've talked about many datapath building blocks -- we start here by discussing a few more that are useful in digital system design.



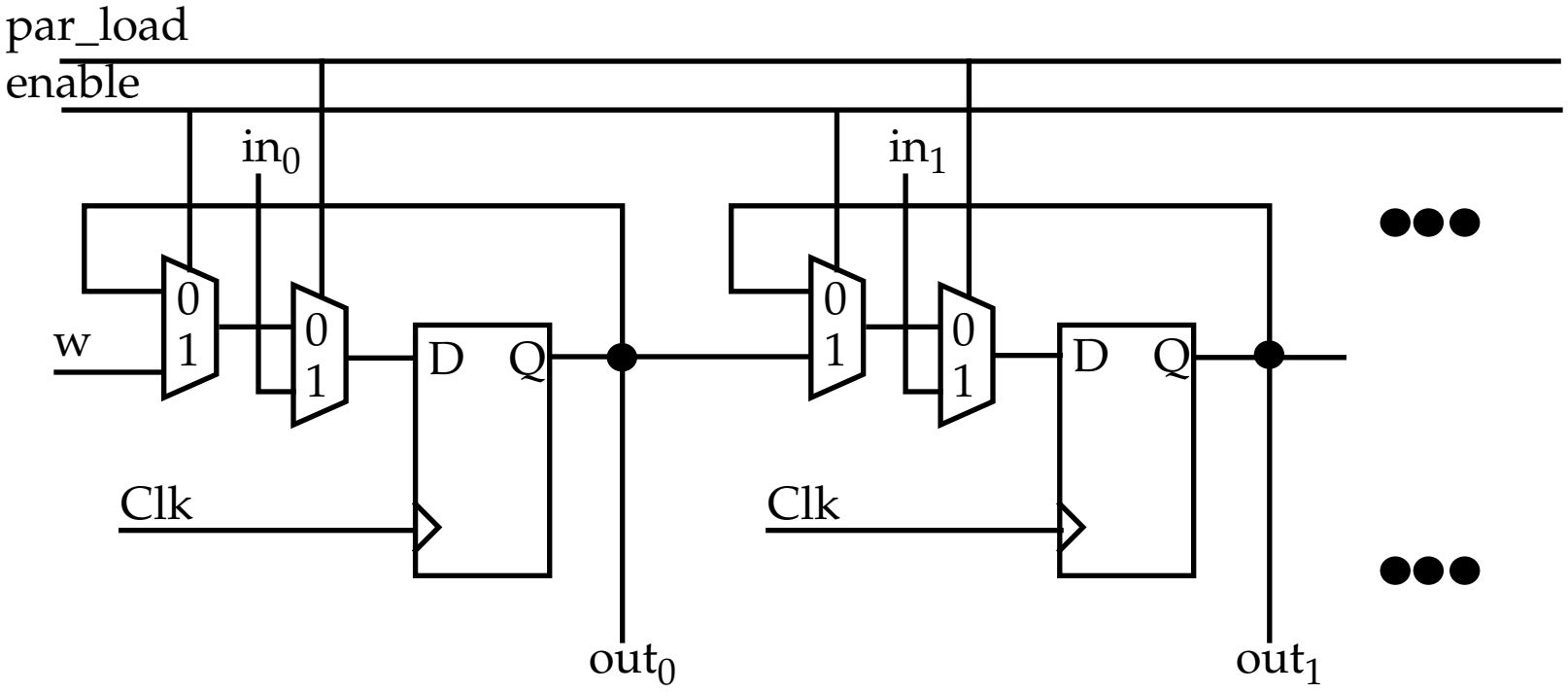

```
module regne(in, Clk, resetn, enable, Q);
   parameter n = 8;
   input [n-1:0] in;
   input Clk, resetn, enable;
   output reg [n-1] Q;
   always @(posedge Clk or negedge Resetn)
      if (resetn == 0)
         Q <= 0;
      else if (enable)
         Q <= in;
endmodule
```

The code describes *n* D FFs with an asynchronous reset and an enable inputs.

The *enable* input allows selective loading of the FFs.

Building Blocks

Left shift register with parallel load and enable inputs.

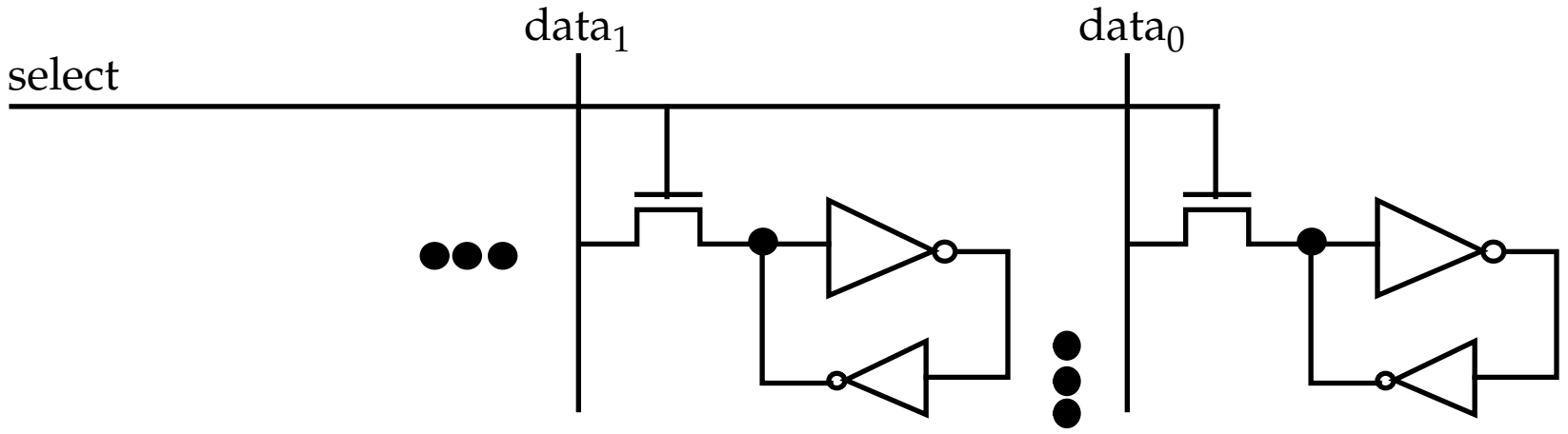



```
module shiftlne(in, par_load, enable, w, Clk, Q);
    parameter n = 4;
    input [n-1:0] in;
    input par_load, enable, w, Clk;
    output reg [n-1] Q;
    integer k;
```

Building Blocks

```
always @(posedge Clk)
    begin
        if (par_load)
            Q <= in;
        else if (enable)
            begin
                Q[0] <= w;              // non-blocking -- all RHS sampled FIRST.
                for (k = 1; k < n; k = k+1)
                    Q[k] <= Q[k-1];
            end
    end
endmodule
```

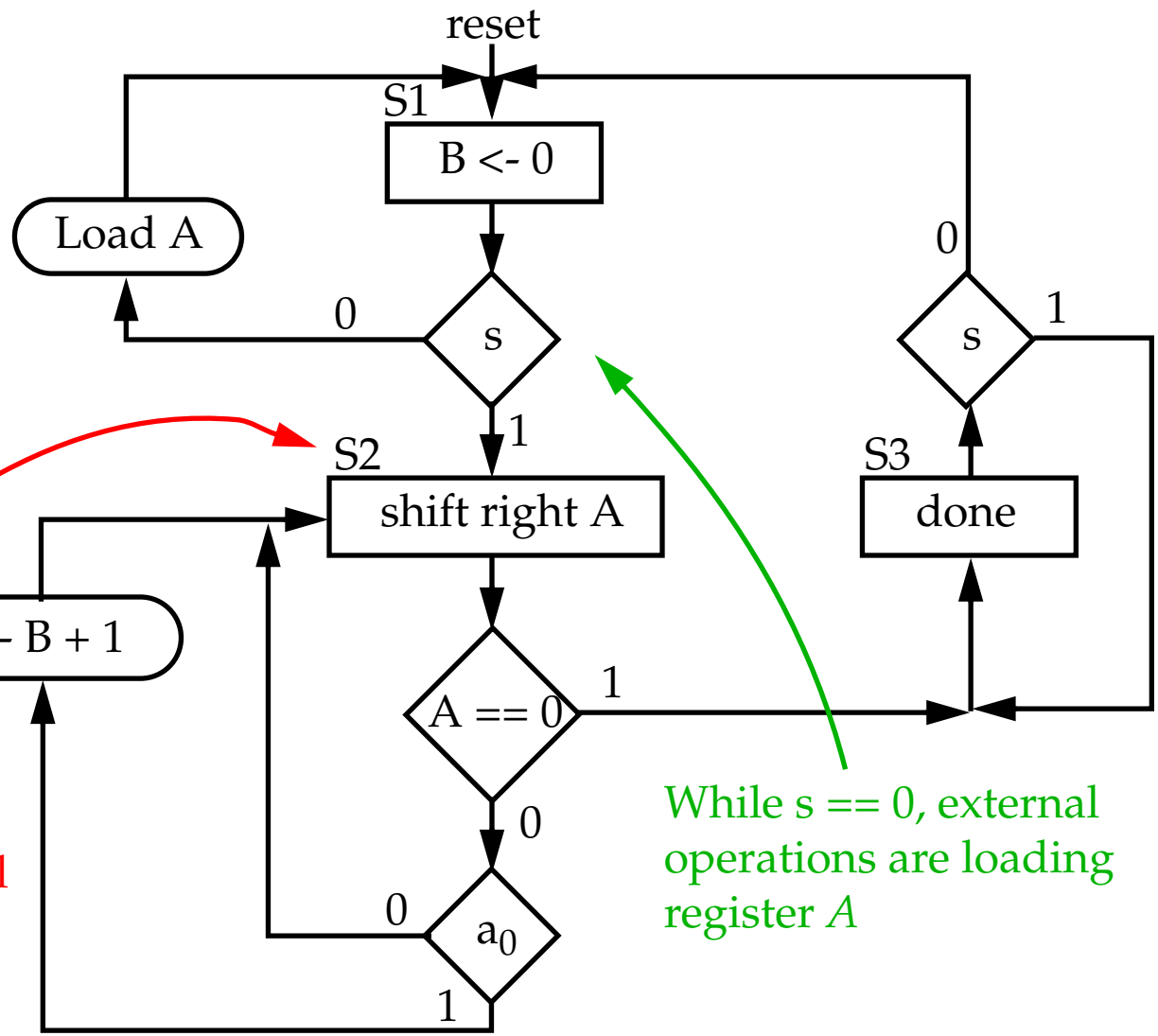
SRAM:



Bit Counting

Count the number of bits in a register that have the value 1.

```
B = 0;
while A /= 0 do
   if a0 = 1 then
      B = B + 1;
   endif
   Right-shift A;
end while;
```



When S2 is entered, A is NOT shifted until the following clock cycle -- we will set an enable signal to allow this.

While s == 0, external operations are loading register *A*

Bit Counting

```
module bitcount (Clk, resetn, load_A, A_ready, data, cnt, done);
  input Clk, resetn, load_A, A_ready;
  input [7:0] data;
  output reg [3:0] B;
  output reg done;
  wire [7:0] A;
  wire A_zero;
  reg [1:0] cur_state, next_state;
  reg en_shift, inc_B, init_B;
  parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;

  always @(A_ready, cur_state, A_zero)              // Next state logic
     begin: State_table
        case (cur_state)
           S1: if (!A_ready) next_state = S1;
                 else next_state = S2;
           S2: if (A_zero == 0) next_state = S2;
                else next_state = S3;
           S3: if (A_ready) next_state = S3;
                 else next_state = S1;
           default: next_state = 2'bxx;
        endcase
    end
```

Bit Counting

```
always @(posedge Clk or negedge resetn)        // Sequential logic
   begin: State_flipflops
      if (resetn == 0)
         cur_state <= S1;
      else
         cur_state <= next_state;
   end

always @(cur_state or A[0])          // Combo logic for output signals
   begin: FSM_outputs
      en_shift = 0; inc_B = 0; init_B = 0; done = 0;
      case (y)
         S1: init_B = 1;
         S2:  begin
                 en_shift = 1;          // Enabling the shift control signal
                 if (A[0]) inc_B = 1;   // is asserted on entering state S2 --
                 else inc_B = 0;        // so it's not available until next Clk
              end
         S3: done = 1;
      endcase
   end
```

Same is true for inc_B signal -- they are not set/unset until state S2 is active so actions taken by asserting signals in this state don't take place until next Clk.

Bit Counting

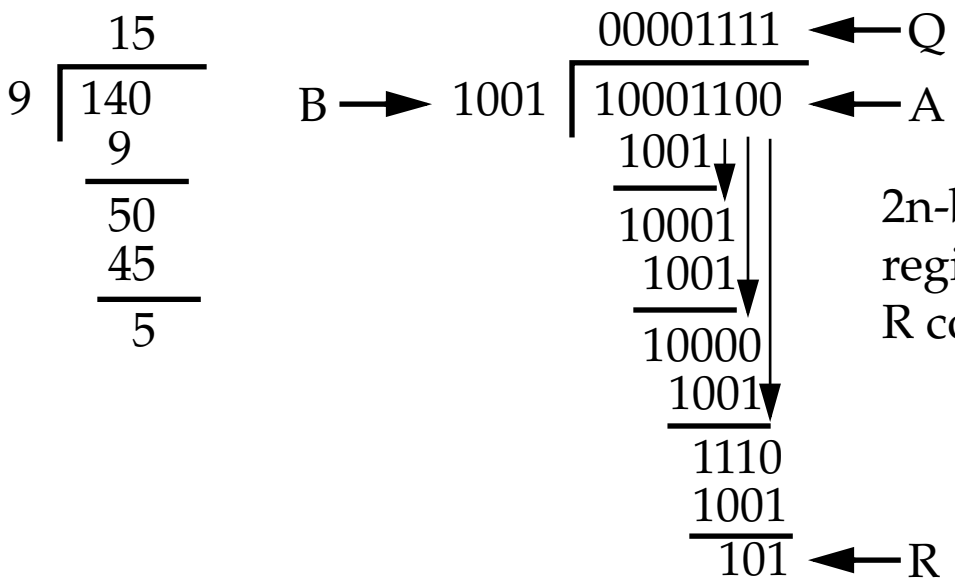
```
always @(posedge Clk or negedge resetn)        // Seq. logic for cnter B
    if (resetn)
        B <= 0;
    else if (init_B)
        B <= 0;
    else if (inc_B)
        B <= B + 1;

    shiftrne shift_A(data, load_A, en_shift, 1'b0, Clk, A)    // RIGHT shifter
    assign A_zero = ~|A;
endmodule
```

Note that resetn is NOT used to initialize *B* after the initial reset of the machine during 'power up'.

A separate signal *init_B* is used to do this when the FSM transitions from state S3 to S1.

Divider

The example illustrates division using a the traditional long-hand approach.



2n-bit shift register with R concat with A

```
R = 0;
for i = 0 to n-1 do
   left-shift R || A;
   if R >= B then
      q_i = 1;
      R = R - B;
   else
      q_i = 0;
   endif;
endfor;
```

Pseudo-code illustrates operation where A is left shifted, one bit at a time, into R and then R (**not** R || A) is compared with B.

Q is computed by left shifting a 1 or 0 into the least significant digit based on the comparison of R with B, i.e., if R >= B, shift a 1, else a 0.

The remainder, R, is what remains after n clks.

Divider

We use a left-shift register with parallel load for R to handle two cases.

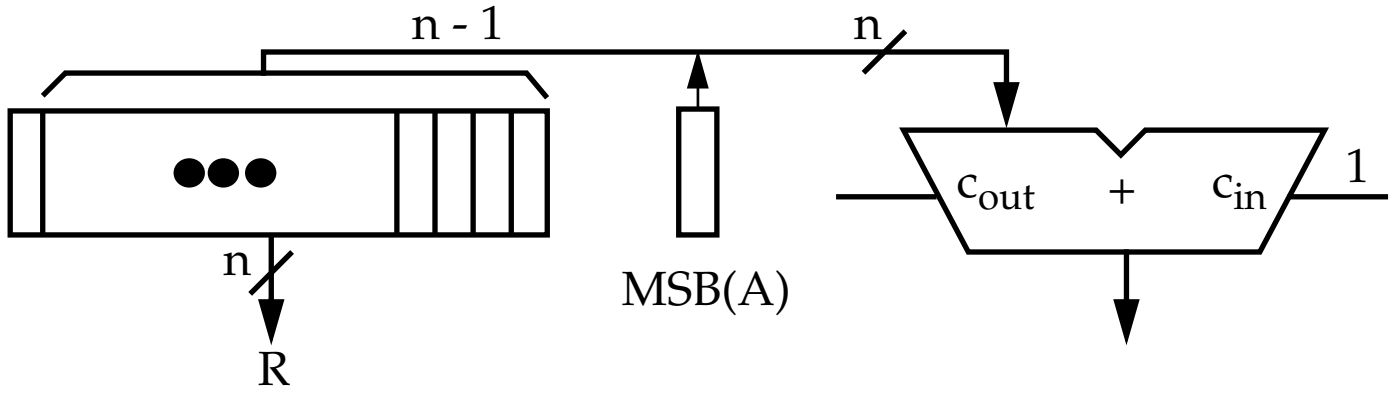
- If R becomes greater than B (remember, A is shifted into R one bit at a time), then the new value of R is R - B.
- If R is less than B, than we shift the MSB bit of A into R.

In either case, the R inputs to the subtractor must be driven with the low order *n-1* bits of the register R concatenated with the MSB of A.

R inputs of adder = R[n-2:0] | | MSB(A)

However, R itself is R[n-1:0] when the division is completed.

To accomplish this, we keep the MSB(A) in a separate 1-bit register:



Divider

Register *A* is used to store the quotient by left shifting as *A* is shifted out:

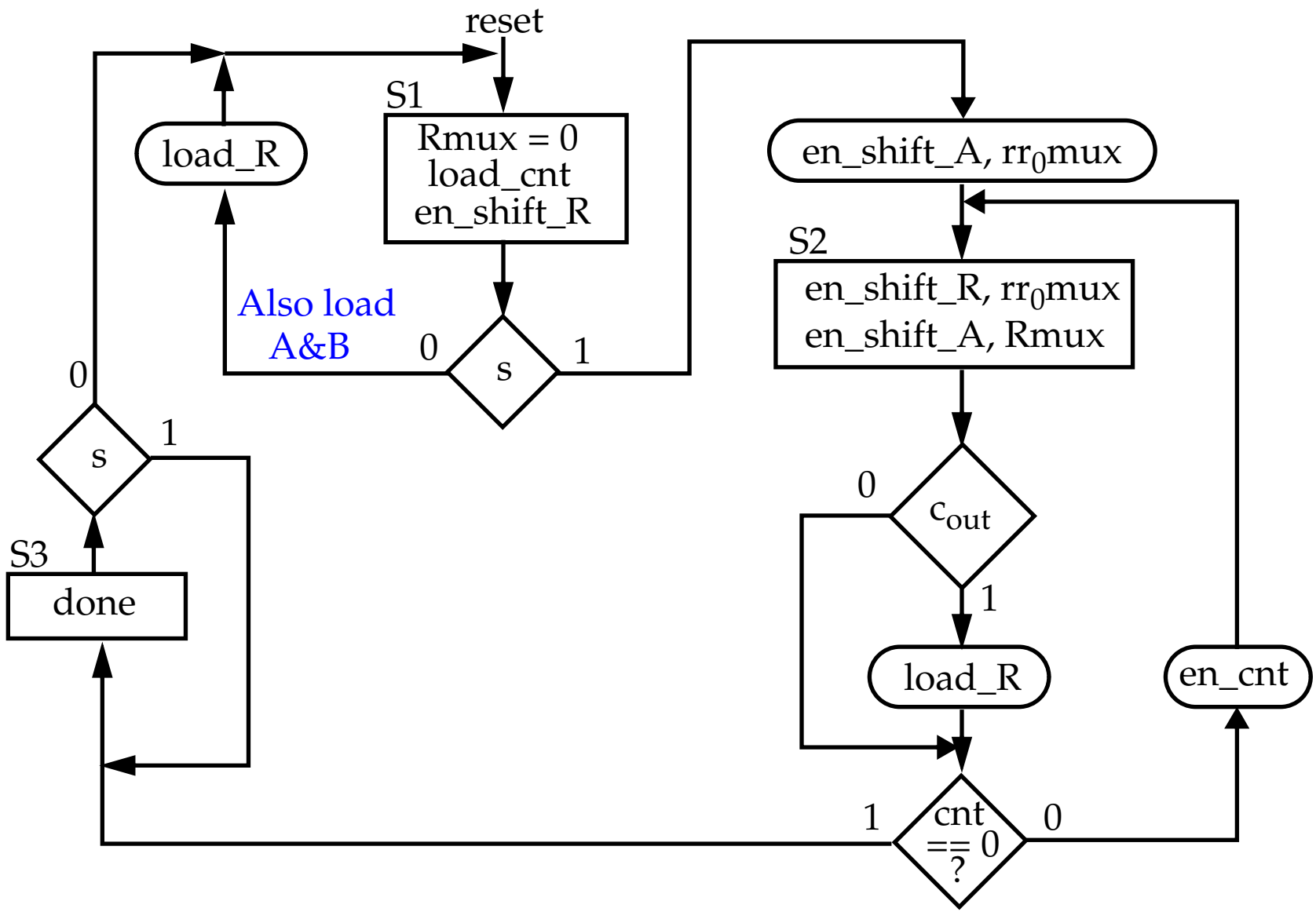
Clk cycle	R	rr_0	A/Q
Load A, B	0 0 0 0 0 0 0 0	0	1 0 0 0 1 1 0 0
0 Shift left	0 0 0 0 0 0 0 0	1	0 0 0 1 1 0 0 0
1 Shift left, Q_0<-0	0 0 0 0 0 0 0 1	0	0 0 1 1 0 0 0 **0**
2 Shift left, Q_0<-0	0 0 0 0 0 0 1 0	0	0 1 1 0 0 0 **0 0**
3 Shift left, Q_0<-0	0 0 0 0 0 1 0 0	0	1 1 0 0 0 **0 0 0**
4 Shift left, Q_0<-0	0 0 0 0 1 0 0 0	1	1 0 0 0 **0 0 0 0**
5 Subtract, Q_0<-1	0 0 0 0 1 0 0 0	1	0 0 0 **0 0 0 0 1**
6 Subtract, Q_0<-1	0 0 0 0 1 0 0 0	0	0 0 **0 0 0 0 1 1**
7 Subtract, Q_0<-1	0 0 0 0 0 1 1 1	0	0 **0 0 0 0 1 1 1**
8 Subtract, Q_0<-1	0 0 0 0 0 1 0 1	0	**0 0 0 0 1 1 1 1**

At clk cycle 0, A's MSB is left shifted into rr_0, yielding $R \,||\, rr_0$ = 0_0000_0001, which is smaller than *B* (1001).

At clk cycle 1, rr_0 is left shifted into R while A's MSB moves into rr_0. Also, a 0 is shifted into the LSB of *A* to indicate a 0 in the quotient.

At clk cycle 4, $R \,||\, rr_0$ = 0_0001_0001, which is > *B*, so in clk cycle 5, the result of subtraction 0001_0001 - 1001 = 0000_1000 is loaded into *R*.

Divider


reset
S1
Rmux = 0
load_cnt
en_shift_R
load_R
Also load
A&B
0
s
1
en_shift_A, rr0mux
S2
en_shift_R, rr0mux
en_shift_A, Rmux
0
cout
1
load_R
en_cnt
1
cnt
== 0
?
0
0
s
1
S3
done

Divider

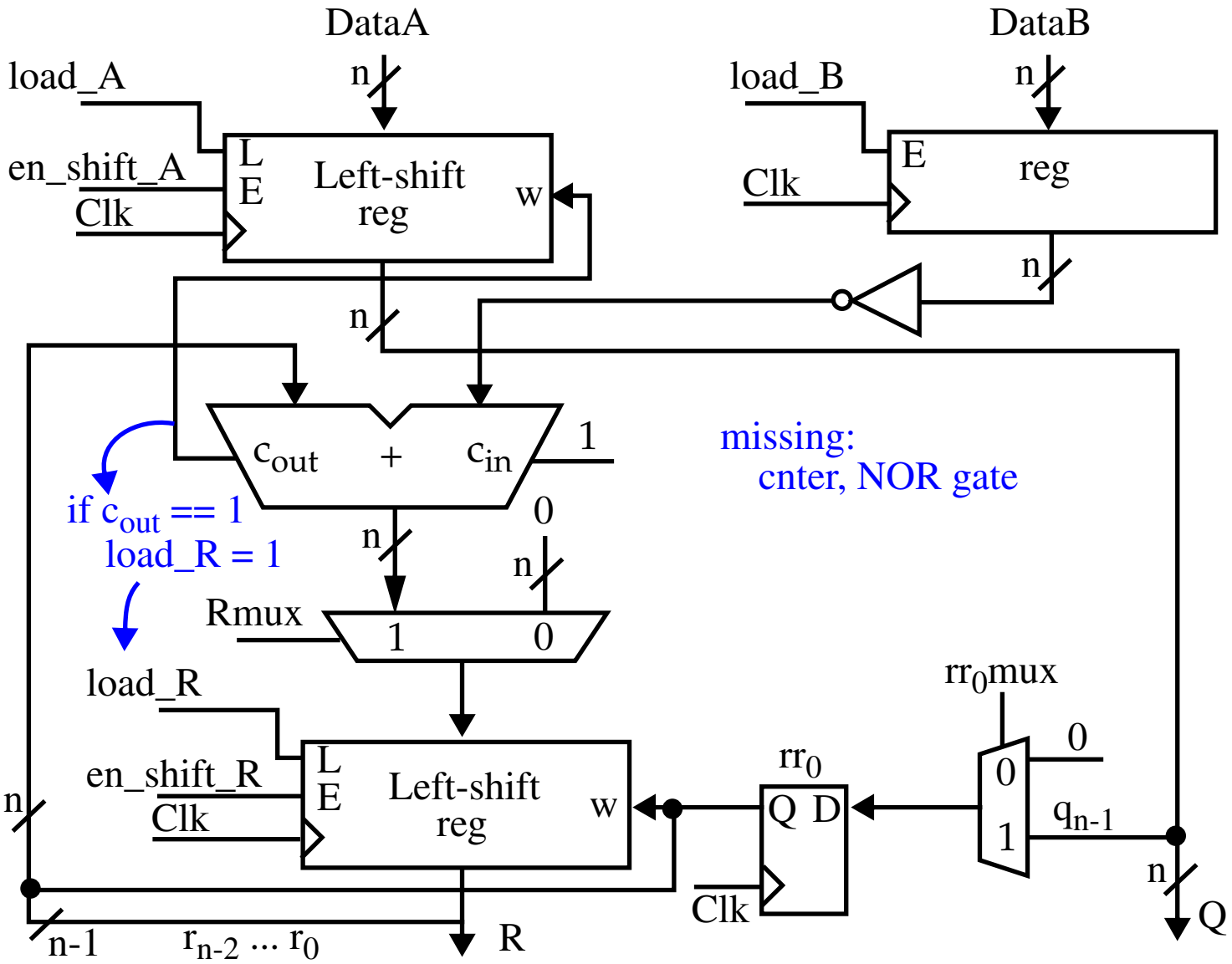
Hardware requirements:

- Register for B.
- Two shift registers for A and R.
- A subtractor for R-B (implemented as an adder with carry = 1 and B complemented).

  The c_{out} of this module is 1 if R >= B.

  c_{out} connected to the serial input of the shift reg that stores Q.
- A multiplexer feeding the input to R because it is loaded with 0 in state S1 and from the output of the adder in S3.
- A down counter to implement *cnt*.
- A NOR gate to determine when C == 0.

Divider


DataA
DataB
load_A
load_B
n
en_shift_A
Clk
L
E
Left-shift
reg
w
E
reg
missing:
cnter, NOR gate
c_{out}
+
c_{in}
1
0
if c_{out} == 1
load_R = 1
Rmux
load_R
en_shift_R
rr_0mux
rr_0
Q D
q_{n-1}
n-1
r_{n-2} ... r_0
R
Q

Divider

```
module divider (Clk, resetn, s, load_A, load_B, DataA, DataB, R, Q, done);
  parameter n = 8, logn = 3;
  input Clk, resetn, s, load_A, load_B;
  input [n-1:0] DataA, DataB;
  output [n-1:0] R, Q;
  output reg done;
  wire Cout, cnt_zero, rr0;
  wire [n-1:0] DataR;
  wire [n:0] Sum;
  reg [1:0] cur_state, next_state;
  wire [n-1:0] A, B;
  wire [logn-1:0] cnt;
  reg en_shift_A, Rmux, load_R, en_shift_R, rr0mux, load_cnt, en_cnt;

  parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;
```

Divider

```
always @(s, cur_state, cnt_zero)                    // Next state logic
  begin: State_table
    case (cur_state)
      S1: if (s == 0) next_state = S1;
          else next_state = S2;
      S2: if (cnt_zero == 0) next_state = S2;
          else next_state = S3;
      S3: if (s == 1) next_state = S3;
          else next_state = S1;
      default: next_state = 2'bxx;
    endcase
end

always @(posedge Clk or negedge resetn)        // Sequential logic
  begin: State_flipflops
    if (resetn == 0)
      cur_state <= S1;
    else
      cur_state <= next_state;
  end
```

Divider

```
always @(cur_state or s or Cout or cnt_zero)        // Combo logic for
   begin: FSM_outputs                               // output signals
      load_R = 0; en_shift_R = 0; rr0mux = 0;
      load_cnt = 0; en_cnt = 0; en_shift_A = 0;
      case (y)
         S1: begin
                load_cnt = 1; en_shift_R = 1;
                if (s == 0) begin
                     load_R = 1; rr0mux = 0;
                  end
                else begin
                     load_R = 0; en_shift_A = 1, rr0mux = 1;
                  end
         S2:  begin
                Rmux = 1; en_shift_R = 1; rr0mux = 1; en_shift_A = 1;
                if (Cout) load_R = 1;
                else load_R = 0;
                if (cnt_zero == 0) en_cnt = 1;
                else en_cnt = 0;
               end
         S3: done = 1;
      endcase
   end
```

Divider

```
// Datapath

        regne RegB(DataB, Clk, resetn, en_shift_B, B);
          defparam RegB.n = n;
        shiftlne ShiftR(DataR, load_R, en_shift_R, rr0, Clk, R);      // Shift reg R
          defparam ShiftR.n = n;
        muxdff FF_rr0(1'b0, A[n-1], rr0mux, Clk, rr0);                // Single bit
        shiftlne ShiftA(DataA, load_A, en_shift_A, Cout, Clk, A);     // Shift reg A
          defparam ShiftA.n = n;

        assign Q = A;            // Output of A is Q once calc performed

        downcount Cnter(Clk, en_cnt, load_cnt, Count);                // Counter
          defparam Cnter.n = logn;

        assign cnt_zero = (Count == 0);
        assign Sum = {1'b0, R[n-2:0], rr0} + {1'b0, ~B} + 1;   // Adder is n+1
        assign Cout = Sum[n];                                  // bits to save
                                                               // Cout
        assign DataR = Rmux ? Sum : 0;                         // n 2-to-1 MUXs

      endmodule
```