

Mastering Digital Design

with Verilog on FPGAs

John Wickerson

Lecture 2

This lecture

- Arithmetic and logical operators in Verilog
- Clocked circuits
- Shift registers
- Converting from binary to binary-coded decimal (BCD)

Arithmetic and Logical Operators in Verilog


Integer Arithmetic

- Integer arithmetic without carries:

```
module add32 (a, b, sum);  
  input[31:0] a, b;  
  output[31:0] sum;  
  assign sum = a + b;  
endmodule
```

- Integer arithmetic with carry-in and carry-out:

```
module add32_carry (a, b, cin, sum, cout);  
  input[31:0] a, b;  
  input cin;  
  output[31:0] sum;  
  output cout;  
  assign {cout, sum} = a + b + cin;  
endmodule
```



What is the width of the **+**-operator?

Boolean operators

- Bitwise operators:

Boolean operators

- Bitwise operators:
`~(4'b0101)`

Boolean operators

- Bitwise operators:

$\sim(4'b0101) = \{\sim 0, \sim 1, \sim 0, \sim 1\}$

Boolean operators

- Bitwise operators:

$\sim(4'b0101) = \{\sim 0, \sim 1, \sim 0, \sim 1\} = \{1, 0, 1, 0\}$

Boolean operators

- Bitwise operators:

$$\sim(4'b0101) = \{\sim 0, \sim 1, \sim 0, \sim 1\} = \{1, 0, 1, 0\} = 4'b1010$$

Boolean operators

- Bitwise operators:

$\sim(4'b0101) = \{\sim 0, \sim 1, \sim 0, \sim 1\} = \{1, 0, 1, 0\} = 4'b1010$

$4'b0101 \& 4'b0011$

Boolean operators

- Bitwise operators:

$\sim(4'b0101) = \{\sim 0, \sim 1, \sim 0, \sim 1\} = \{1, 0, 1, 0\} = 4'b1010$

$4'b0101 \& 4'b0011 = 4'b0001$

Boolean operators

- Bitwise operators:

$\sim(4'b0101) = \{\sim 0, \sim 1, \sim 0, \sim 1\} = \{1, 0, 1, 0\} = 4'b1010$

$4'b0101 \& 4'b0011 = 4'b0001$

- Logical operators:

Boolean operators

- Bitwise operators:

$\sim(4'b0101) = \{\sim 0, \sim 1, \sim 0, \sim 1\} = \{1, 0, 1, 0\} = 4'b1010$

$4'b0101 \& 4'b0011 = 4'b0001$

- Logical operators:

$!(4'b0101)$

Boolean operators

- Bitwise operators:

$\sim(4'b0101) = \{\sim 0, \sim 1, \sim 0, \sim 1\} = \{1, 0, 1, 0\} = 4'b1010$

$4'b0101 \& 4'b0011 = 4'b0001$

- Logical operators:

$!(4'b0101) = 0$

Boolean operators

- Bitwise operators:

$\sim(4'b0101) = \{\sim 0, \sim 1, \sim 0, \sim 1\} = \{1, 0, 1, 0\} = 4'b1010$

$4'b0101 \& 4'b0011 = 4'b0001$

- Logical operators:

$!(4'b0101) = 0$

- Reduction operators:

Boolean operators

- Bitwise operators:

$\sim(4'b0101) = \{\sim 0, \sim 1, \sim 0, \sim 1\} = \{1, 0, 1, 0\} = 4'b1010$

$4'b0101 \& 4'b0011 = 4'b0001$

- Logical operators:

$!(4'b0101) = 0$

- Reduction operators:

$\&(4'b0101)$

Boolean operators

- Bitwise operators:

$\sim(4'b0101) = \{\sim 0, \sim 1, \sim 0, \sim 1\} = \{1, 0, 1, 0\} = 4'b1010$

$4'b0101 \& 4'b0011 = 4'b0001$

- Logical operators:

$!(4'b0101) = 0$

- Reduction operators:

$\&(4'b0101) = 0 \& 1 \& 0 \& 1$

Boolean operators

- Bitwise operators:

$$\sim(4'b0101) = \{\sim 0, \sim 1, \sim 0, \sim 1\} = \{1, 0, 1, 0\} = 4'b1010$$

$$4'b0101 \& 4'b0011 = 4'b0001$$

- Logical operators:

$$!(4'b0101) = 0$$

- Reduction operators:

$$\&(4'b0101) = 0 \& 1 \& 0 \& 1 = 1'b0$$

Boolean operators

Bitwise

$\sim a$	NOT
$a \& b$	AND
$a b$	OR
$a \wedge b$	XOR
$a \sim \wedge b$	XNOR


Logical

$!a$	NOT
$a \&\& b$	AND
$a b$	OR

Reduction

$\&a$	AND
$\sim \&$	NAND
$ $	OR
$\sim $	NOR
\wedge	XOR

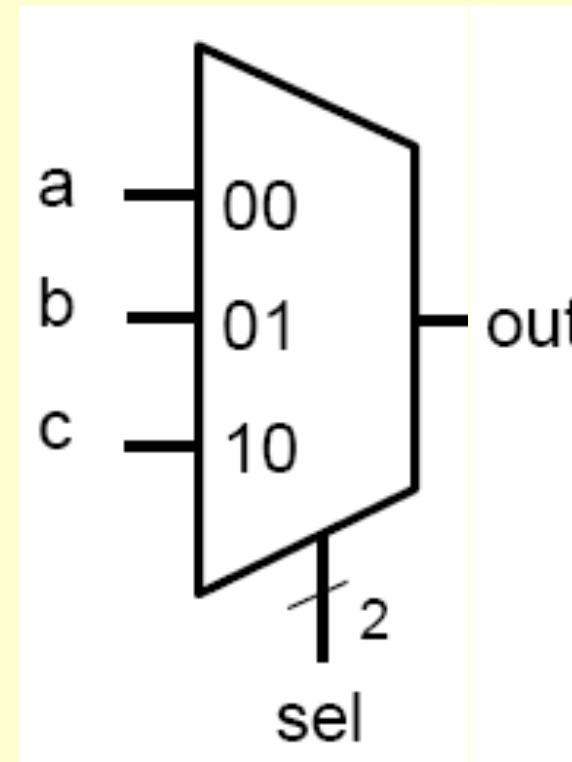
What is the difference
between $\sim a$ and $!a$?



Incomplete specification

- If **out** is not assigned in an always block, the previous value must be retained.

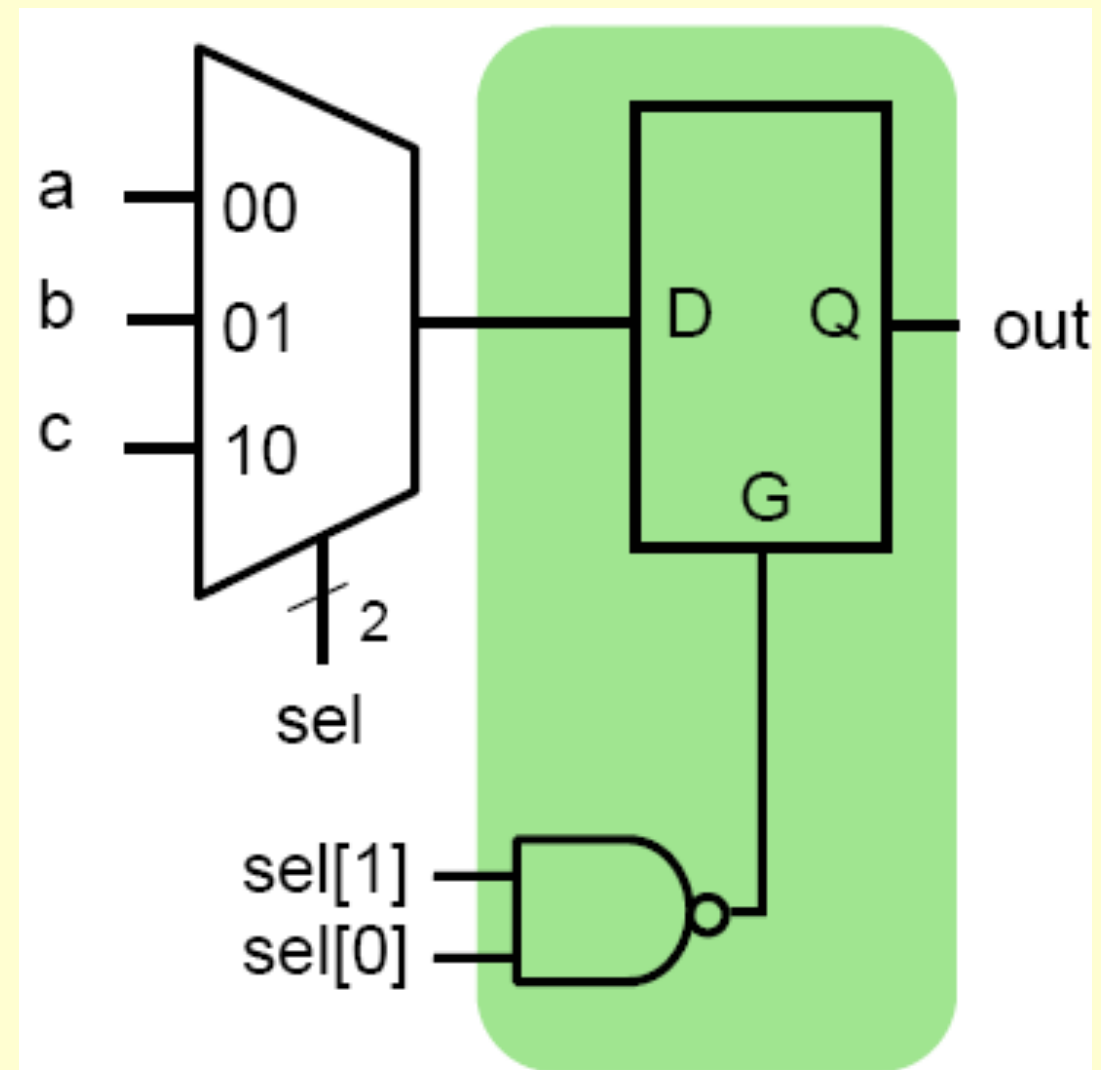
```
module maybe_mux_3to1
    (a, b, c, sel, out);
    input[1:0] sel;
    input a, b, c;
    output out;
    reg out;
    always @*
    begin
        case (sel)
            2'b00: out = a;
            2'b01: out = b;
            2'b10: out = c;
        endcase
    end
endmodule
```



Incomplete specification

- If **out** is not assigned in an always block, the previous value must be retained.

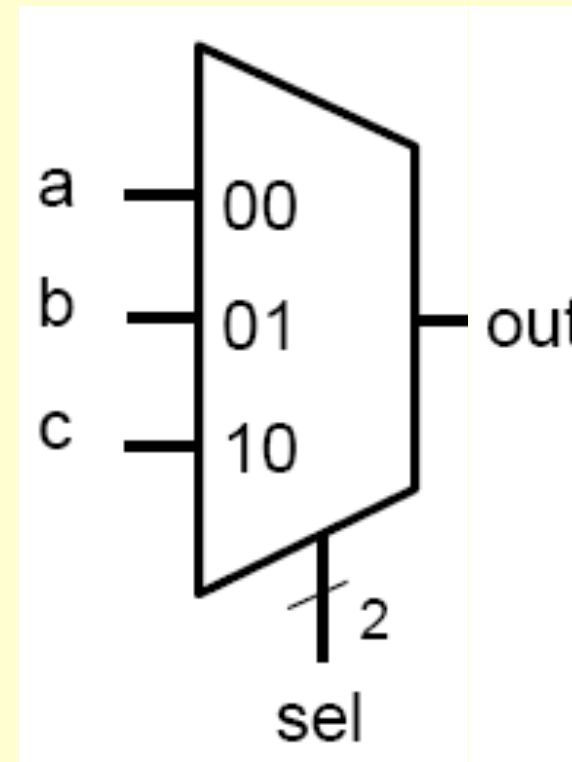
```
module maybe_mux_3to1
    (a, b, c, sel, out);
    input[1:0] sel;
    input a, b, c;
    output out;
    reg out;
    always @*
    begin
        case (sel)
            2'b00: out = a;
            2'b01: out = b;
            2'b10: out = c;
        endcase
    end
endmodule
```



Incomplete specification

- Could fix by preceding with a default assignment...

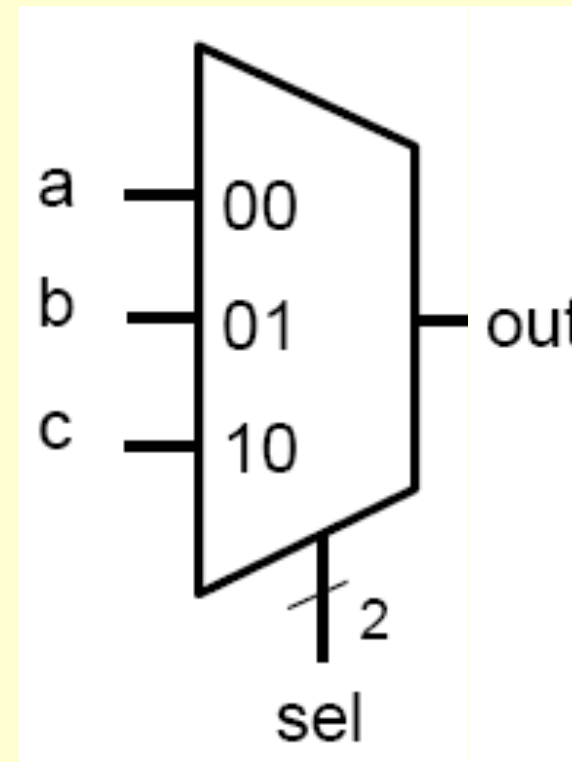
```
module maybe_mux_3to1
    (a, b, c, sel, out);
    input[1:0] sel;
    input a, b, c;
    output out;
    reg out;
    always @*
    begin
        out = 1'bX;
        case (sel)
            2'b00: out = a;
            2'b01: out = b;
            2'b10: out = c;
        endcase
    end
endmodule
```



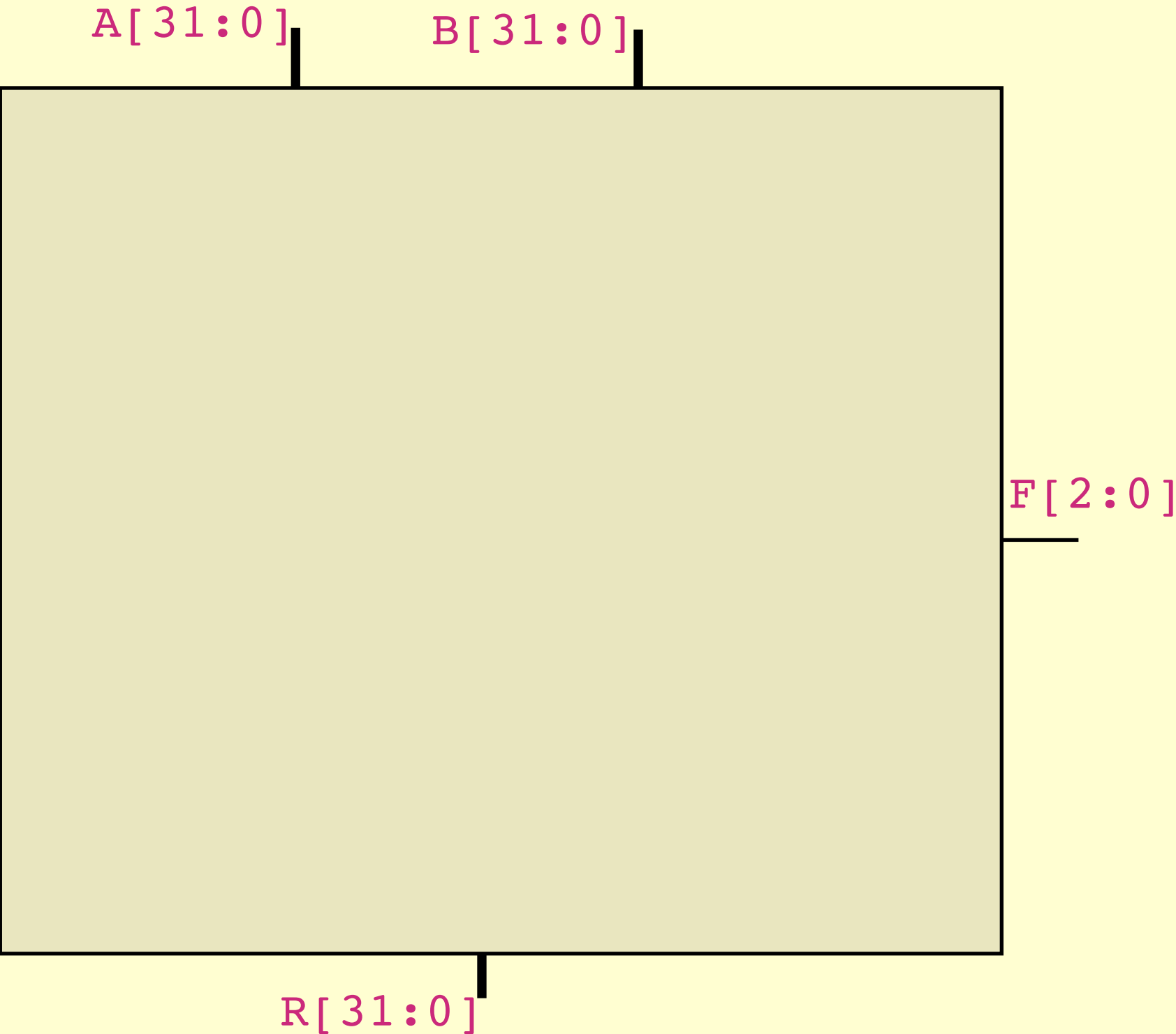
Incomplete specification

- Could fix by preceding with a default assignment...
... or by using a **default** case.

```
module maybe_mux_3to1
    (a, b, c, sel, out);
    input[1:0] sel;
    input a, b, c;
    output out;
    reg out;
    always @*
    begin
        case (sel)
            2'b00: out = a;
            2'b01: out = b;
            2'b10: out = c;
            default: out = 1'bX;
        endcase
    end
endmodule
```



An ALU



F[2:0]	R
{0,0,0}	A + B
{0,0,1}	A + 1
{0,1,0}	A - B
{0,1,1}	A - 1
{1,0,X}	A * B

An ALU

```
module mux32two (a, b, sel, out);
  input sel;
  input[31:0] a, b;
  output[31:0] out;
  assign out = sel ? b : a;
endmodule
```

```
module alu (A, B, F, R);
  input[31:0] A, B;
  input[2:0] F;
  output[31:0] R;
  wire [31:0] w1,w2,w3,w4,w5;
  mux32two add_mux (B, 32'd1, F[0], w1);
  mux32two sub_mux (B, 32'd1, F[0], w2);
  add32 (A, w1, w3);
  sub32 (A, w2, w4);
  mul16 (A[15:0], B[15:0], w5);
  mux32three (w3, w4, w5, F[2:1], R);
endmodule
```

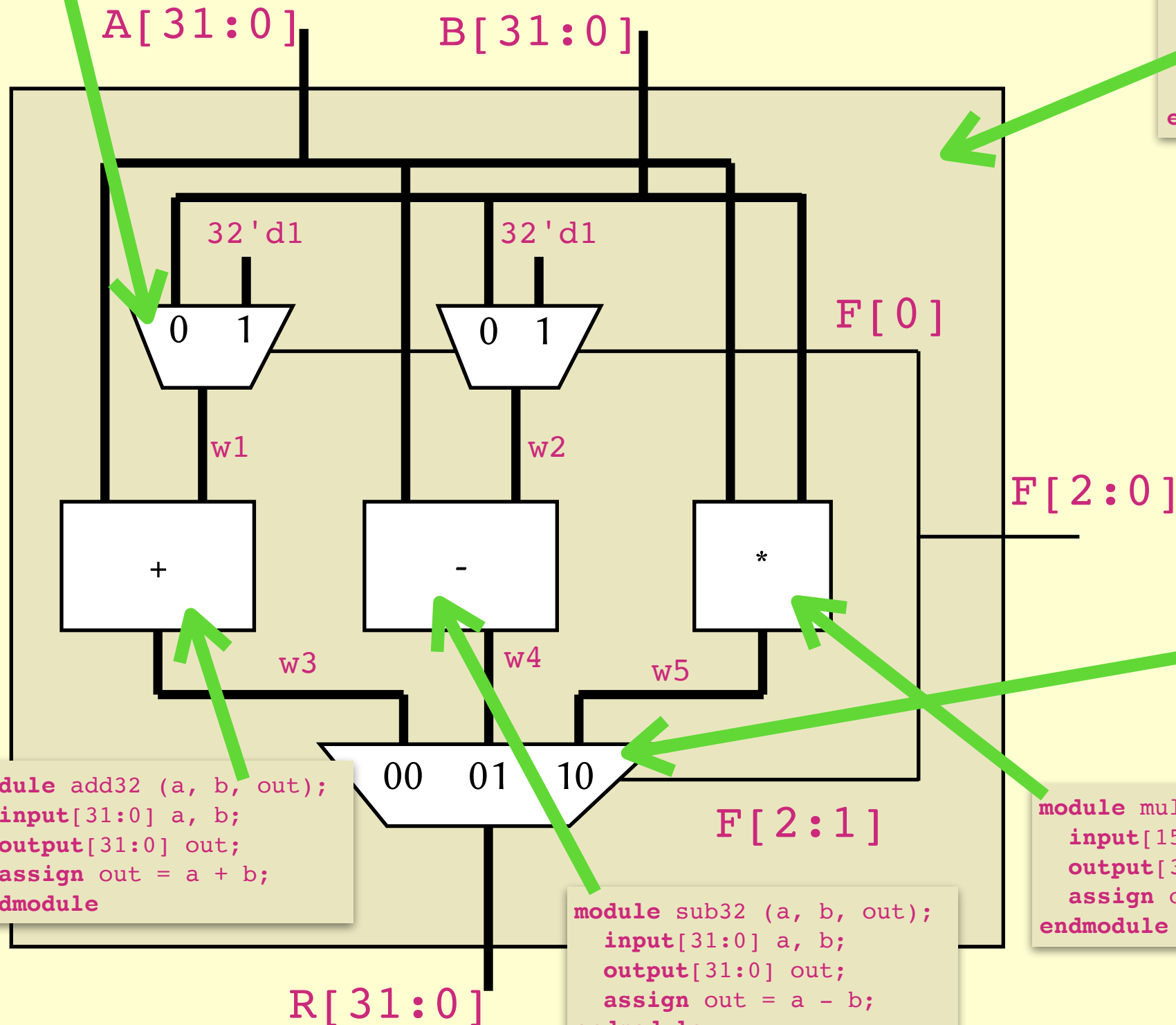
{0,1,1}	A - 1
{1,0,X}	A * B

```
module mux32three (a, b, c, sel, out);
  input[1:0] sel;
  input[31:0] a, b, c;
  output[31:0] out;
  reg[31:0] out;
  always @*
  begin
    case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
      default: out = 32'bx;
    endcase
  end
endmodule
```

```
module add32 (a, b, out);
  input[31:0] a, b;
  output[31:0] out;
  assign out = a + b;
endmodule
```

```
module sub32 (a, b, out);
  input[31:0] a, b;
  output[31:0] out;
  assign out = a - b;
endmodule
```

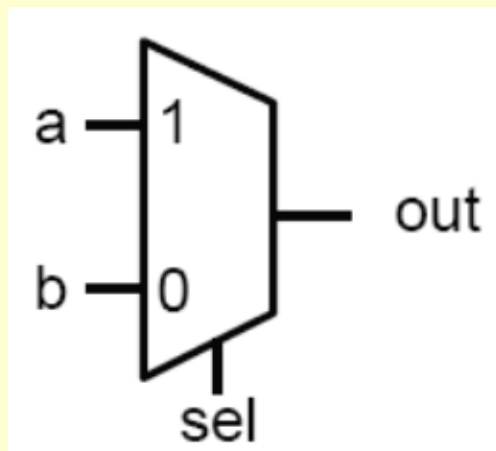
```
module mul16 (a, b, out);
  input[15:0] a, b;
  output[31:0] out;
  assign out = a * b;
endmodule
```



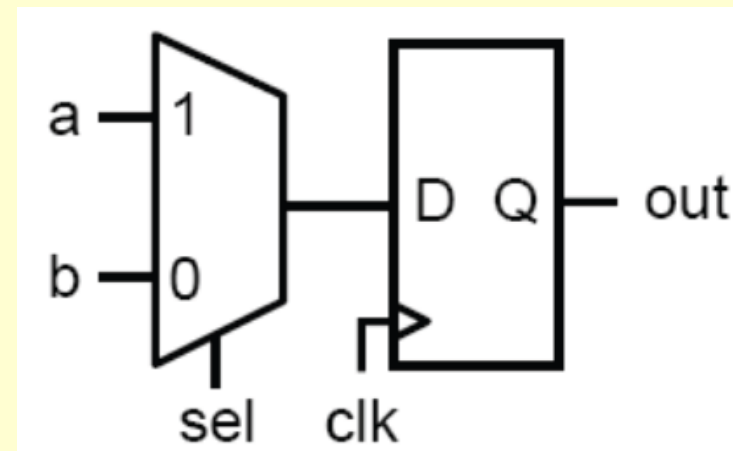
Clocked circuits

Sequential logic

```
module comb (a, b, sel, out);  
  input a, b;  
  input sel;  
  output out;  
  reg out;  
  always @ (a or b or sel)  
  begin  
    if (sel) out = a;  
    else out = b;  
  end  
endmodule
```

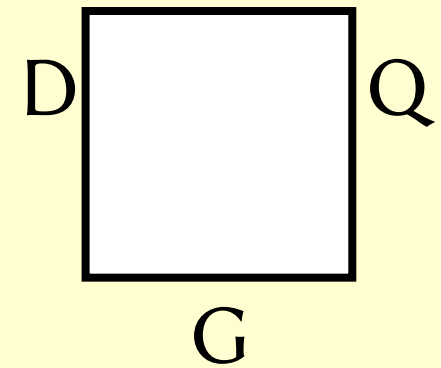
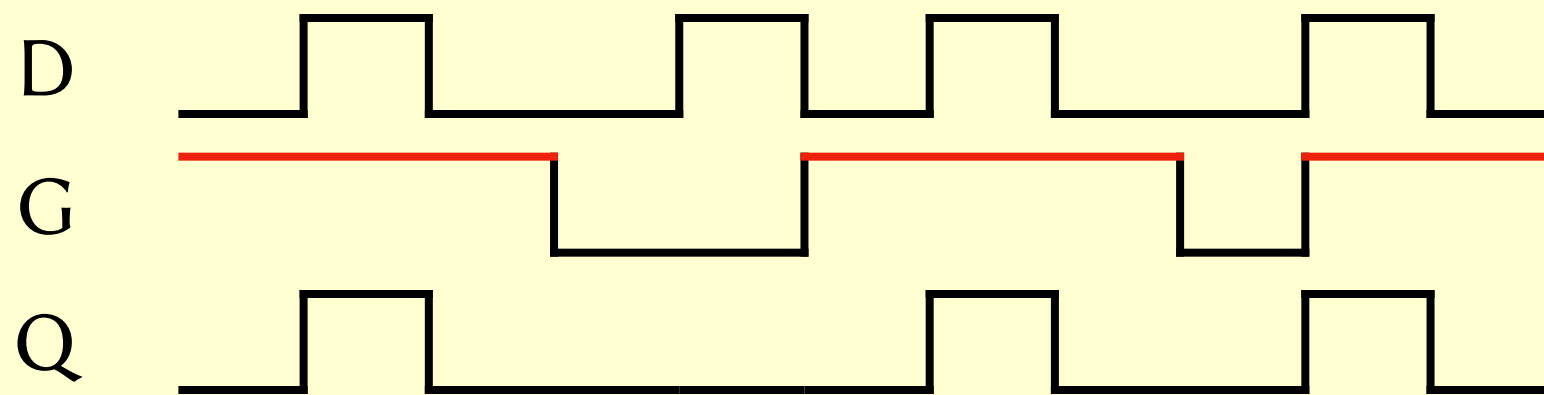


```
module seq (a, b, sel, clk, out);  
  input a, b;  
  input sel, clk;  
  output out;  
  reg out;  
  always @ (posedge clk)  
  begin  
    if (sel) out <= a;  
    else out <= b;  
  end  
endmodule
```

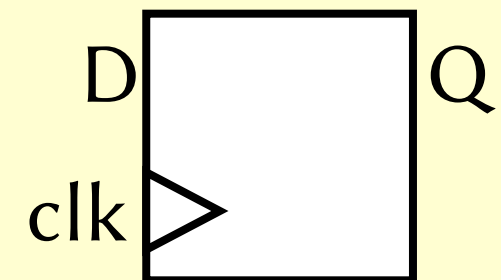
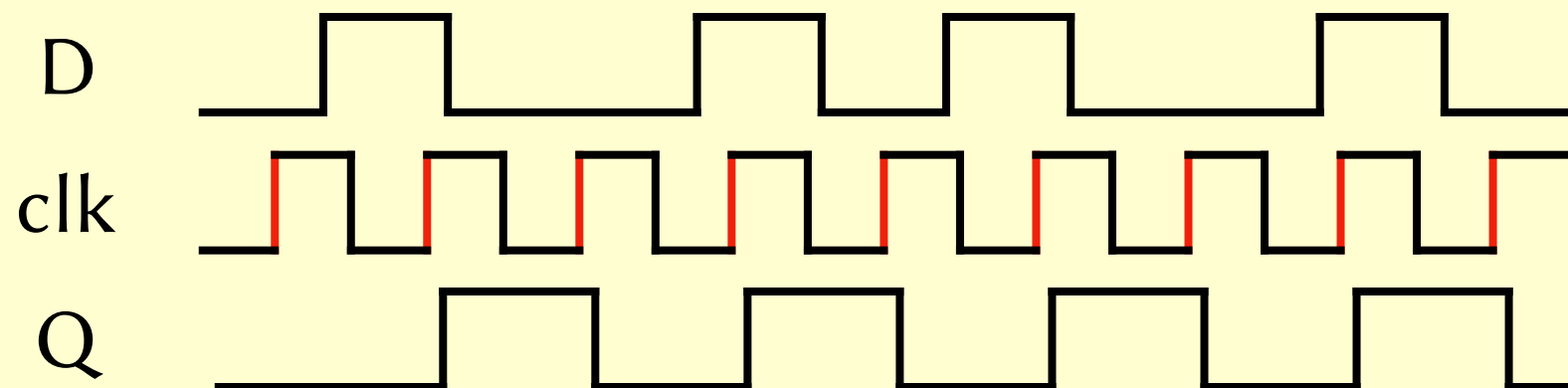


Flip flop vs latch

- Latch:

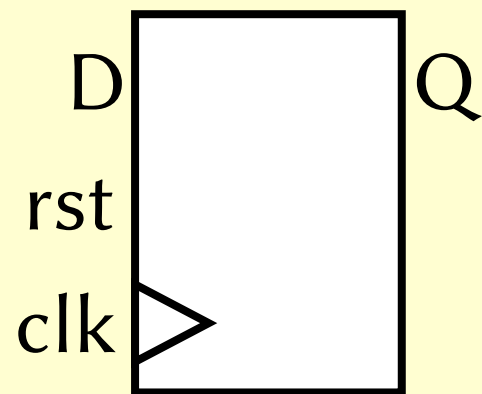


- Flip flop:



Defining a flip flop

```
module dff (d, rst, clk, q);  
  input d, rst, clk;  
  output q;  
  reg q;  
  always @ (posedge clk)  
  begin  
    if (!rst) q <= 1'b0;  
    else q <= d;  
  end  
endmodule
```



Note that you can have multiple **always** blocks with the same sensitivity list. They'll be run in parallel, so watch out for races!

Blocking vs non-blocking assignment

- Two different types of procedural assignments:

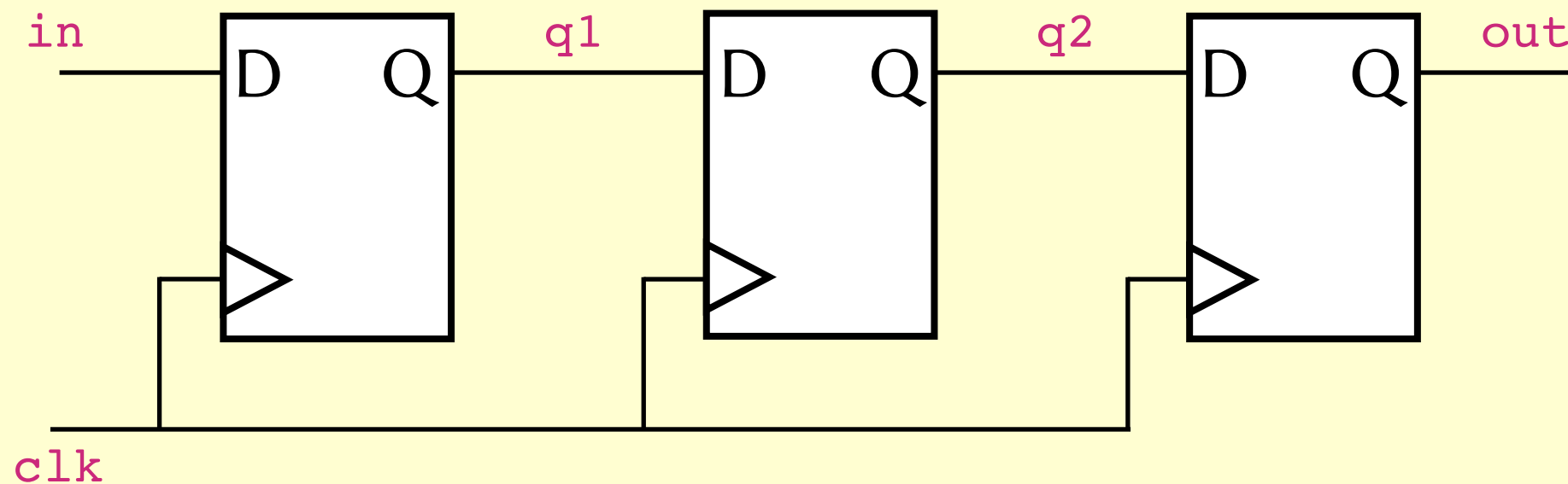
```
always @*  
begin  
    a = b & c;  
    b = a + c;  
end
```

The second statement is blocked from executing until the first statement has finished.

```
always @*  
begin  
    a <= b & c;  
    b <= a + c;  
end
```

All assignments performed in parallel.

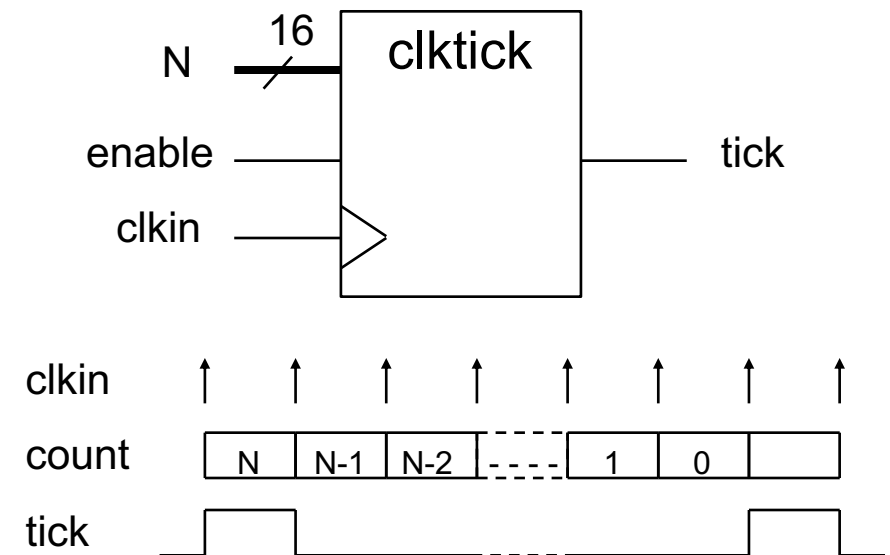
Blocking vs non-blocking assignment



```
module sr_v1 (in, clk, out);  
  input in, clk;  
  output out;  
  reg q1, q2, out;  
  always @ (posedge clk)  
  begin  
    q1 = in;  
    q2 = q1;  
    out = q2;  
  end  
endmodule
```

```
module sr_v2 (in, clk, out);  
  input in, clk;  
  output out;  
  reg q1, q2, out;  
  always @ (posedge clk)  
  begin  
    q1 <= in;  
    q2 <= q1;  
    out <= q2;  
  end  
endmodule
```

A flexible clock tick



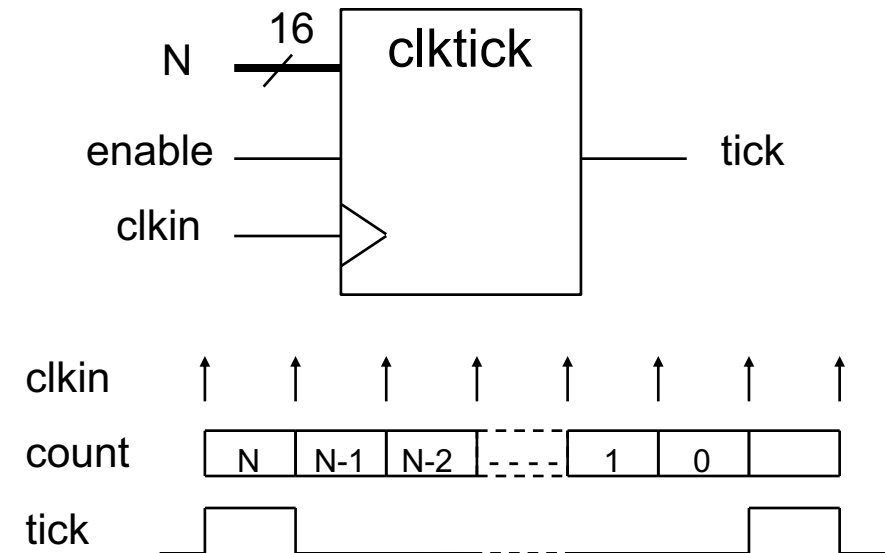
```
module clktick (clkin, enable, N, tick);  
    parameter N_BIT = 16;  
    input clkin; // clock input  
    input enable; // low => clkin ignored  
    input[N_BIT-1:0] N; // pulse every N+1 cycles  
    output tick;
```


A flexible clock tick

```

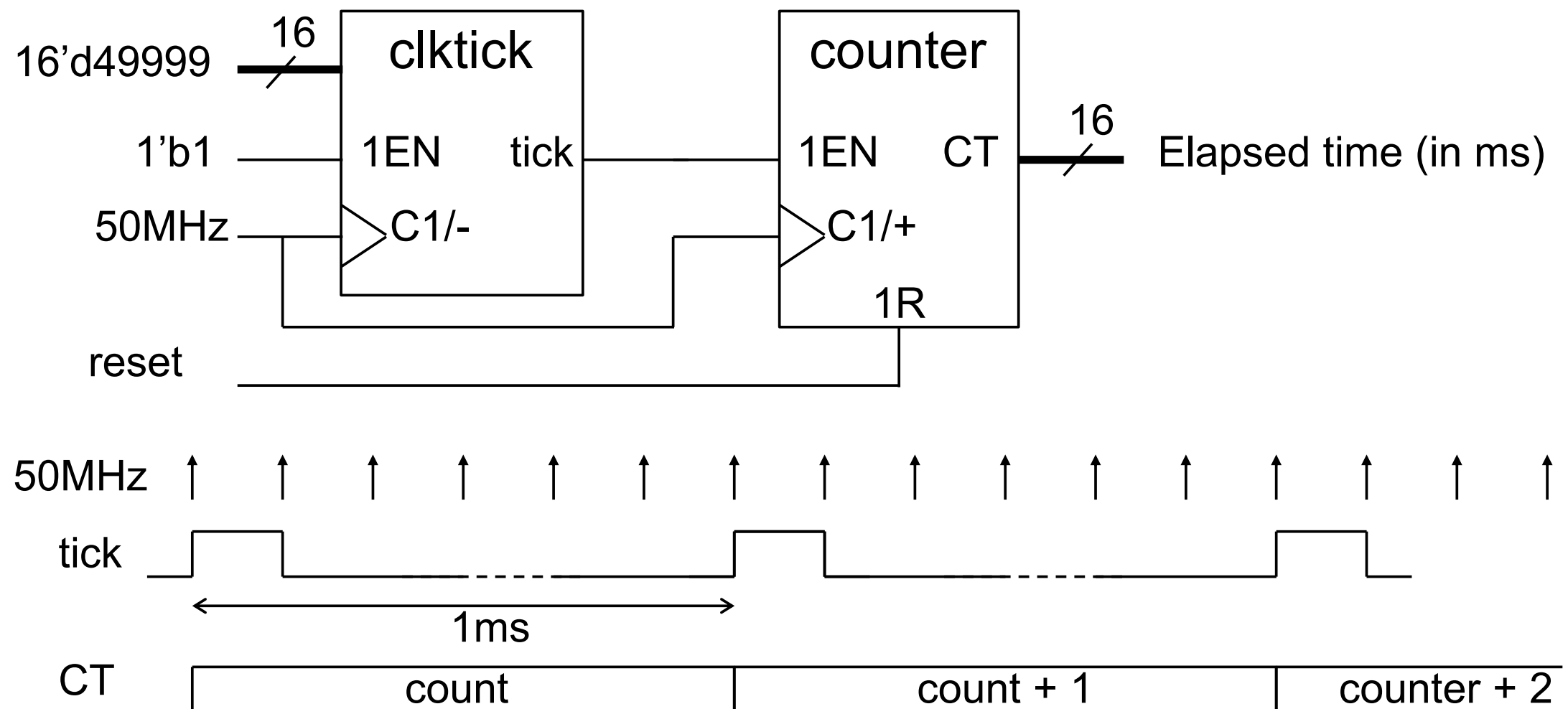
module clktick (clkin, enable, N, tick);
  parameter N_BIT = 16;
  input clkin; // clock input
  input enable; // low => clkin ignored
  input[N_BIT-1:0] N; // pulse every N+1 cycles
  output tick;
  reg[N_BIT-1:0] count;
  reg tick;
  initial tick = 1'b0;
  always @ (posedge clkin)
    if (enable == 1'b1)
      if (count == 0) begin
        tick <= 1'b1;
        count <= N;
      end else begin
        tick <= 1'b0;
        count <= count - 1'b1;
      end
endmodule

```



Cascading counters

- By connecting our clock tick module in series with a counter module, we can count milliseconds:

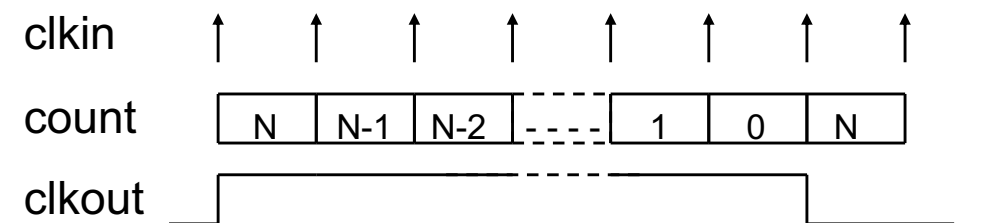
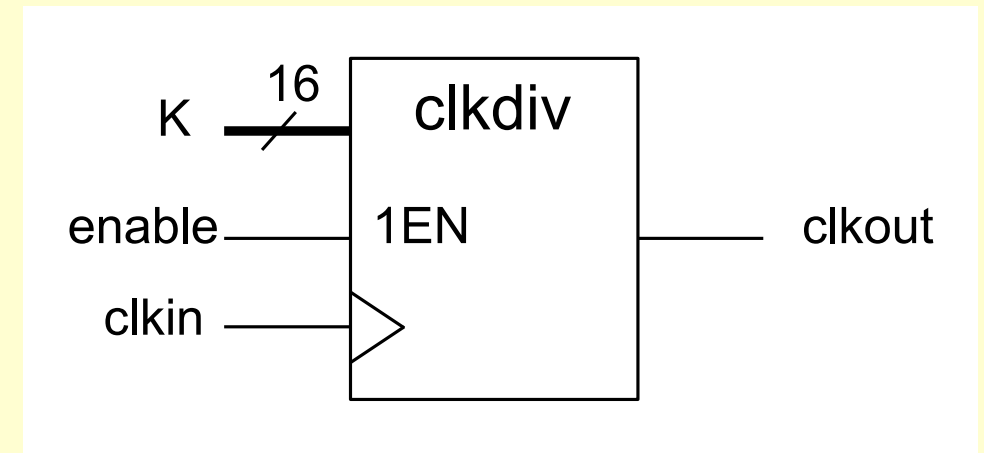


A clock divider

```

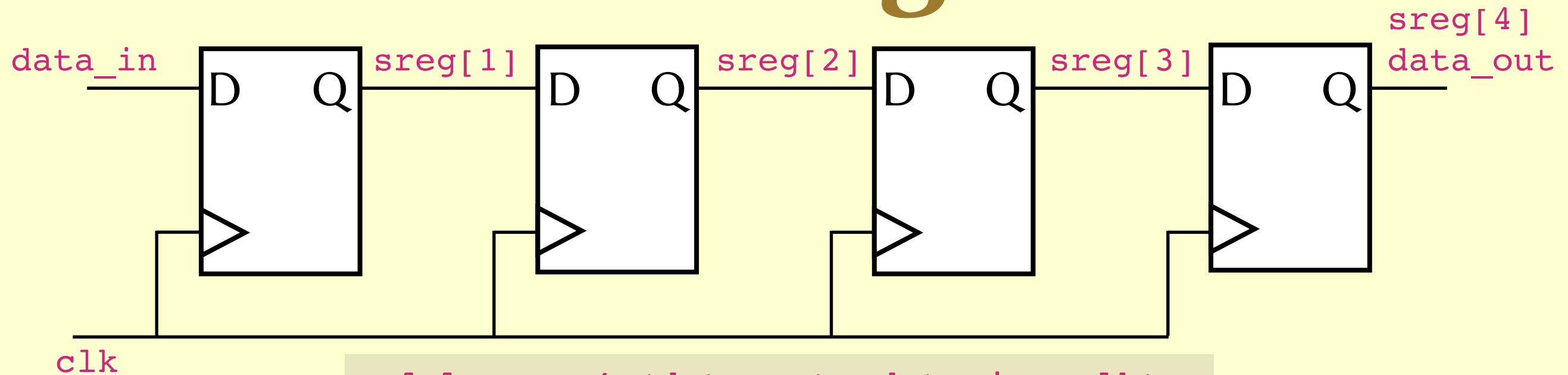
module clkdiv (clkkin, enable, K, clkout);
  parameter K_BIT = 16;
  input clkkin; // clock input
  input enable; // low => clkkin ignored
  input [K_BIT-1:0] K;
  output tick; //  $F_{out} = F_{in} / (2^{*(K+1)})$ 
  reg [K_BIT-1:0] count;
  reg clkout;
  initial clkout = 1'b0;
  always @ (posedge clkkin)
    if (enable == 1'b1)
      if (count == 0) begin
        clkout <= ~clkout; // toggle output
        count <= K;
      end else
        count <= count - 1'b1;
endmodule

```



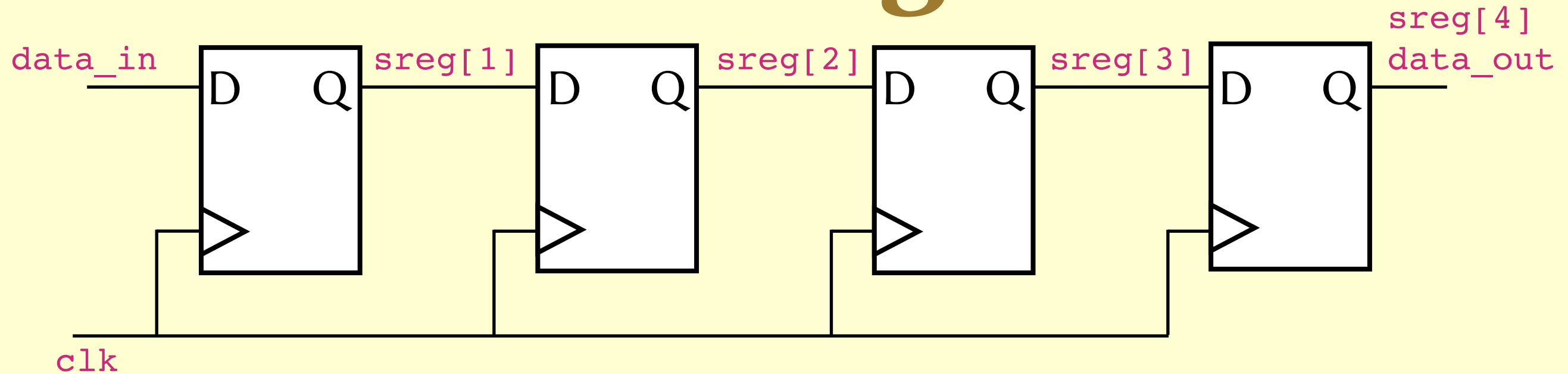
Shift registers

A shift register



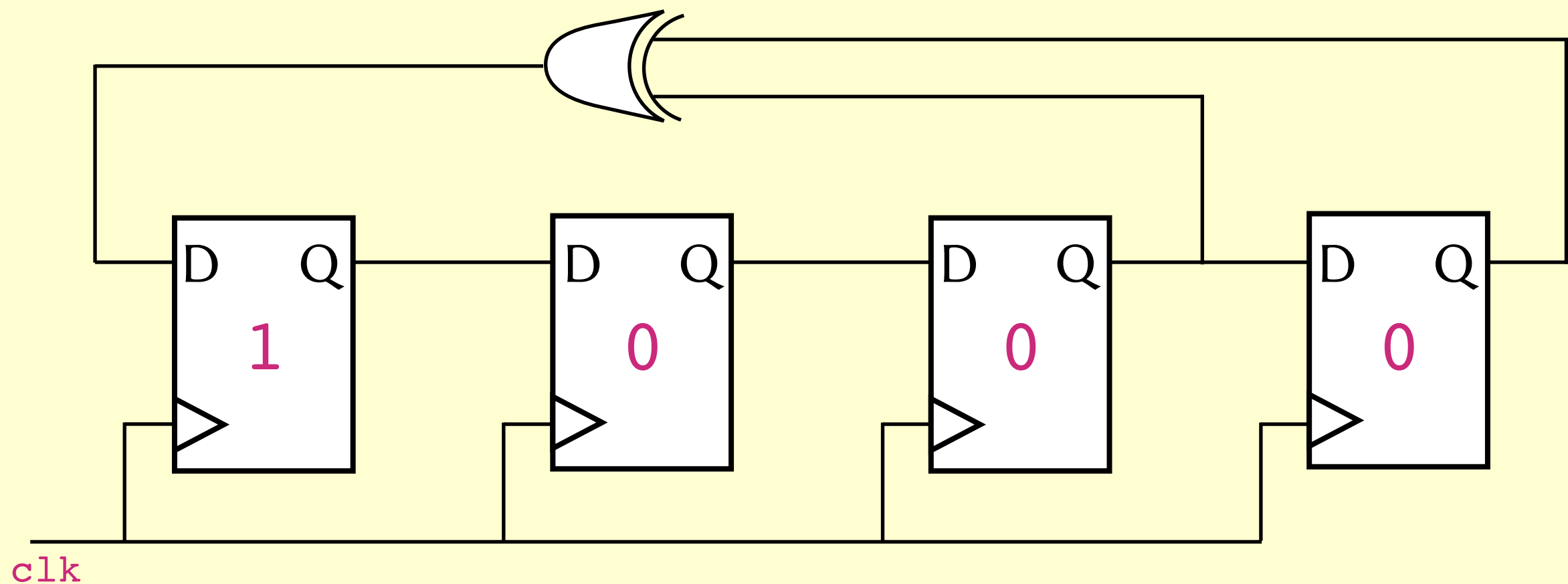
```
module sreg4 (data_out, data_in, clk);  
    output data_out;  
    input data_in, clk;  
    reg[4:1] sreg;  
    initial sreg = 4'b0;  
    always @ (posedge clk) begin  
        sreg[4] <= sreg[3];  
        sreg[3] <= sreg[2];  
        sreg[2] <= sreg[1];  
        sreg[1] <= data_in;  
    end  
    assign data_out = sreg[4];  
endmodule
```

A shift register

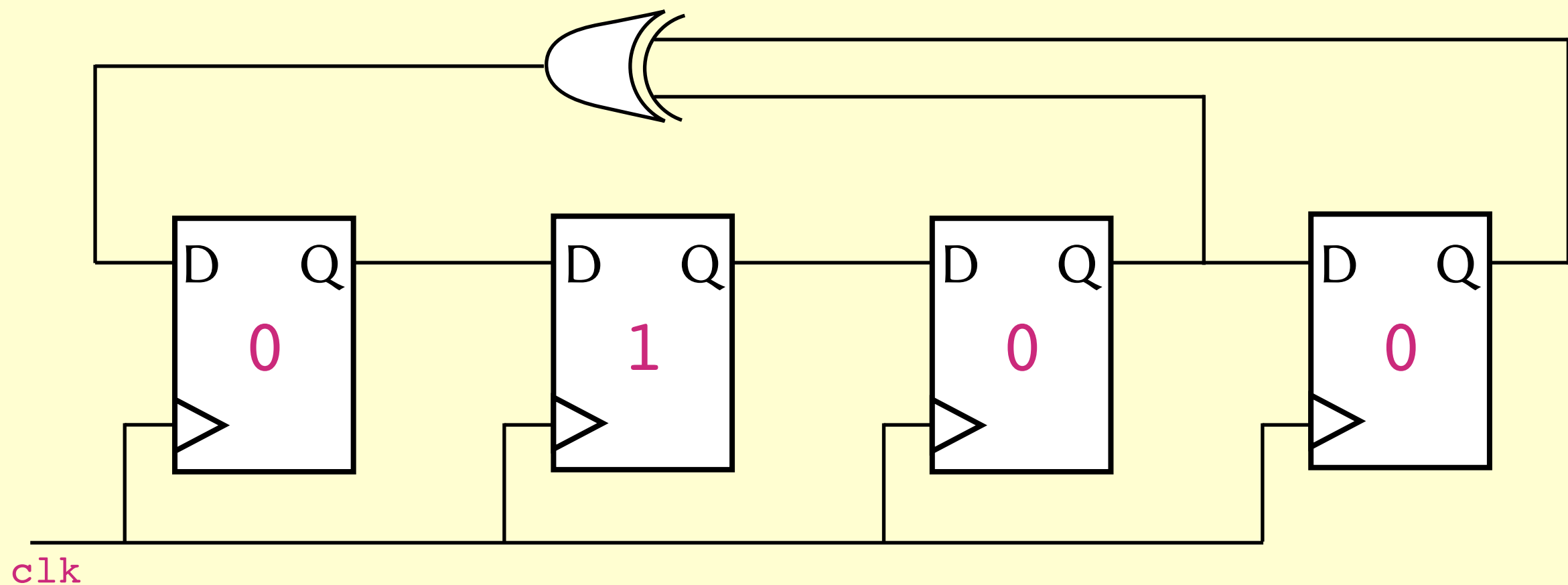


```
module sreg4 (data_out, data_in, clk);  
    output data_out;  
    input data_in, clk;  
    reg[4:1] sreg;  
    initial sreg = 4'b0;  
    always @ (posedge clk)  
        sreg <= {sreg[3:1], data_in};  
    assign data_out = sreg[4];  
endmodule
```

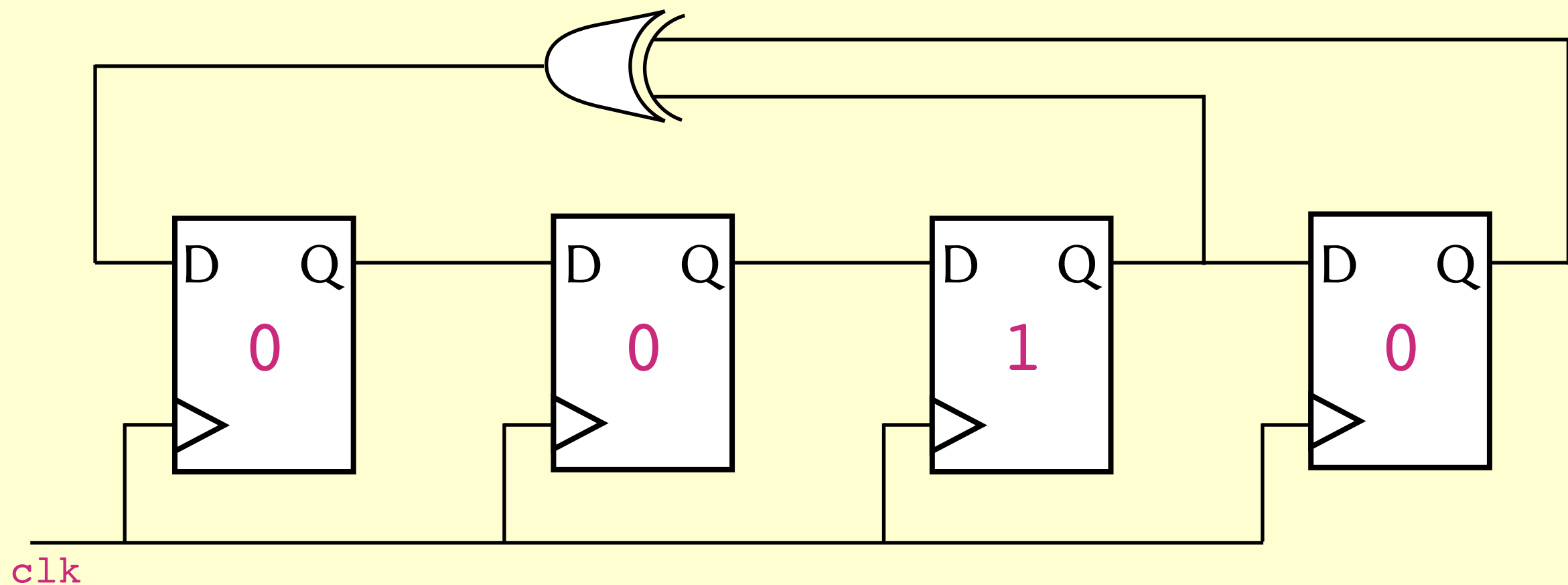
Linear feedback shift register



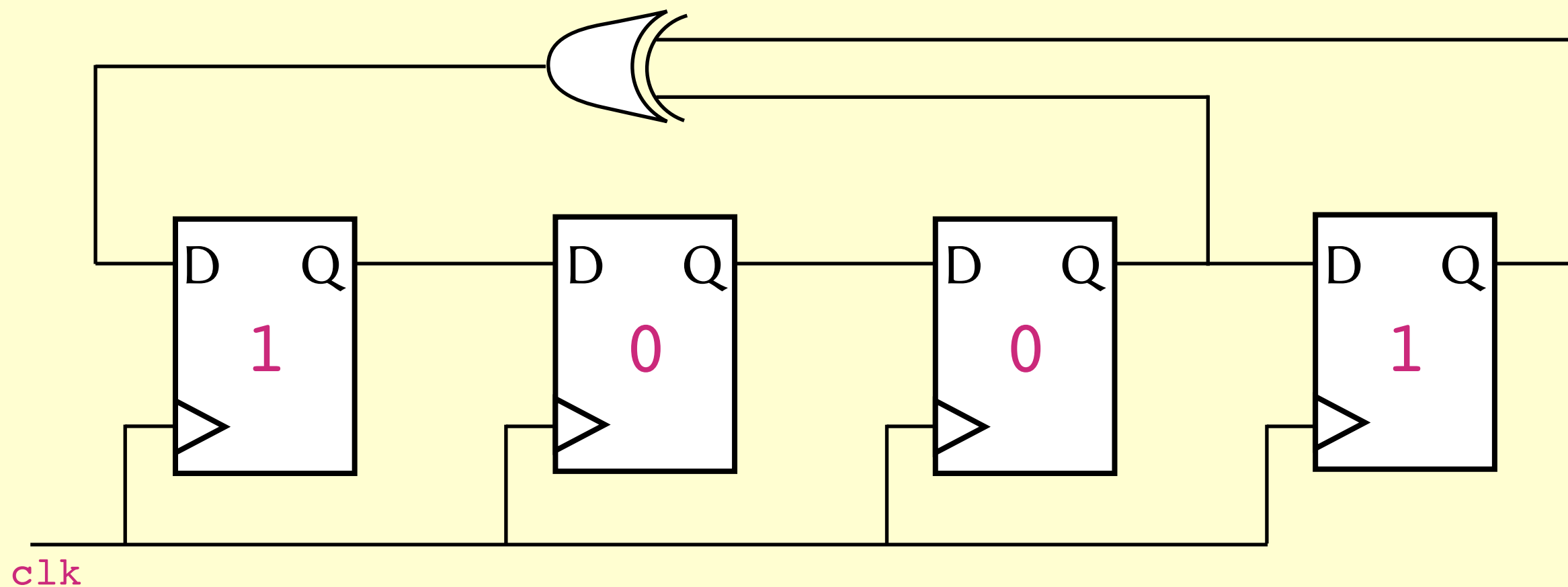
Linear feedback shift register



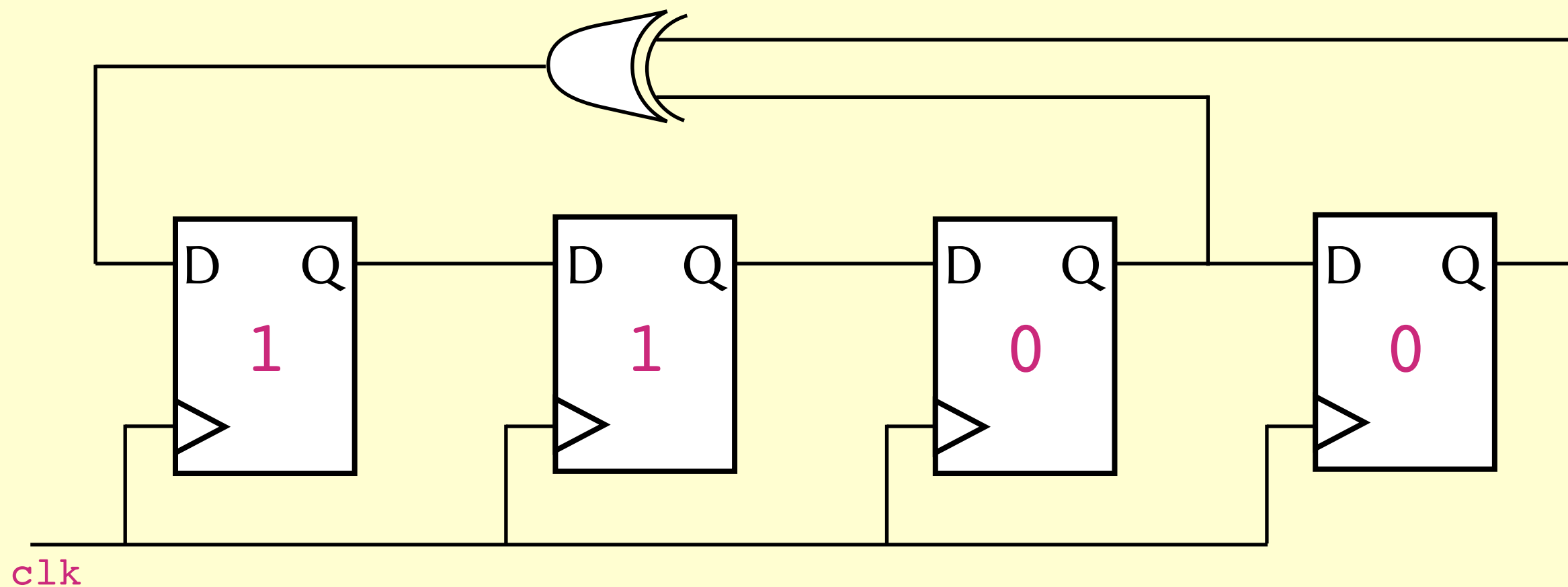
Linear feedback shift register



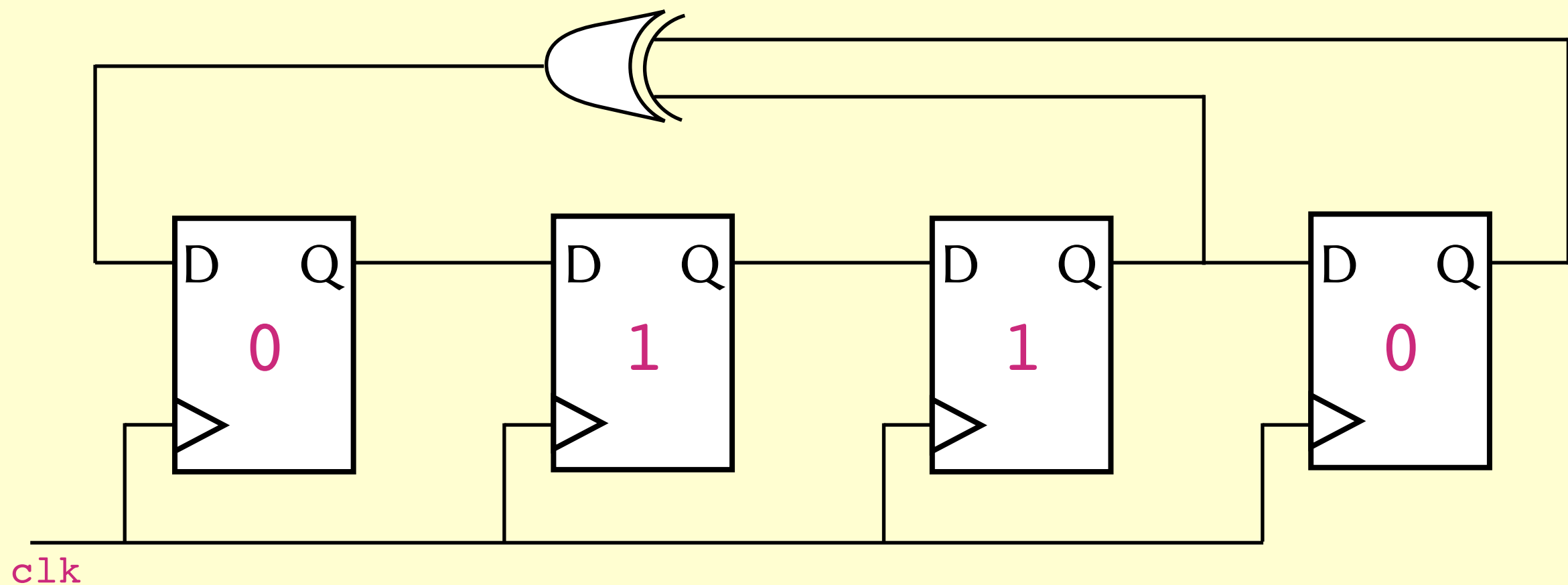
Linear feedback shift register



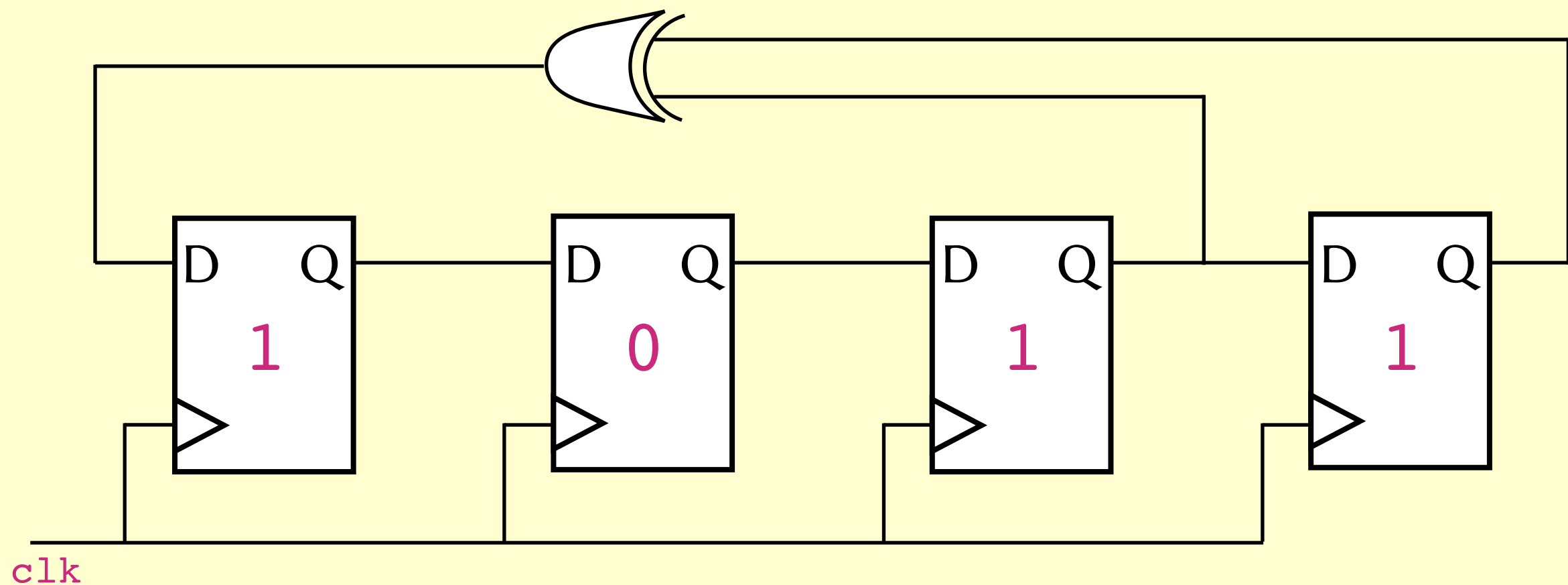
Linear feedback shift register



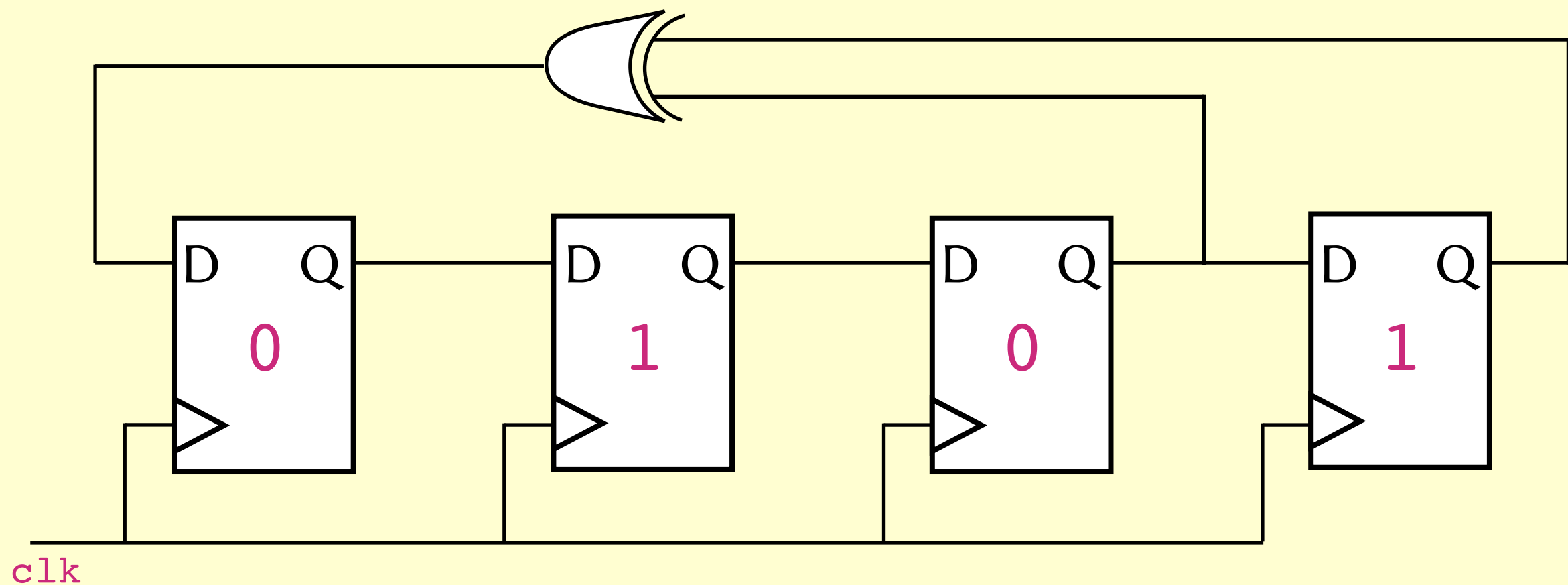
Linear feedback shift register



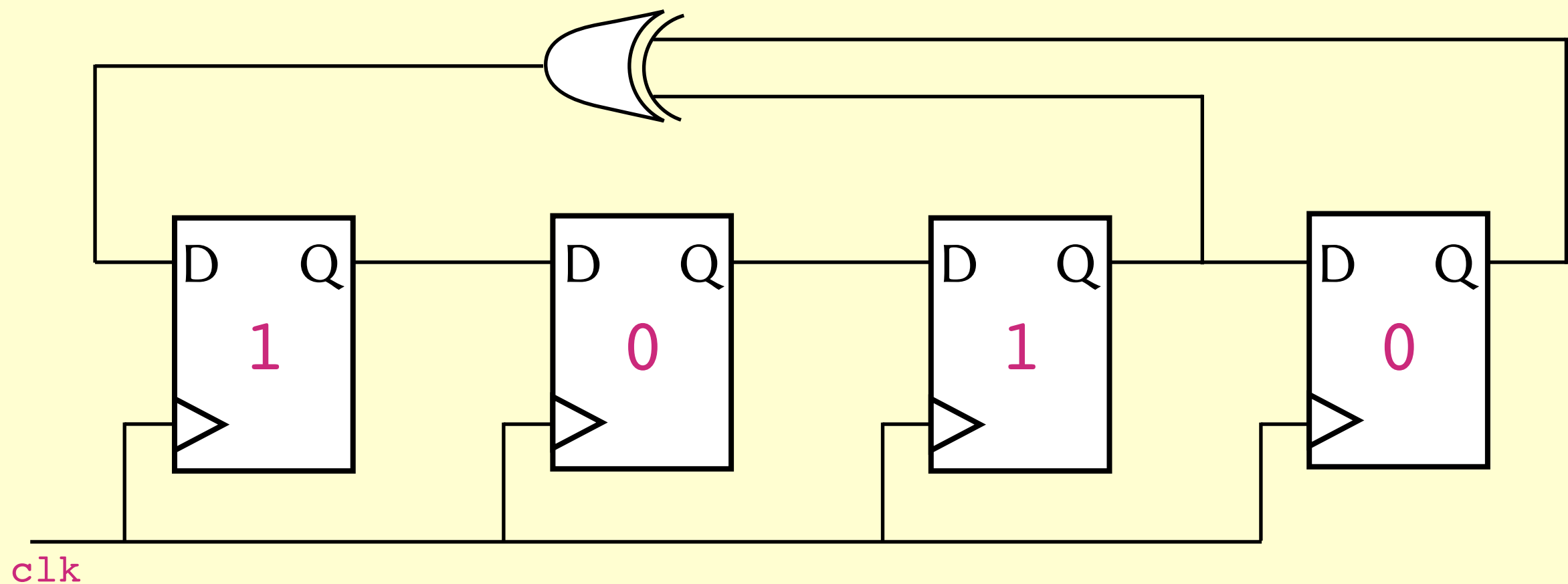
Linear feedback shift register



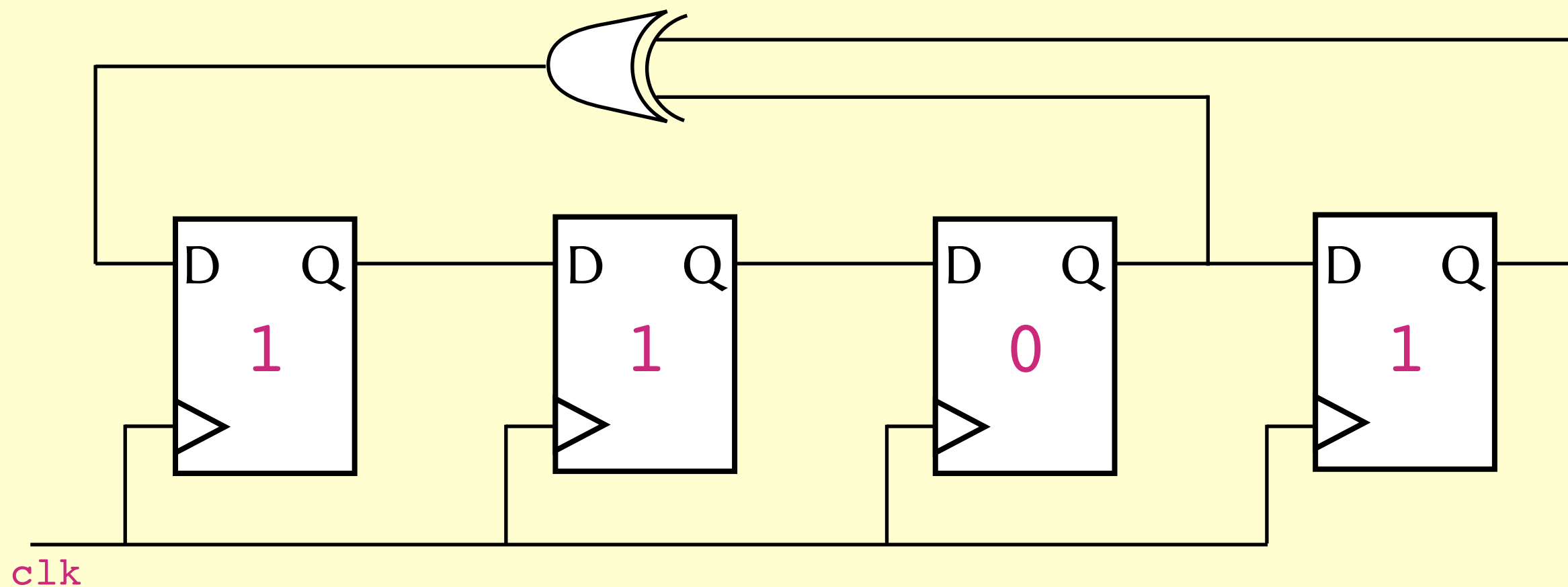
Linear feedback shift register



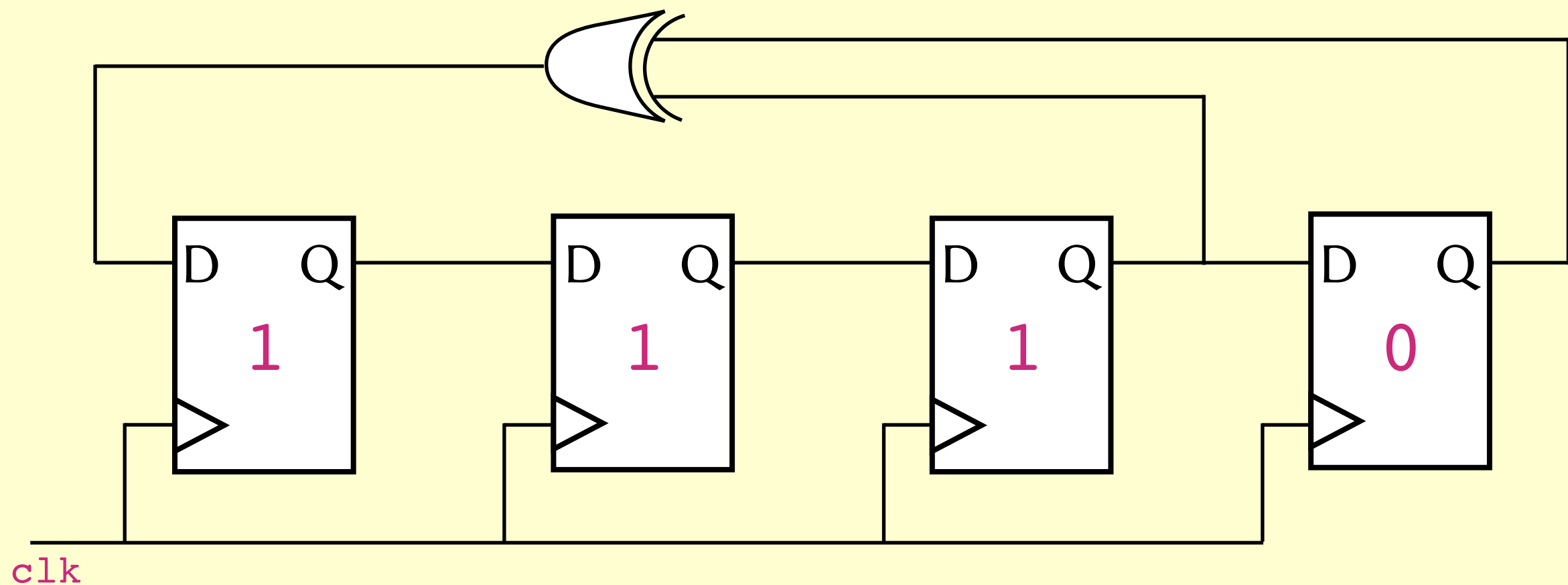
Linear feedback shift register



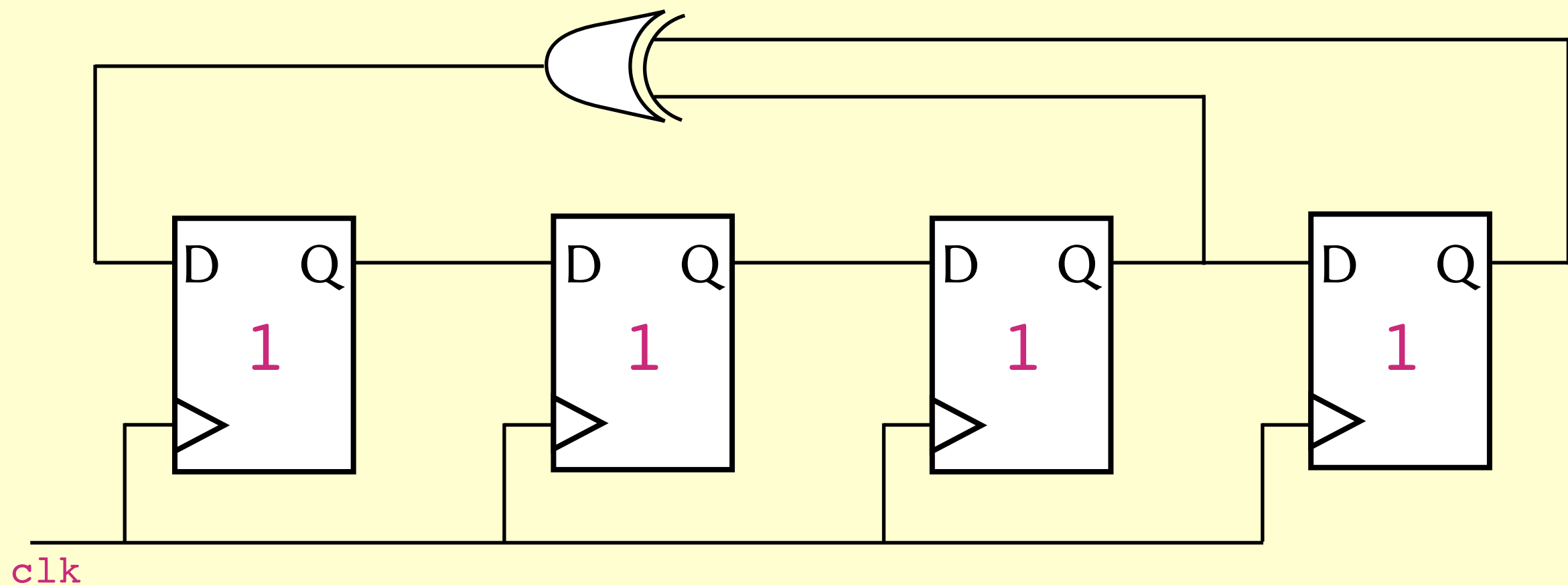
Linear feedback shift register



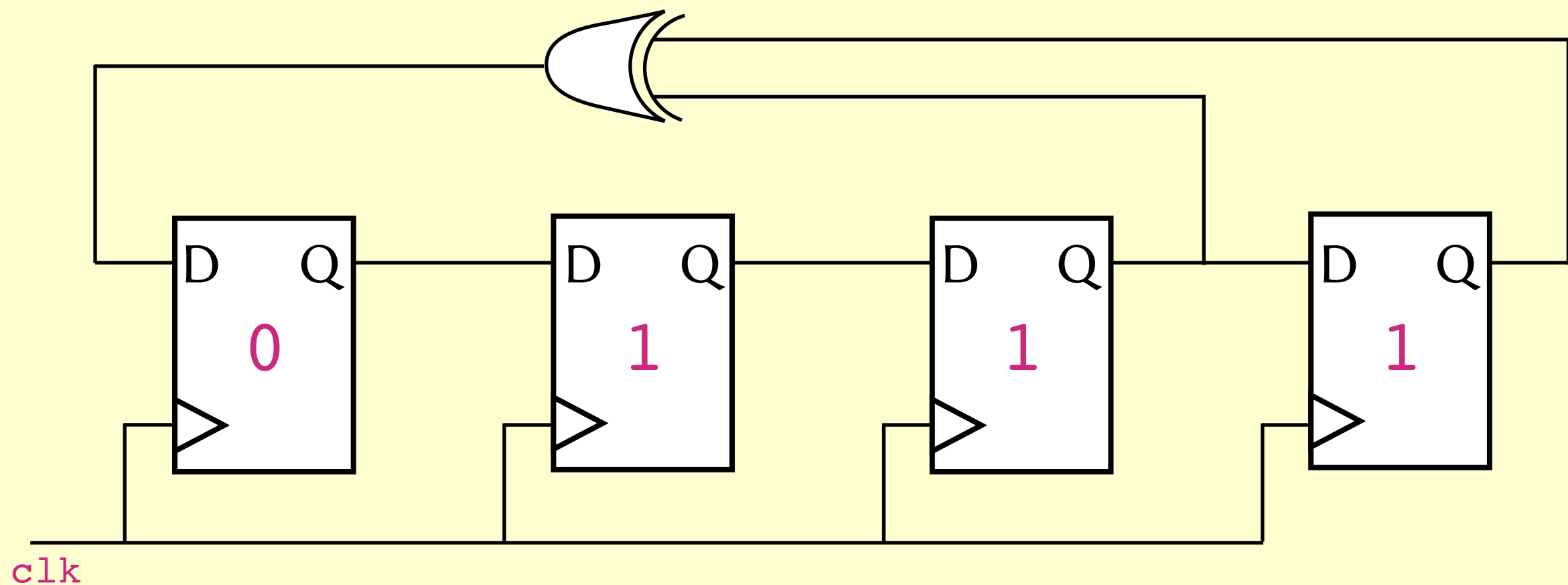
Linear feedback shift register



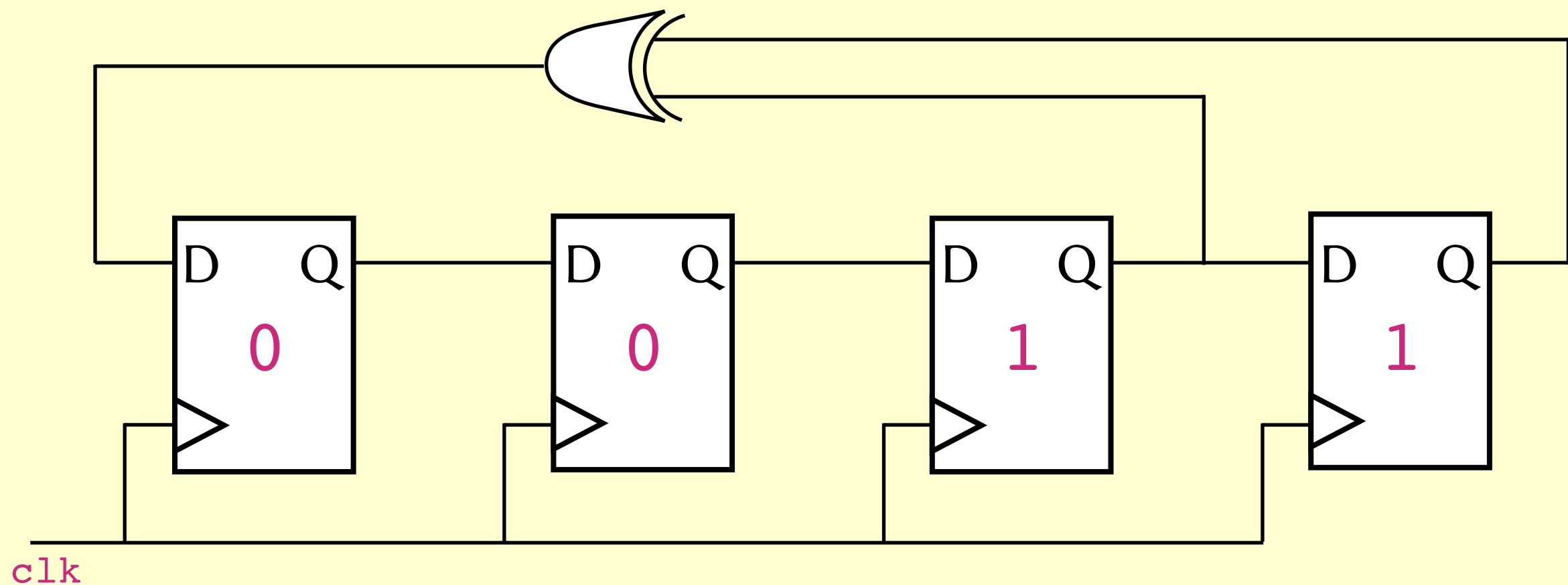
Linear feedback shift register



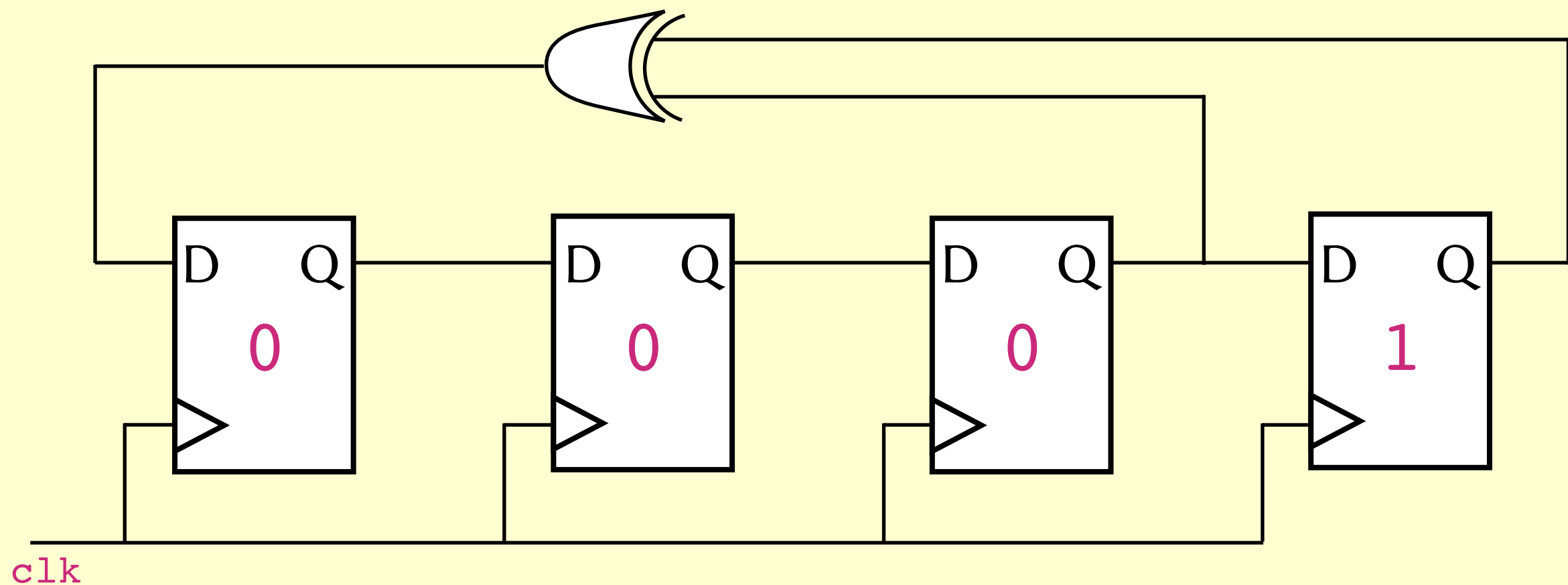
Linear feedback shift register



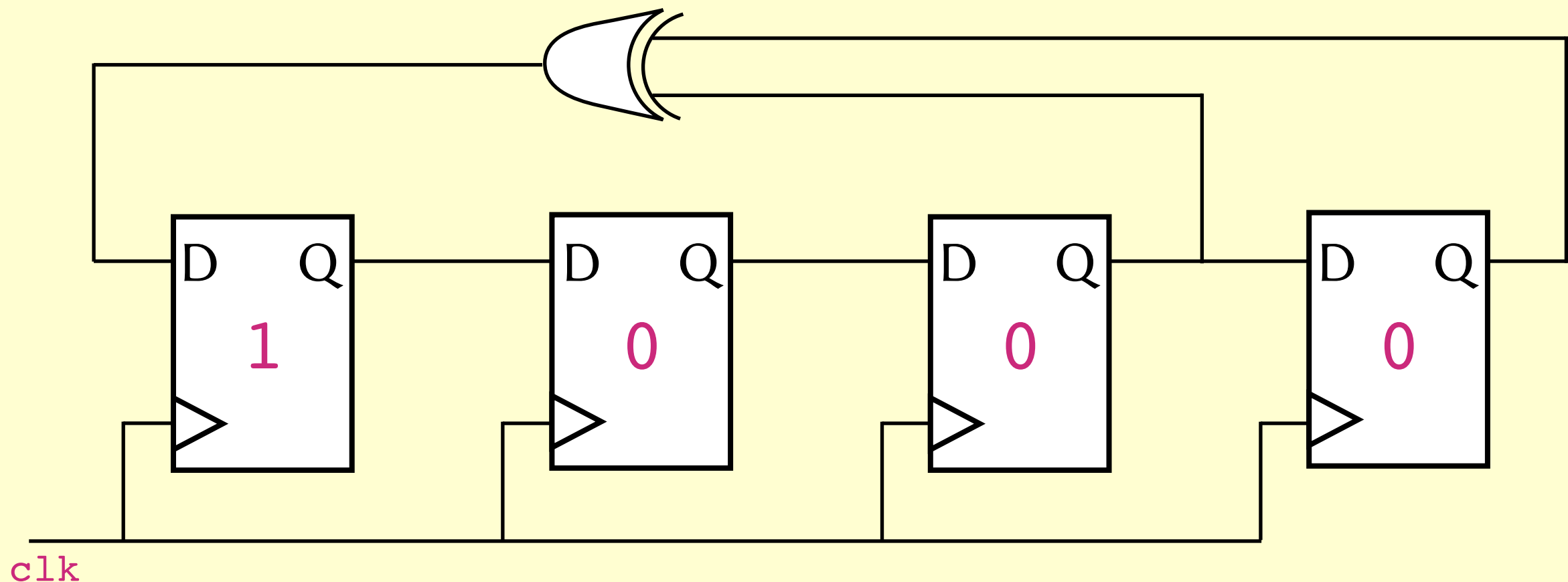
Linear feedback shift register



Linear feedback shift register



Linear feedback shift register



- This LFSR corresponds to the polynomial $1 + x^3 + x^4$. Other LFSRs are possible, with different polynomials.

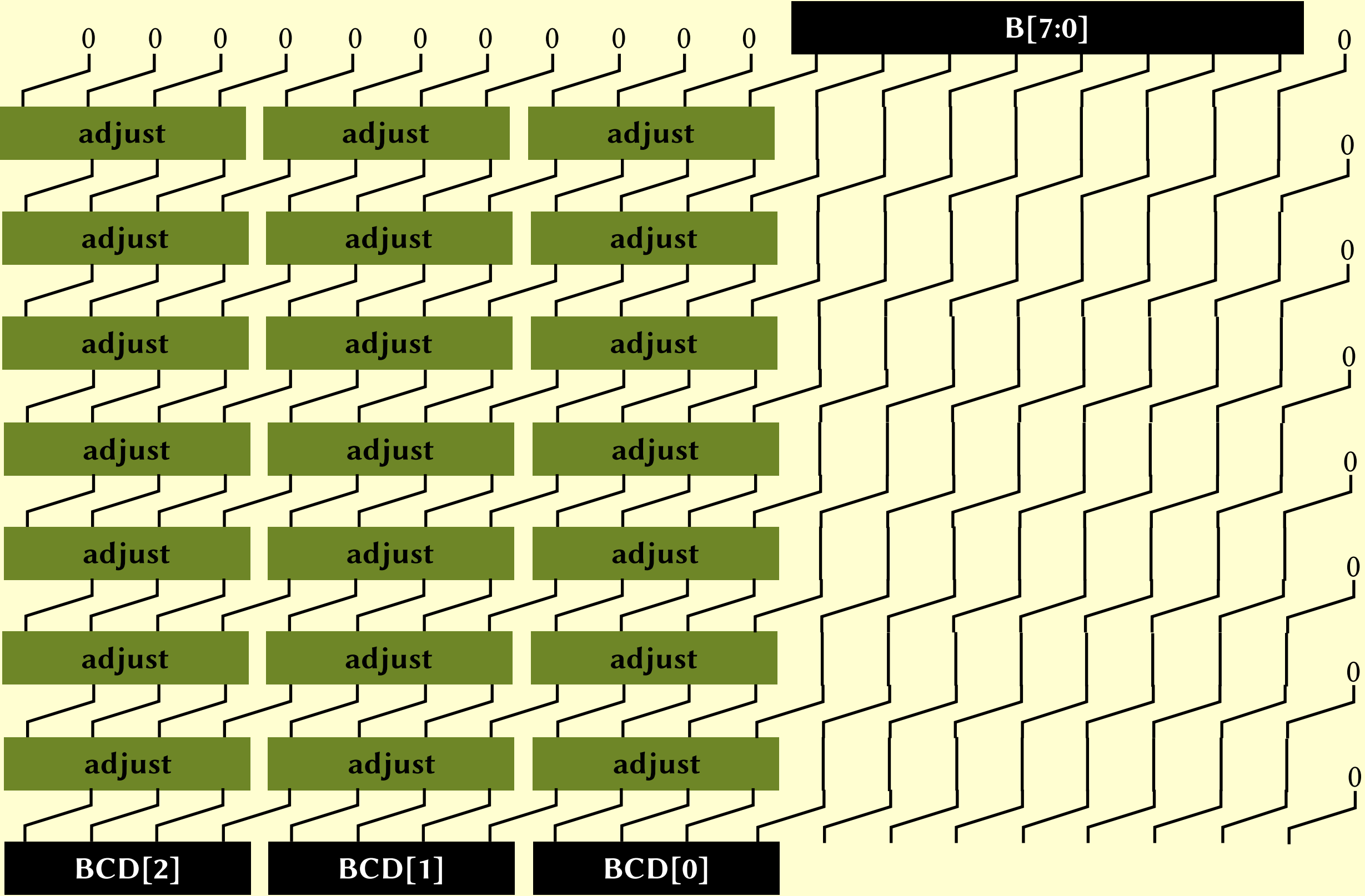
Linear feedback shift register

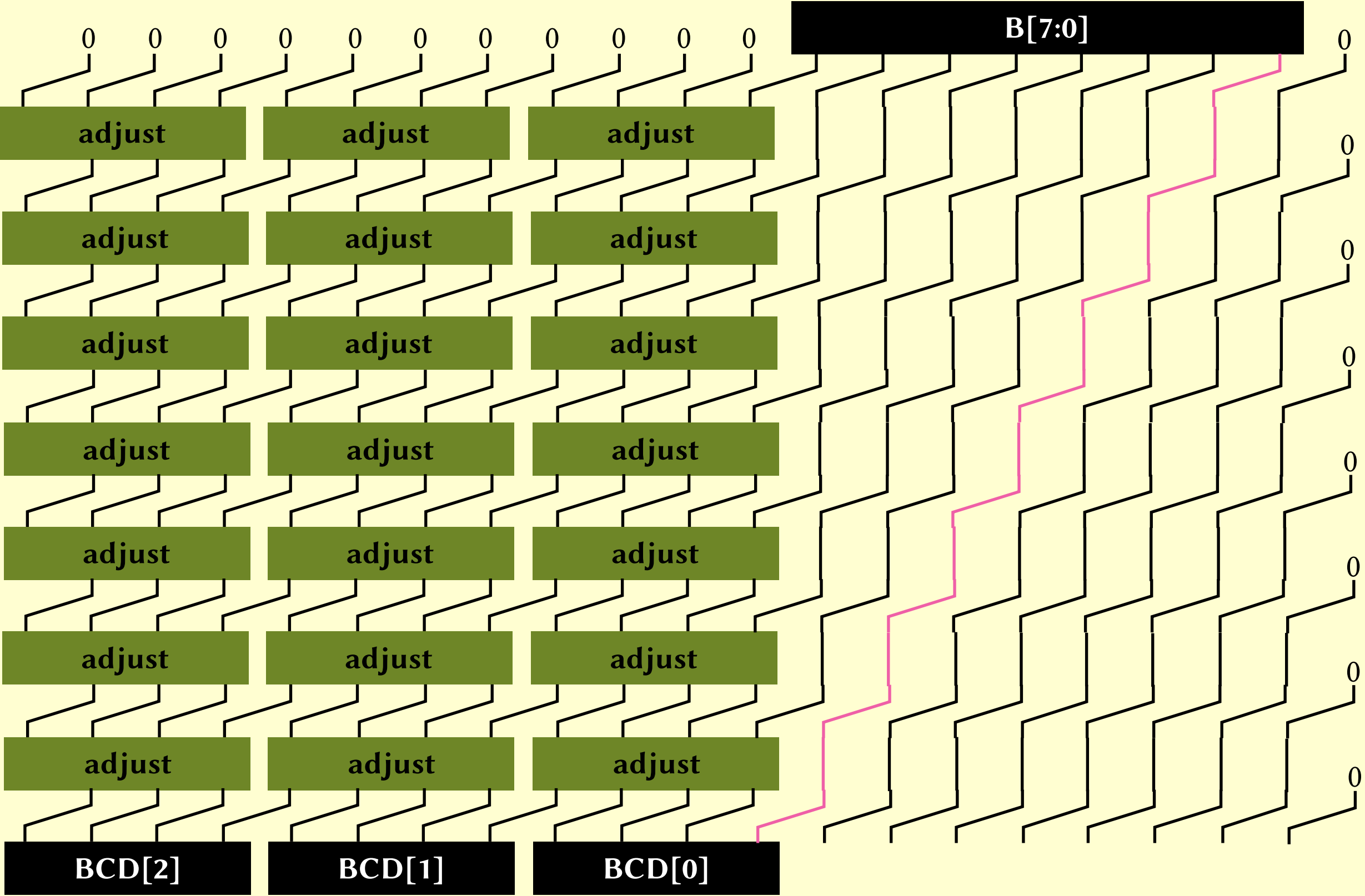
```
module lfsr4 (data_out, clk);  
    output[4:1] data_out;  
    input clk;  
    reg[4:1] sreg;  
    initial sreg = 4'b1;  
    always @ (posedge clk)  
        sreg <= {sreg[3:1], sreg[4] ^ sreg[3]};  
    assign data_out = sreg;  
endmodule
```

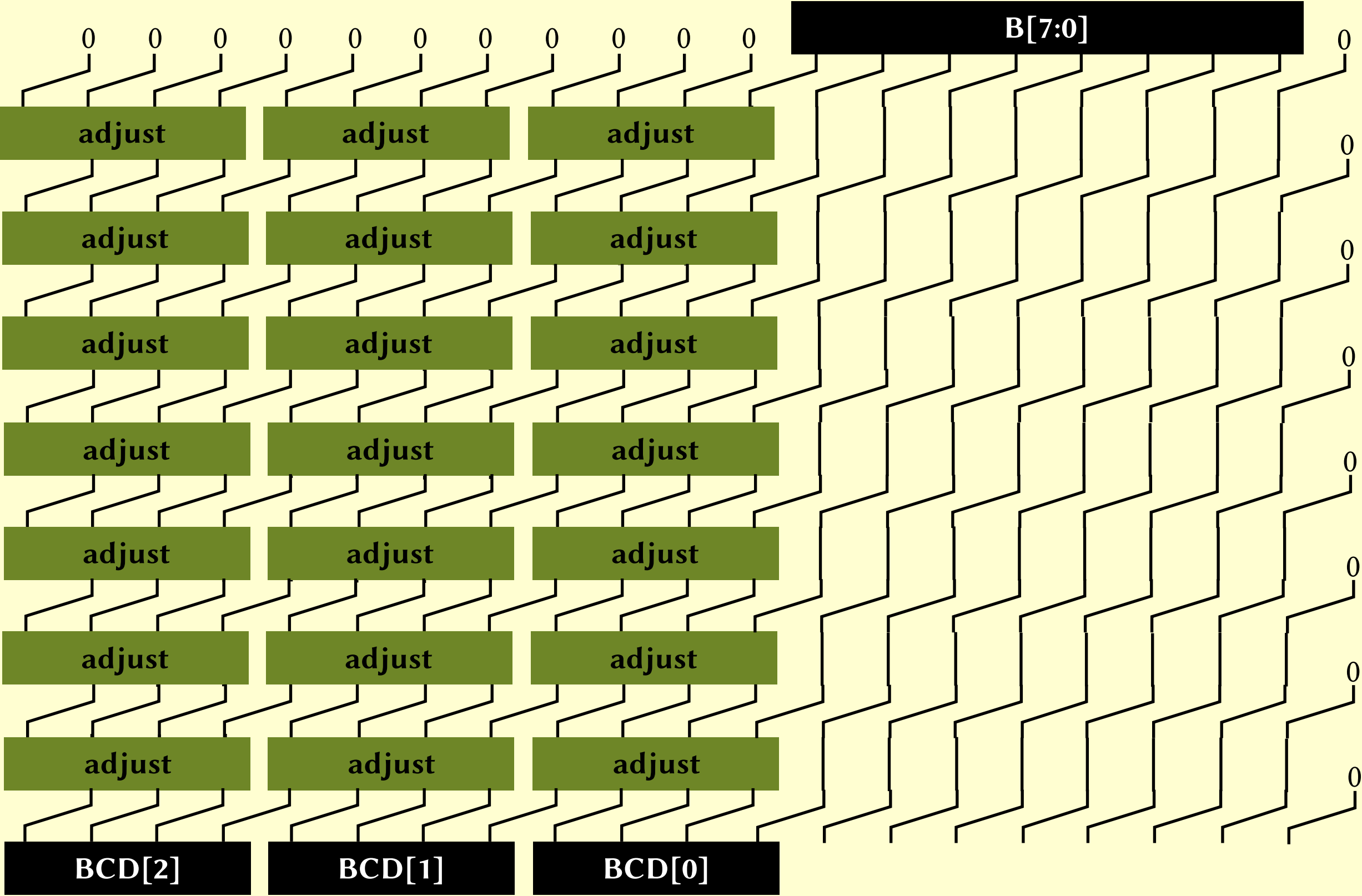
Binary to BCD

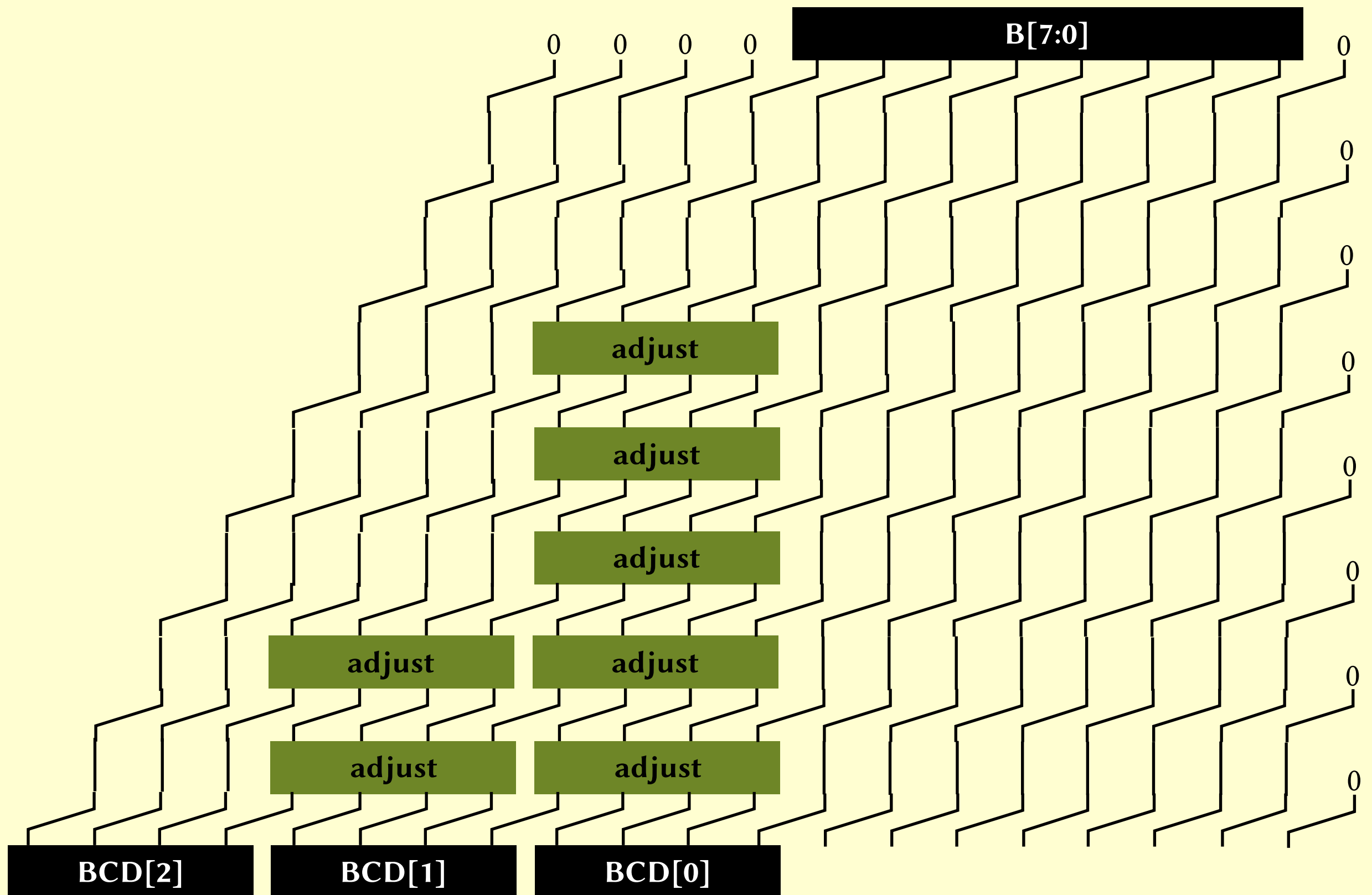
Binary to BCD

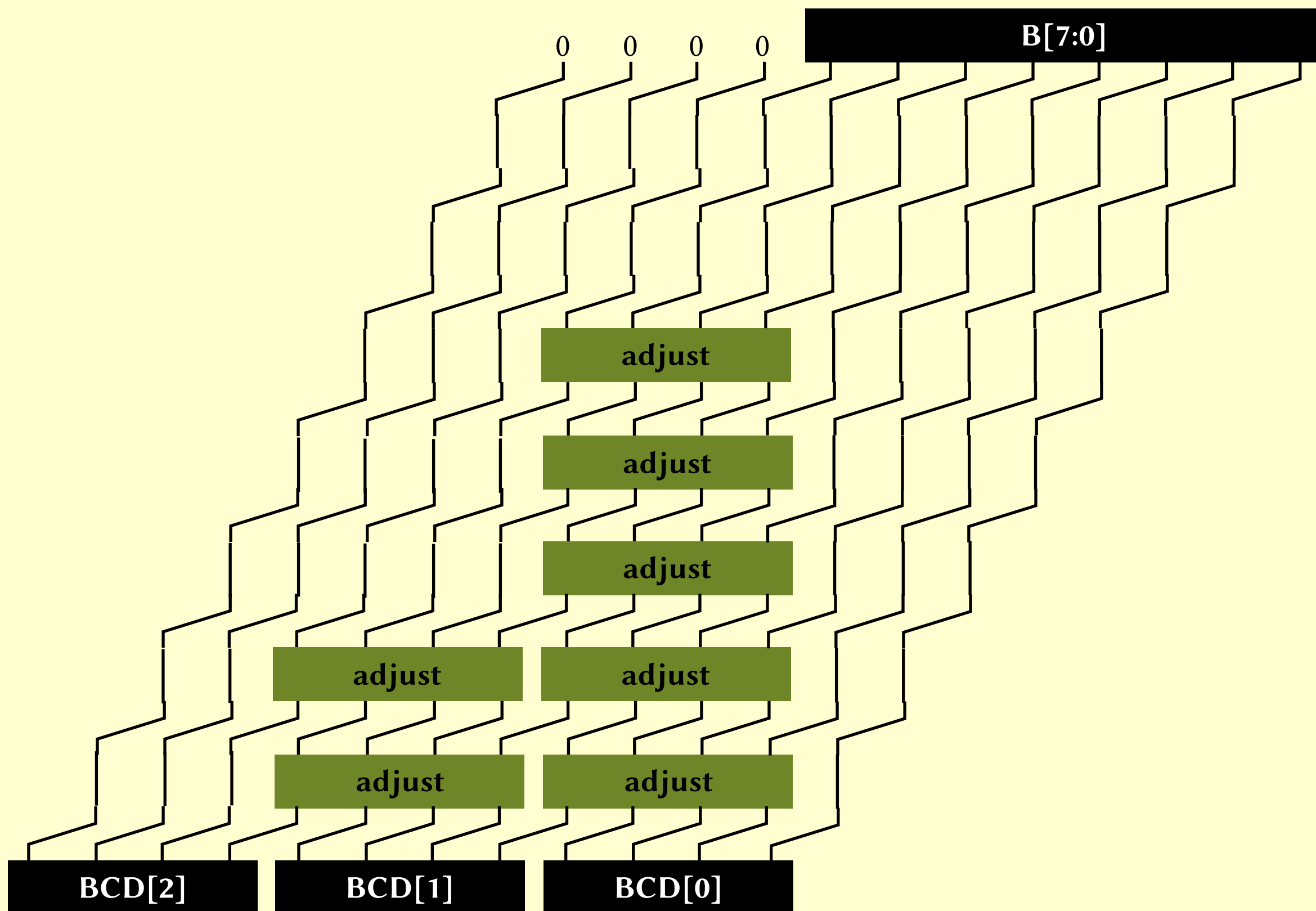
binary-coded decimal (BCD)			8-bit binary	
hundreds	tens	units		
			0111	1100
shift:		0	1111	100
shift:		01	1111	00
shift:		011	1110	0
shift:		0111	1100	
add 3:		1010	1100	
shift:	1	0101	100	
add 3:	1	1000	100	
shift:	11	0001	00	
shift:	110	0010	0	
add 3:	1001	0010	0	
shift:	1	0010	0100	

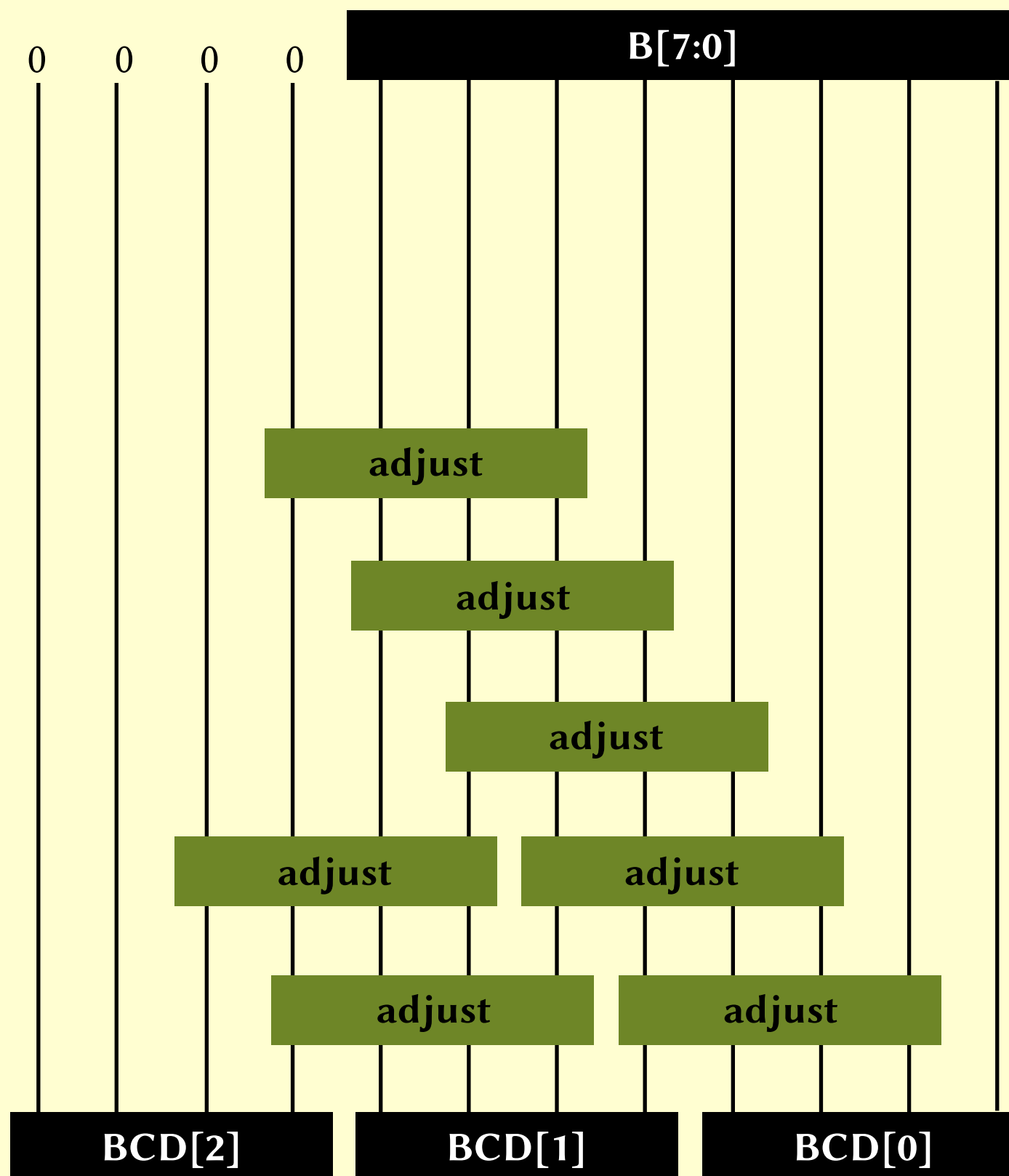












Binary to BCD

```
module bin2bcd8 (B, BCD_0, BCD_1, BCD_2);  
  
    input [7:0] B;      // binary input number  
    output [3:0] BCD_0, BCD_1, BCD_2;  // BCD digit LSD to MSD  
  
    wire [3:0] w1,w2,w3,w4,w5,w6,w7;  
    wire [3:0] a1,a2,a3,a4,a5,a6,a7;  
  
    // Instantiate a tree of add3-if-greater than or equal to 5 cells  
    // ... input is w_n, and output is a_n  
    add3_ge5 A1 (w1,a1);  
    add3_ge5 A2 (w2,a2);  
    add3_ge5 A3 (w3,a3);  
    add3_ge5 A4 (w4,a4);  
    add3_ge5 A5 (w5,a5);  
    add3_ge5 A6 (w6,a6);  
    add3_ge5 A7 (w7,a7);  
  
    // wire the tree of add3 modules together  
    assign w1 = {1'b0, B[7:5]};      // wn is the input port to module An  
    assign w2 = {a1[2:0], B[4]};  
    assign w3 = {a2[2:0], B[3]};  
    assign w4 = {1'b0, a1[3], a2[3], a3[3]};  
    assign w5 = {a3[2:0], B[2]};  
    assign w6 = {a4[2:0], a5[3]};  
    assign w7 = {a5[2:0], B[1]};  
  
    // connect up to four BCD digit outputs  
    assign BCD_0 = {a7[2:0], B[0]};  
    assign BCD_1 = {a6[2:0], a7[3]};  
    assign BCD_2 = {2'b0, a4[3], a6[3]};  
  
endmodule
```


This lecture

- Arithmetic and logical operators in Verilog
- Clocked circuits
- Shift registers
- Converting from binary to binary-coded decimal (BCD)