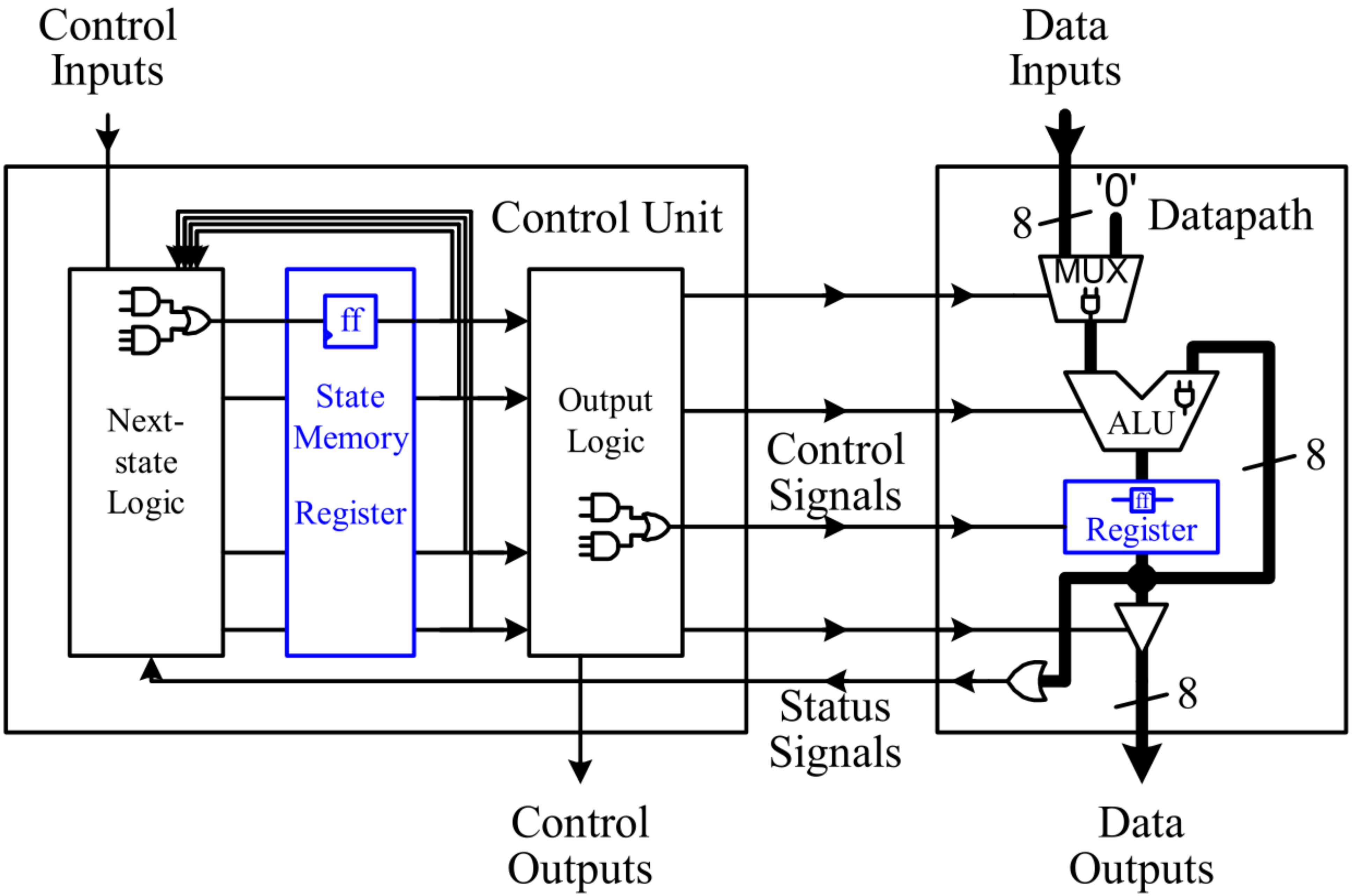
Digital Logic Design

Lecture 6

Sequential logic blocks

Standard sequential components



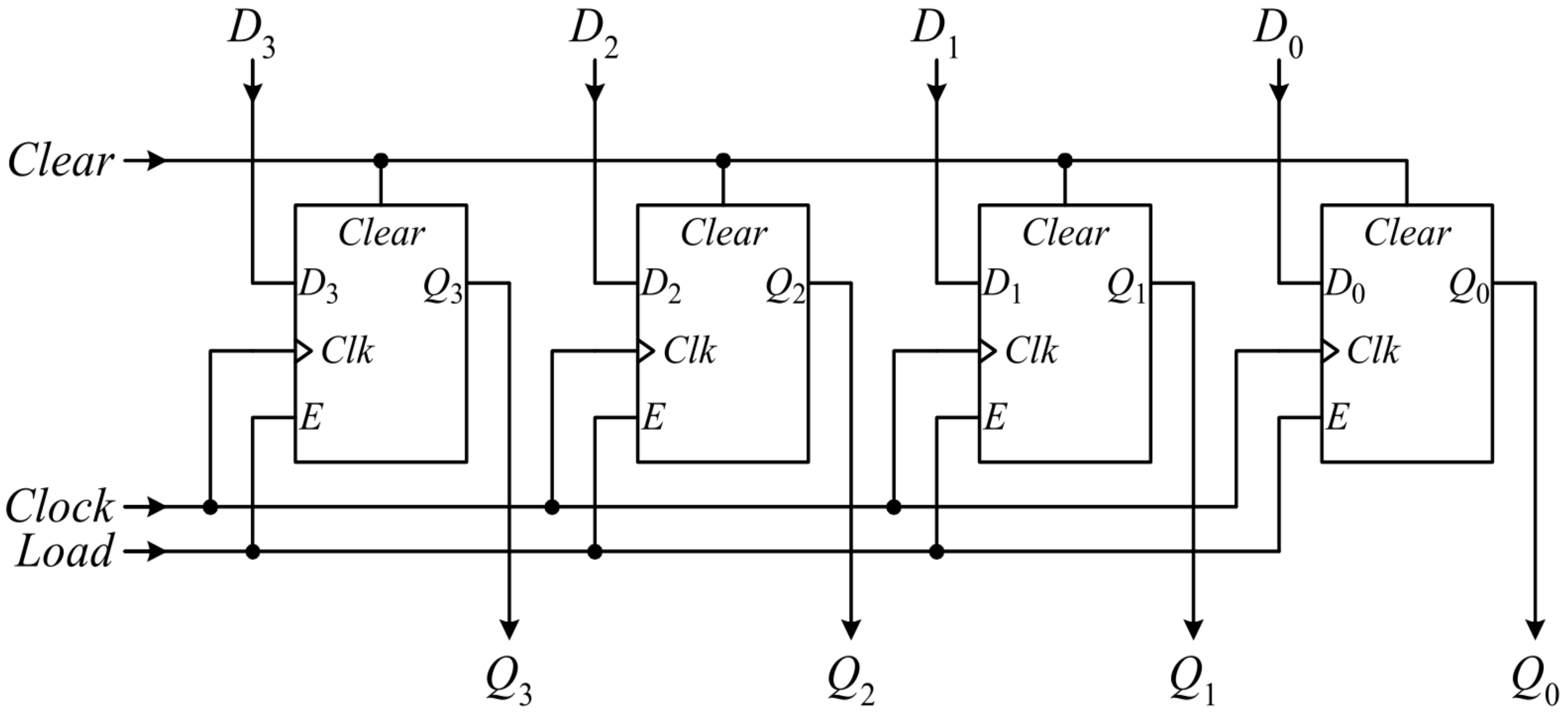
Registers

A **register** is a circuit with two or more D flip-flops connected together in such a way that they all work exactly the same way and are synchronized by the same clock and enable signals.

Each flip-flop in the register is used to store a different bit of the data.

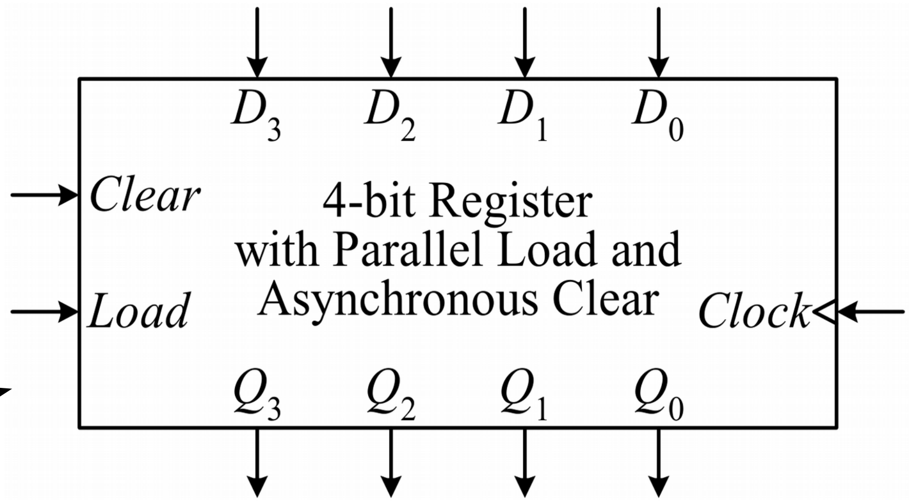
For example, if we want to store a byte of data, we need to combine eight flip-flops together and have them work together as a unit.

A 4-bit register with parallel load and asynchronous clear



operation table

Clear	*Load*	Operation
1	×	Reset register to zero asynchronously
0	0	No change
0	1	Load in a value at rising clock edge



logic symbol

Registers in Verilog

```
module flop (input clk,
             input clear,
             input load,
             input [3:0] d,
             output reg [3:0] q);
    always @ (posedge clk, posedge clear)
        if (clear) q <= 4'b0;
        else if (load) q <= d;
endmodule
```

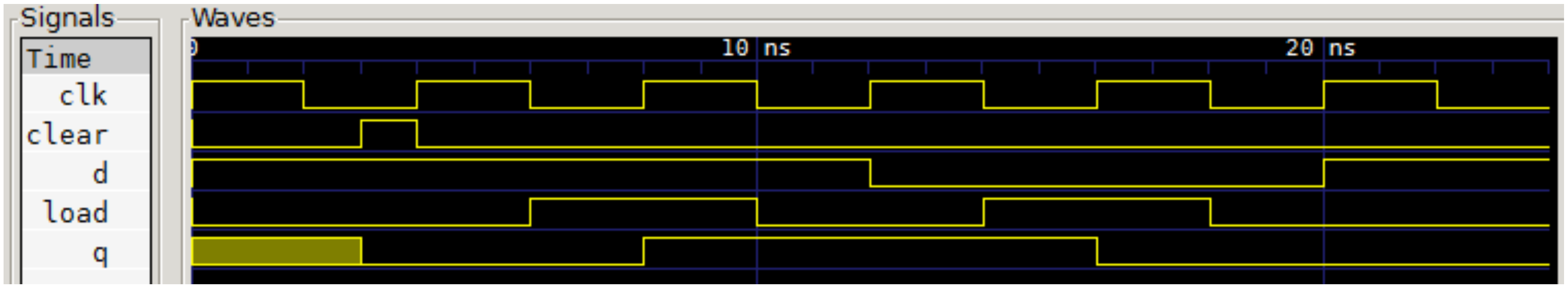
Multiple signals in an `always` statement sensitivity list are separated with a comma or the word `or`

Waveform of a flip-flop

```
module flop (input clk,
             input clear,
             input load,
             input d,
             output reg q);
    always @ (posedge clk, posedge clear)
        if (clear) q <= 1'b0;
        else if (load) q <= d;
endmodule
```

asynchronous input

synchronous input



Shift registers

The operations of shifting and rotating are performed on the value that is stored in a register.

The main usage is for converting from a serial data input stream to a parallel data output or vice versa.

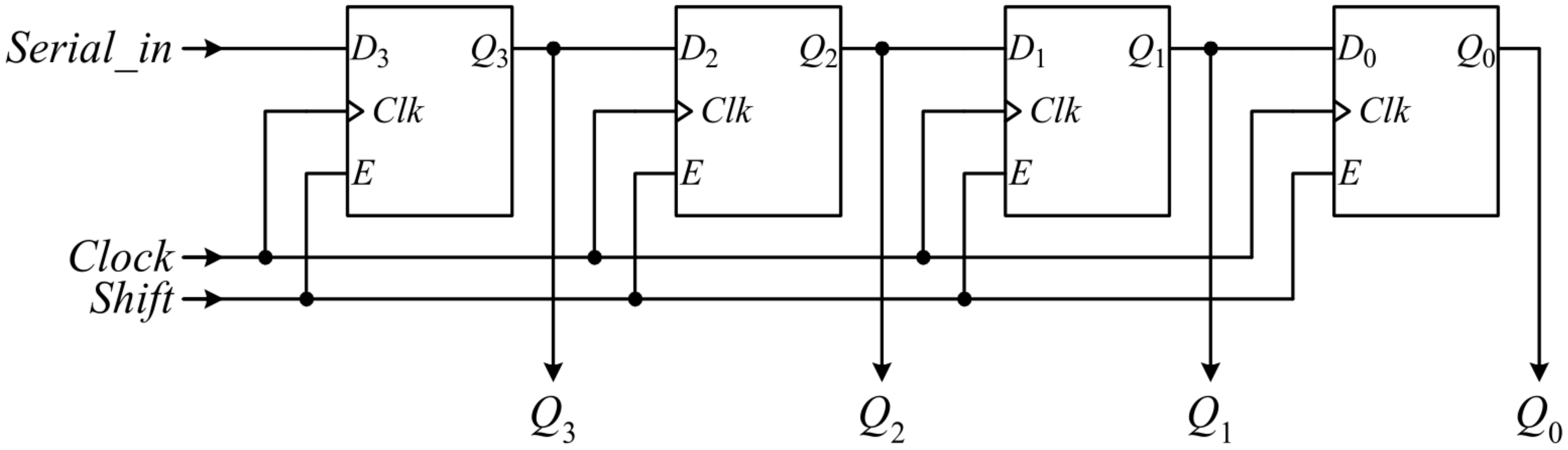
For a **serial-to-parallel data conversion**, the bits are shifted into the register at each clock cycle.

When all of the bits (usually 8 bits) are shifted in, the 8-bit register can be read to produce the 8-bit parallel output.

For a **parallel-to-serial conversion**, the 8-bit register is first loaded with the input data.

The bits are then individually shifted out, one bit per clock cycle, on the serial output line.

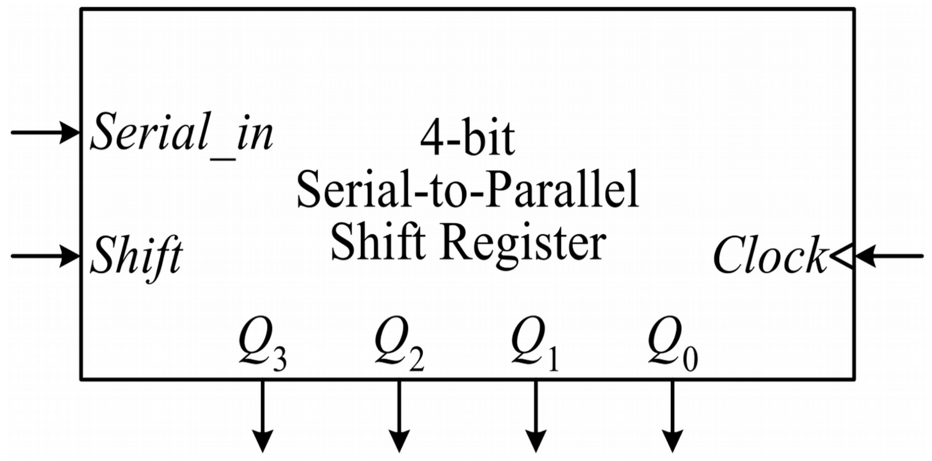
Serial-to-parallel shift register



operation table

Shift	Operation
0	No change
1	One bit from *Serial_in* is shifted in

logic symbol



Serial-to-parallel shift register

The input data bits come in on the Serial_in line at a rate of one bit per clock cycle.

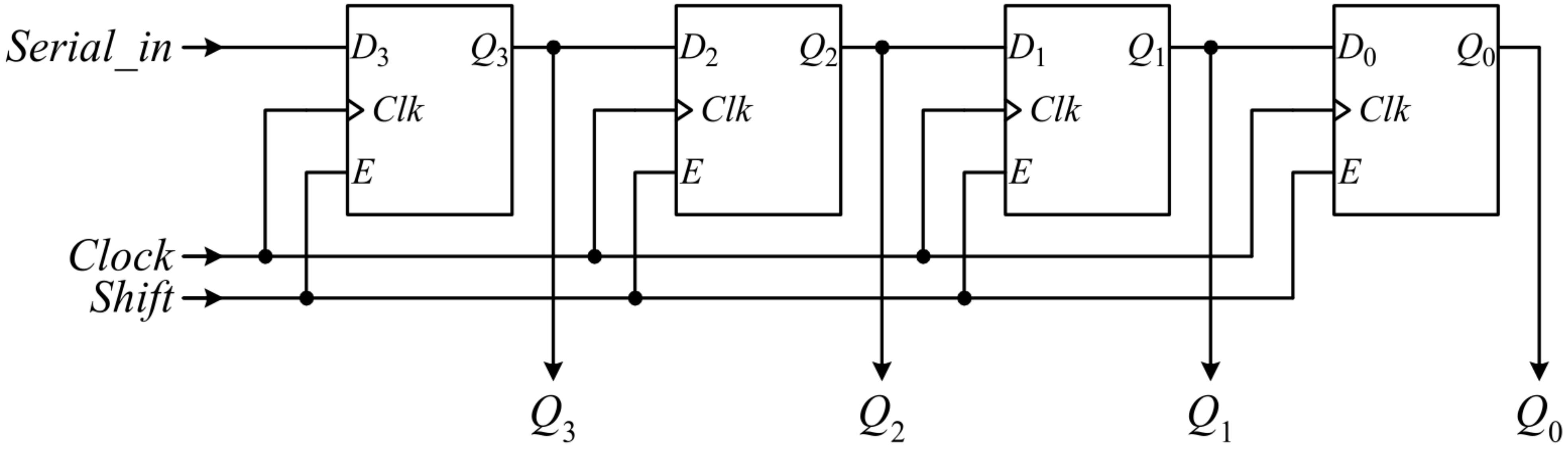
When Shift is asserted, the data bits are loaded in one bit at a time.

In the 1st clock cycle, the 1st bit from the serial input stream, Serial_in, gets loaded into Q_3, while the original bit in Q_3 is loaded into Q_2, Q_2 is loaded into Q_1, etc.

In the 2nd clock cycle, the bit that is in Q_3 (i.e., the 1st bit from the Serial_in line) gets loaded into Q_2, while Q_3 is loaded with the 2nd bit from the Serial_in line.

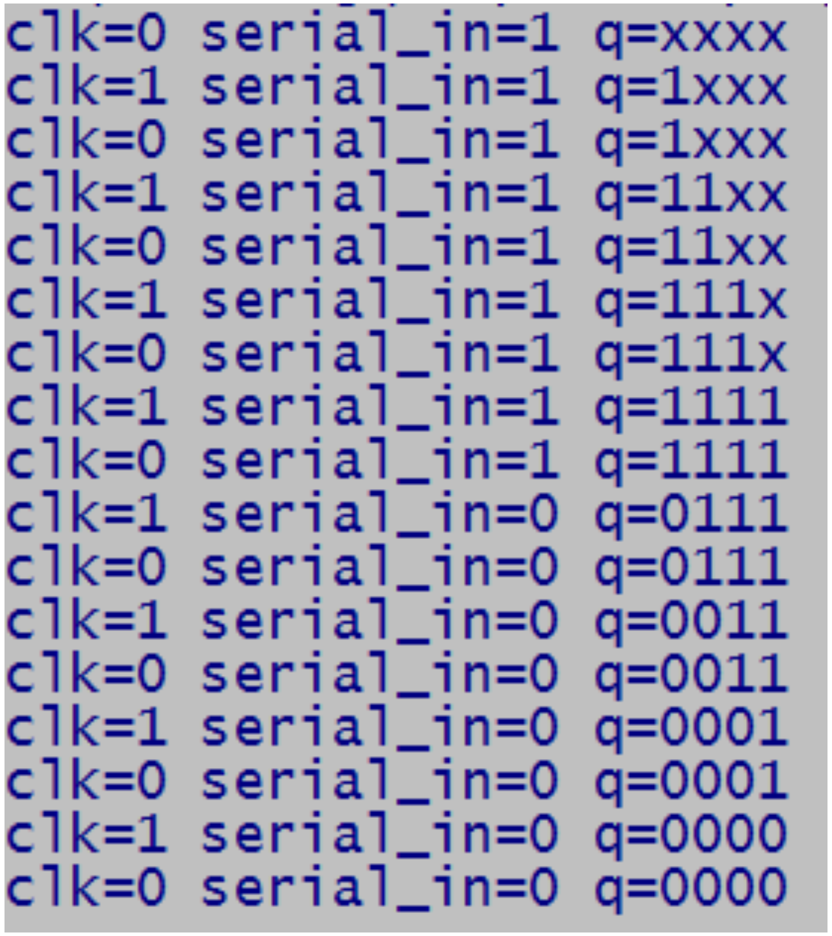
This continues for 4 clock cycles until 4 bits are shifted into the 4 flip-flops, with the 1st bit in Q_0, 2nd bit in Q_1, etc.

These four bits are then available for parallel reading through the output Q.



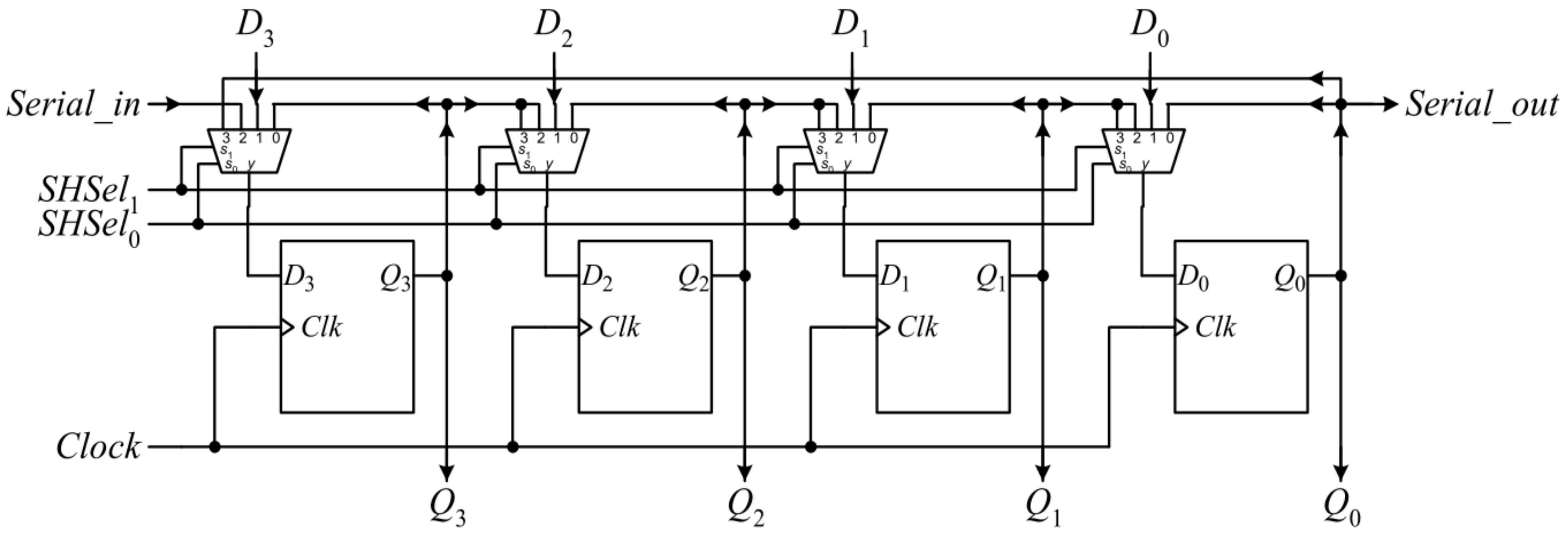
```
module spshift (input serial_in, clk, shift,
                output reg [3:0] q);
    always @ (posedge clk)
        if (shift) q <= {serial_in, q[3:1]};
endmodule
```

```
module spshift_tb ();
    reg serial_in, clk, shift;
    wire [3:0] q;
    spshift dut (serial_in, clk, shift, q);
    initial begin
        $monitor(
        "clk=%b serial_in=%b q=%b",
        clk, serial_in, q);
        clk = 0; serial_in = 1; shift = 1; #1;
        clk = 1; serial_in = 1; #1;
        clk = 0; serial_in = 1; #1;
        clk = 1; serial_in = 1; #1;
        clk = 0; serial_in = 1; #1;
        clk = 1; serial_in = 1; #1;
        clk = 0; serial_in = 1; #1;
        clk = 1; serial_in = 1; #1;
        clk = 0; serial_in = 1; #1;
        clk = 1; serial_in = 0; #1;
        clk = 1; serial_in = 0; #1;
        clk = 0; serial_in = 0; #1;
        clk = 1; serial_in = 0; #1;
        clk = 0; serial_in = 0; #1;
        clk = 1; serial_in = 0; #1;
        clk = 0; serial_in = 0; #1;
        clk = 1; serial_in = 0; #1;
        clk = 0; serial_in = 0; #1;
        $finish;
    end
endmodule
```

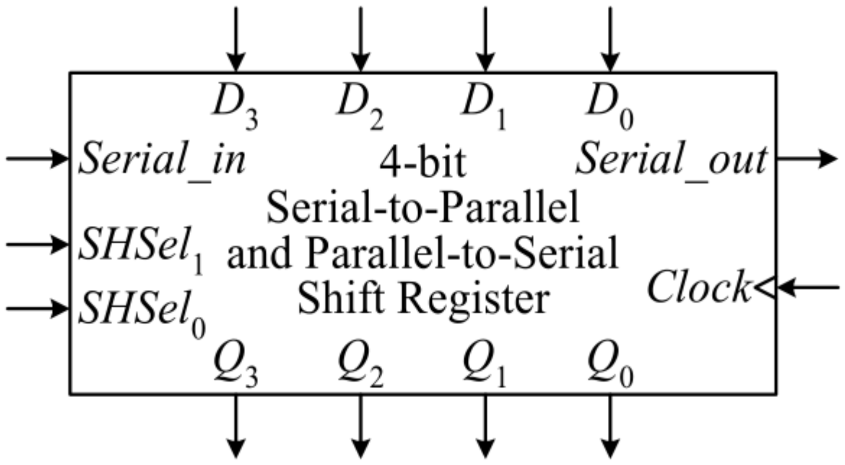


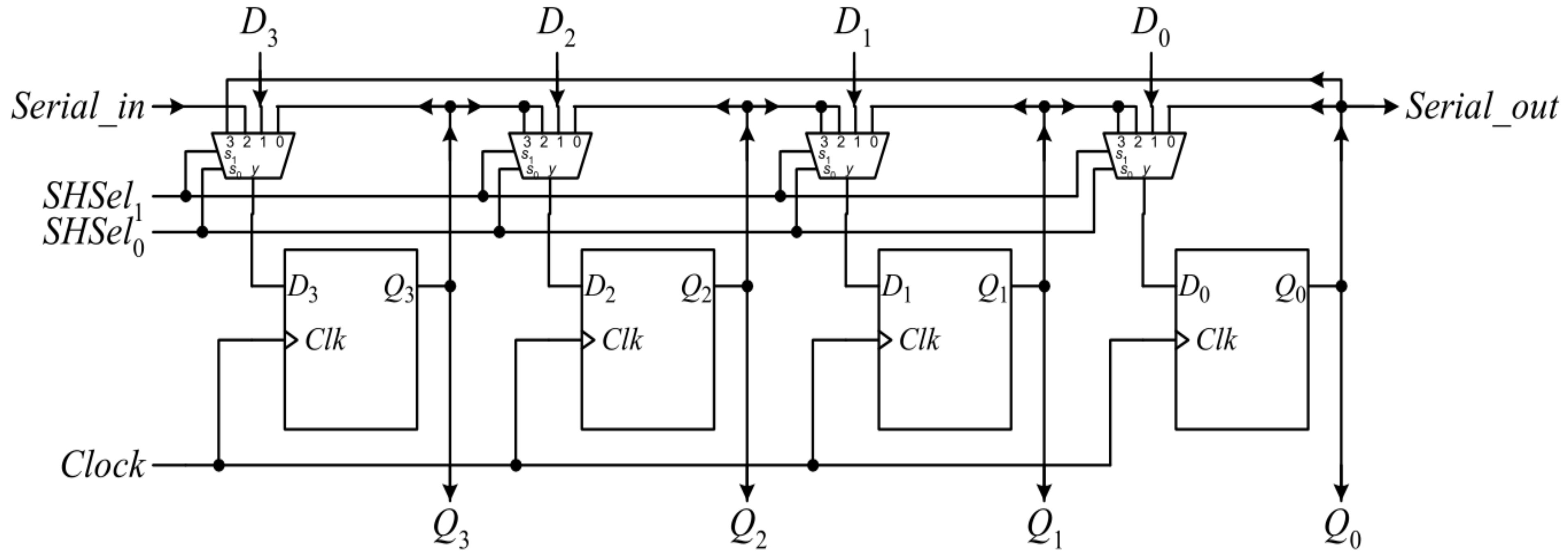
```
clk=0 serial_in=1 q=xxxx
clk=1 serial_in=1 q=1xxx
clk=0 serial_in=1 q=1xxx
clk=1 serial_in=1 q=11xx
clk=0 serial_in=1 q=11xx
clk=1 serial_in=1 q=111x
clk=0 serial_in=1 q=111x
clk=1 serial_in=1 q=1111
clk=0 serial_in=1 q=1111
clk=1 serial_in=0 q=0111
clk=0 serial_in=0 q=0111
clk=1 serial_in=0 q=0011
clk=0 serial_in=0 q=0011
clk=1 serial_in=0 q=0001
clk=0 serial_in=0 q=0001
clk=1 serial_in=0 q=0000
clk=0 serial_in=0 q=0000
```

Serial-to-parallel and parallel-to-serial shift register



$SHSel_1$	$SHSel_0$	Operation
0	0	No operation (i.e., retain current value)
0	1	Parallel load in new value
1	0	Shift right
1	1	Rotate right





```
module shift (input serial_in, clk,
              input [1:0] sh,
              input[3:0] d,
              output serial_out,
              output reg [3:0] q);
    always @ (posedge clk)
        if (~sh[1] && sh[0]) q <= d;
        else if (sh[1] && ~sh[0]) q <= {serial_in, q[3:1]};
        else if (sh[1] && sh[0]) q <= {q[0], q[3:1]};

    assign serial_out = q[0];
endmodule
```

Counters

Counters count a sequence of values.

Modulo-n counter: Counts from decimal 0 to $n - 1$ and back to 0.

Binary coded decimal (BCD) counter: A modulo-*n* counter, with $n = 10$.

Thus, the sequence is always from 0 to 9.

n-bit binary counter: Similar to modulo-*n* counter, but the range is from 0 to $2^n - 1$ and back to 0, where *n* is the number of bits used in the counter.

For example, a 3-bit binary counter sequence in decimal is 0, 1, 2, 3, 4, 5, 6, 7.

… more on the next slide

Counters

Gray-code counter: The sequence is coded so that any two consecutive values must differ in only one bit.

Example: one possible 3-bit gray-code counter sequence is 000, 001, 011, 010, 110, 111, 101, and 100.

Ring counter: The sequence starts with a string of 0 bits followed by one 1 bit, as in 0001.

This counter simply rotates the bits to the left on each count.

For example, a 4-bit ring counter sequence is 0001, 0010, 0100, 1000, and back to 0001.

Binary up counter

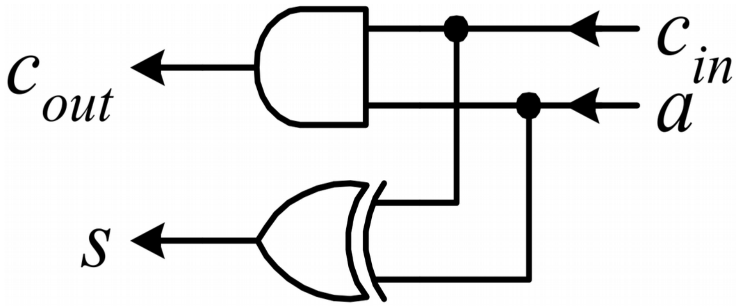
To get to the next up-count sequence from the value that is stored in a register, we simply have to add a 1 to it.

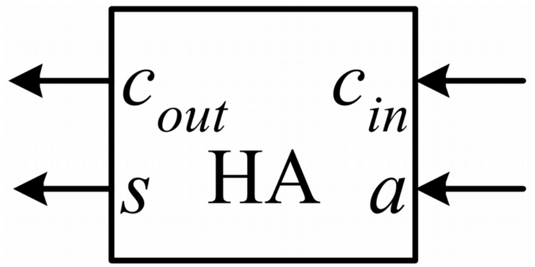
The 1 can be added using a half adder (HA) via the carry-in signal.

$$c_{out} = a\, c_{in}$$

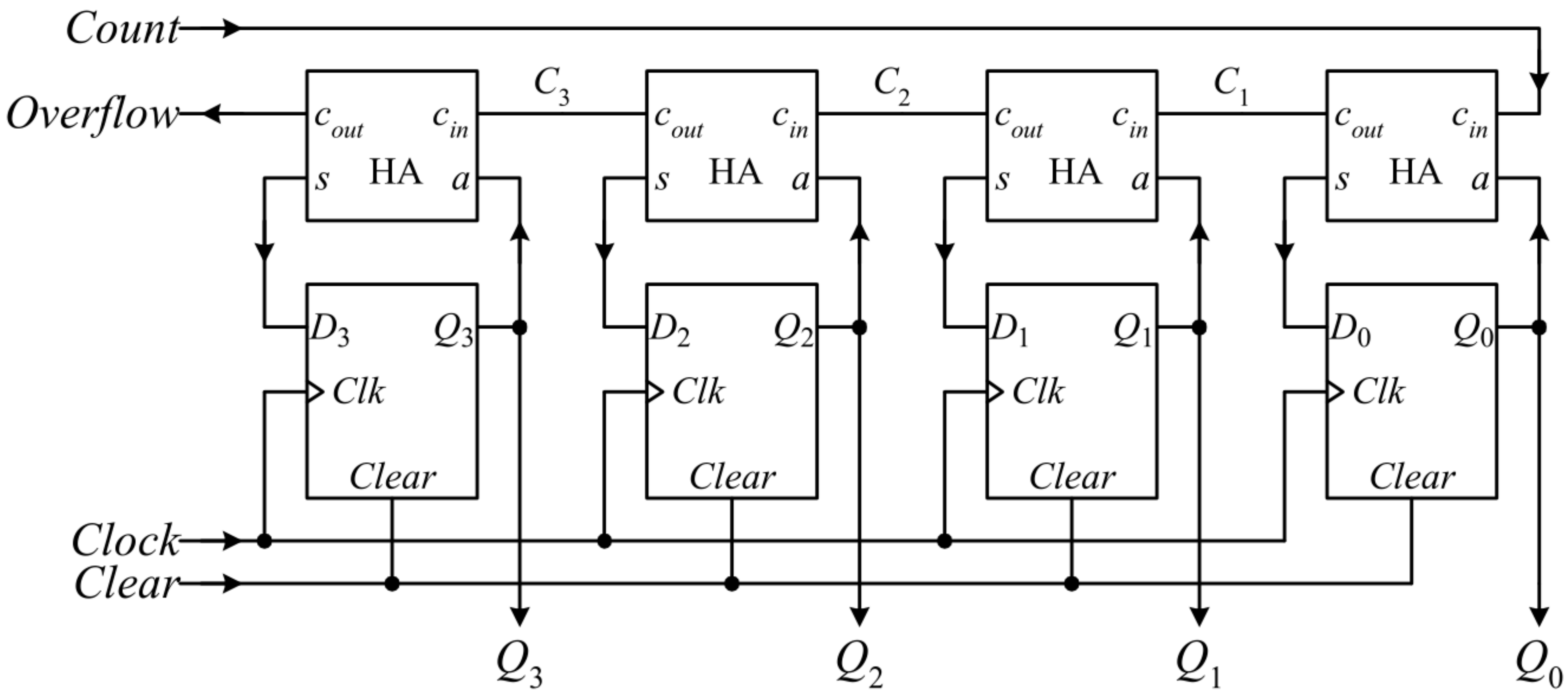
$$s = a \oplus c_{in}$$

a	c_{in}	c_{out}	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

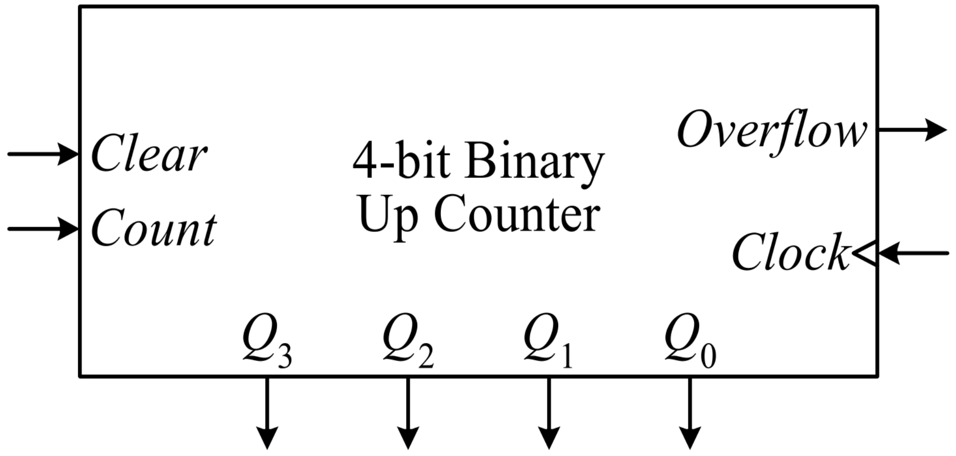




The 4-bit binary up counter



Clear	*Count*	Operation
1	×	Reset counter to zero
0	0	No change
0	1	Count up



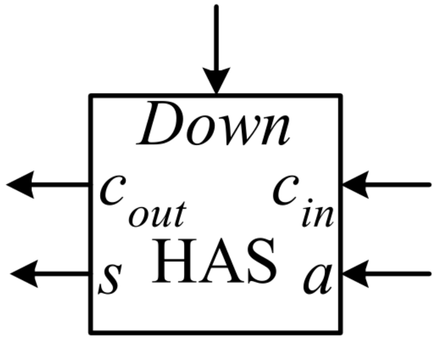
A binary up counter with asynchronous reset in Verilog

```
module counter #(parameter N = 4)
                (input clk,
                 input clear,
                 output reg [N-1:0] q);
    always @ (posedge clk or posedge clear)
        if (clear) q <= 0;
        else q <= q + 1;
endmodule
```

Binary up-down counter

We need both an adder and a subtractor for the data input to the register.

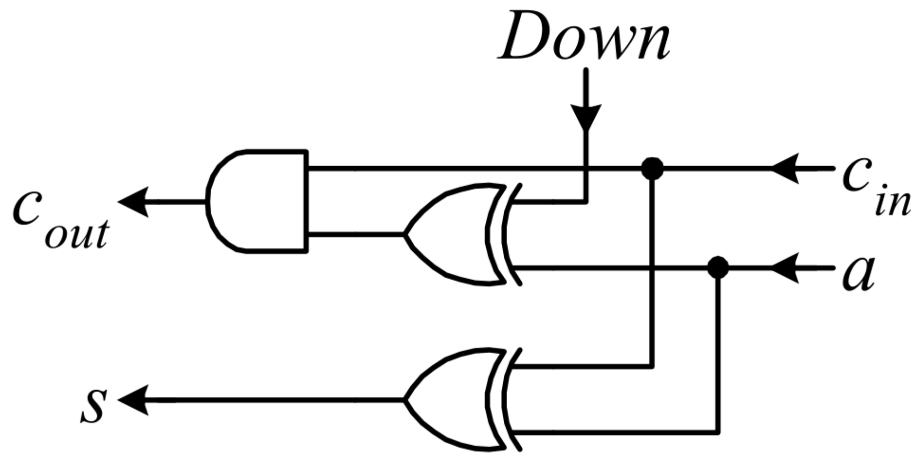
Half adder-subtractor (HAS):



$$c_{out} = Down'\,a\,c_{in} + Down\,a'\,c_{in} = (Down \oplus a)\,c_{in}$$

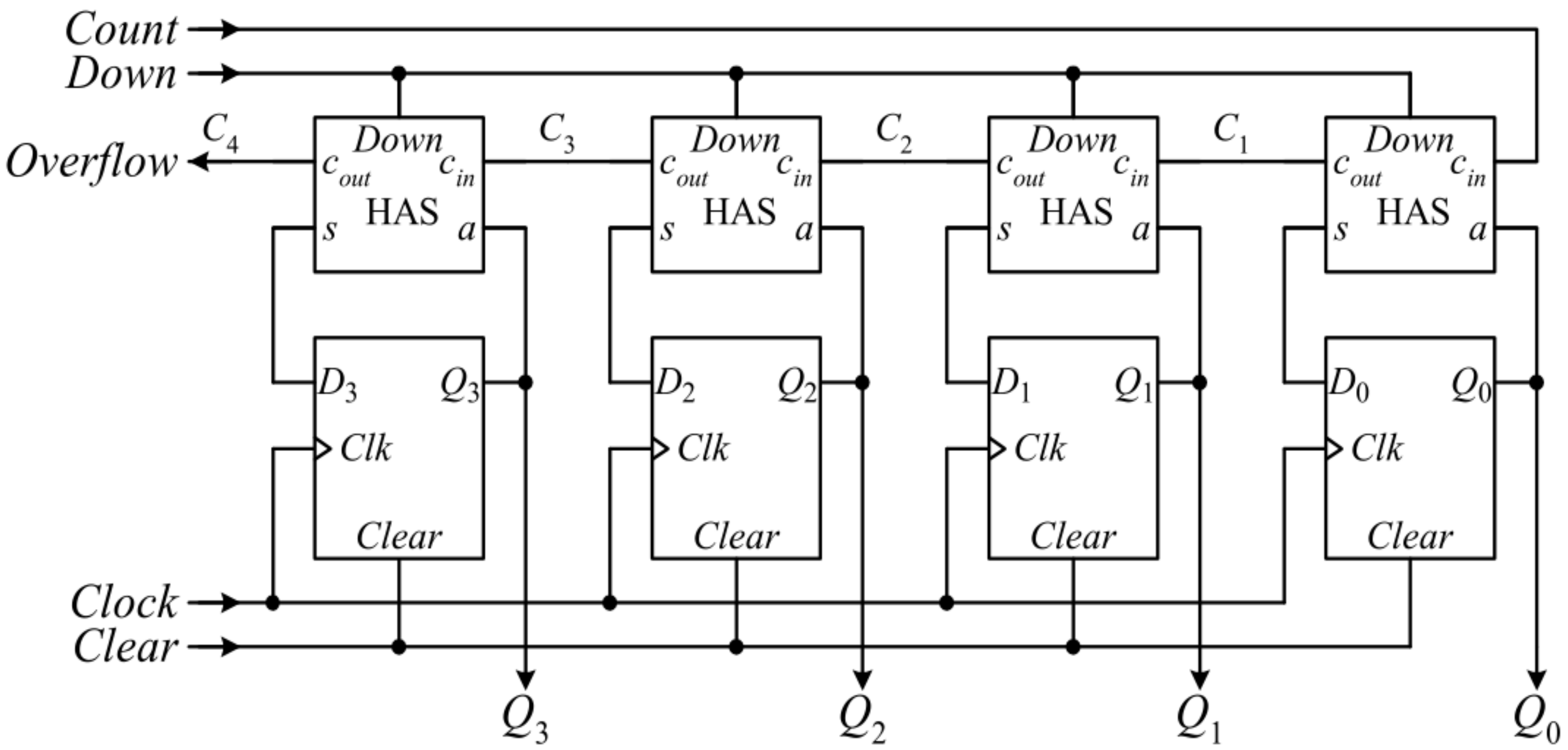
$$s = Down'\,(a \oplus c_{in}) + Down\,(a \oplus c_{in}) = a \oplus c_{in}$$

Down	*a*	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	0	1
1	1	1	0	0

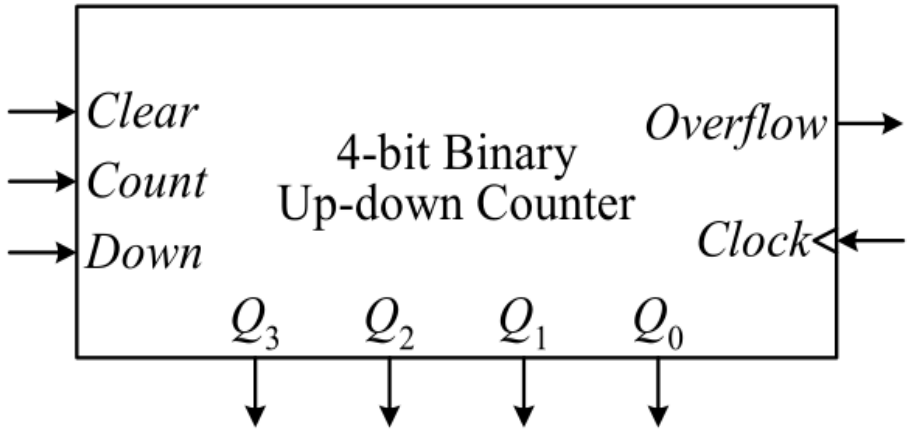


The Down signal is to select if we want to count up or down.

A 4-bit binary up-down counter



Clear	*Count*	*Down*	Operation
1	×	×	Reset counter to zero
0	0	×	No change
0	1	0	Count up
0	1	1	Count down

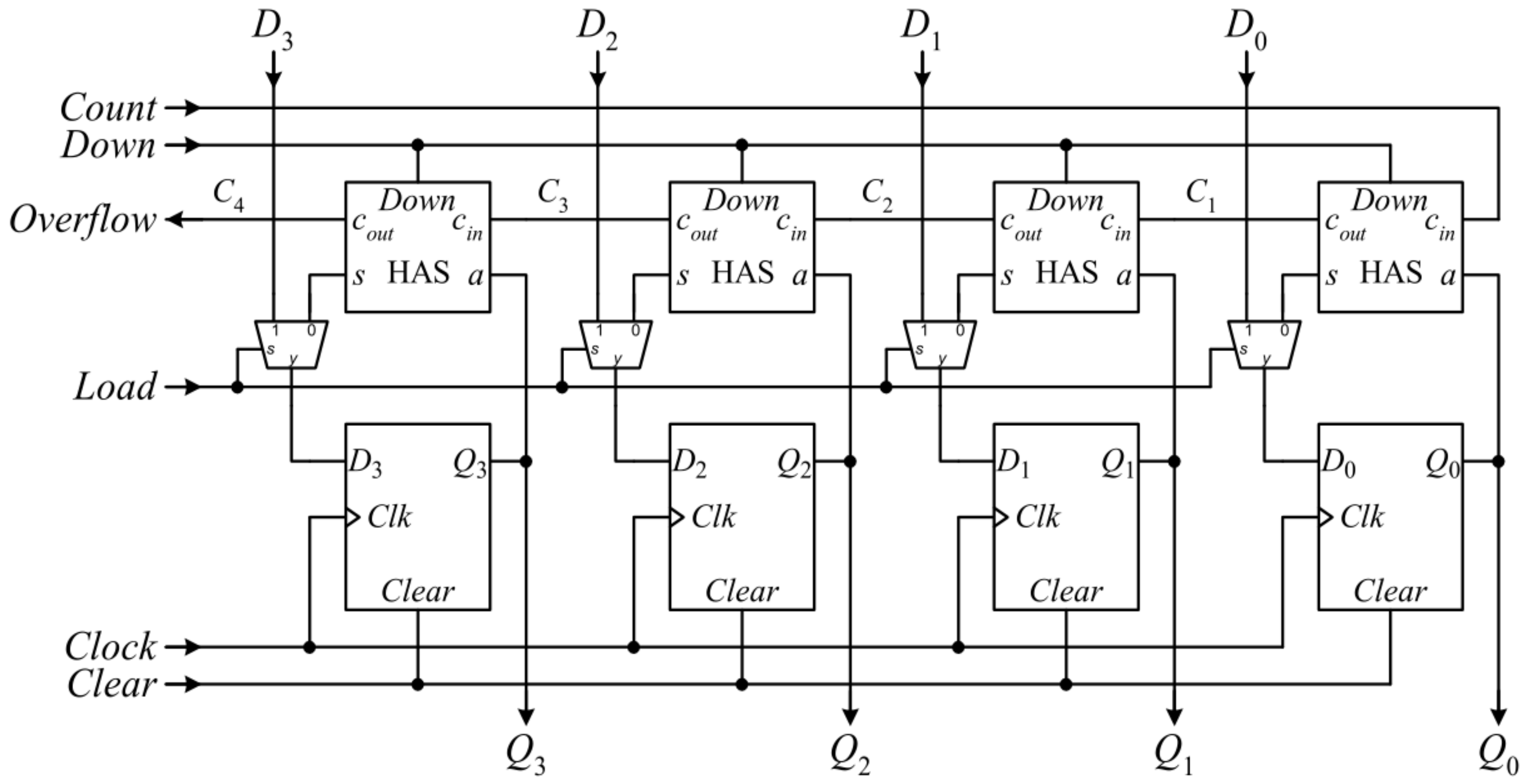


Binary up-down counter with parallel load

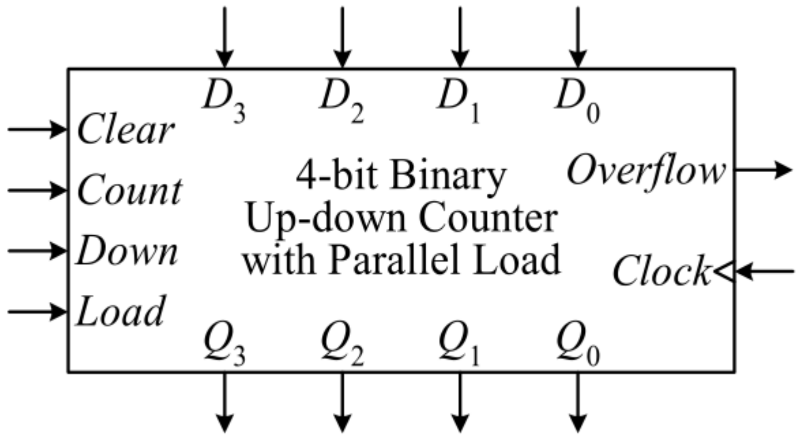
To start the count sequence with any number other than zero can be accomplished by modifying our counter circuit to allow it to load in an initial value.

With the value loaded into the register, we can now count starting from this new value.

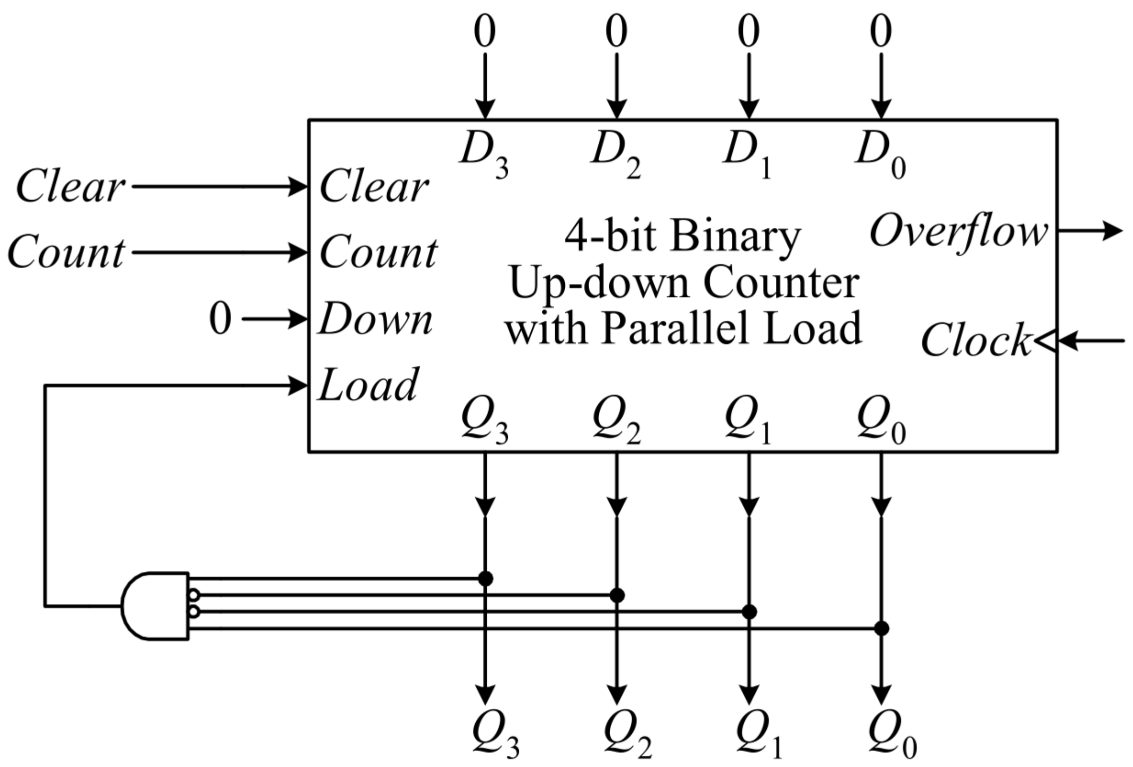
A 4-bit binary up-down counter with parallel load and asynchronous clear



Clear	Load	Count	Down	Operation
1	×	×	×	Reset counter to zero
0	0	0	×	No change
0	0	1	0	Count up
0	0	1	1	Count down
0	1	×	×	Load value



BCD up counter



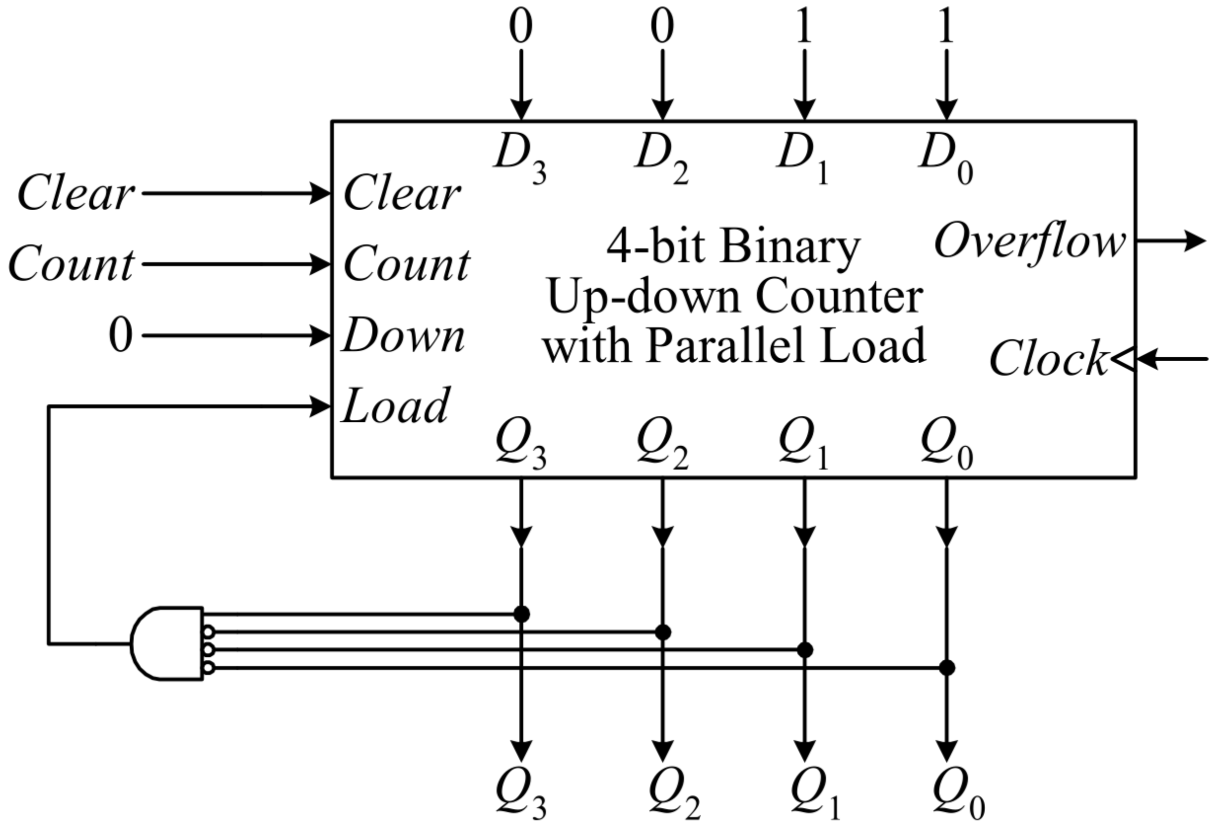
When the count value is 9, the AND gate comparator outputs a 1 to assert the Load

Once the Load is asserted, the next counter value will be the value loaded in from the input D.

Since D is connected to all 0's, the counter will cycle back to 0 at the next rising clock edge.

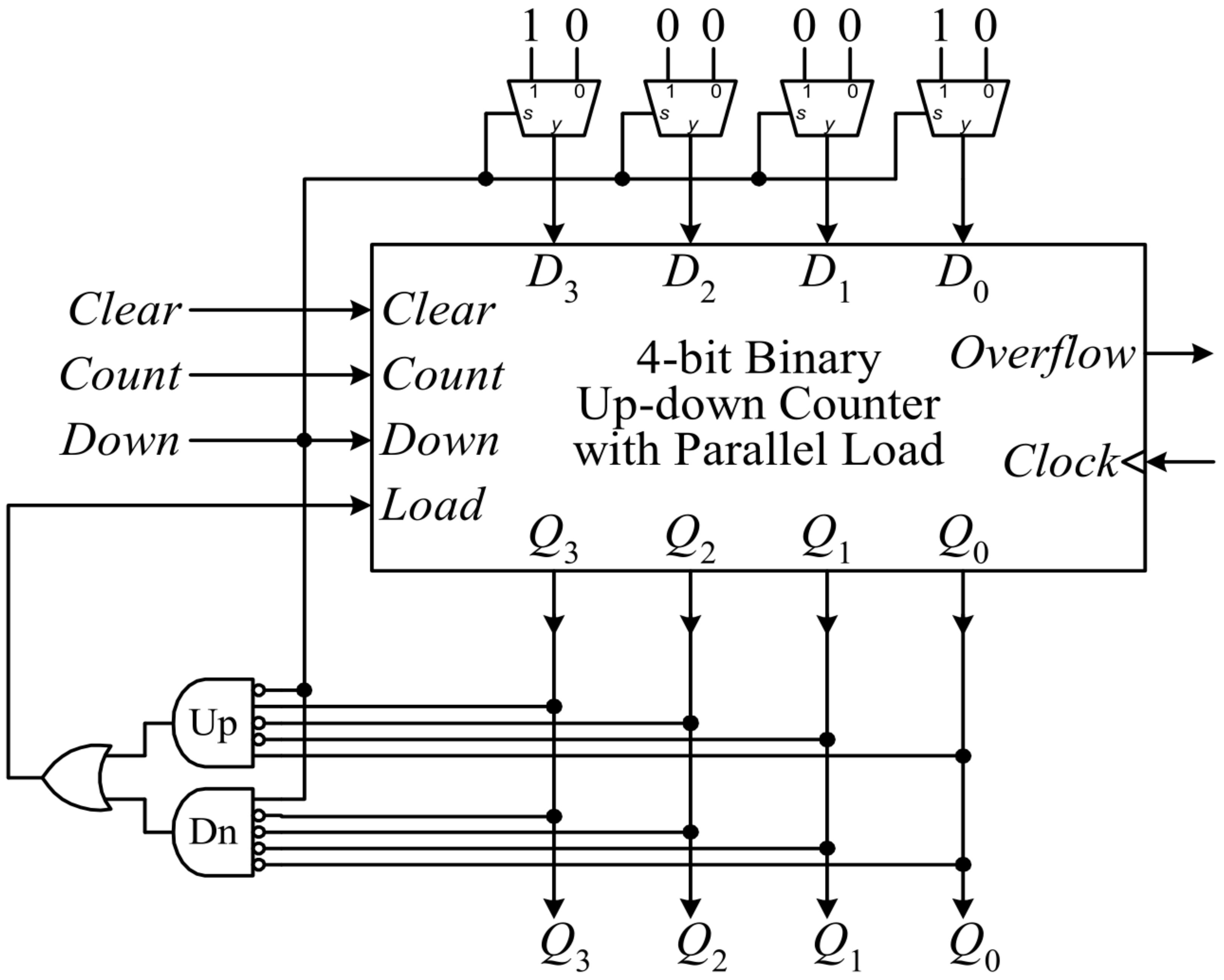
The Down line is connected to a 0, since we only want to count up.

4-bit binary up counter from 3 to 8 (in decimal), and back to 3



Here, we test for the number 8 as the last number in the sequence, and the first number to load in is a 3.

BCD up-down counter



Non-consecutive counter

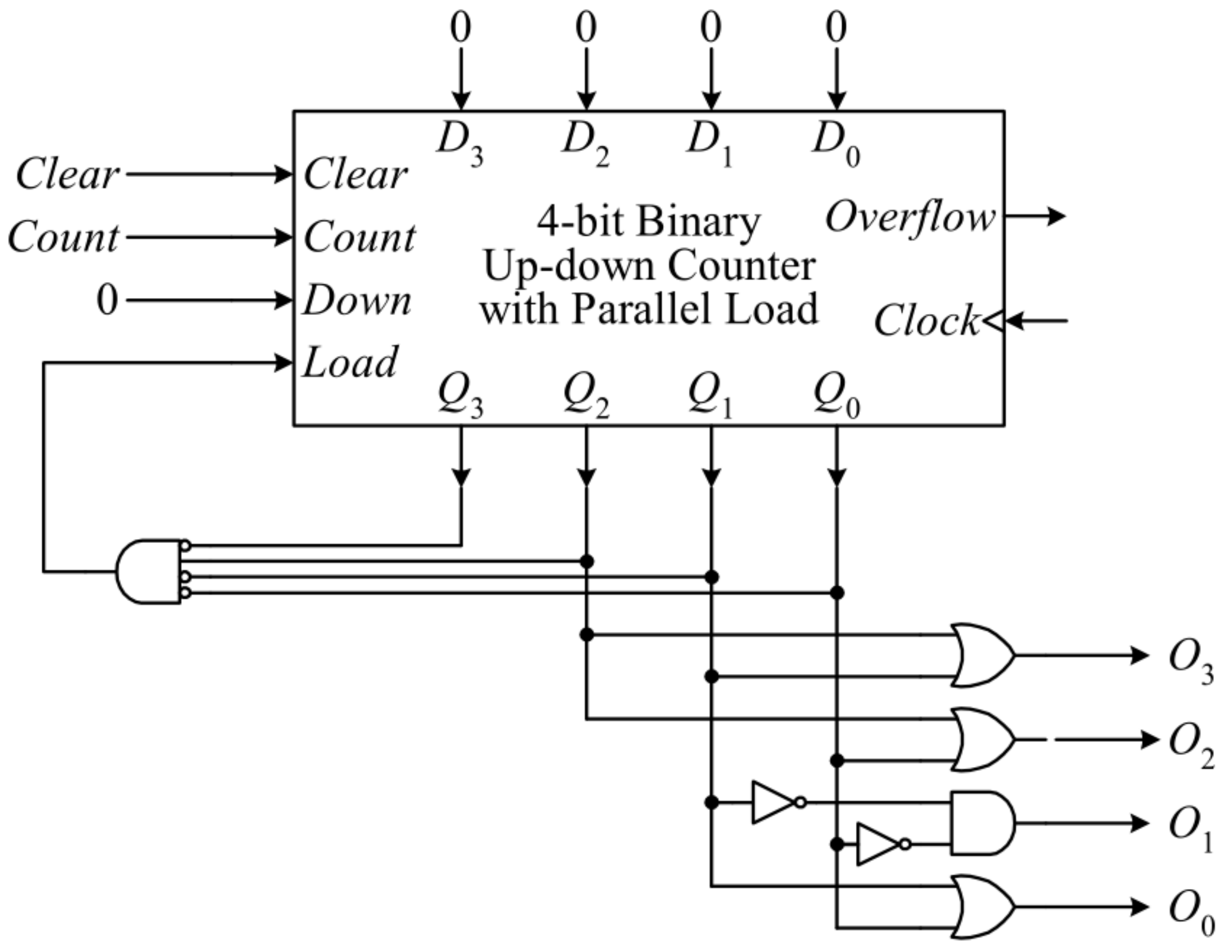
In order to output numbers that are not consecutive, we need to design an output circuit that maps from one number to another number.

Example: use the 4-bit binary up-down counter with parallel load to construct an up-down counter circuit that outputs the sequence, 2, 5, 9, 13, 14, repeatedly.

The required sequence has five numbers, so we will first design a counter to count from 0 to 4.

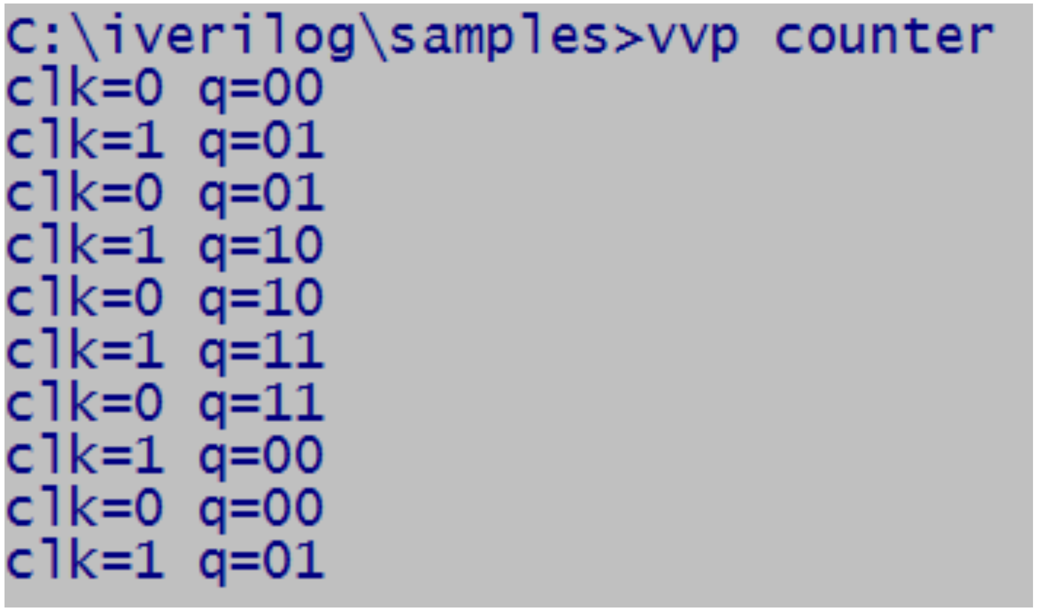
The output circuit will then map the numbers, 0, 1, 2, 3, 4 to the required output numbers, 2, 5, 9, 13, 14, respectively.

Decimal Input	Q_3	Q_2	Q_1	Q_0	Decimal Output	O_3	O_2	O_1	O_0
0	0	0	0	0	2	0	0	1	0
1	0	0	0	1	5	0	1	0	1
2	0	0	1	0	9	1	0	0	1
3	0	0	1	1	13	1	1	0	1
4	0	1	0	0	14	1	1	1	0
Rest of the Combinations						×	×	×	×



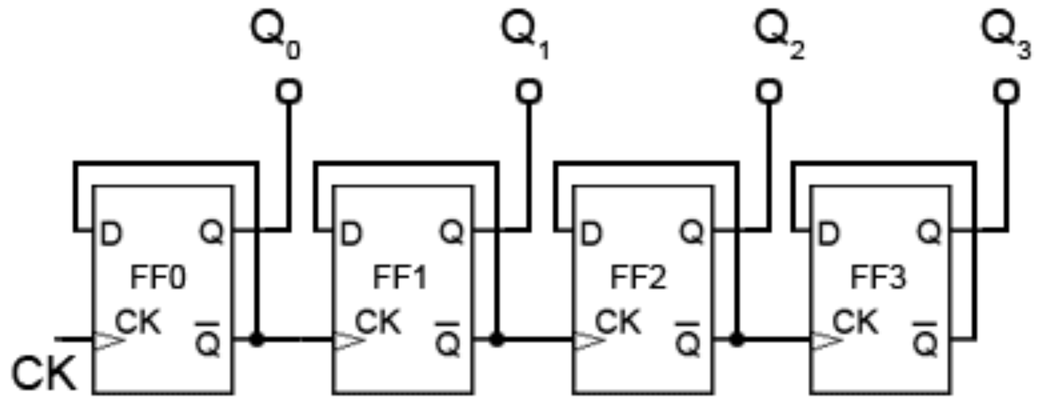
Testing a 2-bit binary counter

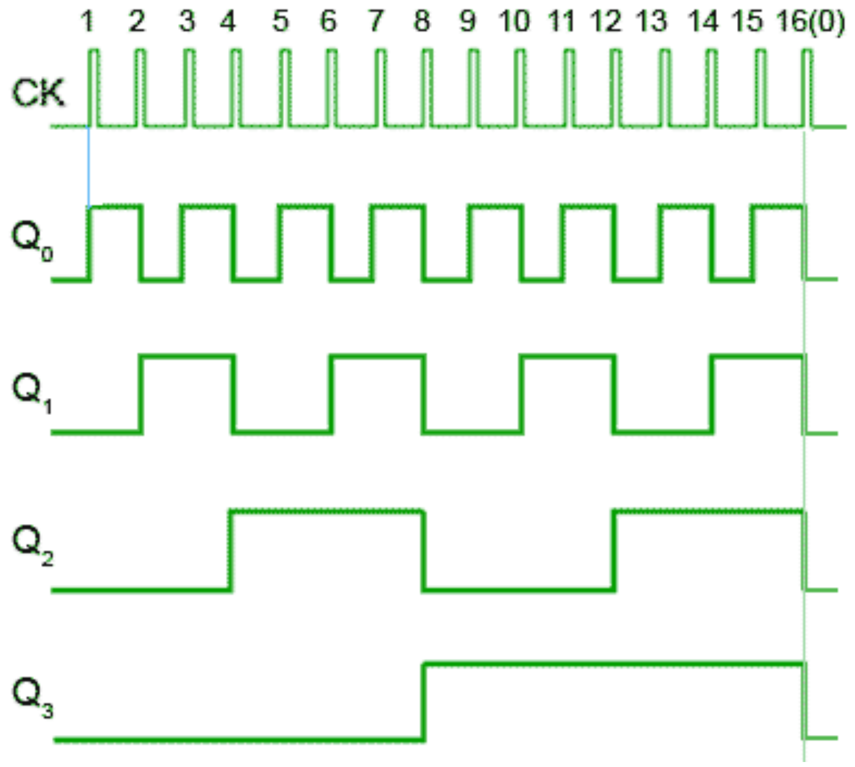
```
`timescale 1ns/1ps
module counter (input clk,
                input clear,
                output reg [1:0] q);
    always @ (posedge clk or posedge clear)
        if (clear) q <= 0;
        else q <= q + 1;
endmodule

module counter_tb ();
    reg clk, clear;
    wire [1:0] q;
    counter dut (clk, clear, q);
    initial begin
        $monitor(
        "clk=%b q=%b",
        clk, q);
        clk = 0; clear = 1; #1;
        clk = 1; clear = 0; #1;
        clk = 0; #1;
        clk = 1; #1;
        clk = 0; #1;
        clk = 1; #1;
        clk = 0; #1;
        clk = 1; #1;
        clk = 0; #1;
        clk = 1; #1;
        $finish;
    end
endmodule
```



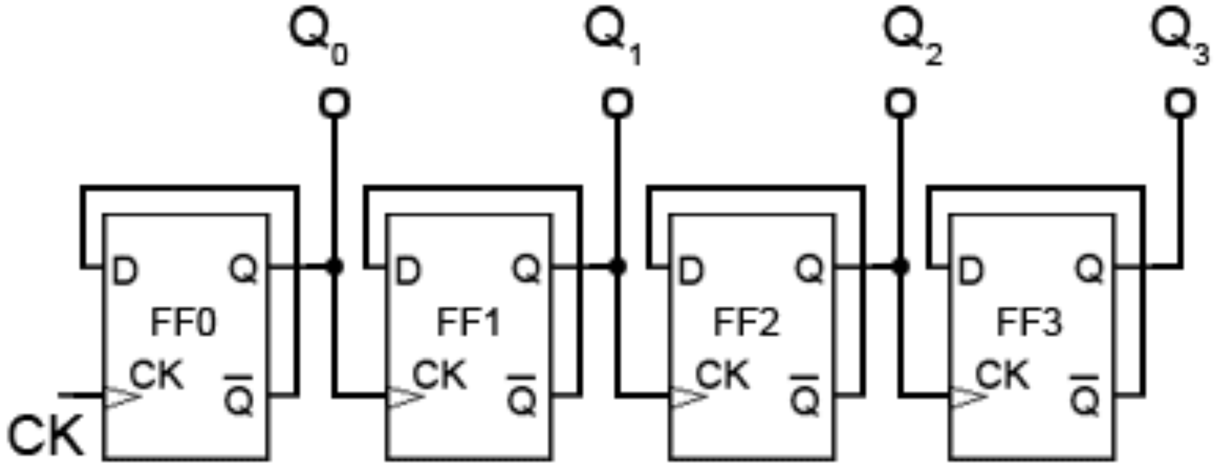
```
C:\iverilog\samples>vvp counter
clk=0 q=00
clk=1 q=01
clk=0 q=01
clk=1 q=10
clk=0 q=10
clk=1 q=11
clk=0 q=11
clk=1 q=00
clk=0 q=00
clk=1 q=01
```

Four-bit asynchronous up counter





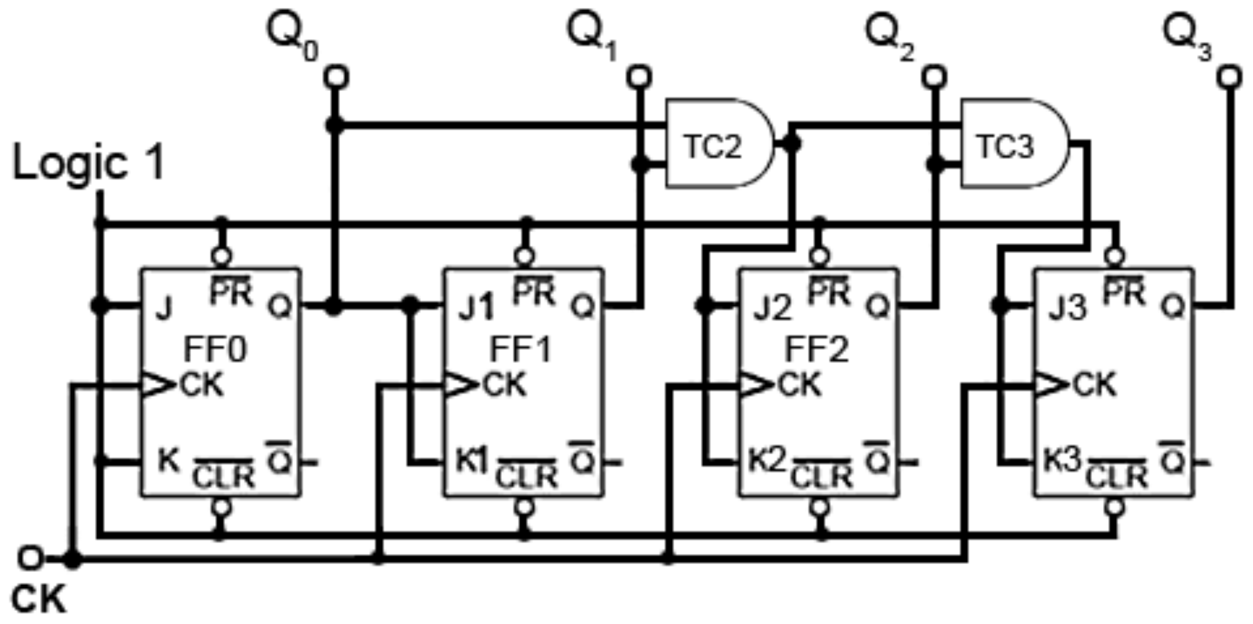
Four-bit asynchronous down counter



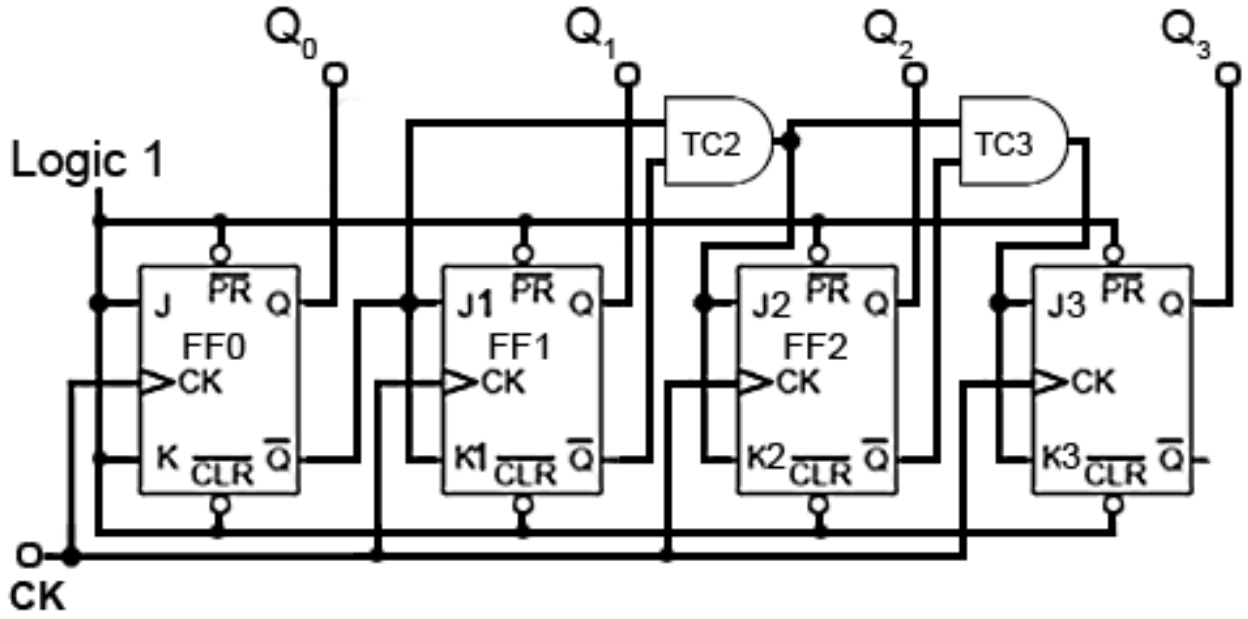
To convert the up counter to count DOWN instead, is simply a matter of modifying the connections between the flip-flops.

By taking both the output lines and the CK pulse for the next flip-flop in sequence from the Q output, a positive edge triggered counter will count down from 1111 to 0000.

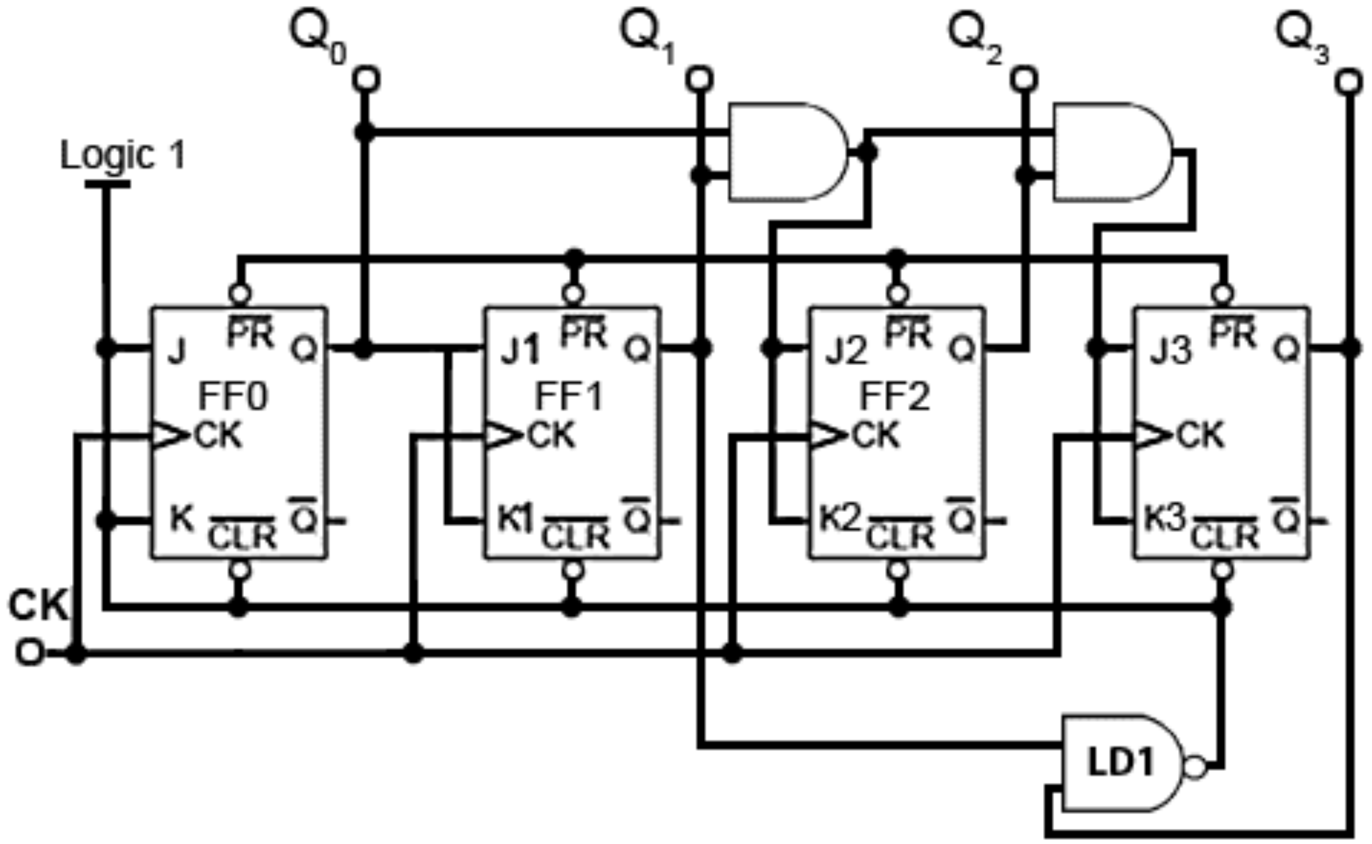
Four-bit synchronous up counter using JK flip-flops



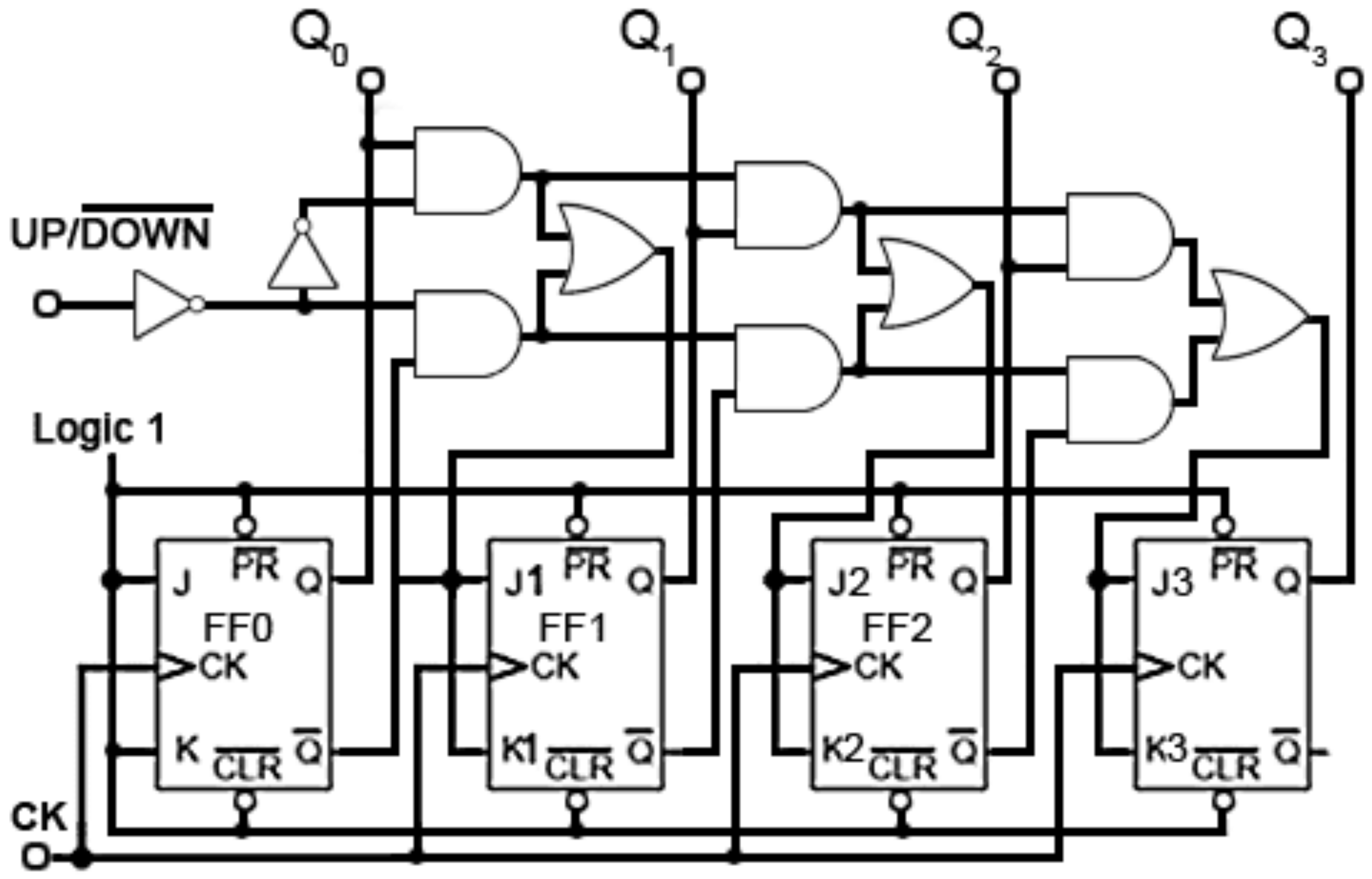
Four-bit synchronous down counter using JK flip-flops



Synchronous BCD up counter using JK flip-flops



Four-bit synchronous up/down counter using JK flip-flops



Register files

Often, instead of having several individual registers, we want to have an array of registers.

This array of registers is known as a **register file**.

In a register file, all of the respective control signals for the individual registers are connected in common.

All of the respective data input and output lines for all of the registers are also connected in common.

In addition, address lines are used to specify which register in the register file is to be accessed.

In a microprocessor, the register file usually is used for the source operands of the ALU.

Since the ALU usually takes 2 input operands, we like the register file to be able to output 2 values from possibly 2 different locations of the register file at the same time.

So, a typical register file will have one write port and two read ports.

Each port has its own enable line and address lines.

When the read enable line is de-asserted, the read port will output a 0.

When the read enable line is asserted, the content of the register specified by the read address lines is passed to the output port.

The write enable line is used to load a value into the register specified by the write address lines.

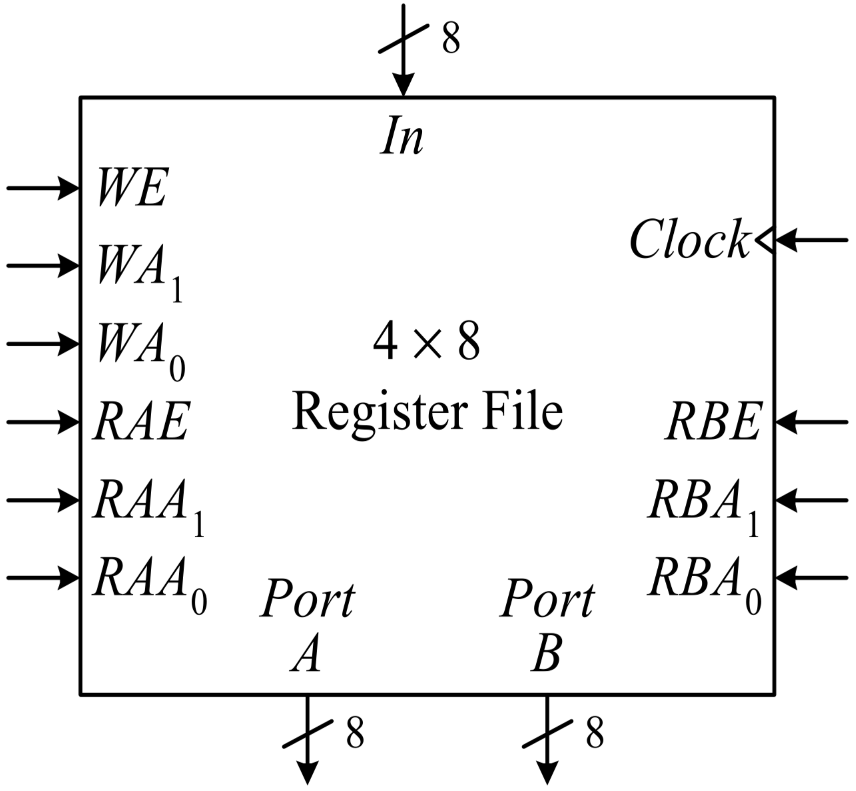
A 4 × 8 register file (4 registers, 8-bits wide each):

In: the 8-bit write port

Port A, *Port B*: the 8-bit read ports

WE: the active-high write enable line.

To write a value into the register file, this line must be asserted.



WA_1, *WA_0*: the address lines for selecting the write location

4 locations in this register file => 2 address lines

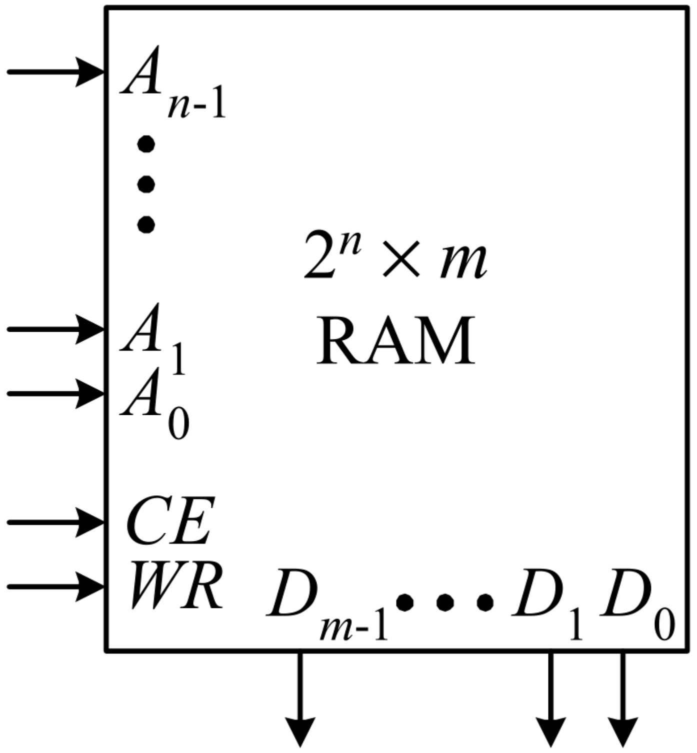
RAE: the read enable line for *Port A*.

RAA_1, *RAA_0*: the read address select lines for *Port A*.

For *Port B*, we have the enable line, *RBE*, and the two address lines, *RBA_1* and *RBA_0*.

Memory

We make memory from latches because we usually want a lot of memory and we want it very cheap, so we need to make each memory cell as small as possible.

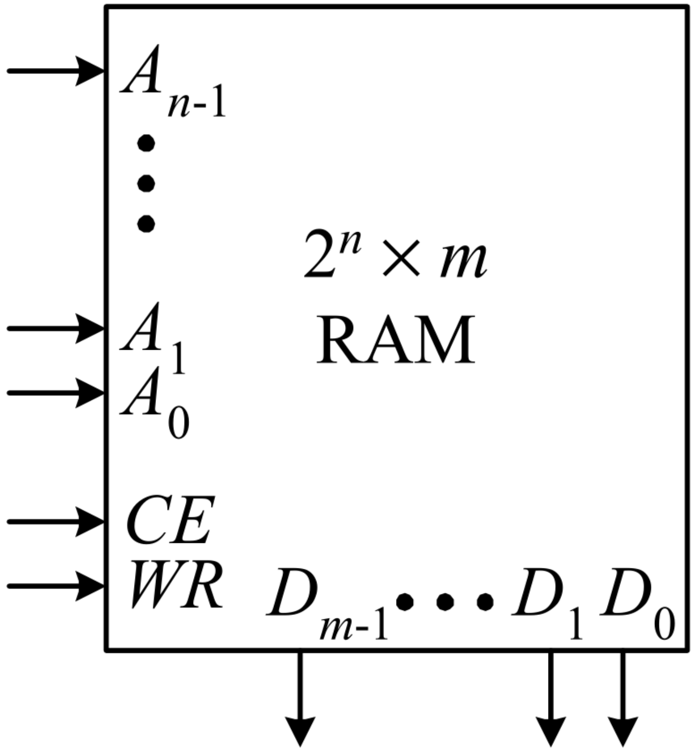


D_i – data lines A_i – address lines

The data lines serve for both input and output of the data to the location that is specified by the address lines.

The number of data lines depends on how many bits are used for storing data in each memory location.

The number of address lines depends on how many locations are in the memory chip.



Two control lines: chip enable (*CE*), and write enable (*WR*).

In order for a microprocessor to access memory, either with the read operation or the write operation, the active-high *CE* line must first be asserted.

WR = 0: read (data from the memory is retrieved)

WR = 1: write (data from the microprocessor is written into the memory)

CE	WR	Operation
0	$\times$	None
1	0	Read from memory location selected by address lines
1	1	Write to memory location selected by address lines

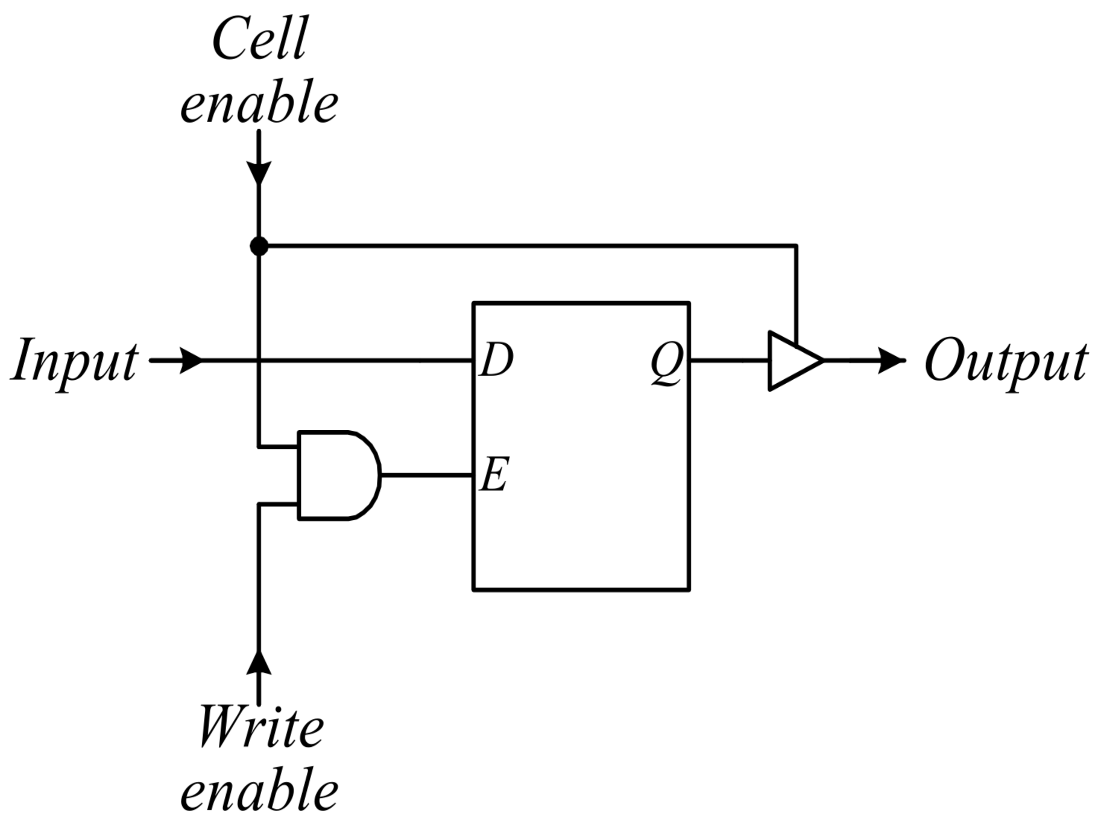
The RAM chip does not require a clock signal.

Both the read and write memory operations are not synchronized to the global system clock.

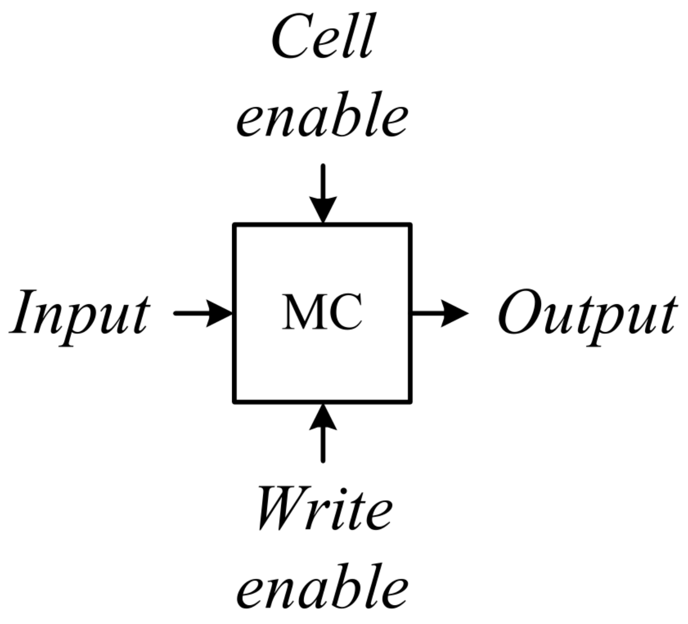
Instead the data operations are synchronized to the two control lines, CE and WR.

Each bit in a static RAM chip is stored in a memory cell.

Memory cell in a static RAM



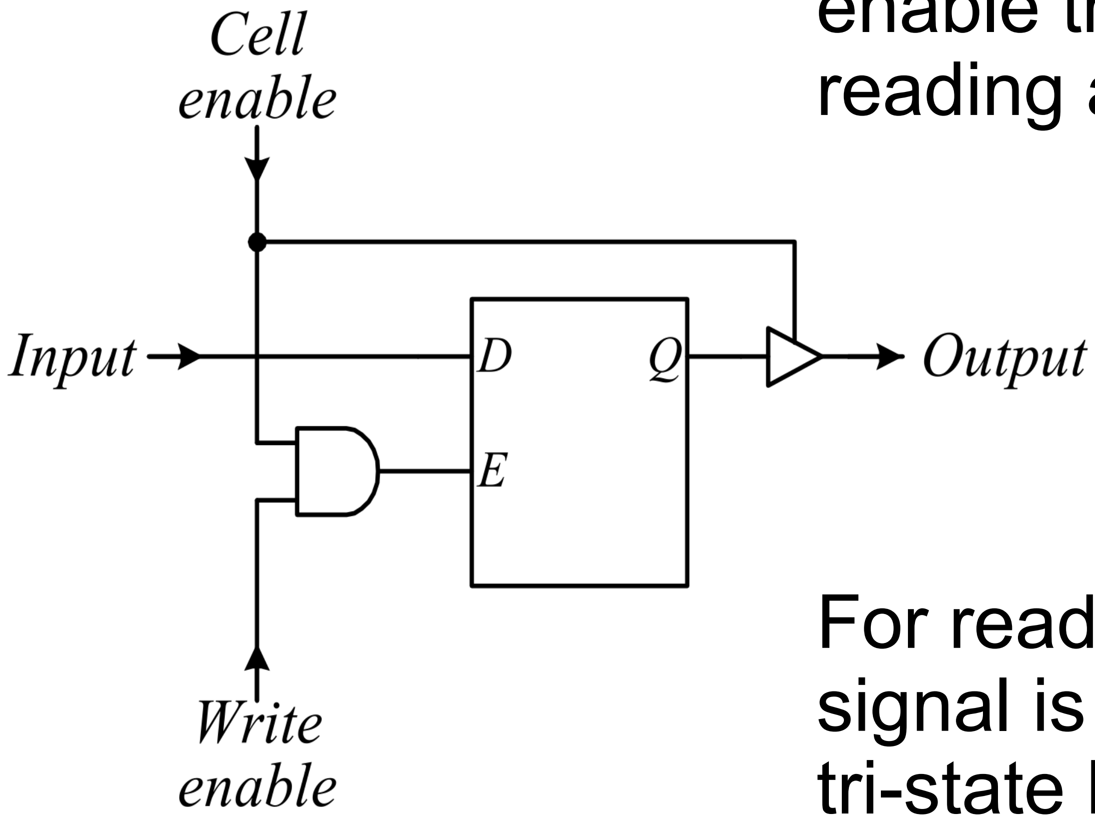
logic symbol



The main component in the cell is a D latch with enable.

A tri-state buffer is connected to the output of the D latch so that it can be selectively read from.

Memory cell in a static RAM



The Cell enable signal is used to enable the memory cell for both reading and writing.

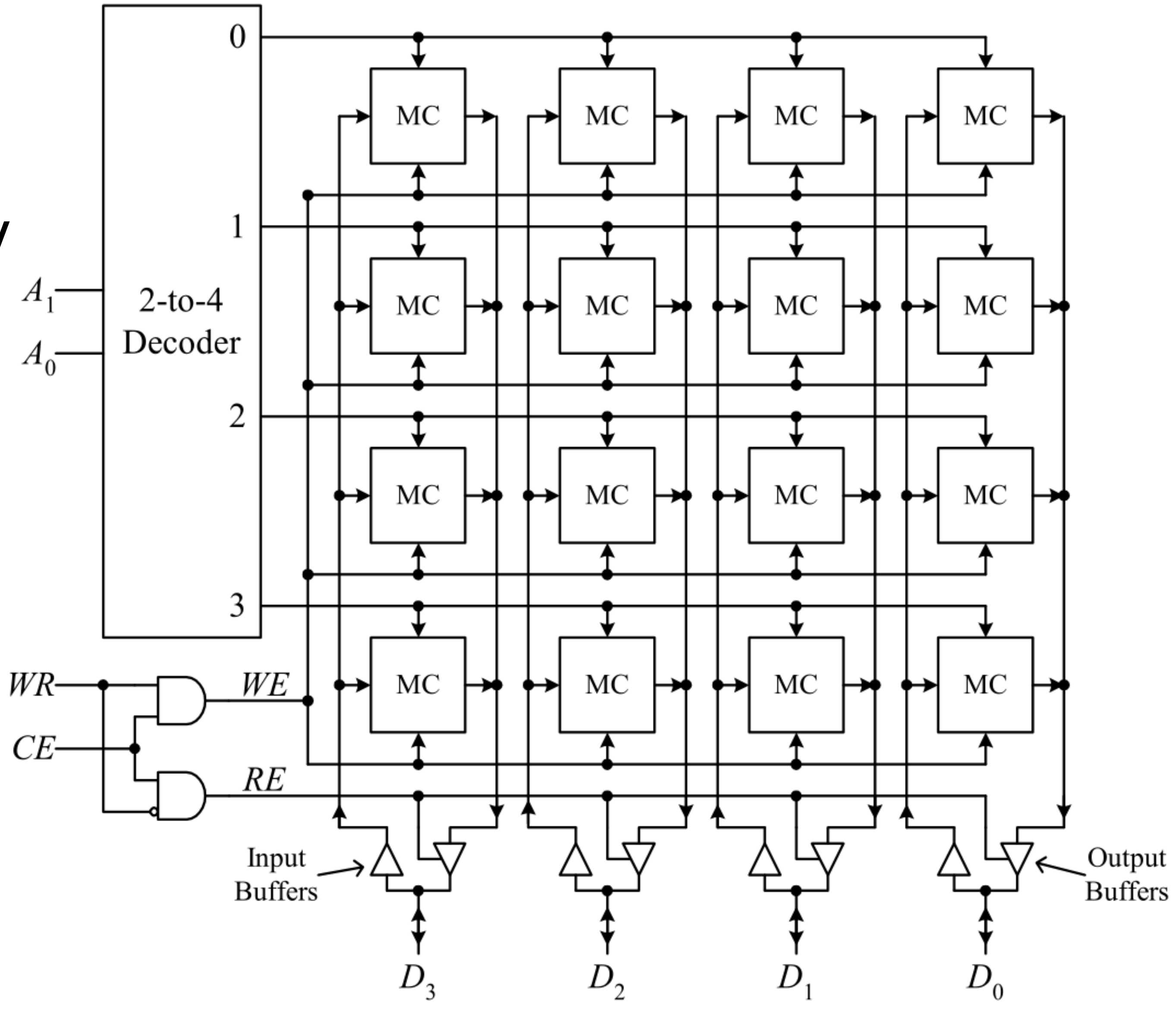
For reading, the Cell enable signal is used to enable the tri-state buffer.

For writing, the Cell enable together with the Write enable signals are used to enable the D latch so that the data on the Input line is latched into the cell.

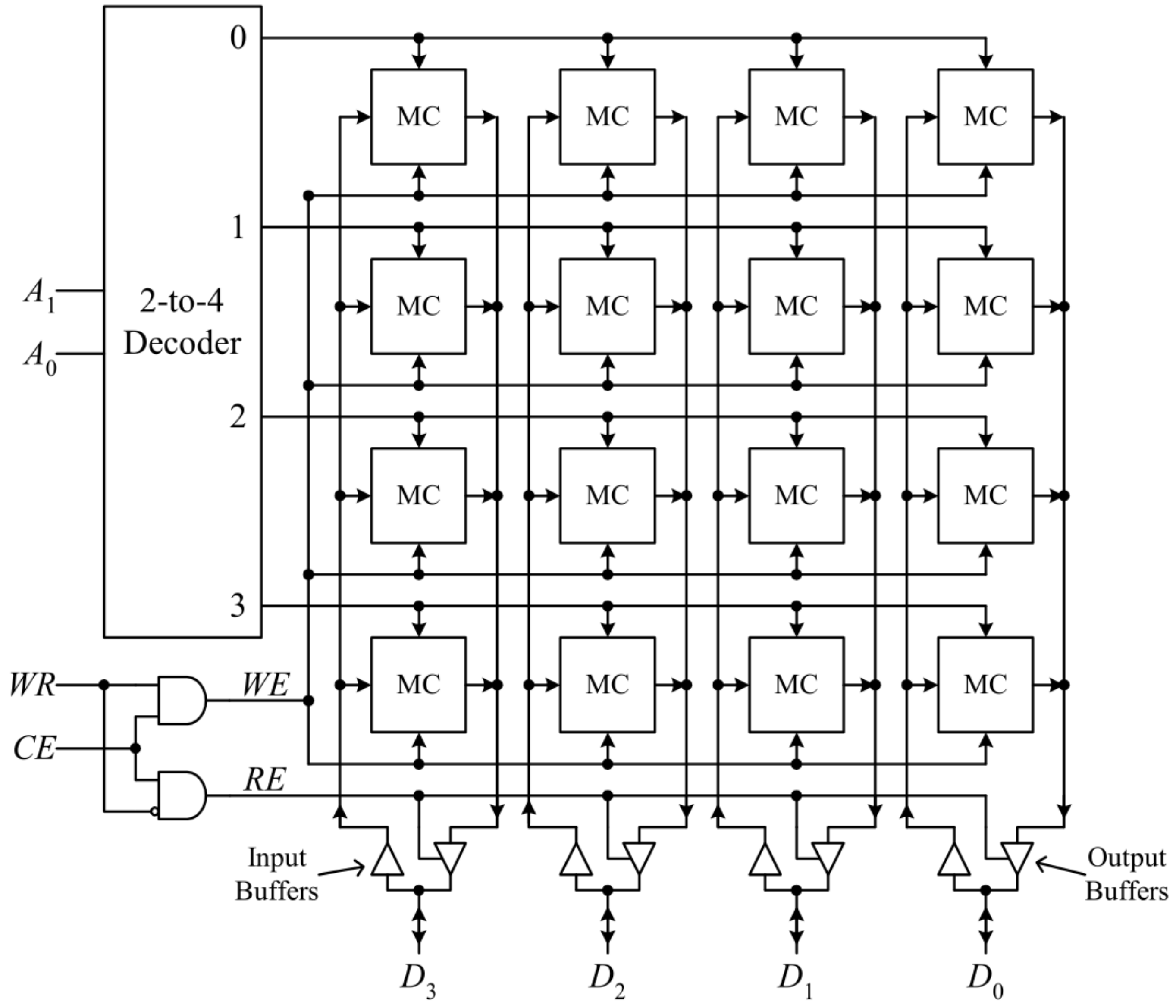
To create a 4 × 4 static RAM chip, we need 16 memory cells forming a 4 × 4 grid.

Each row forms a single storage location, and the number of memory cells in a row determines the bit width of each location.

So all of the memory cells in a row are enabled with the same address.



A 2-to-4 decoder is used to decode the address lines, A_0 and A_1 (four address locations).



Larger memories are constructed from multiple, smaller memory chips.