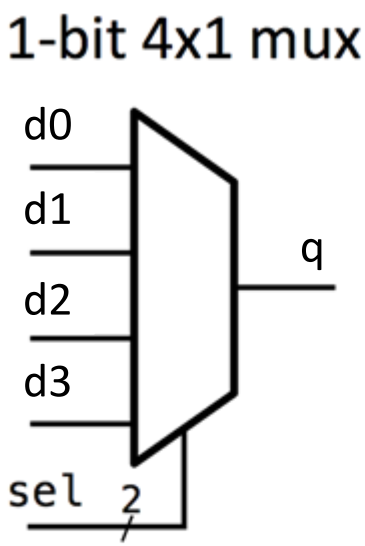
CX1005
Digital Logic

Sequential Circuits in Verilog

Combinational Verilog

- All the Verilog we have looked at so far allows us to implement purely combinational circuits
- It is important to be wary of the rules introduced there to avoid inadvertently writing sequential code that would not function as intended



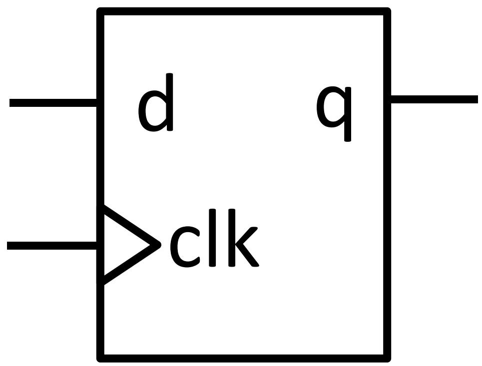

```
module mux4 (output reg q,
             input [3:0] d,
             input [1:0] sel);
   always @* begin
      case (sel)
         2’b00 : q = d[0];
         2’b01 : q = d[1];
         2’b10 : q = d[2];
         2’b11 : q = d[3];
      endcase
   end
endmodule
```

Sequential Verilog

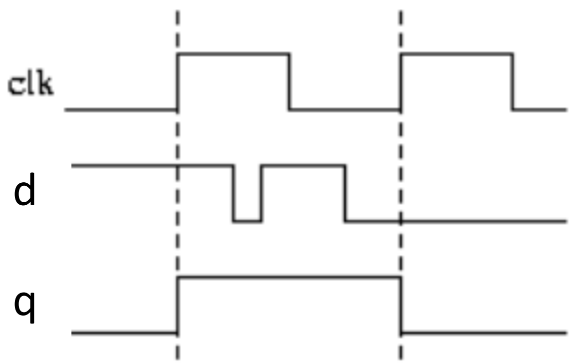
- We will only introduce the basics of sequential Verilog in this course
- The small blocks we show can be incorporated into larger designs using instantiation
- While it is possible to describe general sequential blocks in Verilog, it is generally used to describe ***synchronous*** circuits, i.e. **edge-triggered components**
- Synthesis tools will generally convert designs to D-type flip-flops/registers
- We can design many variations of the basic components

Registers in Verilog

- The basic structure of an edge-sensitive block in Verilog is as follows:



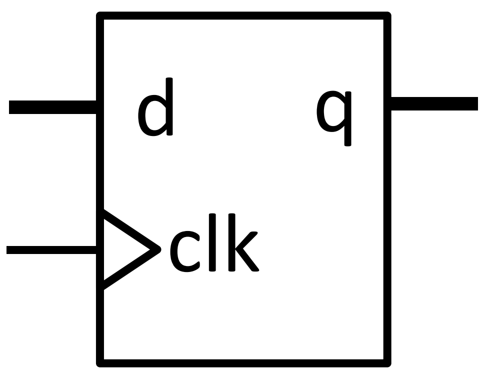

```
module simplereg (input d, clk,
                  output reg q);

always@(posedge clk)
    q <= d;

endmodule
```



- This creates a 1-bit register/D flip-flop with input *d* and output *q*

Registers in Verilog

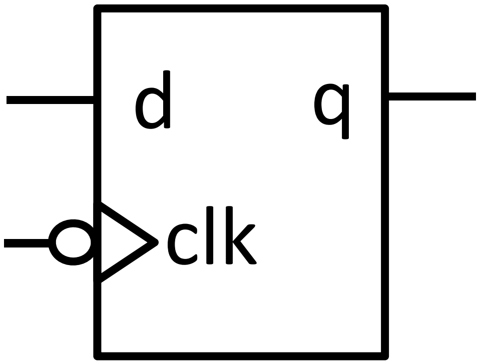
- Note the new **always** block format:
  - For combinational, we list signals, or use always@*
  - For synchronous, we use always@(**posedge** clk)
- This tells the synthesis tools that the block's behavior should only happen at the **clock rising edge**, hence creating a flip-flop/register
- We can create a multi-bit register:




```
module simplereg (input [7:0] d, input clk,
                  output reg [7:0] q);

always@(posedge clk)
    q <= d;

endmodule
```

Clock and Reset

- In the previous examples, we used *clk* for the clock input
- The signal name should be whatever the clock signal is: *i_clk*, *gen_clk*, *clock*, *clk2400*
- If we're naming it, we often just use *clk*
- Remember to add the clock input to your module port list
- All **synchronous always blocks** should use the same clock signal
- For falling-edge triggered, use always@(**negedge** clk)

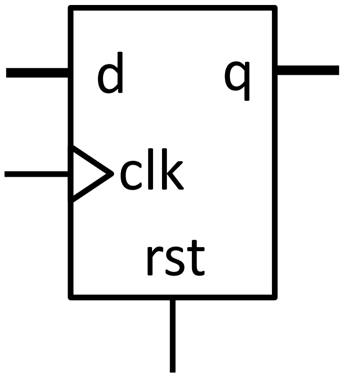


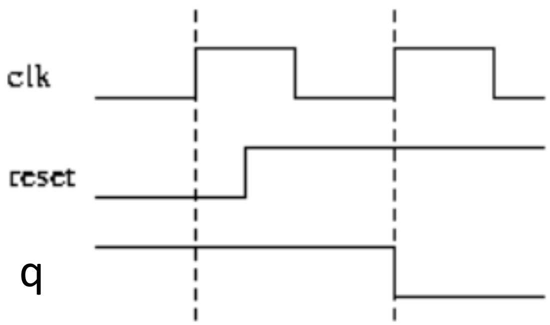
Clock and Reset

- Registers are very useful, but often, we want to be able to *reset* the value in a register
- Two types of reset:
  - ***Asynchronous***: whenever the reset input is asserted, the contents of the register are set to the reset value
  - ***Synchronous***: at a rising edge, if reset is asserted, the contents of the register are set to the reset value
- In modern FPGA design, we use synchronous reset
- Generally, **every** synchronous component should be implemented with a reset

Clock and Reset

- A register with *synchronous* reset:

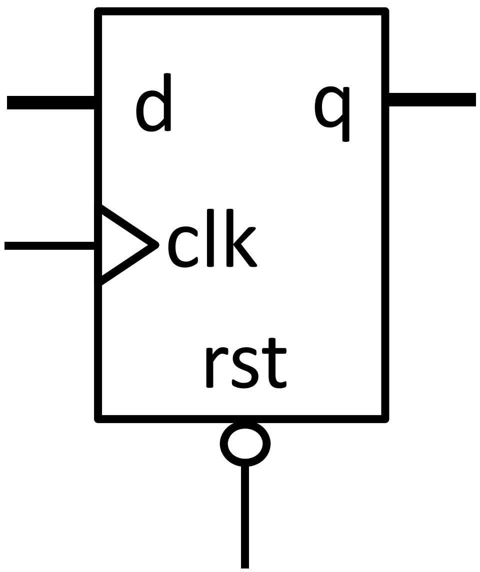





```
module simplereg (input [7:0] d,
                  input clk, rst,
                  output reg [7:0] q);

always@(posedge clk)
begin
    if(rst) // same as (rst==1'b1)
        q <= 8'b0000_0000;
    else
        q <= d;
end

endmodule
```

Clock and Reset

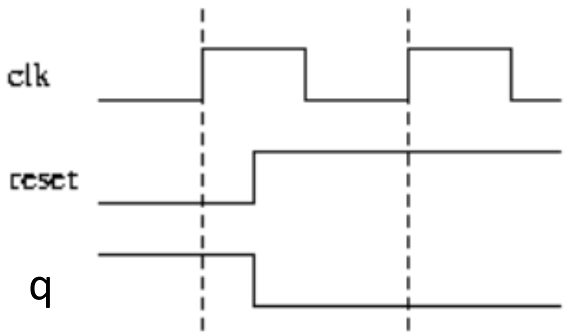
- Sometimes the reset is *active low*:




```
module simplereg (input [7:0] d,
                  input clk, rst,
                  output reg [7:0] q);

always@(posedge clk)
begin
    if(!rst)
        q <= 8'b0000_0000;
    else
        q <= d;
end

endmodule
```

Clock and Reset

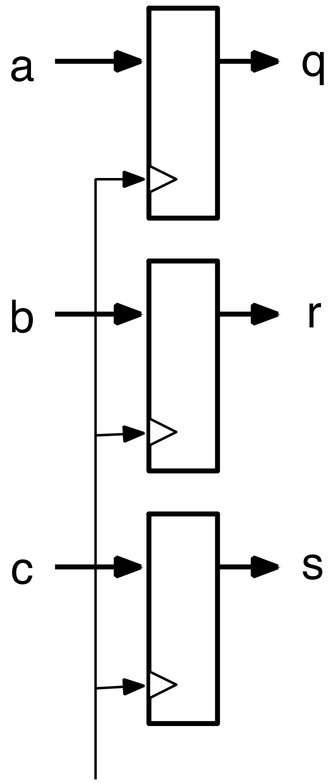
- For an *asynchronous* reset, we would need to add the reset signal to the sensitivity list:




```
module simplereg (input [7:0] d,
                  input clk, rst,
                  output reg [7:0] q);

always@(posedge clk or posedge rst)
begin
    if(rst)
        q <= 8'b0000_0000;
    else
        q <= d;
end

endmodule
```

Registers in Verilog

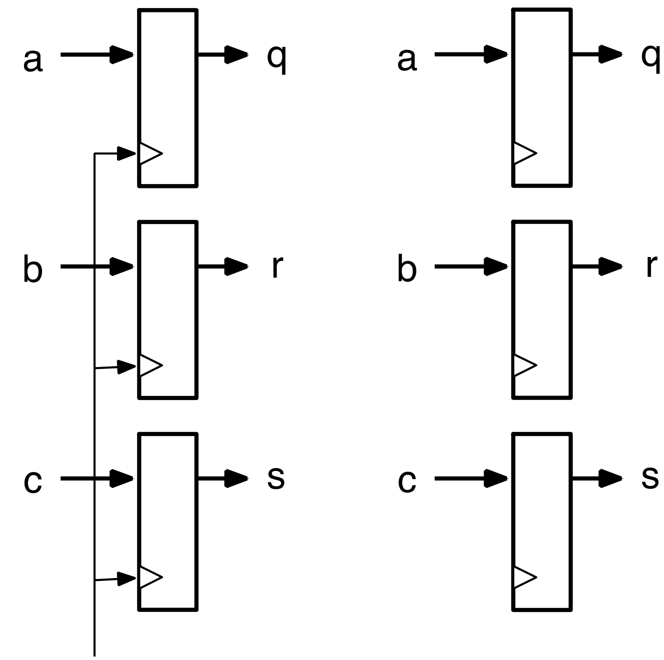
- We can create multiple registers by including multiple assignments
- Each *assignment* in a synchronous always block results in a *register*




```
module multireg (input [7:0] a, b, c,
                 input clk, rst,
                 output reg [7:0] q, r, s);

always@(posedge clk)
begin
    if(!rst) begin
        q <= 8'b0000_0000;
        r <= 8'b0000_0000;
        s <= 8'b0000_0000;
    end else begin
        q <= a;
        r <= b;
        s <= c;
   end
end

endmodule
```

Registers in Verilog

- Even if registers are not connected, we can combine them in the same always block
- Each assignment in an always @ posedge clk block results in a register
- Remember, we **always** have a reset for registers
- We usually leave out the reset wires, and sometimes the clk wires in diagrams



Assignments in Always Blocks

- You may have noticed we are using a new assignment operator: <=
- This is called a *non-blocking* assignment
- For ***combinational*** always blocks, we *always* use a blocking assignment (=), and **order matters**
- For ***synchronous*** always blocks, we *always* use non-blocking assignments (<=), and **order does not matter**
- This explanation is sufficient for this course

Synchronous Components

- Registers are useful for storing values, and organizing the timing in a circuit
- There are a number of other basic sequential blocks we can use within our designs
  - Counters
  - Shift Registers
  - Serial-to-Parallel and Parallel-to-Serial Converters
  - Memories

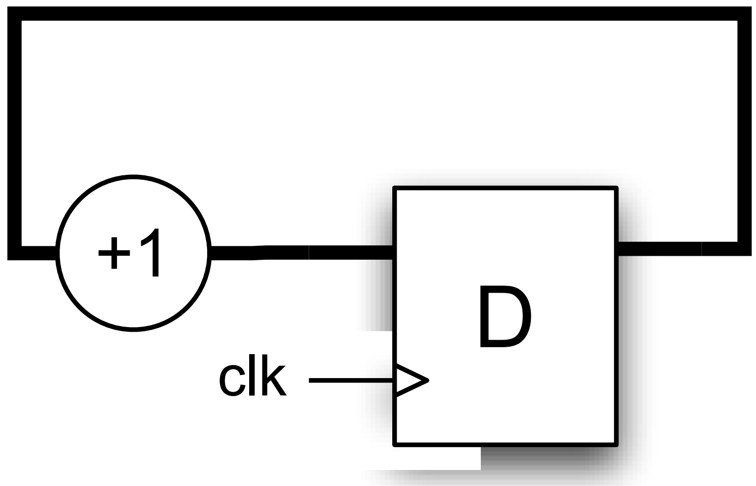
Binary Counters

- A *binary counter* is a circuit that outputs an increasing output value in each clock cycle
- Consider the count sequence of a 3-bit counter:

0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

Synchronous Counters

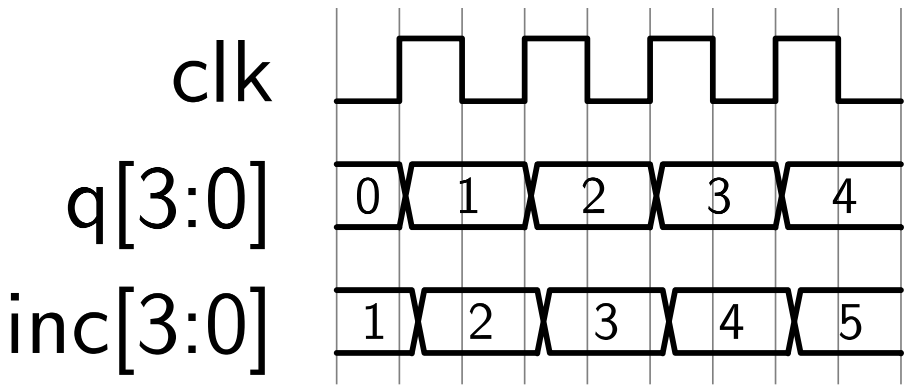
- There is another way to think about counters that may be closer to what we would do in Verilog:

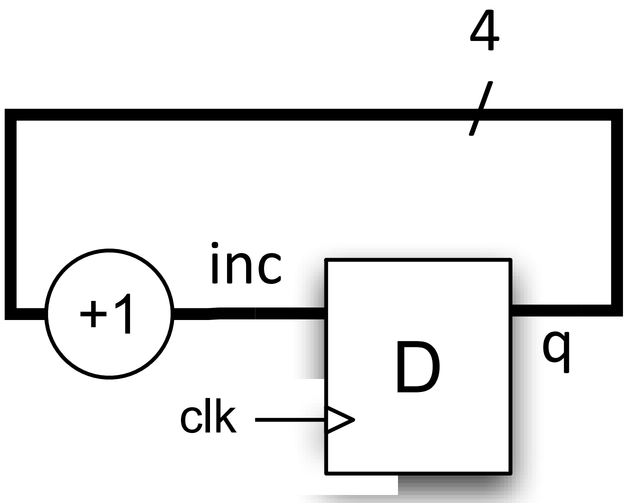


- At each rising edge, we pass through the incremented value of the current count
- The data width can be any number of bits

Synchronous Counters

- Timing diagram helps:

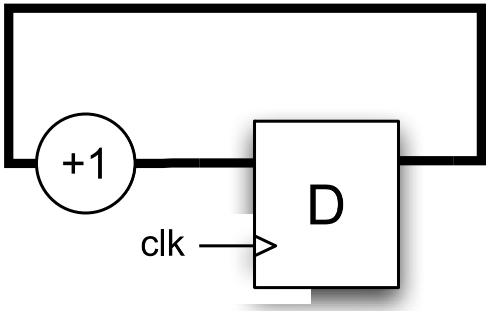




- At each clock edge, the incremented value, *inc*, derived from the current output, is passed to *q*

Synchronous Counters in Verilog

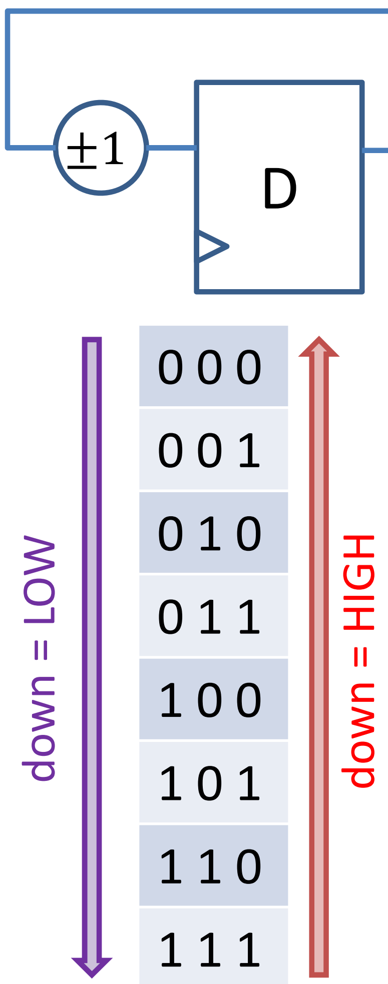
- We can describe such a counter in Verilog as follows:




```
module simplecnt (input clk, rst,
                  output reg [3:0] q);

always@(posedge clk)
begin
    if(rst)
        q <= 4'b0000;
    else
        q <= q + 1'b1;
end

endmodule
```

Synchronous Counters in Verilog

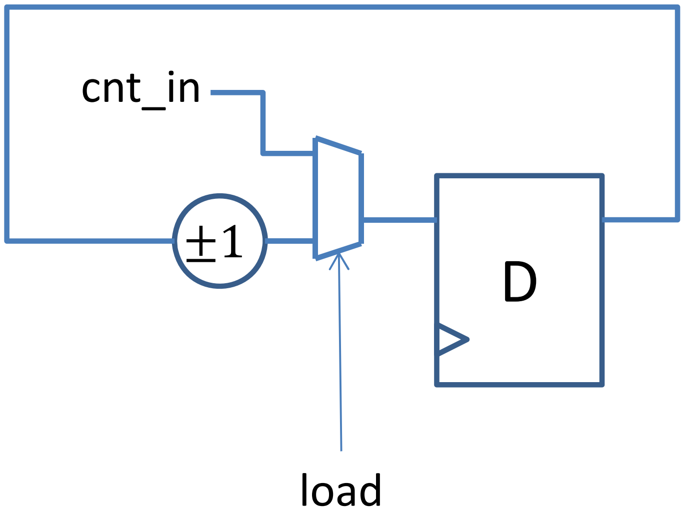
- We can extend the capabilities of our counter quite easily; here, an up-down counter:




```
module simplecnt (input clk, rst, down,
                  output reg [3:0] q);

always@(posedge clk)
begin
    if(rst)
        q <= 4'b0000;
    else
        if(down)
            q <= q – 1'b1;
        else
            q <= q + 1'b1;
end

endmodule
```

Synchronous Counter in Verilog

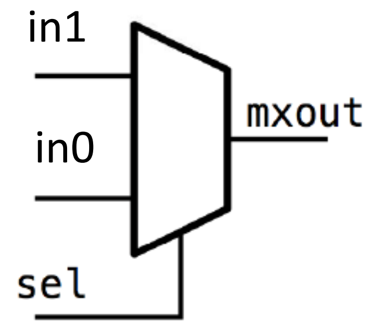
- The important thing to remember, is the *q* output is only updated at the rising edge, so when we use its value inside the block, it is the **old** value
- What if we want to be able to load a custom value into the counter?
  - load input: when high, the counter takes its value from cnt_in
  - cnt_in: custom load value
  - Hence, should load if required, otherwise count up or down depending on the down input

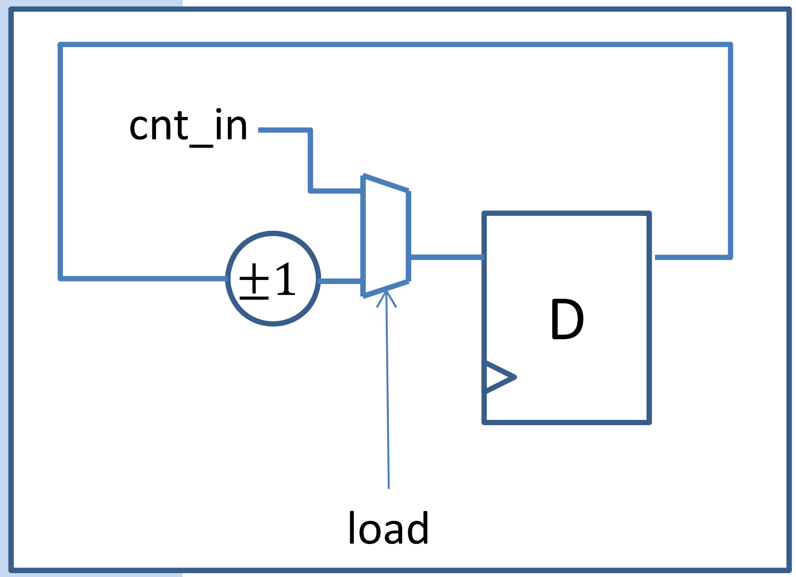


Synchronous Counters in Verilog

```
module simplecnt (input clk, rst,
                  input down, load,
                  input [3:0] cnt_in,
                  output reg [3:0] q);

always@(posedge clk)
begin
    if(rst)
        q <= 4'b0000;
    else
        if(load)
            q <= cnt_in;
        else
            if(down)
                q <= q - 1'b1;
            else
                q <= q + 1'b1;
end
endmodule
```

1-bit 2x1 mux

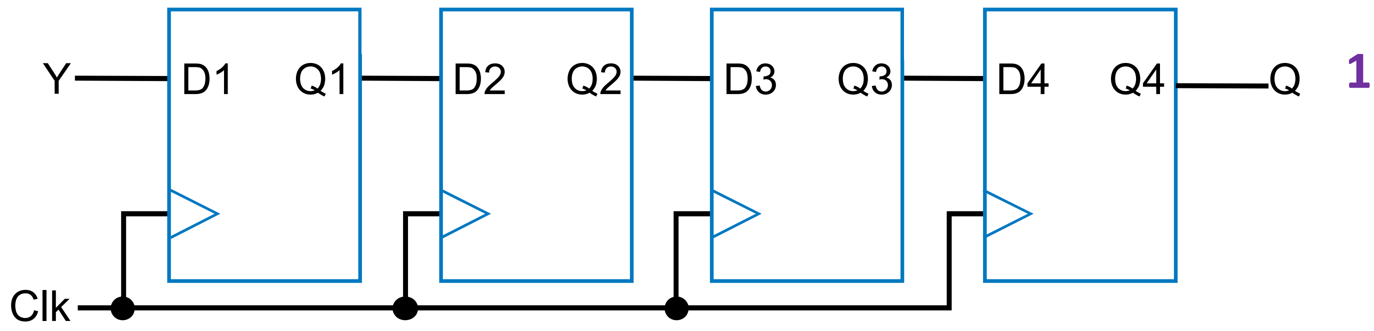




Note: Synthesis tool may not instantiate a multiplexer

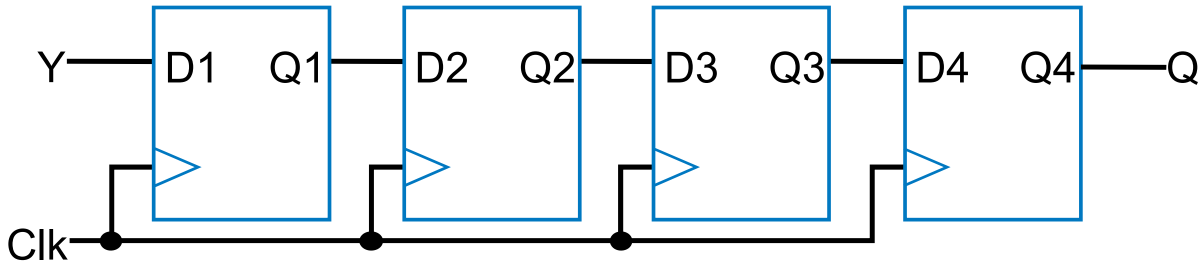
Shift Registers in Verilog

- Shift registers take a single input and pass it through a chain of flip-flops
- At each clock cycle, the input progresses one stage



- We need internal signals to connect the intermediate registers, must be declared as **reg** type
- We then write a single assignment for each register:

Shift Registers in Verilog




```verilog
module shiftreg (input clk, y,
                 output reg q);

reg q1, q2, q3;

always@(posedge clk)
begin
    q1 <= y;
    q2 <= q1;
    q3 <= q2;
    q  <= q3;
end

endmodule
```

```verilog
module shiftreg (input clk, y,
                 output reg q);

reg q1, q2, q3;

always@(posedge clk)
begin
    q2 <= q1;
    q1 <= y;
    q  <= q3;
    q3 <= q2;
end

endmodule
```

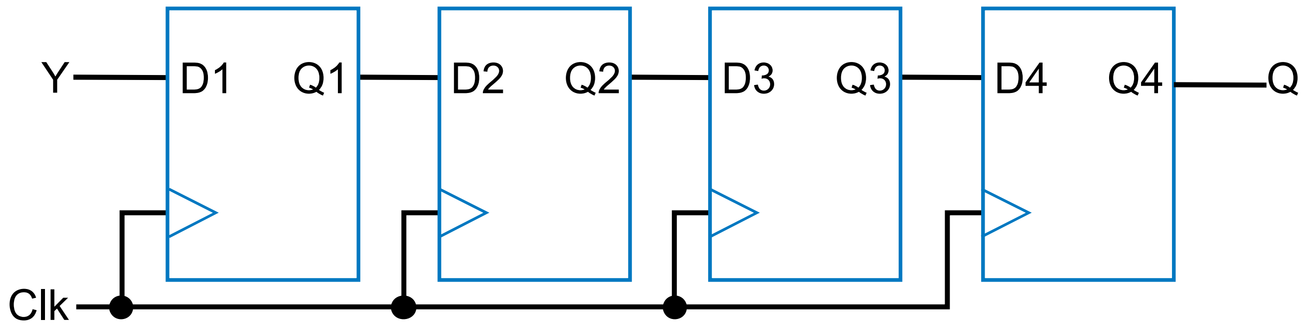
What if I change the order of the assignments?

Shift Registers in Verilog

- What if I change the order of the assignments?
  - The circuit will function identically
  - Each assignment is a register, and that assignment only occurs on the rising edge
  - Hence, they all transfer their input just **before** the rising edge to their output just **after** the rising edge
  - It would take 4 clock cycles for an input value to reach the final output in that example

Shift Registers in Verilog

- We can use vectors to make the code easier:



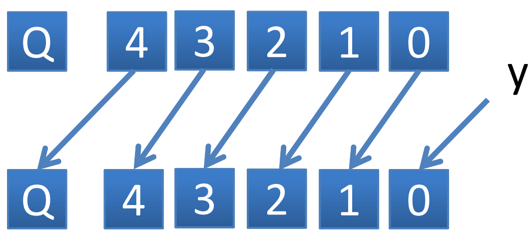

```
module shiftreg (input clk, y,
                 output reg q);

reg q1, q2, q3;

always@(posedge clk)
begin
    q1 <= y;
    q2 <= q1;
    q3 <= q2;
    q  <= q3;
end

endmodule
```

```
module shiftreg (input  clk, y,
                 output reg q_out);

reg [3:1] q;

always@(posedge clk)
begin
    q[1]   <= y;
    q[3:2] <= q[2:1];
    q_out  <= q[3];
end

endmodule
```

Shift Registers in Verilog

- So now it's easy to extend this to more flip-flops:

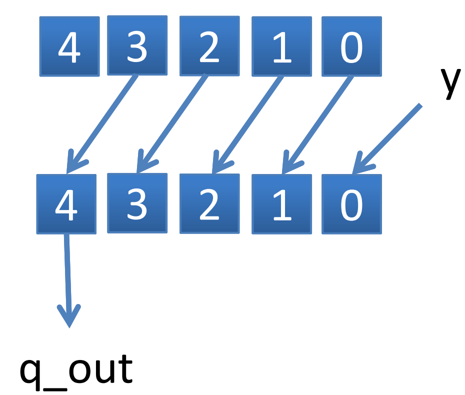
```
module shiftreg (input clk, y,
                 output reg Q);

reg [4:0] q;

always@(posedge clk)
begin
    q[0]   <= y;
    q[4:1] <= q[3:0];
    Q      <= q[4];
end

endmodule
```



More Shift Registers

- We can now think of the contents as a single word
- We now hard wire the output to the MSB

```
module shiftreg (input clk, y,
                 output q_out);

reg [4:0] q;

always@(posedge clk)
begin
    q[0]   <= y;
    q[4:1] <= q[3:0];
end

assign q_out = q[4];

endmodule
```



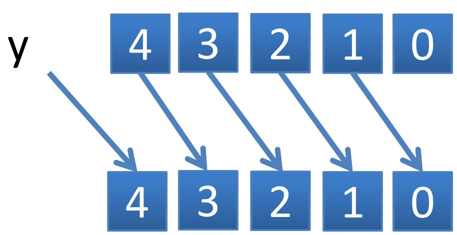
More Shift Registers

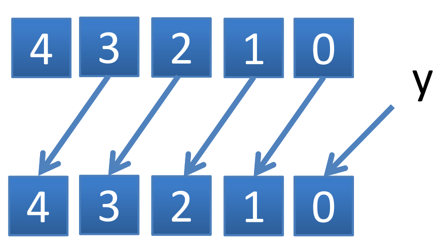
- What if we want to only shift sometimes?
- Now, a shift will only occur when the *sh* input is high

```
module shiftreg2 (input clk, y, sh,
                  output q_out);

reg [4:0] q;

always@(posedge clk)
begin
    if (sh) begin
        q[0]   <= y;
        q[4:1] <= q[3:0];
    end
end

assign q_out = q[4];

endmodule
```

More Shift Registers

- We can also add the capability to shift in the other direction
- When the *rt* input is high, the new sample is registered at the MSB, and the rest shift right

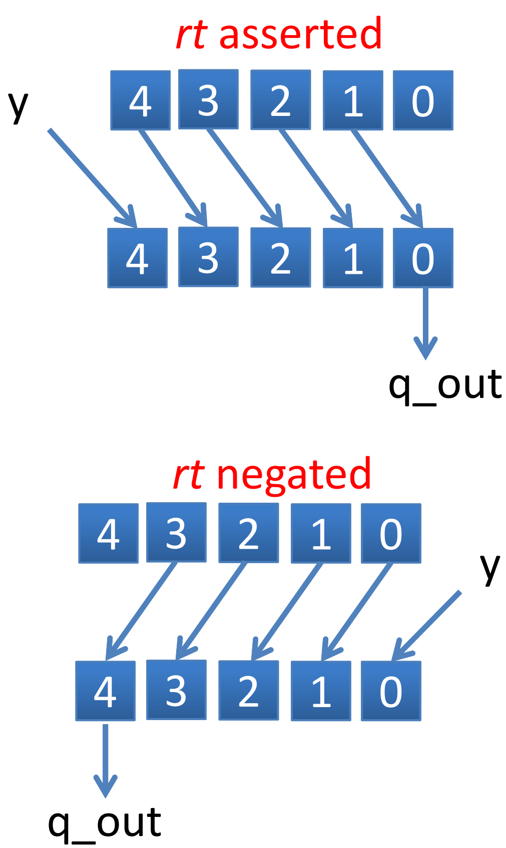





```
module shiftreg3 (input clk, y, sh, rt,
                  output q_out);

reg [4:0] q;

always@(posedge clk)
begin
    if (sh) begin
        if (rt) begin
            q[4] <= y; q[3:0] <= q[4:1];
        end else begin
            q[0] <= y; q[4:1] <= q[3:0];
        end
    end
end

assign q_out = rt ? q[0] : q[4];

endmodule
```

More Shift Registers

- We might want to be able to access the whole word



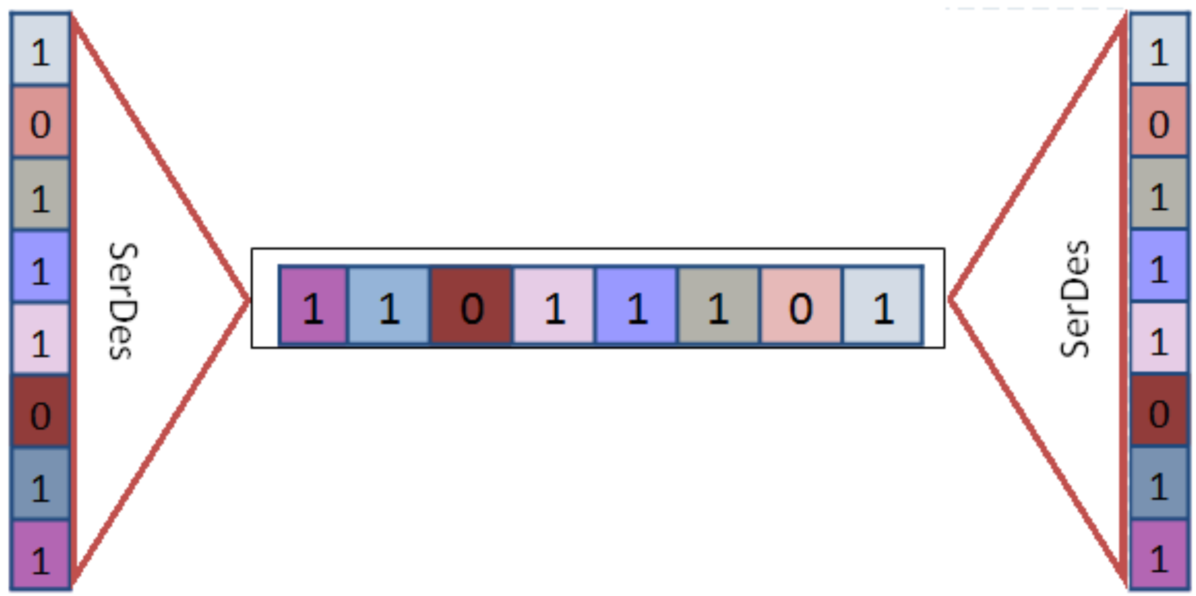

```
module shiftreg4 (input clk, y, sh, rt,
                  output q_out,
                  output [4:0] q_word);

reg [4:0] q;

always@(posedge clk)
begin
    if (sh) begin
        if (rt) begin
            q[4] <= y; q[3:0] <= q[4:1];
        end else begin
            q[0] <= y; q[4:1] <= q[3:0];
        end
    end
end

assign q_out = rt ? q[0] : q[4];
assign q_word = q;

endmodule
```

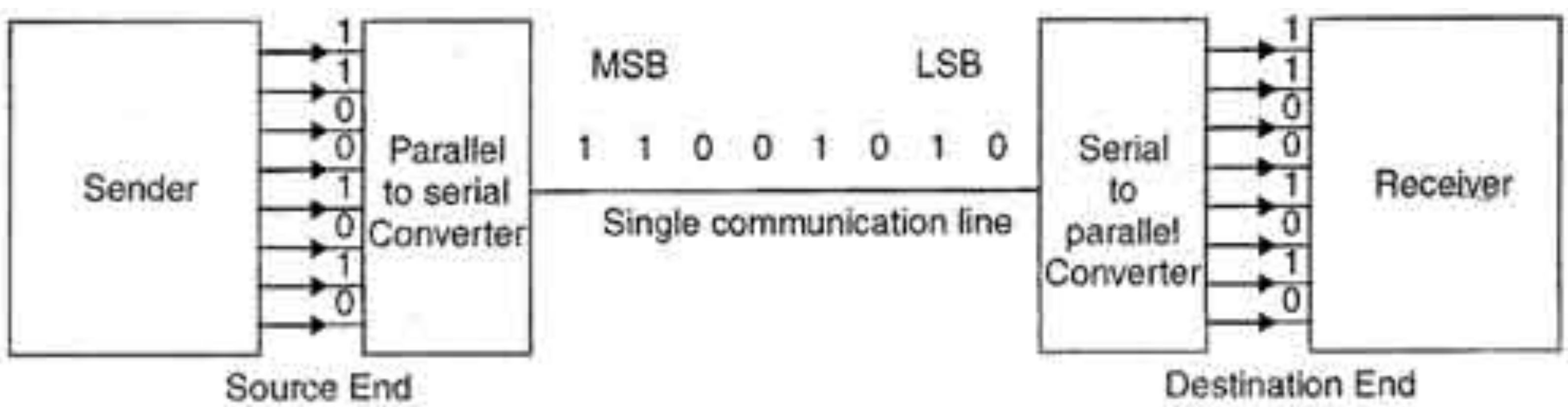
More Shift Registers

- Shift registers can also have a load input
- Whenever *ld* is high, the shift register is loaded with the value on *ld_val*

```
module shiftreg5 (input clk, y, sh, rt, ld,
                  input [4:0] ld_val,
                  output q_out,
                  output [4:0] q_word);
reg [4:0] q;

always@(posedge clk)
begin
    if (ld) q <= ld_val; else
    if (sh) begin
        if (rt) begin
            q[4] <= y; q[3:0] <= q[4:1];
        end else begin
            q[0] <= y; q[4:1] <= q[3:0];
        end
    end
end

assign q_out = rt ? q[0] : q[4];
assign q_word = q;

endmodule
```

Serial Data Transfer

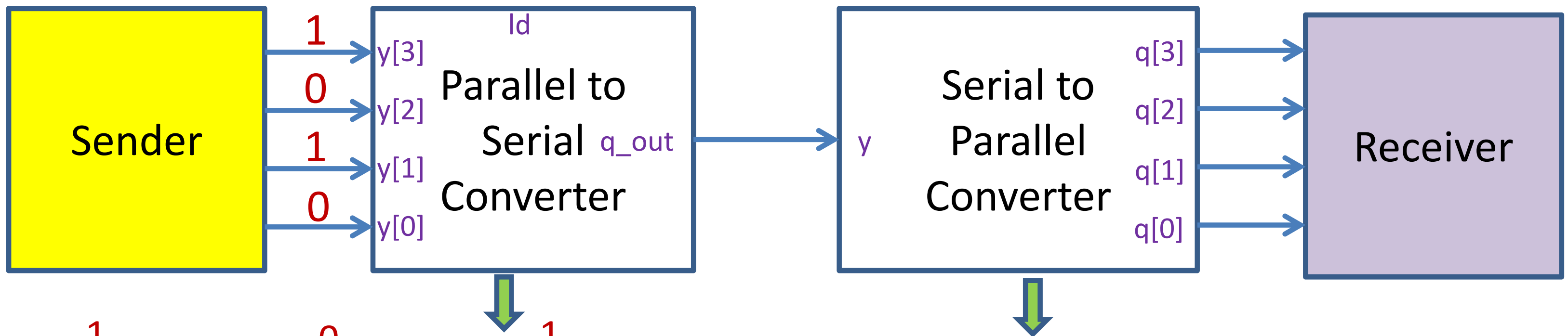


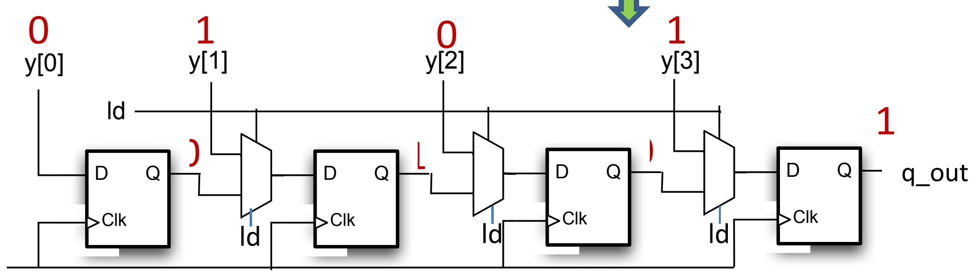


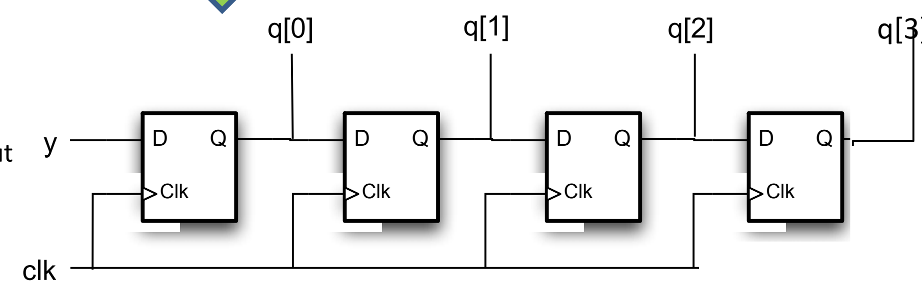
Source:
http://ecomputernotes.com/computernetworkingnotes/communication-networks/data-transmission

Serial transmission

Serial Data Transfer

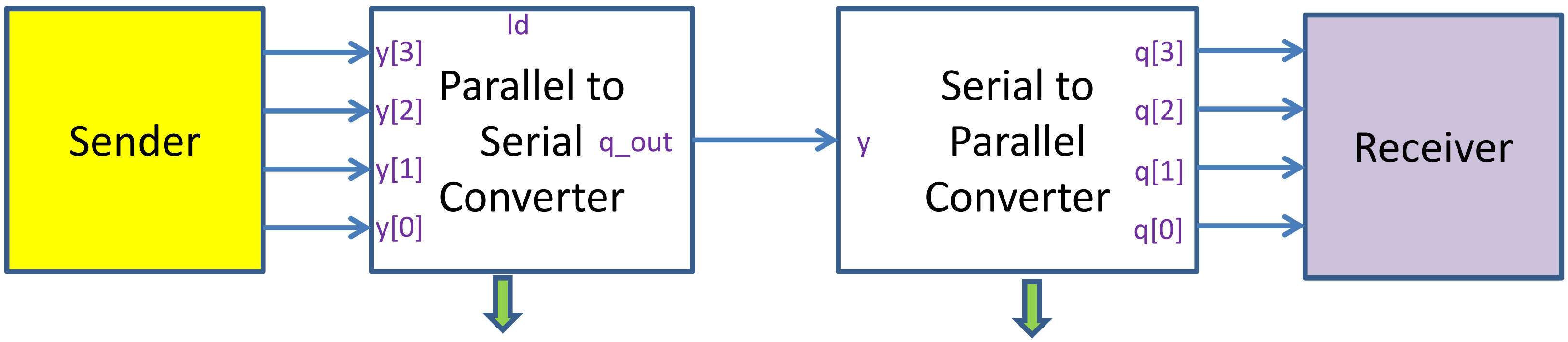


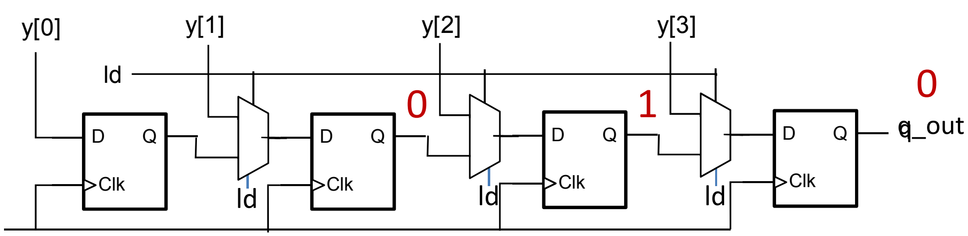


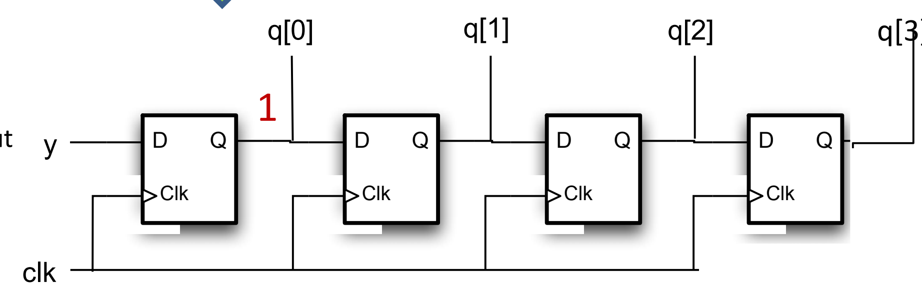



```
module piso4 (input clk, ld,
              input [3:0] y,
              output q_out);
reg [3:0] q;

always@(posedge clk)
begin
   if (ld) q <= y;
   else begin
      q[0]   <= y[0];
      q[3:1] <= q[2:0];
   end
end
assign q_out = q[3];

endmodule
```

Serial Data Transfer

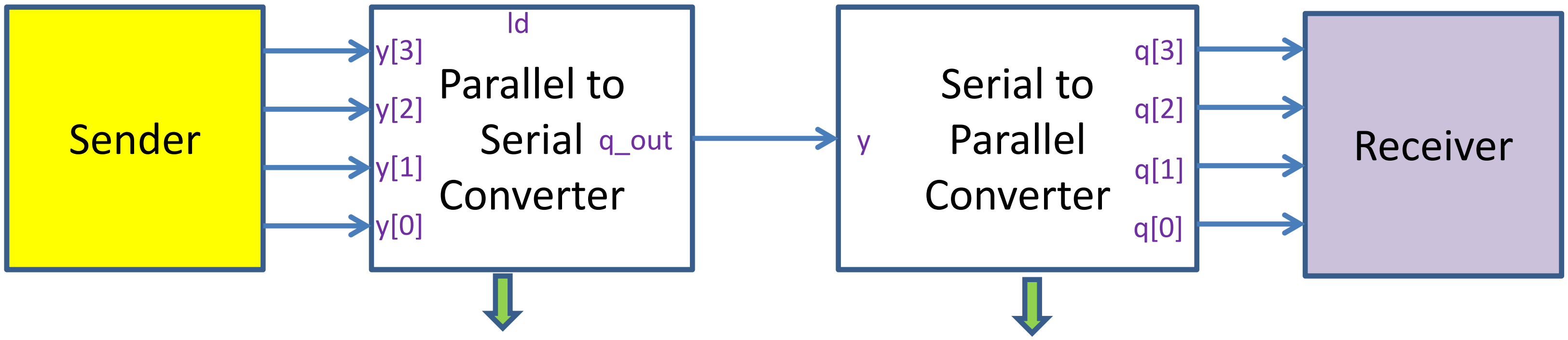


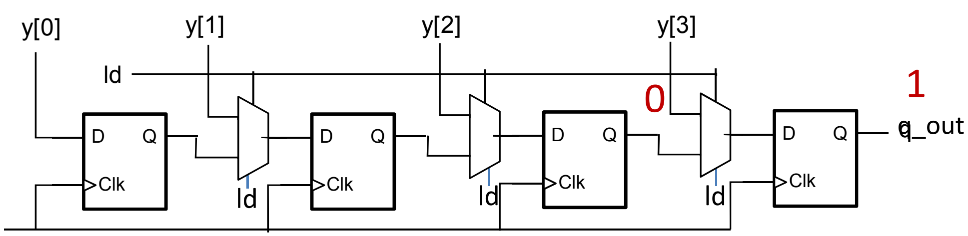


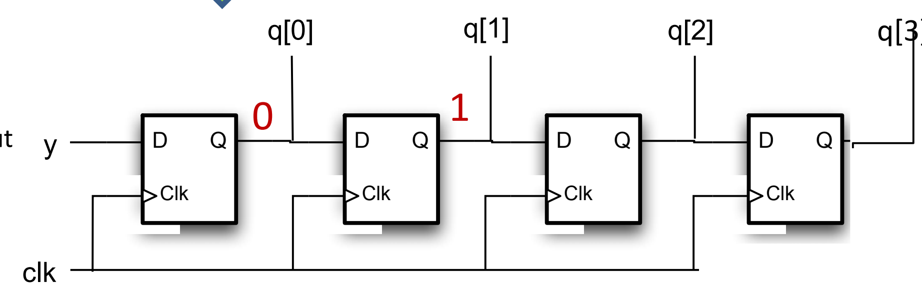



```
module piso4 (input clk, ld,
              input [3:0] y,
              output q_out);
reg [3:0] q;

always@(posedge clk)
begin
    if (ld) q <= y;
    else begin
        q[0]   <= y[0];
        q[3:1] <= q[2:0];
    end
end
assign q_out = q[3];

endmodule
```

Serial Data Transfer

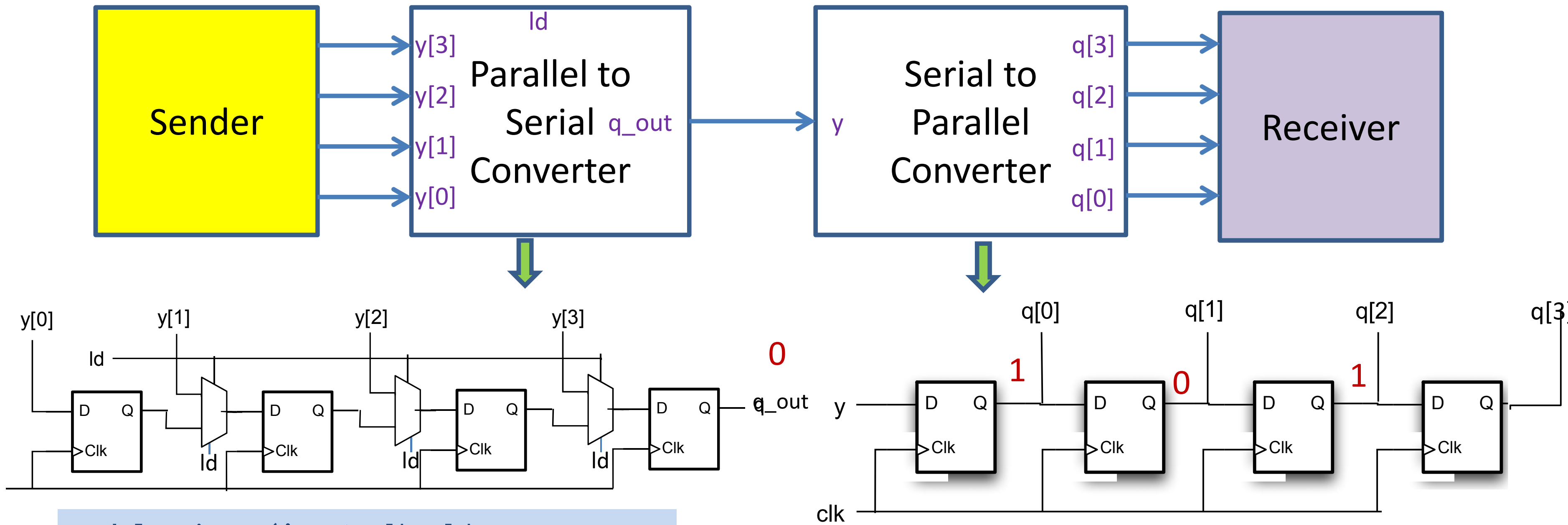






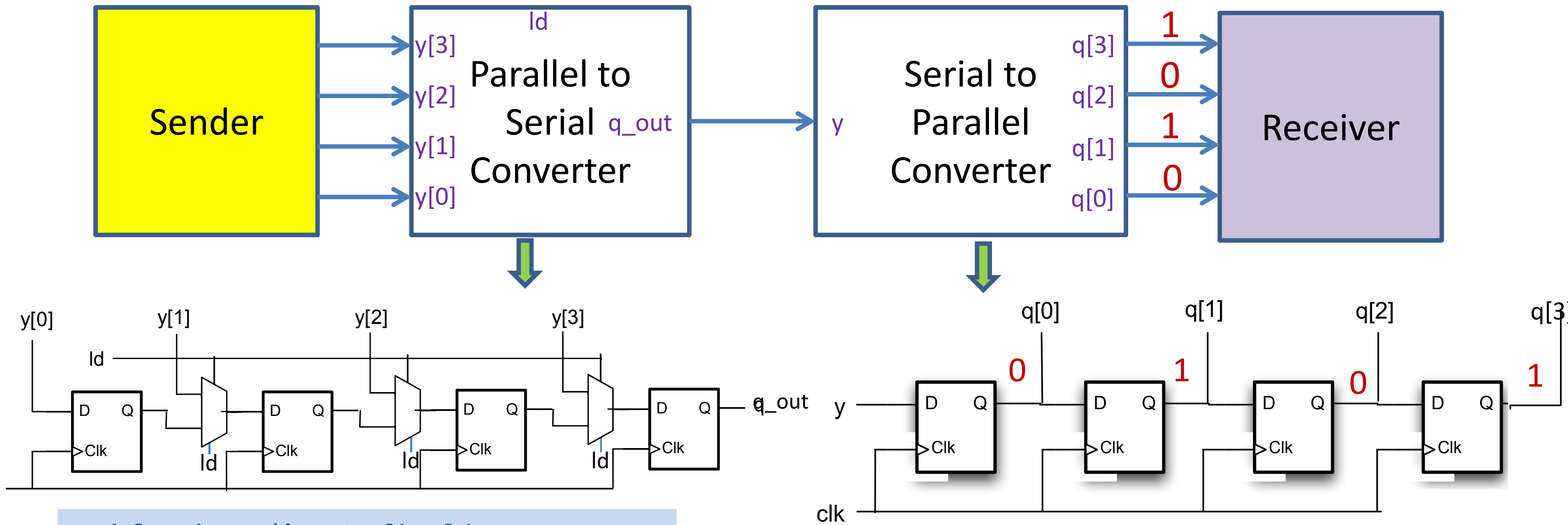

```
module piso4 (input clk, ld,
              input [3:0] y,
              output q_out);
reg [3:0] q;

always@(posedge clk)
begin
   if (ld) q <= y;
   else begin
      q[0]   <= y[0];
      q[3:1] <= q[2:0];
   end
end
assign q_out = q[3];

endmodule
```

Serial Data Transfer




```
module piso4 (input clk, ld,
              input [3:0] y,
              output q_out);
reg [3:0] q;

always@(posedge clk)
begin
    if (ld) q <= y;
    else begin
        q[0]   <= y[0];
        q[3:1] <= q[2:0];
    end
end
assign q_out = q[3];

endmodule
```

Serial Data Transfer



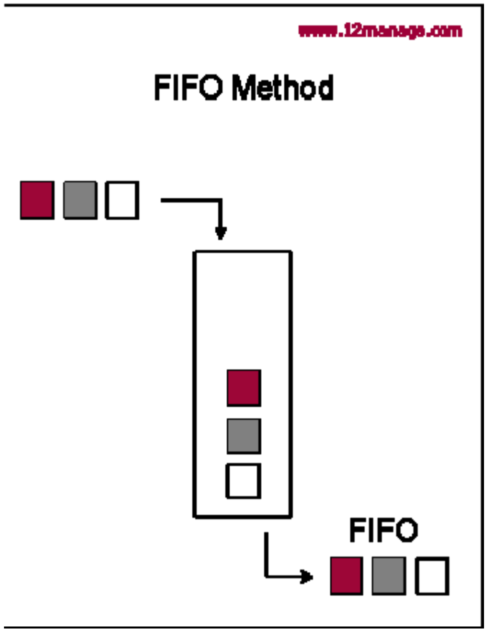

```
module piso4 (input clk, ld,
              input [3:0] y,
              output q_out);
reg [3:0] q;

always@(posedge clk)
begin
    if (ld) q <= y;
    else begin
        q[0]   <= y[0];
        q[3:1] <= q[2:0];
    end
end
assign q_out = q[3];

endmodule
```

```
module sipo4 (input clk, y,
              output reg [3:0] q);
always@(posedge clk)
begin
    q[0]   <= y;
    q[3:1] <= q[2:0];
end

endmodule
```

First-In-First-Out Buffers

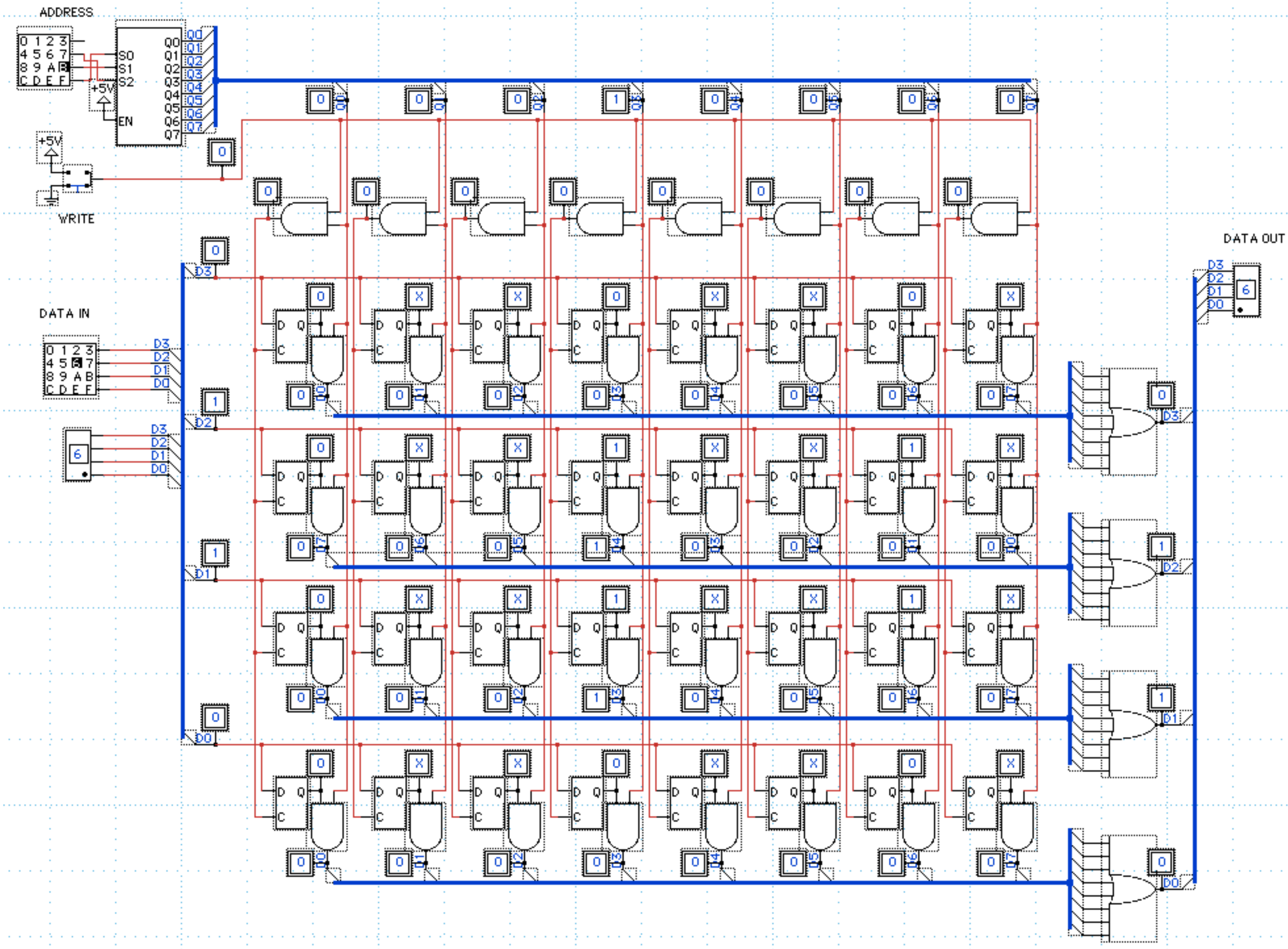
- When we build shift register structures for multi-bit signals, we typically refer to them as FIFOs (First-In-First-Out)
- They are useful for managing the flow of data when building datapaths (later)
- The structure is identical to shift registers, but now each register is multi-bit
- The input, output, and internal signal widths should match:

```
module fifo4 (input clk, input [3:0] y,
              output reg [3:0] q);

reg [3:0] q1, q2, q3;

always@(posedge clk)
begin
    q1 <= y;
    q2 <= q1;
    q3 <= q2;
    q  <= q3;
end

endmodule
```



Memories

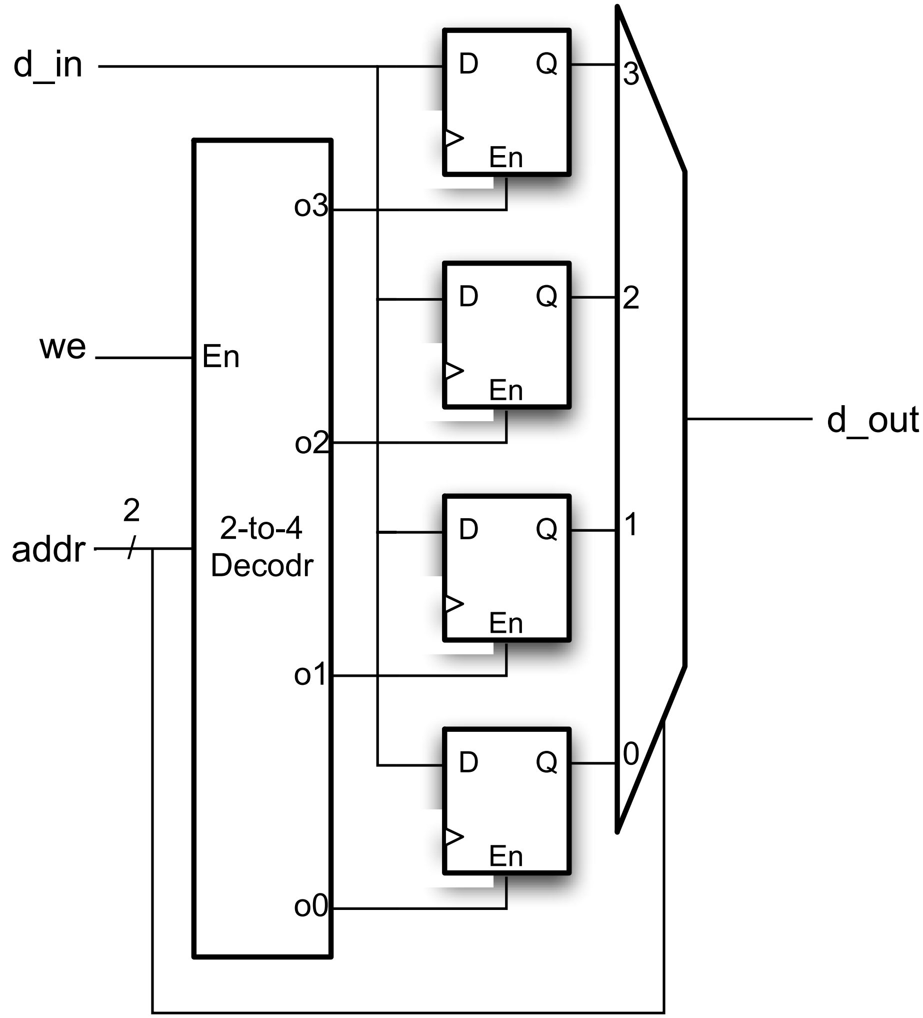
- Memories consists of an array of storage elements
- You can think of a register as one storage element
- Each storage element has a unique address
- We should be able to select which register to store to using an address
- We should be able to read the value out of any register, again using the address



ADDRESS
WRITE
DATA IN
DATA OUT
+5V
S0
S1
S2
EN
Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7
D3
D2
D1
D0

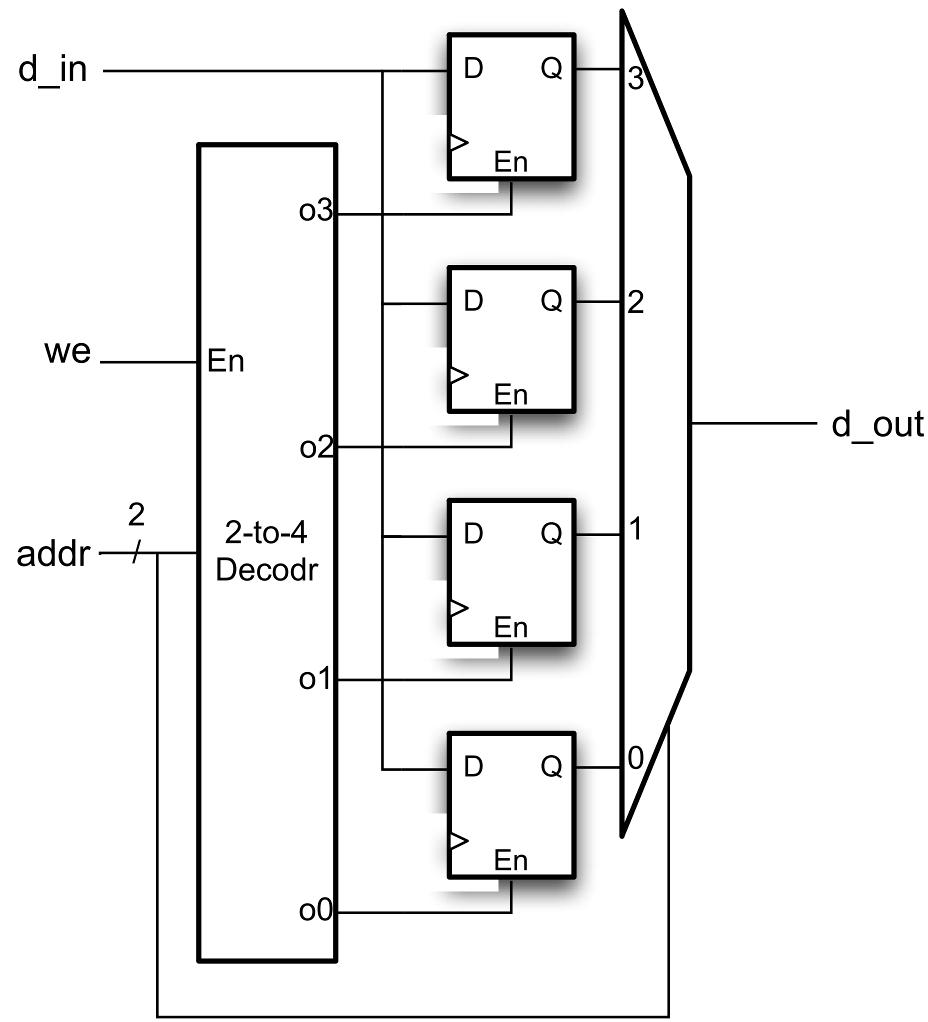
Memories

- In order to store a value, we must assert *we* (write enable) and provide an address
- The decoder outputs a 1 to enable the corresponding register to store whatever is on *d_in* at the clock edge



Memories

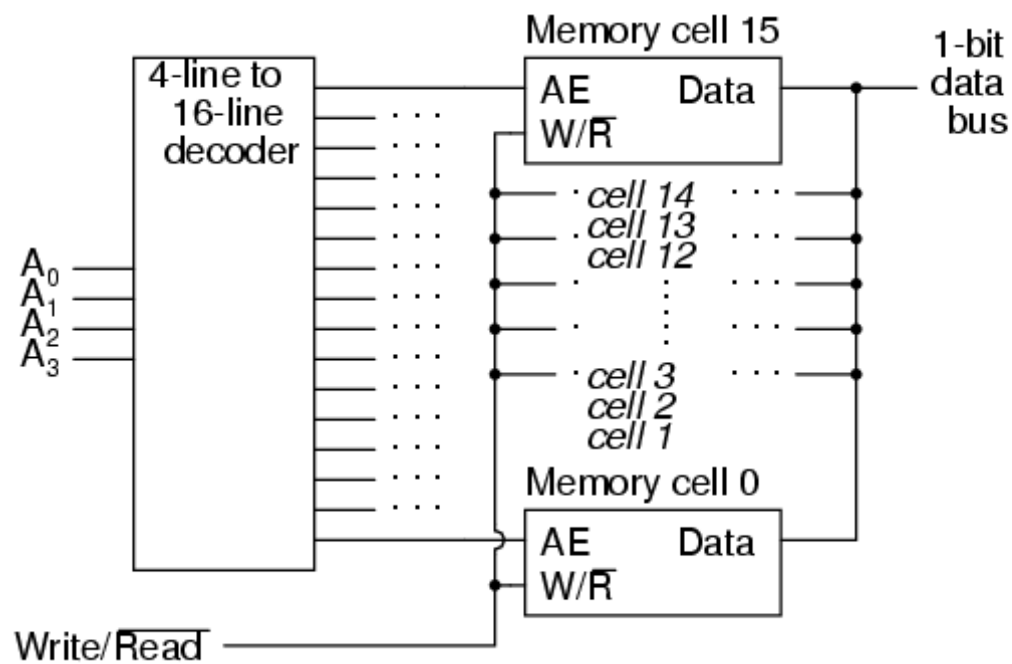
- This basic structure shows one possible arrangement
- The *addr* signal selects which location we want to read from or write to
- For reading, a mux connects the corresponding register to the *d_out* output



Memories

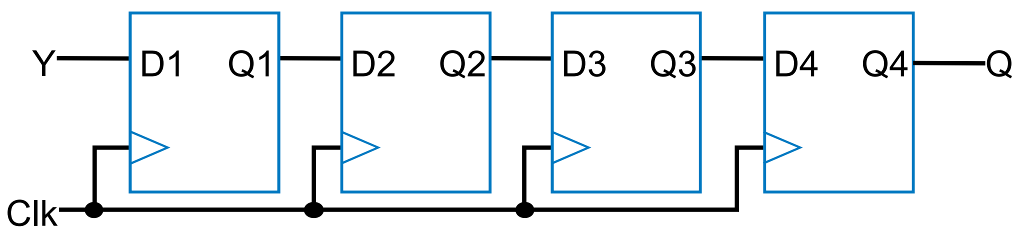
- Many structures exist for addressable memories
- Standard signal names
  - *we* : write enable
  - *re*: read enable
  - *addr*: address
  - *ce* : chip enable
  - *ae*: address enable

16 x 1 bit memory



What Gets Synthesized?

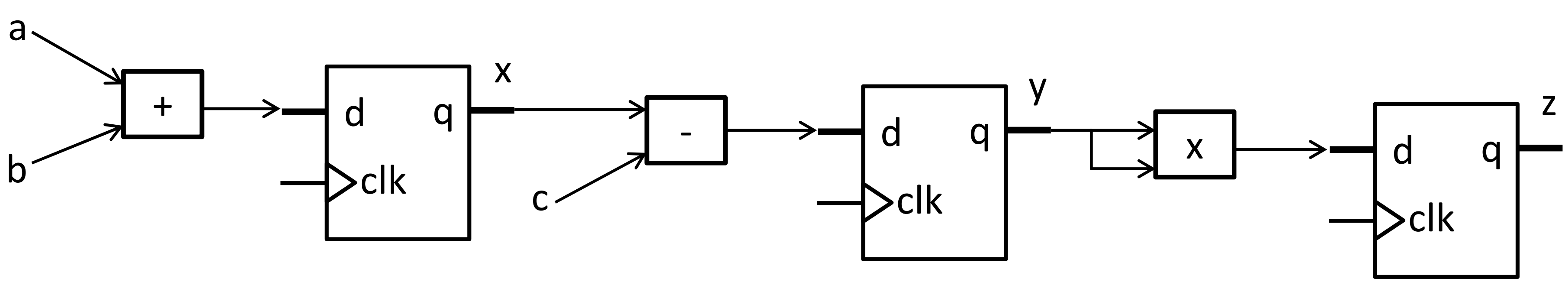
- Important to understand what is synthesized from a synchronous always block
- Each (non-blocking) assignment results in a register, with its input connected to a circuit based on the right hand side, and the output connected to the signal the left hand side
- **Every assignment is a register (i.e. synchronous)**




```
module shiftreg (input clk, y,
                 output reg q);
reg q1, q2, q3;
always@(posedge clk)
begin
    q1 <= y;
    q2 <= q1;
    q3 <= q2;
    q  <= q3;
end
endmodule
```

What Gets Synthesized?

```
always@(posedge clk)
begin
    x <= a + b;
    y <= x – c;
    z <= y * y;
end
```

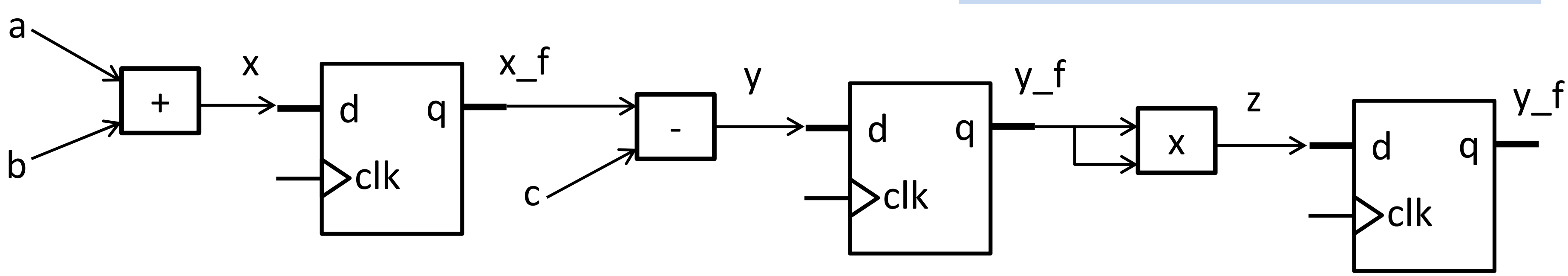


What Gets Synthesized?

- Some prefer to code with the combinational and synchronous aspects separated

```
always@(posedge clk)
begin
    x <= a + b;
    y <= x – c;
    z <= y * y;
end
```

```
always@(posedge clk)
begin
    x_f <= x;
    y_f <= y;
    z_f <= z;
end

always@*
begin
    x = a + b;
    y = x_f – c;
    z = y_f * y_f;
end
```



What Gets Synthesized?

- The important thing is to make your code readable
- Breaking it down into smaller statements makes it easier to debug
- Remember when reading code:
  - The values "read" – i.e. on the right hand side – are considered **just before** the rising edge
  - The values "written" – i.e. on the left hand side – are only updated **just after** the rising edge
- Remember you cannot assign to a signal from more than one place!