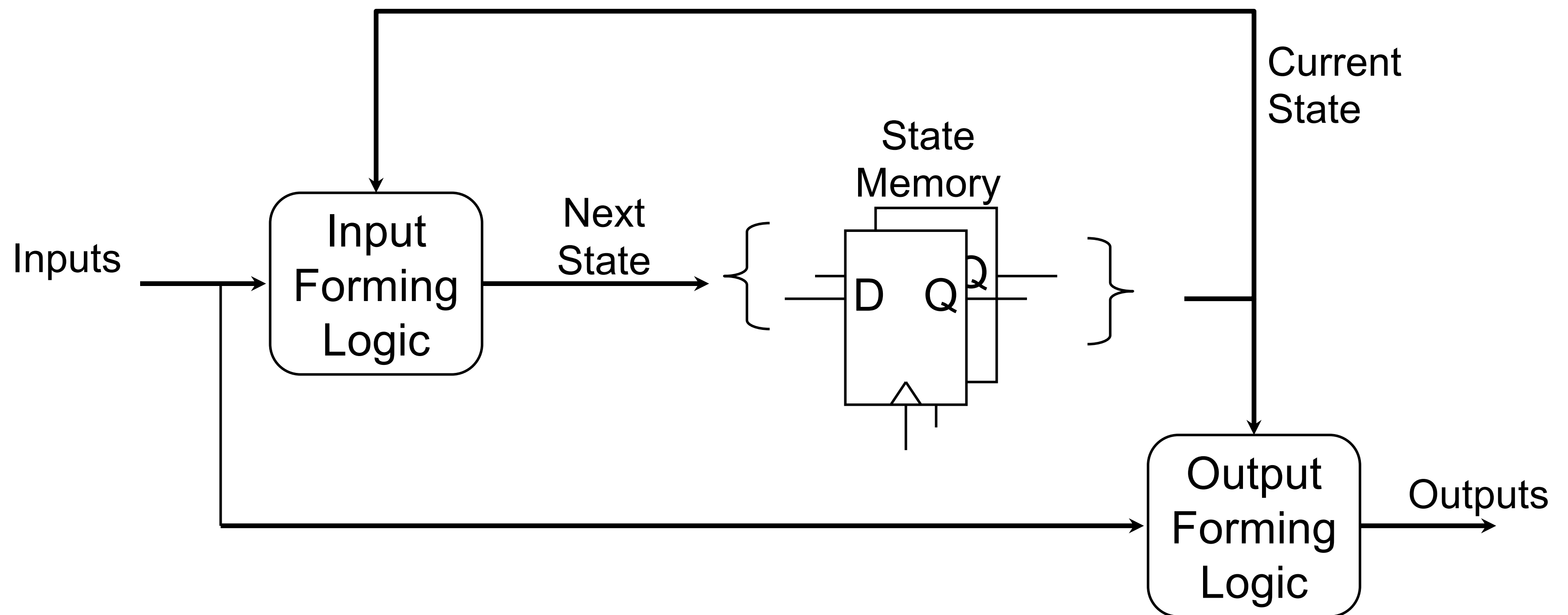
Register Transfer Level (RTL)

- Outputs are a function of current state ***and*** inputs



© 2018 BYU

The always_ff Block

- Contains the input forming logic for the flip flops you are inferring.
- Uses a sensitivity list to describe when the flip flops load the new values
  - always_ff @(posedge clock)
    - Body of always block will not evaluate until a positive edge of the clock (positive edge FFs)
  - always_ff @(negedge clock)
    - Negative edge triggered flip-flops

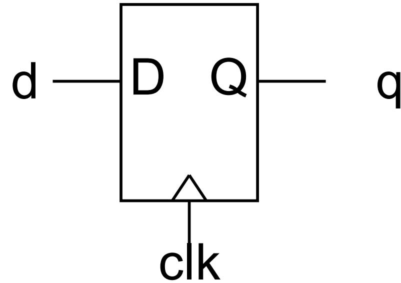
© 2018 BYU

Assignment Statements

- Use non-blocking assignment statements (<=)
- Flip-Flops will be inferred for all signals that are assigned within always_ff blocks
- Can use "if" and "case" statements
  - Do not need to cover all cases (memory element will hold previous value)

© 2018 BYU

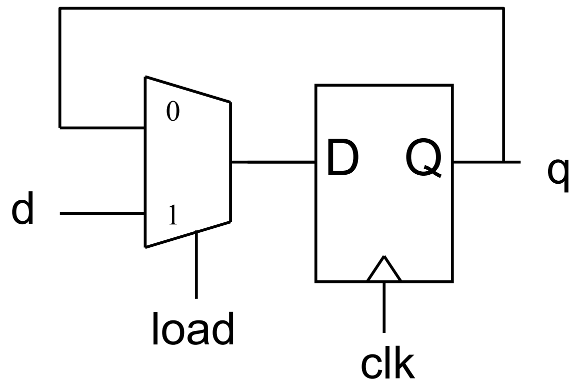
1-Bit Flip-Flop




```
module ff(
  input wire logic clk, d,
  output logic q
  );

  always_ff @(posedge clk)
    q <= d;

endmodule
```


© 2018 BYU

Loadable 1-Bit Flip-Flop



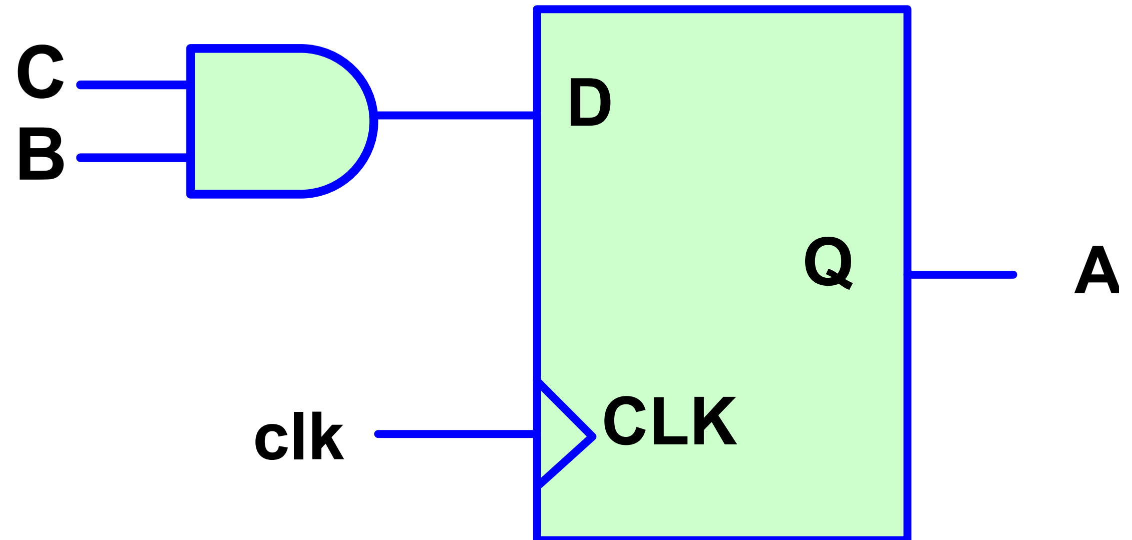

```
module ff(
  input wire logic clk, load, d,
  output logic q
  );

  always_ff @(posedge clk)
    if (load)
      q <= d;

endmodule
```

if load is '1', load d to q
else nothing happens

© 2018 BYU

Flip-Flops and Logic

```
always_ff @(posedge clk)
begin
   a <= b&c;
end
```

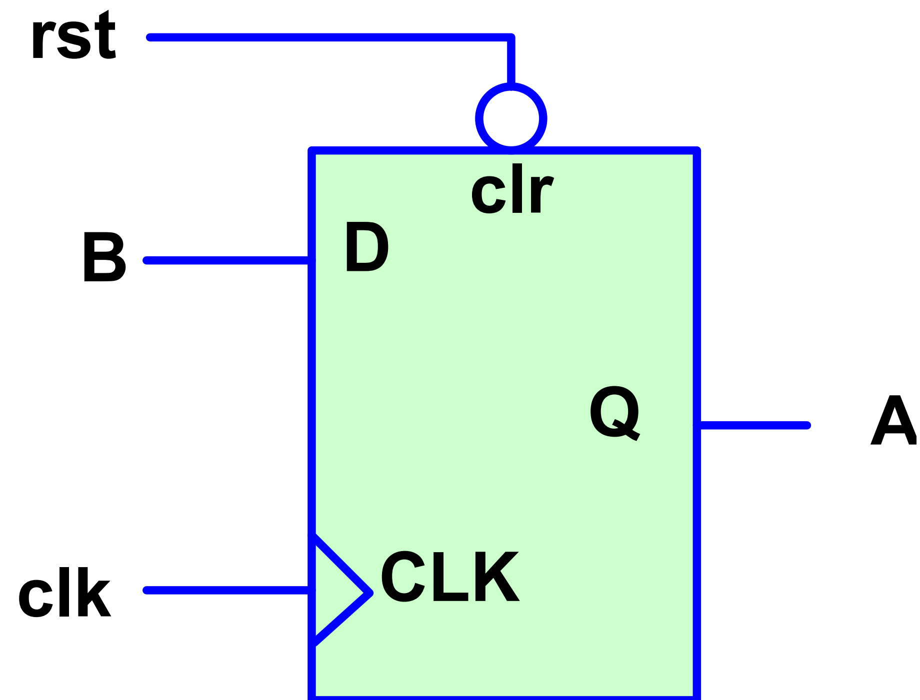


© 2018 BYU

D Flip-Flop with Asynchronous Reset

```
always_ff @(posedge clk
   or negedge rst)
begin
   if (!rst) A <= 0;
   else A <= B;
end
```

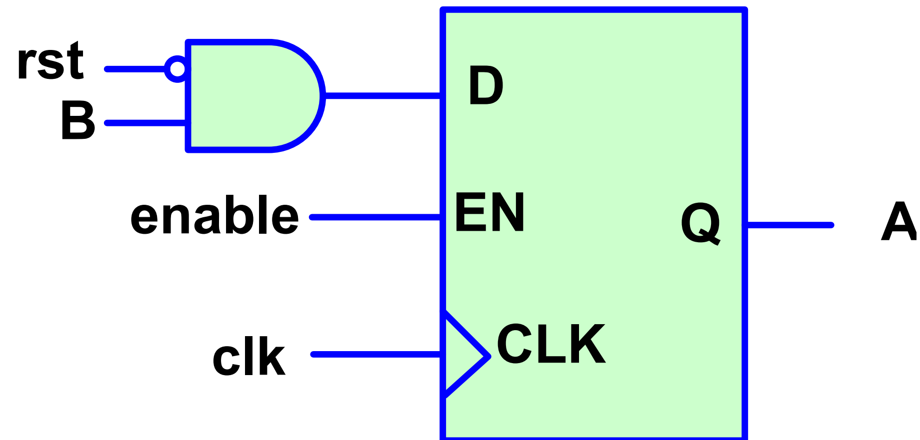
To make the reset synchronous, just remove `negedge rst` from the sensitivity list.



© 2018 BYU

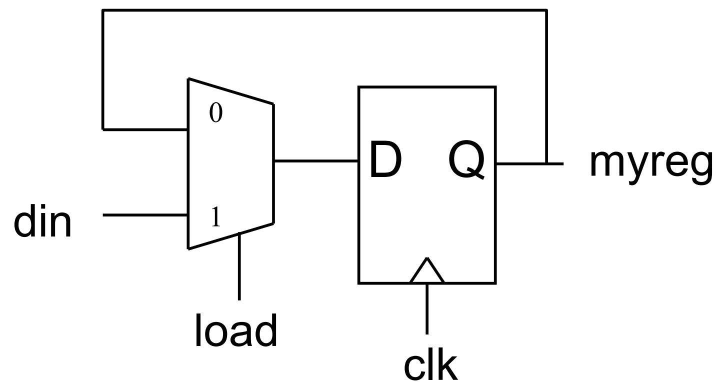
D Flip-flop with Synchronous Reset and Enable

```
always_ff @(posedge clk)
begin
  if (rst)
    A <= 0;
  else if (enable)
    A <= B;
end
```



© 2018 BYU

A Parameterized Loadable Register




```
module LoadableReg #(WID=4)(
  input wire logic clk, load,
  input logic[WID-1:0] din,
  output logic[WID-1:0] myreg
  );

  always_ff @(posedge clk)
    if (load)
      myreg <= din;

endmodule
```

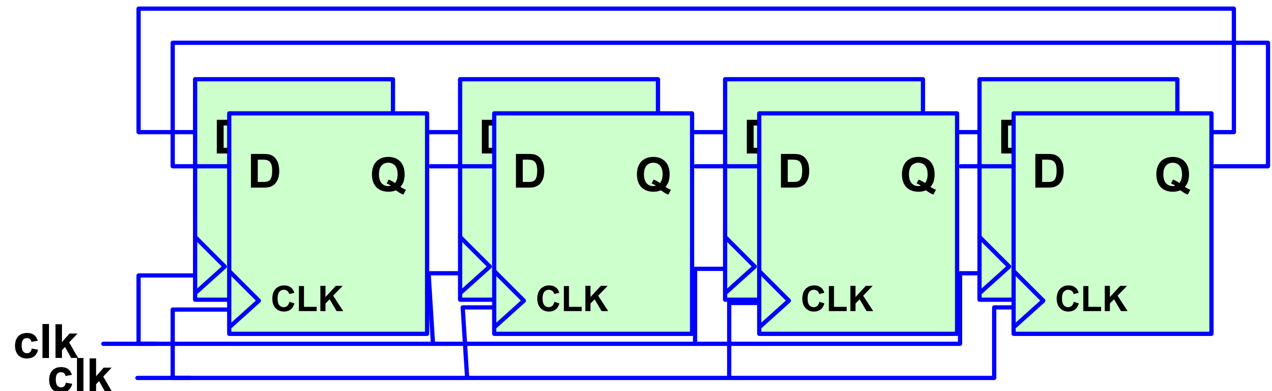

© 2018 BYU

A Shift Register

```
logic[3:0] Q;
always_ff @(posedge clk)
begin
    Q <= {Q[2:0],Q[3]}
end
```

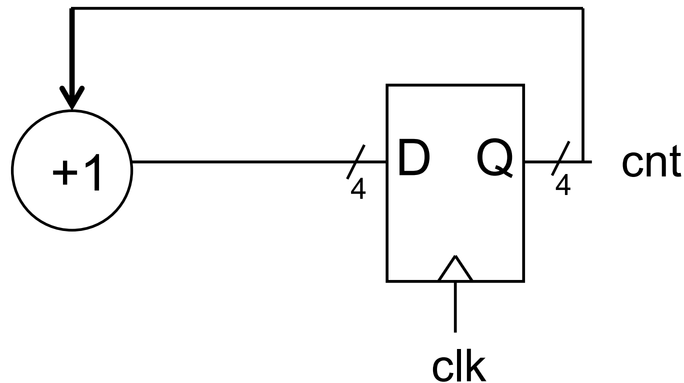
Q[3] <= Q[2];
Q[2] <= Q[1];
Q[1] <= Q[0];
Q[0] <= Q[3];

Without a reset or load, this circuit can never be initialized.



© 2018 BYU

A Parameterized Counter



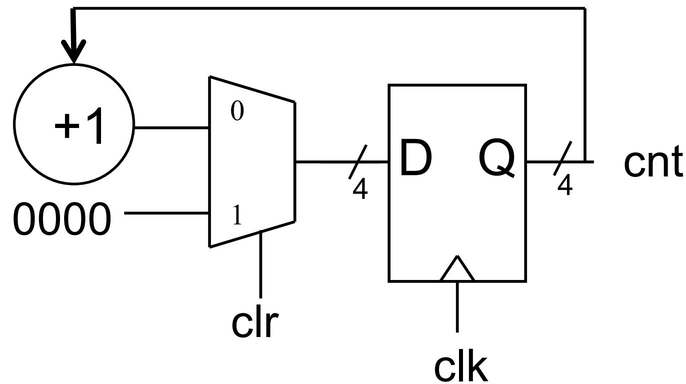

```
module upCnt #(WID=4)(
   input wire logic clk,
   output logic[WID-1:0] cnt
   );

   always_ff @(posedge clk)
      cnt <= cnt + 1;

endmodule
```

Without a clear or load, this circuit can never be initialized.

Must have a clr signal to reset it to 0.
Otherwise, cnt would be in X state on startup.

© 2018 BYU

A Counter with a Clear

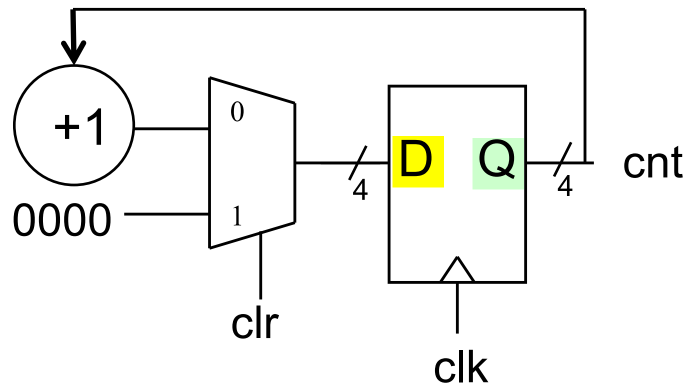



```
module upCnt #(WID=4)(
  input wire logic clk, clr,
  output logic[WID-1:0] cnt
  );

  always_ff @(posedge clk)
    if (clr)
      cnt <= 0;
    else
      cnt <= cnt + 1;
endmodule
```

Must have a clr signal to reset it to 0.
Otherwise, cnt would be in X state on startup.

© 2018 BYU

Current/Next State Values

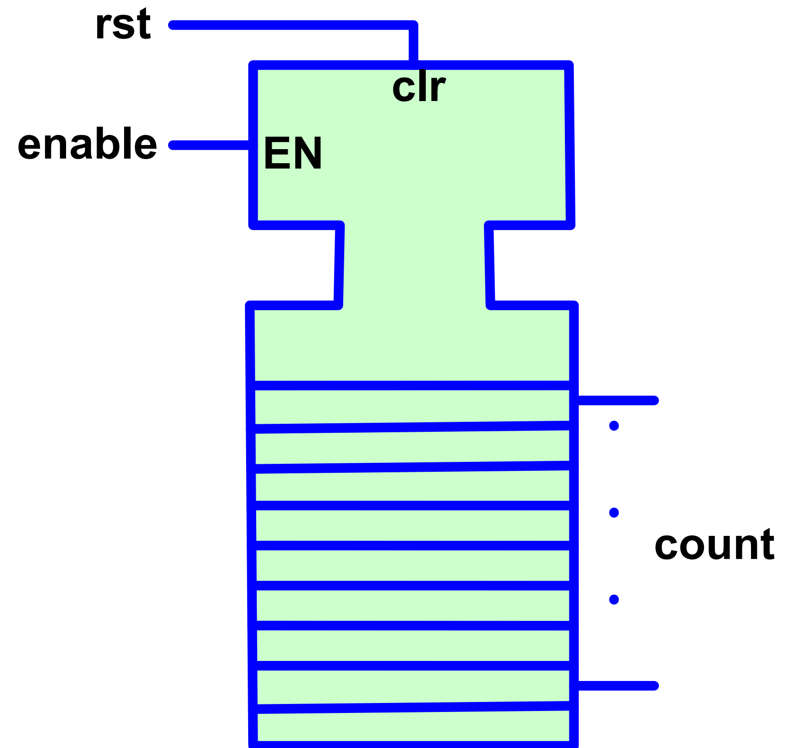



```
module upCnt #(WID=4)(
  input wire logic clk, clr,
  output logic[WID-1:0] cnt
  );

  always_ff @(posedge clk)
    if (clr)
      cnt <= 0;
    else
      cnt <= cnt + 1;
endmodule
```

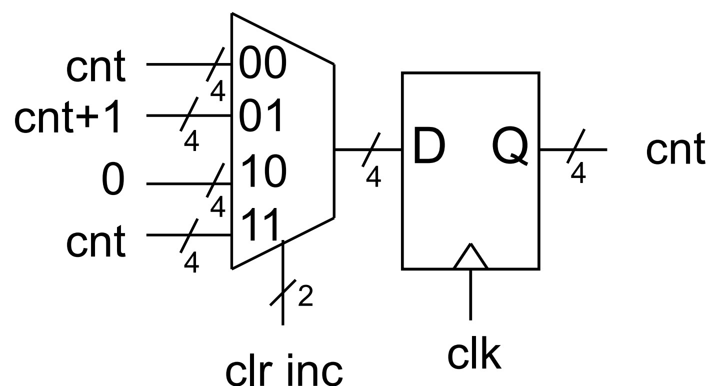

© 2018 BYU

A Counter with Synchronous Reset and Enable

```
parameter WID=8;
logic[WID-1:0] count;
logic enable, reset;

always_ff @(posedge clk)
begin
   if (reset)
      count<=0;
   else if (enable)
      count<=count+1;
end
```



Another Counter

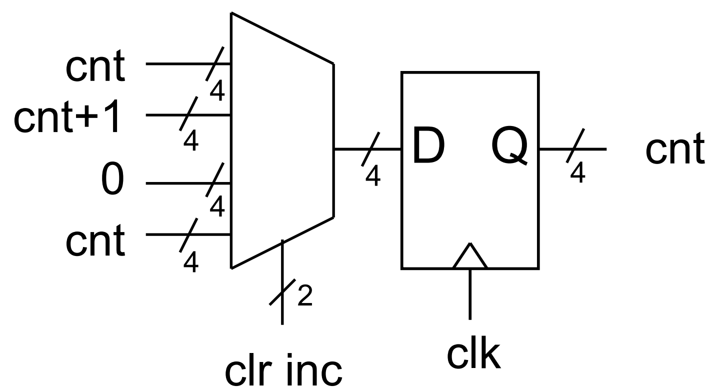



```
module upCnt #(WID=4)(
   input wire logic clk, clr, inc,
   output logic[WID-1:0] cnt
   );

   always_ff @(posedge clk)
     if ( ? )
       cnt <= 0;
     else if ( ? )
       cnt <= cnt + 1;
endmodule
```

```
A.  !clr && !inc
B.  !clr &&  inc
C.   clr && !inc
D.   clr &&  inc
```


© 2018 BYU

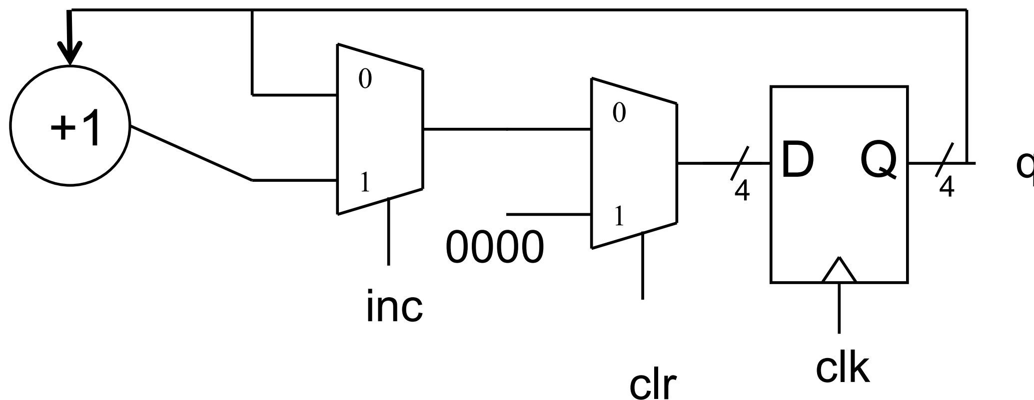
Another Counter




```
module upCnt #(WID=4)(
  input wire logic clk, clr, inc,
  output logic[WID-1:0] cnt
  );

  always_ff @(posedge clk)
    if (clr && !inc)
      cnt <= 0;
    else if (!clr && inc)
      cnt <= cnt + 1;
endmodule
```


© 2018 BYU

Yet Another Counter

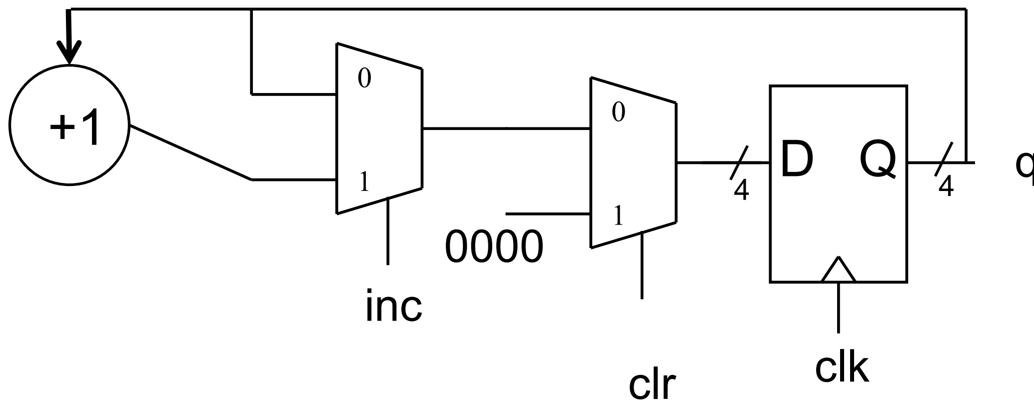



```
module upCnt #(WID=4)(
  input wire logic clk,
  input wire logic clr, inc,
  output logic[WID-1:0] q
  );

  always_ff @(posedge clk)
    if (?)
      q <= 0;
    else if (?)
      q <= q + 1;
endmodule
```

A. !clr
B. clr
C. !inc
D. inc

© 2018 BYU

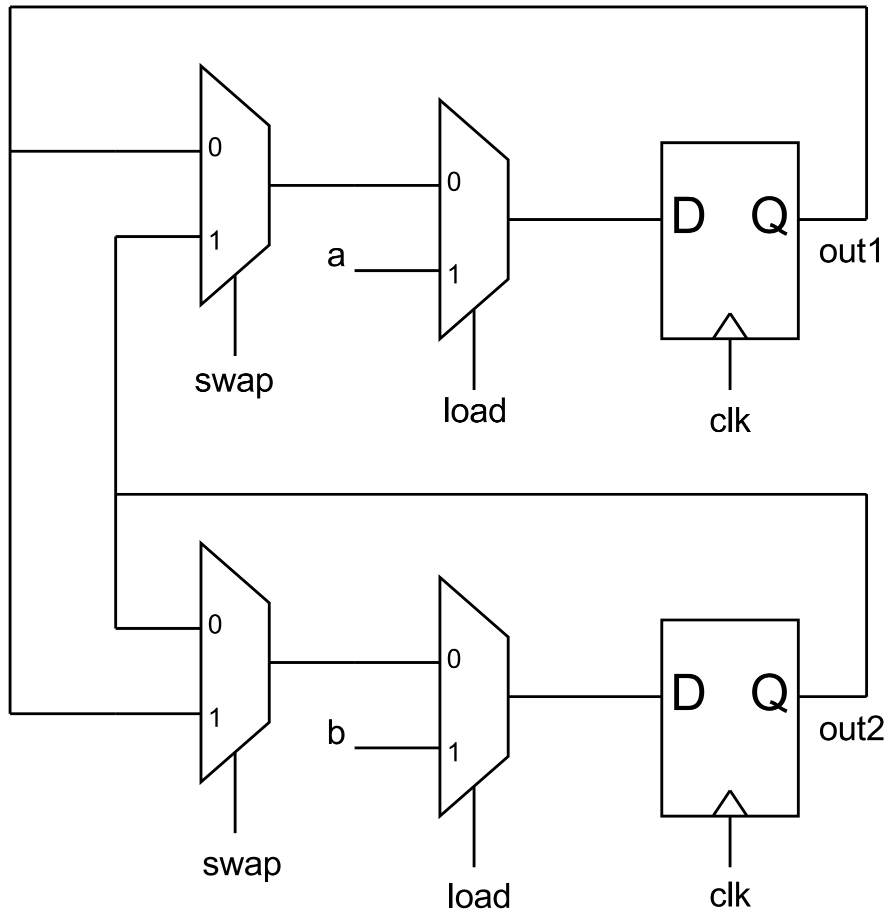
Yet Another Counter




```
module upCnt #(WID=4)(
  input wire logic clk,
  input wire logic clr, inc,
  output logic[WID-1:0] q
  );

  always_ff @(posedge clk)
    if (clr)
      q <= 0;
    else if (inc)
      q <= q + 1;
endmodule
```


© 2018 BYU

A Swapper

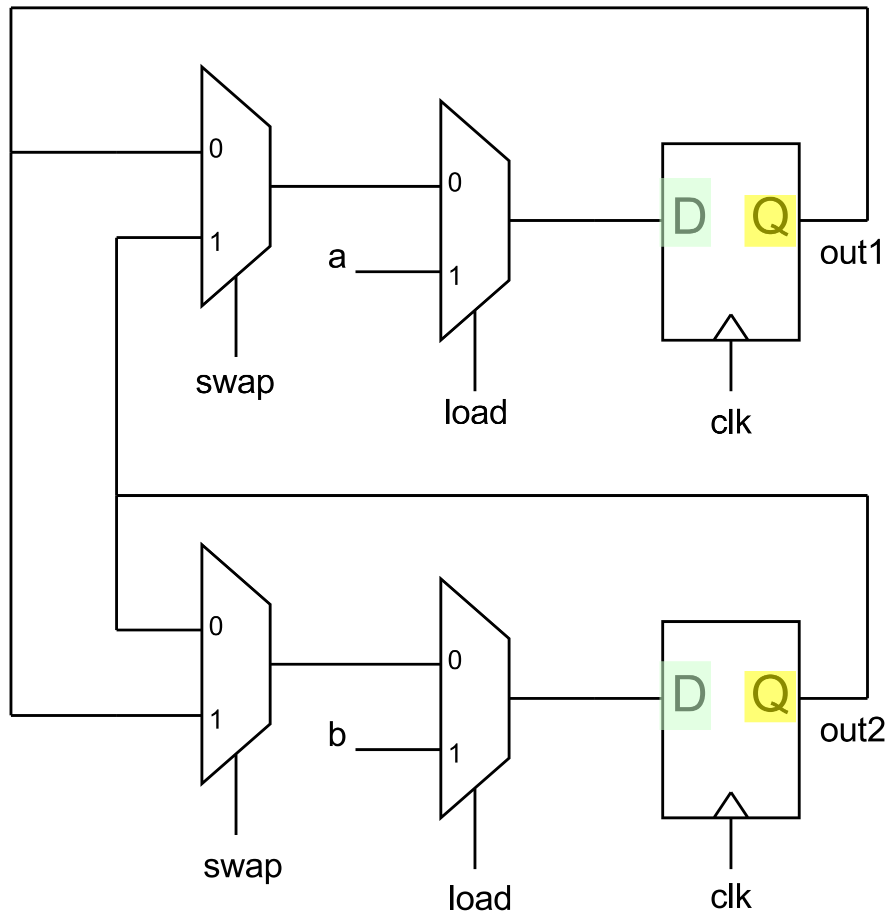



```
module swapper (
  input wire logic clk, load, swap,
  input wire logic a, b,
  output logic out1, out2
  );

  always_ff @(posedge clk)
    if (?)
    begin
      out1 <= a;
      out2 <= b;
    end
    else if (?)
    begin
      out1 <= out2;
      out2 <= out1;
    end
endmodule
```

A. !load
B. load
C. !swap
D. swap

© 2018 BYU

A Swapper



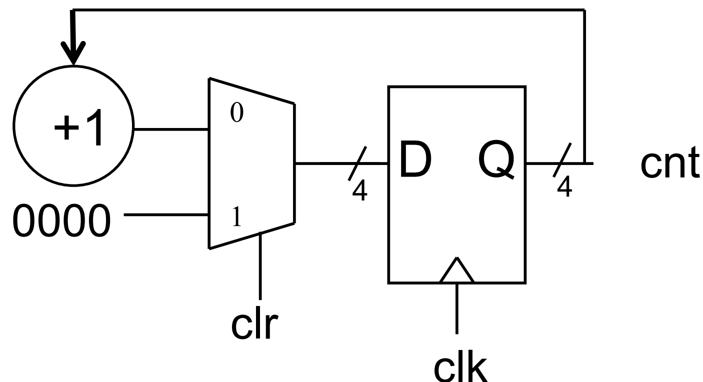

```
module swapper (
  input wire logic clk, load, swap,
  input wire logic a, b,
  output logic out1, out2
  );

  always_ff @(posedge clk)
    if (load)
    begin
      out1 <= a;
      out2 <= b;
    end
    else if (swap)
    begin
      out1 <= out2;
      out2 <= out1;
    end
endmodule
```


© 2018 BYU

Multiple Assignments to a Signal

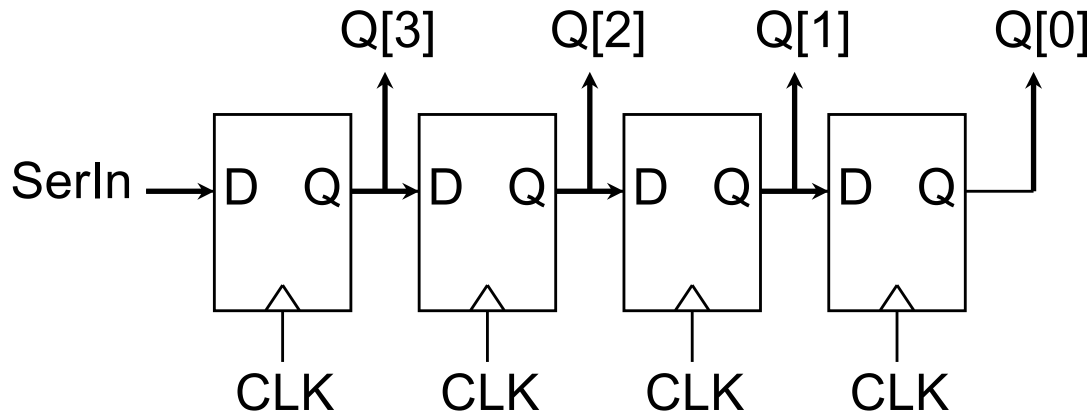
- When you assign twice…
  - Last one takes effect on the signal value

```
module multiAssign (
  input wire logic clk, clr,
  output logic[3:0] cnt
  );

  always_ff @(posedge clk)
  begin
    cnt <= 4'b0000;
    if (!clr)
      cnt <= cnt+1;
  end
endmodule
```



© 2018 BYU

A Shift Register

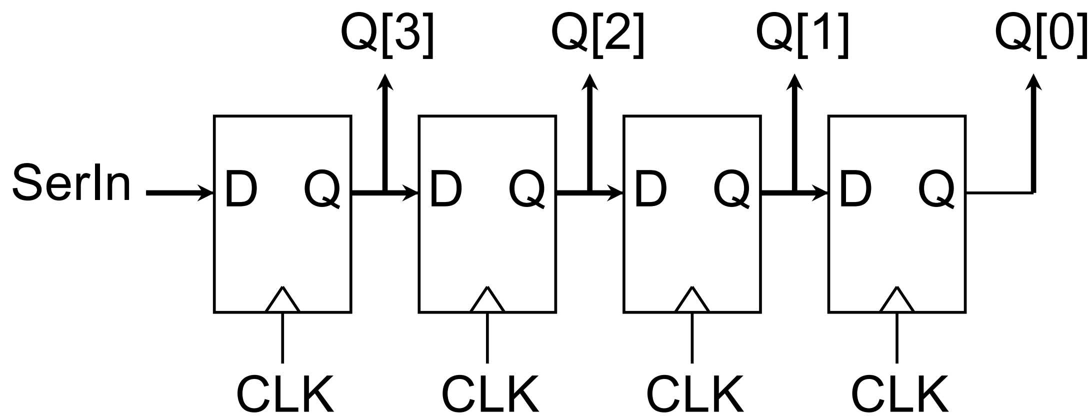



```
module delay4 (
  input wire logic clk, SerIn,
  output logic[3:0] Q
  );

  always_ff @(posedge clk)
    Q <= ?;
endmodule
```

A. {SerIn, Q[2:0]}
B. {Q[3:1], SerIn}
C. {SerIn, Q[3:1]}
D. {Q[2:0], SerIn}
E. {Q[2:1], SerIn}

© 2018 BYU

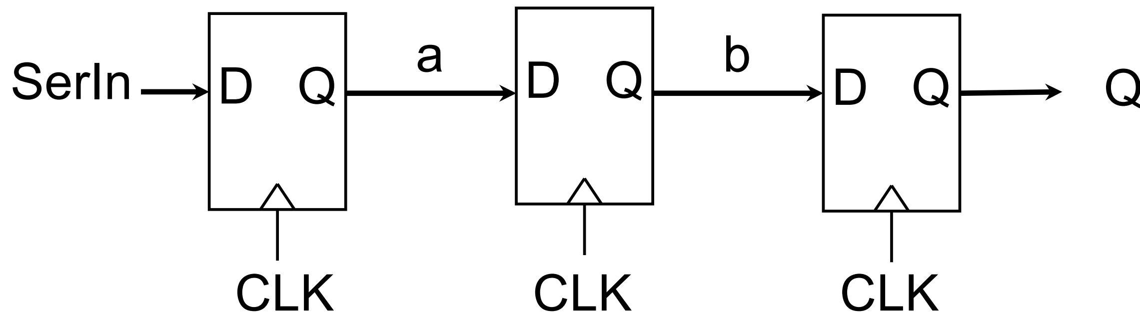
A Shift Register




```
 module delay4 (
  input wire logic clk, SerIn,
  output logic[3:0] Q
  );

  always_ff @(posedge clk)
    Q <= {SerIn, Q[3:1]};
endmodule
```


© 2018 BYU

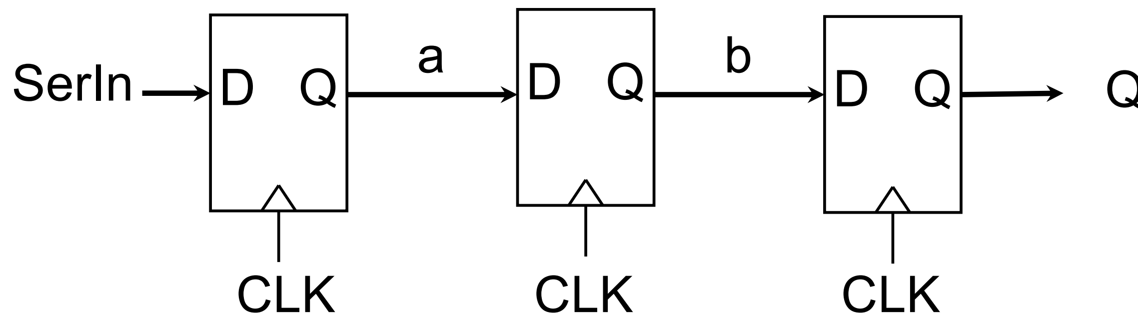
Another Shift Register




```
module delay3 (
 input wire logic clk,
 input wire logic SerIn,
 output logic Q
 );
 logic a, b;

 always_ff @(posedge clk)
 begin
   a <= SerIn;
   b <= a;
   Q <= b;
 end
endmodule
```


© 2018 BYU

Yet Another Shift Register

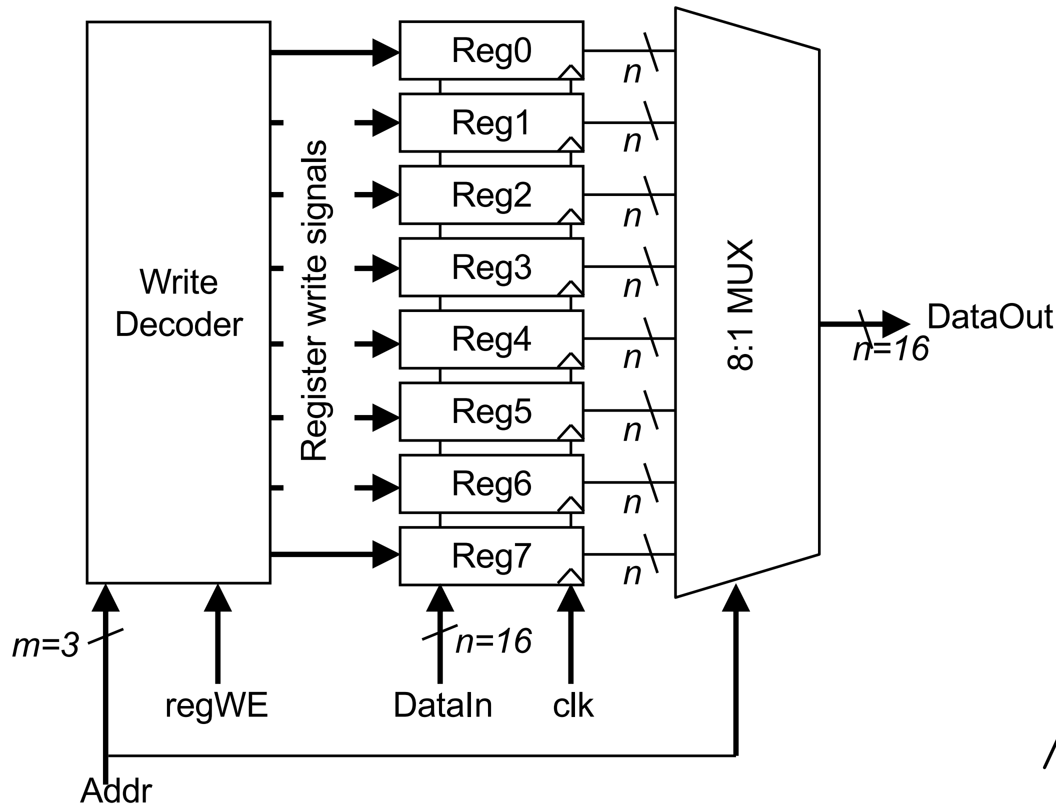



```
module delay3 (
  input wire logic clk,
  input wire logic SerIn,
  output logic Q
  );
  logic a, b;

  always_ff @(posedge clk)
  begin
    Q <= b;
    b <= a;
    a <= SerIn;
  end
endmodule
```

Order of assignments DOES NOT MATTER…

© 2018 BYU

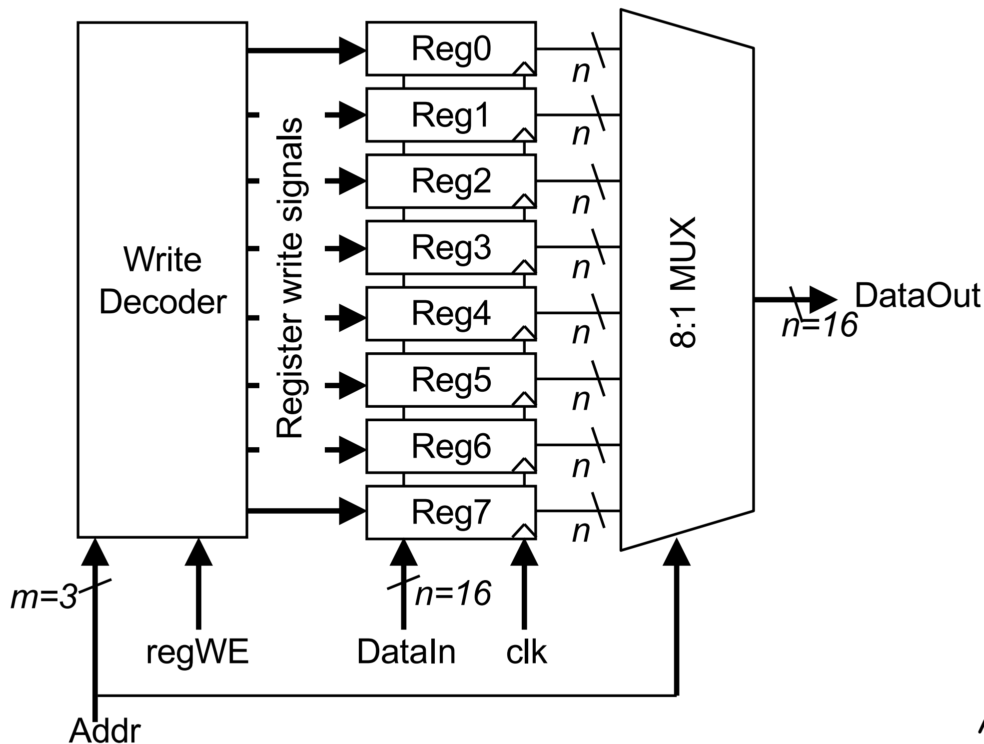
A Register File




```
module regFile (
  input wire logic clk, regWE,
  input wire logic [2:0] Addr,
  input wire logic [15:0] DataIn,
  output logic[15:0] DataOut
  );

  logic [15:0] registers [8];

  // The synchronous write logic
  always_ff @(posedge clk)
  if (regWE)
     registers[Addr] <= ?;

// The asynchronous read logic
    assign DataOut = ?;

endmodule
```


© 2018 BYU

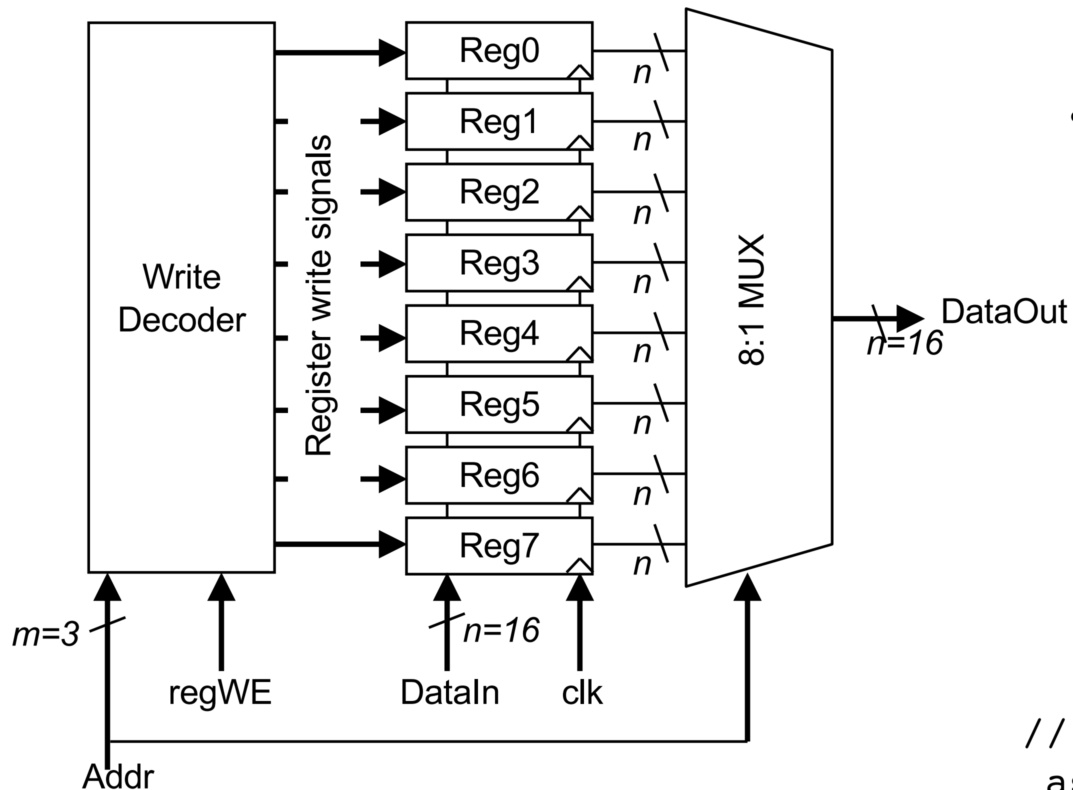
A Register File




```
module regFile (
   input wire logic clk, regWE,
   input wire logic [2:0] Addr,
   input wire logic [15:0] DataIn,
   output logic[15:0] DataOut
   );

   logic [15:0] registers [8];

   // The synchronous write logic
   always_ff @(posedge clk)
   if (regWE)
      registers[Addr] <= DataIn;

// The asynchronous read logic
     assign DataOut = registers[Addr];

endmodule
```


© 2018 BYU

A Register File




```
module regFile (
  input wire logic clk, regWE,
  input wire logic [2:0] Addr,
  input wire logic [15:0] DataIn,
  output logic[15:0] DataOut
  );
  logic[15:0] Reg0,Reg1,Reg2,Reg3,
              Reg4,Reg5,Reg6,Reg7;

  // The synchronous write logic
  always_ff @(posedge clk)
    if (regWE)
     case(Addr)
       3'b000: Reg0 <= DataIn;
       3'b001: Reg1 <= DataIn;
       3'b010: Reg2 <= DataIn;
       3'b011: Reg3 <= DataIn;
       3'b100: Reg4 <= DataIn;
       3'b101: Reg5 <= DataIn;
       3'b110: Reg6 <= DataIn;
       default: Reg7 <= DataIn;
     endcase

 // The asynchronous read logic
  assign DataOut = (Addr==3'000)? Reg0:
                   (Addr==3'001)? Reg1:
                    …….
                   reg7;
endmodule
```


© 2018 BYU