Verilog Tutorial

ELE/COS 475 – Spring 2012

Agenda

- ELE475 Verilog Infrastructure
  - iverilog, gtkwave, PARC processor
- Verilog tutorial
- Lab 0 – Practice Lab
  - iterative multiplier and divider
- Any other class-related questions

ELE 475 Verilog Infrastructure

- Icarus Verilog (iverilog) – open-source Verilog simulation and synthesis tool
  - iverilog converts Verilog files to “vvp assembly”
  - vvp executes the compiled “vvp assembly”
    - Writes VCD-format log file as output
- gtkwave is an open-source waveform viewer that displays VCD (and other) files graphically

PARC Processor

- Verilog teaching processor developed for a class at Cornell
- Models a simplified MIPS32-like instruction set
- Our goal in the labs will be to build new parts of the microarchitecture to improve the performance of the processor

Verilog Tutorial

Why Verilog (or VHDL?)

- **Hardware** description language
  - text description – no graphical schematics
- Model digital circuits
- Easier to prototype new hardware in HDL first
- Easy to create testbenches
- Verilog has C-like syntax
  - and C-style comments!

What does Verilog Model?

- Anything gate-level and above
- Only types in Verilog are "bit" (0, 1, X, Z) and bit-vector
- Constant syntax:
  - 1'b0 = 1 bit long, binary 0
  - 8'b01011010 = 8'h5A = 8'd90

Symbol	Meaning
0	Logic 0
1	Logic 1
X	Conflict
Z	Floating

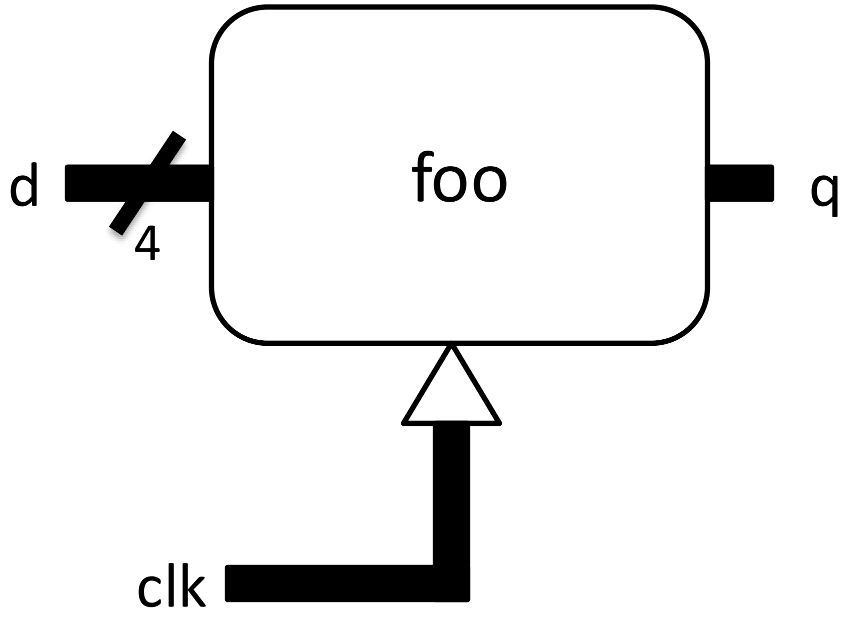
Basic Syntax

- Systems are described as a set of modules
- Logic is specified either as **structural** (combinational) or **behavioral** (sequential)

- Two types of signal:
  - "**Wire**" is a simple connection between two components
    - Should have just one driver
    - no persistence
  - "**Reg**" stores its value until updated again

Modules

- Logic is contained within **modules**
- External interface: declared set of inputs, outputs, and “inout”s
- Internal contents: wires, registers, logic, and/or submodules



Modules

- Logic is contained within **modules**
- External interface: declared set of inputs, outputs, and “inout”s
- Internal contents: wires, registers, logic, and/or submodules

```
module foo(clk, d, q);
  input clk;
  input [3:0] d;
  output q;

  // internal “foo”
  // logic

endmodule
```

Structural Verilog

- For combinational logic
- Continuous assignment using “assign” and “=“
- In Verilog, “=“ is blocking assignment – it takes place immediately
  - Can cause other events to trigger on the same cycle

```
module mux2(a,b,s,q);
  input a, b, s;
  output q;
  wire q;

  assign q = (a & s) |
    (b & ~s);
endmodule
```

Behavioral Verilog

- For sequential Logic
- Delayed assignment using "<=“
- In Verilog, "<=“ is non-blocking assignment – no changes take effect until next cycle

```
always @(condition)
begin
  // desired behavior
end
```

```
module d_ff(clk,d,q);
  input clk, d;
  output q;
  reg q;
  always @(posedge clk)
  begin
    q <= d;
  end
endmodule
```

Example – Swap?

```
// Blocking
always @(posedge clk)
begin
  a = b;
  b = a;
end
```

```
// Non-blocking
always @(posedge clk)
begin
  a <= b;
  b <= a;
end
```

What is the difference between the two?

What is the result of each?

Example – Swap?

```
// Blocking
always @(posedge clk)
begin
  a = b;
  b = a;
end
```

```
// Non-blocking
always @(posedge clk)
begin
  a <= b;
  b <= a;
end
```

What is the difference between the two?

What is the result of each?

Both get the original value of b (immediately)!

The values are swapped (on the next cycle)

Non-Synthesizable Subset

- A large part of Verilog is not synthesizable directly into hardware
  - For example, the divider in lab 0
- This subset is still useful for:
  - prototyping of a module that will later be elaborated
  - Making testbenches which will not be synthesized
- The code describing the hardware itself should be synthesizable Verilog!

Non-Synthesizable Subset - Examples

```
// Division
always @(posedge clk)
begin
  q <= d / 7;
end
```

Not synthesizable, but often useful for quick prototyping

```
// Delays
always
begin
  #1 clk = ~clk;
end
```

Not synthesizable, but often used when creating testbenches

Other Useful Syntax

- If/then/else/end
- Case(value)
- Conditional
  - (condition) ? (result-if-true) : (result-if-false)
- initial blocks

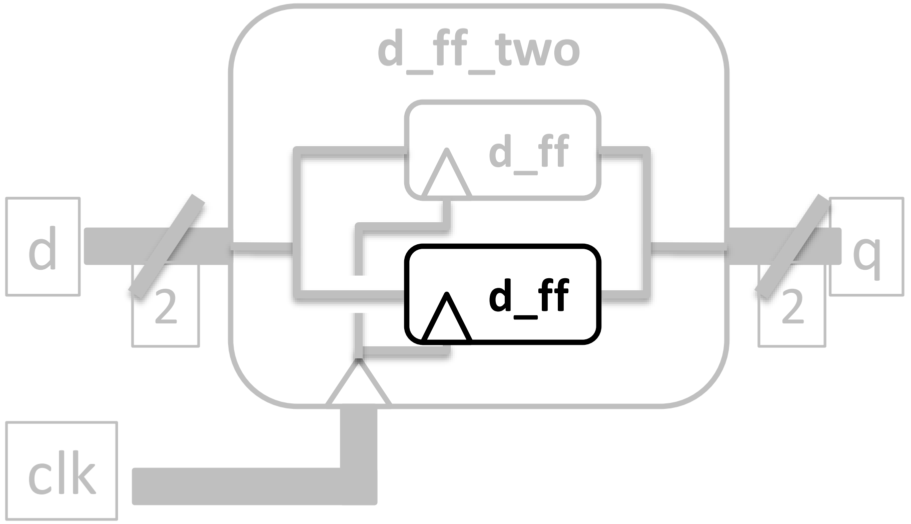
We will see examples of these shortly…

```
initial //not synthesizable!
begin
  q = 2’b00;
end
```

Module Instantiation

```
module d_ff(clk,d,q);
input clk;
input d;
output q;
// module contents
endmodule
```

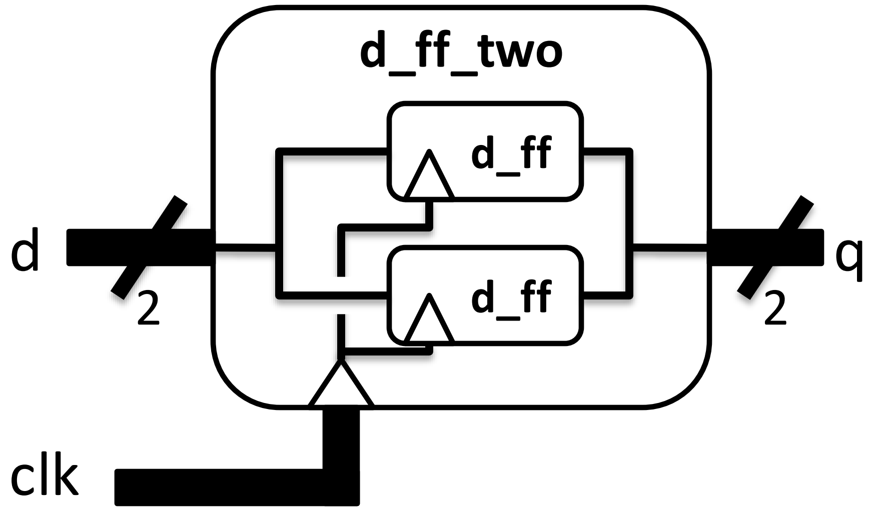
```
module d_ff_two(clk,d,q);
input clk;
input [1:0] d;
output [1:0] q;
// two d_ff instances
endmodule
```



Module Instantiation

```
module d_ff(clk,d,q);
input clk;
input d;
output q;
// module contents
endmodule
```

```
module d_ff_two(clk,d,q);
input clk;
input [1:0] d;
output [1:0] q;
// two d_ff instances
endmodule
```



Module Instantiation

```
module d_ff(clk,d,q);
input clk;
input d;
output q;
reg q;
always @(posedge clk)
  begin
    q <= d;
  end
endmodule
```

```
module d_ff_two(clk,d,q);
input clk;
input [1:0] d;
output [1:0] q;
d_ff d_ff_instance1(
 .clk (clk),
 .d   (d[0]),
 .q   (q[0])
);
d_ff d_ff_instance2(
   clk, d[1], q[1]);
endmodule
```

Finite State Machine

- Split the design into parts:
  - Next-state logic (combinational): given state and inputs, calculate next output state
  - Output values: given current state and inputs, calculate output values
  - State advancement (sequential): on positive edge of clock, set “state” to calculated “next-state”

Finite State Machine – Output Logic

```
module statemachine(clk, in,
   reset, out);
 input clk, in, reset;
 output [3:0] out;
 reg [3:0] out;
 reg [1:0] next_state;
 reg [1:0] state;
 parameter zero=0, one=1,
   two=2, three=3;
 always @(*)
   begin

   end
```

```
case (state)
        zero:
            out = 4'b0000;
        one:
            out = 4'b0001;
        two:
            out = 4'b0010;
        three:
            out = 4'b0011;
        default:
            out = 4'b0000;
    endcase
```

Finite State Machine – Next State

```
always @(*)
    begin
    case (state)


      default:
          next_state = zero;
    endcase
  end
endmodule
```

```
zero:
    next_state = one;
one:
    if (in)
        next_state = zero;
    else
        next_state = two;
two:
    next_state = three;
three:
    next_state = zero;
```

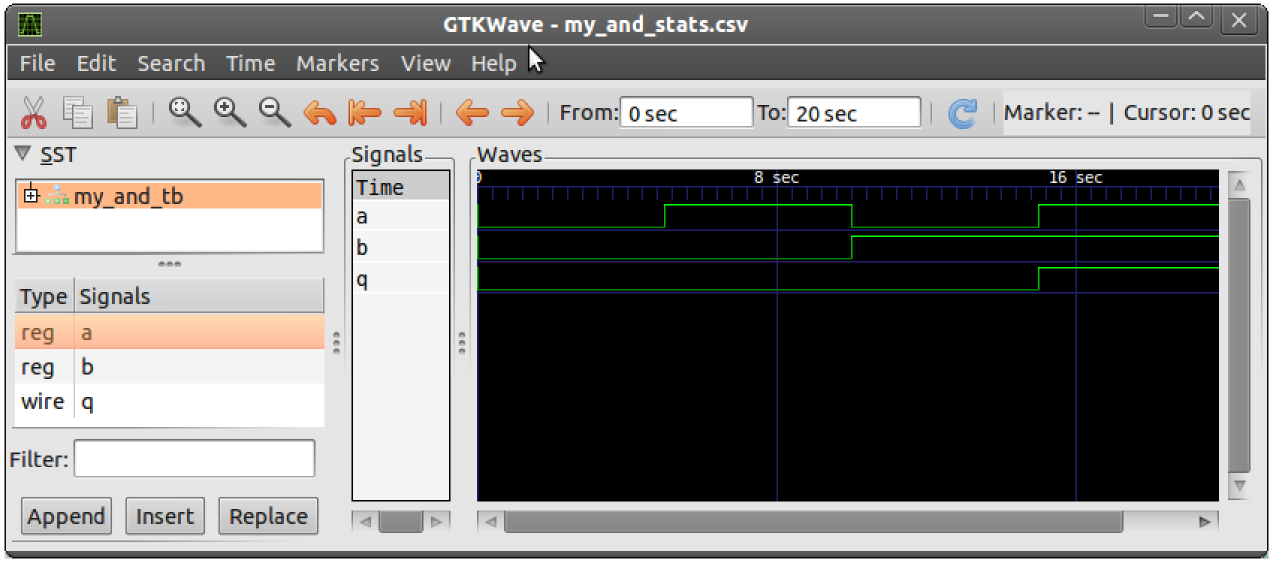
Finite State Machine – State Update

```
// asynchronous reset!
always @(posedge clk or
    posedge reset)
    begin
        if (reset)
            state = zero;
        else
            state = next_state;
    end
endmodule
```

Test Benches

- Each module should have a test bench!
  - a new module containing only the **unit-under-test** and a comprehensive set of varying inputs
  - Outputs should demonstrate that the module works as expected

module my_and
(a,b,q);



Sample Testbench

```
module foo(clk, d, q);
…
endmodule

module foo_tb();
  reg clk, d;
  wire q;
  $dumpfile(“waveform.vcd”);
  $dumpvars(0, foo_tb);

  foo uut(clk, d, q);
```

```
initial begin
  clk = 0;
  d = 0;
  #20 $finish;
end
always begin
  #5 clk = ~clk;
end
always begin
  #10 d = ~d;
end
```

Verilog Debugging

- $display – like printf in C
  - $display($time, " Count changed to %d", count);
- $monitor – watch for any changes to the value of a net
  - $monitor($time, “Count is now %d”, count);
- $dumpfile and $dumpvars – dump all signal values to a file for later viewing
  - $dumpfile(“output.log”)
  - $dumpvars(0, top_level_module)

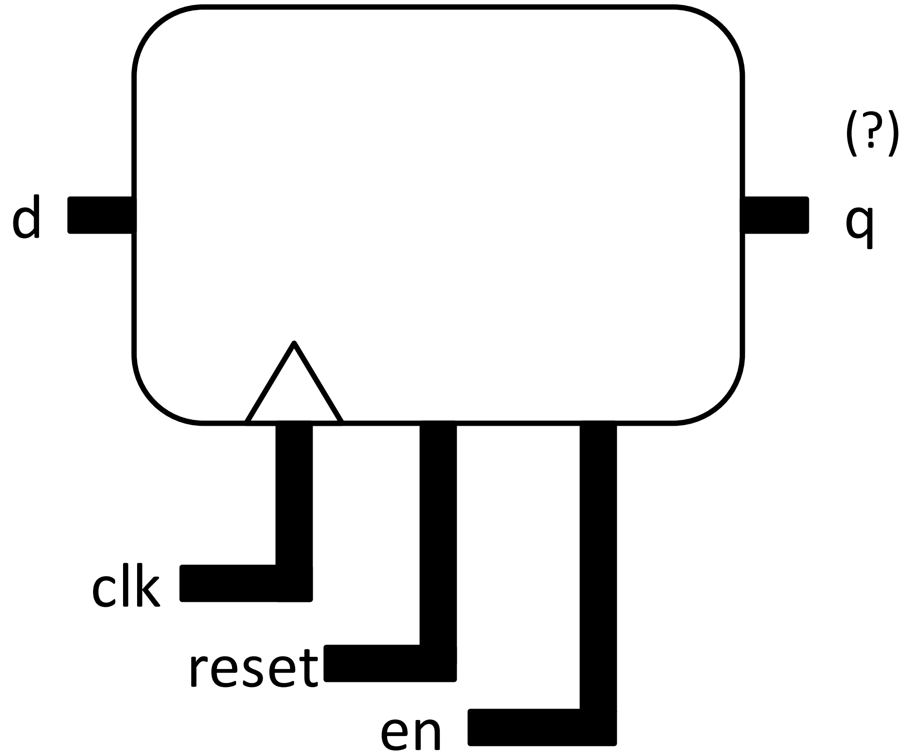
Design Advice

- Leaf vs. Non-leaf modules
  - “Leaf” modules contain logic, registers, etc.
  - Non-leaf modules contain only submodules and connecting wires
- This keeps the design hierarchy clean
- Don’t mix structural and behavioral Verilog code in the same module
- Write testbenches for every module
- Use comments the same way you would in any other language

Common Mistakes
(and how to avoid them!)

What is the Output?

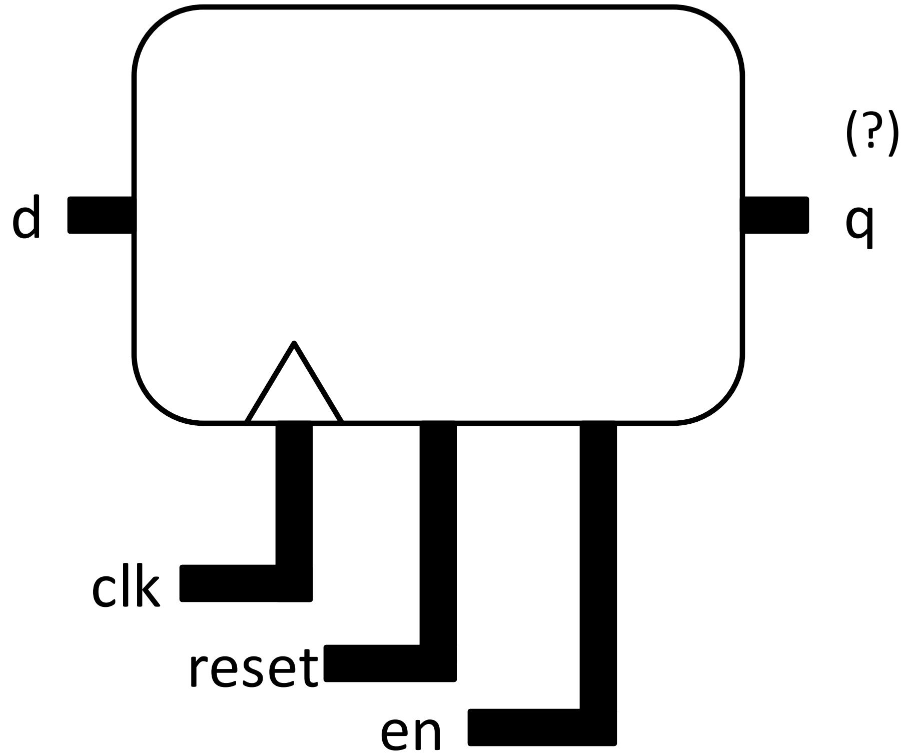
```
always @(posedge clk)
begin
 if(reset == 1)
  q <= 0;
 if(en == 1)
  q <= d;
end
```



What is the Output?

```
always @(posedge clk)
begin
  if(reset == 1)
    q <= 0;
  if(en == 1)
    q <= d;
end
```

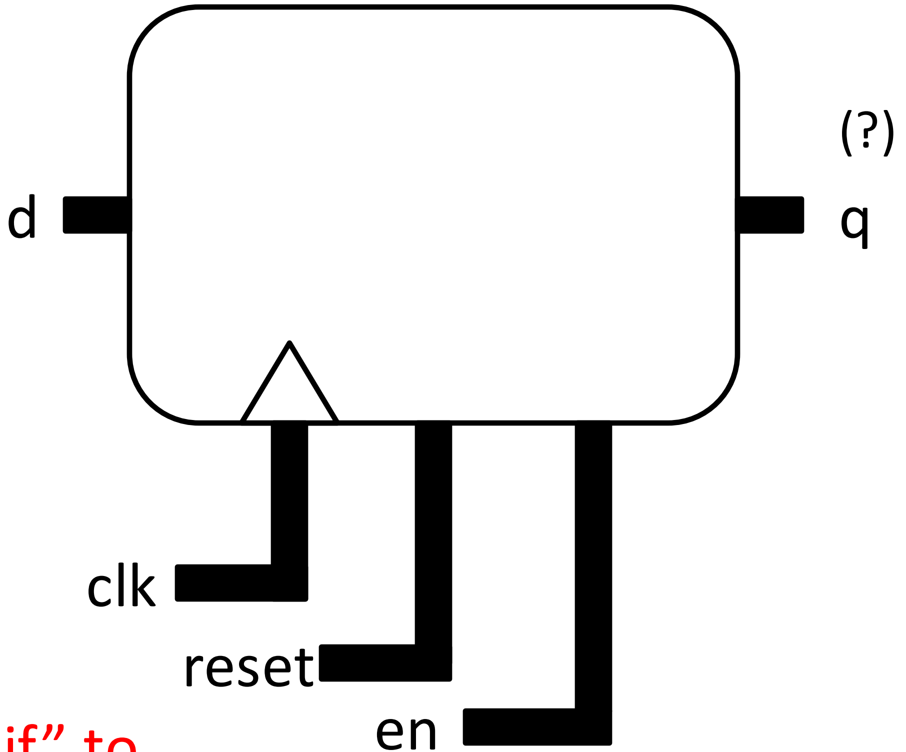
When reset == 1 and en == 1?
conflict at q!



What is the Output?

```
always @(posedge clk)
begin
  if(reset == 1)
    q <= 0;
  else if(en == 1)
    q <= d;
end
```

When reset == 1 and en == 1?
conflict at q!

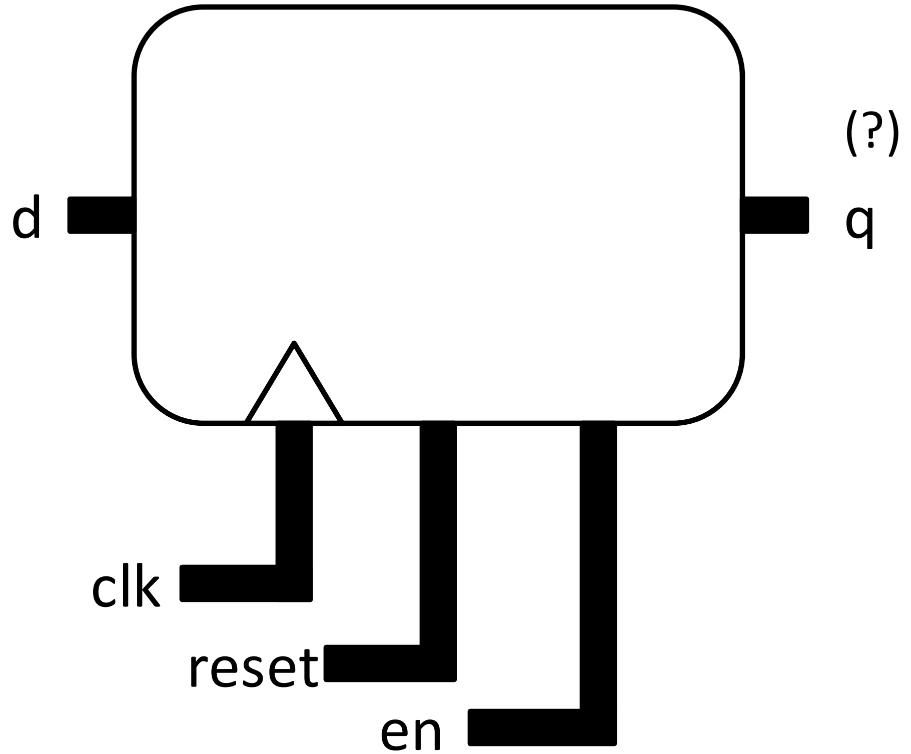


Solution: use “else if” to eliminate the conflict

What is the Output?

```
always @(posedge clk)
begin
 if(reset == 1)
   q <= 0;
 else if(en == 1)
   q <= d;
end
```

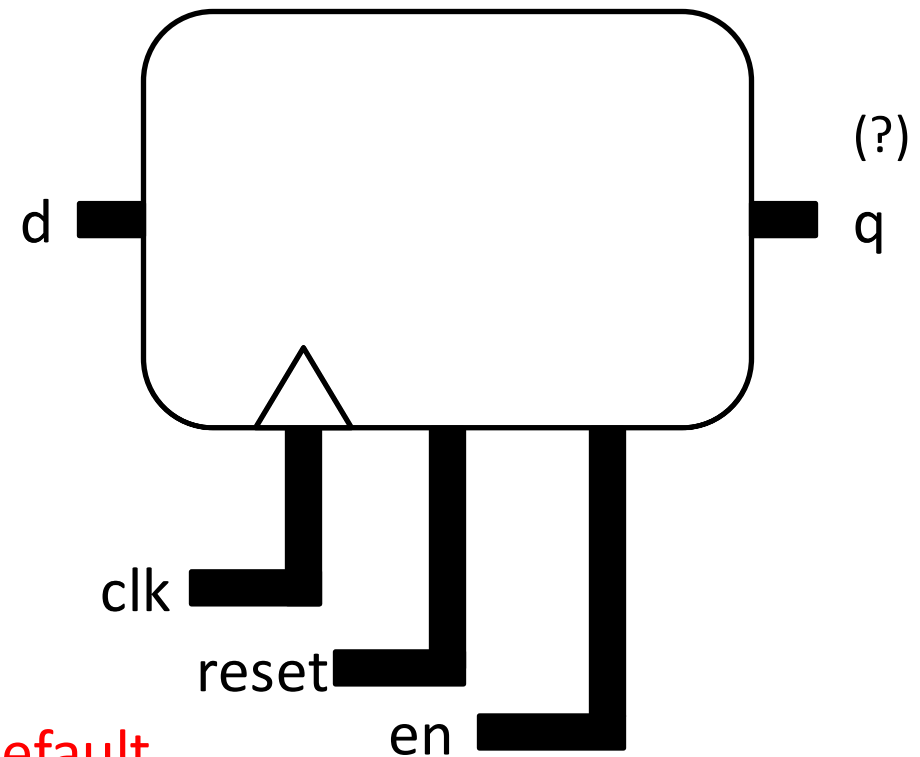
When reset == 0 and en == 0?
no assignment to q!



What is the Output?

```
always @(posedge clk)
begin
  if(reset == 1)
    q <= 0;
  else if(en == 1)
    q <= d;
  else
    q <= 0;
end
```

When reset == 0 and en == 0?
no assignment to q!



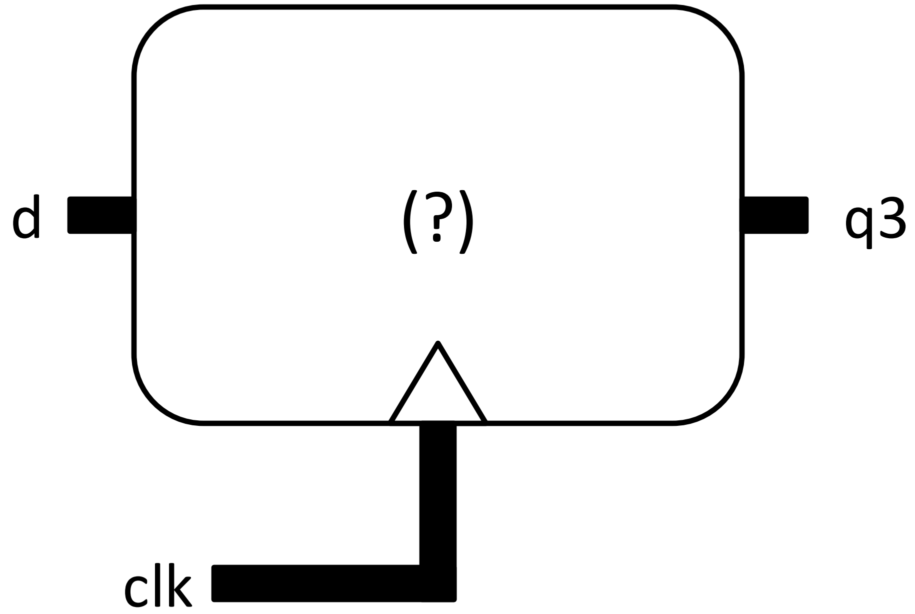
Solution: set the default condition explicitly

What circuit does this model?

```
module MyModule (q3, d, clk);
  output q3;
  input d;
  input clk;

  reg q3, q2, q1;

  always @(posedge  clk) begin
    q1 = d;
    q2 = q1;
    q3 = q2;
  end
endmodule
```

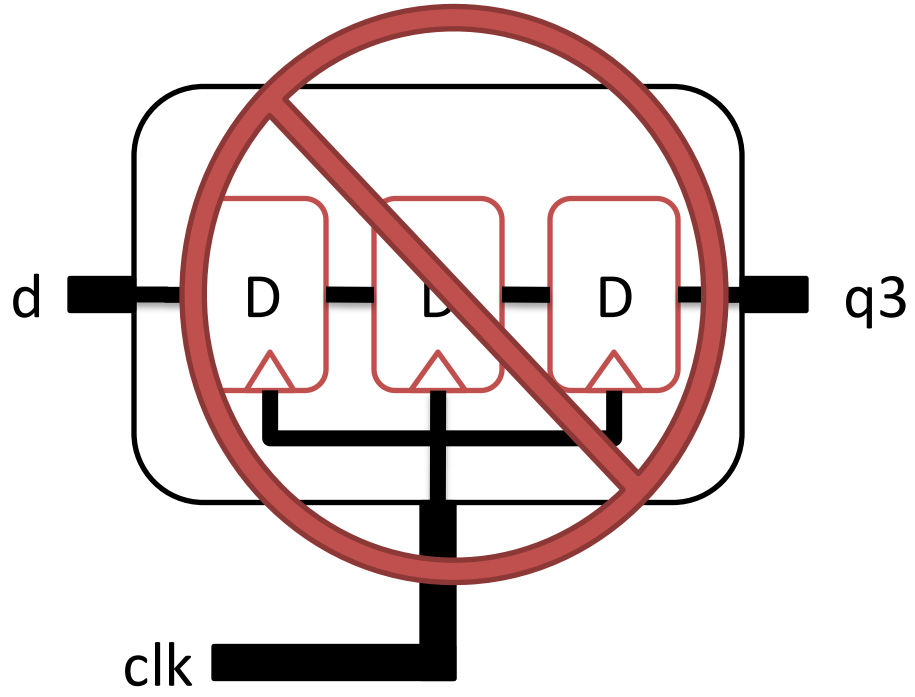


What circuit does this model?

```
module MyModule (q3, d, clk);
  output q3;
  input d;
  input clk;

  reg q3, q2, q1;

  always @(posedge  clk) begin
    q1 = d;
    q2 = q1;
    q3 = q2;
  end
endmodule
```



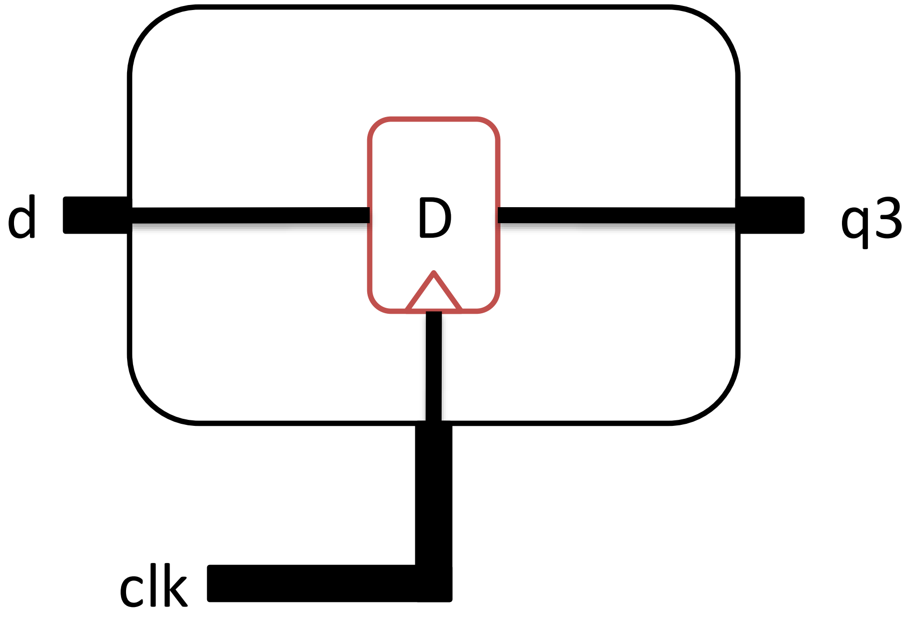
What circuit does this model?

```
module MyModule (q3, d, clk);
  output q3;
  input d;
  input clk;

  reg q3, q2, q1;

  always @(posedge clk) begin
    q1 = d;
    q2 = q1;
    q3 = q2;
  end
endmodule
```

Moral: Only use non-blocking assignment for sequential logic!

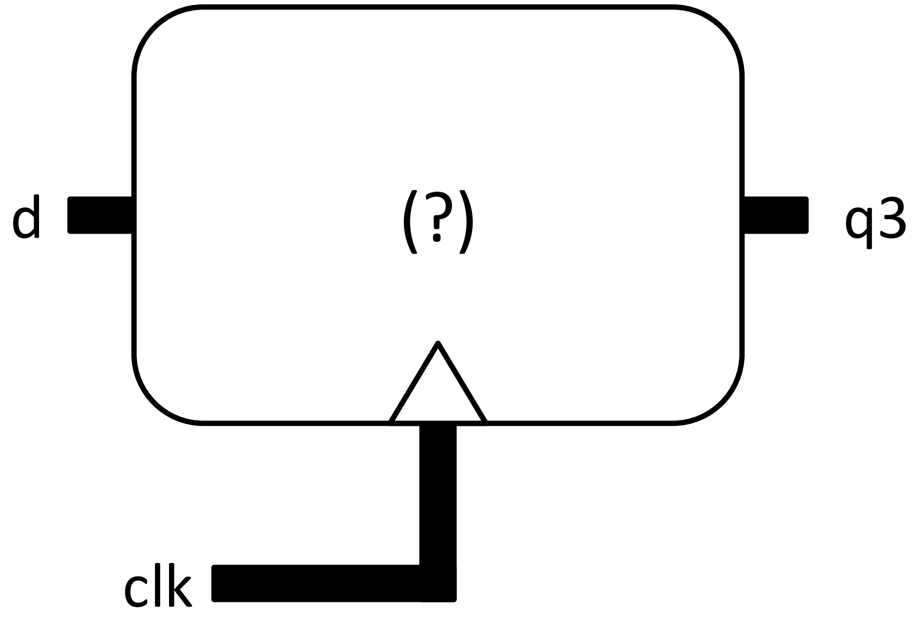


What circuit does this model?

```
module MyModule2 (q3, d, clk);
  output q3;
  input d;
  input clk;

  reg q3, q2, q1;

  always @(posedge clk) q1 = d;
  always @(posedge clk) q2 = q1;
  always @(posedge clk) q3 = q2;
endmodule
```



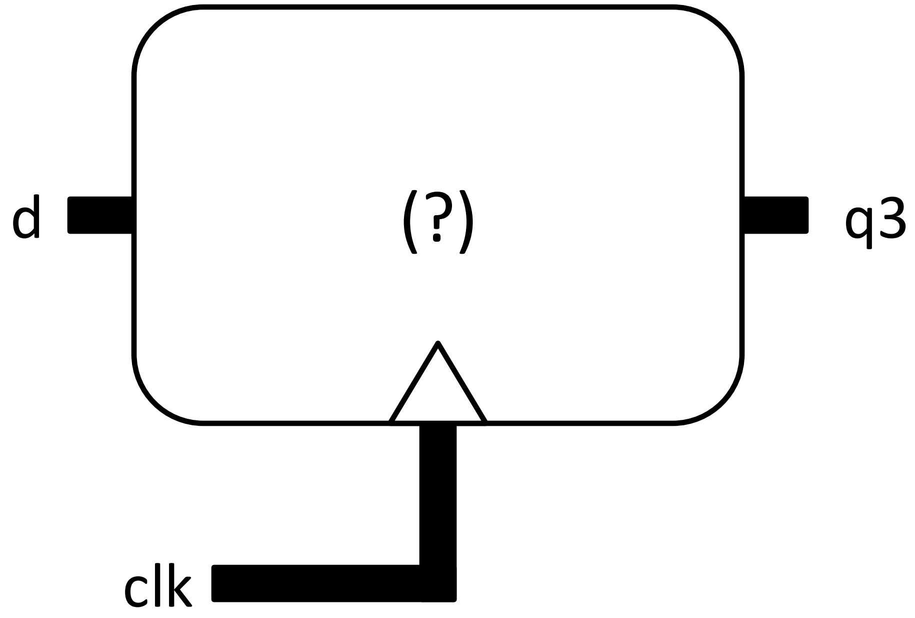
What circuit does this model?

Race condition: order is not specified!

```
module MyModule2 (q3, d, clk);
  output q3;
  input d;
  input clk;

  reg q3, q2, q1;

  always @(posedge clk) q1 = d;
  always @(posedge clk) q2 = q1;
  always @(posedge clk) q3 = q2;
endmodule
```



What circuit does this model?

Moral: understand where race conditions occur and how to avoid them

```
module MyModule2 (q3, d, clk);
  output q3;
  input d;
  input clk;

  reg q3, q2, q1;

  always @(posedge clk) q1 <= d;
  always @(posedge clk) q2 <= q1;
  always @(posedge clk) q3 <= q2;
endmodule
```