ECE 2700 Lab 4
Sequential Circuits
& Finite State Machines

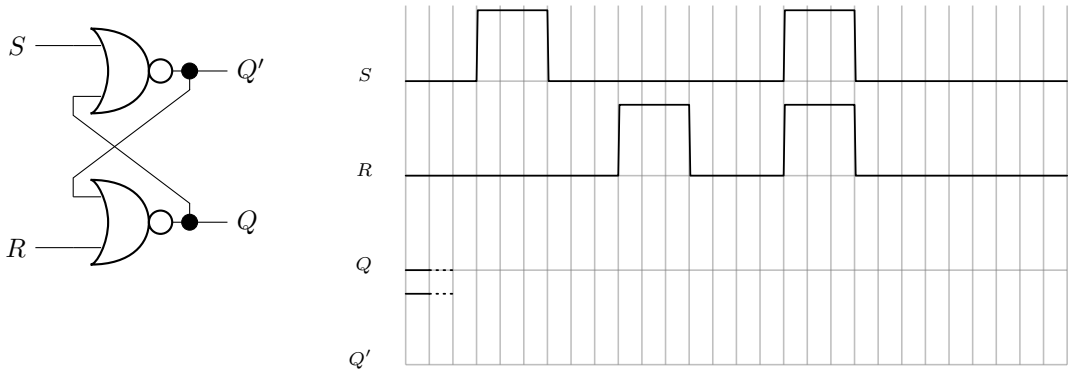
Due at the end of your registered lab session (130 points)

Objectives

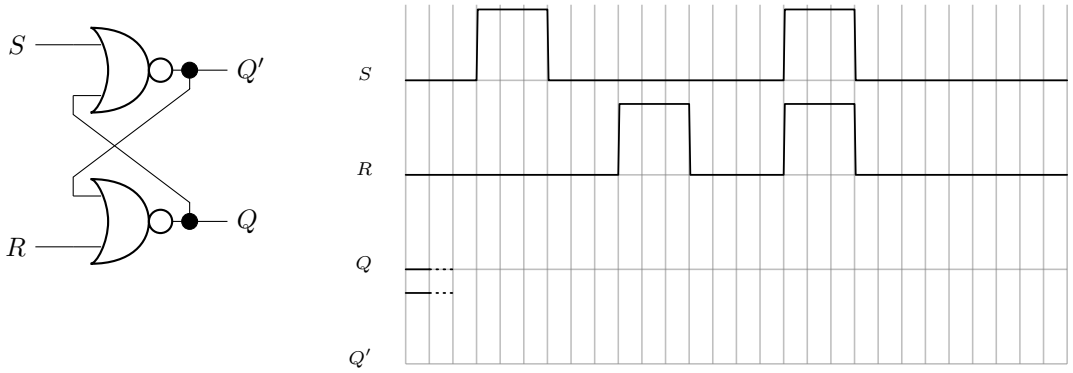
- Theory Objectives:
  - R/S and D latches
  - Master-slave flip-flops: R/S, D, J/K and T
  - Basic register design with `set` and `clear` (i.e. `reset`) capability.
  - Register architectures for implementing shift-register and binary counter modules.
- Verilog Syntax and Design Objectives
  - Verilog description of level-sensitive transparent latches vs edge-sensitive flip-flops.
  - Inferred latches and flip-flops.
  - Synthesizable Structural or Behavioral designs, vs Data Flow modeling techniques.

1 Pre-Lab Exercises

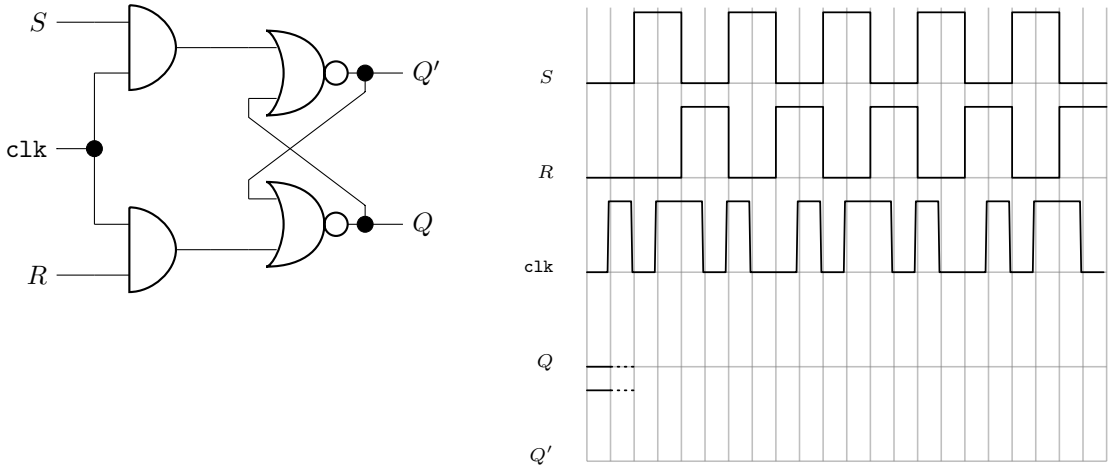
1. A standard RS latch is shown below. Assume each gate has a delay of one "time unit" (`tu` for short). A grid of time units is shown in the diagram below, along with the S and R signals at each time. The initial values of Q and Q' are shown. Complete the timing diagram for signals Q and Q', and explain what will happen at the end after $S = R = 1$.



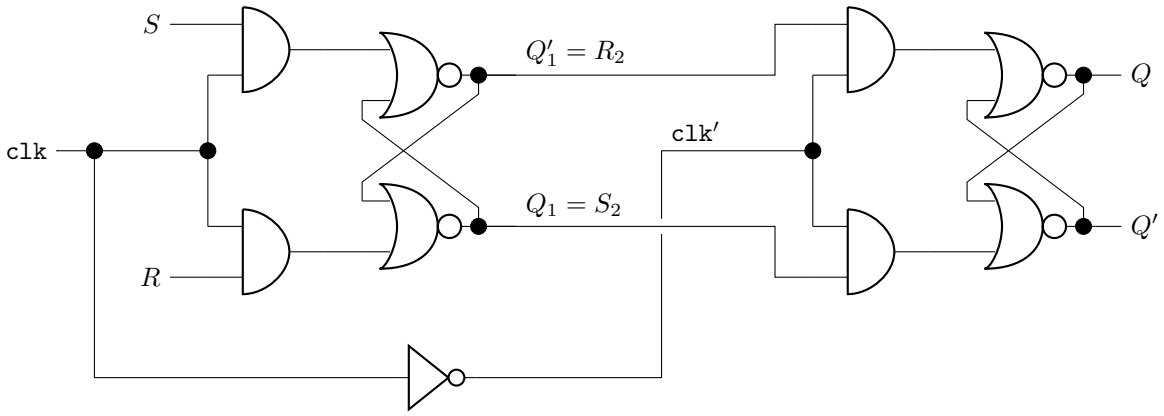
2. Repeat exercise 1 under the assumption that the second (bottom) NOR gate has a delay of 2`tu`. How will that affect the events at the end?

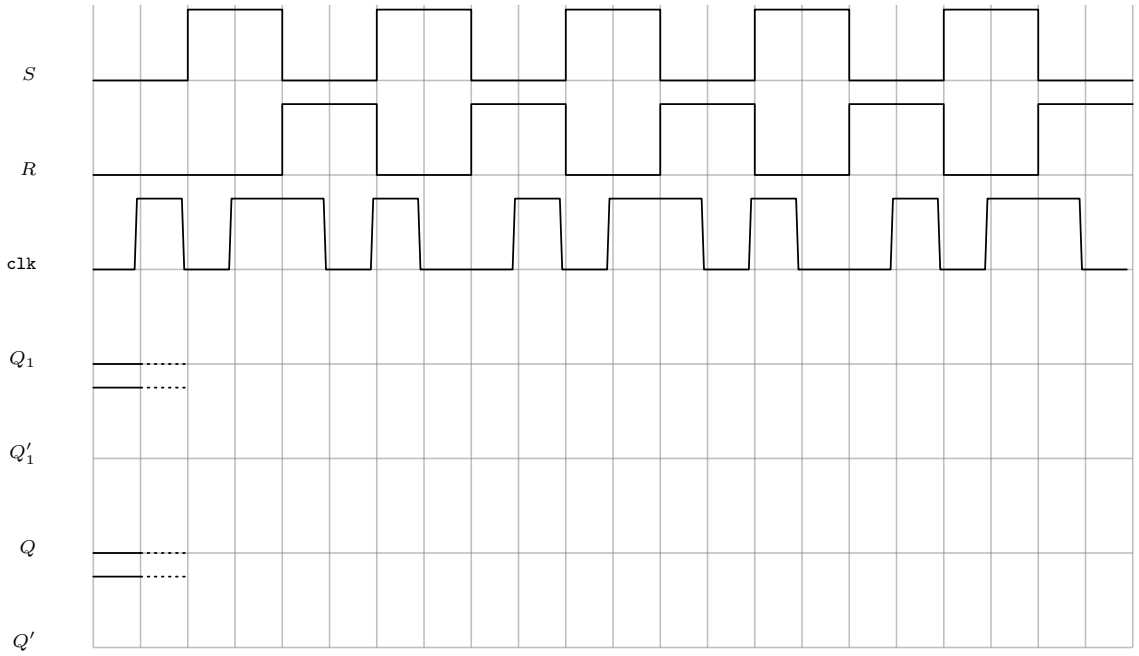


3. A clock-enabled RS latch is shown below, along with some signal timings. Assuming effectively zero gate delay, predict the signal outputs Q and Q'.

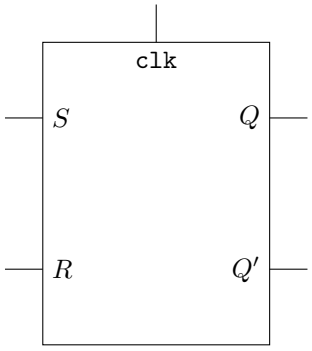


4. An edge-sensitive master/slave RS flip-flop is shown below, along with some signal timings. Assuming effectively zero gate delay, predict the signal outputs Q and Q'.





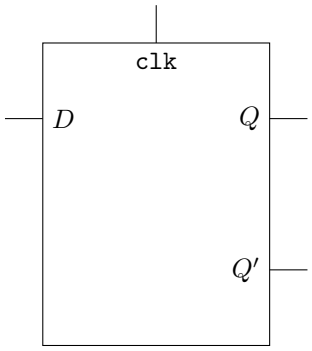
5. The flip-flop symbol shown below represents the RS flip flop schematic from the previous exercise. Add the necessary additional components to convert it to a D flip flop.

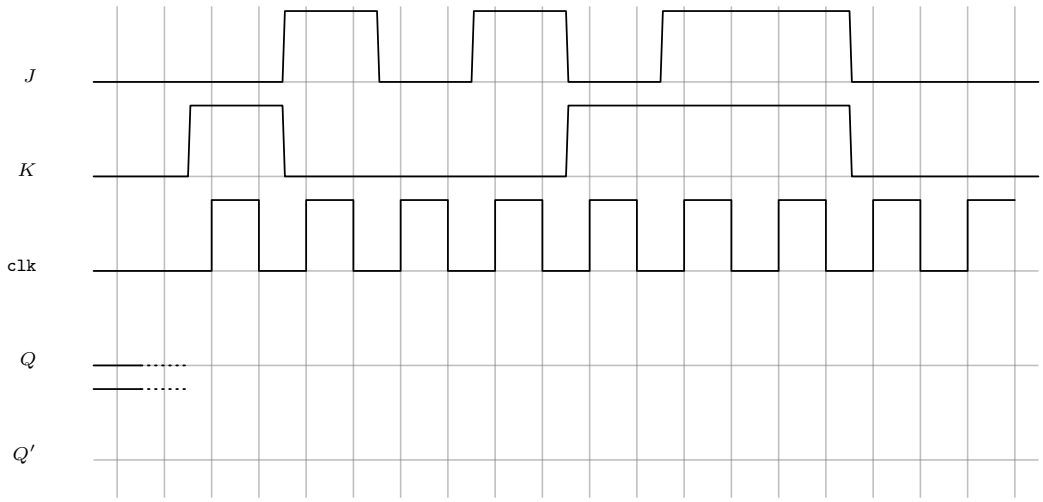


6. A J/K flip-flop is an edge-sensitive memory element with two inputs J and K, with behavior defined by the truth table shown below. Recall that the notation Q^- refers to the value of Q immediately preceding the clock edge.

J	K	Q
0	0	Q^- hold state (no change)
1	0	1
0	1	0
1	1	$\sim Q^-$ toggle state

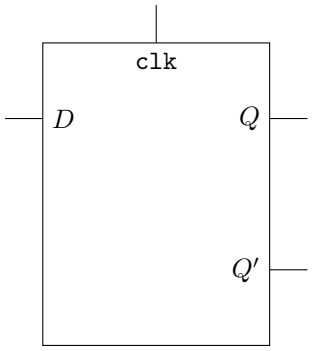
The flip-flop symbol shown below represents the D flip flop you designed in the previous exercise. Add additional components to convert it to a J/K flip flop, then complete the timing diagram below.





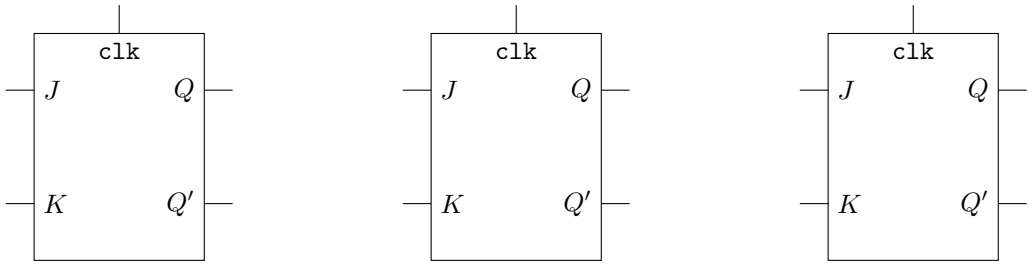
7. Modify your D flip flop again to produce a 1-bit register with extra inputs `rst`, `shift`, D_0, and these added functions:

Function Name	Signal	Operation
Reset	`rst`	when `rst == 1`, Q := D_0
Shift	`shift`	when `rst == 0` and `shift == 1`, Q := D
Hold	`shift'`	when `rst == 0` and `shift == 0`, Q := Q



8. Show how you can interconnect J/K flip-flops to build a three-bit counter with the following functions:

Function	Signal	Description
Reset	`rst`	When `rst==1`, set all flip-flops to zero
Increment	`incr`	When `rst==0` and `incr==1`, increment the binary count by 1.



2 RS Latch: Data Flow Model

One of Verilog's many uses is *hardware modeling*, which provides realistic simulations that can include things like signal delays and environmental effects. In this step, we will examine a *data flow model* (DFM) of a D flip-flop made from RS latches. A data flow model accounts for physical delays as a signal moves from one gate to another. We have to carefully distinguish DFMs from synthesizable designs, since **we cannot directly synthesize or modify physical gate delays**. Therefore *in this section* you will not program this design onto your Basys board, you will just complete a simulation analysis. The purpose of a DFM is to gain deeper understanding of physical signal flow, which sometimes helps us discover rare faults, hazards and glitches, or identify opportunities for physical optimization. The DFM is essentially the same analysis that you performed in Pre-Lab Exercises 1–4.

In Xilinx ISE, create a new project called `DFF_dataflow`, and then create a new Verilog module by the same name. Your code will model the physical structure of the RS latch, with extra logic to implement a D latch:

```
`timescale 1ns / 1ps

module latch_dataflow(
    input d,
    output q,
    input clk
    );


       wire S;
       wire R;
       wire qb;

       assign #1 S=~d&clk;
       assign #1 R=d&clk;
       assign #2 qb=~(R|q);
       assign #2 q=~(S|qb);
```

Notice that each `assign` statement includes a # statement to declare the assignment's delay. The amount of delay is in units specified by the `timescale` directive, 1 ns in this example. This is the essence of a DFM. Up to now, your Verilog models have presumed instantaneous assignments. That's usually sufficient for fully synchronous clocked designs that use flip-flop registers. When using combinational logic with level-sensitive latches, the pattern of delays can sometimes cause unintended side effects, so there are times when we need to take a close look at it.

Next, create a module named `DFF_dataflow` and build it from two instances of your `latch_dataflow` module:

```
`timescale 1ns / 1ps

module DFF_dataflow(
    input d,
    output q,
    input clk
    );

       wire q1;
       wire clkb = ~clk;

       latch_dataflow d1(.d(d),.q(q1),.clk(clk));
       latch_dataflow d2(.d(q1),.q(q),.clk(clkb));
endmodule
```

Finally, create a Verilog Test Fixture to simulate the model. In order to reproduce the `clk` pattern from Pre-Lab Exercises 3–4, we will create a separate "master clock" signal called `mclk` as the driver for the testbench logic. The irregular `clk` pattern from the Pre-Lab was generated from a three-bit `count` signal as shown in the code below.

```
`timescale 1ns / 1ps

module DFF_test;

	// Inputs
	reg d;
	reg clk;

	// Outputs
	wire q;

	// Other signals:
	reg mclk;
	reg [2:0] count;

	// Instantiate the Unit Under Test (UUT)
	DFF_dataflow uut (
		.clk(clk),
		.d(d),
		.q(q)
	);

	initial begin
		// Initialize Inputs
		clk   = 0;
		d     = 0;
		count = 0;
		mclk  = 0;
		clk   = 0;

		forever #10 mclk=~mclk;
	end

	always @(posedge mclk) begin
		count = count + 1;
		clk   = count[2]^count[0];
		d     = count[1];
	end
endmodule
```

After completing your test fixture, run a syntax check and then simulate the model. In the ISim window, click to expand the hierarchy view in the upper left panel. Expand the `uut` module to see the `Dlatch_dataflow` modules beneath it. Click on each of them, and you should notice that their internal signals appear in the panel just to the right of the hierarchy view. Select the `S`, `R`, `q` and `qb` signals, and drag them into the waveform viewer panel. Do this for both of the `Dlatch_dataflow` submodules, then restart the simulation. You should see a pattern of signals that looks very similar to your predictions from the Pre-Lab. Carefully compare the ISim signals to your predictions and verify the same pattern of information. Also take note of any differences that appear. Use the cursors to measure the signal delays, and verify that they match the delays specified in your source code.

3 D Latch: Behavioral Implementation

Verilog provides a variety of behavioral methods for implementing latches and flip-flop. In fact, **any incomplete conditional in an `always` block will be inferred as a latch.** As a simple example, create a new Verilog module called `dlatch`, defined as follows:

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

        always @(clk or d) begin
                if (clk)
                        q <= d;
        end

endmodule
```

This simple module is sufficient to define the behavior of a clocked D latch. When `clk` is asserted, the output `q` is assigned to equal the input `d`. When `clk` is low, the output `q` is held constant regardless of the value of `d`. Next, create a new Verilog module named `DFF`, and copy all the code from `DFF_dataflow`. Modify the `DFF` code so that it instantiates `dlatch` instead of `latch_dataflow`. Then create a new Verilog test fixture, and copy over all code from the previous dataflow test. Then edit the test fixture so that it tests `DFF` instead of `DFF_dataflow`, and run the simulation. You should see the same pattern of outputs.

Since this behavioral model is fully synthesizable, you may proceed to create an implementation constraint file. To demonstrate your flip-flop, you will connect the `clk` signal to switch `SW0` (**for this exercise, we will NOT use the usual sys_clock pin**), and the `d` signal to `SW1`. In order to drive the clock from a board switch, Xilinx requires disabling the `CLOCK_DEDICATED_ROUTE` parameter as shown in the code below (this is usually not recommended, but we will do it for this one demonstration). Finally route the output `q` to one of the LEDs.

```
# Switch SW[0]:
set_property PACKAGE_PIN V17 [get_ports {clk}]
set_property IOSTANDARD LVCMOS33 [get_ports {clk}]
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets clk_IBUF]

# Switch SW[1]:
set_property PACKAGE_PIN V16 [get_ports {d}]
set_property IOSTANDARD LVCMOS33 [get_ports {d}]

# LED 0:
set_property PACKAGE_PIN U16 [get_ports {q}]
set_property IOSTANDARD LVCMOS33 [get_ports {q}]
```

Set `dlatch` as your top module, then Synthesize, Implement, and Generate the bit file for your design. Program it onto your Basys board and demonstrate the transparent latch function. Turn on `clk` and observe that `q` responds when you toggle the `d` switch. Turn off `clk` and observe that `q` no longer responds. Demonstrate this to the TA.

Next, set `DFF` as the top module and repeat the Synthesis, Implementation and programming procedure. Verify that the output `q` responds to the value that `d` has *at the precise time when the clock switches*, but it is not at all sensitive to `d` at other times. Demonstrate this result to the TA. **This is the main difference between "level sensitivity" in latches (which respond when `clk` is high) vs "edge sensitivity" in flip-flops (which respond when `clk` changes value).** Edge sensitivity is essential for reliable synchronous operation in sequential circuits.

4 J/K Flip Flop: Structural Implementation

Now create a new Verilog module called `JKFF`. Instantiate your `DFF` module and add any other logic needed to implement the J/K flip-flop you designed in Pre-Lab exercise 6. Simulate your J/K flip-flop using a Verilog test fixture based on the test procedure shown below:

```
module Dlatch_test;
        // Inputs
        reg j;
        reg k;
        reg clk;

        // Outputs
        wire q;

        integer idx;

        // J/K test patterns (read from right to left)
        wire [11:0] jvals = 8'b0110_1010;
        wire [11:0] kvals = 8'b0111_0001;

        // Instantiate the Unit Under Test (UUT)
        JKflipflop uut (
                .clk(clk),
                .j(j),
                .k(k),
                .q(q)
        );

        initial begin
                // Initialize Inputs
                j = 0;
                k = 1;
                idx=0;
                clk = 1;

                forever #10 clk=~clk;
        end

        always @(posedge clk) begin
                // use modulo arithmetic to increment idx from 0 up to 7,
                // then wrap back to 0:
                idx = (idx + 1) % 8;

                j = jvals[idx];
                k = kvals[idx];
        end
endmodule
```

Verify that your flip-flop's behavior matches the prediction from your pre-lab exercise, then proceed to Synthesize, Implement and Program your board using `SW0` as the `clk`, `SW1` as `j`, and `SW2` as `k`, and `LED0` as `q`. Verify proper function on your Basys board and demonstrate it to the TA.

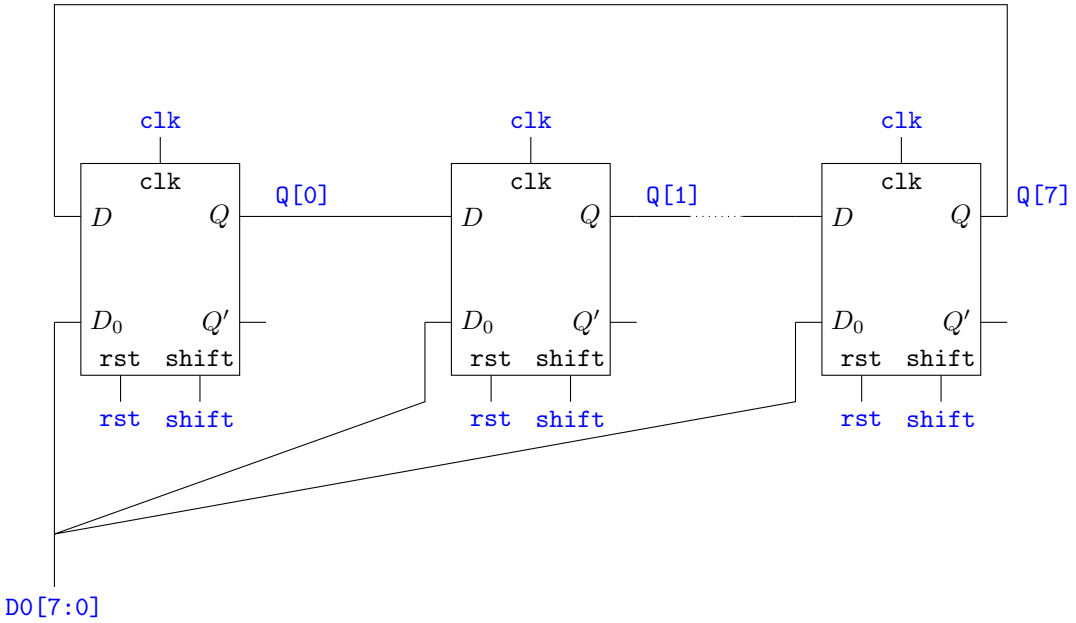
5 Shift-Register for Rotating LED Display

Now create a new project called `rotateLED`, and add the following module sources into the project:

- `dlatch.v`
- `DFF.v`
- `clockdivider.v` (from Lab 1)

Now make a new Verilog module called `Dregister`. Create an instance of your `DFF` module, along with any other logic required to implement the multi-function register that you designed in Pre-Lab exercise 7.

Then create a new Verilog module called `rotateLED.v` to implement a multi-bit shift register module based on the schematic shown below. Three registers are shown; **your complete design should use eight registers. You should use a generate block to define the register instances and their interconnections.** For better clarity, this schematic uses connection-by-label, with the top-level module signals shown in blue. At every location labeled `shift`, the same `shift` signal wire should be connected.



The top-level I/O ports are shown in the table below, along with their intended Basys board mappings.

direction	name	vector	bits	board resource
input	mclk	no	1	MCLK
input	shift	no	1	BTN0
input	rst	no	1	BTN1
input	D0	yes	7:0	SW0 to SW7
output	Q	yes	7:0	LD0 to LD7

Use an instance of your `ClockDivider` module to generate the `clk` signal for your shift register, so that you can observe sequential events in your implementation. You will probably want to edit the `ClockDivider` module to use a divider ratio around 25,000,000 (decimal) so that the local clock is visible but not too slow. Prepare an Implementation Constraints File for these connections, then Synthesize, Implement and program your design onto the Basys board. Use the switches to define a non-zero initial state, then press (and hold) `BTN1` to reset the shift register. Then, press and hold `BTN0` to start shifting the pattern. You should see the pattern rotate on the LED display. The process may run very slowly, depending on the divider ratio used in your `ClockDivider` module.

5.1 Counter Module

Lastly, create a new project called `counter` to implement the J/K flip-flop counter that you designed in Pre-Lab exercise 8. Add the following sources to your project:

- `dlatch.v`
- `DFF.v`
- `JKFF.v`
- `ClockDivider.v`

Create a new Verilog source to define the top-level `counter` module with the following I/O signals:

direction	name	vector	bits	board resource
`input`	`mclk`	no	1	`MCLK`
`input`	`incr`	no	1	`BTN0`
`input`	`rst`	no	1	`BTN1`
`output`	`Q`	yes	7:0	`LD0` to `LD7`

Use a `generate` loop to instantiate J/K flip-flops and build an eight-bit counter module. Initially prepare the design with no `ClockDivider` module. Create a Verilog test fixture to simulate your model and verify that your counter increments from 0 up to 255 when `incr` is asserted. Once verified, add the `ClockDivider` module and then Synthesize, Implement and program the design onto your Basys board. The LEDs should clear when you press `BTN1`, and should show a successive binary sequence when you press and hold `BTN0`. Demonstrate your design to the TA.

6 TA Checkoff

- (32 points) Complete pre-lab work prior to start of the lab.
- (12 points) RS latch and DFF dataflow simulation.
- (12 points) Behavioral D latch and structural DFF simulation and implementation.
- (20 points) J/K flip-flop simulation and implementation.
- (24 points) Shift Register design, simulation and implementation.
- (30 points) Counter design, simulation and implementation.