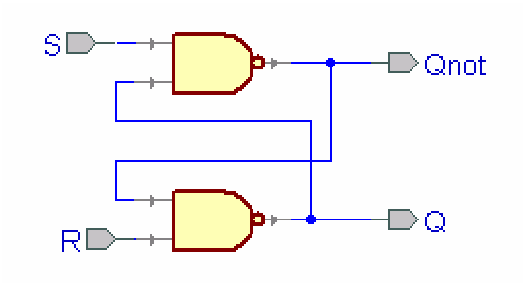
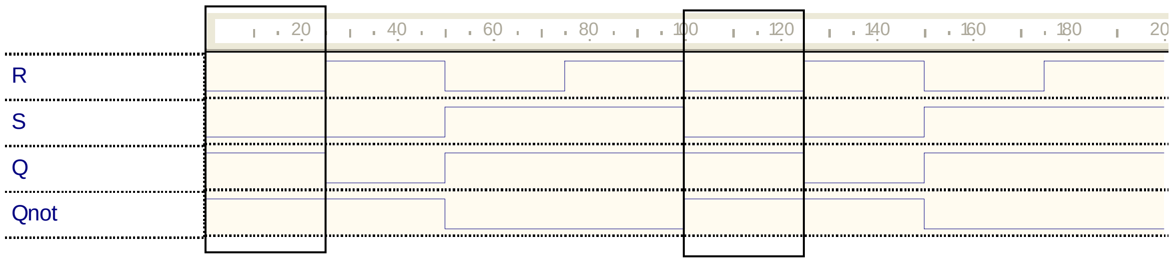
Exercise 6.1

The cross-coupled NAND gates would produce the following latch:



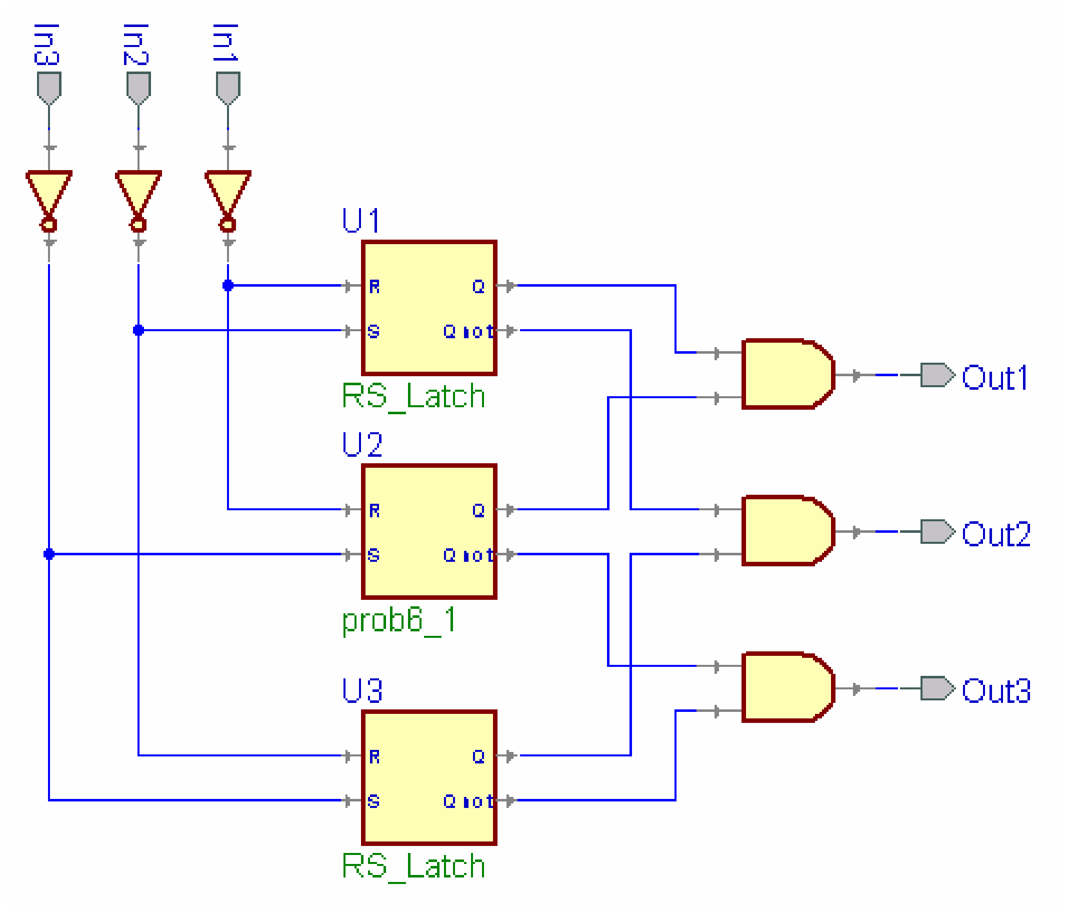
This produces the following waveform diagram:



The shaded regions above denote forbidden inputs because they cause both Q and Q' to be asserted. The input signal S causes Q^+ to be asserted, and the input signal R causes Q^+ to be reset; however when both R and S are asserted Q^+ will take whatever the input value of Q is.

S	R	Q	Q^+
0	0	X	Forbidden
0	1	X	0
1	0	X	1
1	1	0	0
1	1	1	1

Exercise 6.2



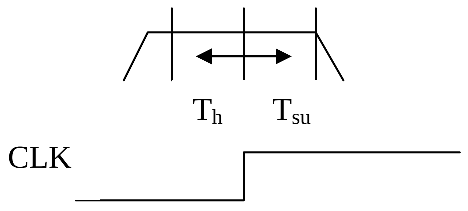
Extending this solution to either n = 12 or n = 30 inputs requires

$$\sum_{i=2}^{n}(i-1)$$

R-S latches. Where each output would have an AND gate with n-1 inputs. This would cause significant delay and could be quite noticeable.

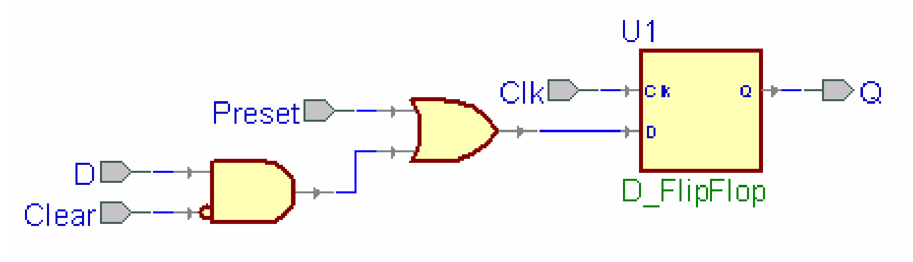
Exercise 6.3

If there were such a thing as negative setup and hold time it might look something like the following:

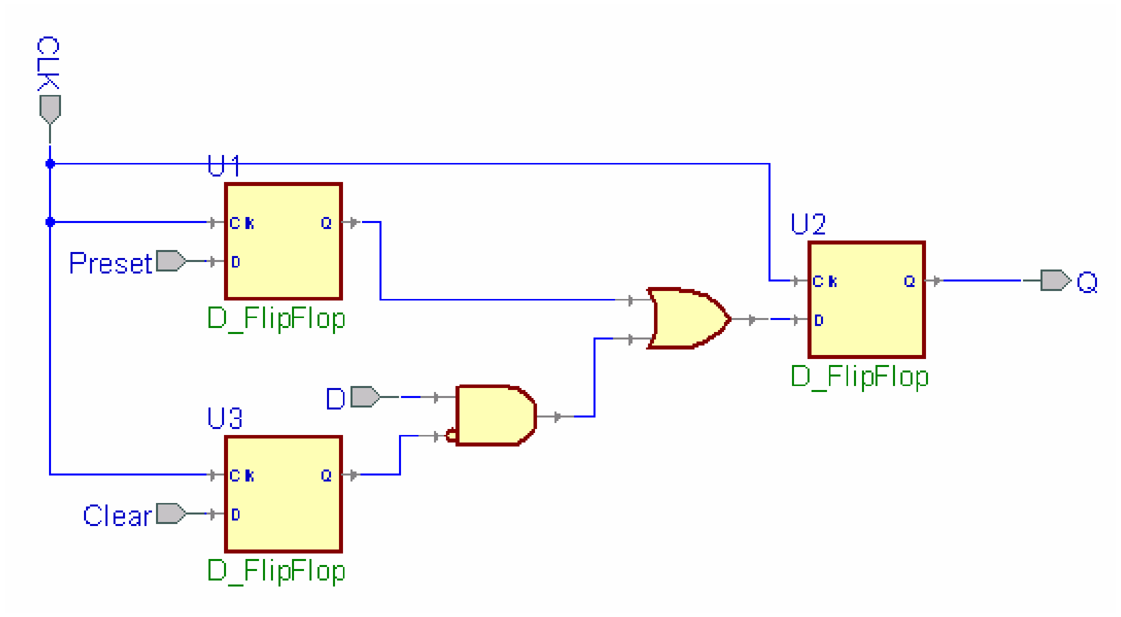


This however does not make a lot of sense because T_{su} is defined to be the time before the CLK event in which the signal must remain stable, and T_h is the time after the CLK event in which the signal must remain stable. Thus negative times would in effect switch the function of T_{su} and T_h so that T_{su} is the amount of time after the clock and T_h is the time before the clock.

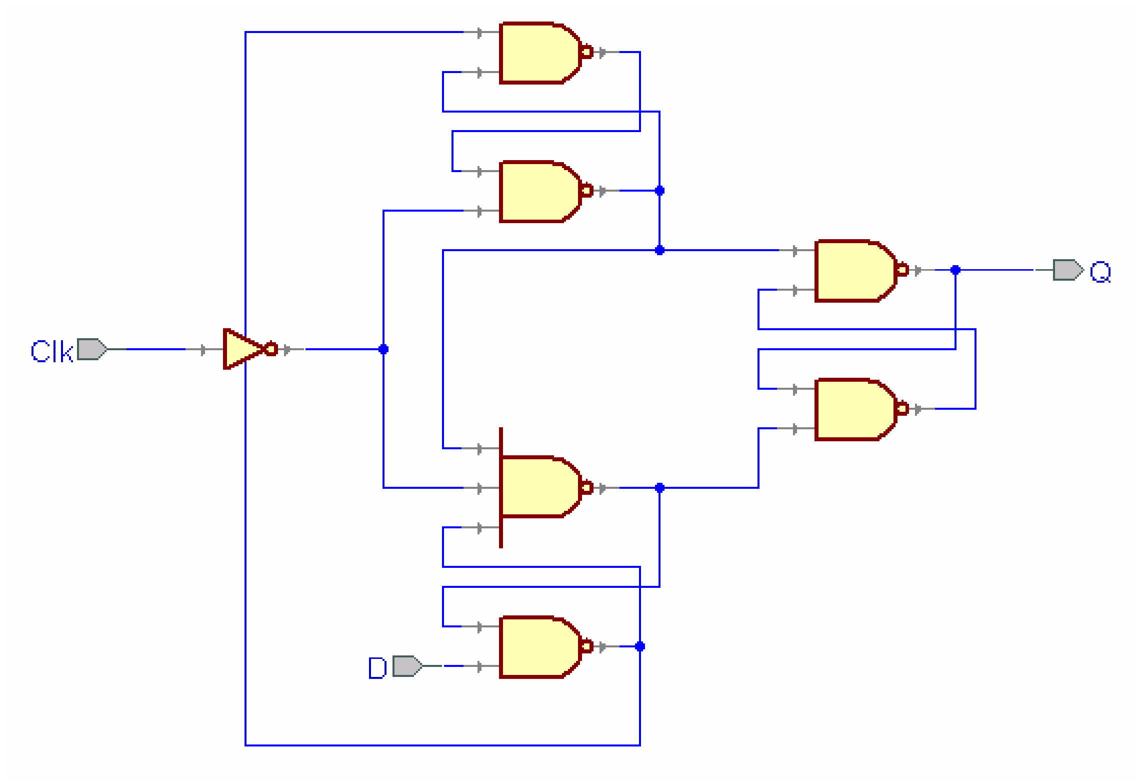
Exercise 6.4



Exercise 6.5

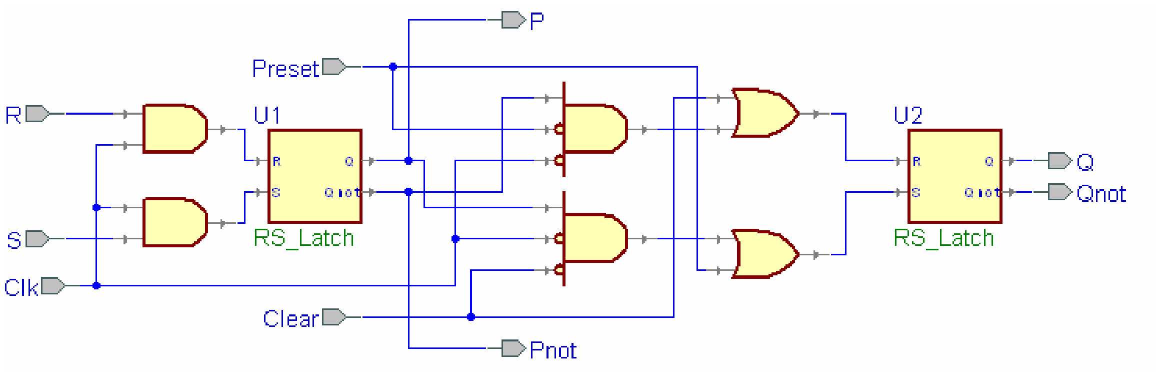

CLK
U1
Clk
Q
D
Preset
D_FlipFlop
U2
Clk
Q
D
Q
D_FlipFlop
D
U3
Clk
Q
D
Clear
D_FlipFlop

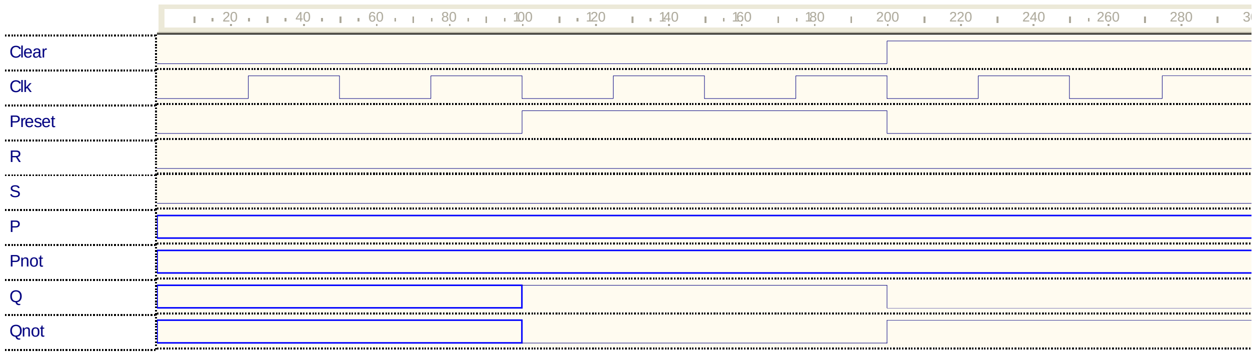
Exercise 6.6



Removing the inverter on the clock will make this a positive-edge triggered D-flip flop.

Exercise 6.7


P
Preset
R
U1
RS_Latch
S
Clk
Clear
Pnot
U2
Q
Qnot
RS_Latch


20
40
60
80
100
120
140
160
180
200
220
240
260
280
Clear
Clk
Preset
R
S
P
Pnot
Q
Qnot

Exercise 6.8

Ones-catching happens when one of the inputs glitches and gets asserted while the clock is high. This causes the Master Output to pickup the signal and get set, even though the glitch is finished before the clock goes low. After the Master Output picks up the glitch, the Slave Outputs catch the one as well.

Zeroes-catching can happen depending on the construction of the Master-Slave flip-flop. If instead of constructing the flip-flop with only NOR gates, NAND gates were used (i.e. a R’-S’ flip-flop), then the Master-Slave will pick up zeroes instead of ones.

<u>Exercise 6.9</u>

J-K flip-flop's characteristic equation is:

$$Q+ = QK' + Q'J$$

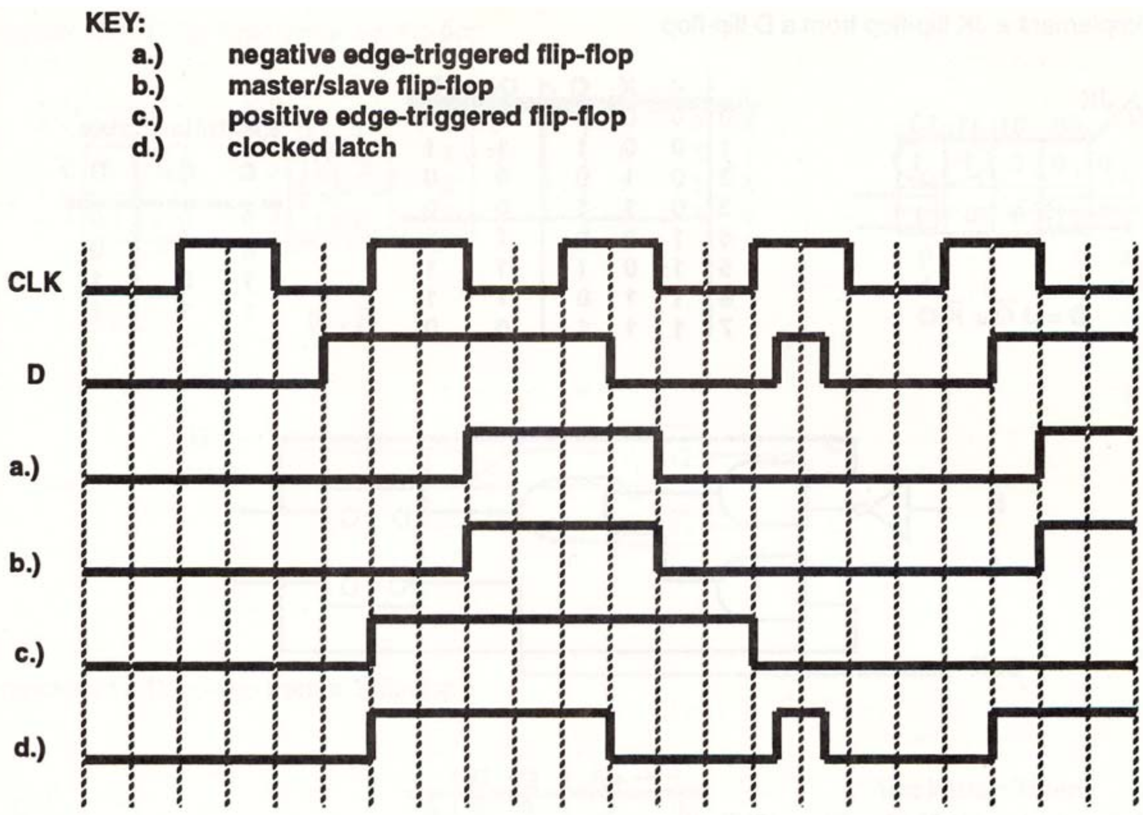
Substituting D' for K and D for J:

$$Q+ = QD + Q'D = (Q + Q')D = D$$

Which is precisely the characteristic equation for a D flip-flop:

$$Q+ = D$$

Exercise 6.10


KEY:
a.) negative edge-triggered flip-flop
b.) master/slave flip-flop
c.) positive edge-triggered flip-flop
d.) clocked latch
CLK
D
a.)
b.)
c.)
d.)

Exercise 6.11

(a) False. The level-sensitive latch continuously updates its outputs based on its inputs only when the clock is asserted. When the clock is unasserted, the outputs hold their previous state, and the inputs can fluctuate in order to compute the next values for the outputs.

(b) False. The time after the clock edge until the outputs change is called the propagation delay, or T_p. As stated at the end of 6.2.1, T_p must be greater than T_h in order for flip-flops to be cascaded properly.

(c) True. Provided that forbidden inputs are not fed into the clocked latches or flip-flops. Because there is zero setup or hold times, then events occur instantaneously after a clock-edge, this means that there is no time at which a signal could be misread during the setup or hold times.

(d) False. While it is true that a master-slave flip-flop does behave similarly to a clocked latch, and that its output can change only near an edge of the clock, depending on the design of the master-slave flip-flop, this change can occur at the rising edge, the falling edge, or both (but both is rarely done in practice).

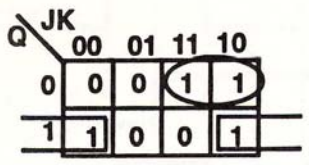
(e) False. The D Flip-flop is constructed using an R-S master-slave latch with an additional inverter for the signal D'. In the case of the J-K flip flop, there is an R-S master-slave latch as well as two additional AND gates for the inputs and feedback. Therefore the J-K takes one more gate than the D Flip-flop.

Exercise 6.12

a) Master-slave RS flip-flop
b) Master-slave D flip-flop
c) Master-slave T flip-flop
d) Clocked RS latch
e) Positive edge-triggered D flip-flop

Exercise 6.13

Implement a JK flip-flop from a D flip-flop

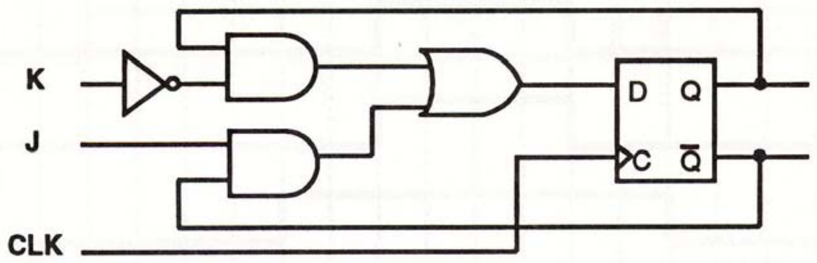


$$D = J\overline{Q} + \overline{K}Q$$

	J	K	Q	Q+ →	D
0	0	0	0	0	0
1	0	0	1	1	1
2	0	1	0	0	0
3	0	1	1	0	0
4	1	0	0	1	1
5	1	0	1	1	1
6	1	1	0	1	1
7	1	1	1	0	0

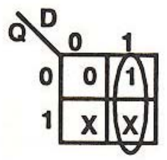
Excitation Table

Q	Q+	D
0	0	0
0	1	0
1	0	1
1	1	1

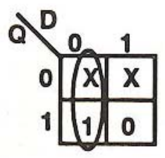


Exercise 6.14

Implement a D flip-flop from a J-K flip-flop



J = D

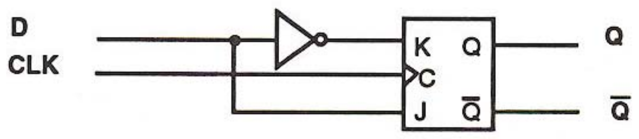


K = $\overline{D}$

	D	Q	Q+ →	J	K
0	0	0	0	0	X
1	0	1	0	X	1
2	1	0	1	1	X
3	1	1	1	X	0

Excitation Table

Q	Q+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0



Exercise 6.15

Implement a D flip-flop from a T flip-flop

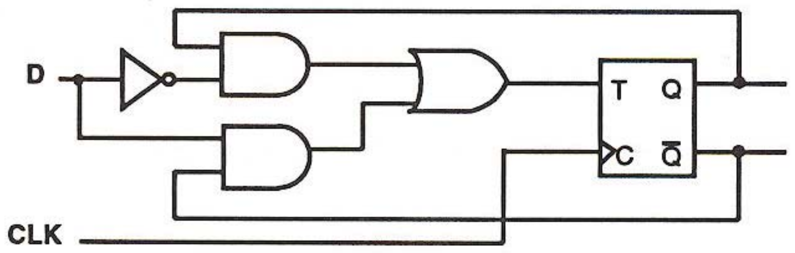
Q \ D	0	1
0	0	1
1	1	0

$T = D\overline{Q} + \overline{D}Q$

	D	Q	Q+ →	T
0	0	0	0	0
1	0	1	0	1
2	1	0	1	1
3	1	1	1	0

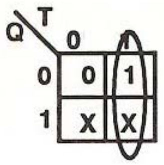
Excitation Table

Q	Q+	T
0	0	0
0	1	1
1	0	1
1	1	0

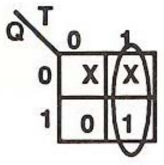


Exercise 6.16

Implement a T flip-flop from a J-K flip-flop



J = T

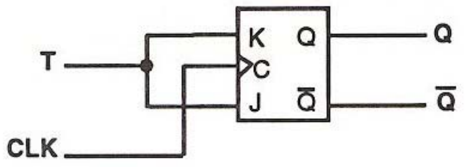


K = T

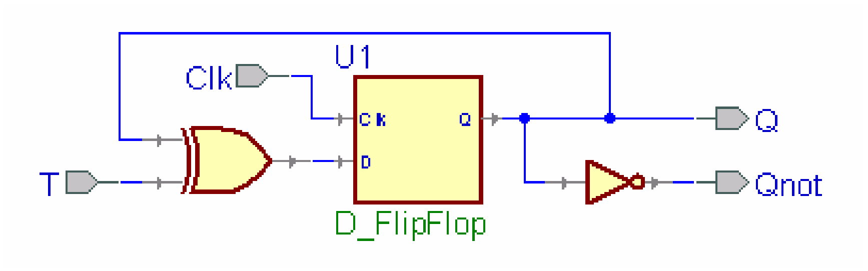
	T	Q	Q+ →	J	K
0	0	0	0	0	X
1	0	1	1	X	0
2	1	0	1	1	X
3	1	1	0	X	1

Excitation Table

Q	Q+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0



Exercise 6.17



In order to get the T flip-flop behavior out of the D flip-flop, whenever T is unasserted, Q will have to maintain its current value, and whenever T is asserted, Q^+ will take on Q’. This is achieved by using the equation: $Q^+ = Q \otimes T$.

Exercise 6.18

The worst case time skew is calculated by taking the equation:

$$T_p > T_{skew} + T_h$$

Which in this case takes on the form:

$$3.6 > T_{skew} + 0.5$$

or

$$3.1 > T_{skew}$$

Exercise 6.19

(a) The problem specifies that T_{su} = 20 ns, T_{pd} = 13 ns, and T_h = 5 ns. Frequency is measured in cycles per second, so f = 1 / T_{period}, where T_{period} is defined to be greater than T_{su} + T_{pd}. Therefore the math shows:

$$T_{period} > T_{pd} + T_{comb} + T_{su}$$

$$T_{period} > 13 \text{ ns} + 0 \text{ ns} + 20 \text{ ns}$$

$$T_{period} > 33 \text{ ns}$$

$$f < 1 / T_{period}$$

$$f < 30.3 \text{ MHz}$$

(b) Since the clock period must account for the worst-case propagation delay we will use 100ns for the combinational logic delay. The calculation is the same as part (a). Thus:

$$T_{period} > T_{pd} + T_{comb} + T_{su}$$

$$T_{period} > 13 \text{ ns} + 100 \text{ ns} + 20 \text{ ns}$$

$$T_{period} > 133 \text{ ns}$$

$$f < 1 / T_{period}$$

$$f < 7.5 \text{ MHz}$$

Exercise 6.20

The system failure could be caused by the fact that an asynchronous signal is received every 200 ns, and the period of the 25 MHz receiver is 40 ns. In general, clock skew will provide enough difference between when a signal is sent and received, such that the signal is received properly and respects the setup and holding times of the synchronizer.

There are three common ways for dealing with this problem:

- Increasing the cycle time of the receiver, so that it has more time to determine respect the setup and hold times of its components.
- Adding a second synchronizer, which helps lower the probability of a synchronizer failure.
- Implement a handshaking methodology between the client and server, so that it is easy to confirm whether or not information has been communicated effectively.

The first option severely has the side-effect that it severally slows down the receiver, and is not very tolerable in today's high-performance requirements.

The second option helps reduces the error, but also reduces how quickly a signal can be interpreted and used in the receiving system. This can make a huge difference in systems that are trying to communicate quickly.

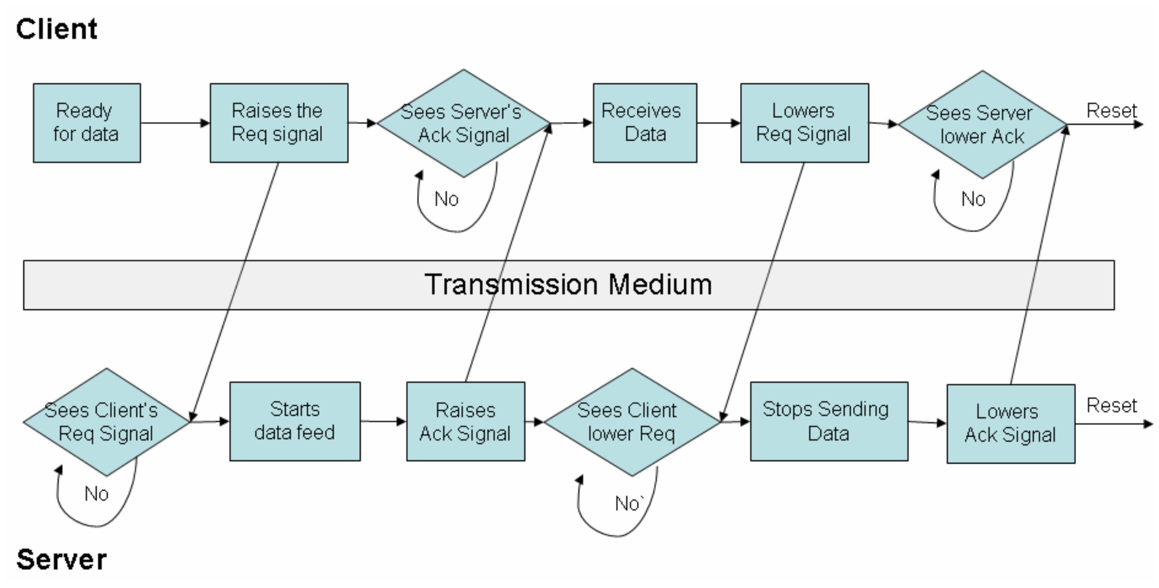
The final option is more commonly practiced, because it allows both the sender and receiver to communicate when they are ready to both send and receive signals. This coordination, helps reduce the amount of errors, and does not impose to much of a time penalty.

Exercise 6.21

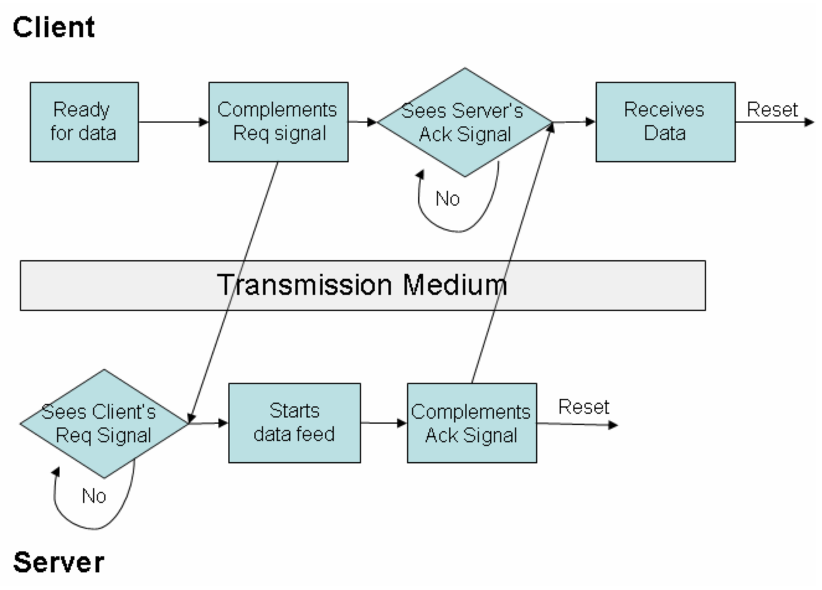
Placing two synchronizers in series helps reduce the probability of synchronizer failure because it is in effect providing the system two clock periods to stabilize instead of one. This effect is very similar to increasing the clock period of the receiver so that it has more time to decide. Even though it seems logical to put more synchronizers in series to even further reduce the probability of synchronizer failure, this approach has the severe penalty in that each synchronizer adds another full clock period before the input can be used. The Oscilliscope sketch in Figure 6.41 demonstrates that, in general, signals have low probability of staying in a metastable state for long durations of time. This means that after two synchronizers, the series is paying heavy time delays for minor improvements to the probability.

Exercise 6.22

The figure below shows the process for four-cycle handshaking:

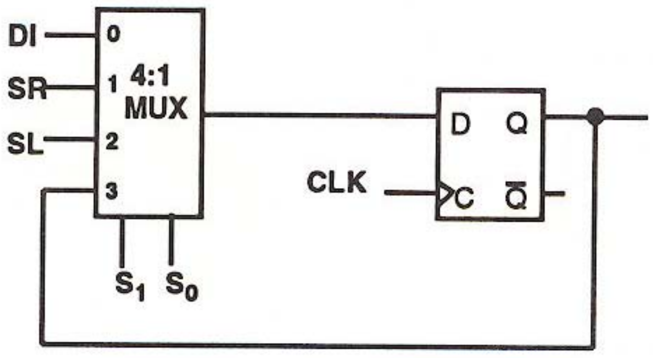


In this figure, reset means that both Client and Server are back in their initial states, with the client preparing to receive data, and the server waiting for the client's request.



In two-cycle handshaking, the Client and Server both keep track of the state provided by the other. This way the Server can recognize when the Client complements the Req signal, and the Client can in turn recognize the Server complementing the Ack signal. In general, maintaining the state in a two-cycle handshaking model is more complicated than implementing the four-cycle handshaking model.

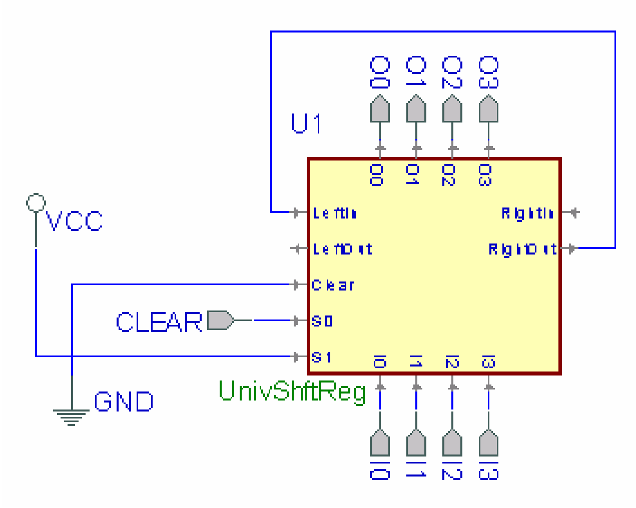
Exercise 6.23



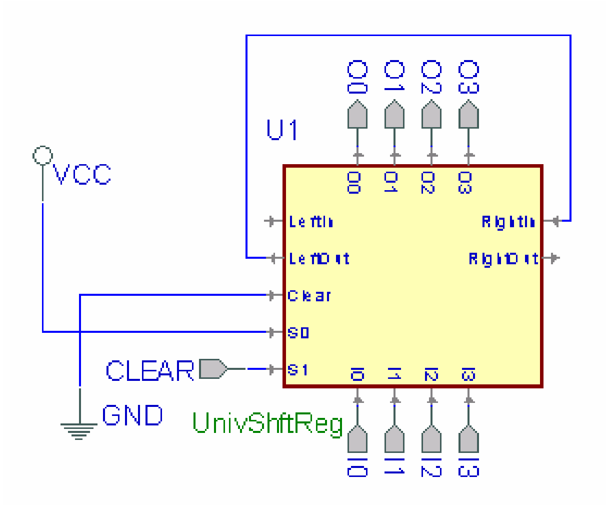
<u>Exercise 6.24</u>

In the diagrams below, the CLEAR signal is used to do a parallel load of data into the universal shift register. The clear functionality is not required, but in general will provide more interesting, and useable shift registers.

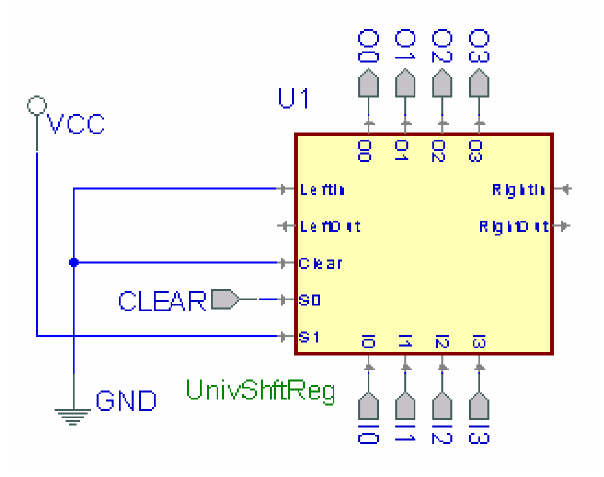
(a) For the circular shift right, S1 will always be enabled, and the output from RightOut will be fed into LeftIn. If CLEAR is enabled as well, the shift register will load in values I0…I3 instead of performing the circular shift.



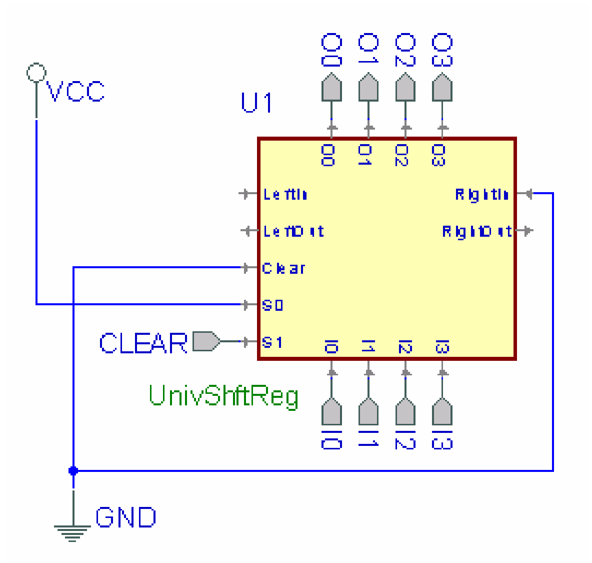
(b) The implementation of the left circular shift register is very similar to that of the right:



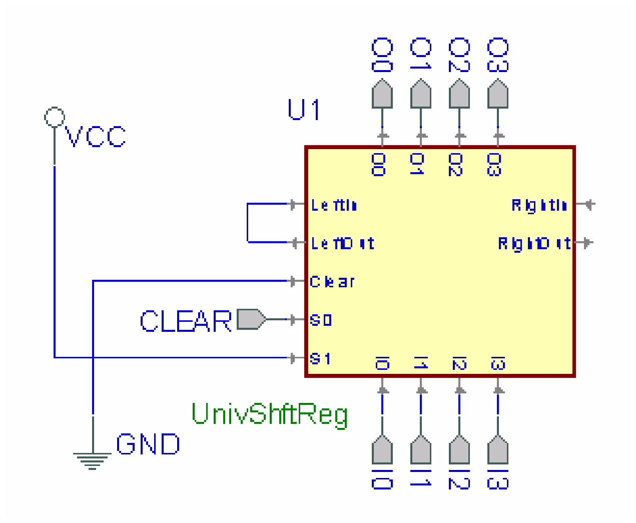
(c) The logic shift right:



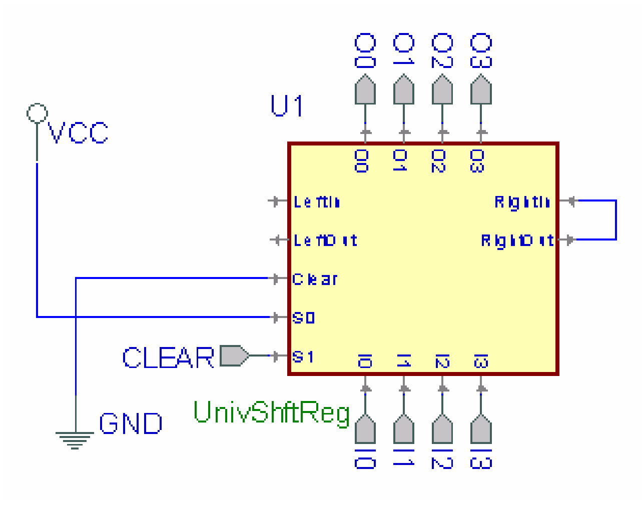
(d) The logic shift left:



(e) Arithmetic shift right:

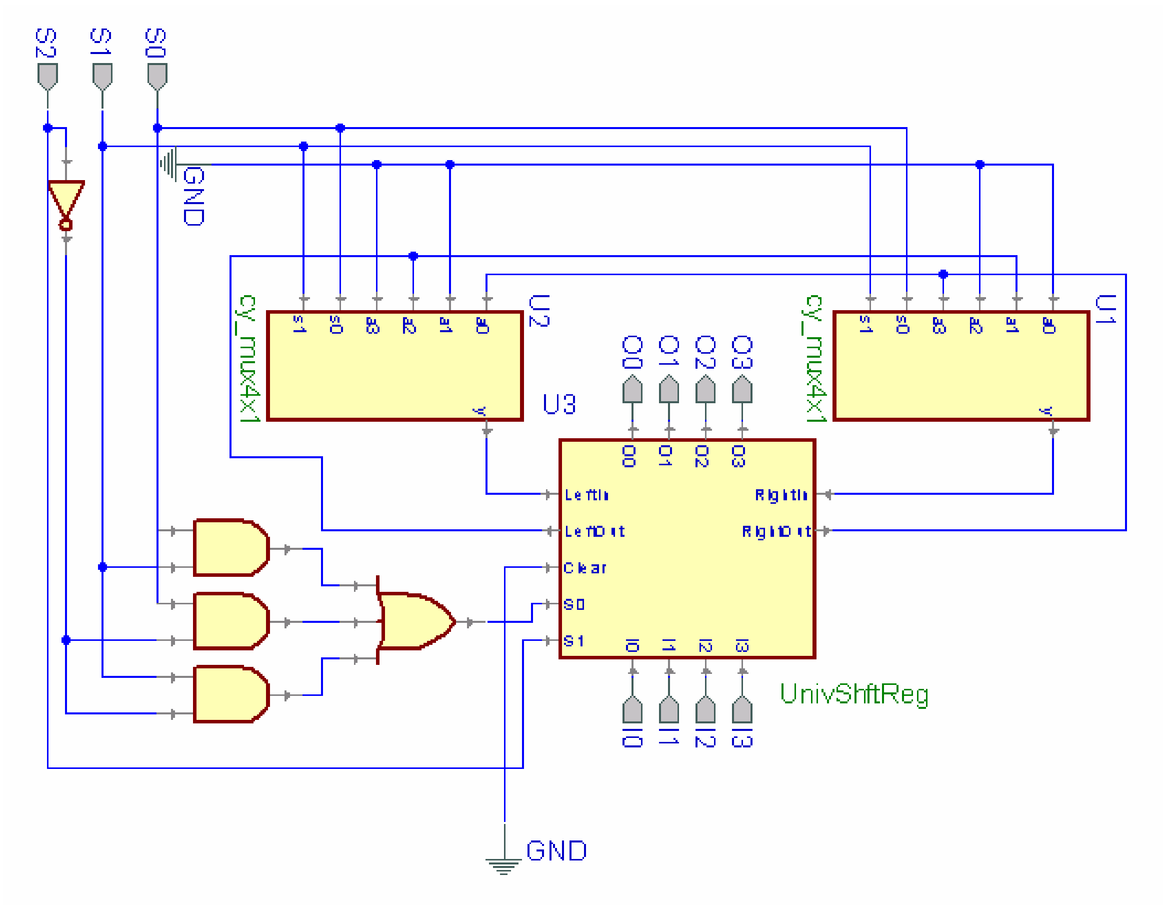


(f) Arithmetic shift left:



Exercise 6.25

(a) Here is the full shift register system:



(b) Since the 74194 uses S_0 and S_1 as its control signals, and those are already being used in this problem, let C_0 and C_1 represent the 74194 control signals instead.

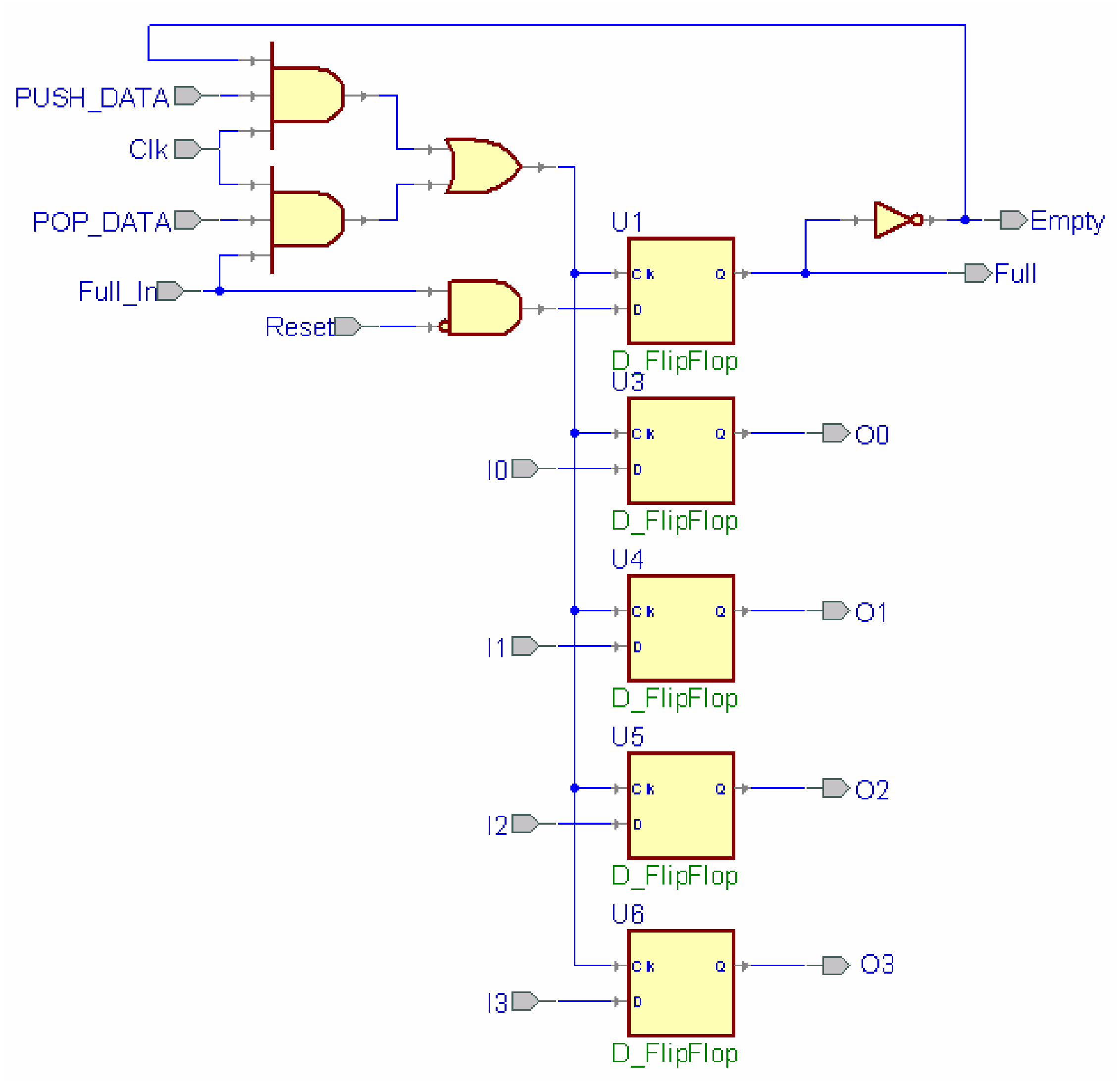
$C_0 = S_0S_1 + S_0S_2' + S_1S_2'$

$C_1 = S_2$

Notice that the shift register input functions are only dependent on S_0 and S_1. This is because S_3 alternates whether or not the function is a "left" or "right" operation, except where hold and preset are concerned. However, hold and preset do not look at the input values, and thus are don't-cares in terms of the the LeftIn and RightIn inputs.

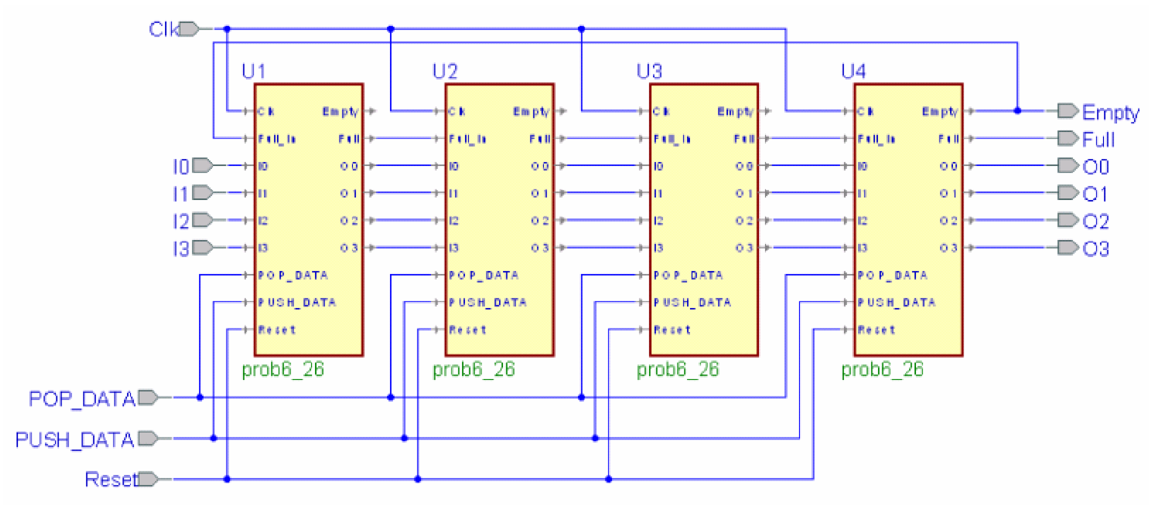
Exercise 6.26

Since there is a lot of repeated functionality, the implementation given is broken up into two main modules. The first module handles a single 4-bit word, and the shift register responsible for the Full and Empty status.



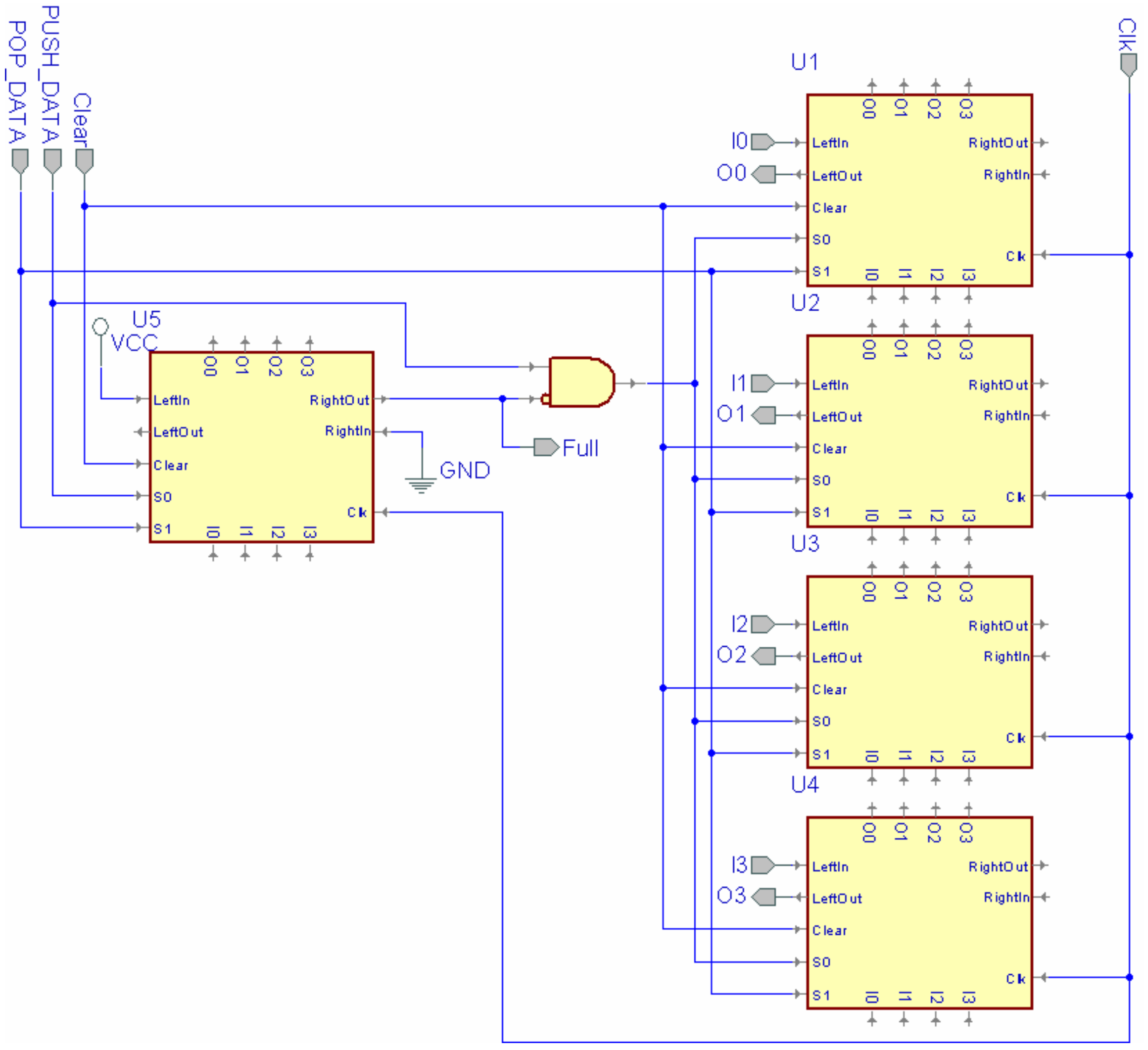
The restraints on Pushing and Popping are maintained, since the Clock signal will only be enabled if the module feeding Full_In is asserted and it is a Pop instruction or this module is outputting Empty and it is a Pop instruction.

The diagram below shows how to chain the individual components together.



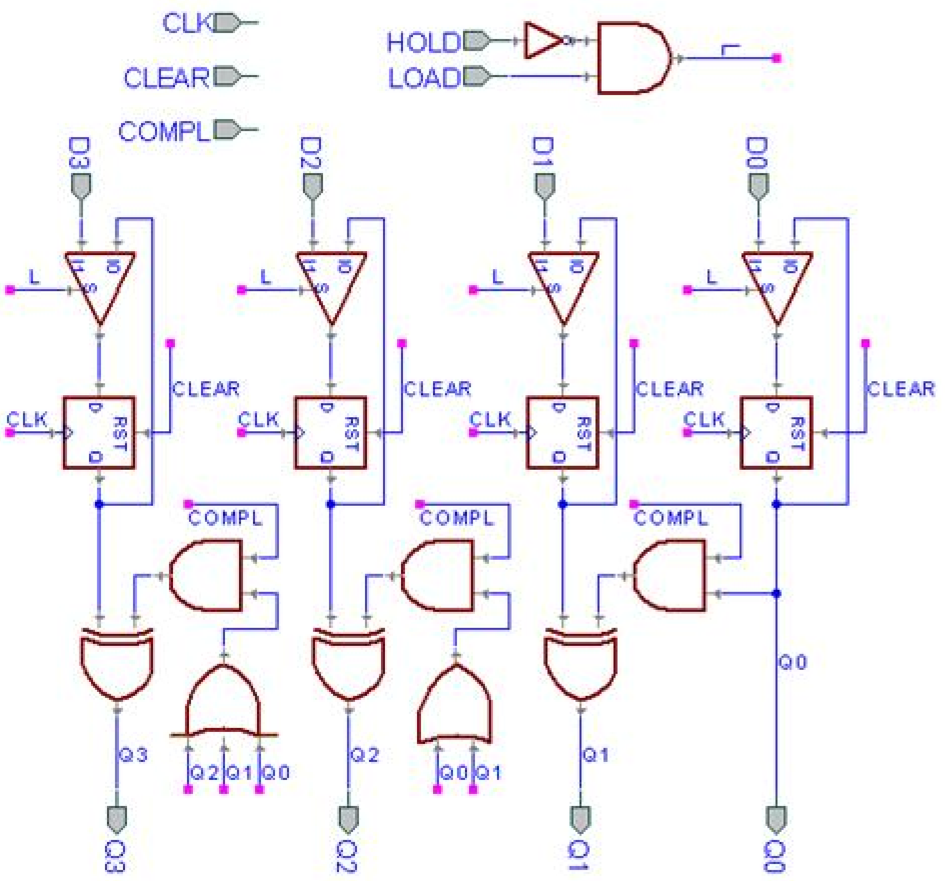
<u>Exercise 6.27</u>

Using 5 universal shift registers, the following implementation implements the 4-bit LIFO stack. The stack is determined to be full if the left-most shift register in the diagram has RightOut asserted. This is to say that whenever something is pushed onto the stack, a logical 1 will move over one space to the right in the shift register. Whenever a pop occurs, a logic 0 is shifted onto the left of the register.

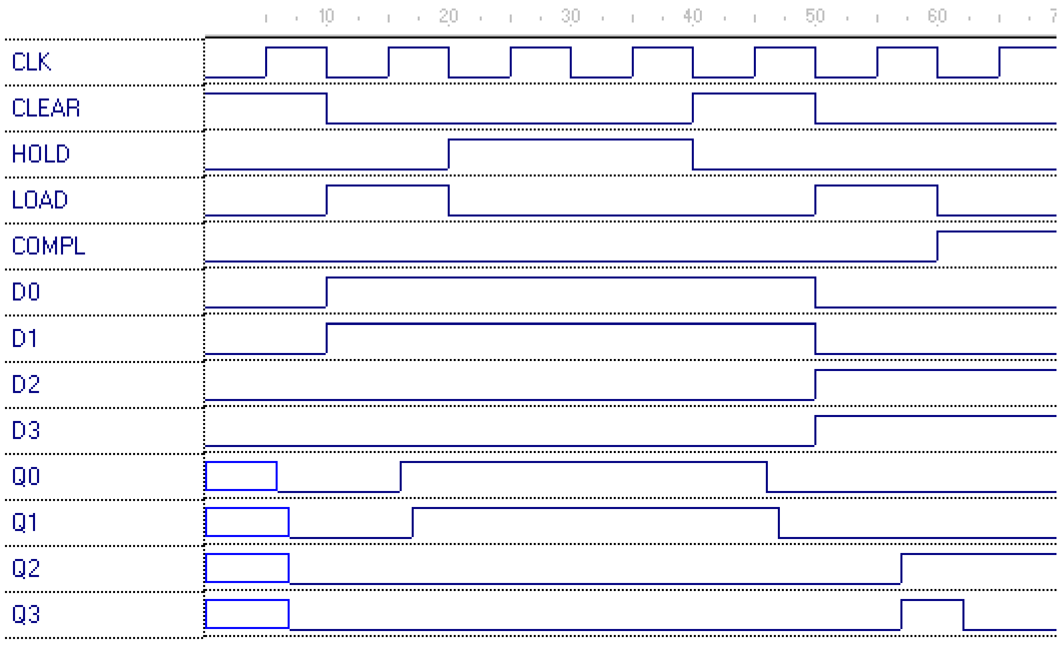


Exercise 6.28

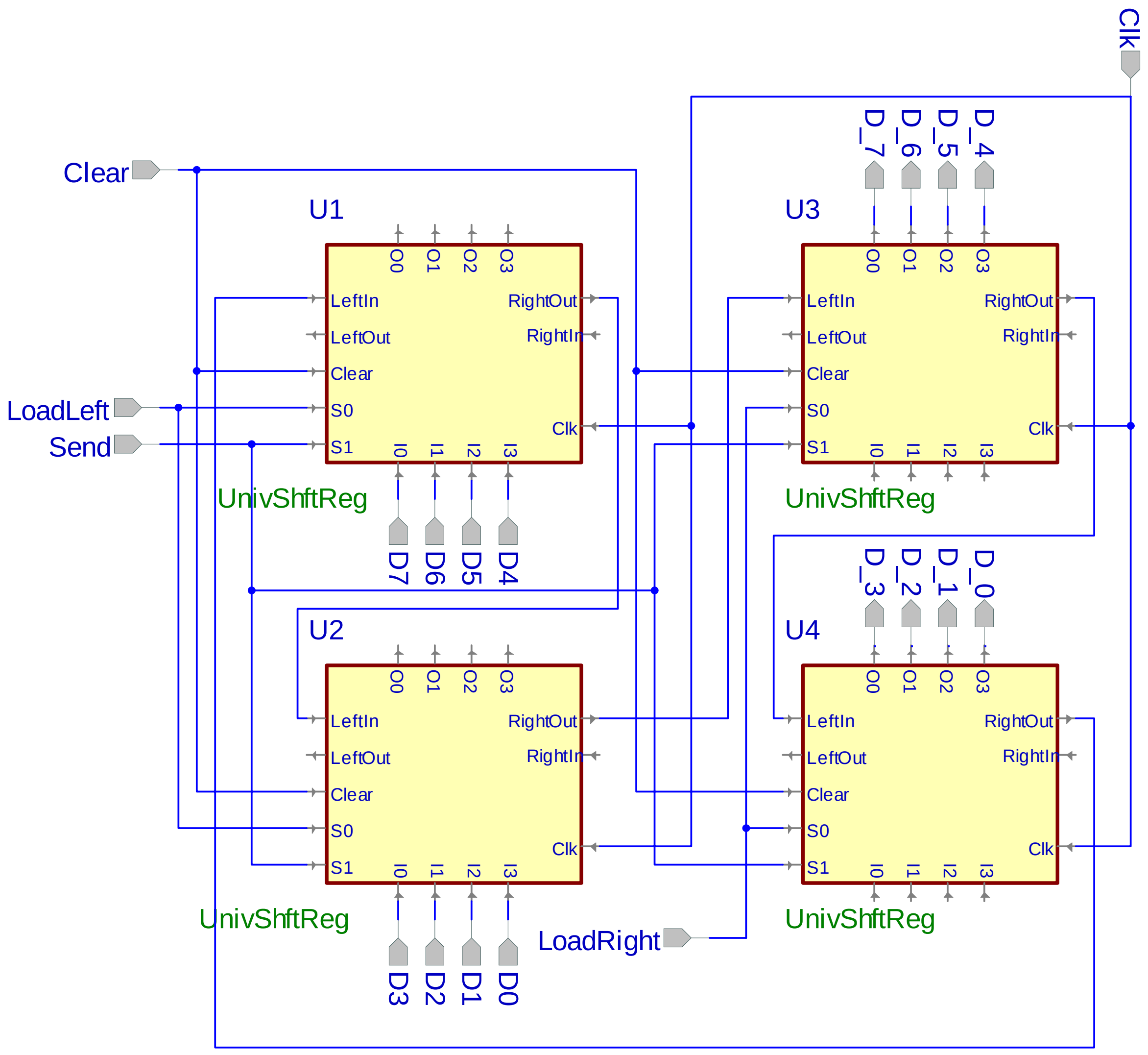
One of many possible implementations:



Simulation waveforms for the specified sequence of operations:



Exercise 6.29



Exercise 6.30

A master-slave flip-flop looks like the following when expressed in Verilog. Note the two `always` blocks, triggered by different edges of the clock. This description is reset-dominant.

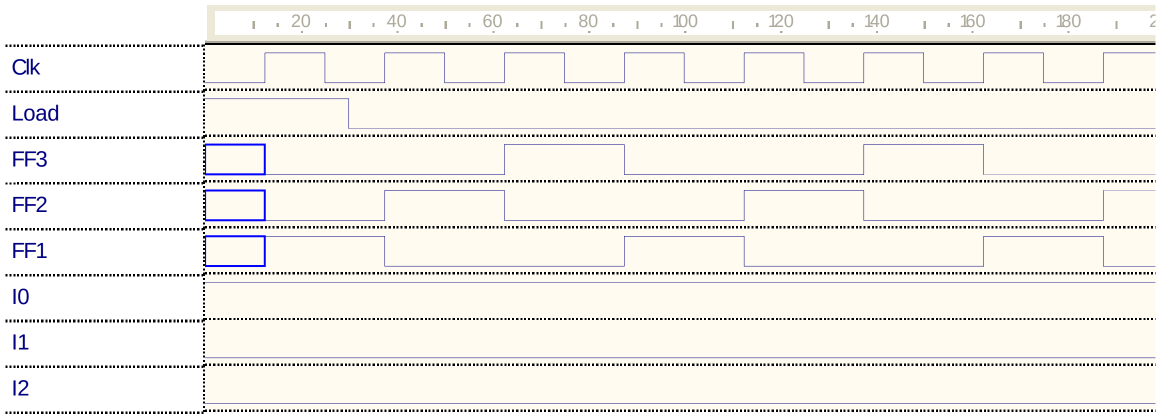
```
module prob6_30 (P, Q, Clk, R, S);

input R, S, Clk;
output P, Q;
reg P, Q;

always @(posedge Clk) begin
      if ( R ) P = 1'b0;
      else if ( S ) P = 1'b1;
end

always @(negedge Clk) begin
      if ( P ) Q = 1'b1;
      else Q = 1'b0;
end

endmodule
```

Exercise 6.31

The implementation below uses a Load signal to initialize FF_1 with I0, FF_2 with I1, and FF_3 with I2. If the load signal is not present, then the module performs a circular shift behavior.

```
module prob6_31 (Load, I0, I1, I2, Clk, FF1, FF2, FF3);

input Load, I0, I1, I2, Clk;
output FF1, FF2, FF3;
reg FF1, FF2, FF3;

always @(posedge Clk) begin
      if (Load) begin
            FF1 = I0;
            FF2 = I1;
            FF3 = I2;
      end
      else begin
            FF3 <= FF2;
            FF2 <= FF1;
            FF1 <= FF3;
      end
end

endmodule
```

The waveform on the next page demonstrates the functionality of this module.



Exercise 6.32

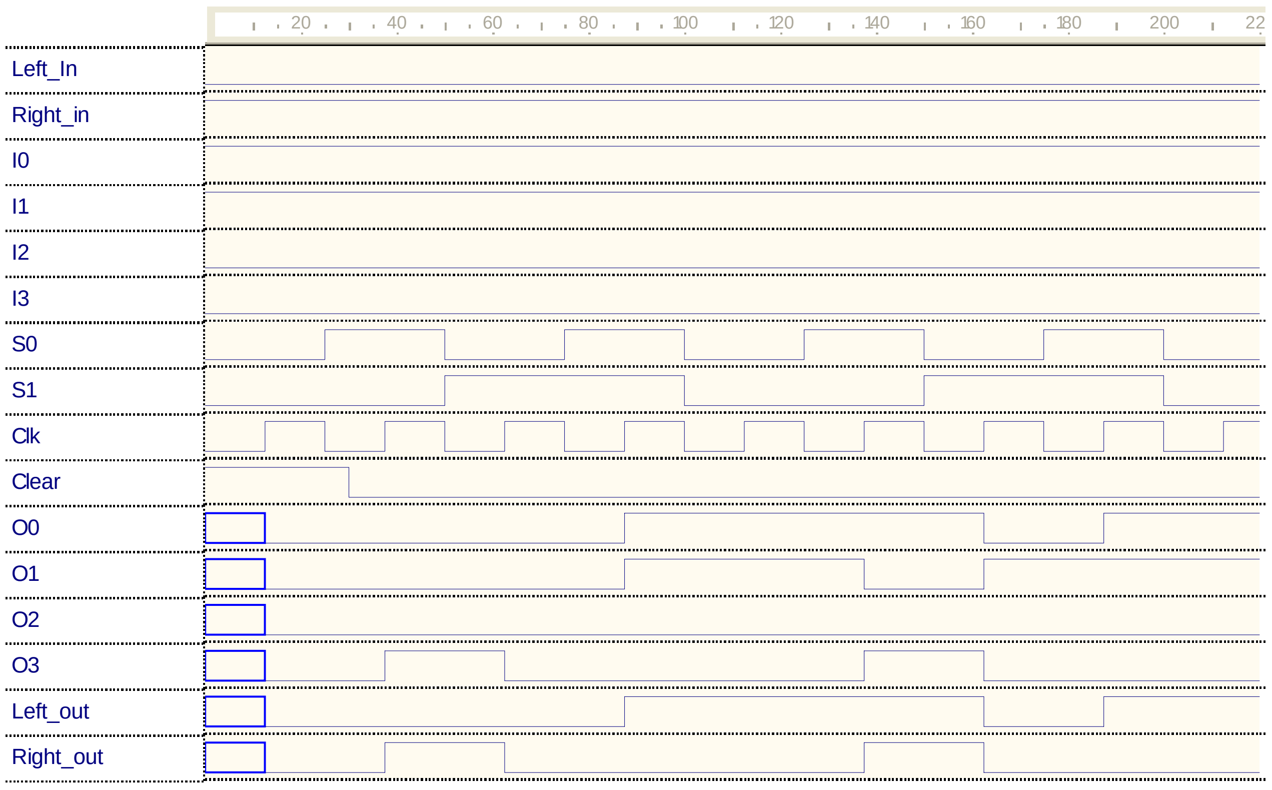
Here is the implementation for the 1 bit module that is shown in Figure 6.54.

```
module prob6_32_1bit (In, Left, Right, Clear, S0, S1, Clk, Out);

input In, Left, Right, Clear, S0, S1, Clk;
output Out;
reg Out;

always @(posedge Clk) begin
      if ( Clear ) Out = 1'b0;
      else begin
            if ( !S0 && !S1 ) Out = Out;
            if ( !S0 && S1 ) Out = Left;
            if ( S0 && !S1 ) Out = Right;
            if ( S0 && S1 ) Out = In;

      end
end

endmodule
```

The 4 bit implementation is shown on the next page using 4 instances of the `prob6_32_1bit` module and appropriate wiring done by name:

```
module prob6_32_4bit (I0, I1, I2, I3, Left_in, Right_in, Clear, S0, S1,
Clk, O0, O1, O2, O3, Left_out, Right_out);

input I0, I1, I2, I3, Left_in, Right_in, Clear, S0, S1, Clk;
output O0, O1, O2, O3, Left_out, Right_out;

prob6_32_1bit FirstBit( .In(I0), .Left(Left_In), .Right(O1), .Clk(Clk),
.Clear(Clear), .S0(S0), .S1(S1), .Out(O0));
prob6_32_1bit SecondBit( .In(I1), .Left(O0), .Right(O2), .Clk(Clk),
.Clear(Clear), .S0(S0), .S1(S1), .Out(O1));
prob6_32_1bit ThirdBit( .In(I2), .Left(O1), .Right(O3), .Clk(Clk),
.Clear(Clear), .S0(S0), .S1(S1), .Out(O2));
prob6_32_1bit FourthBit( .In(I3), .Left(O2), .Right(Right_in),
.Clk(Clk), .Clear(Clear), .S0(S0), .S1(S1), .Out(O3));

assign Right_out = O3;
assign Left_out = O0;

endmodule
```

Here is a timing waveform showing that the operations work correctly:


Left_In
Right_in
I0
I1
I2
I3
S0
S1
Clk
Clear
O0
O1
O2
O3
Left_out
Right_out