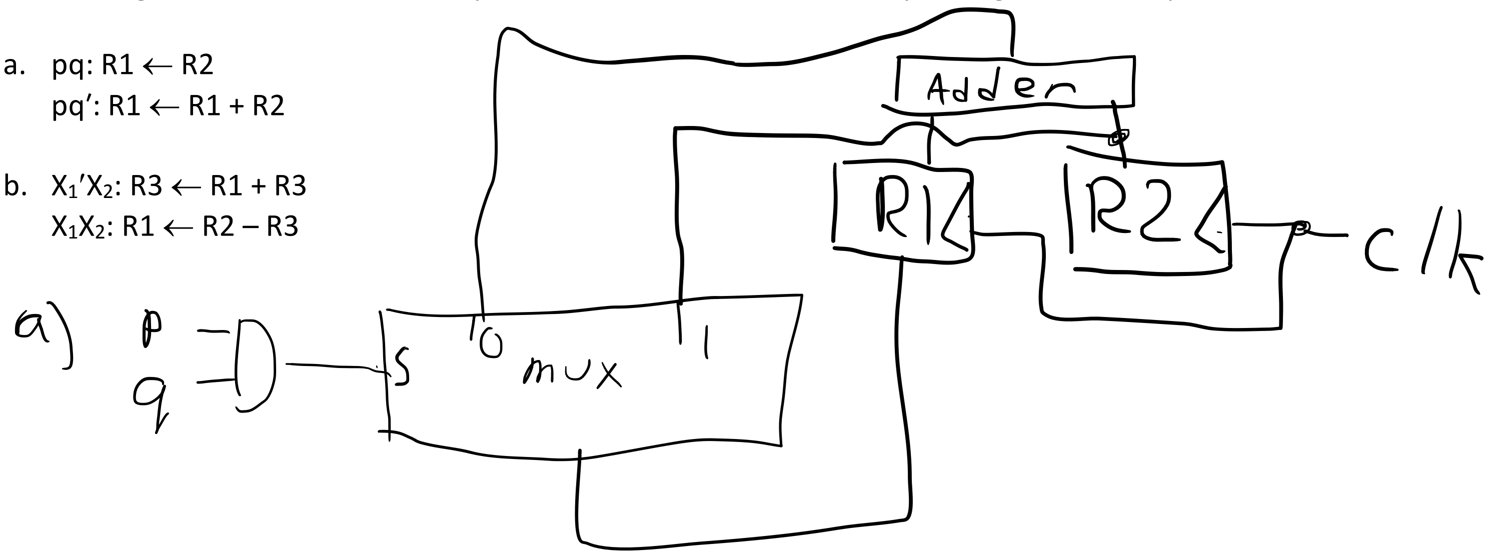
ECE 2504: Introduction to Computer Engineering
Homework Assignment 7 (70 points)

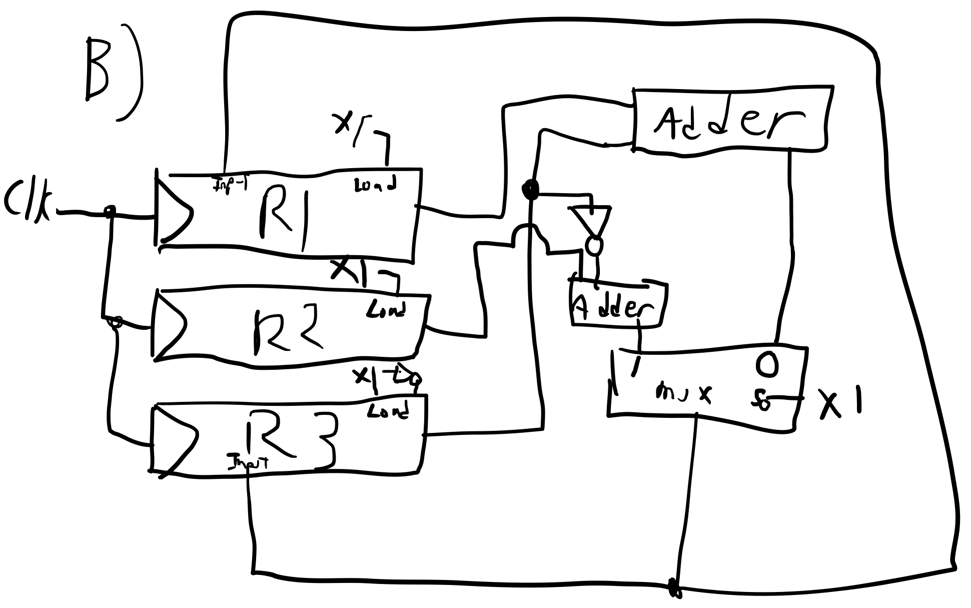
Please show your work, and clearly indicate your answers.

Problem 1 (8 points)
Draw hardware block diagrams that implement each of the following register-transfer language statements. Where appropriate, you may use hardware elements such as logic gates, multiplexers, decoders, encoders, adders, registers, and counters, but you should use seek to minimize your logic wherever possible.

a. pq: R1 $\leftarrow$ R2
   pq': R1 $\leftarrow$ R1 + R2

b. $X_1'X_2$: R3 $\leftarrow$ R1 + R3
   X_1X_2: R1 $\leftarrow$ R2 – R3





Explanation for b: Since X1 is the only value that differs between them, it is used as the n-bit select line of the 2^n Mux that is thus used to decide on whether to output the value of the first adder or that of the second adder. Where, the output of the mux goes both into R3 and R1 because we are assigning inputs of the load to each register. In where R3 are only able to load a new value if X1' = 1 and where R1 only if X1 is 1. This makes it so that in X1'X2, that the output of the MUX will not be able to enter into R1, but only into R3. In X1X2, where R2-R3 is mearley just R2+ R3(compliment), it can be determined that the values should only load into R1 when X1 = 1 and where we can make sure it wont load into R3 because X1' is thus 0.

Problem 2 (8 points)

Consider the following conditional control statement:

if (X_1 = 1) then (R1 ← R2 + R3), *else if* (X_2 = 1) then (R1 ← R2)

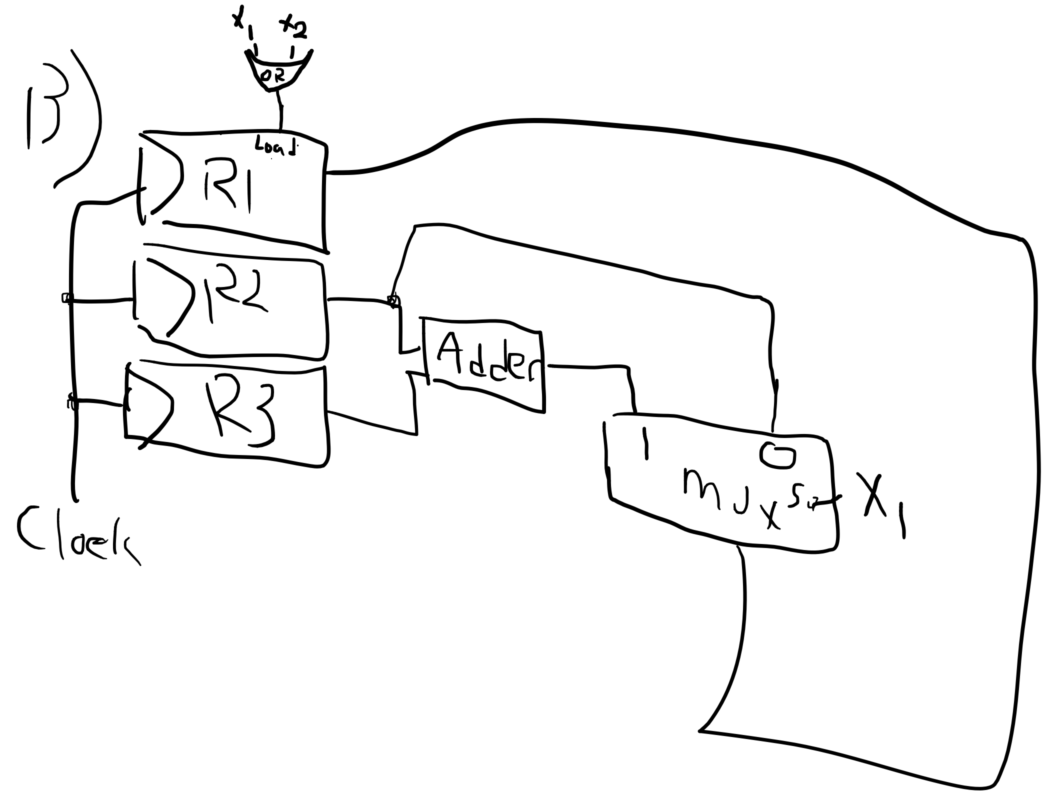
a. Formulate a pair of register-transfer language statements that describe the above conditional statement. Make sure you correctly incorporate the idea of an "if-else if" into the control conditions of your RTL statements. (Ask yourself what the specific logic condition must be for the "else if" to occur.)

b. Show a hardware block diagram that implements the register-transfer language statement of part (a). Follow the instructions of Problem 1 in deciding what elements to use.

a) $X_1 X_2'$: $R1 \leftarrow R2 + R3$

$X1' X2$: $R1 \leftarrow R2$

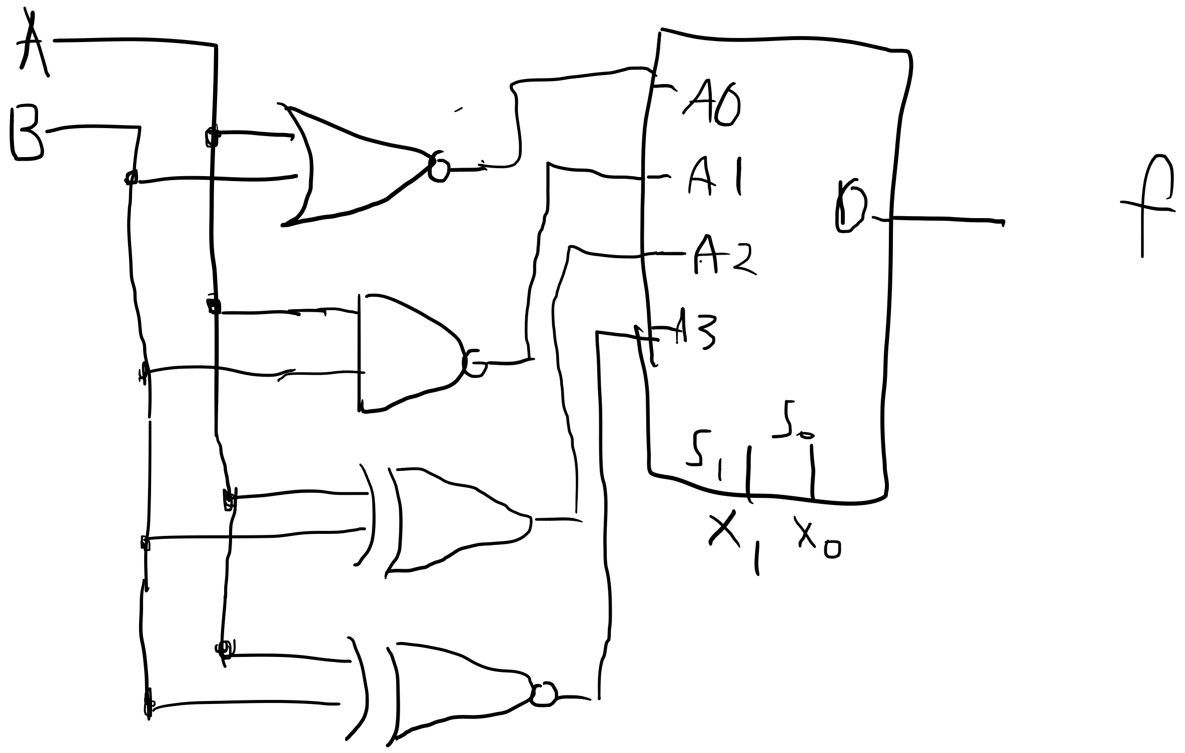
B)



Problem 3 (4 points)
Consider the following function table for a logic unit. Take A and B to be n-bit operands.

X_1	X_0	**Result?**
0	0	A NOR B
0	1	A NAND B
1	0	A XOR B
1	1	A XNOR B

Design a logic unit that performs the functions given in the table. Follow the instructions of Problem 1 in deciding what elements to use. Since logical operations are bitwise, your circuit need only represent a one-bit "slice" or the complete logic unit.



Problem 4 (6 points)
A particular processor can perform the AND, OR, and XOR operators on two 8-bit operands. The first operand is contained in a register, while the second is a *literal*, or constant value. The processor then stores the result in a destination register. For example, the notation shown below ANDs the contents of register R1 (whatever they might be) with the binary value 00110011 and places the result into R2.

operation (*operand register*, *literal operand*, *destination register*)

AND (R1, 00110011, R2)

Using notation similar to that shown above, indicate how a user might carry out the following tasks:

a. Register R1 contains a value. We wish to place the value into register R2 so that we may operate upon it, but the only relevant bits are in bit positions 7, 6, 5, and 4 of R1. Thus, we want to *clear* the other bits and place the result into R2.
b. We need to move the value in register R3 to register R4 so that we can use the value to enable a peripheral. To correctly enable the device, we must set the bits in positions 6, 5, 2, and 1; the bits in the remaining positions must remain unchanged.
c. With the value already in R4, we most now change the status of certain aspects of the peripheral. Thus, we must toggle the values in bit positions 7, 4, 1, and 0; the bits in the remaining positions must remain unchanged. The value should be returned to R4.

In each case, you don't know what value is in the register operand acting as the source. You only know that it needs to be changed in the way described in each part of this problem.

$R1 = \underbrace{\underset{7}{0}\,\underset{6}{0}\,\underset{5}{1}\,\underset{4}{1}}_{\text{keep}}\,\underset{3}{0}\,\underset{2}{0}\,\underset{1}{1}\,\underset{0}{1}$ so keep

a) AND (R1, 11110000, R2)

B) OR (R3, 01100110, R4)

C) XOR(R4, 1001 0011, R4)

Problem 5 (12 points)
The 8-bit registers R1, R2, R3, and R4 have the following starting values:

R1 = 00000000 R2 = 11000101 R3 = 01101000 R4 = 11000011

A processor is capable of carrying out the following operations. Each micro-operation takes one or more operands. The operands represent register addresses, and the micro-operations are carried out on the contents of the registers as described below.

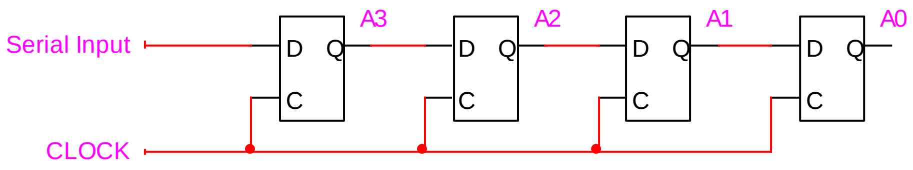
Mnemonic	RTL	Description
BMSK(A,B)	A ← A AND B	Perform a selective-mask on the contents of register A. Use the contents of register B as the mask. Store the result in register A.
BSET(A,B)	A ← A OR B	Perform a selective-set on the contents of register A. Use the contents of register B as the mask. Store the result in register A.
BCMP(A,B)	A ← A XOR B	Perform a selective-complement on the contents of register A. Use the contents of register B as the mask. Store the result in register A.
ASL(A)	A ← asl(A)	Perform an arithmetic-shift left on the contents of register A. Store the result in register A.
ASR(A)	A ← asr(A)	Perform an arithmetic-shift right on the contents of register A. Store the result in register A.
CIL(A)	A ← cil(A)	Perform a circular-shift left on the contents of register A. Store the result in register A.
CIR(A)	A ← cir(A)	Perform a circular-shift right on the contents of register A. Store the result in register A.
LSL(A)	A ← lsl(A)	Perform a logical-shift left on the contents of register A. Store the result in register A.
LSR(A)	A ← lsr(A)	Perform a logical-shift right on the contents of register A. Store the result in register A.

Suppose that the processor executes following sequence of micro-operations:

BSET(R1,R2)	R1 = 11000101
LSL(R1)	R1 = 10001010
BCMP(R1,R3)	R1 = 01000101
ASR(R1)	R1 = 00100010
BMSK(R1,R4)	R1 = 00000010
CIR(R1)	R1 = 00000001

Show the contents of R1 after each step of the program. Remember that since R1 receives the result of each operation, that the R1 you find as the result of each step is the R1 that the processor uses in the next step.

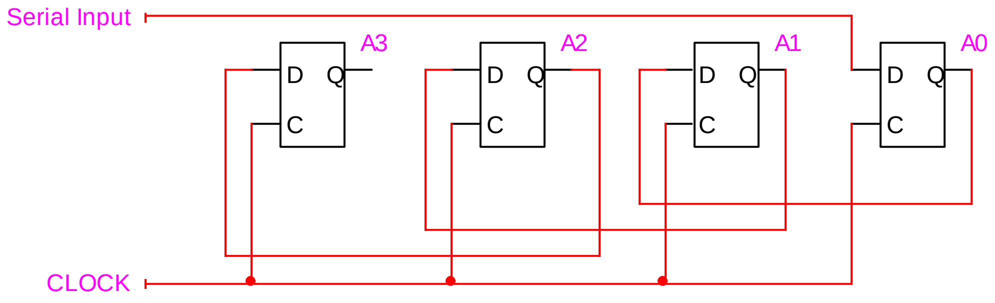
Problem 6 (4 points)
The circuit shown below operates as a 4-bit shift register. Suppose that it contains the initial value 1001. The bits are shown in order of significance, with higher-order bits on the left.



The register's contents are shifted four times to the right. The values applied to the serial input on each of the four shifts are (in the order that they are shifted into the register from first to last) 0 – 1 – 1 –0. What are the contents of the shift register following each of the fourshifts?

n	shift
0	1001
1	0100
2	1010
3	1101
4	0110

Problem 7 (4 points)
A **ring counter** is a shift register that connects the output of the bit that is normally lost to the register's serial input. The circuit below implements a *normal* shift-left operation.



Suppose that we used this shift register to implement a ring counter. (The shift register does not implement a ring counter by itself, but I just described how to use it to implement one.) If the initial value of this ring counter is 0010, list the sequence of states attained by the ring counter in each of four successive clock cycles.

n	
0	0010
1	0001
2	1000
3	0100
4	0010

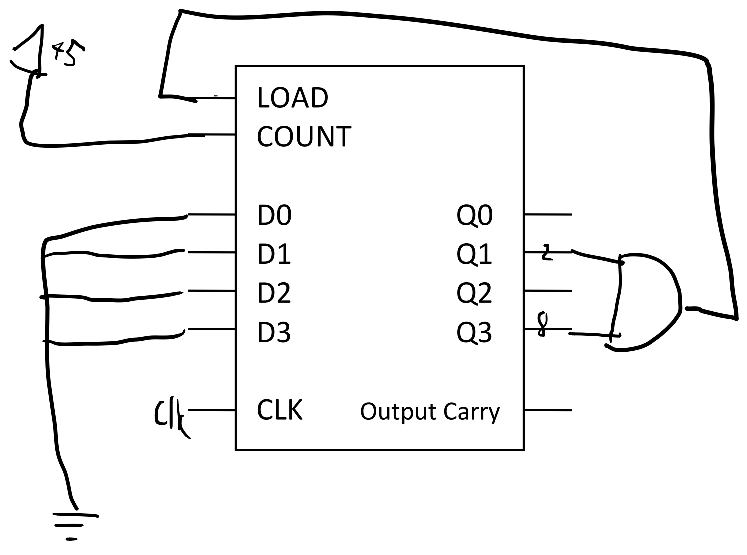
<u>Problem 8</u> (4 points)
Slide 27 of the Section 6 Notes features the internal circuitry of a binary counter with parallel load. Below, you will find a block diagram for this counter.

Whenever LOAD = 1, the counter state (Q3 Q2 Q1 Q0) takes on the value supplied by the parallel load inputs (D3 D2 D1 D0) on a clock trigger. When COUNT = 1 and LOAD = 0, the counter state increments by 1 on a clock trigger.

Notice that the LOAD input takes precedence over the COUNT input; when LOAD = 1, the counter will perform a load on the next clock trigger regardless of what value COUNT has. COUNT only performs its function when LOAD = 0 ***and*** *COUNT = 1.*

A **divide-by**-*n* counter is one that cycles through the binary states that correspond to the numbers from 0 to (*n* – 1). The binary counter shown above operates naturally as a divide-by-16 counter, since it cycles through the binary states that correspond to the numbers from 0 to 15.

With an additional AND gate, a designer could use the same counter to implement a divide-by-*n* counter for any $n \leq 16$. Using the block diagram shown above, **show the implementation for a divide-by-10 counter**.



Problem 10 (12 points)
Complete the table below. Remember that each memory characteristic is specified by the total number of addresses in the module times the number of **bits** in each word. For the capacities, use an appropriate "metric" prefix such as KB, MB, or GB along with a power of 2 expressed as a decimal.

Characteristic	Address Bits	Data Bits	Capacity in Bytes
32K × 8	2^{15} or 32768	8	262144
128K × 16	2^{17} or 131072	16	2097152
1M × 32	2^{20} or 1048576	32	33554432
2G × 64	2^{31} or 2147483648	64	137438953472

$32 \times 1024 (2^{10}) = 2^{15}$ or $32768 \times 8 = \uparrow$

$128 \times 1024 (2^{10}) = 2^{17}$ or $131072 \times 16 = \uparrow$

$1 \times 1048576 (2^{20}) = 2^{20}$ or $1048576 \times 32 = \uparrow$

$2 \times 1073741824 (2^{30}) = 2^{31}$ or $2147483648 \times 64 = \uparrow$

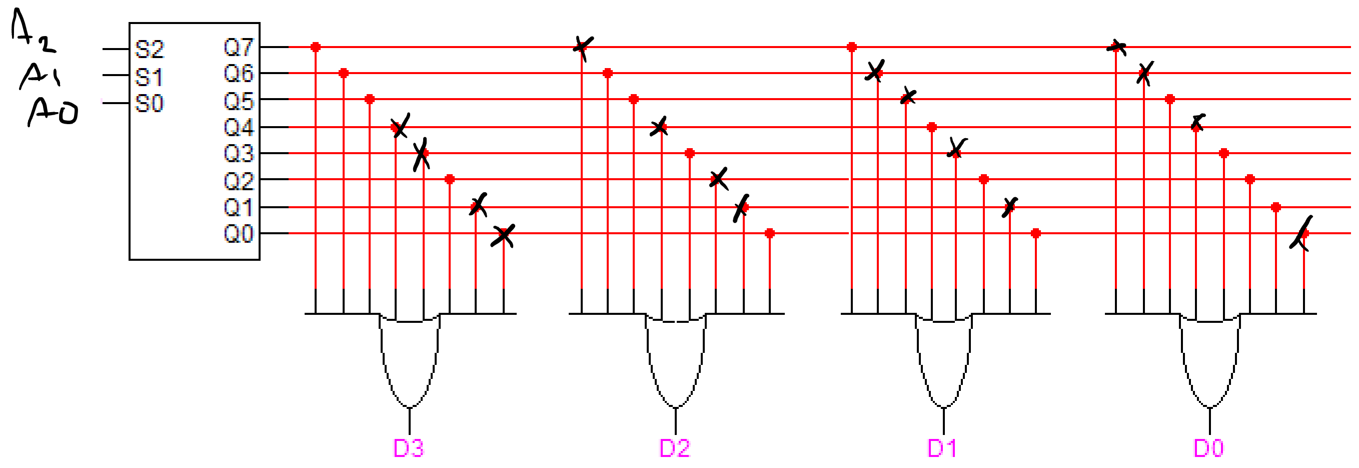
Problem 11 (8 points)

Show a block diagram for an 8 × 4 PROM that has the following values encoded into its addresses:

	Address			Data Contents			
	A_2	A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	1	1	0
1	0	0	1	0	0	0	1
2	0	1	0	1	0	1	1
3	0	1	1	0	1	0	1
4	1	0	0	0	0	1	0
5	1	0	1	1	1	0	1
6	1	1	0	1	1	0	0
7	1	1	1	1	0	1	0

Use A2, A1, and A0 as the address inputs. A2 is the most significant address input, and A0 is the least significant address input. *Make sure that you label the "decoder inputs" appropriately.*

Each 8-input OR gate represents a data output. Take D3, D2, D1, and D0 as the data outputs. D3 is the most significant data output, and D0 is the least significant data output. To represent the blown fuses, *cross out all of the nodes that the user has "disconnected" as part of the programming process*.


A2
A1
A0
S2
S1
S0
Q7
Q6
Q5
Q4
Q3
Q2
Q1
Q0
D3
D2
D1
D0