Question 1 — 8 pts

Consider the following C-like code to search a linked list

```
int* search_linked_list(int *node, int x) {
  while (node != NULL) {
    if (node->data == x)
       return node;
    node = node->next;
  }
  return -1;
}
```

You are given the following code:

```
      .data
LIST: .word 1, NEXT1
NEXT1: .word 2, NEXT2
NEXT2: .word 3, NEXT3
NEXT3: .word 6, -1
      .text
      .global _start
_start:
   LDR SP, =0x20000
   LDR R0, =LIST
   MOV R1, #6 // value searching for
   BL SEARCH_LIST
END: B END
```

Implement the SEARCH_LIST subroutine. If the search value is found, you should return the address of the node containing that value in R0; if the value is not found, -1 should be returned in R0. An example linked list is given in the .data section. Your code should work for linked lists of arbitrary lengths. Each node in the linked list consists of word sized piece of data and a pointer to the next node in the list. The last node in the list will have its next pointer be -1 (consider this as NULL).

```
SEARCH_LIST:
        PUSH {R4, LR}
CHECK_END:      CMP R0, #-1
                BEQ END_LIST
                LDR R4, [R0]
                CMP R4, R1
                BEQ RETURN // item found, return addr in R0
                LDR R0, [R0, #4]
                B CHECK_END
END_LIST:
        MOV R0, #-1
        B RETURN
RETURN:
        POP {R4, PC}
```

Question 2 A — 5 pts

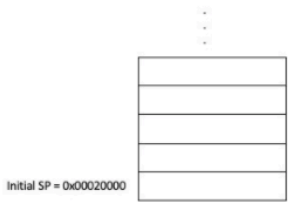
Consider the following code that performs a Binary Search (you are given both the C-like version and the assembly version):

```
int BinarySearch(int arr[], int L, int R, int x) {
  if (R >= L) {
    int mid = L + (R-L)/2;
    if (arr[mid] == x) return mid;
    if (arr[mid] > x) return BinarySearch(arr, L, mid-1, x);
    return BinarySearch(arr, mid+1, R, x);
  }
  return -1; // element not present in array
}
```

```
     .text
     .global _start
_start:
    LDR SP, =0x20000
    LDR R0, =ARR
    MOV R1, #0 // Left
    LDR R4, =SIZE
    LDR R4, [R4]
    SUB R2, R4, #1 // Right
    MOV R3, #2 // value searching for
    MOV R5, #0xAA
    MOV R6, #0xBB
    MOV R7, #0xCC
CALL1: BL BinarySearch
    END: B END
BinarySearch:
    PUSH {R5-R7, LR}
    CMP R2, R1
    BGE FIND_MID
    MOV R0, #-1
    B RETURN
FIND_MID:
    SUB R5, R2, R1
    LSR R5, R5, #1
    ADD R5, R5, R1
    LSL R6, R5, #2
    ADD R6, R6, R0
    LDR R7, [R6]
    CMP R7, R3
    BGT SEARCH_LEFT
    BLT SEARCH_RIGHT
HERE:   MOV R0, R5
    B RETURN
SEARCH_LEFT:
    SUB R2, R5, #1
CALL2:   BL BinarySearch
    B RETURN
SEARCH_RIGHT:
    ADD R1, R5, #1
CALL3:   BL BinarySearch
    B RETURN
RETURN:
    POP {R5-R7, PC}

     .data
ARR: .word 1, 2, 3, 4, 7, 9, 10, 12, 14
SIZE: .word 9
```

Using the provided data section (above), show the contents of the stack when you reach the instruction labelled: "HERE". Assume the address of the instruction labelled CALL1 is 0x00000028, the address of instruction labelled CALL2 is 0x00000074, and CALL3 is 0x00000080. Labels have been bolded for your convenience.

Please format your answer similar to this picture. Be sure to clearly indicate what the value of SP at instruction labelled "HERE" is.



Answer

```
SP 0x0001FFE0 --> 0x00000004
                  0x000000A8
                  0x00000007
                  0x00000078
                  0x000000AA
                  0x000000BB
                  0x000000CC
                  0x0000002C
0x00020000 --> Unknown
```

Question 2 B

5 pts

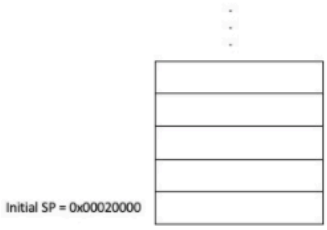
Consider the following code that performs a Binary Search (you are given both the C-like version and the assembly version):

```
int BinarySearch(int arr[], int L, int R, int x) {
  if (R >= L) {
    int mid = L + (R-L)/2;
    if (arr[mid] == x) return mid;
    if (arr[mid] > x) return BinarySearch(arr, L, mid-1, x);
    return BinarySearch(arr, mid+1, R, x);
  }
  return -1; // element not present in array
}
```

```
    .text
    .global _start
_start:
    LDR SP, =0x20000
    LDR R0, =ARR
    MOV R1, #0 // Left
    LDR R4, =SIZE
    LDR R4, [R4]
    SUB R2, R4, #1 // Right
    MOV R3, #10 // value searching for
    MOV R5, #0xDD
    MOV R6, #0xEE
    MOV R7, #0xFF
CALL1: BL BinarySearch
END: B END
BinarySearch:
    PUSH {R5-R7, LR}
    CMP R2, R1
    BGE FIND_MID
    MOV R0, #-1
    B RETURN
FIND_MID:
    SUB R5, R2, R1
    LSR R5, R5, #1
    ADD R5, R5, R1
    LSL R6, R5, #2
    ADD R6, R6, R0
    LDR R7, [R6]
    CMP R7, R3
    BGT SEARCH_LEFT
    BLT SEARCH_RIGHT
HERE:   MOV R0, R5
    B RETURN
SEARCH_LEFT:
    SUB R2, R5, #1
CALL2:   BL BinarySearch
    B RETURN
SEARCH_RIGHT:
    ADD R1, R5, #1
CALL3:   BL BinarySearch
    B RETURN
RETURN:
   POP {R5-R7, PC}

    .data
ARR: .word 1, 2, 3, 4, 7, 9, 10, 12, 14
SIZE: .word 9
```

Using the provided data section (above), show the contents of the stack when you reach the instruction labelled: "HERE". Assume the address of the instruction labelled CALL1 is 0x00000028, the address of instruction labelled CALL2 is 0x00000074, and CALL3 is 0x00000080. Labels have been bolded for your convenience.

Please format your answer similar to this picture. Be sure to clearly indicate what the value of SP at instruction labelled "HERE" is.



Answer

```
SP 0x0001FFE0 --> 0x00000004
                  0x000000A8
                  0x00000007
                  0x00000084
                  0x000000DD
                  0x000000EE
                  0x000000FF
                  0x0000002C
0x00020000 --> Unknown
```

Question 3 — 10 pts

A) Draw a circuit that has 4 D Flip-Flops with outputs Q3, ... Q0 and inputs Load, Rotate, N, D3, ... D0, clock and resetn that functions as a circular shift register (performs a rotate operation). Your circuit should have the following functionality:

- If Load = 1, all bits of the register are loaded in parallel with a value D3-D0.
- If Load = 0 and Rotate = 0, the register is rotated right by a specific number of bits (in one clock cycle). If N = 0, the register is rotated 1 bit. If N = 1, the register is rotated 2 bits.
- If Load = 0 and Rotate = 1, the register is rotated left by a specific number of bits (in one clock cycle). If N = 0, the register is rotated 1 bit. If N = 1, the register is rotated 2 bits.

You can use flip flops, multiplexers and any gates you wish to implement your circuit. The register should also have an asynchronous active low reset.

B) Write the Verilog to implement your circuit design. Implement two modules: one that implements a single bit in the register and a top-level module (RotateReg) that implements the 4 bit register. Your answer must be written using hierarchical Verilog.

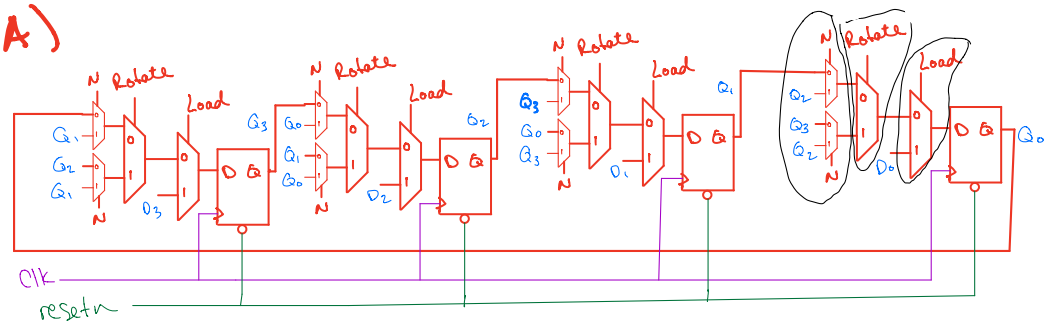
You are provided with the following D Flip-Flop module so you do not need to write it yourself:

```
module D_Flip_Flop (data, clk, resetn, q);
  input data, clk, resetn;
  output reg q;

  always @ (posedge clk) begin
    if (resetn == 0)
      q <= 1'b0;
    else
      q <= data;
  end
endmodule
```

Your top-level module should begin as follows:

```
module RotateReg(Load, Rotate, N, D, clock, resetn, Q);
  input Load, Rotate, clock, resetn, N;
  input [3:0] D;
  output [3:0] Q;
```

A)



B)

```
module RotateReg(Load, Rotate, N, D, clock, resetn, Q);
   input Load, Rotate, clock, resetn, N;
   input [3:0] D;
   output [3:0] Q;

   RotateBit bit3(Load, Rotate, N, D[3], clock, resetn, Q[3], {Q[0], Q[1]}, {Q[2], Q[1]});
   RotateBit bit2(Load, Rotate, N, D[2], clock, resetn, Q[2], {Q[3], Q[0]}, {Q[1], Q[0]});
   RotateBit bit1(Load, Rotate, N, D[1], clock, resetn, Q[1], {Q[2], Q[3]}, {Q[0], Q[3]});
   RotateBit bit0(Load, Rotate, N, D[0], clock, resetn, Q[0], {Q[1], Q[2]}, {Q[3], Q[2]});

endmodule

module RotateBit(Load, Rotate, N, D, clock, resetn, Q, R, L);

   input Load, Rotate, clock, resetn, N, D;
   input [1:0] R, L;
   output Q;

   wire  w1, w2, w3, w4;

   D_Flip_Flop dff(w4, clock, resetn, Q);

   always @ (*) begin
     if (N==1'b0) begin
        w1 = R[0];
        w2 = L[0];
     end
     else
       w1 = R[1];
       w2 = L[1];
     end
   end

   assign w3 = Rotate ? w1 : w2;
   assign w4 = Load ? w3 : D;

endmodule
```

Question 4 A

10pts

You have purchased a new ARM-based system that includes an analog to digital converter (ADC) I/O device. The ADC takes multiple analog inputs (called channels) and converts the analog input to a 12-bit digital value that can be read by the processor. The memory mapped interface for the ADC is as follows:

	Bits 31-6	5	4	3	2-0
Address 0xEE201000	unused	I	D	Go	Channel select

	Bits 31-16	15-4	3-0
Address 0xEE201004	unused	Converted Data	unused

- The "D" bit is set to 1 by the ADC when it is Done converting the signal.
- The programmer writes to the "Channel select" bits to select one of 8 channels to convert.
- The programmer writes a 1 to the "Go" bit to start the conversion.
- The digital value for the selected channel is available in the "Converted Data" bits when the "D" bit is set to one. The programmer should write a 1 to the D bit to clear the D bit.
- The "I" bit is used to enable interrupts. Setting I to 0 will disable interrupts.

Using **Polled I/O**, write a program that loops through the 8 channels starting with channel 0 and reads the digital values. Each converted value should be stored in an array of halfwords in memory indexed by the channel ID. Once all 8 channels have been read, the program should start converting again at channel 0 and will overwrite the old channel 0 data in the CONVERTED_DATA array. The array is declared and initialized to zero as follows:

```
    .data
CONVERTED_DATA: .hword 0, 0, 0, 0, 0, 0, 0, 0
```

Answer

```
 .global _start
_start:
        LDR R0, =0xEE201000
        LDR R1, =CONVERTED_DATA
        MOV R2, #0 // channel number

LOOP: MOV R3, R2
      ORR R3, R3, #8 // to set Go bit
          STR R3, [R0]

POLL: LDR R4, [R0]
      AND R4, R4, #16
          CMP R4, #0 // check done bit
          BEQ POLL

          STR R4, [R0] // clear done bit
          LDR R5, [R0, #4] //Load converted data
          LSR R5, R5, #4
          LSL R6, R2, #1 // offset to store halfword
          ADD R6, R6, R1 // add offset to base addr
          STRH R5, [R6] // store data
          ADD R2, R2, #1 // increment channel
          CMP R2, #8
          BLT LOOP
          MOV R2, #0
          B LOOP
```

Question 4 B

You have purchased a new ARM-based system that includes an analog to digital converter (ADC) I/O device. The ADC takes multiple analog inputs (called channels) and converts the analog input to a 12-bit digital value that can be read by the processor. The memory mapped interface for the ADC is as follows:

Address 0xCE201000

Bits 31-6	5-3	2	1	0
unused	Channel select	I	D	Go

Address 0xCE201004

Bits 31-20	19-0
Converted Data	unused

- The "D" bit is set to 1 by the ADC when it is Done converting the signal.
- The programmer writes to the "Channel select" bits to select one of 8 channels to convert.
- The programmer writes a 1 to the "Go" bit to start the conversion.
- The digital value for the selected channel is available in the "Converted Data" bits when the "D" bit is set to one. The programmer should write a 1 to the D bit to clear the D bit.
- The "I" bit is used to enable interrupts. Setting I to 0 will disable interrupts.

Using **Polled I/O**, write a program that loops through the 8 channels starting with channel 0 and reads the digital values. Each converted value should be stored in an array of halfwords in memory indexed by the channel ID. Once all 8 channels have been read, the program should start converting again at channel 0 and will overwrite the old channel 0 data in the CONVERTED_DATA array. The array is declared and initialized to zero as follows:

```
    .data
CONVERTED_DATA: .hword 0, 0, 0, 0, 0, 0, 0, 0
```

Answer

```
.global _start
_start:
        LDR R0, =0xCE201000
        LDR R1, =CONVERTED_DATA
        MOV R2, #0 // channel number

LOOP: MOV R3, R2
          LSL R3, R3, #3
          ORR R3, R3, #1 // set go bit
          STR R3, [R0]

POLL: LDR R4, [R0]
         AND R4, R4, #2
         CMP R4, #0 // check done bit
        BEQ POLL

        STR R4, [R0] // clear done bit
        LDRH R5, [R0, #6] // Load converted data
        LSR R5, R5, #4
        LSL R6, R2, #1 // offset to store halfword
        ADD R6, R6, R1 // add offset to base addr
        STRH R5, [R6] // store data
        ADD R2, R2, #1 // increment channel
        CMP R2, #8
        BLT LOOP
        MOV R2, #0
        B LOOP
```

Q5A

You are to write an ARM assembly language program that computes the length of the longest sequence of even integers in an array. For example, in the array: 2,1,4,6,8,7, the longest sequence of even integers is 3, owing to the sequence 4,6,8. Your program should put the result in register R0. The array is at a memory address with label LIST; the number of array elements is at a memory address with label N. You may assume all integers are positive and that N is greater than or equal to 1. A suggestion: write out C or pseudo code for the algorithm first. Starter code is as follows:

```
.data
LIST:
  .word 2,1,4,6,8,7
N:
  .word 6

.text
.global _start
_start:
```

Q5

```
int LIST[] = {2,1,4,6,8,7};
int N = 6;

int findLongestEven(int *list, int n) {

  int longest = 0; // R0
  int current = 0; // R2
  int *item = list; // R1

  int i;  R3
  for (i = n; i > 0; i--) {
        R4
    int val = *item;

    val = val & 0x1;

    if (val == 1) { // odd number
      current = 0;
    }
    else { // even number
      current++;
      if (current > longest)
        longest = current;
    }
    item++;
  }
  return longest;
}
```

```
1  .data
2  LIST:
3    .word 2,1,4,6,8,7
4  N:
5    .word 6
6
7  .text
8  .global _start
9  _start:
10     MOV R0,#0 // longest initially 0
11     LDR R1, =LIST
12    MOV R2,#0 // current count of even sequence
13     LDR R3, =N
14     LDR R3, [R3] // R3 <= N
15 LOOP:
16     CMP R3, #0 // check if the for loop is done
17     BEQ END
18     LDR R4, [R1] // load a value from the LIST
19     AND R4, R4, #1 // AND the LSB of the value with 1
20     CMP R4, #1 // check if ODD
21     BEQ ODD // branch if ODD
22     ADD R2, R2, #1 // bump up the current count
23     CMP R2, R0 // see if current count is > MAX count
24     BLE NEXT
25     MOV R0, R2 // replace MAX count (we have a new MAX)
26     B NEXT
27 ODD:
28     MOV R2, #0 // reset current count
29 NEXT:
30     ADD R1, R1, #4 // increment array pointer
31     SUB R3, R3, #1 // decrement N
32     B LOOP
33 END:
34     B END
35
```

Q5B

You are to write an ARM assembly language program that computes the length of the longest sequence of strictly ascending integers in an array. For example, in the array: 2,1,4,6,8,7, the longest sequence of ascending integers is 4, owing to the sequence 1,4,6,8. Your program should put the result in register R0. The array is at a memory address with label LIST; the number of array elements is at a memory address with label N. You may assume that N is greater than or equal to 1. A suggestion: write out C or pseudo code for the algorithm first. Starter code is as follows:

```
.data
LIST:
  .word 2,1,4,6,8,7
N:
  .word 6

.text
.global _start
_start:
```

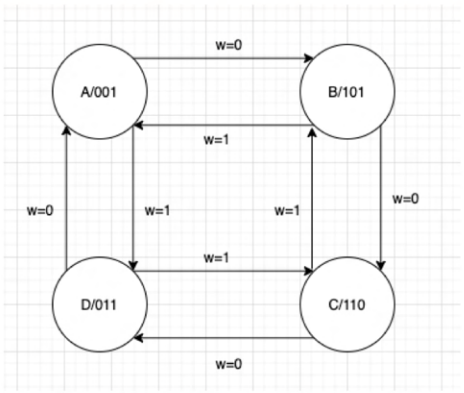
Q5

```
File Edit Options Buffers Tools C Help
#include <stdio.h>

int LIST[] = {2,1,4,6,8,7};
int N = 6;

int findLongestAscending(int *list, int n) {

  int longest = 0; // R0
  int current = 0; // R2
  int previous = -1; // R5
  int *item = list; // R1

  int i; R3
  for (i = n; i > 0; i--) {
    int val = *item; R4

    if (val <= previous) { // equal or decreasing
      current = 1;
    }
    else { // increasing
      current++;
      if (current > longest)
        longest = current;
    }
    previous = val;
    item++;
  }
  return longest;
}
```

```
1  .data
2  LIST:
3    .word 2,1,4,6,8,7
4  N:
5    .word 6
6
7  .text
8  .global _start
9  _start:
10     MOV R0,#0 // longest initially 0
11     LDR R1, =LIST
12     MOV R2,#0 // current count of ascending sequence length
13     MVN R5,#0 // set R5 to -1 initially... R5 will hold the previous list value
14     LDR R3, =N
15     LDR R3, [R3] // R3 <= N
16 LOOP:
17     CMP R3, #0 // check if the for loop is done
18     BEQ END
19     LDR R4, [R1] // load a value from the LIST
20     CMP R4, R5 // compare it with R5 (the previous list value)
21     BLE NONASCENDING
22     ADD R2, R2, #1 // bump up the current count
23     CMP R2, R0 // see if current count is > MAX count
24     BLE NEXT
25     MOV R0, R2 // replace MAX count (we have a new MAX)
26     B NEXT
27 NONASCENDING:
28     MOV R2, #1 // reset current count to 1 (as this is a new sequence)
29 NEXT:
30     MOV R5, R4
31     ADD R1, R1, #4 // increment array pointer
32     SUB R3, R3, #1 // decrement N
33     B LOOP
34 END:
35     B END
36
```

Q6A

Consider the following state diagram for an FSM with one input, w, and three outputs z_2, z_1, z_0.



a) Give the state table corresponding to the state diagram. Use A,B,C,D to represent the states in your state table.

b) Using the state encoding below, give the logic functions for the next-state logic, as well as the outputs z_2, z_1, z_0. Following the notation in the course text book, use Y_1, Y_0 to represent the next-state logic functions. Use K-maps to minimize the functions in SOP form.

State	Code y_1y_0
A	00
B	01
C	11
D	10

Q6)

a)

Curr	next w=0	next w=1	z_2 z_1 z_0
A	B	D	0 0 1
B	C	A	1 0 1
C	D	B	1 1 0
D	A	C	0 1 1

b)

Curr y_1y_0	next w=0 Y_1Y_0	next w=1 Y_1Y_0	z_2 z_1 z_0
A 00	01	10	0 0 1
B 01	11	00	1 0 1
C 11	10	01	1 1 0
D 10	00	11	0 1 1

$Y_1 = y_1y_0$

w \ y_1y_0	00	01	11	10
0	0	1	1	0
1	1	0	0	1

$Y_1 = \bar{w}y_0 + w\bar{y}_0$

$Y_0 = \bar{y}_1\bar{y}_0$

w \ y_1y_0	00	01	11	10
0	1	0	0	0
1	0	0	1	0

$Y_0 = \bar{w}\bar{y}_1 + w\bar{y}_1$

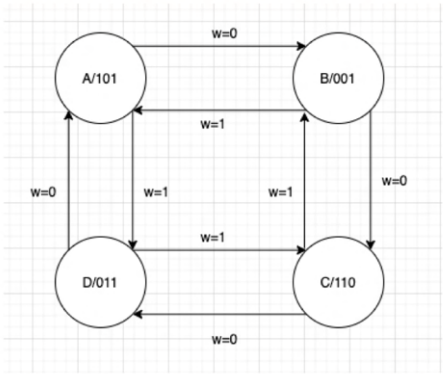
next →

$z_2 = y_0$ $z_1 = y_1$ $z_0 = \overline{y_1 + y_0}$

by inspection.

Q6B

Consider the following state diagram for an FSM with one input, w, and three outputs z_2, z_1, z_0.



a) Give the state table corresponding to the state diagram. Use A,B,C,D to represent the states in your state table.

b) Using the state encoding below, give the logic functions for the next-state logic, as well as the outputs z_2, z_1, z_0. Following the notation in the course text book, use Y_1, Y_0 to represent the next-state logic functions. Use K-maps to minimize the functions in SOP form.

State Code y_1y_0

A	00
B	01
C	11
D	10

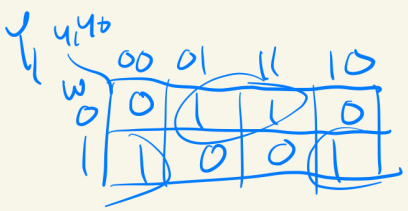
Q6)

a)

curr	next w=0	next w=1	$z_2z_1z_0$
A	B	D	101
B	C	A	001
C	D	B	110
D	A	C	011

b)

curr y_1y_0	next w=0 Y_1Y_0	next w=1 Y_1Y_0	$z_2z_1z_0$
A 00	01	10	101
B 01	11	00	001
C 11	10	01	110
D 10	00	11	011



$Y_1 = \bar{w}y_0 + w\bar{y}_0$

$Y_0 = \bar{w}\bar{y}_1 + wy_1$

$z_2 = \bar{y}_1\bar{y}_0 + y_1y_0$ // XNOR can't be minimized.

$z_1 = y_1$ $z_0 = \bar{y}_1 + \bar{y}_0$

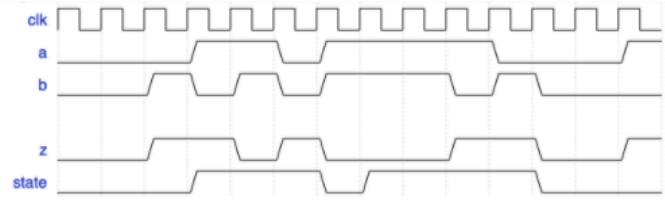
by inspection.

Question 7

4 pts

The waveform below represents a sequential circuit. The circuit consists of combinational logic and one bit of memory (i.e., one flip-flop). The output of the flip-flop is observable on the signal state. Output signal z is a function of a, b, and state.

Use the simulation waveforms below to determine what the circuit does and then write the logic expressions for z and for the next state value.



$\text{next state} = \overline{\text{state}}\,b + \text{state}\,a$

$z = a\bar{b} + \bar{a}b + \bar{a}\,\text{state}$

Q8A

Use Boolean algebra to prove the following:

$$\overline{ab + ac + bc} = \bar{a}\bar{b} + \bar{a}\,\bar{c} + \bar{b}\bar{c}$$

Show all your steps for full marks.

Q8) prove

$$\overline{ab + ac + bc} = \bar{a}\bar{b} + \bar{a}\bar{c} + \bar{b}\bar{c}$$

LHS : De Morgan's

$$(\overline{ab})(\overline{ac})(\overline{bc}) = (\bar{a}+\bar{b})(\bar{a}+\bar{c})(\bar{b}+\bar{c})$$

$$= (\bar{a} + \bar{a}\bar{c} + \bar{b}\bar{a} + \bar{b}\bar{c})(\bar{b} + \bar{c})$$

$$= \bar{a}\bar{b} + \bar{a}\bar{c} + \bar{a}\bar{c}\bar{b} + \bar{a}\bar{c} + \bar{b}\bar{a} + \bar{b}\bar{a}\bar{c} + \bar{b}\bar{c} + \bar{b}\bar{c} =$$

$$= \bar{a}\bar{b} + \underbrace{\bar{a}\bar{c} + \bar{a}\bar{c}\bar{b}}_{\bar{a}\bar{c}} + \underbrace{\bar{b}\bar{a} + \bar{b}\bar{a}\bar{c}}_{\bar{b}\bar{a}} + \bar{b}\bar{c}$$

$$= \bar{a}\bar{b} + \bar{a}\bar{c} + \bar{b}\bar{c}$$

= RHS

Q8B

Consider the *majority* logic function: $f = ab + ac + bc$

a) Use Boolean algebra to prove that: if a = b, then f = a.

b) Use Boolean algebra to prove that: if a = b', then f = c. (Note that b' means (not b))

c) Use Boolean algebra to prove that: $(a \oplus b)c + ab = f$

Show all your steps for full marks.

Q8) $f = ab + ac + bc$

a) if $a = b$ then

$f = a + ac + ab$

$= a(1 + c + b)$

$= a$

b) if $a = \bar{b}$

$f = \bar{b}b + \bar{b}c + bc$

$= \bar{b}c + bc$

$= c(\bar{b} + b) = c$

c) $(a \oplus b)c + ab = f$ (prove)

$(a\bar{b} + \bar{a}b)c + ab =$

$a\bar{b}c + \bar{a}bc + ab =$

$a(\bar{b}c + b) + b(\bar{a}c + a) =$

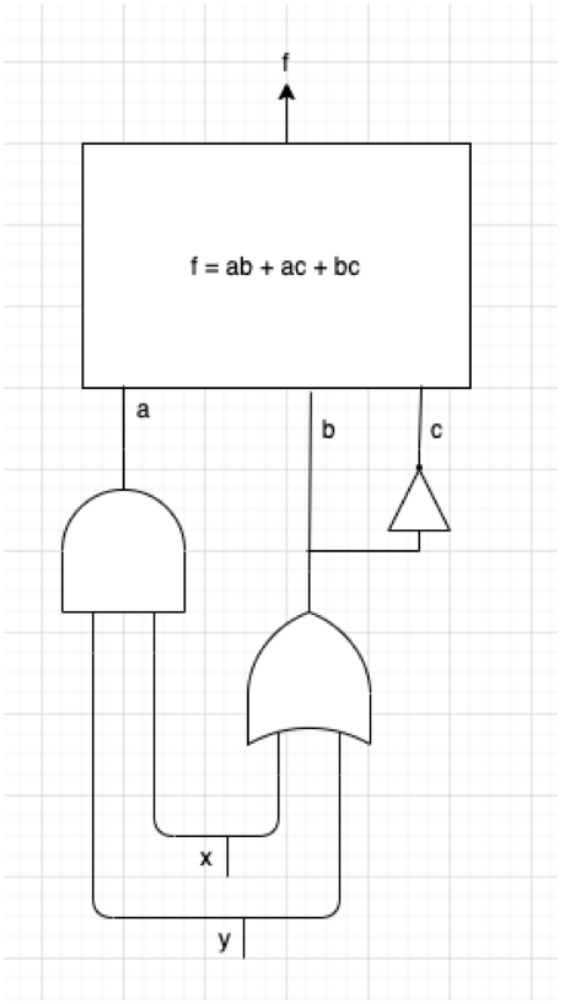
$= a(c + b) + b(c + a) =$

$ac + ab + bc + ab =$

$ac + ab + bc$

Q9A

Consider the following logic circuit.



Q9 $f = ab + ac + bc$

a)

x y	a b c
0 0	0 0 1
0 1	0 1 0
1 0	0 1 0
1 1	1 1 0

3 diff combinations can occur

abc = 000, 011, 100, 101, 111 can never occur

b)

a \ bc	00	01	11	10
0	X	0	X	0
1	X	X	X	1

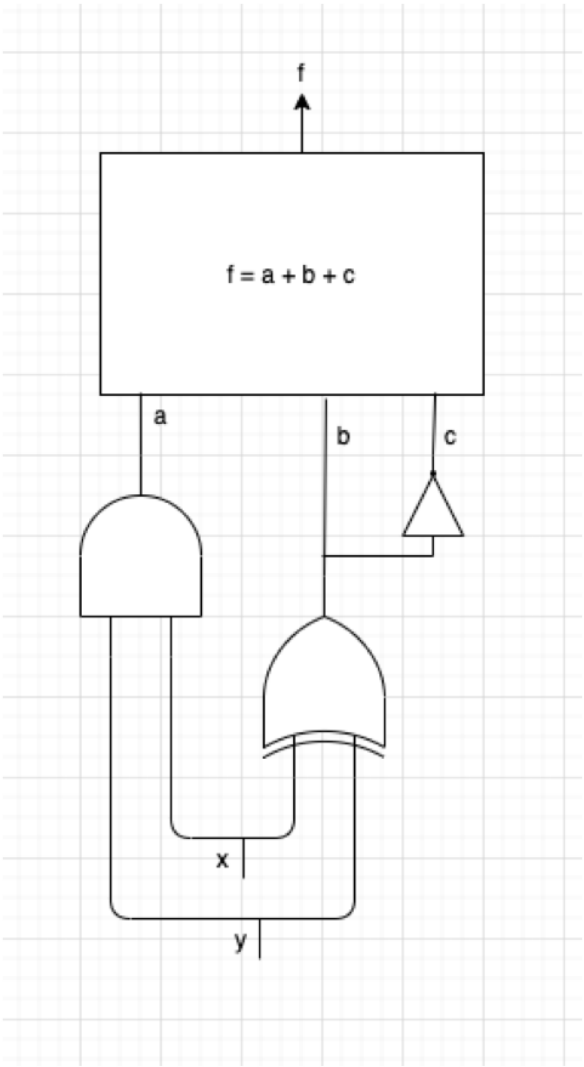
f = a

a) Because of the AND, OR and not gates shown, certain combinations of the logic signals a,b,c can never occur. For example, since c is the inverted form of b, these two signals can never take on the same value. List the combinations of a,b,c that can never occur.

b) Use a Karnaugh map to find the minimal sum-of-products form for function f (as given in the box in the figure), where the combinations from part (a) are treated as don't-cares. The function f should be represented in terms of a,b,c.

Q9B

Consider the following logic circuit.



Q9) $f = a + b + c$

x y	a b c
0 0	0 0 1
0 1	0 1 0
1 0	0 1 0
1 1	1 0 1

3 can occur

a) a, b, c = 000, 011, 100, 110, 111
Can never occur

b)

a \ bc	00	01	11	10
0	X	1	X	1
1	X	1	X	X

$f =$

$f = 1$

a) Because of the AND, EXOR and not gates shown, certain combinations of the logic signals a,b,c can never occur. For example, since c is the inverted form of b, these two signals can never take on the same value. List the combinations of a,b,c that can never occur.

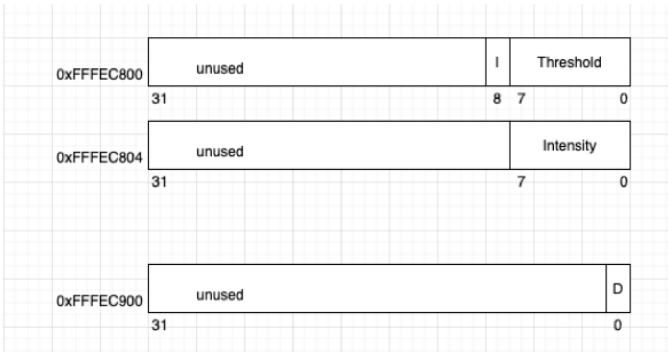
b) Use a Karnaugh map to find the minimal sum-of-products form for function f (as given in the box in the figure), where the combinations from part (a) are treated as don't-cares. The function f should be represented in terms of a,b,c.

Question 10

10 pts

Consider an ARM processor in a car that connects to memory-mapped devices that handle the airbag deployment in the event of a crash. For this question, you are to write an ARM program that uses interrupts to deploy the airbag in the event of a crash of sufficient intensity. The three relevant memory-mapped locations are shown below.

Interrupts are enabled by writing a 1 into the **I** bit shown. The 8-bit crash intensity threshold, **Threshold**, indicates the severity of crash required to cause an interrupt. Crashes below this intensity will not cause an interrupt. For this question, the threshold should be set to 128. When an interrupt occurs, the intensity of the crash causing the interrupt will be placed in the **Intensity** field. The interrupt can be cleared by a store (of any value) to the **Intensity** field. Finally, your interrupt service routine should deploy the airbag by storing a 1 in the **D** field.



a) Complete the code below to enable interrupts for the airbag deployment program. Set the stack pointer for SVC mode to 0x100000; set the stack pointer for IRQ mode to 0x200000. You may assume that the exception vector table has already been setup, and that a subroutine, CONFIG_GIC, exists to configure the GIC (generic interrupt controller).

```
.global _start
_start:
   // write your here code to setup the SPs

   BL   CONFIG_GIC    // configure the ARM generic interrupt controller

   // write your code here to enable interrupts for the airbag deployment devic
e and for the ARM
```

b) Assuming that the interrupt ID for the airbag device is 101 (in decimal), complete the code below for the IRQ_HANDLER subroutine. When the airbag interrupt is detected, your code should call a subroutine called AIRBAG_ISR. Your code should behave appropriately if an unknown interrupt occurs; specifically, your code should ignore the interrupt and return.

```
.global IRQ_HANDLER
IRQ_HANDLER:
    /* save R0–R3, because subroutines called from here might modify
       these registers without saving/restoring them. Save R4, R5
     because we modify them in this subroutine */
       PUSH  {R0–R5, LR}

       /* Read the ICCIAR from the CPU interface */
       LDR  R4, =MPCORE_GIC_CPUIF
       LDR  R5, [R4, #ICCIAR]    // read the interrupt ID

AIRBAG_CHECK:
       // write your code here to call the AIRBAG_ISR as appropriate

EXIT_IRQ:
       /* Write to the End of Interrupt Register (ICCEOIR) */
       STR  R5, [R4, #ICCEOIR]

       POP  {R0–R5, LR}
       SUBS  PC, LR, #4      // return from interrupt
```

c) Write code for the AIRBAG_ISR subroutine to deploy the airbag.

```
AIRBAG_ISR:
```

Q10)

Part a)

```
        MOV R0, #0b10010 // IRQ MODE
        MSR CPSR, R0
        LDR SP,=0x200000 // SP FOR IRQ MODE
        MOV R0, #0b10011 // SVC MODE
        MSR CPSR, R0
        LDR SP,=0x100000 // SP FOR SVC MODE
        BL CONFIG_GIC // EXISTING CODE
        LDR R0,=0xFFFEC800 // base address for airbag control
        MOV R1,#0b110000000 // set I = 1, set bit 7 to 1 so the threshold is 128.
        STR R1,[R0]
        MOV R0, #0b00010011 // enable interrupts in SVC mode
END:
        B END
```

Part b)

```
.global IRQ_HANDLER
IRQ_HANDLER:
    /* save R0-R3, because subroutines called from here might modify
       these registers without saving/restoring them. Save R4, R5
     because we modify them in this subroutine */
       PUSH  {R0-R5, LR}

       /* Read the ICCIAR from the CPU interface */
       LDR  R4, =MPCORE_GIC_CPUIF
       LDR  R5, [R4, #ICCIAR]     // read the interrupt ID

AIRBAG_CHECK:
       // write your code here to call the AIRBAG_ISR as appropriate
       CMP R5, #101
       BNE EXIT_IRQ
       B AIRBAG_ISR

EXIT_IRQ:
       /* Write to the End of Interrupt Register (ICCEOIR) */
       STR  R5, [R4, #ICCEOIR]

       POP  {R0-R5, LR}
       SUBS  PC, LR, #4       // return from interrupt
```

Part c)

```
AIRBAG_ISR:
        LDR R0, =0xFFFEC900
        LDR R1, =0xFFFEC800
        LDR R2, [R1,#4] // load the intensity field
        STR R2, [R1,#4] // clear the interrupt
        CMP R2, #0b10000000 // compare intensity with 128
        BLT EXIT_ISR // should never happen
        MOV R3, #1
        STR R3, [R0] // deploy the airbag
EXIT_ISR:
        MOVE PC, LR
```