1. Information representation question: (you knew there was gonna be one of these!) Consider a system that uses **12 bits**, with the radix point right in the middle. (That is, p=6.) Fill in the missing elements of the following table.

Number	Bit Pattern	Value of Representation
Value of this bit pattern; two's complement	101100.100100	-19.4375
Value of this bit pattern; unsigned binary	101100.100100	44.5625
Generate bit pattern for: two's complement	101110.110000	$-17\frac{1}{4}$
Value of this bit pattern: bit two's complement	111000.110100	-7.1875
Value of this bit pattern; unsigned binary	101010.101010	42.65625

2. General information question:

a) What is the basic fundamental principle on which all stored program computers are based?

Fetch Decode Execute

b) How is the concept of Memory Mapped I/O demonstrated in the PowerPC architecture?

It makes I/O look like memory so writing/reading values to the memory location where the I/O is located will read/write the output of the I/O like the LED's in Lab, you write to their memory location, the LEDs will light up

c) What instructions in the PowerPC move multiple register values to and from memory?

stmw → store multiple words

lmw → load multiple words

d) George is building a vector table for use with the PowerPC. His table starts at location 0x00190000. An Interrupt Service Routine (ISR) has been created to handle privilege violations, and this routine is smaller than 0x100 bytes. At what address should this ISR be located? (Or, at what address should the jump-to-the-ISR instruction be located? Same answer.) Priveledge violations occur at 0x700 offset the EUPR start

0x00190700

e) List the four instruction types and give an example of each from the PowerPC instruction set.

work → add r1, r2, r3

movement → stw r1, 0(r2)

program ctrl → branch = b loop

system ctrl → interrupts → rfi

f) System mode/user mode question 1: Why does the PowerPC have both a user mode and a system mode? That is, what is the reason for having a dual mode of operation?

For protection reasons

g) System mode/user mode question2: How does the system enter user mode? That is, what instruction activity/register activity results in the system entering user mode?

set the PR bit in the MSR

li r1, 0x0001 0000
mtspr msr, r1

h) Assume that the GPIO module for interacting with the push buttons is accesses ad address 0x81490000. Give the 4 instruction sequence needed to read the values of the buttons into R11.

lis r1, 0x81490000@h

loop: lwz r11, 0(r1)

b loop

#This is all I did in the lab, it would work for me but its less than 4 instructions need 2 more to set direction

-1

3. Instruction execution question 1: The following routine has created by a user for some strange manipulations on data. Hence, there appears to the observer some un-explainable operations (in other words, don't try to make sense out of what this routine does…) The routine "SUBR" is called in the code below, and the purpose of this question is to determine the final value of the registers *after* the code fragment executes. When the program activity is at the location indicated "before", a snapshot of the registers is given in the "Before" area of the table. Please indicate in the "After" area of the table the register contents when "after" is reached after execution of the subroutine (remember to enter values only for those registers that have changed). *Note that only half of the PowerPC registers are shown; the instructions do not touch the other registers.*

```
 Addr     Bits         Instruction
10024 60000000   before: nop
10028 48000305           bl QUES3
1002C 60000000   after:  nop

1032c 3D401234   QUES3:  lis r10,0x1234
10330 614A5678           ori r10,r10,0x5678
10334 39C00020           li r14,0x20
10338 7DC903A6           mtctr r14
1033c 7CADFE70   over:   srawi r13,r5,0x1f
10340 4200FFFC           bdnz over
10344 7022FFFF   otra:   andi. r2,r1,0xffff
10348 4182FFFC           beq 0,otra
1034c 7CC74B78           or r7,r6,r9
10350 7D086038           and r8,r8,r12
10354 61691221           ori r9,r11,0x1221
10358 4E800020           blr
```

Before		After	
r0 = 0x00000000	r1 = 0x11111111	r0 =	r1 =
r2 = 0x22222222	r3 = 0x33333333	r2 =	r3 =
r4 = 0x44444444	r5 = 0x55555555	r4 =	r5 =
r6 = 0x66666666	r7 = 0x77777777	r6 =	r7 =
r8 = 0x88888888	r9 = 0x99999999	r8 =	r9 =
r10 = 0xAAAAAAAA	r11 = 0xBBBBBBBB	r10 =	r11 =
r12 = 0xCCCCCCCC	r13 = 0xDDDDDDDD	r12 =	r13 =
r14 = 0xEEEEEEEE	r15 = 0xFFFFFFFF	r14 =	r15 =
LR →	0x00000000	LR →	
CTR →	0x00000000	CTR →	

4. Instruction execution question 2: Consider the code segment below. As in the previous question, this question basically is to determine the work done by the instructions, and to modify the registers and the memory segment shown below to reflect the instructions shown. And, once again, don't try to attach any meaning to the work done. And, once again, only half the registers are shown.

```
6788 3DC00005 before: lis r14,0x0005
678c 61CE0400         ori r14,r14,0x0400
6790 89EE0016         lbz r15,0x16(r14)
6794 7C855214         add r4,r5,r10
6798 7D495B78         or r9,r10,r11
679c 806E001C         lwz r3,0x1c(r14)
67a0 912E002C         stw r9,0x2c(r14)
67a4 990E0032         stb r8,0x32(r14)
67a8 7C4033B8         nand r0,r2,r6
```

Before		After	
r0 = 0x00000000	r1 = 0x11111111	r0 = 0xdddddddd	r1 =
r2 = 0x22222222	r3 = 0x33333333	r2 =	r3 = 0x54321065
r4 = 0x44444444	r5 = 0x55555555	r4 = 0xffffffff	r5 =
r6 = 0x66666666	r7 = 0x77777777	r6 =	r7 =
r8 = 0x88888888	r9 = 0x99999999	r8 =	r9 = 0xBBBBBBBB
r10 = 0xAAAAAAAA	r11 = 0xBBBBBBBB	r10 =	r11 =
r12 = 0xCCCCCCCC	r13 = 0xDDDDDDDD	r12 =	r13 =
r14 = 0xEEEEEEEE	r15 = 0xFFFFFFFF	r14 = 0x00050400	r15 = 0x000000CD

	Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00504	40000400	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF	01
05041	40000410	23	45	67	89	AB	CD	EF	FE	DC	BA	98	76	54	32	10	65
05042	40000420													BB	BB	BB	BB
05043	40000430			BB													

0x 55555555
0x AAAAAAAA

A = 1010
B = 1011
D = 1011

2 = 0010
6 = 0110
& = 0010
n& = 1101
D

5. Interrupt question #1: This question deals with setting up I/O modules and interrupts. In the space below provide the code to configure the ML403 sort-of like we had it in the laboratory, but fixed so that the only active interrupt is from the UART. So, order things appear in registers is UART – Push Buttons – Timer 1 – Timer 2. UART to be set up for receive interrupt only, Baud Rate of 19,200 (divider value: 0x145), located at address 0x83e00000. LEDs at 0x82600000 (4 bits), LEDs at 0x82610000 (32 bits), LEDs at 0x82620000 (32 bits), LEDs at 0x82630000 (5 bits), push buttons at 0x82640000 (5 bits), don't include instructions for the LCD, timer 1, or timer 2, do include instructions to configure interrupt controller at 0x83800000. Interrupt table lives at 0x20000000.

```
.set LCR,0x100c      lis r10,0x83e00000@h  # UART
.set DLM,0x1004      li  r11,0x80          # set LCR(7) to 1
.set DLL,0x1000      stw r11,  LCR(r10)    # set LCR(7) to 1
.set IER,0x1004      li  r12,0x01          # Load Divisor latch MSB
.set LED1,0x8263xxxx stw r12, DLM(r10)     # store Divisor Latch MSB
.set LED2,0x8262xxxx li  r13,0x45          # Load Divisor Latch LSB
.set LED3,0x8261xxxx stw r13,DLL(r10)      # store Divisor Latch LSB
.set LED4,0x8260xxxx li  r14,0x08          # Enable UART Interrupts
.set ISR,0x0         stw r14,IER(r10)      # enable parity, 2 stop bits 8 bits/char
.set IAR,0x08        li  r11,0x07
.set MER,0x0c        stw r11,LCR(r10)
                     lis r15,0x2000        # EVPR address
                     mtevpr r15
                     lis r16,0x8380        # interrupt ctrler
                     li  r17,0x8
                     stw r17,ISR(r16)      # set ISR what about IER
                     stw r17,IAR(r16)      # set IAR
                     li  r18,0x03
                     stw r18,MER(r16)
                     li  r19,0
                     lis r20,LED1@h        # set LEDs as output
                     stw r19,4(r20)
                     lis r21,LED2@h
                     stw r19,4(r21)
                     lis r22,LED3@h
                     stw r19,4(r22)
                     lis r23,LED4@h
                     stw r19,4(r23)
                     wrteei 1              # enable interrupts
```

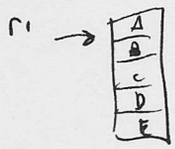
1100

6. Interrupt question #2. This question deals with the steady state activities associated with interrupt set up in Question 5. On this page, give the code for the Interrupt Service Routine for the UART, since we don't have the interrupts turned on for switches or timers. So, include here the code needed to implement a receive-the-character routine with the UART. When received-new-character is detected – the interrupt condition – take the appropriate steps – in code – to place the character in the mailbox located at 0xffff2200. Be sure to reset any flags that need to be reset, etc.

```
.set RBR, 0x1000
.set LSR, 0x1014
.set IAR, 0xc
        lis r1, 0x8380 # Interrupt ctrlr   ] this would be
        lis r2, 0x83e0 # UART              ] set up before ISR
                                             (no - set up in ISR)
        .org 0x500 (20000)

        lwz r10, RBR(r2)              # load character from buffer
        li r11, 0x60                  # set Line status register
        stw r11, LSR(r2)
        li r12, 0x8                   # clear Interrupt Available
        stw r12, IAR(r1)
        lis r13, 0xffff2200@h         # load mailbox location
        ori r13, r13, 0xffff2200@l
        stw r10, 0(r13)               # store recived int mbox
        rfi
```

7. In the space provided below, provide two sets of instructions, one for the "push" operation and one for the "pop" operation. Assume that R1 is the stack pointer, and it is currently pointed at the top-of-stack. The operation to perform in this case is a call-to-subroutine operation that uses the stack to do both parameter passing and subroutine linkage. The parameter passing is accomplished by pushing registers R24-R31 onto the stack, then pushing the current return address. The pop is the reverse of these operations. On the left side below, put instructions to implement the "push"; on the right side, put instructions to implement the corresponding "pop" operation. That is, put instructions to push or pop the various registers with the required operations on the stack, and also adjust the stack pointer appropriately. Remember that stacks grow toward lower addresses.



7.4 = 28

-32

stor r24 to r31 into memory

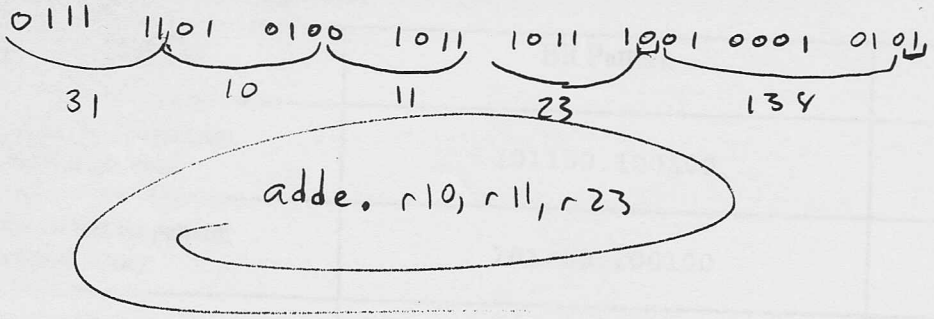
```
stmw  r24, 0(r1)
mflr  r2          don't destroy current val
stw   r2, 32(r1)
addi  r1, r1, -36
```

```
addi  r1, r1, +36      don't
lwz   r2, 32(r1)
mtlr  r2
lmw   r24, 0(r1)
```

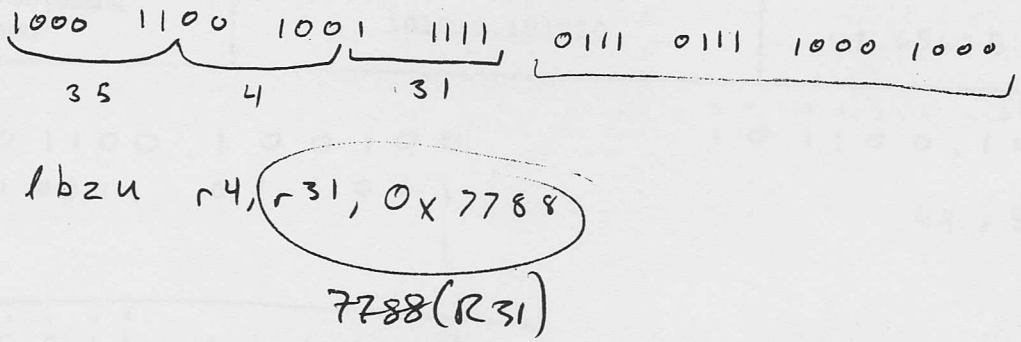
0	24
4	25
8	26
12	27
16	28
20	29
24	30
28	31

8. Instruction coding question:

a) What is the instruction represented by the bit pattern 0x7D4BB915?



b) What is the instruction represented by the bit pattern 8C9F7788.



c) Give the coding for the instruction `andc. R8,r9,r10`