1. General information question:
   a) What is the basic tenet of all stored program computers?

Fetch → Decode → Execute

   b) Identify the four different types of instructions and give an example of each from the PowerPC instruction set.

Work: add

Move: lwz

Program Control: bl

System Control: rfi, wrtee

   c) After a normal interrupt occurs, where will the system store the address of the instruction that should be executed when the Interrupt Service Routine completes?

SRR0

   d) True or false: 0x01234000 is a valid content for the EVPR.

True (-2)

16 lsb must be zero.

   e) An `add. r3,r4,r5` instruction has just completed and the value written to r3 is 0xFFF45678. What are bits 0, 1, and 2 of the Condition Register?

result compared to zero!

bits	value
0	1
1	0
2	0

1111 → negative. (-1)

   f) What is the purpose of the TRI register (also known as the DDR register) of the Xilinx GPIO IP?

This sets the GPIO unit as an input (1) or an output (0)

flag of the Uartlite system to be set; then to clear the bit and return. This subroutine was used in a system that needed a UART, but without using the interrupt system. The programmer called this routine from a larger system handling routine. This code fragment is as follows:

```
     Addr   Bits          Instr
6600 60000000          nop
6604 48000009          bl getc
6608 60000000          nop

660c 3D008400  getc:   lis r8,0x8400      # UART at addr 0x84000000
6610 81280008  again:  lwz r9,0x8(r8)     # Stat reg offset by 8 bytes
6614 712A0001          andi. r10,r9,0x0001
6618 4182FFF8          bt 2,again
661c 81680000          lwz r11,0(r8)      # Receive FIFO offset by 0 bytes
6620 2C8B0033          cmpwi 1,r11,0x0033 # 0x33 is ascii '3'
6624 4086FFEC          bf 6,again
6628 4E800020          blr
```

This question deals with the registers used in the routine. Below is a before and after representation for some of the registers. The before values are given (values of registers before executing the instruction at 0x6604); fill in the after values (values of registers after executing the instruction at 0x6608). The UART system has been initialized elsewhere, so don't worry about that aspect. Only write in the *After* area those registers that have been changed by the above code fragment, and in those boxes place the correct final value for the register.

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 = 0x84000000	r9 = 0x84000008 (-2)
r10 = 0xAAAAAAAA	r11 = 0xBBBBBBBB	r10 = 0x00000003	r11 = 0x84000002
r12 = 0xCCCCCCCC	r13 = 0xDDDDDDDD	r12 =	r13 =
r14 = 0xEEEEEEEE	r15 = 0xFFFFFFFF	r14 =	r15 =
lr = 0xABCDEF00	ctr = 0x00000000	lr = (-2)	ctr =
cr = 0x00000000	tsr = 0x00000000	cr =	tsr =

lr - link register = 6608. return address.

r9 = 0x84000001

r10 = 0x00000001

r11 = 0x00000033

cr = 0x42000000

Data movement question: In the first laboratory you explored moving information to and from memory with various load and store instructions. Below is a small code fragment, followed by another memory and register contents description. The code fragment is set up to be somewhat tricky, and not particularly straightforward, but implement the work of each instruction and you should be okay. Identify the locations in memory and the registers that are changed by the code fragment, and give the updated values.

```
Addr    Bits              Insruction
5110 38603000 strt:  li r3,0x3000
5114 39C00004        li r14,4
5118 7C647030        slw r4,r3,r14    # shift left word
511c 7C841A14        add r4,r4,r3
5120 38840040        addi r4,r4,0x40
5124 80C40014        lwz r6,20(r4)
5128 A0E40012        lhz r7,18(r4)
512c 89040007        lbz r8,7(r4)
5130 99240029        stb r9,0x29(r4)
5134 B1440032        sth r10,0x32(r4)
5138 9164003C        stw r11,0x3C(r4)
```

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 =

Address																
00033040	01	23	45	56	89	AB	CD	EF	00	11	22	33	44	55	66	77
00033050	88	99	AA	BB	CC	DD	EE	FF	12	13	14	15	16	17	18	19
00033060																
00033070																

4. Instruction/data structure question: In the space provided below, present code for the following problem. There is an array of words starting at address 0x8000 that contains 0x1230 signed binary numbers. Determine how many of these values are greater than zero, how many are less than zero, and how many are equal to zero. Leave these counts in registers r8, r9, and r10.

```
.org .
r8 = 0
r9 = 0      # used to count how many are LT, GT or EQ
r10 = 0

loop:  li r1, x8000        # base address
       li r2, x1230        # counter
       stw r3, r2(r1)      # store word from memory into r3, where r1 is the base address and r2 is the offset.
       subi r2, 8, r2
       cmpwi r3, 0         # compare word to zero.
       blt LESSTHAN
       bgt GREATERTHAN
       beq EQUAL
       cmp r2, 0           # compare counter to 0, if it is, go to the end, we are done counting
       bt END
LESSTHAN:
       addi r8, r8, 1
       b loop
GREATERTHAN:
       addi r9, r9, 1
       b loop
EQUAL:
       addi r10, r10, 1
       b loop.
END:
here: b here

r8 = # of <
r9 = # of >
r10 = # of =
```

(1230 #'s. signed starting at x8000.)

(decrement the counter by size of "word")

(Grader notes: signextend; ori; won't work; lwz?; why? ?; never get here; bdnz?)

5. Interrupt question: This question has two parts. The first is setup/initialization, the second is steady state. Much of the configuration of the UARTLite from Xilinx is done in the initialization of the board, as opposed to the initialization of the processor. Nevertheless, we need to initialize the system for interrupt driven activity. In the space provided below, give instructions that will set up the system (UART, Processor, etc.) for interrupt driven applications for receiving characters. Assume that the system under consideration is configured as we have done it in the laboratory (UARTLite at 0x84000000; interrupt controller at 0x81800000; UARTLite interrupt bit in the interrupt controller is the '8' bit (IBM numbering this would be 28, normal numbering would put it at position/weighting 3)).

```
.set UART@H  0x84000000        IC uart bit = 8
.set Rx FIFO@H  x00000004      # Initialize
.set IC@H    0x81800000
.set IER     0x00000008
lw r1, UART@H
lw r2, Rx FIFO@H
li r3, IC@H.
li r4, IER.
li r4, 1  # Enable the Interrupt Register
```

5 instr?

Load into registers r1 - r4.

EVPR?

For the second part of this question, give code for an interrupt service routine that will handle the flags of the UART as needed to verify that there is a character available and then place the character in a mail box located at address 0x00FF0800. Don't worry about context-switch of the registers; assume the ISR can use any register it deems appropriate.

```
li r5, 0x00FF0800@H   # r5 holds the mailbox
ori r5, r5, 0x00FF0800@L   address.

loop: li r6, r2(r1).   # Load value from Rx FIFO
                       into r6.
cmp r6, 0              # is r6 empty?
bf MAILBOX             # if not, go to mailbox.
b loop

MAILBOX:
stw r6, 0(r5)          # store the value from
                       the Rx FIFO into the
                       MAILBOX.
```

incomplete!

5. Interrupt question. This question has two parts. The first is setup/initialization, the second is steady state. Much of the configuration of the UARTLite from Xilinx is done in the initialization of the board, as opposed to the initialization of the processor. Nevertheless, we need to initialize the system for interrupt driven activity. In the space provided below, give instructions that will set up the system (UART, Processor, etc.) for interrupt driven applications for receiving characters. Assume that the system under consideration is configured as we have done it in the laboratory (UARTLite at 0x84000000; interrupt controller at 0x81800000; UARTLite interrupt bit in the interrupt controller is the '8' bit (IBM numbering this would be 28, normal numbering would put it at position/weighting 3)).

```
.set UART@H   0x84000000      IC uart bit = 8
.set RxFIFO@H x00000004       # Initialize
.set IC@H     0x81800000
.set IER      0x00000008
lis r1, UART@H
lis r2, RxFIFO@H              # Load into registers
li r3, IC@H                     r1 - r4.
li r4, IER
li r4, 1      # Enable Interrupt Register
```

5 instr?

EUPR?

also?

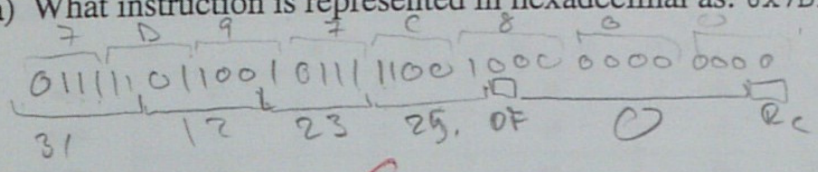
For the second part of this question, give code for an interrupt service routine that will handle the flags of the UART as needed to verify that there is a character available and then place the character in a mail box located at address 0x00FF0800. Don't worry about context-switch of the registers; assume the ISR can use any register it deems appropriate.

```
lis r5, 0x00FF0800@H        # r5 holds the mailbox address
ori, r5, r5 0x00FF0800@L

li r6, r2(r1).              # Load value from RxFIFO into r6.
cmp r6, 0                   # is r6 empty?
bf MAILBOX                  # if not, go to mailbox.
b loop

MAILBOX:
   stw r6, 0(r5)            # store the value from the RxFIFO into the MAILBOX.
```

loop?

incomplete!

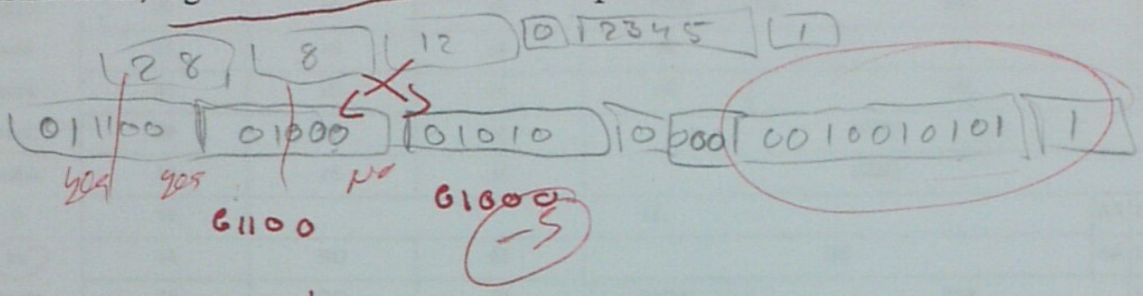
a) What instruction is represented in hexadecimal as: `0x7D97C800`?



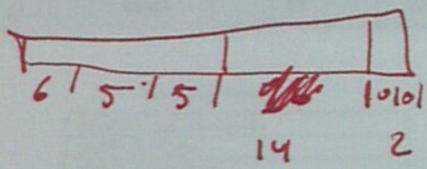
add r12, r23, r25, -5

cmp r23, r25

b) Give the coding for the instruction '`andi. r8,r12,0x2345`'. Remember that on logical instructions, register order is reversed in bit pattern.



c) Assume that a conditional branch instruction is located at address 0x00080000. What is the highest address that can be reached by this instruction? That is, at what address is located the label that is the target of the branch?



Route, first bit = 0, rest of bits are 1's.

0111 1111 1111 1100

7 F F C.

-5

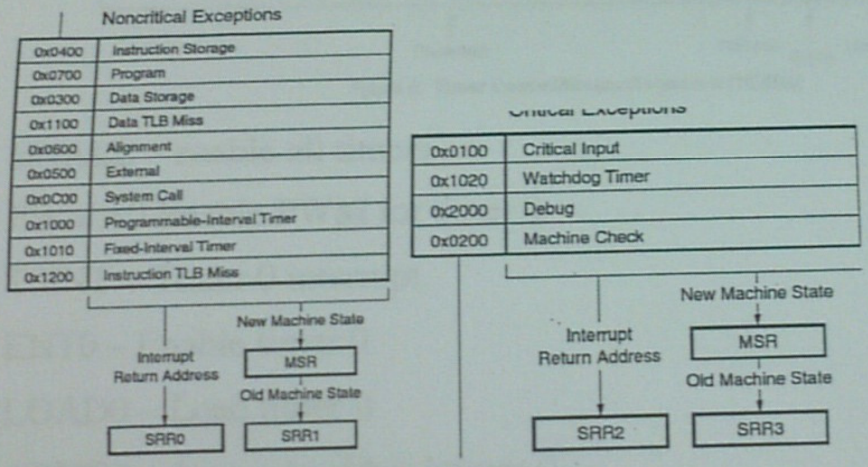
80000
+ 7FFC

87FFC.

Table B-1 lists the PPC405 instruction set in alphabetical order by mnemonic.

Table B-1: **Instructions Sorted by Mnemonic**

	0	6	9	11 12	14	16 17	20	21	22	26	30	31
add	31	rD		rA		rB		OE		266		Rc
addc	31	rD		rA		rB		OE		10		Rc
adde	31	rD		rA		rB		OE		138		Rc
addi	14	rD		rA		SIMM						
addic	12	rD		rA		SIMM						
addic.	13	rD		rA		SIMM						
addis	15	rD		rA		SIMM						
addme	31	rD		rA		00000		OE		234		Rc
addze	31	rD		rA		00000		OE		202		Rc
and	31	rS		rA		rB				28		Rc
andc	31	rS		rA		rB				60		Rc
andi.	28	rS		rA		UIMM						
andis.	29	rS		rA		UIMM						
b	18	LI									AA	LK
bc	16	BO		BI		BD					AA	LK
bcctr	19	BO		BI		00000				528		LK
bclr	19	BO		BI		00000				16		LK
cmp	31	crfD	00	rA		rB				0		0
cmpi	11	crfD	00	rA		SIMM						
cmpl	31	crfD	00	rA		rB				32		0



Register Name	Base Address + Offset (Hex)	Access Type	Abbreviation	Reset Value
Interrupt Status Register	C_BASEADDR + 0x0	Read / Write	ISR	All Zeros
Interrupt Pending Register	C_BASEADDR + 0x4	Read only	IPR	All Zeros
Interrupt Enable Register	C_BASEADDR + 0x8	Read / Write	IER	All Zeros
Interrupt Acknowledge Register	C_BASEADDR + 0xC	Write only	IAR	All Zeros
Set Interrupt Enable Bits	C_BASEADDR + 0x10	Write only	SIE	All Zeros
Clear Interrupt Enable Bits	C_BASEADDR + 0x14	Write only	CIE	All Zeros
Interrupt Vector Register	C_BASEADDR + 0x18	Read only	IVR	All Ones
Master Enable Register	C_BASEADDR + 0x1C	Read / Write	MER	All Zeros

Notes:

Table 4: **XPS Timer/Counter Register Address Map**

Register	Address (Hex)	Size	Type	Description
TCSR0	C_BASEADDR + 0x00	Word	Read/Write	Control/Status Register 0
TLR0	C_BASEADDR + 0x04	Word	Read/Write	Load Register 0
TCR0	C_BASEADDR + 0x08	Word	Read	Timer/Counter Register 0
TCSR1	C_BASEADDR + 0x10	Word	Read/Write	Control/Status Register 1
TLR1	C_BASEADDR + 0x14	Word	Read/Write	Load Register 1
TCR1	C_BASEADDR + 0x18	Word	Read	Timer/Counter Register 1

Control/Status Register 0 (TCSR0)

The Figure 6 and Table 7 shows the Control/Status register 0. Control/Status Register 0 contains the control and status bits for timer module 0.

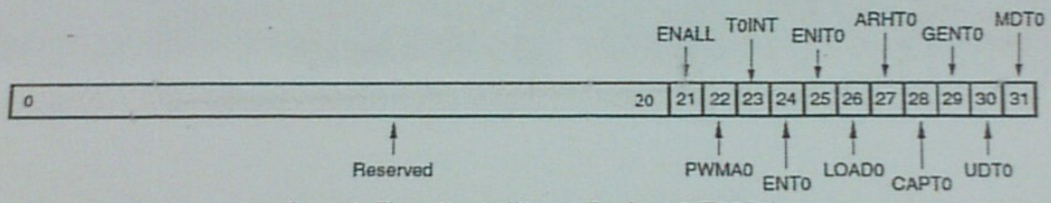


Figure 6: **Timer Control/Status Register 0 (TCSR0)**

ENALL – enable all timers

PWMA0 Enable PWM for timer 0

T0INT – Timer 0 interrupt

ENT0 – Enable timer 0

LOAD0 – Load timer 0

ARHT0 – Auto reload/hold timer 0

CAPT0 – enable capture trigger timer 0

GENT0 – Enable External Generate

UDT0 - Up Down Timer0

MDT0 – Timer 0 mode