1. Information representation. After all the time we spent worrying about this, we need to have some question about number representation. Consider number system(s) that contain 10 bits, with the radix point just to the right of the fourth bit from the right. That is, numbers are formed according to the pattern xxxxxx.xxxx. For that arrangement of bits, fill in the missing elements of the following table. (Remember that maximum is right-most on the number line; minimum is left-most on the number line.)

Value	Unsigned binary pattern	Twos-complement pattern
Maximum	111111.1111	011111.1111
Minimum	000000.0000	100000.0000
$12\ ^{11}/_{16}$	001100.1011	001100.1011
$-7\ ^{5}/_{8}$	N/A	100101.1010 (-2)
$21\ ^{13}/_{16}$	010101.1101	010101.1101
$-13\ ^{9}/_{16}$ (-2)	N/A	110010.1001

2. General information question:

a) Where is the return address stored on a go-to-subroutine instruction?

LR

link register.

b) True or false: It is okay to start the vector table at 0x45678000.

multiples of 64K.

False

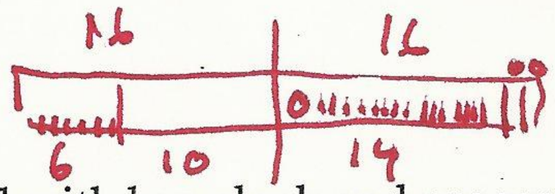
c) The PPC system is placed in User Mode by doing what?

Switch between user, supervisor mode by writing a 1, 0 to the MSR.

d) A conditional branch instruction (say, bt 2, label) is located at address 0x00110000? What is the highest address at which the target of the branch can be located?

0X00117FFC

16 bits are address



just add 7FFC to the above.

e) Assume that the ML403 board's PPC processor is configured with bus clock and processor clock both functioning at 100 MHz. How long does it take for TBU to change values?

40(s)

20 s. (-2)

3. Interrupt Controller Question: In the table below – which shows registers in the interrupt controller – identify the registers that need to be initialized, and give values for each for the following system: The modules have been configured differently from the system in the lab. The new arrangement calls for three UARTs, a Timer, and three GPIO modules: UART1, UART2, UART3, Timer, Buttons1, Buttons2, and Buttons3. For this question, enable UART1, UART2, Buttons2, and Buttons3. Also, the software activation of interrupts is not to be utilized. Assume that you want to assert the appropriate bits to reset any flags that may have remained from an earlier program.

Addr Offset	Register	Bit Pattern
0x00	ISR	0000 clear
0x04	IPR	Read only
0x08	IER	1111, 0x00000003
0x0C	IAR	1111, 0x00000003
0x1C	MER	11, 0x00000003

status
pending
enabled
acknowledge
Enable

Now, in the space provided below, give instructions that will establish the bit patterns given above as well as to a) correctly initialize the EVPR to its lowest legal value and b) set up any enabling activity needed to allow interrupts to occur. The interrupt controller has been located at address **0x81820000.**

```
.org 0x500
li r31, 0x8182          # pointer to INTC.
or. r31, r31, 0000.
li r30, 0
mtevpr r30              # set EVPR
li r29, 0x63            # bit patterns
li r28, 3.              # send bit patterns to:... IER IAR of.

stw r29, 8(r31)         } #INTC
stw r29, 0xC(r31)       }
stw r29, 8(r1)          } # UART1
stw r29, C(r1)          }
stw r29, 8(r2)          } # UART2
stw r29, C(r2)          }
stw r29, 8(r3)          } # Button2
stw r29, C(r3)          }
stw r29, 8(r4)          } # Button3.
stw r29, C(r4)          }
stw r28, 1C(r31)        # Set MER
```

UART 1 & Buttons 2 & 3

```
li r1, 0x8183
li r2, 0x8184
li r3, 0x8188
li r4, 0x8189
```

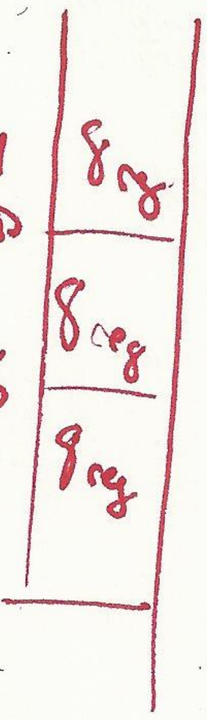
??

write: IL

4. ISR question: A system utilizes the PIT interrupt mechanism to cause activity every so often. The programmer is utilizing register R3 to point to a data storage/exchange area where the contexts are stored. Assume the contexts are 8 register values (R24-R31) as we did in the class examples. Create an Interrupt Service Routine for the PIT that handles PIT stuff according to this algorithm: Save current context at R3 offset 0; load context at offset 64; handle any PIT activity needed to service the interrupt, put a flag (any nonzero value) in the mailbox located at 0xF00004; return contexts to appropriate locations; return from interrupt.

```
# JNTC.
# Poll
#  - should be.
#  - set bits.
#  - done.

why? → li r3, 0x00F0  # mailbox.
       ori, ori, x0084
R3 already busy
       li r31, 8182 0h.
       li r2, 1          # flag.
       stmw r24, 0(R3)   # context switch.
       lmw r24, 64(R3)
```



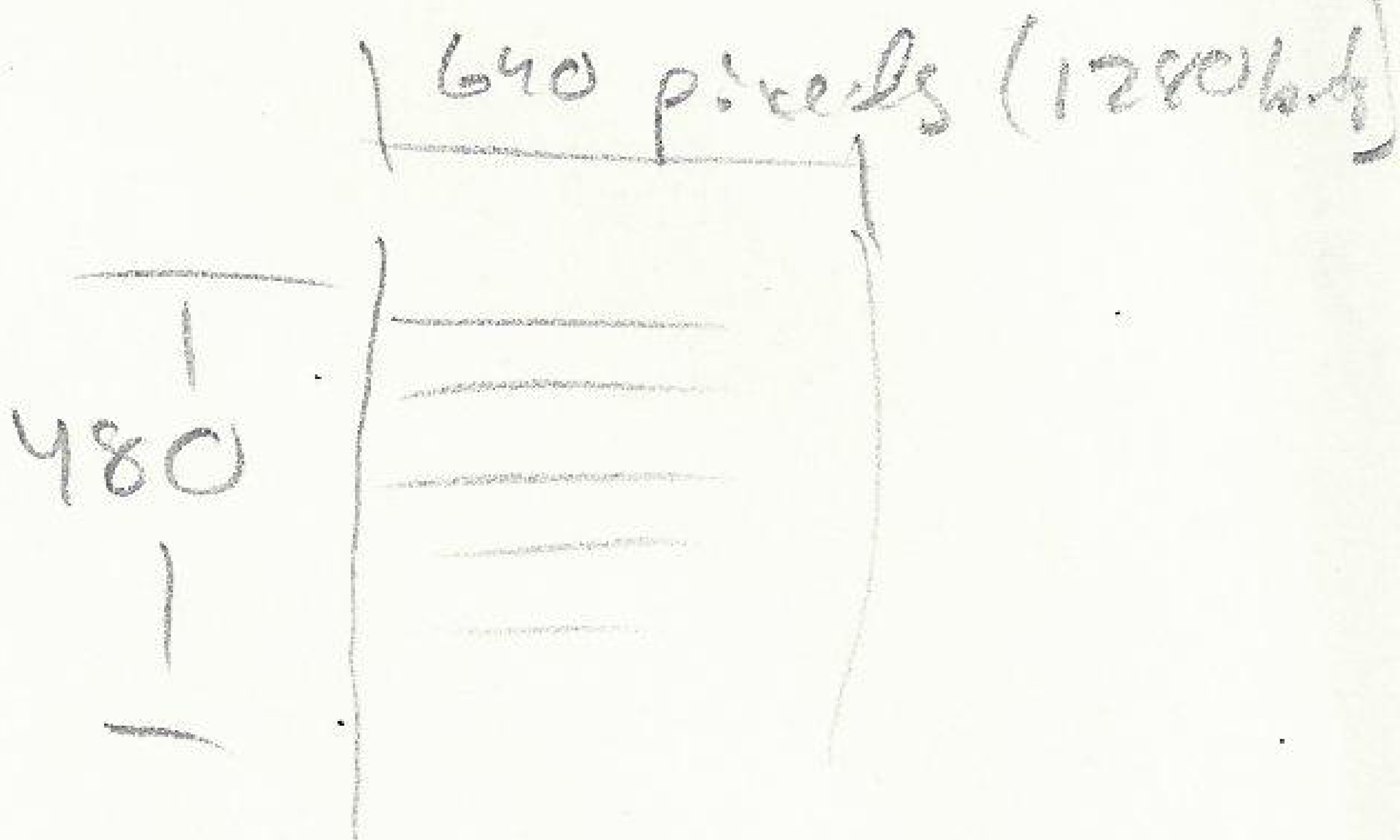

```
poll:  lwz r1, 4(r31)          # last data from pending register.
       andi. r1, r1, 0b1000    # check for interrupt.
never put loop in ISR
       beq poll                # if no interrupt look again.
       li r30, 0b0000          # if there is an interrupt.
       stw r30, C(r31)         # clear IAR.
       stw r2, 8(r3)           # set flag in mailbox.
       stmw r24 64(r3)         # return from context switch.
       lmw r24 0(r3)
rf:                            # return from interrupt
```

same?

P15 bit clear - bit 4.

5. Data structure question: A user has created an image for 'normal' television, which is 480 rows of 640 pixels stored in a two dimensional array of bytes. (Each pixel consists of one byte). The array starts at address 0x40000. Create code that will look through the data and count the number of pixels that have an absolute value greater than 0x38. Note: the pixels are always considered positive; that is, the representation is unsigned binary.

8 16 32 64
bit byte hw word
↑
1 pixel.




```
      .org 0x3000.                       which is set where?
                                          won't fit.
      li r31 0x0004@h.   # pointer to base.
loop: cmpwi r2, 307,200  # are we at the end      use ctr!
      beq end            # if so, exit,
      lbz r30, 0(r31).   # get value from array.
      addi r31, r31, 2   # go to next location.
      addi r2, r2, 1     # counter for what pixel we are on.
      cmpwi r30, 0x38    # compare to x38.
ble?  blt loop           # if <, loop again.
      addi r1, r1, 1     # if >, increment counter for
      b loop.            the pixels >x38.

end:  here: b here.
```

the array is stored in consecutive memory locations.

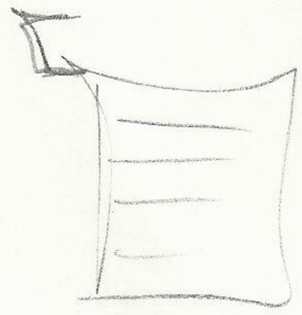
640 pixels x 480 rows = 307,200 pixels.

right idea
couple of oop's

```
   1
   3
  640
1 480
 51200
256000
307200
```

6. Another data structure/register manipulation question: We recently revisited the recursion problem, where current 'stuff' – information in registers – needs to be saved on the stack and later restored from the stack. In the space below, provide code for dealing with four registers on the stack in a manner to permit recursion. Use R31 for the return address. Remember that the stack pointer is located in R1.

li r1, 0x---- ptr



push

mr r1, r31 ?

addi, r1, r1, -16

stmw r28, 0(r1).

mr r31, r1 ?

↑

± load return address.

± move to higher in the stack (-16).

± store registers.

± set return address to the top of the stack.

nop

bl target

nop.

4 registers.

pop

mr r1, r31

ldmw, r28, 0(r1).

stuff missing here

± load return address.

± load values from the stack, starting with the current return address.

.org 0x3000

nop

mflr r31

add: r1, r1, -16

stmw r28, 0(r1)

bl target.

lmw r28, 0(r1)

add: r1, r1, 16.

mtlr r31

nop

.org 0x3200

target: nop.