Name:____________________________________

READ AND FOLLOW THESE INSTRUCTIONS.

- Do not begin until you are told to do so.
- Write your name legibly on *every page*.
- You have 50 minutes; budget your time. The questions are not of equal weight; do not spend too much time on a question that is not worth many points.
- Read through all of the questions before starting to work.
- If you run out of space, continue working on the back of the *same* sheet. Your exams may be taken apart so that the questions can be graded separately.
- This exam is open book, open notes. You may use any reference material you brought with you. You may *not* share reference materials with other students.

Honor Code statement: I have neither given nor received aid on this exam.

Signature: ________________________________

Question	**Points**	**Score**
1	15	
2	40	
3	45	
Total	100	

1. (15 pts) Consider the following assembly-language program fragment:

```
        .data
array:  .byte  0x12, 0x34, 0x89, 0xAB
        .text
_start: li     r1, array   # load the address ‘array’ into r1
        lwa    r3, 0(r1)   # load a signed word into r3
        lwz    r4, 0(r1)   # load an unsigned word into r4
        lha    r5, 2(r1)   # load a signed halfword into r5
        lhz    r6, 2(r1)   # load an unsigned halfword into r6
        lba    r7, 2(r1)   # load a signed byte into r7
        lbz    r8, 2(r1)   # load an unsigned byte into r8
```

a. (9 pts) One or more of the instructions in the program are invalid or do not do what the comment indicates. In the space below, write each incorrect instruction and give a valid instruction or instruction sequence that would do what the comment indicates.

`li    r1, array`	should be	`lis   r1, array@h`
		`ori   r1, r1, array@l`
			(can’t load 32-bit immediate in one instruction)

`lwa   r3, 0(r1)`	should be	`lwz   r3, 0(r1)`	(no `lwa` instruction exists)

`lba   r7, 2(r1)`	should be	`lbz   r7, 2(r1)`
		`extsb r7, r7`
			(no `lba` instruction exists)

b. (6 pts) After applying your fixes and executing the instructions (from `_start` to the end), what would be the full 32-bit contents of the following registers (in hexadecimal)?

Register	**Contents**
R3	0x123489AB
R4	0x123489AB
R5	0xFFFF89AB
R6	0x000089AB
R7	0xFFFFFF89
R8	0x00000089

2. (40 pts) Use the following code fragment and memory dump to answer the following questions. *Read all of the questions before you begin.*

```
        li     r14,0x1FFC
        li     r15,4
        lwzux  r1,r14,r15
        lwzux  r2,r14,r15
        lwzux  r3,r14,r15
        lwzux  r4,r14,r15
        li     r5,0

loop1:  cmp    r5,r1
        beq    done1
        lbzx   r6,r5,r2
        lbzx   r7,r5,r3
        add    r8,r6,r7
        stbx   r8,r5,r4
        addi   r5,r5,1
        b      loop1

done1:  li     r5,0

loop2:  cmp    r5,r1
        beq    done2
        li     r6,0
        subf   r7,r5,r1
        addi   r7,r7,-1

loop3:  cmp    r6,r7
        beq    done3
        addi   r8,r6,1
        lbzx   r9,r6,r4
        lbzx   r10,r8,r4
        cmp    r9,r10
        ble    skip
        stbx   r10,r6,r4
        stbx   r9,r8,r4
skip:   addi   r6,r6,1
        b      loop3

done3:  addi   r5,r5,1
        b      loop2

done2:  addi   r4,r4,-4
        lwzu   r29,4(r4)
        lwzu   r30,4(r4)
        lwzu   r31,4(r4)
```

Mem Address	**+0**	**+1**	**+2**	**+3**
1FFC	00	01	02	03
2000	00	00	00	0C
2004	00	00	20	10
2008	00	00	20	20
200C	00	00	20	30
2010	0B	0A	09	08
2014	07	06	05	04
2018	03	02	01	00
201C	FF	FF	FF	FF
2020	10	10	10	10
2024	10	10	10	10
2028	10	10	10	10
202C	FF	FF	FF	FF
2030	FF	FF	FF	FF
2034	FF	FF	FF	FF
2038	FF	FF	FF	FF

a. (6 pts) Step through the first six instructions. For each instruction, indicate the *new* values of the registers that are modified by that instruction.

Instruction	**R1**	**R2**	**R3**	**R4**	**R14**	**R15**
`li    r14,0x1FFC`					0x1FFC	
`li    r15,4`						0x4
`lwzux r1,r14,r15`	0xC				0x2000	
`lwzux r2,r14,r15`		0x2010			0x2004	
`lwzux r3,r14,r15`			0x2020		0x2008	
`lwzux r4,r14,r15`				0x2030	0x200C	

b. (4 pts) What are the values in the following registers after the first iteration of the loop at loop1?

R5	**R6**	**R7**	**R8**
0x1	0xB	0x10	0x1B

c. (10 pts) What are the values in memory locations 2030-203B after loop1 has finished?

Memory Address	**+0**	**+1**	**+2**	**+3**
2030	0x1B	0x1A	0x19	0x18
2034	0x17	0x16	0x15	0x14
2038	0x13	0x12	0x11	0x10

d. (5 pts) What are the values in the following registers after the first iteration of loop3?

R6	**R7**	**R8**	**R9**	**R10**
0x01	0x0B	0x01	0x1B	0x1A

e. (15 pts) What are the values in the following registers after the code fragment finishes running?

R4	**R29**	**R30**	**R31**
0x2038	0x10111213	0x14151617	0x18191A1B

3. (45 pts) Here is an excerpt from the Unix man page for the C library function memcpy:

SYNOPSIS

void *memcpy(void *dest, const void *src, size_t n);

DESCRIPTION

The memcpy() function copies n bytes from memory area src to memory area dest. The memory areas may not overlap.

RETURN VALUE

The memcpy() function returns a pointer to dest.

a. (20 pts) Write a simple PowerPC assembly-language version of the memcpy function. Use the ABI conventions discussed in class. Focus on correctness rather than performance.

```
memcpy: mr     r7, r3       # save dest pointer for return value
loop:   cmpwi  r5, 0        # n == 0 ??
        beq    done         # yes... all done
        lbz    r6, 0(r4)    # load byte from *src
        stb    r6, 0(r3)    # store byte to *dest
        addi   r4, r4, 1    # src++
        addi   r3, r3, 1    # dest++
        addi   r5, r5, -1   # n--
        b      loop
done:   mr     r3, r7       # restore dest pointer for return value
        blr                 # return
```

Notes:

- Since memcpy doesn’t call any other functions (it’s a leaf function) there’s no need to allocate a stack frame.
- Putting the compare at the top of the loop makes memcpy act correctly when called with n = 0.
- There are a number of optimizations that could be used, such as using update-mode addressing to eliminate the pointer increments, or using “`addi. r5, r5, -1`” to eliminate the `cmpwi`. None of these were necessary for full credit.

b. (25 pts) Copying data using word accesses (loads and stores) obviously takes fewer instructions than copying the same amount of data with byte accesses. However, using word accesses is likely to be *faster* than using byte accesses *only if all of the accesses are aligned*. Assume that you have two functions:

1. bytecpy, which is identical to your program from part (a).
2. wordcpy, which has the same arguments and return value as bytecpy, except that:
   a. it copies data using word loads and stores
   b. it interprets its third parameter as a word count rather than a byte count (that is, it copies 'n' words rather than 'n' bytes).

Write another complete assembly-language version of the memcpy function. This version should not copy any data itself; instead, it should check its parameters and call wordcpy if appropriate and bytecpy otherwise.

It is only appropriate to call wordcpy if *both* the src and dest pointers are word-aligned *and* the byte count n is a multiple of four (that is, you're not copying a fractional number of words). Otherwise, call bytecpy.

- Since this function is not a leaf function, you *must* allocate a stack frame and save the link register on the stack.
- The pointer word-alignment checks and the byte count check are really the same operation: check if the value is a multiple of four. There are several ways to do this. Many people divided the value by four, multiplied the result by four, and compared that result to the original value. This works, but is extremely inefficient: both multiply and divide are relatively time-consuming instructions. It is much more efficient to test whether the low-order two bits are 00. You can do this in a single instruction with an `andi.` or an `rlwinm.`.

```
memcpy: stwu   r1, -8(r1)   # set up stack frame
        mflr   r0           # save LR
        stw    r0, 4(r1)

        andi.  r0, r3, 3    # is dest word-aligned? (result in r0 is unused)
        bne    do_bytecpy   # no... call bytecpy
        andi.  r0, r4, 3    # is src word-aligned?
        bne    do_bytecpy   # no... call bytecpy
        andi.  r0, r5, 3    # is n a multiple of 4?
        bne    do_bytecpy   # no... call bytecpy
        srawi  r5, r5, 2    # change byte count to word count (divide by 4)
        bl     wordcpy
        b done

do_bytecpy:
        bl     bytecpy

done:   lwz    r0, 4(r1)
        mtlr   r0
        addi   r1, r1, 8
        blr
```