Problem 1 (3 parts, 27 points) **Understanding Code**

Part 1A (9 points) What values are in registers $1 and $2 after this MIPS code fragment executes? Express your answers in hexadecimal and explicitly specify all digits of each register. Please mark only the answer in each box.

```
lui  $1, 0x1234
ori  $2, $0, 0xABCD
andi $2, $2, 0x0F0F
add  $1, $1, $2
```

$1: 0x12340B0D
$2: 0x00000B0D

Part 1B (9 points) Assuming *big* endian byte ordering, values are in registers $4 and $5 after this MIPS code fragment executes? Express your answers in hexadecimal and explicitly specify all digits of each register. Please mark only the answer in each box.

```
.data
Input: .word 0xA1B2C3D4
.text
       addi $3, $0, Input
       lb   $4, Input($0)
       lbu  $5, 2($3)
```

$4: 0xFFFFFFA1
$5: 0x000000C3

Part 1C (9 points) What does the following code fragment print?

```
int V[] = {1, 5, -7, 6, -9, 17, -20, 0, -3};
int j, i;
for(i=0; V[i] < 16; i++)
  {
    if (V[i] < 0) continue;
    printf("V[%d]: %d\n", i,V[i]);
    for (j=i+1; j<9; j++)
       {
         if (V[j] < V[i])
           {
              printf("Next lower value: V[%d]: %d\n", j, V[j]);
              break;
           }
       }
  }
```

```
V[0]: 1
Next lower value: V[2]: -7
V[1]: 5
Next lower value: V[2]: -7
V[3]: 6
Next lower value: V[4]: -9
```

Problem 2 (3 parts, 25 points) **Conditionals: Compound Predicates**

Part 2A (8 points) What are the values of the variables x, y, z, and w after the following C code fragment executes? Express your answers in decimal. Hint: remember how C implements compound predicates.

```
int x, y, z;
x = y = z = 33;
int w = 10;
if (((x = 44) || (y = (w < z))) && (z == 8))  // note the "="
  w = 77;
```

Variable:	Value:
x	**44**
y	**33**
z	**33**
w	**10**

Parts 2B&C: Consider the following C code fragment:

```
int y, Extras = 0, Sum = 0;
for (y=-2500; y<2500; y++)
  if ((y>=0) && (y<10) && (y&1))
    Sum = Sum + y;
  else
    Extras++;
printf("Sum: %d\n", Sum);
```

Part 2B (8 points) What does this code fragment print? **Sum: 25**

Part 2C (7 points) The following code is supposed to turn the if-then-else statement in the C code fragment above into an equivalent nested if-then-else statement which does not use a compound predicate (i.e., does not use the && and || operators). However, it is buggy. For which values of y does it behave differently than the original code? Give your answer as integer range(s) or series with precise bounds, not as a list of individual integers.

```
if (y>=0)
  if (y<10)
    if (y&1)
      Sum = Sum + y;
else
  Extras++;
```

Answer:

-2500 <= y < 0, 10 <= y < 2500

Problem 3 (15 points) **Implementing Compound Predicates**

Complete the following MIPS code below to implement the following if-then-else statement (note that this is a *different* if-then-else from the one in Problem 2C):

```
if (((y>=i) && (y&1)) || (Extras % Sum <= y))
    Sum = Sum + y;
  else
    Extras++;
```

Only translate the *predicate* of the if-then-else; the rest is provided for you below. Your code should branch to the Then and/or Else labels according to the predicate conditions. **Assume register $1, $2, $3 , and $4 hold the integers y, i, Sum, and Extras, respectively.** Use additional registers if necessary. There are more blank lines provided than you need.

Label	Instruction	Comment
	slt $5, $1, $2	**# is (y < i)? or (not (y>=i))**
	bne $5, $0, OR	**# if so, check the OR clause**
	andi $5, $1, 1	**# y&1**
	bne $5, $0, Then	**# if (y&1) do Then**
OR:	**div $4, $3**	**# Extras / Sum**
	mfhi $5	**# Extras % Sum**
	slt $5, $1, $5	**# is y < Extras % Sum?**
	bne $5, $0, Else	**# if so, branch to Else**
Then:	**add $3, $3, $1**	**# Sum = Sum + y;**
	j EndIf	**# jump to stuff after the if**
Else:	**addi $4, $4, 1**	**# Extras++;**
EndIf:	**…**	**# stuff after the if**

Problem 4 (2 parts, 35 points) **Loops in C and MIPS**

Part 4A (15 points) Given an array **`CardNums`** of 500 unsigned integers and an unsigned integer **`x`** that is smaller than 3900, write C code that uses a **do while** loop to count the number of elements of CardNums whose lower 12 bits match those of **`x`**. Be sure to add any necessary variable declarations and initializations.

```
unsigned x = 0x00000C35;  // given (for example)
unsigned CardNums[500] = {0xA1B2DC35, 0xFEA43678, 0x123ABC35, ...};
int      Count = 0, i = 0;
do {if (x == CardNums[i] & 0xFFF)
       Count++;
    i++;
    }while(i<500);
```

Part 4B (20 points) Write a MIPS code fragment that is equivalent to the C code above (it should use the **do while** control flow). *For maximum credit, include comments.* (Note: there are more blank lines provided than you need.)

Label	Instruction	Comment
	`.data`	
`Xaddr:`	`.word 0x...`	`# int x = 0x...`
`CardNums:`	`.word 0xA1B2DC35, 0xFEA43678, ...`	`# int CardNums[500]={...};`
	`.text`	
	`addi $1, $0, 0`	`# i = 0`
	`addi $2, $0, 0`	`# Count = 0`
	`lw   $3, Xaddr($0)`	`# Read in x`
`Loop:`	`lw   $4, CardNums($1)`	`# Read CardNums[i]`
	`andi $4, $4, 0xFFF`	`# CardNums[i] & 0xFFF`
	`bne  $3, $4, Skip`	`# if x != masked element,` `# skip increment of Count`
	`addi $2, $2, 1`	`# Count++`
`Skip:`	`addi $1, $1, 4`	`# i++`
	`slti $5, $1, 2000`	`# is i < 500?`
	`bne  $5, $0, Loop`	`# if so, keep looping`

Optimizations

If you have a comment on your solutions concerning inefficient implementation, one or more of these optimizations might apply.

Inefficient use of branch/jump:

	`bne/beq $1, $2, Label`			`beq/bne $1, $2, L2`
	`j L2`	→		
`Label:`				

	`bne/beq $1, $2, Label`	→	`Label:`	`instruction`
`Label:`	`instruction`			

	`J Label`	→	`Label:`	`instruction`
`Label:`	`instruction`			

	`bne/beq $1, $2, Label`			`beq/bne $1, $2, Skip`
	`J L2`	→		`instructions`
`Label:`	`instructions`		`Skip:`	`J L2`
	`J L2`			

	`beq $1, $2, Then`			`bne $1, $2, Else`
	`bne $1, $2, Else`	→	`Then:`	`instruction`
`Then:`	`instruction`			

	`sub $1, $2, $3`			`beq/bne $2, $3, Label`
	`beq/bne $1, $0, Label`	→		

Inefficient implementation of "$2 <= $1":

	`addi $9, $1, 1`			`slt $4, $1, $2`
	`slt  $4, $2, $9`	→		`beq/bne $4, $0, Label`
	`bne/beq  $4, $0, Label`			

	`beq $2, $1, Label`			`slt $4, $1, $2`
	`slt  $4, $2, $1`	→		`beq $4, $0, Label`
	`bne  $4, $0, Label`			

	`sub  $4, $2, $1`			`slt $4, $1, $2`
	`slti $4, $4, 1`	→		`beq $4, $0, Label`
	`bne  $4, $0, Label`			

Inefficient use of immediate:

	`addi $1, $0, 1`			`addi $2, $3, -1`
	`sub $2, $3, $1`	→		

	`addi $1, $0, 1`			`andi $2, $3, 1`
	`and $2, $3, $1`	→		

	`addi $1, $0, 2`			`andi $1, $2, 1`
	`div  $2, $1`	→		
	`mfhi $1`			