Laboratory 6: Prelab Exercise #1

Date ______________________ Section ______________________

Name ____

Integer Arithmetic in MIPS

During the prelab you will become familiar with integer arithmetic in MIPS. Since you have already used addition and subtraction in the previous labs, we will focus on issues related to the difference between signed and unsigned operations, and overflows. An *overflow* is a situation in which the result of an operation can not be represented using the assigned number of bits for that result.

Integer addition and subtraction

Unsigned integers are positive integers. With n bits for representation, the smallest unsigned is zero and the largest is $2^n - 1$. In a C/C++ program, integer variables which can not possibly be negative, should be declared as `unsigned int` (though you will see many programs where all integers are declared as plain `int`, i.e. signed integers)

Signed integers can be both positive and negative. With n bits, in 2's complement representation, the smallest integer is -2^{n-1} and the largest is $2^{n-1} - 1$

At the C/C++ programming level the difference between unsigned and signed integers is sometimes ignored. Note, however that declaring an integer variable that is always positive as `int` (when it should in fact be declared as `unsigned int`), reduces the range of that variable in half.

Ex 1:

```
// Assume that integers are 32 bit wide
{
unsigned int ex1;       // values for ex1 between 0 and 4,294,967,295
int ex2;    // values for ex2 between -2,147,483,648 and 2,147,483,647
}
```

If the variable `ex2` is always positive, then its possible values will be in the range 0 to 2,147,483,647, half of the range an unsigned integer has. ■

Q 1:

A negative number is represented as a bit pattern whose most significant bit is 1 (in 2's complement representation). The same bit pattern, when looked at as an unsigned integer, will be a large number. Fill out the following table (assume 8 bit number representation for simplicity)

Bit pattern	Value if bit pattern represents a signed integer	Value if bit pattern represents an unsigned integer
`00000000`		
`01111111`		
`10000000`		
`11111111`		

At the assembly language level the difference between signed and unsigned is more subtle. Most instructions that do arithmetic on signed numbers may overflow and the overflow will be signaled. Thus the trap handler[1] can take appropriate action to deal with the situation: ignoring an overflow may result in the computation of results that are completely wrong.

Instructions that do arithmetic on unsigned integers on the other hand, will not signal an overflow even if the operation itself yields a result which is not representable with the given number of bits for the result.

Q 2:

In what cases can the operation overflow? In the 'Addition' and 'Subtraction' columns of the following table use a 'Y' to indicate that the operation can overflow, and 'N' to indicate that the operation can not overflow.

Possible cases of overflow for integer addition and subtraction

Operand_1	Operand_2	Addition	Subtraction[a]
positive[b]	positive		
positive	negative[c]		
negative	positive		
negative	negative		
0	positive		
0	negative		
positive	0		
negative	0		

a. Operand_2 is subtracted from Operand_1
b. Strictly greater than 0
c. Strictly smaller than 0

1. More about trap handlers in Lab #7

Step 1

Write a program, called *lab6.1.c* which:

- declares two integer variables, `a` and `b`. The initial value for `a` and `b` is the maximum possible value for an integer (INT_MAX if you include the *limits.h* file)
- adds `a` and `b` and prints the result

Q 3:

Do you think the addition in *lab6.1.c* overflows? Explain.

Step 2

Compile and run the executable created from *lab6.1.c*.

`a`	`b`	Printed result

Q 4:

Is the result your program prints correct? Explain.

Most compilers generate code that does not signal integer overflows. An unsuspecting programmer may easily generate incorrect output without any error message to indicate it.

Step 3

Create the program *lab6.1.asm* based on the following description:

- in 'main' prompts the user to enter two integers; store them in **`$t0`** and **`$t1`**
- adds the two numbers using the native signed instruction; store the result in **`$t2`**
- prints the result

Run the program and fill out the following test plan. Wherever possible use numbers that would make the addition overflow. In the 'Overflows' column use a 'Y' to indicate the operation overflows and a 'N' to indicate it does not. In the 'Comment' column mark with a star those cases when an overflow error is actually reported.

Note: carefully choose the numbers you enter for test. The system function that reads an integer from the user (read_int) will truncate a very big number as to fit an integer register (32 bits). What you have in the register may be something you did not expect (try for example to enter a number like 8,589,934,593 and then look in the register where you store the number to see what it has actually been stored there).

Test plan for *lab6.1.asm* **(signed addition)**

Operand_1	Operand_2	Printed result	Overflows	Comment
+2000000000	+2000000000		Y	
+	-			
-	+			
-	-			

Step 4

Create the program *lab6.2.asm* based on the following description:

- in 'main' prompts the user to enter two integers; store them in **`$t0`** and **`$t1`**
- subtracts the second number from the first using the native signed instruction; store the result in **`$t2`**
- prints the result

Run the program and fill out the following test plan. Wherever possible use numbers that would make the subtraction overflow. In the 'Overflows' column use a 'Y' to indicate the operation overflows and a 'N' to indicate it does not. In the 'Comment' column mark with a star those cases when an overflow error is actually reported.

Test plan for *lab6.2.asm* **(signed subtraction)**

Operand_1	Operand_2	Printed result	Overflows	Comment
+	+			
+	-			
-	+			
-	-			

Step 5

Testing unsigned addition and subtraction requires some attention. The SPIM simulator always prints the content of a general purpose register as a *signed* integer. The fact that a large valid unsigned integer is printed as a negative number may confuse you. Unsigned numbers larger or equal to 2^{n-1} (but smaller than 2^n) have the most significant bit 1. Therefore they will be interpreted as negative numbers by the printing function.

Create the program *lab6.3.asm* based on the following description:

- in 'main' prompts the user to enter two integers; store them in **`$t0`** and **`$t1`**
- adds the two numbers using the native unsigned instruction; store the result in **`$t2`**
- prints the result

Run the program and fill out the following test plan. Wherever possible use numbers that would make the addition overflow. In the 'Overflows' column use a 'Y' to indicate the operation overflows and a 'N' to indicate it does not. In the 'Comment' column mark with a star those cases when an overflow error is actually reported.

Test plan for *lab6.3.asm* (unsigned addition)

Operand_1	Operand_2	Printed result	Overflows	Comment
+	+			

Step 6

Create the program *lab6.4.asm* based on the following description:

- in 'main' prompts the user to enter two integers; store them in **`$t0`** and **`$t1`**
- subtracts the second number from the first using the native unsigned instruction; store the result in **`$t2`**
- prints the result

Run the program and fill out the following test plan. Wherever possible use numbers that would make the subtraction overflow. In the 'Overflows' column use a 'Y' to indicate the operation overflows and a 'N' to indicate it does not. In the 'Comment' column mark with a star those cases when an overflow error is actually reported.

Test plan for *lab6.4.asm* (unsigned subtraction)

Operand_1	Operand_2	Printed result	Overflows	Comment
+	+			
0	+			