

HW2: Simple Multiplication in MIPS

This assignment will help you become more comfortable coding in MIPS including the use of comparison and branch/jump instructions. You should complete this program with your partner (see partners1 on Moodle).

Deliverables: Your program in a file named *partner1_partner2.asm*

NOTE: You should use good coding standard including comments in this program (if anything assembly language programs require more descriptive comments than other programs you've written).

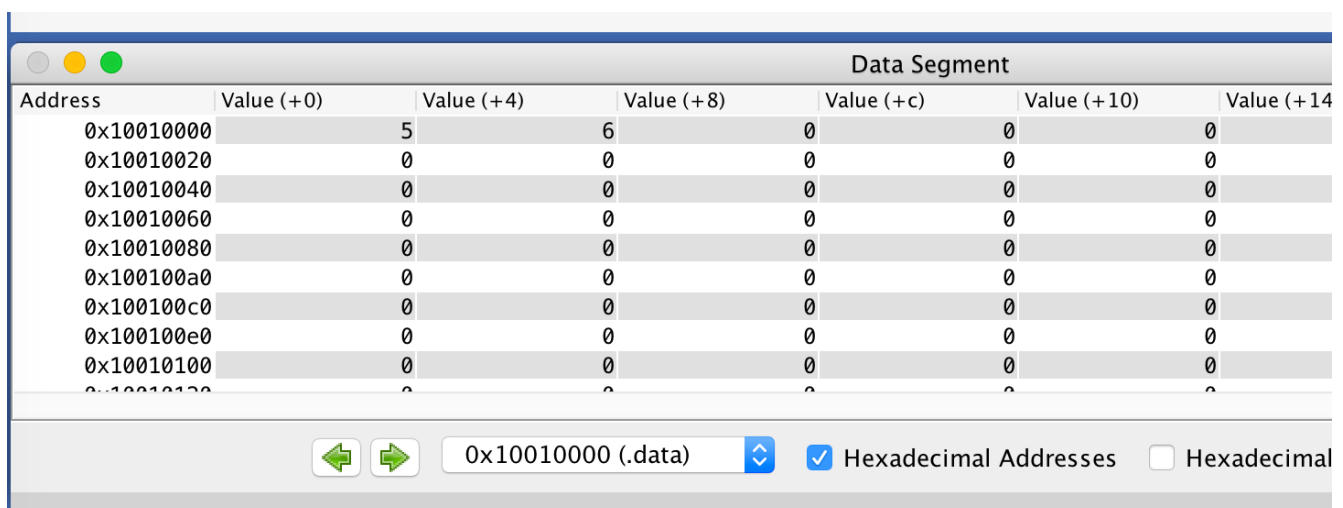
Your Task:

Write a program in MIPS to multiply 2 **signed** integers using **only** the following subset of MIPS instructions:

- any of the load and store instructions (e.g. *lw*, *sw*, *li*)
- any of the add and subtract instructions (e.g. *add*, *addi*, *sub*)
- any of the branch/jump instructions (e.g. *beq*, *bne*, *j*)
- the comparison instructions *slt* and *slti*

Note that you may NOT use instructions *mult* or *div*, or any others not listed above, rather you will use addition and loops to do the multiplication. You do not need to worry about overflow, but make sure to consider any other edge cases if necessary, your program should be able to multiply any 2 integers that don't result in overflow.

I have provided the initial program *start.asm*, you must use this to load the 2 numbers to be multiplied from the **given locations** in memory. You may not change the memory locations of the operands or the result. This program also includes an example of how to initialize memory to contain specific values as soon the program is built (assembled), so you don't have to edit the memory by hand before running the program. Try building *start.asm*, then look at the data segment in the execute screen – it should look something like this



Data Segment						
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)
0x10010000	5	6	0	0	0	0
0x10010020	0	0	0	0	0	0
0x10010040	0	0	0	0	0	0
0x10010060	0	0	0	0	0	0
0x10010080	0	0	0	0	0	0
0x100100a0	0	0	0	0	0	0
0x100100c0	0	0	0	0	0	0
0x100100e0	0	0	0	0	0	0
0x10010100	0	0	0	0	0	0
0x10010120	0	0	0	0	0	0

Navigation: Previous, Next, 0x10010000 (.data), Hexadecimal Addresses (checked), Hexadecimal

It will probably be easiest to test if you uncheck the hexadecimal values box so that you can see the values in memory as standard decimal integers. Note that the first 2 slots in memory now have the values 5 and 6. When the MIPS assembler sees the label *.data*, it stores the values listed after the label into sequential words starting at the beginning of the data section in memory (address 0x10010000). You can also use keywords to store other sizes of data, such as a single byte or a 64 bit value, but the default is the 32 bit word size. Now run the program, you should see an 11 appear in the 3rd slot in memory.

Replace the line that adds the 2 operand values with your multiplication algorithm. Start by creating a loop that can multiply 2 positive integers successfully, then add the ability to handle negative integers as well. Don't forget about the \$zero register – it is useful not only for comparisons but also for things like converting between negative and positive versions of a number ($0 - (-x) = x$). Test multiplying different numbers and be sure the correct result is appearing in the 3rd slot of the data section of memory. The grader will only look at that value, so getting the correct result into a register is not enough.

Save your program as partner1_partner2.asm to submit. Try to make the code as efficient as possible and don't forget to still include comments!

Advanced – Detect Overflow (optional, just for fun)

This task is completely optional, and just for your own learning/practice, it will earn no extra grade points.

Add the ability to detect overflow in your multiplication program. If you do find overflow, change the result of the multiplication to 0 and set the 4th value in memory to be all 1's (0xffffffff). There are many methods of detecting overflow in multiplication, most will require looking at instructions in MIPS that we have not discussed. Try to find any detection method that consistently works for both positive and negative operands, then try to find a more efficient method.