Assembly Language Problem Set

Test and expand your understanding of Assembly Language principles with these free-response questions.

Here are some tips to help get you through this problem set:

- These Assembly questions will use a specific Assembly Language, MIPS32, for each question, however, the concepts underlying each question are language agnostic.
- An integral skill in all software developer's toolkits is the ability to read source documentation and extract relevant information, with that said, it will be required to reference the MIPS32 documentation. There is:
    - a Quick Reference Guide, which will provide most of the reading for the problems
    - the MIPS Documentation (Introduction Manual)
    - an In-Depth Instruction Manual for a deeper dive into actual instruction processing.
- This is a cumulative knowledge worksheet, meaning you will draw on the combined knowledge of the Binary numbers, Instruction Set Architecture, and Assembly Language modules.

Proceed to the problems when you are ready!

Problem 1

Free response

Given the following instruction and register assignments, what will the values of V0 (the 'return value' register), LO (Special Accumulator Register), and HI (Special Accumulator Register) be?

Instruction:

MULTU A0, A1

Register	**Value (32-bit Binary)**
A0	00000000000000000000000000101001
A1	00000000000000000000000111100111
V0	Unassigned
LO	Unassigned
HI	Unassigned

Your response

V0 = Unnasigned LO = 20887 HI = 0

Our answer

The first thing to do is find the opcode in the documentation in order to determine what operation is being conducted. In this case, the MULTU opcode refers to the multiplication of two unsigned numbers, the values stored in registers A0 and A1.

Our second task is to multiply the contents of A0 and A1. In binary, the multiplication of two 32-bit numbers will result in a 64-bit answer, it is up to you if you wish to do binary or decimal division.

We converted the numbers back to decimal, so A0 becomes 41 and A1 becomes 487 so that we can better visualize the answer.

41 x 487 = 19967

This number is then converted back into binary, except using 64-bits. Since these are unsigned integers, all the extra leading bits of the 64-bit answer will be filled with 0's. $19967_{10} = 101000010111_{10}$, plus all the leading zeroes.

0000000000000000000000000000000000000000000000000100110111111111

Since the MIPS32 architecture only has 32-bit registers, the solution to this was the creation of two special-purpose registers, HI and LO. The high-order bits, the first 32-bits (left to right) of the result are stored in the HI register and the low-order bits, the last 32-bits of the result are stored in the LO register.

After this single MULTU instruction, the registers will now look like this. Note that no value has been stored in the return value register, V0 even though this is the result we want to return. That will require another instruction altogether to read the HI or LO registers. After our MULTU instruction, our registers now look like this:

Register	**Value (32-bit Binary)**
A0	00000000000000000000000000101001
A1	00000000000000000000000111100111
V0	Unassigned
LO	00000000000000000100110111111111
HI	00000000000000000000000000000000

Seemingly simple arithmetic becomes a multi-stage operation on most computers because of hardware limitations and there are some operations that can reduce the overall number of operations needed to perform simple math. For example, sometimes it is not necessary to call a standard multiplication instruction when you are sure the answer will not use one of the HI or LO registers.

Problem 2

Free response

Given the following instruction and register assignments, how would we complete this operation to ensure that the correct value is stored in V0 when we know that the value will not exceed 4,294,967,295 (the largest 32-bit unsigned number)?

Instruction:

MULTU A0, A1

Register	Value (32-bit Binary)
A0	00000000000000000000000000101001
A1	00000000000000000000000111100111
V0	Unassigned
LO	Unassigned
HI	Unassigned

Your response

V0 00000000000000000100110111111111 LO 00000000000000000100110111111111 HI
00000000000000000000000000000000

Our answer

There are two ways to approach this problem, one from a speed and optimization point of view and the other from a legacy code perspective.

Reading the documentation you will see that legacy implementation of unsigned multiplication is done with the MULTU function followed by a call to check the HI and LO accumulator register for the answer. Since we know that the answer will not exceed the LO register, we need to find an instruction that reads the LO register, that function is MFLO.

Our code now consists of two lines:

```
MULTU A0, A1
MFLO V0
```

If you read carefully, you will see that this legacy implementation also prevents us from performing another instruction that will affect the HI or LO registers within two instructions of the MULTU instruction due to the way MIPS is pipelined, which you will learn about in an upcoming module.

The second way to achieve the desired result is to rewrite the code to use the MULU instruction, which multiplies two registers together and stores the 32 low-order bits in the destination register, disregarding the 32 high-order bits altogether.

Our reimplementation of the multiplication is completed with this single line of code:

```
MULU V0, A0, A1
```

Since MIPS is typically used in embedded applications, with limited memory, optimizing code down to the instruction level is an important task that can't get overlooked.

Problem 3

Free response

What is the result of the following operation given the contents of the registers?

Instructions:

JR T2
NOP

Register	**Value (32-bit Binary)**
T0	01010101010101010101010101010101
T1	Unassigned
T2	00110001001010001111111111111111

Your response

Result: The result of the operation is that the program will jump to address 0x32AFFFFF and start executing instructions from there. The NOP instruction will not be executed because the jump occurs first. Thus, after the JR T2 instruction is executed, the program counter (PC) will be set to the value stored in T2, which is 0x32AFFFFF. The NOP instruction is effectively skipped, and control is transferred to the address 0x32AFFFFF.

Our answer

This problem requires an understanding of how the CPU handles control flow operations like jump and branch commands.

Since our hardware doesn't understand if-else statements, for loops, or iterators, one method it uses is to "jump" to a specific instruction. Most Assembly languages have a special register called the "Program Counter" or PC that contains the next instruction to process. We can manipulate the contents of the PC with Jump and Branch commands.

This particular command, JR, sets the PC equal to the contents of what is stored in the designated register, T2.

- 00110001001010001111111111111111 is the 32-bit instruction in T2

From our ISA lesson, we know that the first six bits are the opcode. We can search our Instruction Manual for the string 001100 and find that the match operation is the ANDI (And Immediate) instruction. This command takes a register and an immediate value as operands and stores the result into the destination register.

An issue you may have run across was that the registers in the binary ANDI instruction are just numbers and the table we gave you references registers by their mnemonic names. To solve this problem you need to look at the Registers Table in the Quick Reference Guide to translate that register 9 is the T1 register and register 8 is the T0 register.

Once we have the two numbers:

- 01010101010101010101010101010101 from T0
- 00000000000000001111111111111111 as our zero-extended immediate

We can draw on our knowledge of logic gates to perform the bitwise AND of the two numbers.

The final registers, including the result in T1, is below:

Register	**Value (32-bit Binary)**
T0	01010101010101010101010101010101
T1	00000000000000000101010101010101
T2	00110001001010001111111111111111

As in most Assembly languages, a jump command normally also causes the next line of code to operate as the next command is being loaded into the PC. It is common practice to follow a jump or branch command with a NOP or No Operation command; in fact, if you read the Programming Notes associated with the NOP, you will see it referenced. It does exactly that- performs no operations for one cycle. The actual execution of code would go: JR -> NOP -> ANDI.

Problem 4

Free response

What is the result of the following code:

Instruction:

XOR T1, T1, T1

Register	**Value (32-bit Binary)**
T1	11011000111000111010100110001101

Your response

T1 = 00000000000000000000000000000000

Our answer

This instruction sets the register to zero.

The goal of the XOR, or Exclusive OR, is to return true if one, and only one, of the inputs to the gate is true.

Therefore if we XOR a value and itself it will always result in zero because it can’t be exclusive with itself. If we take a look at the truth table in the hint you can see this as well.

This method is a common way to zero out a register across all Assembly languages, particularly because it can be done in one cycle. MIPS has a dedicated hardwired zero register that can also be used but don’t be surprised if you see XOR being used just out of habit.

Register management is an important task at the hardware level and as with all things, errors can occur when overwriting data. Zeroing a register before storing a value to it ensures that the desired data is in fact stored there.