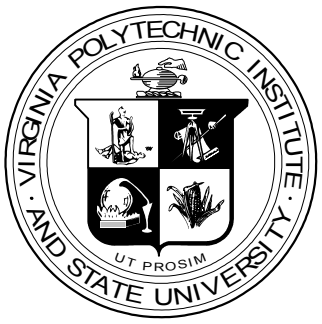


Instructions:

- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page formula sheet and the MIPS reference card. No calculators or other computing devices may be used.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 7 questions, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- Note that either failing to return this test, or discussing its content with a student who has not taken it is a violation of the Honor Code.

Do not start the test until instructed to do so!

Name _____
printed

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

_____
signed

1. Given the MIPS source shown at the left below, a MIPS assembler (MARS) replaces the single `la` instruction with the sequence of instructions shown at the right below:

```
list:   .word   2, 3, 5, 7
. . .
        la      $a1, list
```

```
        lui     $at, 4097
        ori     $a1, $at, 4
```

   a) [10 points] Why may it be logically necessary for the assembler to replace the pseudo-instruction on the left with a sequence of different instructions? Hint: consider the fact the end result must be MIPS machine code.

The value of the label `list` is an address, which may require 32 bits. Since MIPS machine instructions are limited to 32 bits, there is no way to fit the address into a single machine instruction along with the other necessary fields (opcode and register specifier).

Therefore, the assembler must generate an alternative representation that places the address of the label into a register so that a 5-bit register specifier can be used instead of an immediate address.

   b) [5 points] How did the need to support pseudo-instructions influence the conventions regarding the use of the 32 general-purpose registers by developers writing MIPS assembly code? Give a specific example.

If the assembler is to perform transformations like that shown above, it will often need to use registers in order to store the results of intermediate computations. That is explicit in the handling of the `la` instruction above, where the assembler uses the register `$at` to store the intermediate form of the address that eventually is placed into the register `$a1`.

The issue is that the assembler must also not overwrite the contents of a register that is being used by the programmer. The solution is to adopt the convention that certain registers are reserved for use by the system. In MIPS, the register `$at` is reserved for the assembler.

---

2. [10 points] One of the design principles that guided the development of the MIPS platform was that "simplicity favors regularity". In other words, in order to promote simplicity in a system, the designer should look for opportunities to use the same design patterns over and over. Give an example where this principle was applied in the design of MIPS assembly language (not machine language).

One of the best illustrations is the stipulation that all arithmetic-logical instructions follow the pattern:

<operation mnemonic> <dest register>, <left operand>, <right operand>

3. [6 points] Recall that the basic components of a computer system are frequently connected by one or more hardware bus structures, consisting of collections of one or more wires. The number of wires used in a bus is often called the *width* of the bus. Make a conjecture regarding the width of the bus that would connect memory to the CPU in a MIPS system, and justify your conjecture by referring to a property of MIPS language.

Since each MIPS instruction is 32 bits wide, it would make sense for the efficiency of the fetch-execute cycle for the CPU-memory bus to also be 32 bits wide so an instruction can be fetched in a single transfer.

---

4. Consider the following MIPS assembly code fragment:

```
[0x00400040]  loop:  bgez  $t0, done    #1
[0x00400044]                            #2
                . . .
[0x00400080]  done:  li    $t1, 1000    #3
```

a) [5 points] What is the value of the program counter register when the system begins to execute the instruction in line #1?

The PC stores the address of the currently-executing instruction, at least until something causes the PC to be updated: `0x00400040`

b) [10 points] When the assembler translates the instruction in line #1, it will replace it with an instruction of the form:

```
bgez  $t0, <integer>
```

What value, in base-10, will the assembler insert for the field `<integer>`? Why?

The value would be the offset from the current instruction (see part a) to the address of the label `done`. The only subtleties are that in MIPS, the offset is expressed in words rather than in bytes and in base-10.

The distance in bytes is 0x40, which would be 64 in base-10. Since the word size is 4 bytes, this implies an offset field of 16.

c) [6 points] Explain how the program counter is updated when the instruction shown in line #3 is executed:.

The instruction in #3 is not a branch or jump, so the PC would just be set to the address of the next instruction, which would be at the address of #3 plus 4.

5. Assume the following data segment of a MIPS assembly program:

```
Size:   .word   10
List:   .word   2, 3, 5, 7, 9, 11, 13, 17, 19, 23
```

a) [8 points] Write a sequence of MIPS assembly instructions to transfer the fourth word of the array `List` into register `$s0`.

There are a number of ways to accomplish this. Here are a couple:

```
#1:
        lw      $s0, List + 12  # note that the fourth word would be 12 bytes
                                #    from the beginning of the array

#2:
        la      $t0, List
        lw      $s0, 12($t0)
```

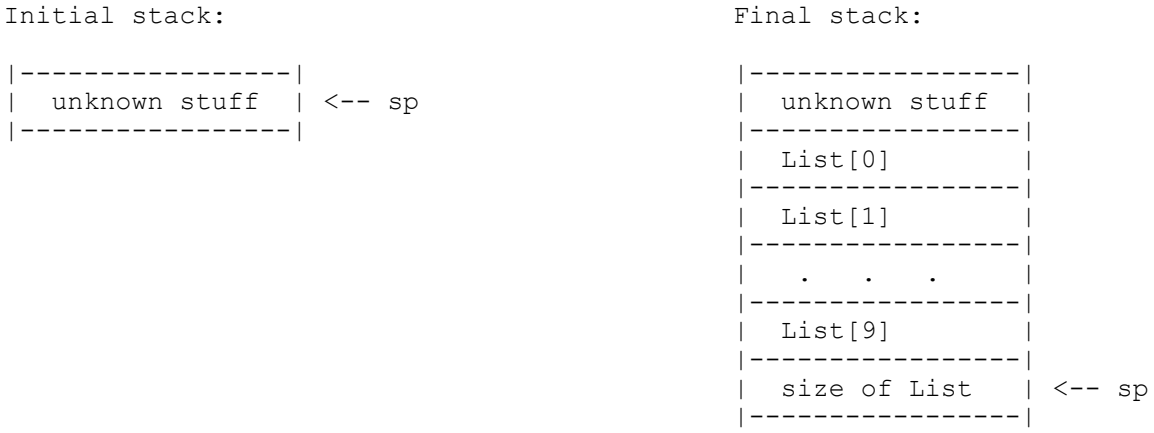
b) [6 points] Write a sequence of MIPS assembly instructions to put the square of the fourth word of the array `List` into register `$s1`. You may assume your answer to part a has been executed.

This was so easy it's hard:

```
        mul     $s1, $s0, $s0     # this will NOT report an overflow

        mulo    $s1, $s0, $s0     # this WILL report an overflow
```

6. [16 points] The author of a MIPS assembly program needs to push the elements of the array `List` (from the previous question) onto the stack, and the push the size of the array onto the stack. The initial and resulting states of the stack are shown below:



Write MIPS assembly instructions to modify the stack as described above.

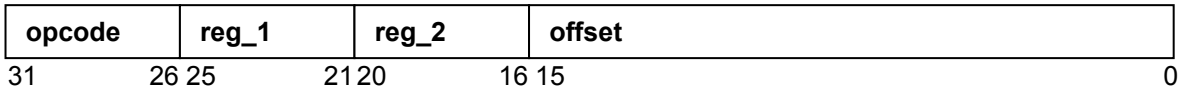
Here's one solution:

```
        lw    $t1, Size              # get size of list
        la    $t2, List              # get address of 1st element

  copy: blt   $zero, $t1, done
        lw    $t3, ($t2)             # load next list element
        addi  $sp, $sp, -4           # make room on stack for element
        sw    $t3, ($sp)             # push element onto stack
        addi  $t2, $t2, 4            # step to next list element
        addi  $t1, $t1, -1           # count this element
        j     copy


  done: lw    $t1, Size              # reload list size
        addi  $sp, $sp, -4           # make room on stack for size
        sw    $t1, ($sp)             # push size onto stack
```

7. In MIPS machine language, the I-format instructions, such as `lw`, have the following form:



a) [5 points] How many different I-format instructions could there be? Why?

There are 6 bits in the opcode field, so there would be 2^6 different possible opcodes. Of course, there are also going to be R- and J-format instructions, so not all of the 64 possible patterns would correspond to I-format instructions.

Nevertheless, 64 is certainly an upper bound on the number of I-format instructions.

b) [5 points] For the `beq` instruction in particular, the value in the **offset** field is used to compute the address of the next instruction to be fetched. What is the range of values of the offset? Why?

There are 16 bits for the offset, but it's represented as a two's complement (signed) integer, so the range for the offset is:

[-2^15, 2^15 - 1]

c) [8 points] I-format instructions are so-called because they include an *immediate* field, that is, a field that stores a literal (constant) value is encoded directly into the instruction. Does having such instructions speed up the execution of programs? Why or why not?

Yes. If we did not have immediate instructions, then the immediates would have to be stored in memory separately from the instructions and loaded into registers (as separate instructions) at runtime.

Extra instructions may still be necessary in some situations (see question 1), but if the immediate value is guaranteed to fit into the instruction field (as is the case with branch instructions) then we will save at least one machine instruction at runtime.