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. [12 points] The native MIPS assembly language does not support an addressing mode in which a register is used to specify an offset to be applied relative to another register. The following hypothetical pseudo-instruction does so:

```
lw   $sr, $of($ba)   # load the word at the memory address $ba + $of
```

Show how an assembler might replace the pseudo-instruction above with a sequence of one or more native instructions to achieve the same effect. Include comments to explain your logic, and be sure to observe any relevant register conventions.

Here's one straight-forward solution:

```
addu    $at, $of, $ba   # add registers, use assembler temp register
lw      $sr, 0($at)     # use native-mode lw instruction for transfer
```

---

2. [12 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. [10 points] Explain briefly but clearly how 2's complement representation differs from sign-magnitude representation.

Both schemes represent non-negative numbers in exactly the same way, a sign bit (of zero) followed by the base-2 representation of the number itself.

For a negative number, say –x (where x is positive):

- **sign magnitude representation is a sign bit (of one) followed by the base-2 representation of the number itself**
- **2's complement representation is the bit-wise complement of x plus 1**

---

4. Consider the following MIPS assembly code fragment:

```
[0x00400100]  loop:  bgez  $t0, done    #1
[0x00400104]                            #2
                . . .
[0x00400180]  done:  li    $t1, 1000    #3
```

a) [4 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 current instruction: `0x00400100`

b) [8 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 is relative to the current address, so it needs to be the distance to the instruction that is the target of the jump, which is at address `0x00400180`.

However, the PC will have been incremented by 4 before the instruction has been decoded, so the value would actually correspond to the distance from the next instruction to the target, which would be `0x0000007C`; in base-10 this would be 7x16 + 12 or 124.

But, the distance is expressed in words, not bytes, so it would be 31.

c) [8 points] Describe carefully how the program counter is updated when the instruction shown in line #1 is executed:.

First the PC is "optimistically" incremented by 4, yielding the address of the next instruction in memory.

If the condition of the `bgez` instruction is true, the relative offset given in that instruction is shifted 2 bits to the left (i.e., multipled by 4) and then added to the PC.

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

There are a number of solutions. Here are four:

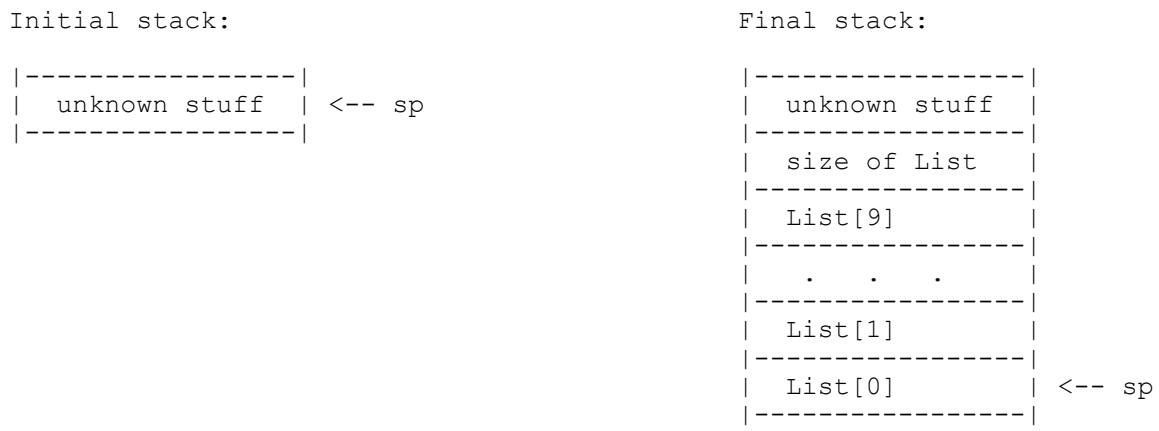
```
soln1:  la    $t0, List
        lw    $s0, 24($t0)

soln2:  la    $t0, List
        li    $t1, 6
        sll   $t1, $t1, 2
        add   $t1, $t1, $t0
        lw    $s1, ($t1)

soln3:  lw    $s2, List + 24

soln4:  li    $t1, 24
        lw    $s3, List($t1)
```

b) [10 points] Write a sequence of MIPS assembly instructions to replace each element of the array `List` with its square. Your solution should be general in the sense that it should not depend on the specific values shown in the data segment given above. Comment your code to explain the logical significance of each statement.

Here's one solution:

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

        .text
        la    $t0, List         # get a pointer into the array
        lw    $t1, Size         # get upper limit for loop iterations
        li    $t2, 0            # initialize loop counter

loop:   bge   $t2, $t1, exit    # check iteration count
        lw    $s0, ($t0)        # load current value from list
        mul   $s0, $s0, $s0     # square current value
        sw    $s0, ($t0)        # write square back to list
        addi  $t0, $t0, 4       # move pointer to next element
        addi  $t2, $t2, 1       # increment loop counter
        b     loop              # restart loop

exit:   li    $v0, 10           # exit program
        syscall
```

6. 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:



[16 points] Write MIPS assembly instructions to modify the stack exactly as described above. You must use a loop in your solution.

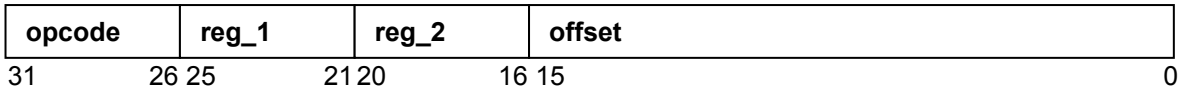
Here's one solution:

```
        lw    $t1, Size          # get size of list
        addi  $sp, $sp, -4       # make room on stack
        sw    $t0, ($sp)         # put size on stack

        la    $t0, List          # get pointer to first list element
        li    $t3, 4
        mul   $t3, $t1, $t3      # compute offset to end of list
        addi  $t3, $t3, -4
        add   $t0, $t0, $t3      # now $t0 points to last elem of list

push:   ble   $t1, $zero, done   # exit loop after last list value is pushed
        lw    $t2, ($t0)         # load current list value
        addi  $sp, $sp, -4       # make room on stack
        sw    $t2, ($sp)         # push current list value on stack
        addi  $t0, $t0, -4       # step pointer to previous list element
        addi  $t1, $t1, -1       # decrement push counter
        b     push               # go back to loop test

done:
```

[4 points] The stack operations above are performed in order to prepare to make a call to a MIPS procedure, which will do something with the values in the array. Is there any with the manner in which the values have been organized on the stack? If yes, explain.

There is a problem. In order to find the size of the list, the called procedure must know where it is on the stack. But, the size is stored "below" the list elements themselves, so there is no obvious way for the called procedure to retrieve the size from the stack without knowing what the size is.

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



a) [6 points] In theory, what is the maximum number of possible I-format instructions? Why?

The opcode field is 6 bits wide, so there are 2^6 different patterns. However, one of those is used as for R-format instructions and a small number must be reserved for J-format instructions.

b) [6 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. Assuming the values in the two registers are equal, how is the offset used in determining the next instruction?

The offset field is added to the value of the program counter register, which will already have been incremented by four (but that is taken into account when the value of the offset is computed by the assembler).