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 6 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 Solution
printed

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

signed

1. [16 points] The native MIPS assembly language does not include a `nand` (not-and) instruction:

```
nand  $rd, $rs, $rt     # $rd <-- !($rs && $rt) in bitwise fashion
```

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 the logic of your design, and be sure to observe any relevant register conventions.

```
# Native logical instr:
and  $rd, $rs, $rt
nor  $rd, $rs, $rt
or   $rd, $rs, $rt
xor  $rd, $rs, $rt
```

Here's one solution (there are a number of others):

```
and  $at, $rs, $rt      # form $rs & $rt
nor  $rd, $at, $at      # x nor x == ~(x or x)
                        #         == ~x and ~x
                        #         == ~x
```

Note that no solution should modify any registers except `$rd` and, possibly `$at`.

Some alternate solutions can be based on the following facts:

- **A xor 1 == ~A (bitwise) and so ($rs and $rt) xor 0xFFFFFFFF == $rs nand $rt**
- **A nor 0 == ~A (bitwise) and so ($rs and $rt) nor $zero == $rs nand $rt**
- **$rs nand $rt == ~($rs and $rt) == (~$rs) or (~$rt) == ($rs nor $rs) or ($rt nor $rt)**

---

2. [16 points] 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
```

Write MIPS assembly instructions to transfer the fourth word of the array `List` (i.e., 7) into register `$s0`.

There are a number of solutions. Here are four:

```
soln1:  la    $t0, List
        lw    $s0, 12($t0)


soln2:  la    $t0, List
        li    $t1, 3
        sll   $t1, $t1, 2
        add   $t1, $t1, $t0
        lw    $s0, ($t1)


soln3:  lw    $s0, List + 12


soln4:  li    $t1, 12
        lw    $s0, List($t1)
```

3. Consider the following MIPS assembly code fragment:

```
Adddress      Assembly Instruction
-----------------------------------------------
[0x00400100]  loop:  blez  $t0, done       #1
[0x00400104]         addi  $t0, $t0, -4    #2
                     . . .
[0x0040017C]         b      loop           #3
[0x00400180]  done:  li    $t1, 1000      #4
```

a) [2 points] What is the value of the program counter register when the system begins to execute the instruction in line #1?

At the beginning of execution, PC = `0x00400100`. While the opcode is being decoded, the PC will be incremented by 4.

b) [8 points] Describe carefully what possible values the program counter may have after the instruction shown in line #1 is executed, and why.

```
if $t0 <= 0 then
     PC <-- 0x00400180
else
     PC <-- 0x00400104
```

c) [6 points] If the instruction in line #1 is executed and the value in `$t0` is 0, what value must be added to the program counter? Give your answer in hexadecimal.

By the time the registers have been compared, the PC will already have been incremented by 4. So, if $t0 == 0 then the value added to the PC must equal

```
0x00400180 – 0x00400104 = 0x7C
```

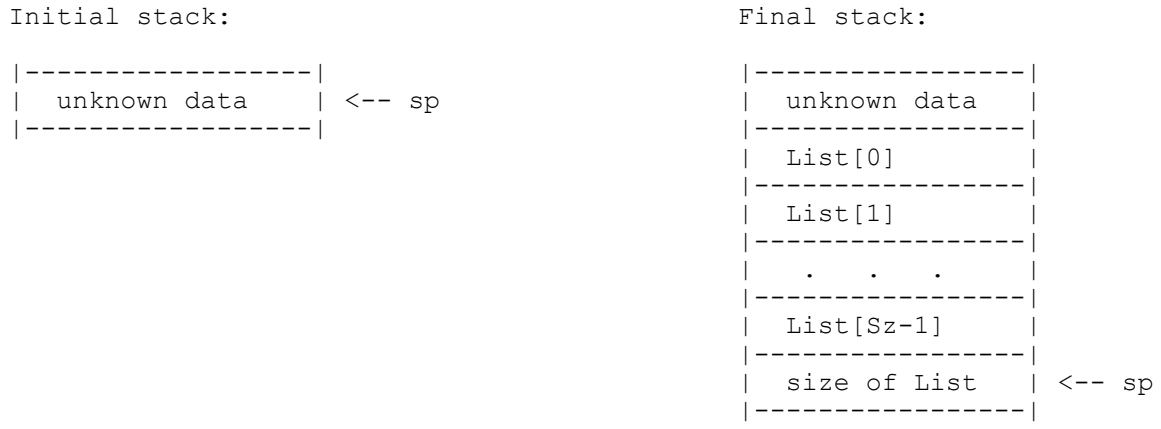
4. [20 points] Complete the following MIPS assembly procedure, which shifts elements of the array parameter to delete the specified element from an array. Your solution should not make use of the stack. Comment your code to explain the logical significance of each statement.

```
#############################################################################
# Pre:  $a0 points to the array in question
#       $a1 points to the size of the array
#       $a2 specifies the C-style index of the element to be deleted
#
# Post: $a0 is unchanged
#       $a1 is unchanged
#       $a2 the target of $a2 has been decremented
#       The element of the array at the specified index has been deleted
#       $v0 == 1 if deletion is successful; $v0 == 0 otherwise
#
arraydelete:
        # check whether the index is in bounds
        blt  $a2, $zero, fail
        lw   $t0, 0($a1)      # $t0 stores the size of the array
        blt  $a2, $t0, del    # see if target index is in range
fail:   li   $v0, 0           # if not, set fail code and return
        j    exit

        # iterate from $a2 to next-to-last element, shifting as we go
del:
        addi $t0, $t0, -1     # calculate terminal index for traversal
        move $t1, $a2         # $t1 tracks current leading index
        beq  $t1, $t0, decr   # if last elem, no need to shift
        move $t3, $a0         # $t3 points to current leading elem
        move $t4, $t1
        sll  $t4, $t4, 2
        add  $t3, $t3, $t4    # $t3 points to arr[$a2]

loop:   lw   $t4, 4($t3)      # grab next element to shift
        sw   $t4, 0($t3)      # store it in previous array cell

        addi $t3, $t3, 4      # step pointer forward
        addi $t1, $t1, 1      # count elem just shifted
        blt  $t1, $t0, loop

decr:   sw   $t0, 0($a1)      # decrement array size

        li   $v0, 1









exit:   jr   $ra
```

Note that if the element to be deleted is at the end of the array then no shifting takes place; if you don't pre-test for this (as shown above) then you'll probably access a memory location past the end of the array.

5. [16 points] The author of a MIPS assembly program needs to push the elements of the array `Stuff` and the size of the array (from the data segment given below) onto the stack. The initial and resulting states of the stack are shown below:

```
        .data
Sz:     .word  <value not specified>
Stuff:  .word  <list of Sz integer values>
```

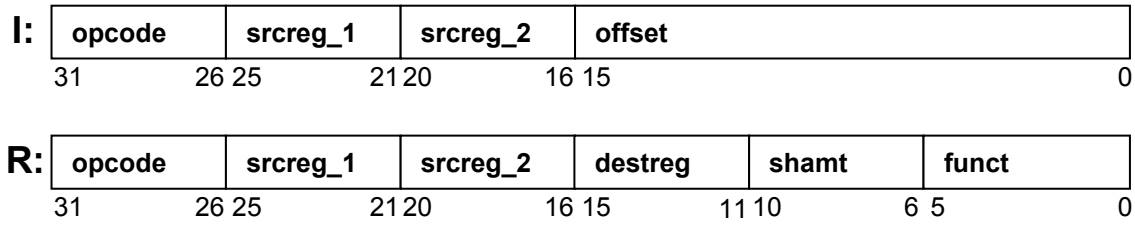


Write MIPS assembly instructions to modify the stack exactly as described above. Your solution should be general in the sense that it should not depend on any specific values for `Sz` or the list elements. Comment your code to explain the logical significance of each statement.

```
        lw    $t0, Sz              # get array size
        li    $t1, 0               # set loop counter
        la    $s0, Stuff           # get pointer to first array elem

loop:   beq   $t1, $t0, endloop    # exit loop when reach final elem
        lw    $s1, 0($s0)          # load current array elem
        addi  $sp, $sp, -4         # alloc room on stack for current elem
        sw    $s1, 0($sp)          # push current array elem to stack
        addi  $s0, $s0, 4          # move pointer to next array elem
        addi  $t1, $t1, 1          # count current elem
        b     loop                 # restart loop
endloop:

        addi  $sp, $sp, -4         # alloc room on stack for array size
        sw    $t0, 0($sp)          # push array size to stack
```

Note that it is logically necessary to perform a test before entering the loop, in case the size of the array is zero.

6. In MIPS machine language, the I-format and R-format instructions have the following formats:



a) [8 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. Discuss how this principle is illustrated by the MIPS machine language formats above.

- **both formats place the opcode in bits 31:26**
- **both formats place the two source registers into bits 25:21 and 20:16**

b) [8 points] Discuss how the fact that the **offset** field in the definition of the I-format is 16 bits wide imposes a compromise on the implementation of the conditional branch instructions.

The offset is limited to the approximate range $\pm 2^{15}$, which limits the distance that a conditional branch can "jump".

The limitation can be mitigated somewhat by making the "jump" be relative to the current value of the PC and interpreting it as the number of words to jump rather than the number of bytes, raising the relative distance that can be "jumped".