Lab 1 Report

Adrian Saldana & Noor Fakih

**Assignment/Problem Description:**

Lab 1 required us to complete an assembly version of a mergesort algorithm, originally written in C. The goals of the assignment was to become familiar with the MIPS instruction set, familiarizing ourselves with SPIM, test our understanding of pointer implementations, branches and procedure call conventions, proper convention and register usage, and tradeoffs we make for ease or efficiency.

**Discussion**:

*Solution/Program Description*

Mergesort is a recursive algorithm that takes a user-inputted array of numbers and returns the array sorted in increasing order.

Mergesort verifies the array has at least two elements, calculates the midpoint of the element arrays and calls itself twice; once to pass the arrays but with size being capped to the midpoint- essentially passing on the left half of the arrays and another with the arrays being halfway through - essentially passing on the right half the arrays. Then merge is called. Merge begins to fill in temp\_array with sorted elements by traversing the split array with a while loop and comparing elements from each half. Once it finishes that, it calls cpyarry to transfer the sorted elements in temp\_array into array.

*Any Implementation Issues*

Due to our unfamiliarity with MIPS, it was initially difficult to properly implement all the conditional statements and ensure proper logic flow. It made merge a more challenging implementation than the rest of the assignment.

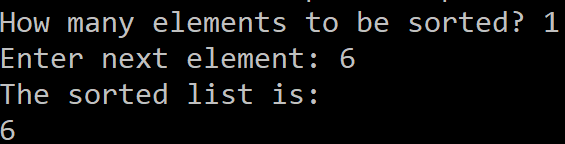
*Known Bugs and/or Errors*

Since we were given a skeleton mergesort.s as starter code for the lab, there was already a procedure for only accepting proper user-inputted data. Our implementation of the assignment does not have bugs, but we are aware that if an extreme amount of elements are passed in, the code would break.

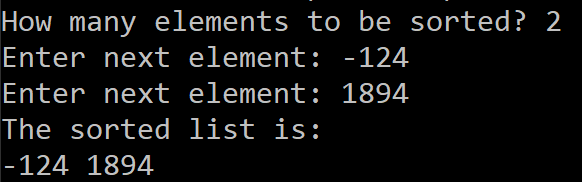
**Test Description and Results**

Below are some of the tests we ran and their results confirming correct, expected output. We looked to cover as many possible variations and cases as possible to root out any hiding issues such as single element, two elements, multiple elements, even amount of elements, odd amount of elements, large number of elements, elements already in order, etc.

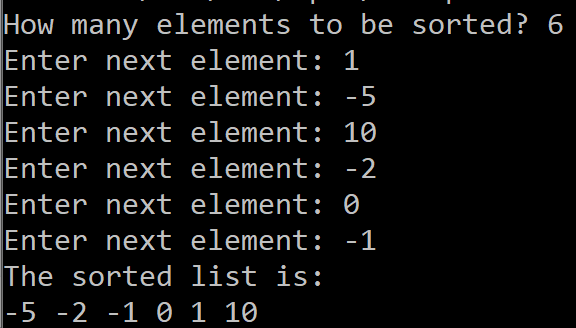
Single Element array



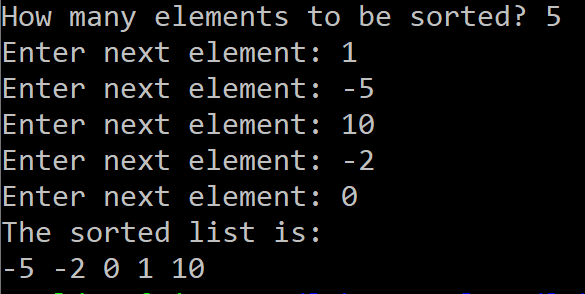
Two Element Array



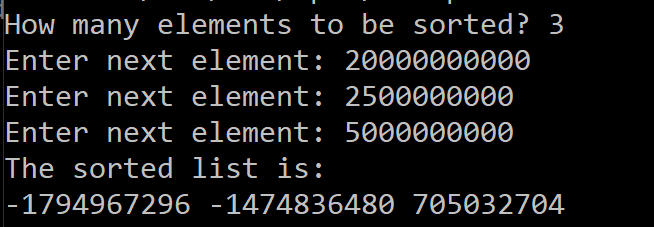
Even Element Array



Odd Element Array



Very Large integers - Checking overflow - Verified with test.sh



**Lessons Learned/Epilogue:**

We completed Lab 1 with confidence that we have achieved the learning objectives of the assignment. We built a better understanding of how the stack operates, memory allocation, and proper register usage. It also served as a good reintroduction to using assembly language and using branches and implementing conditional statements.

For the next assignment we will be able to start debugging as we go since we are now familiar with SPIM. We will also be able to get started on future problems with a better understanding of how to implement solutions in assembly so we will not have as many mistakes with conventions and misunderstandings.

This exercise would have been easier to work through if there were resources available with a clearer written explanation/diagram of how to implement conditionals and looping but stepping through the code with the debugger did allow us to figure it out and correct our mistakes. It would also be nice if there was a handout available that had things like the relevant appendix in the course book to use as a handy reference. We would also appreciate it if the lab handouts could contain guidelines/expectations for lab reports to help alleviate confusion. The lab hand out should also contain a tutorial on how to use the SPIM debugger.

| #============================================================================== # File: mergesort.s (Lab 1) #  # EE 180, Winter 2022 # Lab 1 # Adrian Saldana, asaldana@stanford.edu # Noor Fakih, nfakih@stanford.edu  # Due: January 18th, 2022 #  # Description: Assembly coded version of a mergesort algorithm. Mergesort is a # recursive algorithm that takes a user-inputted array of numbers # and returns the array sorted in increasing order. #  # Functions: mergesort(int \*array, int array\_size, int \*temp\_array) # merge (int \*array, int array\_size, int \*temp\_array, int mid) # arrcpy (int \*dst , int \*src , int array\_size) # #==============================================================================  .data HOW\_MANY: .asciiz "How many elements to be sorted? " ENTER\_ELEM: .asciiz "Enter next element: " ANS: .asciiz "The sorted list is:\n" SPACE: .asciiz " " EOL: .asciiz "\n"  .text .globl main  #========================================================================== main: # Description: receives user-inputted set of numbers, calls mergesort # to return a number set in order and prints the result.   #----------------------------------------------------------  # Register Definitions  #----------------------------------------------------------  # $s0 - pointer to the first element of the array  # $s1 - number of elements in the array  # $s2 - number of bytes in the array  #----------------------------------------------------------    #---- Store the old values into stack ---------------------  addiu $sp, $sp, -32  sw $ra, 28($sp)   #---- Prompt user for array size --------------------------  li $v0, 4 # print\_string  la $a0, HOW\_MANY # "How many elements to be sorted? "  syscall   li $v0, 5 # read\_int  syscall   move $s1, $v0 # save number of elements   #---- create dynamic array --------------------------------  li $v0, 9 # sbrk  sll $s2, $s1, 2 # number of bytes needed  move $a0, $s2 # set up the argument for sbrk  syscall  move $s0, $v0 # the addr of allocated memory    #---- Prompt user for array elements ----------------------  addu $t1, $s0, $s2 # address of end of the array  move $t0, $s0 # address of the current element  j read\_loop\_cond  read\_loop:  li $v0, 4 # print\_string  la $a0, ENTER\_ELEM # text to be displayed  syscall  li $v0, 5 # read\_int  syscall  sw $v0, 0($t0)   addiu $t0, $t0, 4   .globl read\_loop\_cond read\_loop\_cond:  bne $t0, $t1, read\_loop    #---- Call Mergesort ---------------------------------------    # Creating dynamic array - temp\_array  li $v0, 9 # sbrk  sll $s2, $s1, 2 # number of bytes needed  move $a0, $s2 # set up the argument for sbrk  syscall  move $s3, $v0 # the addr of allocated memory    # Passing Parameters  move $a0, $s0 # \*array  move $a1, $s1 # array size  move $a2, $s3 # temp\_array  jal mergesort   #---- Print sorted array -----------------------------------  li $v0, 4 # print\_string  la $a0, ANS # "The sorted list is:\n"  syscall   #---- For loop to print array elements ----------------------    #---- Initializing variables ----------------------------------  move $t0, $s0 # address of start of the array  addu $t1, $s0, $s2 # address of end of the array  j print\_loop\_cond  print\_loop:  li $v0, 1 # print\_integer  lw $a0, 0($t0) # array[i]  syscall  li $v0, 4 # print\_string  la $a0, SPACE # print a space  syscall   addiu $t0, $t0, 4 # increment array pointer  print\_loop\_cond:  bne $t0, $t1, print\_loop   li $v0, 4 # print\_string  la $a0, EOL # "\n"  syscall    #---- Exit -------------------------------------------------  lw $ra, 28($sp)  addiu $sp, $sp, 3  jr $ra   #---- Mergesort --------------------------------------- # Description: a recursive function which calls itself with parts # of the user-inputted set of elements and calls # merge to sort the elements  .globl mergesort mergesort:  #allocate memory  addi $sp, $sp, -28 #Make room for 7 items  sw $ra, 0($sp)  sw $t0, 4($sp)  sw $t1, 8($sp)  sw $t2, 12($sp)  sw $t3, 16($sp)  sw $a1, 20($sp)  sw $a0, 24($sp)    #----------------------------------------------------------  # Register Definitions  #----------------------------------------------------------  # $a0: \*array  # $a1: n = array\_size  # $a2: \*temp\_array  # $t0: if n < 2  # $t1: mid = n/2  # $t2: array + mid  # $t3: n - mid  #----------------------------------------------------------  # if (n < 2) then return  slt $t0, $a1, 2  bne $t0, $0, labelj # jump to return statement if less than 2  # int mid = n/2  move $t1, $a1  srl $t1, $t1, 1 # shift right once = division by 2    # mergesort(array, mid, temp\_array)  move $a1, $t1 #Updating a1 to n/2  jal mergesort  lw $a1, 20($sp)   # mergesort(array+mid, n-mid, temp\_aray)  # a0 = array + mid  sll $t4, $t1, 2 # mid \* 4 -> mult for address   add $t2, $a0, $t4 # array + mid  move $a0, $t2 # $a0 = $t2 = array + mid    # a1 = n - mid   sub $t3, $a1, $t1 # sub a1 - t1  move $a1, $t3 # $a1 = $t3 = n - mid   jal mergesort    # Restore a0 + a1  lw $a1, 20($sp)  lw $a0, 24($sp)   # merge(array, n, temp\_array, mid)  move $a3, $t1  jal merge    labelj:  #restore here  lw $ra, 0($sp)  lw $t0, 4($sp)  lw $t1, 8($sp)  lw $t2, 12($sp)  lw $t3, 16($sp)  lw $a1, 20($sp)  lw $a0, 24($sp)  addiu $sp, $sp, 28  jr $ra  #---- Merge --------------------------------------- # Description: compares elements of the array with one another # and puts them in order inside of a temporary # array. It calls arrcpy to put the newly sorted # temp\_array into array  .globl merge merge:  #----------------------------------------------------------  # Register Definitions  #----------------------------------------------------------  # t0 = tpos  # t1 = lpos   # t2 = rpos  # t3 = rn   # t4 = rarr  #----------------------------------------------------------   #allocate memory  addi $sp, $sp, -56 # Make room for items  sw $ra, 0($sp)  sw $t0, 4($sp)  sw $t1, 8($sp)  sw $t2, 12($sp)  sw $t3, 16($sp)  sw $t4, 20($sp)  sw $a0, 24($sp)  sw $a1, 28($sp)  sw $a2, 32($sp)  sw $a3, 36($sp)  sw $s0, 40($sp)   sw $s1, 44($sp)   sw $s2, 48($sp)   sw $s3, 52($sp)     # Initialize Variables  add $t0, $zero, $zero  add $t1, $zero, $zero  add $t2, $zero, $zero  sub $t3, $a1, $a3 # t3 = rn = n - mid  sll $t4, $a3, 2 # Convert mid to memory addr  addu $t4, $a0, $t4 # t4 = rarr = array + mid   # save og parameter vals  move $s0, $a0  move $s1, $a1  move $s2, $a2  move $s3, $a3   # While loop  # Condition  wcondition:  slt $t5, $t1, $a3 # t5 = lpos < mid  slt $t6, $t2, $t3 # t6 = rpos < rn  and $t7, $t5, $t6 # t7 = t5 && t6  bne $t7, $zero, wloop #   j L0 # Jump to outside loop  wloop:  # if ( array[lpos] < rarr[rpos] )  sll $t5, $t1, 2  addu $t5, $s0, $t5 # array[lpos]   sll $t6, $t2, 2  addu $t6, $t4, $t6 # rarr[rpos]   sll $s5, $t0, 2  addu $s5, $s2, $s5 # temp\_array[tpos++]    lw $t8, 0($t5)  lw $t9, 0($t6)  slt $t8, $t8, $t9 # t8 = array[lpos] < rarr [rpos]   bne $t8, $zero, True   False:  lw $t6, 0($t6) # t6 = rarr[rpos]   sw $t6, 0($s5) # temp\_array[tpos] = rarr[rpos]  addiu $t2, $t2, 1 # rpos++   addiu $t0, $t0, 1 # tpos++   j Exit   True:   lw $t5, 0($t5) # t5 = array[lpos]  sw $t5, 0($s5) # temp\_array[tpos++] = array[lpos++]   addiu $t0, $t0, 1 # tpos++   addiu $t1, $t1, 1 # lpos++  Exit:  j wcondition    # if ( lpos < mid)  L0:  slt $t5, $t1, $a3 # t5 = lpos < mid  beq $t5, $zero, L1 # lpos < mid, copy array    # copy\_array  # temp\_array + tpos = $a0  sll $t7, $t0, 2  addu $a0, $s2, $t7  # $a1 = array + lpos  sll $t8, $t1, 2  addu $a1, $s0, $t8  # mid - lpos = $a2  subu $a2, $s3, $t1    jal arrcpy   L1:  # if (rpos < rn)  slt $t6, $t2, $t3 # t6 = rpos < rn  beq $t6, $zero, L2 # t6 = rpos < rn then copy array  # temp\_array + tpos = $a0  sll $t7, $t0, 2  addu $a0, $s2, $t7    # rarr + rpos = $a1  sll $t8, $t2, 2  addu $a1, $t4, $t8     # rn - rpos = $a2  subu $a2, $t3, $t2    # copy\_array  jal arrcpy  L2:   # copy\_array(array, temp\_array, n)  move $a0, $s0  move $a1, $s2  move $a2, $s1   jal arrcpy  # pop off  lw $ra, 0($sp)  lw $t0, 4($sp)  lw $t1, 8($sp)  lw $t2, 12($sp)  lw $t3, 16($sp)  lw $t4, 20($sp)  lw $a0, 24($sp)  lw $a1, 28($sp)  lw $a2, 32($sp)  lw $a3, 36($sp)  lw $s0, 40($sp)   lw $s1, 44($sp)   lw $s2, 48($sp)   lw $s3, 52($sp)   addi $sp, $sp, 56 #Make room for items  jr $ra   #---- arrycpy --------------------------------------- # Description: puts the element values from the source (src) # array into the destination array (dst).  .globl arrcpy arrcpy:  #----------------------------------------------------------  # Register Values  #----------------------------------------------------------  # $t0 = i  # $t0 = src[i]  # $t2 = dst[i]  # $a0 = int \*dst  # $a1 = int \*src  # $a2 = n aka array size  #----------------------------------------------------------  # make int i = 0  add $t0, $0, $0 # i = 0  j test;   loop:  # $t1 = src[i]  sll $t1, $t0, 2 # $t1 = i\*4  add $t1, $a1, $t1 # &(src[i])  lw $t1, 0($t1) # src[i] at $t1    # $t2 = dst[i]  sll $t2, $t0, 2 # $t2 = i\*4  add $t2, $a0, $t2 # &(dst[i])    # dst[i] = $t2  sw $t1, 0($t2)   # i++  addi $t0, $t0, 1  test:  slt $t1, $t0, $a2 # i < n true or false val stored in $t1  bne $t1, $0, loop # if i < n, goto loop  jr $ra |
| --- |