COEN 4710: Spring 2021

Project #1

Zach Thompson

005906989

25 March 2021

Executive Summary:

The program can be assembled and run successfully. Using the integrated instruction counter within the RARS 1.5 software, I calculated the final execution time to be approximately 0.175 seconds - assuming a CCT of 1 nanosecond. I was unable to write the code using the reduced subset of instructions and am ineligible for the bonus points.

Discussion:

My first step in this process was to become as familiar with the high-level program as possible before even attempting a conversion to RISC-V. I looked up the traditional mergeSort algorithm and rewrote the code on my own style just to know where I might get tripped up during the translation process. When I write my own programs, I like divide everything into little blocks and build out from there, testing as often as possible. To this point, I wrote a separate *print* method at first that I could call from the *main* method so that I could print the array whenever I wanted so that I could see where things were going wrong during the process. Although the *merge* method looked more complicated, I decided to start there since it’s not recursive and I’m not totally comfortable with recursion in assembly. Similar to how to I separate out methods into distinct sections, I did the same with the loops. I assumed that I would be able to establish the variables once I got out of the nesting and started with the simple inner actions before expanding out into their branches and conditional statements. Before I even started building the loops, however, I made a list of all the registers I expected to use and assigned them names and purposes to help me keep track of everything. All of this was only possible due to my in-depth understanding of the base code. After I completed innermost for-loop I wrote the if/else-loop that surrounded it, before moving on to the encompassing while-loop. Once the merge method was completed, I felt like I had a good understanding of how the *mergeSort* was going to work and moved on to it. This time I wrote the outline of the if/else-loop before filling it with the recursive calls to *mergeSort* and *merge*. Once all the methods were completed, I organized them to best match the original code. I know the *print* method is not following the provided code exactly, but it was such an important part of my build process I decided to leave it in. However, it could easily be added to the *main* method without issue since it’s only called once during the final execution.

Verification and Testing:

As far as I can tell the software works as intended, not just with the given array but with arrays of different sizes and values as well. That being said, when I run it there is an “Instruction load access error” that causes the execution to be “Terminated with errors” despite running successfully. My guess for cause of this error is improperly ending a method, but I can’t figure it out exactly. The only thing that will cause the program to fail entirely is by providing a lenY1 that is different from the actual Y1 array length. Too small of lenY1 causes some values to be left out, and too large of a lenY1 results in what appears to be some sort of overflow error.

Performance Analysis:

As stated in the Executive Summary section, the RARS 1.5 software has a built-in instruction counter that can be connected to a selected script. Using this tool, I determined my code to have 2813 instructions consisting of 35% R-type, 38% I-type, 6% S-type, 10% B-type, and 7% J-type instructions. Assuming the requirements of: R-type (4 cycles), I-type (5 cycles), S-type (4 cycles), B-type (4 cycles), and J-type (5 cycles), this yields the equation:

= 4.41 CPI

Assuming CPI = 4.41, IC = 2813, and CCT = 1 ns, the provided equation for CPU gives

Conclusion:

The biggest takeaway from this assignment is that assembly is difficult and that I should definitely take my professors more seriously when they say I need to start on a project sooner rather than later. If I had to do this again (please no, have mercy) I would certainly start it the moment it was assigned so that I could accurately gauge the amount of time and effort I should expect to expend on it and not be submitting it at 4am, 5 hours after the due date as I am now. The most challenging aspect during my work on this project was keeping track of register assignments and where I was in a loop. I’m so accustomed to the visual clarity and familiarity of high level languages such as Java, Python, and C, that switching to a mostly foreign format was very disorienting.

RISC-V Code:

# COEN 4710 Project 1

# Convert Java program to RISC-V

# Author: Zach Thompson

# Edited: 3/25/21

# Function: Execute a sorting algorithm called mergeSort to sort a given array by increasing values

.data

lenY1: .word 13 # length of array

Y1: .word 13, 101, 79, 23, 154, 4, 11, 38, 89, 45, 17, 94, 62, 100 # array

newline: .asciz "\n" # new line

space: .asciz " " # space

.text

############################## MAIN METHOD ##############################

main:

addi sp, sp, -4 # create stack space

sw ra, 0(sp) # set stack

# perform MergeSort(Y1, left, right)

la a0, Y1 # load array

addi a1, x0, 0 # set initial index

lw a2, lenY1 # load length

addi a2, a2, -1 # subtract 1 to get last index

jal ra, mergeSort # jump to mergeSort

# print Y1

la a0, Y1 # load sorted array

lw a1, lenY1 # load length

jal ra, print # print sorted array

# end program

lw ra, 0(sp) # set return address

addi sp, sp, 4 # reset stack

jalr x0, ra, 0 # return

############################## MERGESORT METHOD ##############################

mergeSort:

addi sp, sp, -20 # sp = sp - 20 // create stack space

sw ra, 0(sp) # ra = &sp // set stack

addi s0, a0, 0 # s0 = a0 + 0 // s0 = &addr

addi s1, a1, 0 # s1 = a1 + 0 // s1 = left

addi s2, a2, 0 # s2 = a2 + 0 // s2 = right

bge s1, s2, endMergeSort # if(left < right), else do endMergeSort

add s3, s1, s2 # s3 = s1 + s2 // total = left + right

addi s4, x0, 0 # s4 = x0 + 0 // max = zero when s3 > 0

midCheck:

blt s3, x0, sort # branch to sort if (s3 < x0)

addi s3, s3, -2 # s3 = s3 - 2 // total = total - 2

addi s4, s4, 1 # s4 = s4 + 1 // increment max

jal x0, midCheck # set zero to return, do midCheck

sort:

addi s3, s4, -1 # s3 = s4 - 1

sw s0, 4(sp) # store s0 // &address

sw s1, 8(sp) # store s1 // left

sw s2, 12(sp) # store s2 // right

sw s3, 16(sp) # store s3 // mid

# preparations for left mergeSort

addi a0, s0, 0 # set a0 // &address (s0)

addi a1, s1, 0 # set a1 // left (s1)

addi a2, s3, 0 # set a2 // mid (s3)

jal ra, mergeSort # reset return, mergeSort

# preparation for right mergeSort

lw a0, 4(sp) # load a0 // &addr (4sp)

lw a1, 16(sp) # load a1 // mid (16sp)

addi a1, a1, 1 # mid = mid + 1

lw a2, 12(sp) # load a2 // right (sp+12)

jal ra, mergeSort # reset return, mergeSort

# preparations for merge

lw a0, 4(sp) # a0 = &addr // (4sp)

lw a1, 8(sp) # a1 = left // (8sp)

lw a2, 16(sp) # a2 = mid // (16sp)

lw a3, 12(sp) # a3 = right // (12sp)

jal ra, merge # set return, merge

endMergeSort:

lw ra, 0(sp) # reset return

addi sp, sp, 20 # reset stack spacing

jalr x0, ra, 0 # set zero to return and jump to return address

############################## MERGE METHOD ##############################

merge:

sub t0, a3, a1 # t0 = right - left // count

addi t0, t0, 1 # count++

add t1, t0, t0 # t1 = count \* 2 // half of stack space need

add t1, t1, t1 # t1 = t1 \* t1 // total stack space needeed

xori t2, t1, 0xffffffff # t2 = -t1 // 2's complement

addi t2, t2, 1 # t2++

add sp, sp, t2 # stack pointer = t2

addi t3, a1, 0 # t3 = left // index of old Y1 (memory)

addi t2, x0, 0 # t2 = 0 // index of new Y1 (stack)

read:

blt a3, t3, endRead # if (left < right), else do endRead

add t4, t3, t3 # t4 = left + left // offset space = left \* 2

add t4, t4, t4 # t4 = t4 + t4 // offset space = left \* 4

add t4, a0, t4 # t4 = a0 + t4 // t4 = &addr + offset space

lw t5, 0(t4) # t5 = addr(t4)

add t6, t2, t2 # t6 = right + right // offset space = right \* 2

add t6, t6, t6 # t6 = t6 + t6 // offset\_space = right \* 4

add t6, sp ,t6 # t6 = sp + t6 // stack point + offset

sw t5, 0(t6) # t5 = addr(t6)

addi t2, t2, 1 # t2 = t2 + 1 // increment index

addi t3, t3, 1 # t3 = t3 + 1 // increment left

jal x0, read # set zero to return address, readToStack

endRead:

sub t4, a2, a1 # t4 = a2 - a1 // left\_max = mid - left

sub t5, a3, a1 # t5 = a3 - a1 // right\_max = right - left

addi t2, x0, 0 # t2 = x0 // left\_index = 0

addi t3, t4, 1 # t3 = t4 + 1 // right\_index = left\_max + 1

addi t6, a1, 0 # t6 = a1 // reset index to left

mergeLoop:

slt t0, t4, t2 # t0 = lesser of t4, t2 // offset\_left

slt t1, t5, t3 # t1 = lesser of t5, t3 // offset\_right

or t0, t0, t1 # t0 = t0 || t1 // offset space needed

xori t0, t0, 0x1 # t0 = ~t0 // t0 = t0 \* - 1

beq t0, x0, endMergeLoop # if ((t0 || t1) != 0), else do endMergeLoop

add t0, t2, t2 # t0 = t2 + t2 // offset\_left = left\_index \* 2

add t0, t0, t0 # t0 = t0 + t0 // offset\_left = left\_index \* 2

add t0, sp, t0 # t0 = sp + t0 // offset\_left = stack pointer + offset\_left = left

lw t0, 0(t0) # t0 = &addr // left = left\_address

add t1, t3, t3 # t1 = t3 + t3 // offset\_right = right\_index \* 2

add t1, t1, t1 # t1 = t1 + t1 // offset\_right = offset\_right + offset\_right = right\_index \* 4

add t1, sp, t1 # t1 = sp + t11 // offset\_right = stack pointer + offset\_right = right

lw t1, 0(t1) # t1 = &addr // right = right\_address

blt t1, t0, rightSmaller # if (left <= right), else do rightSmaller

add t1, t6, t6 # t1 = t6 + t6 // offset\_right = memory\_index \* 2

add t1, t1, t1 # t1 = t1 + t1 // offset\_right = offset right + offset right = memory\_index \* 4

add t1, a0, t1 # t1 = a0 + t1 // memory\_index = $addr + memory\_index

sw t0, 0(t1) # t0 = &addr // left = left\_value

addi t6, t6, 1 # t6 = t6 + 1 // increment memory\_index

addi t2, t2, 1 # t2 = t2 + 1 // increment left\_index

jal x0, mergeLoop # set zero to return address, mergeLoop

rightSmaller:

add t0, t6, t6 # t0 = t6 + t6 // offset\_left = mem\_index \* 2

add t0, t0, t0 # t0 = t0 + t0 // offset\_left = offset\_left \* 2 = memory\_index \* 4

add t0, a0, t0 # t0 = a0 + t0 // memory\_addr = &addr + offset\_left

sw t1, 0(t0) # t1 = &addr // right\_value = memory\_addr

addi t6, t6, 1 # t6 = t6 + 1 // increment memory\_index

addi t3, t3, 1 # t3 = t3 + 1 // increment right\_index

jal x0, mergeLoop # set zero to return address, mergeLoop

endMergeLoop:

bge t5, t3, rightLoop # if (right\_max >= right\_index) do rightLoop

leftLoop:

add t0, t2, t2 # t0 = t2 + t2 // offset\_left = left\_index \* 2

add t0, t0, t0 # t0 = t0 + t0 // offset\_left = offset\_left \* 2

add t0, sp, t0 # t0 = stack pointer + t0 // offset\_left = left\_value

lw t0, 0(t0) # t0 = &addr // left\_value = &left\_addr

add t1, t6, t6 # t1 = t6 + t6 // offset\_right = memory\_index \* 2

add t1, t1, t1 # t1 = t1 + t1 // offset\_right = offset\_right \* 2

add t1, a0, t1 # t1 = a0 + t1 // offset\_right = &addr + offset\_right = right\_value

sw t0, 0(t1) # t0 = &addr // left\_value = &addr

addi t6, t6, 1 # t6 = t6 + 1 // increment memory\_index

addi t2, t2, 1 # t2 = t2 + 1 // increment left\_index

bge t4, t2, leftLoop # if (left\_max >= left\_index) do leftLoop

jal x0, endMerge # set zero to return address, do mergeLoop

rightLoop:

add t1, t3, t3 # t1 = t3 + t3 // offset\_right = right\_index \* 2

add t1, t1, t1 # t1 = t1 + t1 // offset\_right = offset\_right \* 2

add t1, sp, t1 # t1 = stack pointer + t1 // offset\_right = right\_value

lw t1, 0(t1) # t1 = &addr // right\_value = &right\_addr

add t0, t6, t6 # t0 = t6 + t6 // offset\_left = memory\_index \* 2

add t0, t0, t0 # t0 = t0 + t0 // offset\_left = offset\_left \* 2

add t0, a0, t0 # t0 = a0 + t0 // offset\_left = &addr + offset\_left = left\_value

sw t1, 0(t0) # t1 = &addr // right\_value = &addr

addi t6, t6, 1 # t6 = t6 + 1 // increment memory\_index

addi t3, t3, 1 # t3 = t3 + 1 // increment right\_index

bge t5, t3, rightLoop # if (right\_max >= right\_index) do rightLoop

jal x0, endMerge # set zero to return address, do endMerge

endMerge:

sub t0, a3, a1 # t0 = a3 - a1 // index\_count = right - left

addi t0, t0, 1 # t0 = t0 + 1 // increment index\_count

add t1, t0, t0 # t1 = t0 + t0 // memory\_space = index\_count \* 2

add t1, t1, t1 # t1 = t1 + t1 // memory\_space = index\_count \* 4

add sp, sp, t1 # sp = sp + t1 // stack pointer = stack pointer + memory\_space

jalr x0, ra, 0 # set zero to return and jump to return address

############################## PRINT METHOD ##############################

print:

addi t0, a0, 0 # t0 = a0 // Y1

addi t1, a1, 0 # t1 = a1 // initial index

addi t2, x0, 0 # t2 = zero // count (i)

printLoop:

add t3, t2, t2 # t3 = t2 + t2 // t3 = i \* 2

add t3, t3, t3 # t3 = t3 + t3 // t3 = i \* 4

add t3, t0, t3 # t3 = t0 + t3 // t3 = Y1[i]

lw a0, 0(t3) # a0 = &addr // a0 = &Y1[i]

addi a7, x0, 1 # a7 = x0 + 1 // load instruction

ecall # perform instruction

addi t2, t2, 1 # t2 = t2 + 1 // i++

bge t2, t1, endPrint # if (t2 > t1) do endPrint

la a0, space # a0 = space

addi a7, x0, 4 # a7 = x0 + 4 // shift a7 left

ecall # perform instruction

jal x0, printLoop # reset return, do printLoop

endPrint:

la a0, newline # a0 = newline

addi a7, x0, 4 # set a7 as left

ecall # perform instruction

jalr x0, ra, 0 # set zero to return and jump to return