Rodrigo Becerril Ferreyra

CECS 440 Section 02

Lab 2

01 October 2021

Part 1

Before I get started with this Part, I would like to post the expected results.

Problem 1:

```
#include <stdio.h>

int main(void)
{
    int var1[4] = {7, 18, 11, 3};
    int var2[4] = {12, 14, 7, 18};
    int result[4] = {0};

    for(int i = 0; i < 4; i++)
    {
        result[i] = var1[i] - var2[i];
        printf("%d ", result[i]);
    }

    return 0;
}
```

The result of this code is

```
-5 4 4 -15
```

Problem 2:

```c
#include <stdio.h>

int main(void)
{
    short var1[4] = {7, 18, 11, 3};
    short var2[4] = {12, 14, 7, 18};
    short result[4] = {0};

    for(int i = 0; i < 4; i++)
    {
        result[i] = var1[i] - var2[i];
        printf("%d ", result[i]);
    }

    return 0;
}
```

The result of this code is

```
-5 4 4 15
```

Problem 3:

```c
#include <stdio.h>

int main(void)
{
    unsigned short var1[4] = {7, 18, 11, 3};
    unsigned short var2[4] = {12, 14, 7, 18};
    unsigned short result[4] = {0};

    for(int i = 0; i < 4; i++)
    {
        result[i] = var1[i] - var2[i];
        printf("%d ", result[i]);
    }

    return 0;
}
```

The result of this code is:

```
65531 4 4 65521
```

Problem 1 Responses:

There is a lot of data in Problem 1. There are three arrays, each with four elements each, which means I needed to use `.data` to create sequential words in memory. The address of `var1` is stored in `$s0`, the address of `var2` is stored in `$s1`, and the address of `result` is stored in `$s2`. The temp variables are used for many different purposes. `$t0` is used to load the initial values into the data, as well as to keep track of the iteration. `$t1` is used to align address values in order to access words; this means that it is `$t0` shifted to the left two times. $t1 can also be the check that is used to see whether the program should loop again or not. `$t2`, `$t3`, and `$t4` hold the address of `var1[i]`, `var2[i]`, and `result[i]`, respectively, but only after they have been aligned with memory; in practice, this means that they are always `$s0`, `$s1`, and `$s2` after they have been added to `$t1`. `$t5`, `$t6`, and `$t7` all hold the values of `var1[i]`, `var2[i]`, and `result[i]`, respectively; the last set of three registers only held the addresses of those values.

The three arrays are stored in memory sequentially. These are stored as follows:

```
                          User data segment [10000000]..[10040000]
[10000000]..[1000ffff]  00000000
[10010000]    00000007  00000012  0000000b  00000003    . . . . . . . . . . . . . . . .
[10010010]    0000000c  0000000e  00000007  00000012    . . . . . . . . . . . . . . . .
[10010020]    fffffffb  00000004  00000004  fffffff1    . . . . . . . . . . . . . . . .
[10010030]..[1003ffff]  00000000
```

Figure 1: The final output for Part 1 Problem 1.

The address 0x10010000 stores the address of the first element in array var1. This array has a total size of four words (which is equal to 16 bytes or 128 bits), with each element taking up one 32-bit word (the size of an `int` in a standard C compiler). Since MIPS is a big-endinan

ISA, the values are added in from the most significant word to the least significant word; these are 0x7, 0x12, 0xb, and 0x3. `var2` has a similar arrangement, with its contents allocated in four words starting from 0x10010010, which is 16 bytes higher than the starting address of `var1`. The last array is only initialized after the loop begins; this is the `result` array, which has a starting address of 0x10010020 (16 bytes higher than `var2`).

The final values located in the `result` array are 0xfffffffb, 0x00000004, 0x00000004, and 0xfffffff1. Translated into signed decimal representation, these values are -5, 4, 4, and -15, which is the expected value.

Below is the final code for `Part1int.asm` (also included in the submission):

```
# Project name: Lab 2
# Author      : Rodrigo Becerril
Ferreyra
# Date written: 30 September 2021

.data

    var1:   .word 0, 0, 0, 0
    var2:   .word 0, 0, 0, 0
    result: .word 0, 0, 0, 0

.text
.globl main

main:

    # int var1[4] = {7, 18, 11,
3};
    la $s0, var1

    addi $t0, $zero, 7
    sw $t0, 0($s0)
    addi $t0, $zero, 18
    sw $t0, 16($s0)
    addi $t0, $zero, 11
    sw $t0, 32($s0)
    addi $t0, $zero, 3
    sw $t0, 64($s0)

    # int var2[4] = {12, 14, 7,
18};
    la $s1, var2

    addi $t0, $zero, 12
    sw $t0, 0($s1)
    addi $t0, $zero, 14
    sw $t0, 4($s1)
    addi $t0, $zero, 7
    sw $t0, 8($s1)
    addi $t0, $zero, 18
    sw $t0, 12($s1)

    # int result
    la $s2, result

    # from this point on:
    # t0 = i
    # t1 can be (i * 4) or can be
(i < 4)
    # t2 = address of var1[i * 4]
    # t3 = address of var2[i * 4]
    # t4 = address of var3[i * 4]
    # t5 = value of *t2
    # t6 = value of *t3
    # t7 = t5 + t6

    # s0 = address of var1
    # s1 = address of var2
    # s2 = address of result

    # int i = 0
    add $t0, $zero, $zero

    loop:
        # loop body

        sll $t1, $t0, 2      # t1
= i * 4

        add $t2, $t1, $s0    #
addr of var1[i * 4]
        lw $t5, 0($t2)

        add $t3, $t1, $s1    #
addr of var2[i * 4]
        lw $t6, 0($t3)

        add $t4, $t1, $s2    #
addr of result[i * 4]

        sub $t7, $t5, $t6    #
result[i*4] = var1[i*4] -
var2[i*4];
        sw $t7, 0($t4)

        addi $t0, $t0, 1     # i =
i + 1

        slti $t1, $t0, 4     # 1
if i < 4
        bne $zero, $t1, loop #
branch if (i < 4) is false

    # exit
    addi $v0, $zero, 10
    syscall
```

Figure 2: Code for Part 1Problem 1.

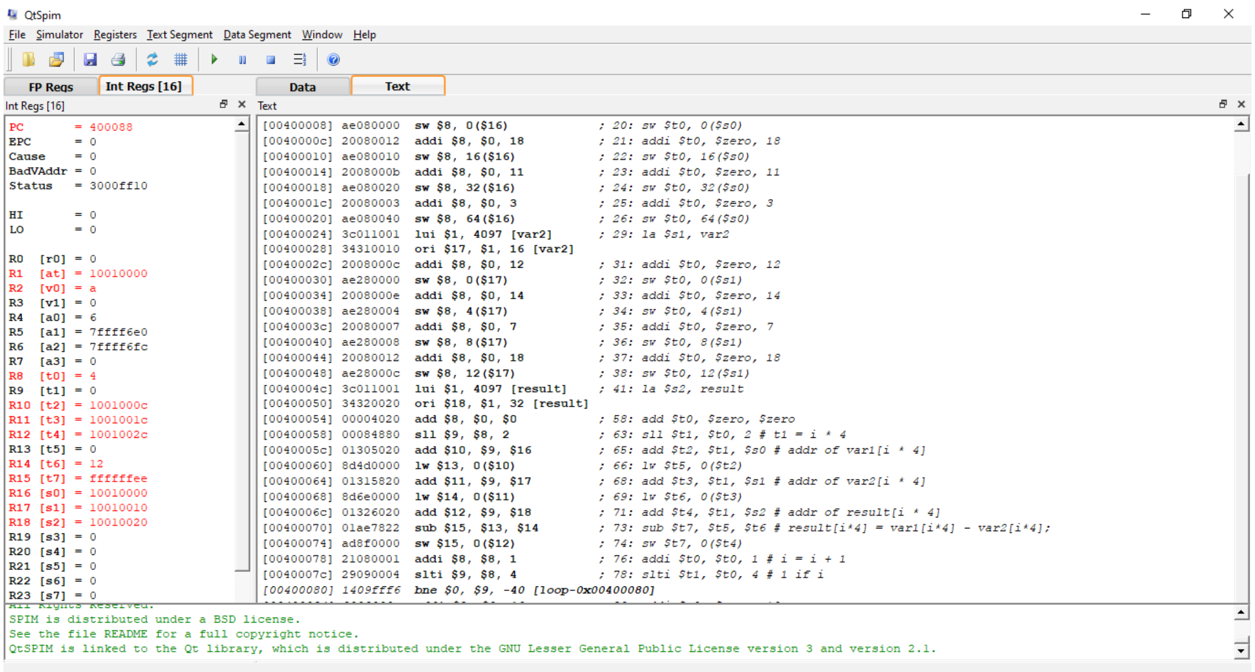

Figure 3: Part 1 Problem 1 running on QtSpim.

Problem 2 Responses:

The only difference between Problem 1 and Problem 2 is that Problem 2 works with half-words instead of words (meaning each value is only 16 bits wide instead of 32). The differences in the program are as follows:

- The commands `lh` and `sh` are used instead of `lw` and `sw`.
- Loads and stores must be aligned to two bytes instead of four (since a half-word is two bytes long).

The same registers used for Problem 1 are used in Problem 2, and for the same purposes.

The following is the memory addresses and values for the three arrays.

```
                       User data segment [10000000]..[10040000]
[10000000]..[1000ffff]   00000000
[10010000]     00120007  0003000b  000e000c  00120007    . . . . . . . . . . . . . . . .
[10010010]     0004fffb  fff10004  00000000  00000000    . . . . . . . . . . . . . . . .
[10010020]..[1003ffff]   00000000
```

The total amount of data here is cut in half because the data size is half of what it was in Problem 1. The same values are present here, even if it is in a different format. The first two words in 0x10010000 are the four values in `val1`: 7, 18, 11, and 3. The order goes from lower 16 bits to upper 16 bits, then from small address to large address. `val2` starts from address 0x10010008, and `result` is stored at 0x10010010. Reading the values off the address like last time, the results are 0xfffb, 0x0004, 0x0004, and 0xfff1; these are the same values achieved in Problem 1.

Below is the final `Part1hword.asm` program.

```
# Project name: Lab 2
# Author      : Rodrigo Becerril
Ferreyra
# Date written: 30 September 2021

.data

    var1:   .half 0, 0, 0, 0
    var2:   .half 0, 0, 0, 0
    result: .half 0, 0, 0, 0

.text
.globl main

main:

    # int var1[4] = {7, 18, 11,
3};
    la $s0, var1

    addi $t0, $zero, 7
    sh $t0, 0($s0)
    addi $t0, $zero, 18
    sh $t0, 2($s0)
    addi $t0, $zero, 11
    sh $t0, 4($s0)
    addi $t0, $zero, 3
    sh $t0, 6($s0)

    # int var2[4] = {12, 14, 7,
18};
    la $s1, var2

    addi $t0, $zero, 12
    sh $t0, 0($s1)
    addi $t0, $zero, 14
    sh $t0, 2($s1)
    addi $t0, $zero, 7
    sh $t0, 4($s1)
    addi $t0, $zero, 18
    sh $t0, 6($s1)

    # int result
    la $s2, result

    # from this point on:
    # t0 = i
    # t1 can be (i * 2) or can be
(i < 4)
    # t2 = address of var1[i * 2]
    # t3 = address of var2[i * 2]
    # t4 = address of var3[i * 2]
    # t5 = value of *t2
    # t6 = value of *t3
    # t7 = t5 + t6

    # s0 = address of var1
    # s1 = address of var2
    # s2 = address of result

    # int i = 0
    add $t0, $zero, $zero

    loop:
        # loop body

        sll $t1, $t0, 1       # t1
= i * 2

        add $t2, $t1, $s0     #
addr of var1[i * 2]
        lh $t5, 0($t2)

        add $t3, $t1, $s1     #
addr of var2[i * 2]
        lh $t6, 0($t3)

        add $t4, $t1, $s2     #
addr of result[i * 2]

        sub $t7, $t5, $t6     #
result[i*2] = var1[i*2] -
var2[i*2];

        sh $t7, 0($t4)

        addi $t0, $t0, 1      # i =
i + 1

        slti $t1, $t0, 4      # 1
if i < 4
        bne $zero, $t1, loop #
branch if (i < 4) is false

    # exit
    addi $v0, $zero, 10
    syscall
```

Figure 4: The final program for Part 1 Problem 2.

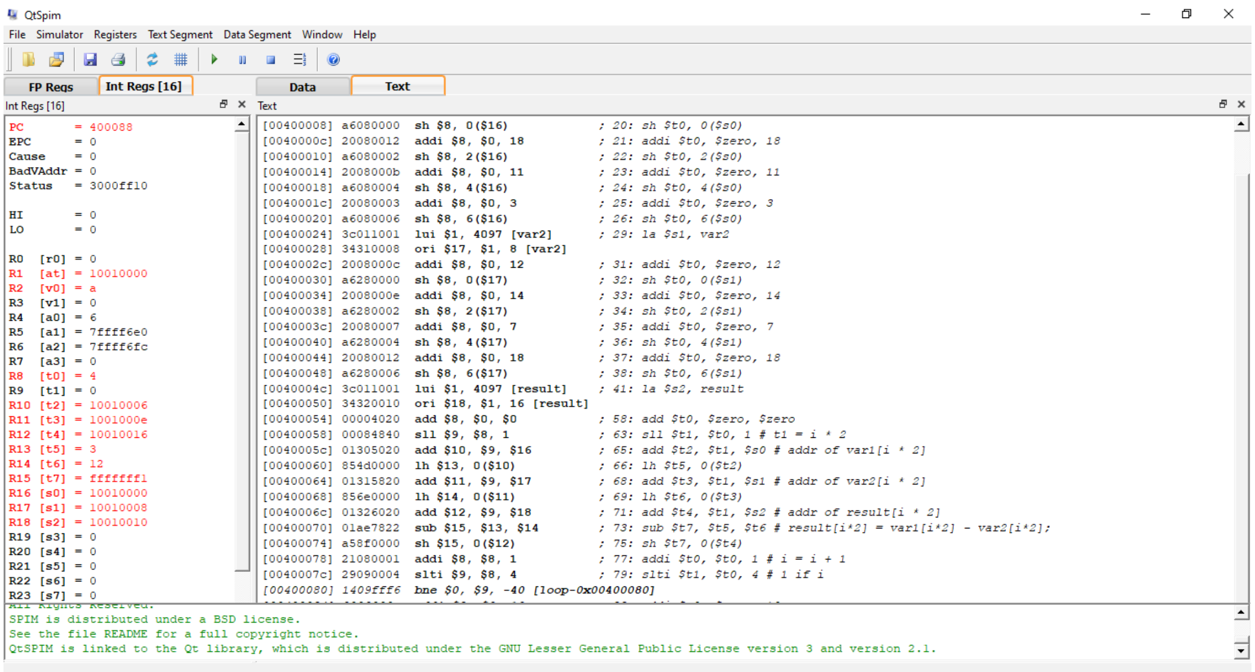

Figure 5: Part 1 Problem 2 running on QtSpim.

Problem 3 Responses:

For Problem 3, I took Problem 2 and replaced all the add, subtract, and load commands with their unsigned variants.

The variable registers are the same as in the previous Problems.

The array locations and format are the same as in Problem 2.

The resulting memory values are presented below.

```
                        User data segment [10000000]..[10040000]
[10000000]..[1000ffff]   00000000
[10010000]      00120007  0003000b  000e000c  00120007    . . . . . . . . . . . . . . . .
[10010010]      0004fffb  fff10004  00000000  00000000    . . . . . . . . . . . . . . . .
[10010020]..[1003ffff]   00000000
```

As you can see, this is actually the same memory output as Problem 2. This is because there is no difference between the add and subtract instructions compared to their unsigned variants. The loads don't affect anything either; and there is no signed store instruction.

Below is the code for `Part1uhword.asm`.

```
# Project name: Lab 2
# Author      : Rodrigo Becerril Ferreyra
# Date written: 30 September 2021

.data

    var1:   .half 0, 0, 0, 0
    var2:   .half 0, 0, 0, 0
    result: .half 0, 0, 0, 0

.text
.globl main

main:

    # int var1[4] = {7, 18, 11, 3};
    la $s0, var1

    addiu $t0, $zero, 7
    sh $t0, 0($s0)
    addiu $t0, $zero, 18
    sh $t0, 2($s0)
    addiu $t0, $zero, 11
    sh $t0, 4($s0)
    addiu $t0, $zero, 3
    sh $t0, 6($s0)

    # int var2[4] = {12, 14, 7, 18};
    la $s1, var2

    addiu $t0, $zero, 12
    sh $t0, 0($s1)
    addiu $t0, $zero, 14
    sh $t0, 2($s1)
    addiu $t0, $zero, 7
    sh $t0, 4($s1)
    addiu $t0, $zero, 18
    sh $t0, 6($s1)

    # int result
    la $s2, result

    # from this point on:
    # t0 = i
    # t1 can be (i * 2) or can be (i < 4)
    # t2 = address of var1[i * 2]
    # t3 = address of var2[i * 2]
    # t4 = address of var3[i * 2]
    # t5 = value of *t2
    # t6 = value of *t3
    # t7 = t5 + t6

    # s0 = address of var1
    # s1 = address of var2
    # s2 = address of result

    # int i = 0
    add $t0, $zero, $zero

    loop:
        # loop body

        sll $t1, $t0, 1      # t1 = i * 2

        add $t2, $t1, $s0    # addr of var1[i * 2]
        lhu $t5, 0($t2)

        add $t3, $t1, $s1    # addr of var2[i * 2]
        lhu $t6, 0($t3)

        add $t4, $t1, $s2    # addr of result[i * 2]

        subu $t7, $t5, $t6    # result[i*2] = var1[i*2] - var2[i*2];

        sh $t7, 0($t4)

        addi $t0, $t0, 1     # i = i + 1

        slti $t1, $t0, 4     # 1 if i < 4
        bne $zero, $t1, loop # branch if (i < 4) is false

    # exit
    addi $v0, $zero, 10
    syscall
```

Figure 6: Code for Part 1 Problem 3.

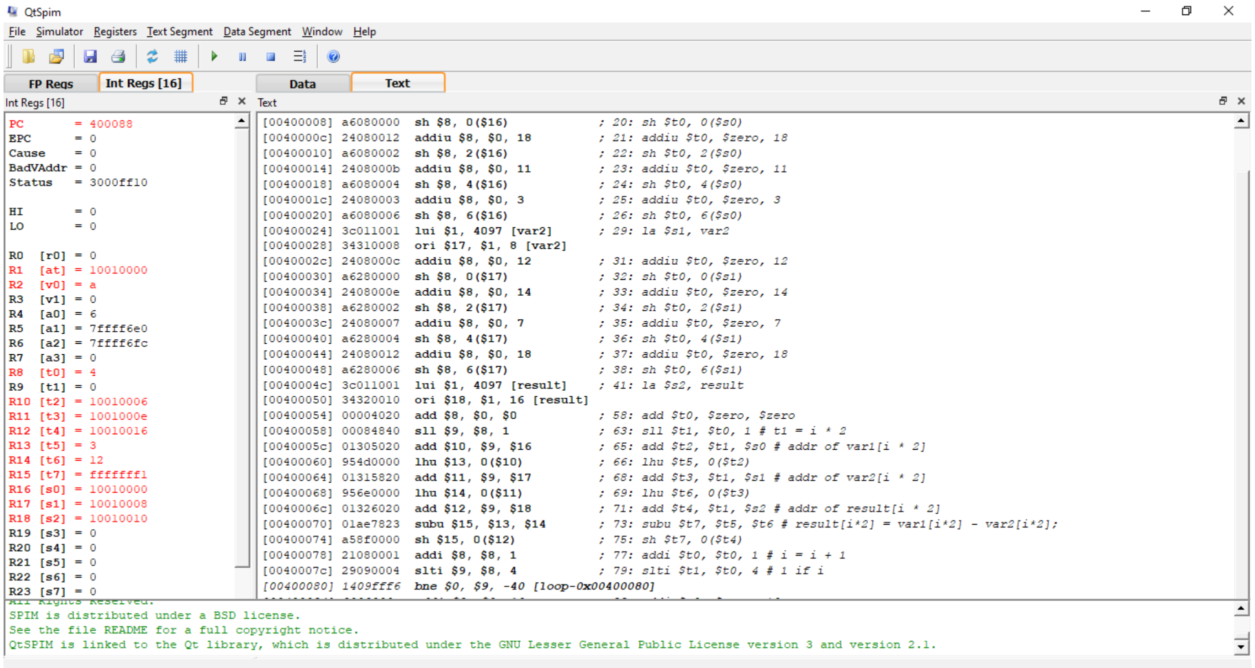

Figure 7: Part 1 Problem 3 running on QtSpim.

Part 2

Problem 1

This problem was a simple implementation of a function. It is important to note that nothing needed to be put on the stack, because the only registers that were modified were `$t0`, `$a0`, and `$a1`, and the function called was a leaf function—that is, it did not call any other function. Technically, if I needed to, I could copy the arguments from the `a` registers to temporary ones if I wanted to avoid moving argument register values around, but I saw no need for doing so, and it would only add more instructions to the program.

It is worth noting that the program works as intended; i.e., it returns the difference of the highest and lowest argument.

```
# Project name: Lab 2
# Author      : Rodrigo Becerril Ferreyra
# Date written: 30 September 2021

.text
.globl main

main:

    # var1 = s0
    # var2 = s1
    # result = s2

    # var1 = 30
    addi $s0, $zero, 30

    # var2 = 210
    addi $s1, $zero, 210

    # function call and usage of return value
    add $a0, $zero, $s0
    add $a1, $zero, $s1
    jal distance
    add $s2, $zero, $v0

    # exit
    addi $v0, $zero, 10
    syscall

distance:
    # a0 = a, a1 = b

    # if b > a then swap
    slt $t0, $a0, $a1        # 1 if a < b, 0 if a >= b
    beq $t0, $zero, skipswap # skip if 0

    add $t0, $zero, $a0
    add $a0, $zero, $a1
    add $a1, $zero, $t0

    skipswap: sub $v0, $a0, $a1
    jr $ra
```

Figure 8: The program for Part 2 Problem 1.

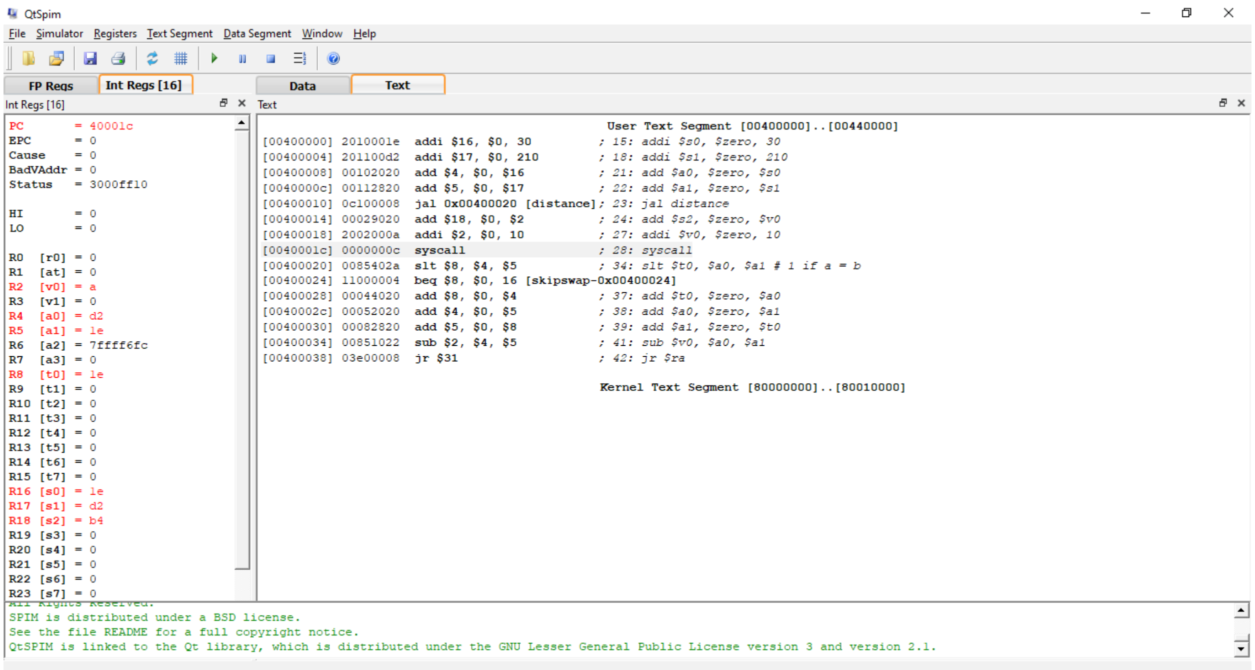

Figure 9: Part 2 Problem 1 running on QtSpim.

Problem 2

This problem was a bit more difficult to implement, since one function calls another. In this case, I had to push the return address of the first function to the stack (because the command `jal` overwrites it), then pop it when the second function returned control to the first. It is worth noting that this program works as intended as well, tested using different values. Also, nothing else needed to be pushed to the stack. The `swap` function given does not swap because the program does not use pointers. If I were to program like a compiler would, I would `distance`'s arguments to the stack, place `swap`'s arguments in the argument registers, call the function, copy the saved arguments from the stack, and pop them. This would result in nothing happening, however, and perhaps `distance` returning a negative number, which is behavior we don't want in this program.

```
# Project name: Lab 2
# Author      : Rodrigo
Becerril Ferreyra
# Date written: 30 September
2021

.text
.globl main

main:

    # var1 = s0
    # var2 = s1
    # result = s2

    # var1 = 30
    addi $s0, $zero, 30

    # var2 = 210
    addi $s1, $zero, 210

    # function call and usage
of return value
    add $a0, $zero, $s0
    add $a1, $zero, $s1
    jal distance
    add $s2, $zero, $v0

    # exit
    addi $v0, $zero, 10
    syscall

distance:
    # a0 = a, a1 = b
    # if b > a then swap
    slt $t0, $a0, $a1        #
1 if a < b, 0 if a >= b
    beq $t0, $zero, skipswap #
skip if 0

    # place $ra on the stack
    sub $sp, $sp, 4
    sw $ra, 0($sp)

    # function call
    jal swap

    # pop $ra from the stack
    lw $ra, 0($sp)
    add $sp, $sp, 4

    skipswap: sub $v0, $a0, $a1
    jr $ra

swap:
    # a0 = a, a1 = b
    # No need to use the stack
here
    # If I did, then the values
themselves would not change

    # swap body
    add $t0, $zero, $a0
    add $a0, $zero, $a1
    add $a1, $zero, $t0

    # return
    jr $ra
```

Figure 10: Code for Part 2 Problem 2.

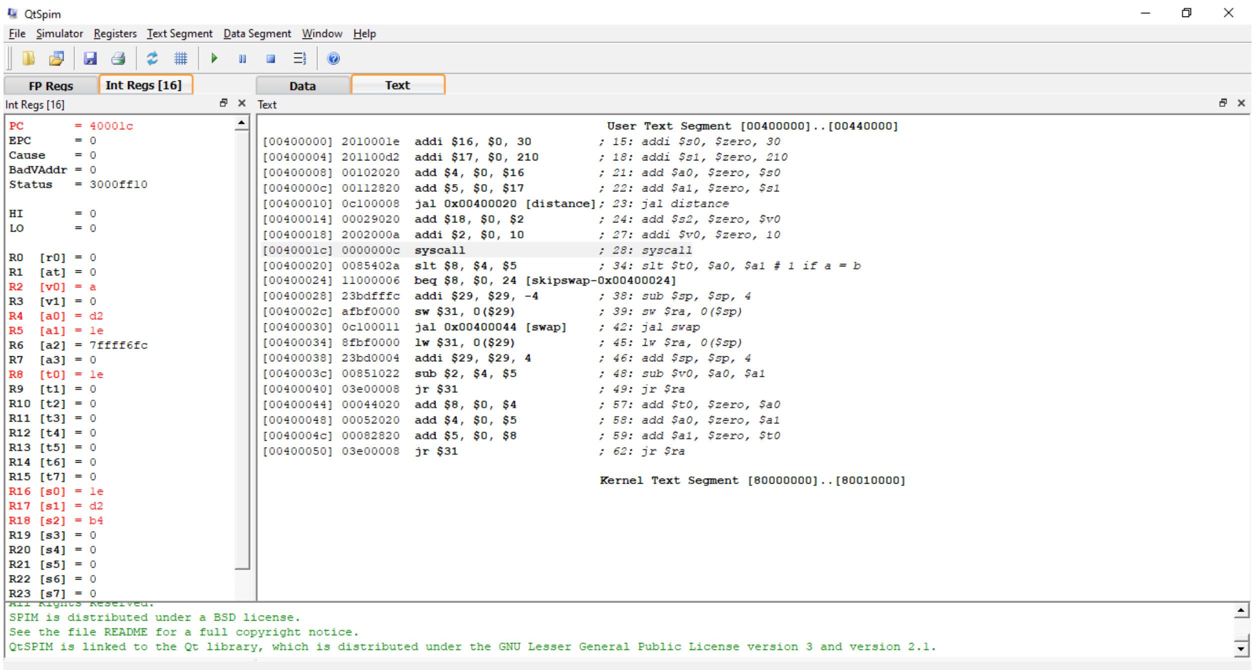

Figure 11: QtSpim running Part 2 Problem 2.