HACETTEPE UNIVERSITY COMPUTER ENGINEERING DEPARTMENT

ADVANCED COMPUTER ARCHITECTURE PROJECT

Ahmet Burak KOÇ | CMP634 | 15 November 2020

1) Arrays using for loops

Converting C code fragment to MIPS code is conducted. In MIPS code, multiplication instructions (mult/mul) are not allowed to be used.

C CODE FRAGMENT

Below C code fragment is to be converted.

```
int A[4];
int i;
int diff;
for(i=0; i<3; i++)
{
        diff = A[i+1] - A[i];
        if (diff > 0)
                A[i] = 5*A[i];
        else
                A[i+1] = -5*A[i];

}
```

MIPS CODE

For given C code fragment, MIPS code is written as below.

```
.data
##; Input data
A: .word 8, 6, 4, 2

.text

##      ;Main function registers
#       Base address: $t0
#       A(i)            : $s0
#       A(i+1)          : $s1
#       diff            : $s2
##      ;For loop registers
#       slt_res_for     : $t1
#       i               : $t2
#       i_limit         : $t3
##      ;Multiplication loop registers
#       slt_res_mult    : $t4
#       j               : $t5
#       multiplier      : $t6
#       product         : $s3
##      ;If registers
#       slt_res_if      : $t7

main:
        la $t0, A                       # $t0, A base address
        addi $s2, $0, 0                 # $s2, diff = 0
        addi $t2, $0, 0                 # $t2, i = 0
        addi $t3, $0, 3                 # $t3, i_limit = 3

##; for loop begin##
loop_for:
        slt  $t1, $t2, $t3              # $t1 = 1 if i is less than i_limit=3
        beq  $t1, $0, done_for          # jump done_for if $t1 = 0 (condition is false)
        lw      $s0, 0($t0)             # $s0, load A(i)
        lw      $s1, 4($t0)             # $s1, load A(i+1)
```

```
            sub  $s2, $s1, $s0              # A(i+1)-A(i) -> diff

##; calculation 5*A[i] begin##
            addi $t5, $0, 0                 # j=0
            addi $t6, $0, 5                 # multiplier=5
            addi $s3, $0, 0                 # product=0
loop_mult:
            slt $t4, $t5, $t6               # slt_res_mult=1 if j is less than multiplier=5
            beq $t4, $0, done_mult          # jump done_mult slt_res_mult=0
            add $s3, $s3, $s0               # product=product+A(i)
            addi $t5, $t5, 1                # increment j by 1
            j    loop_mult                  # jump loop_mult
done_mult:
##; calculation 5*A(i) end##

##; if/else begin ##
            slt $t7, $0, $s2                # slt_res_if=1 if 0 is less than diff
            beq $t7, $0, else_op            # if diff is less than 0 continue, else jump else_op
if_op:
            sw $s3, 0($t0)                  # store product --> A(i)
            j done_if                       # jump done_if
else_op:
            sub $s3, $0, $s3                # product = -product
            sw $s3, 4($t0)                  # store product --> A(i+1)
done_if:
##; if/else end##

            addi $t0, $t0, 4                # increment array address by 4 (index by 1)
            addi $t2, $t2, 1                # increment i by 1
            j    loop_for                   # jump loop_for
done_for:
##; for loop end##

endloop:
   li      $v0, 10                          # terminate program run and
   syscall                                  # Exit
```

OPTIMISATION

For multiplication function addition in for loop is used which is not efficient way of multiplication but one can eaily change of multiplier value. For more efficient multiplication with 5, it can be replaced with

```
sll $s3, $s0, 2           # product=4*A(i)
add $s3, $s3, $s0         # product = product + A(i) = 5*A(i)
```

TESTS & SCREENSHOTS

For given C code fragment, MIPS code is written as below. The program is tested for the following input values:

Test 1: A={2,4,6,8}
Test 2: A={8,6,4,2}
Test 3: A={2,2,6,4}

For each test, screenshots of the memory before and after running the code are given in Figure 1, Figure 2, Figure 3, Figure 4, Figure 5 and Figure 6. **Values are in Hexadecimal form.**

Before test 1, A contains {2, 4, 6, 8} in decimal. After test 1, A contains {10, 20, 30, 8} in decimal.

Before test 2, A contains {8, 6, 4, 2} in decimal. After test 1, A contains {8, -200, 4, -20} in decimal.

Before test 3, A contains {2, 2, 6, 4} in decimal. After test 1, A contains {2, -50, 6, -30 } in decimal.

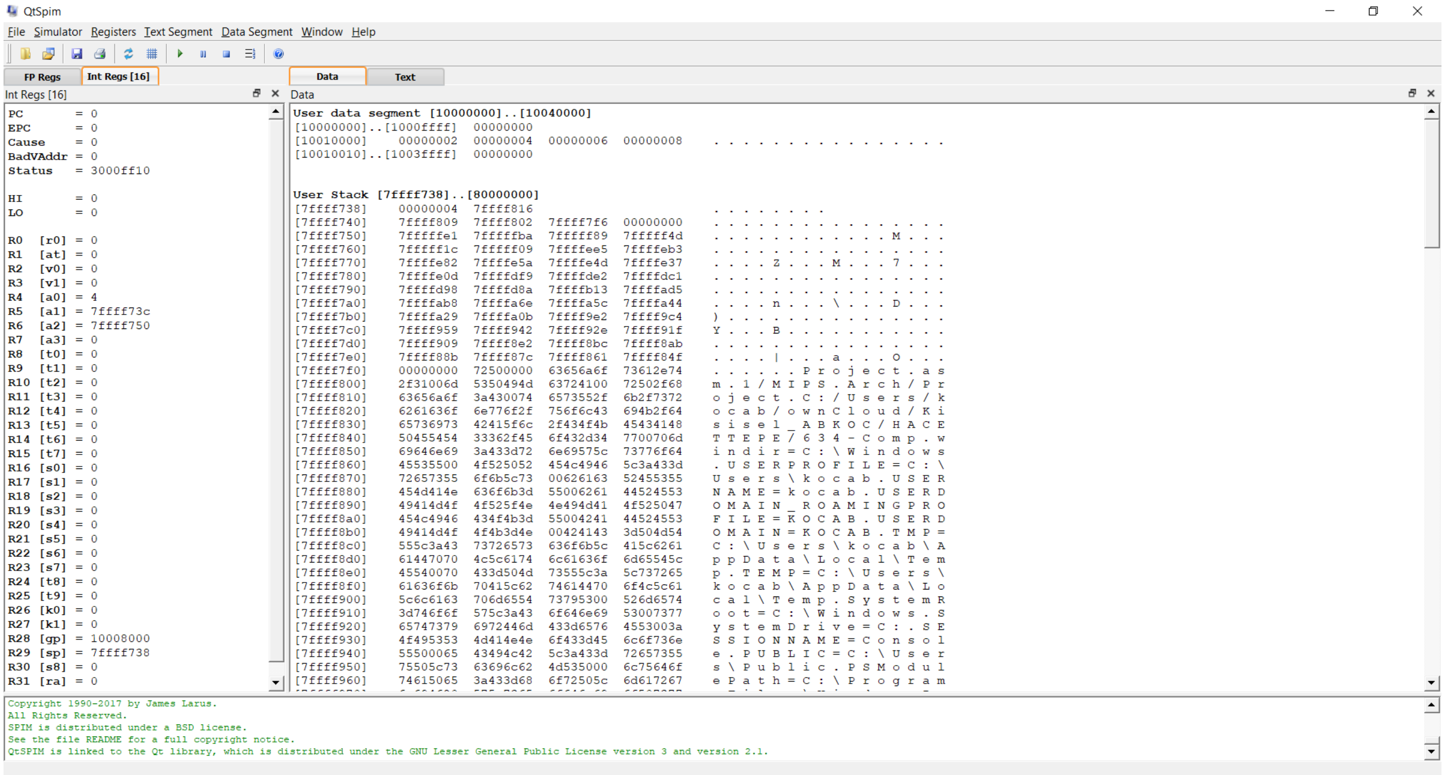

Figure 1: Memory and Register Values (hex) before Test 1: A={2,4,6,8}

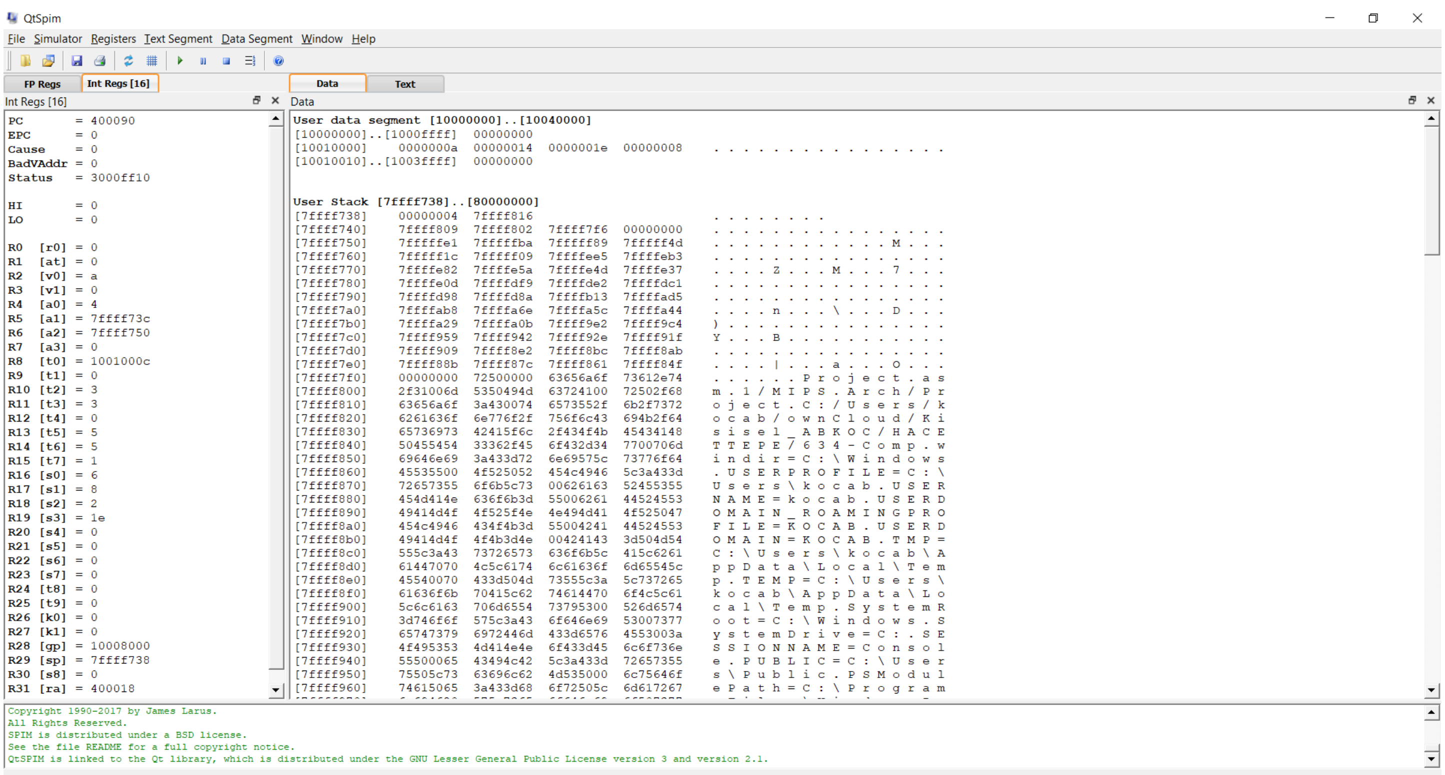

Figure 2: Memory and Register Values (hex) after Test 1: A={2,4,6,8}

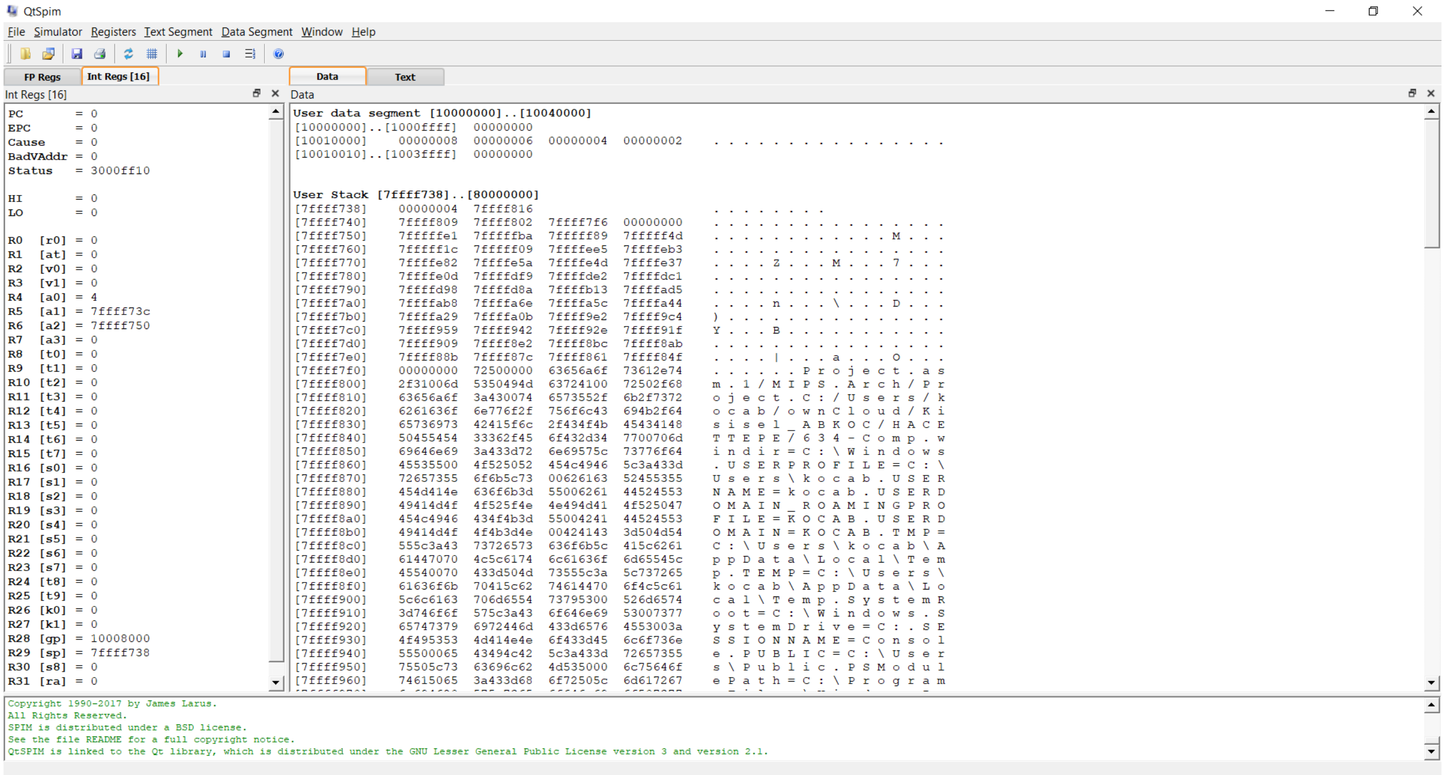

Figure 3: Memory and Register Values (hex) before Test 2: A={8,6,4,2}

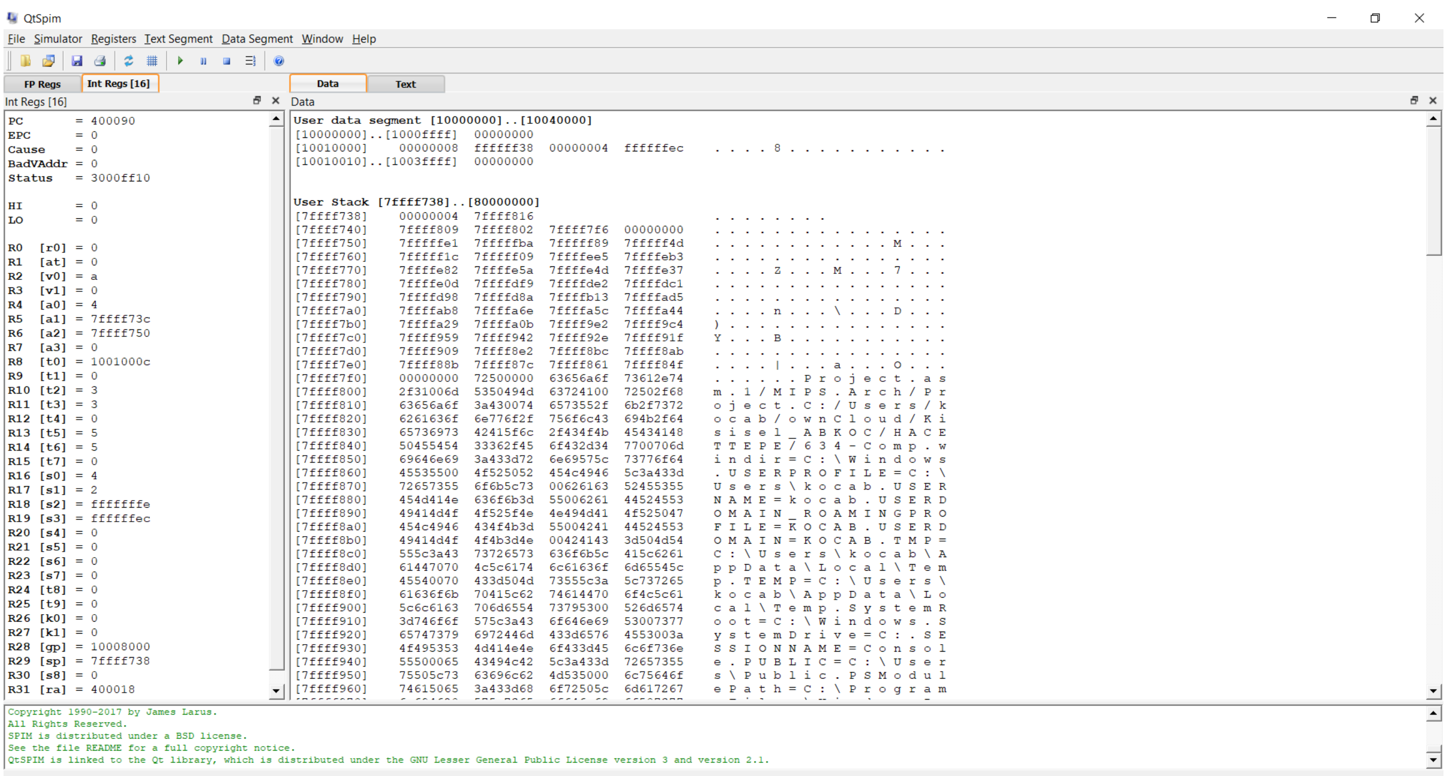

Figure 4: Memory and Register Values (hex) after Test 2: A={8,6,4,2}

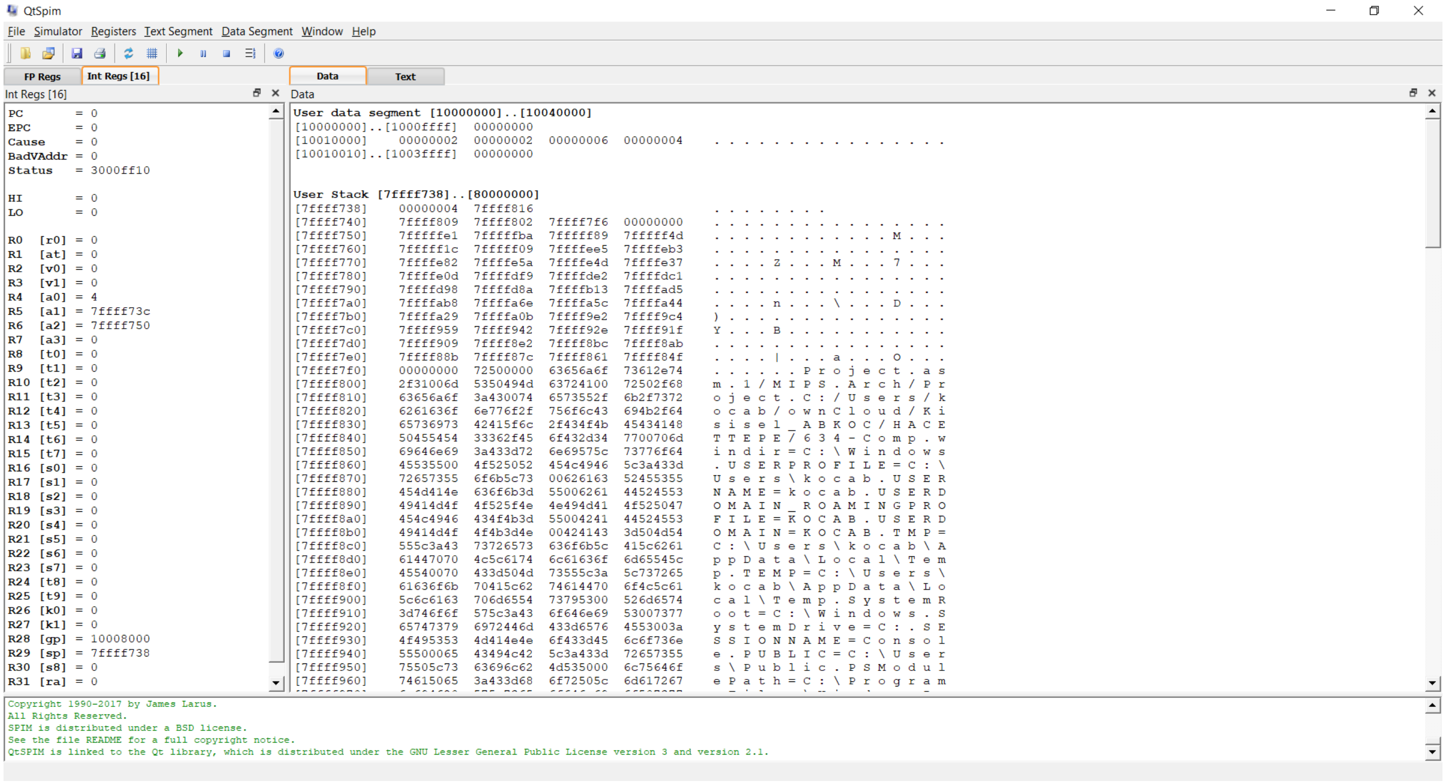

Figure 5: Memory and Register Values (hex) before Test 3: A={2,2,6,4}

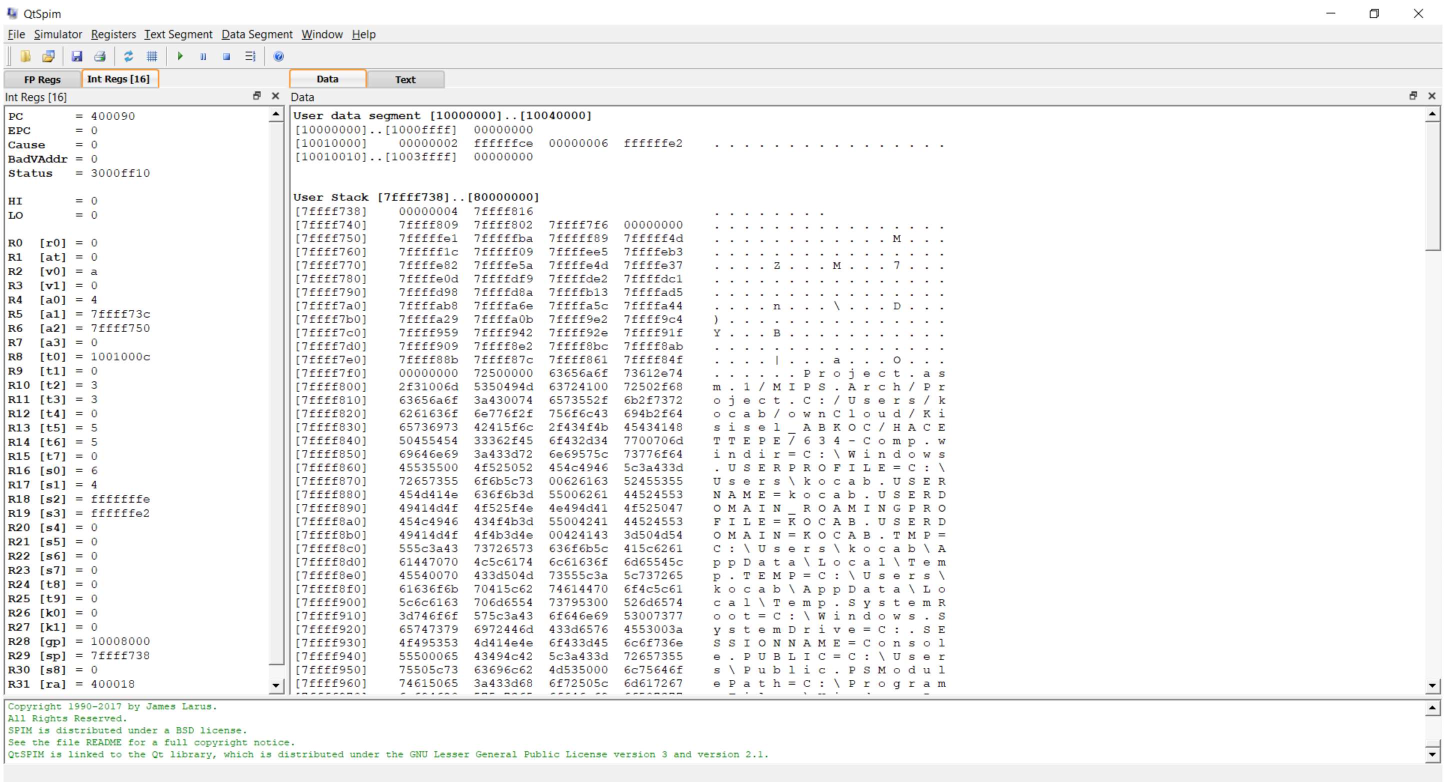

Figure 6: Memory and Register Values (hex) after Test 3: A={2,2,6,4}

2) Function calls

Converting C code fragment to MIPS code is conducted. In MIPS code, multiplication instructions (mult/mul) are allowed to be used.

C CODE FRAGMENT

Below C code fragment is to be converted.

```
int main() {
        int a;
        int b;
        int result = 0;

        if(a == b)
                result = 8*(a + b);
        else
                result = compare(a, b);
        return result;
}

int compare(int a, int b)
{
        if(a<b)
                return punish(a, b);
        else
                return award(a, b);
}

int punish(int a, int b)
{ return (a-b)*2;}

int award(int a, int b)
{ return (a+b)*4;}
```

MIPS CODE

MIPS code is written in modular structure, called functions does not modify saved registers or temporary registers but uses only argument and return registers.

If callee function calls another function (i.e. compare function) it saves function return address register for backup.

For given C code fragment, MIPS code is written as below.

```
.data
##; Input data
A: .word 5
B: .word 3
##; Output Data
Res: .word 0

.text
##      ;Base address registers
#       a_addr          : $t0
#       b_addr          : $t1
#       result_addr     : $t2
##      ;Main function registers
#       a               : $s0
#       b               : $s1
#       result          : $s2
#       arg1            : $a0
#       arg2            : $a1
#       v0              : $v0           ;return value modified after called functions are returned
##      ;Compare function registers
#       slt_compare     : $t3
#       arg1            :               ;defined in main function
```

```
#	arg2		:		;defined in main function
#	ra_mem		: $t4		;return address backup
##	;Punish function registers
#	arg1		:		;defined in main function
#	arg2		:		;defined in main function
#	v0		:		;modified when punish() is called
##	;Award function registers
#	arg1		:		;defined in main function
#	arg2		:		;defined in main function
#	v0		:		;modified when award() is called
#

##; Main function ##
main:
	la $t0, A			# $t0, A base address
	la $t1, B			# $t1, B base address
	la $t2, Res			# $t2, Res base address
	lw $s0, 0($t0)			# take input a
	lw $s1, 0($t1)			# take input b
	addi $s2, $0, 0			# result=0
	bne $s0, $s1, else_main		# jump else_main if a=b is false
	add $s2, $s0, $s1		# result=a+b
	sll $s2, $s2, 3			# result = 8*result (result=8*(a+b))
	j done_main_if			# jump done_main_if
else_main:
	add $a0, $s0, $0		# arg1=a
	add $a1, $s1, $0		# arg2=b
	jal compare			# call compare(), $ra keeps PC address
	add $s2, $v0, $0		# result = compare(a,b)
done_main_if:
	addi $v0, $s2, 0		# return result (v0 contains result)
	sw $v0, 0($t2)			# store return value in data section
	j end				# call program ending function
##; Main function end ##

##; Compare function ## only arguments are used, return adress is backed up
compare:
	slt $t3, $a0, $a1		# slt_compare=1 if a is less than b
	add $t4, $ra, $0		# store $ra in ra_mem
	beq $t3, 0, else_compare	# jump else_compare if condition is false
	jal punish			# call punish()
	add $ra, $t4, $0		# backup $ra from ra_mem
	j $ra				# return caller function
else_compare:
	jal award			# call award()
	add $ra, $t4, $0		# backup $ra from ra_mem
	j $ra				# return caller function
##; Compare function end ##

##; Punish function ## only arguments are used, return register modified
punish:
	sub $v0, $a0, $a1		# return v0=a-b
	sll $v0, $v0, 1			# return v0=2*v0
	j $ra				# return caller function
##; Punish function end ##

##; Award function ## only arguments are used, return register modified
award:
	add $v0, $a0, $a1		# return v0=a+b
	sll $v0, $v0, 2			# return v0=4*v0
	j $ra				# return caller function
##; Award function end ##

end:
   li	$v0, 10				# terminate program run and
   syscall				# Exit
```

TESTS & SCREENSHOTS

For given C code fragment, MIPS code is written as below. The program is tested for the following input values:

Test 1: a=3, b=3

Test 2: a=3, b=5

Test 3: a=5, b=3

For each test, screenshots of the memory before and after running the code are given in Figure 7, Figure 8, Figure 9, Figure 10, Figure 11 and Figure 12. **Values are in Hexadecimal form.**

In test 1 with input values a=3 and b=3, result is 48 in decimal and 00000030 in hexadecimal.

In test 2 with input values a=3 and b=5, result is -4 in decimal and fffffffc in hexadecimal.

In test 3 with input values a=5 and b=3, result is 32 in decimal and 00000020 in hexadecimal.

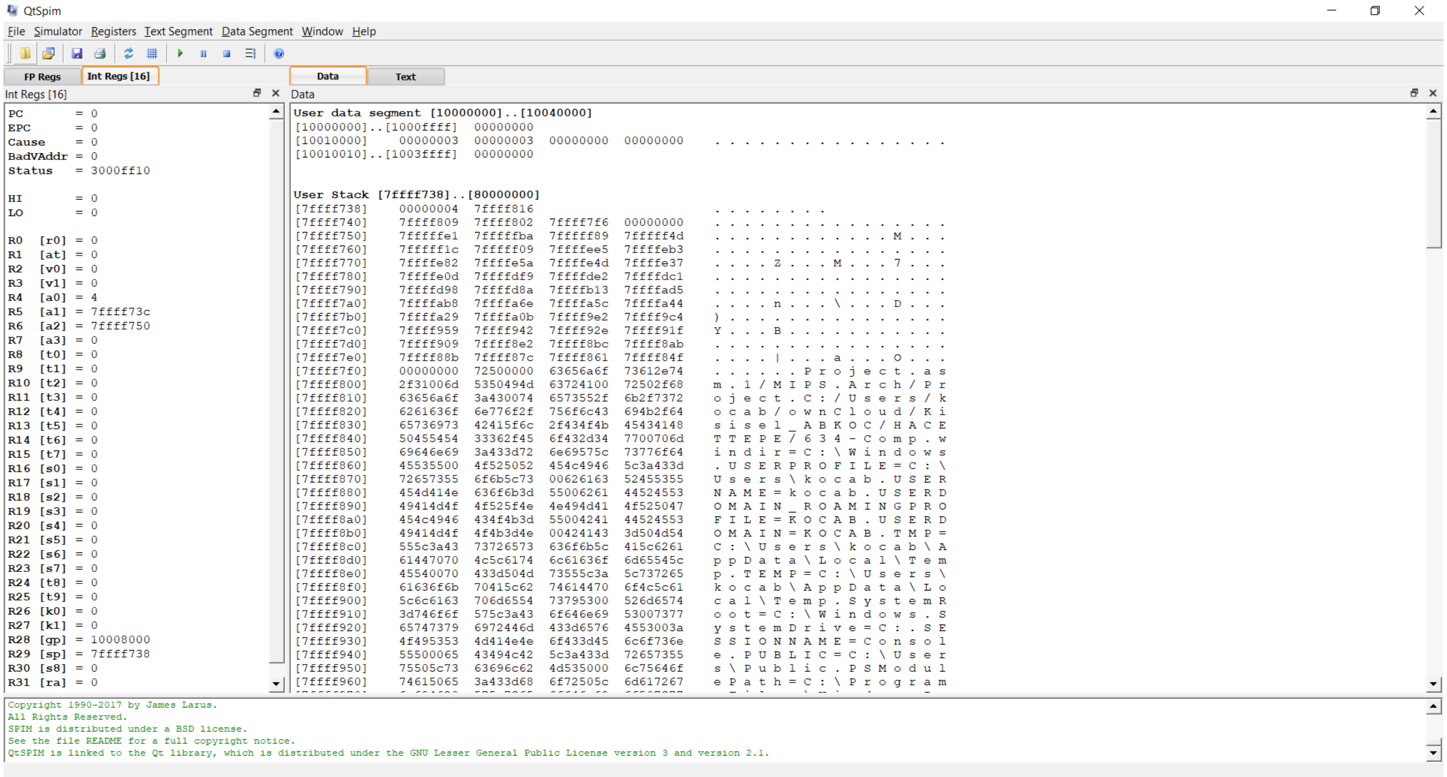

Figure 7: Memory and Register Values (hex) before Test 1: a=3, b=3

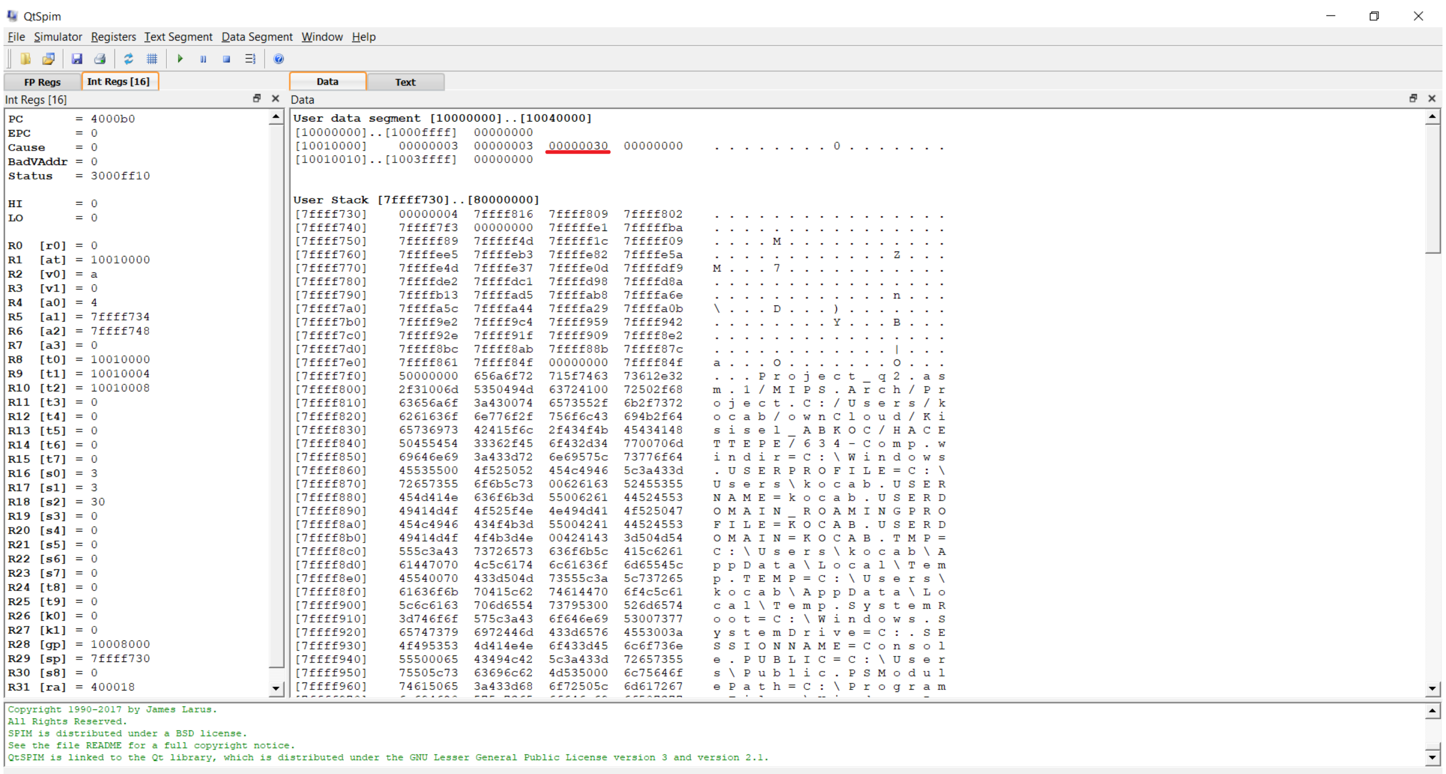

Figure 8: Memory and Register Values (hex) after Test 1: a=3, b=3 (return result in data.res)

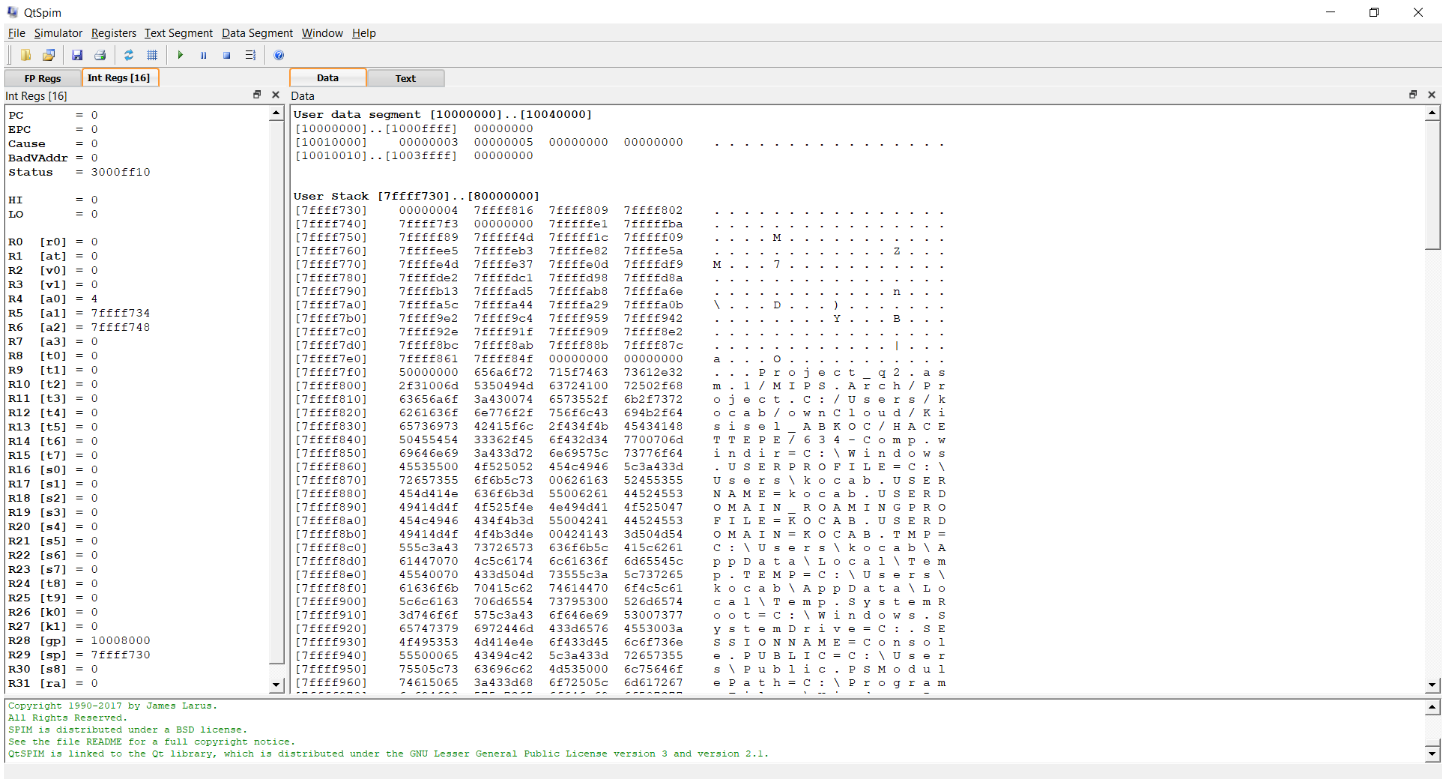

Figure 9: Memory and Register Values (hex) before Test 2: a=3, b=5

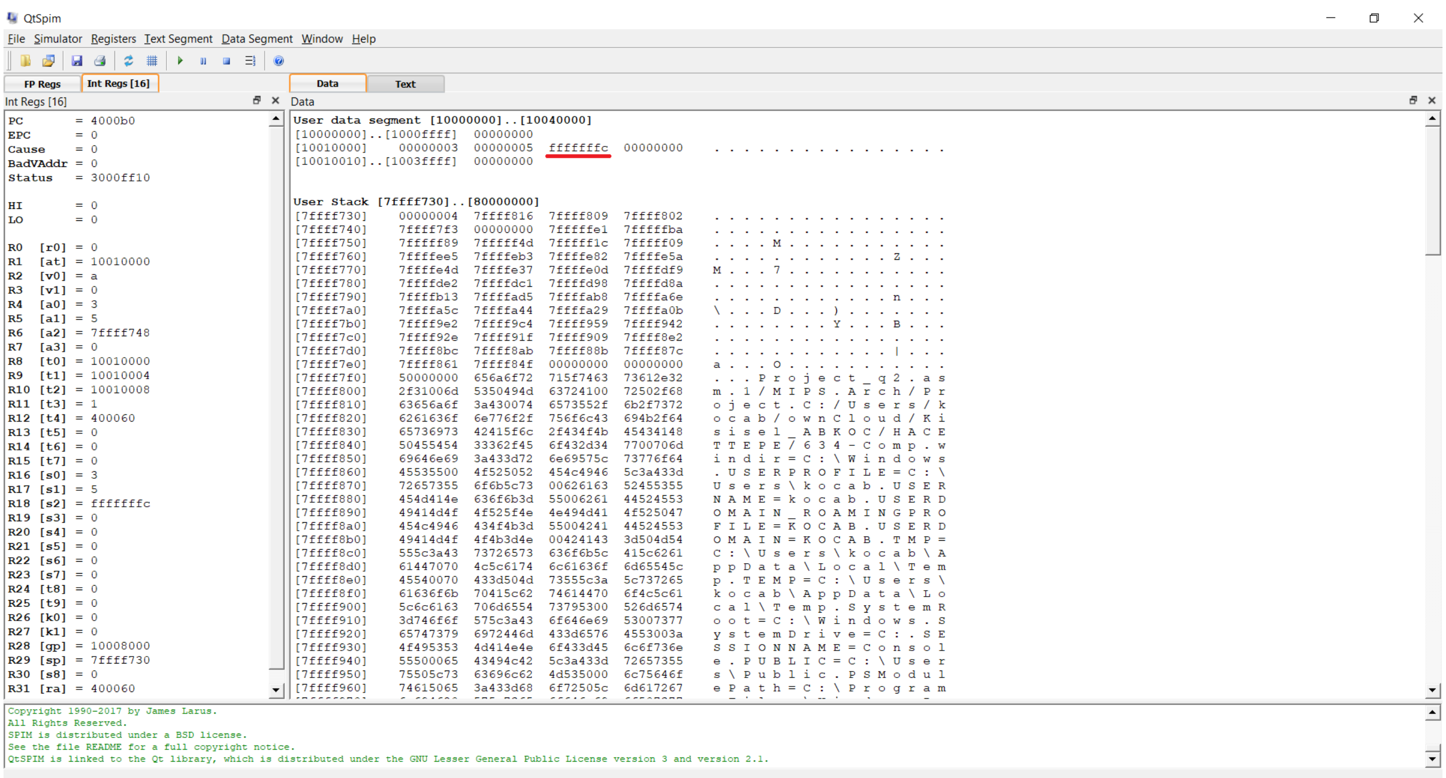

Figure 10: Memory and Register Values (hex) after Test 2: a=3, b=5 (return result in data.res)

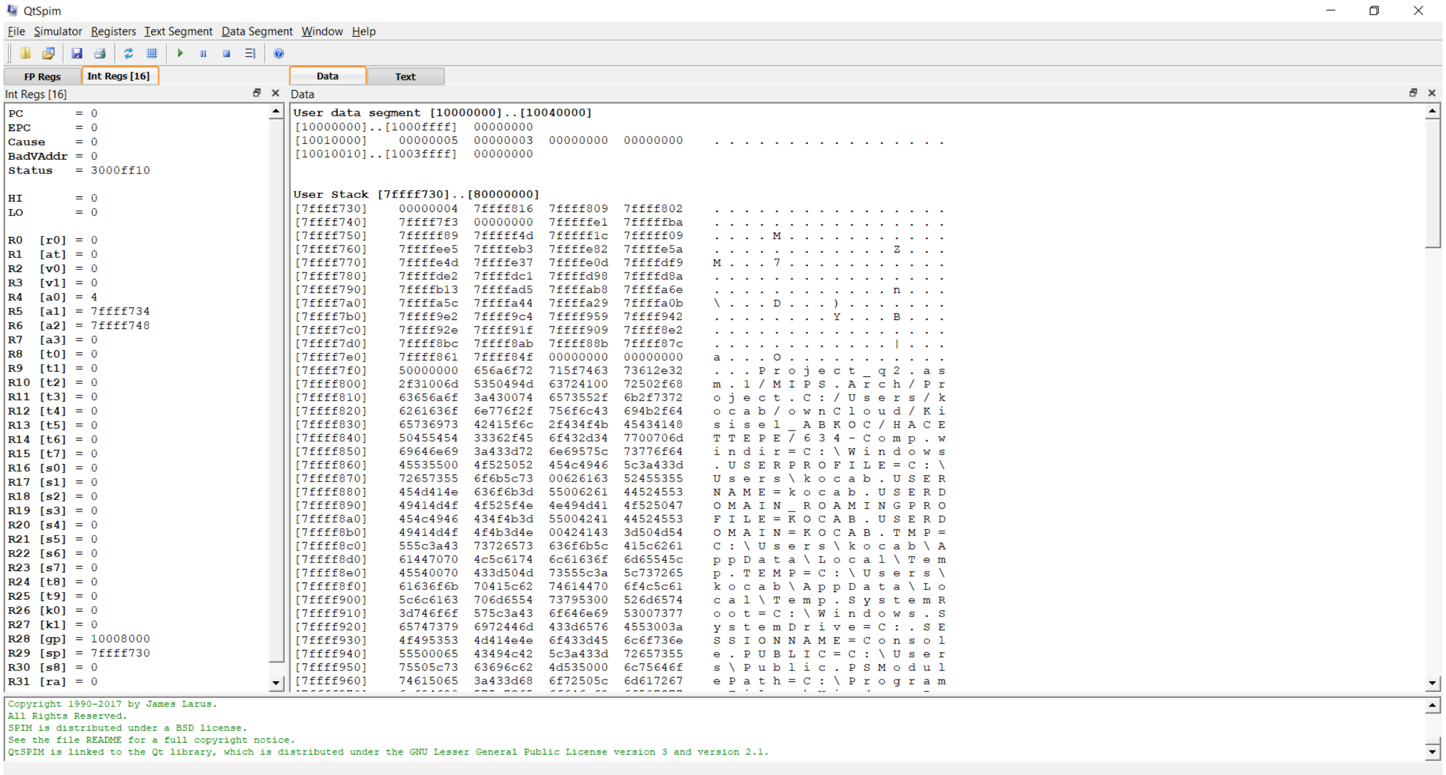

Figure 11: Memory and Register Values (hex) before Test 3: a=5, b=3

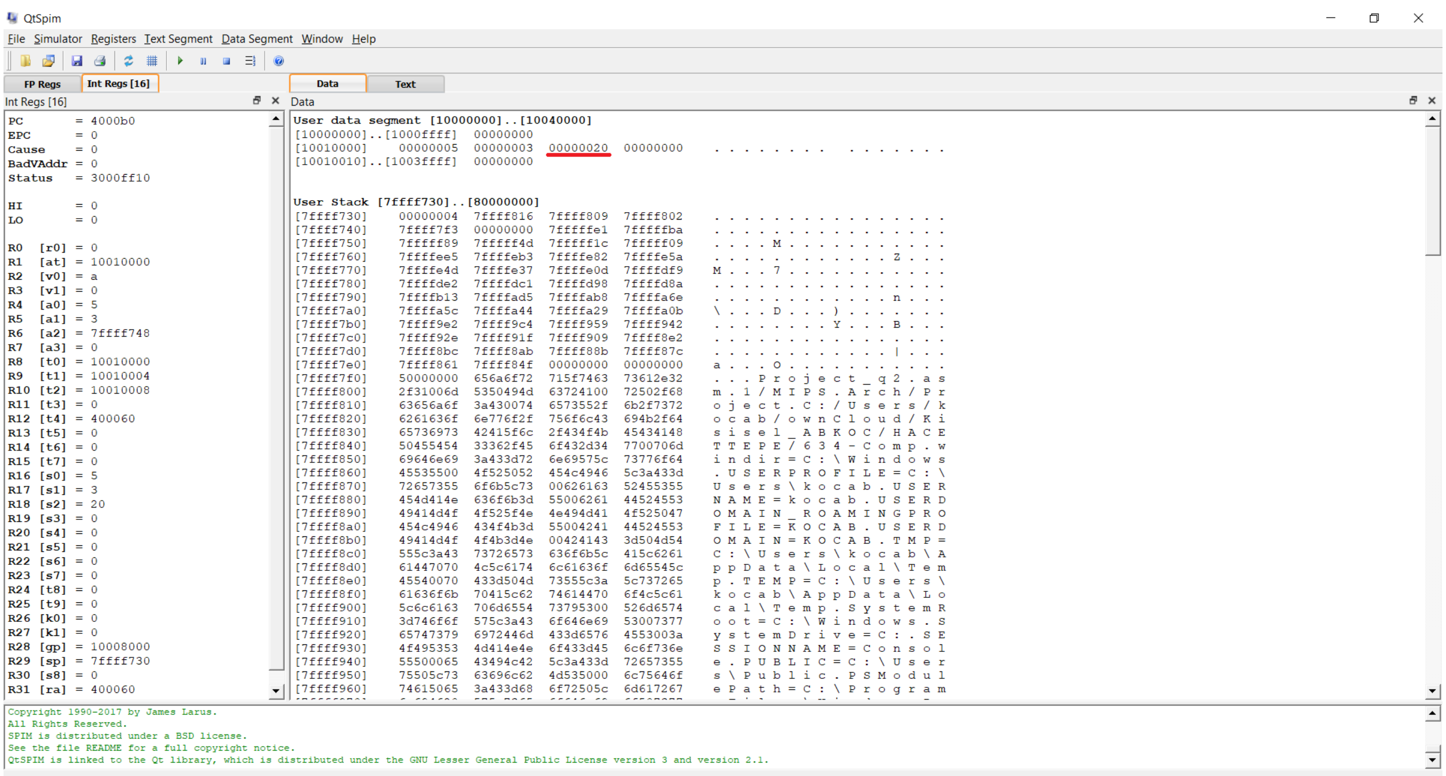

Figure 12: Memory and Register Values (hex) before Test 3: a=5, b=3 (return result in data.res)