# CS232

# LAB 3 REPORT

## QUESTION 1:

### Part a :

Any program executes the main function first. So, let's first go through the instructions of the main section. Starting few instructions of the main function are just storing or loading values to or from memory and modifying values stored in registers. The instructions we need to focus are mainly call instructions.

In the main section, the following call instructions are encountered.

1) <_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
2) <_ZNSt6vectorIiSaIiEEaSESt16initializer_listIiE>
3) <_ZNSirsERi@plt>
4) <srand@plt>

The above functions does not involve any instructions involving secret number

5) <_Z6part_ai>

This function is important when compared to others as this involves getting the secret number. The following instructions are included in this function.

endbr64 : Terminate Indirect Branch in 64 bit

Next 3 instructions involves stack operations,

mov %rsp, %rbp the initial stack pointer address(bottom most address of the stack) is stored in the rbp register.

sub $0x28, %rsp stack pointer is moved up (allocating space in the stack)

Let's denote the value stored in the edx register to be x

mov %edi,-0x24(%rbp) (value in edi is copied into stack at the address -0x24(%rbp) I.e, x is copied into stack at the address -0x24(%rbp))

cmpl $0x1387,-0x24(%rbp) (compares the value stored in the stack at the address -0x24(%rbp) and the value 0x1387 ie, compares x and 0x1387)

jle 1428 <_Z6part_ai+0xbf> ( this is a branch instruction, when x is less than or equal to the value 0x1387 then the executor jumps into <_Z6part_ai+0xbf>, we don't want this to happen. As the current function i.e, <_Z6part_ai> involves secret number implementation we want to stay in this function)

Therefore, we need to ensure that x is greater than 0x1387

0x1387 is the hexadecimal representation of the number 4999.

∴ we need to ensure that x is greater than 4999

lea  0x3c6c(%rip),%rdi     # 5040 <_ZSt4cout@GLIBCXX_3.4>
call   1210 <_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_c@plt>
These above instructions are used to output into the terminal.

The pair of instructions 146a, 1471 outputs the string “------Welcome to Part I!-------” into the
terminal.

The pair of instructions 14d5, 14d6 outputs the string “Enter your roll number:” into the
Terminal.

lea      0x3c71(%rip),%rdi             # 5160 <_ZSt3cin@GLIBCXX_3.4>
call   11a0 <_ZNSirsERi@plt>
These instructions takes input from the terminal

The pair of instructions 14e8,, 14ef takes the roll number input.

The pair of instructions 14fb, 1502 outputs the string “Enter the key to unlock this” into the terminal.

The pair of instructions 150e, 1515 takes the input key.

Before going into the <_Z6part_ai> function, our key is stored in the edx register.
I.e, x = key.
To get the secret number, x should be greater than 4999.
Therefore, key > 4999
∴ to get the secret number, our key should be greater than 4999

Key used: 5000
secret number is: 911284013

## Part b :

Any program executes the main function first. So, let’s first go through the instructions of the main section. Starting few instructions of the main function are just storing or loading values to or from memory and modifying values stored in registers. The instructions we need to focus are mainly call instructions.

In the main section, the following call  instructions are encountered.

1) <_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>,
2) <_ZNSt6vectorIiSaIiEEaSESt16initializer_listIiE>,
3) <_ZNSirsERi@plt>
4) <srand@plt>

The above functions does not involve any instructions involving secret number

5) <_Z6part_biii>

This function is important when compared to others as this involves getting secret number. The following instructions are included in this function.

endbr64 : Terminates Indirect Branch in 64 bit

Next 3 instructions involves stack operations,

mov %rsp, %rbp the initial stack pointer address(last address of the stack) is stored in the rbp register.

sub $0x28, %rsp stack pointer is moved up

let ‘s just denote values stored in the registers edi, esi, edx as x, y, z

mov %edi,-0x24(%rbp) (value of edi is copied into the stack at address -0x24(%rbp) I.e, x is copied into the stack at address -0x24(%rbp))

mov %esi,-0x28(%rbp) (value of esi is copied into the stack at address -0x28(%rbp) ) I.e, y is copied into the stack at address -0x28(%rbp))

mov %edx,-0x2c(%rbp) (value of edx is copied into the stack at address -0x2c(%rbp)) I.e, z is copied into the stack at address -0x2c(%rbp))

mov -0x24(%rbp),%eax (x is copied into eax )

imul %eax,%eax (eax = (eax)^2 i.e, x^2 is stored in eax)

mov %eax,%edx (edx = eax i.e, x^2 is stored in edx )

mov -0x28(%rbp),%eax (y is copied into eax)

imul %eax, %eax ( eax = (eax)^2 i.e, y^2 is stored in eax )

add %eax, %edx (edx = eax + edx i.e, x^2 + y^2 is stored in edx )

mov -0x2c(%rbp),%eax (z is copied into eax)

imul %eax,%eax (eax = (eax)^2 i.e, z^2 is stored in eax )

cmp %eax,%edx (compares values stored in the registers eax, edx)

jne 143f <_Z6part_biii+0xd6> (this is a branch instruction, when the condition ebx not equal to eax is satisfied, then we move into the function <_Z6part_biii+0xd6>, to get the secret number we don’t want this to happen as secret number implementation is part of the current function, we want to stay in the function <_Z6part_biii> ).

Therefore, to get the secret number we must ensure that values stored in eax and edx are equal i.e, x^2+y^2=z^2.

Therefore our key (i.e, 3 integers given as input) should be such that sum of squares of the first two integers is equal to the square of 3rd integer. then only we can get secret number.

Key used: 5 12 13

secret number is: 1670387103

## Part c :

Any program executes the main function first. So, let's first go through the instructions of the main section. Starting few instructions of the main function are just storing or loading values to or from memory and modifying values stored in registers. The instructions we need to focus are mainly call instructions.

In the main section, the following call instructions are encountered.

1) <_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>,
2) <_ZNSt6vectorIiSaIiEEaSESt16initializer_listIiE>,
3) <_ZNSirsERi@plt>
4) <__isoc99_scanf@plt>
5) <srand@plt>

The above functions does not involve any instructions involving secret number

6) <_Z6part_cPc>

This function is important when compared to others as this involves getting the secret number. The following instructions are included in this function.

endbr64 : Terminates Indirect Branch in 64 bit

Next 3 instructions involves stack operations,

mov %rsp, %rbp the initial stack pointer address(last address of the stack) is stored in the rbp register.

sub $0x38, %rsp stack pointer is moved up

Let the values stored in the registers rdi, eax be x,y respectively

mov %rdi, -0x38(%rbp) ( value stored in rdi is copied into the stack at the address -0x38(%rbp) i.e, x is copied into the stack at the address-0x38(%rbp) )

mov -0x38(%rbp), %rax ( value stored in the stack at the address -0x38(%rbp) is stored into the register rax i.e, x is copied into rax )

mov %rax, %rdi (value stored in rax is copied into rdi i.e, x is copied into rdi)

call 11f0 <strlen@plt>

mov %eax, -0x24(%rbp) ( value stored in eax is copied into the stack at the address -0x24(%rbp) i.e, y is copied into the stack at the address-0x24(%rbp) )

cmpl $0x6, -0x24(%rbp) (compares the value 0x6 and the value stored in the stack at the address -0x24(%rbp) i.e, compares value 0x6 and y)

jle 143f <_Z6part_cPc+0x36> ( this is a branch instruction, if the condition y is less than or equal to 0x6 is satisfied then the executor jumps into <_Z6part_cPc+0x36>, we don't want this. As we need our current function <_Z6part_cPc> to continue to get the secret number )

∴ y should be greater than 0x6 ( hexadecimal representation of 6)

I.e, y > 6

cmpl $0xa,-0x24(%rbp) (compares the value 0xa and the value stored in the

stack at the address -0x24(%rbp) i.e, compares value 0xa and y)
jg 143f <_Z6part_cPc+0x36> ( this is a branch instruction, if the condition y is greater than 0xa is satisfied then the executor jumps into <_Z6part_cPc+0x36>, we don't want this. As we need our current function <_Z6part_cPc> to continue to get the secret number )

∴ y should be less than or equal to 0xa ( hexadecimal representation of 10)
I.e, y <= 10

mov -0x24(%rbp), %eax ( value stored in the stack at the address -0x24(%rbp) is stored into the register eax i.e, y is copied into eax )
and $0x1, %eax (stores last bit of eax into eax i.e, if y is even then 0 is stored into eax else if y is odd then 1 is stored into eax )
test %eax, %eax (test checks whether any of the bits of the 2 operands are set at the same position. If the result of the test operation is zero it means that none of the bits in the two operands were set in the same position and the Zero Flag (ZF) in the EFLAGS register will be set. If the test operation result is non zero then it means that at least one bit in the two operands was set in the same position, and the ZF flag will be cleared. )
As we are giving the same operands to the test, the purpose of this instruction is to test if the value in the %eax register is zero or not. If the value is zero, the Zero Flag (ZF) in the EFLAGS register will be set, indicating that the two operands had no bits in common. If the value is non-zero, the ZF flag will be cleared, indicating that the two operands had at least one bit in common.

jne 1457 <_Z6part_cPc+0x4e> (This is a branch instruction, it first checks the state of the ZF flag, If the ZF flag is not set, executor move to <_Z6part_cPc+0x4e>)
We want this jump to happen as the jump skips the instruction 1452 which is a jump to instruction 15a6 <_Z6part_cPc+0x19d> (we want to avoid this jump to get the secret number ) and moves us to the instruction 1457 mov -0x24(%rbp),%eax.
mov -0x24(%rbp),%eax (value stored in the stack at the address -0x24(%rbp) is stored into the register eax i.e, y is copied into eax )
add $0x1, %eax (y+1 is copied into eax)
cltq ( copies the sign bit (bit 31) of the %eax register into all higher-order bits of the %rax register, effectively extending the sign of the 32-bit value to 64 bits )
mov %rax, %rdi (value in rax is copied to rdi)
call 11b0 <_Znam@plt>
mov %rax, -0x20(%rbp) (value in rax is copied to stack address -0x20(%rbp) )
movl $0x0, -0x2c(%rbp) (0 is copied to stack address -0x2c(%rbp))
( address -0x2c(%rbp) is use as loop counter for the loop instruction 1472 to the instruction 14a3 )

Instructions to end loop:
cmp -0x24(%rbp), %eax (compares values at address -0x24(%rbp) and value in eax)
jge 14a5 <_Z6part_cPc+0x9c> (this instruction ends the loop)

Therefore, when value at -0x24(%rbp) < value in eax, the loop ends. And we move to 14a5 instruction (i.e, instruction after loop).

LOOP:

For every run of loop,

First and last bits are compared, if they don't match then loop ends else we continue. In the next run of the loop, the second bit, last but one bits are compared And if they match the loop ends else we continue and this continues till the loop ends.

So we are checking whether the corresponding bits from start, end of the string are the same or not.

The instructions after this loop ensures that if the key is the palindrome then only the instructions involving secret number are not skipped, else the instructions involving secret number are skipped and we can not obtain secret number.

Also, the size of the palindrome should be greater than 6 and less than or equal to 10 and the size of the palindrome should be odd.
Therefore if we enter a palindrome of size 7 or 9 as our key then only we can get the secret number.

Key used: abcdcba
secret number is: 849410635

Final OUTPUT :

CS230{R3v3rse_Engine3ring_is_easy!!}

# QUESTION 2:

I haven't defined any global variables, therefore my .data section in the code is empty. In the .text section, I have defined a recursive function called GCD_Euc, the function has 2 parameters a,m which I have stored in the registers $a1, $a2 respectively.

Idea:

Given A,B(A<B); as their GCD is 1,

There always exist integers x,y such that Ax+By=1

Given A,B; to get x,y following is the procedure:

Eucledian Algorithm: GCD(A,B) = GCD(B%A,A)

∴ GCD(B%A) = 1

∴ There exists x1,y1 such that (B%A)x1+(A)y1=1

Now, using x1,y1 we can get x,y as follows,

B=int(B/A)*A+(B%A)

⇒ (B%A)=B-int(B/A)*A

⇒ (B-int(B/A)*A)x1+(A)y1=1

⇒ A(y1-int(B/A))+B(x1)=1

∴x = y1-int(B/A)

y = x1

∴We can use recursive procedure to get x,y given A,B

Base case of this recursion:

If A=0 then x=0, y=1

Now as 1-ax=my, ax≡1(modm)

Therefore, if we get x we can get inverse of a modulo m

My GCD_Euc function is defined in such a way that it returns x.

In my MIPS code,

In the .main section, after taking inputs a,m from the input.txt files, as I am using $a1, $a2 registers for arguments of GCD_Euc function, I need to store a into $a1, m into $a2 before the instruction jal GCD_Euc to get x for the pair (a,m).

I used $v1 (as register that returns x), $t4(as register that returns y) for given (A,B). These gets update for every function call.

As my function is recursive with 2 arguments, I use stack pointer to store the elements stored in the registers $ra, $a1, $a2 (for a function call stack size 12 bytes).

In the GCD_Euc function, the instruction beq $a1, 0, gcd_base takes care of base case, it tells that if value stored in $a1 is zero then go to gcd_base where $v1 is loaded with 0 (x=0), $t4 is loaded with 1(y=1) and then returns.

To get inverse of a modulo m (which is an integer greater than 0 and less than m) from x, I added instructions that calculates ((x%M)+M)%M (this ensures that inverse a mod m lies in between 0 and m) and stores the result in $v1 before the instruction for print ($v1).

# QUESTION 3:

In the .data section, I have defined the array with space allocation of 40,000 and a variable newline (of type .asciiz, used to print new lines in the output).
.main section:

Register $t0 is used to store base address of the array
We read 1st input from input.txt file which is the size of the array and this is stored in the
register $t1. $s6 is initialised to 0, this is modified in IN_Loop ( 4s6 stores max_elem (we can choose any number greater than the max element(E) of the array as max_elem, I chose 1+E as max_elem ))

In the .text section the following functions are defined:
IN_Loop:

It reads the input from the input.txt file and stores them into the array
For ith(variable i is stored in $t2) input, the following procedure is followed:
Get the address $t0 + ($t2)*4 ($t0 stores base address)
Now store input value to this address using sw instruction

OUT_Loop:

It outputs the elements of the array into the output.txt file
For ith(variable i is stored in $t2) input, the following procedure is followed:
Get the address $t0 + ($t2)*4 ($t0 stores base address)
Load the element stored into this address into $v1
Use syscall print to print this element of the array into output.txt

mergesort:

This is implemented iteratively to ensure that Auxiliary space is of order O(1).
Idea:
Using for loops,
Initially we set size equal to 1,
For a particular size,
We divide the array into sub parts of this size starting from left index and call merge function on pairs of these parts. Parameters for the merge function are left, mid, right (left = starting index of the left array, mid = starting index of the right array, right = ending index of the right array). There is a chance for the size of the last part remained to be less than this particular size. This case is handled as we took mid as minimum among left+size-1, n-1 and right as minimum among left+2*size-1, n-1.
This part is implemented in the Loop3 of the MIPS code.
Now, the above process is carried out for size 1 then 2 then 4 ….. (till size<=n/2).
This loop on sizes is equivalent to the Loop2 of the MIPS code.

In this way iteratively, using divide and conquer(in place) strategy we can implement mergesort function.

$t3 register is used for variable size, $s0,$s1,$s2(parameters for merge function) stores left, mid, right index respectively

Stack pointer(sp) is used to store return address in the stack.

merge:

We define variables curr_index(array sorted(modified) upto this index ), left_index(index of the current pointer of the left array), right_index(index of the current pointer of left array). Registers $a1,$a2,$a3 are used to store left_index, right_index, curr_index respectively

Idea:

We compare elements array[left_index](initial element), array[right_index](initial element). We modify element at array[curr_index] using max_elem in such a way that we can get both, the initial element and the element which is to be present at that position for the array to get sorted, using this modified value.

For this we follow the following procedure:

(Note that elements upto curr_index are modified)

Suppose, array[left_index](initial element) < array[right_index](initial element)

array[curr_index] = array[curr_index](initial element) +
(array[left_index](sorted array element)%max_elem)*max_elem

Getting initial value,

array[curr_index]%max_elem
=(array[curr_index](initial value) + (array[left_index]%max_elem)*max_elem)%max_elem
=(initial value)%max_elem
= initial value (since, initial values are less than max_elem)

∴ initial value = array[curr_index]%max_elem

Getting Element(sorted array),

array[curr_index]/max_elem
=array[curr_index](initial value) +
(array[left_index](sorted array element)%max_elem)*max_elem)/max_elem
=array[curr_index](initial value)/max_elem (= 0) + (sorted array element)%max_elem
= (sorted array element)%max_elem
= sorted array element (as the elements of the array are less than max_elem)

∴sorted array element = array[curr_index]/max_elem

If left_index <= mid(last index of the left array) and right_index <= end(last index of the right array) then we compare and modify the array as explained above.
This part is implemented in the Loop4 of the MIPS code. For accessing array elements and for modifying array, lw, sw instructions are used respectively.
Else if left_index <= mid(last index of the left array), no need to compare we modify array such that sorted array elements are same as elements of left array.

This part is implemented in the Loop5 of the MIPS code.
Else if right_index <= end(last index of the right array), no need to compare we modify array such that sorted array elements are same as elements of right array.
This part is implemented in the Loop6 of the MIPS code.

Now before returning for every index of the array, we need to get the sorted array element from each modified array element and store this sorted array element at this index.
This part is implemented  in the Loop7 of the MIPS code.
Stack pointer(sp) is used to store return address in the stack.

## QUESTION 4:

Testbench:

I used mmap for allocating matrices into memory.
the following instructions are followed:
(memory allocation for 1st matrix)

```
imul rcx, rax
 imul rcx, 8
 mov rsi, rcx
 mov r10, 34
 mov rdi, 0
 mov rax, 9
 mov rdx, 3
 syscall

 mov [a1], rax
```

After the syscall,the base pointer of the row-major array corresponding to  matrix1 is stored in rax. Now the last instruction mov [a1], rax stores rax into [a1]
the same procedure is followed for matrix2 and matrix3.

Matrix Multiplication:
For {ijk},{ikj},{jki},{jik},{kji},{kij}:
  I used r11, r12, r13 as counters for the loops involving variables i,j,k respectively.
  As r12 and r9 are the same, I can modify r12.
For {ijk}
3 loops ( Loop1, Loop2, Loop3 ) are used for the matrix multiplication, for the loops involving variables i,j,k the registers r11, r12, r13 are set as counters respectively.
For each loop, before the loop starts, set the corresponding counter to 0.
For ijk,  before Loop1, r11 is set to 0, (instruction: mov r11, 0), before Loop2, r12 is set to 0, (instruction: mov r12, 0), before Loop3, r13 is set to 0, (instruction: mov r13, 0)
For Loop1,
At the end of every run we increase the counter, (instruction: add r11, 1) and then compare the counter with the limit( size of row1 which is stored in rsi) (instruction: cmp r11, rsi)
Now, if these are not equal the loop continues i.e, we jump to the Loop1 else the loop ends.
For Loop2,
At the end of every run we increase the counter, (instruction: add r12, 1) and then compare the counter with the limit( size of coloumn2 which is stored in r9) (instruction: cmp r12, r9)
Now, if these are not equal the loop continues i.e, we jump to the Loop1 else the loop ends.
For Loop3,

For every run,

we first get the address `rdi+(rdx*r11+r13)*8` as `mat1[i][k]` is stored at this address, using the following set of instructions:

```
mov rax, r8
mul r11
add rax, r13
mov r14, 8
mul r14
add rax, rdi
mov rbx, [rax]
```

Now rax stores the address `rdi+(rdx*r11+r13)*8`, now we load the value at this address into rbx register.

∴ `rbx = mat1[i][k]`

Then we get the address `rcx+(r9*r13+r12)*8` as `mat2[k][j]` is stored at this address, using the following set of instructions:

```
mov rax, r9
mul r13
add rax, r12
mov r14, 8
mul r14
add rax, rcx
mov r14, [rax]
```

rax stores the address `rcx+(r9*r13+r12)*8`, now we load the value at this address into r14 register.

∴ `r14 = mat2[k][j]`

This is followed by the instructions:

```
mov rax, rbx
mul r14
mov rbx, rax
```

1st instruction: `rax = mat1[i][k]`

2nd instruction: `rax = mat1[i][k]*r14 = mat1[i][k]* mat2[k][j]`

3rd instruction: `rbx = rax = mat1[i][k]*r14 = mat1[i][k]* mat2[k][j]`

Now, we get the address `r10+(r9*r11+r12)*8` as `mat3[i][j]` is at this address, using the following set of instructions:

```
mov rbx, rax
mov rax, r9
mul r11
add rax, r12
mov r14, 8
mul r14
add rax, r10
mov r14, [rax]
```

rax stores the address `r10+(r9*r11+r12)*8`. We load the value at this address into r14.

∴ `r14 = mat3[i][j]`

This is followed by the instructions:

```
add r14, rbx
mov [rax], r14
```

1st instruction: `r14 = r14 + rbx`

I.e, `r14 = mat3[i][j] + mat1[i][k]*r14 = mat1[i][k]* mat2[k][j]`

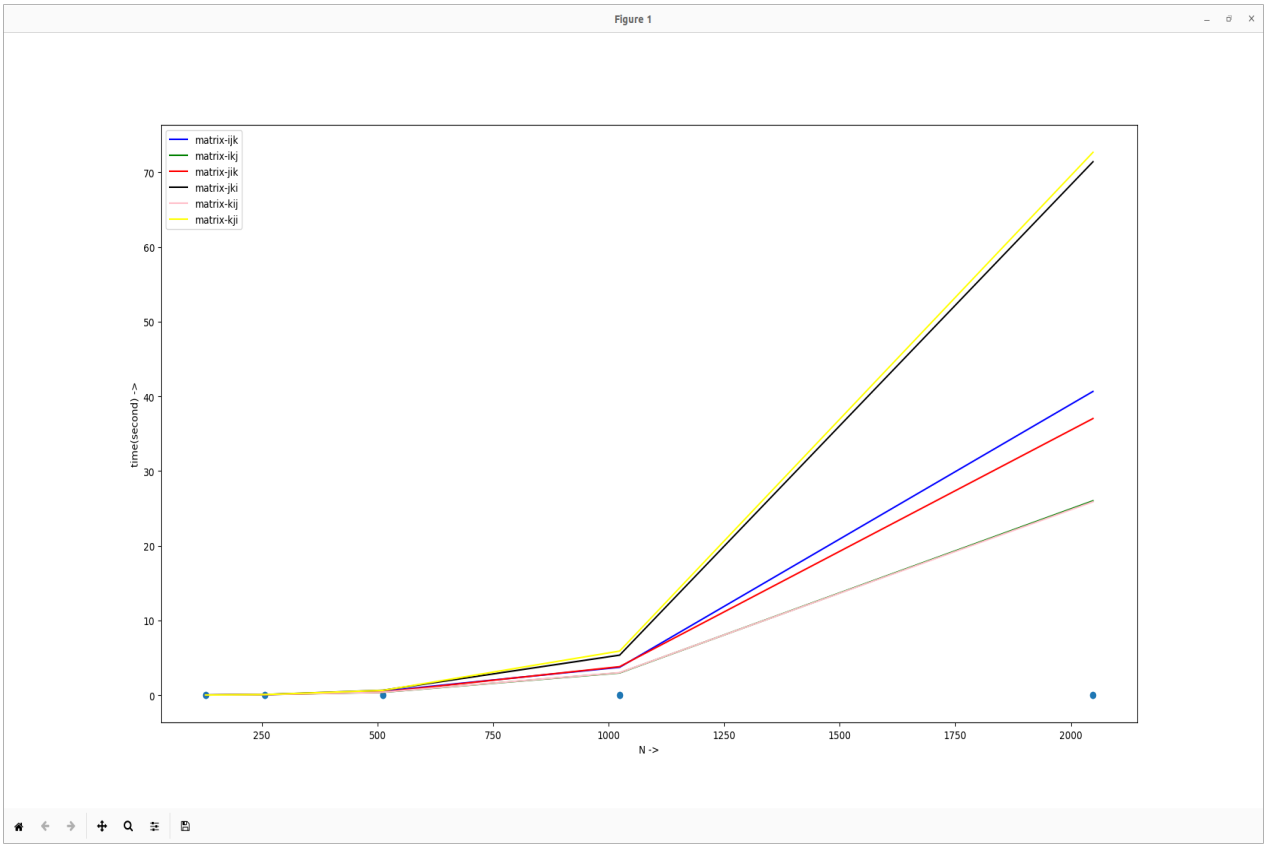
2nd instruction: stores r14 into the address rax.

Now, `mat3[i][k]` got updated.

At the end of every run we increase the counter, (instruction: add r13, 1) and then compare the counter with the limit( size of coloumn1 which is stored in r8) (instruction: cmp r13, r8) Now, if these are not equal the loop continues i.e, we jump to the Loop1 else the loop ends.

The same procedure is followed for the cases {ikj},{jki},{jik},{kji},{kij}

Figure:



TSC Frequency: 3293.848 MHz

Size: 128

| variant | cycles | time(in s) |
|---|---|---|
| ijk | 26702326 | 8.10672684e-03 |
| ikj | 26272250 | 7.97615737e-03 |
| jik | 27469345 | 8.33959096e-03 |
| jki | 27609271 | 8.38207197e-03 |
| kij | 26511081 | 8.04866557e-03 |
| kji | 27032629 | 8.20700561e-03 |

Size: 256

| variant | cycles | time(in s) |
|---|---|---|
| ijk | 219903281 | 6.67618181e-02 |
| ikj | 208199795 | 6.32086833e-02 |
| jik | 218712859 | 6.64004104e-02 |
| jki | 225401609 | 6.84310900e-02 |
| kij | 212587273 | 6.45407053e-02 |
| kji | 230402181 | 6.99492451e-02 |

Size: 512

| variant | cycles | time(in s) |
|---|---|---|
| ijk | 1730865714 | 5.25484392e-01 |
| ikj | 1238551231 | 3.76019546e-01 |
| jik | 1353046341 | 4.10779836e-01 |
| jki | 2124099252 | 6.44868631e-01 |
| kij | 1279894605 | 3.88571241e-01 |

| kji | 2071435868 | 6.28880224e-01 |
|---|---|---|

Size: 1024

| variant | cycles | time(in s) |
|---|---|---|
| ijk | 12274399377 | 3.72646199e+00 |
| ikj | 9787629512 | 2.97148791e+00 |
| jik | 12537412318 | 3.80631174e+00 |
| jki | 17622868771 | 5.35023740e+00 |
| kij | 9891817765 | 3.00311908e+00 |
| kji | 19417115305 | 5.89496398e+00 |

Size: 2048

| variant | cycles | time(in s) |
|---|---|---|
| ijk | 133935474215 | 4.06623117e+01 |
| ikj | 85674444243 | 2.60104426e+01 |
| jik | 121977766392 | 3.70319961e+01 |
| jki | 235235912778 | 7.14167481e+01 |
| kij | 85274488226 | 2.58890174e+01 |
| kji | 239400094200 | 7.26809781e+01 |