

Take Home Test 2:

Factorial

Shirong Zheng
Professor Izidor Gertner
March 22, 2017
CSC34200-G Spring 2017

Contents

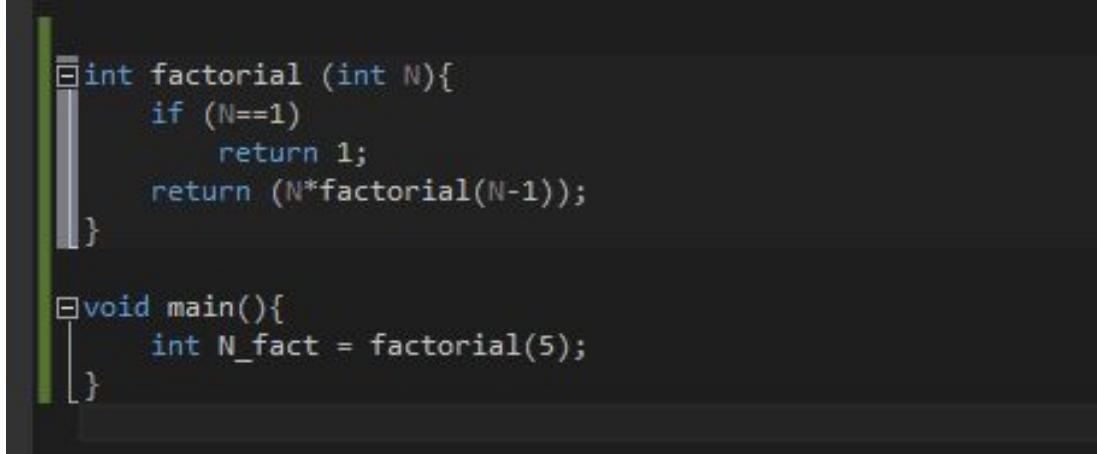
1. Objective	3
2. Part I Analysis In x86 Intel Of Microsoft's Visual Studio	4
3. Part II Analysis In MIPS on MARS Simulator	18
4. Part III Analysis In 64-bit Intel Processor gdb/gcc On Linux	24
5. The Factorial Running Time	31
6. Conclusions	32

❖ 1. Objective

The objective of take home test 2 is going to using the assembly code and C++ code which provide by the professor to analysis in MIPS (assembly compiler which use MARS based on the Java JDK), Microsoft Visual Studio in 32 bit Window(debugger analysis) and GDB/GCC in 64 bit LINUX compiler. This topic of lab is factorial. That require to plot the time of it takes to compute Factorial(N), for N=10,100,1000,10,000.

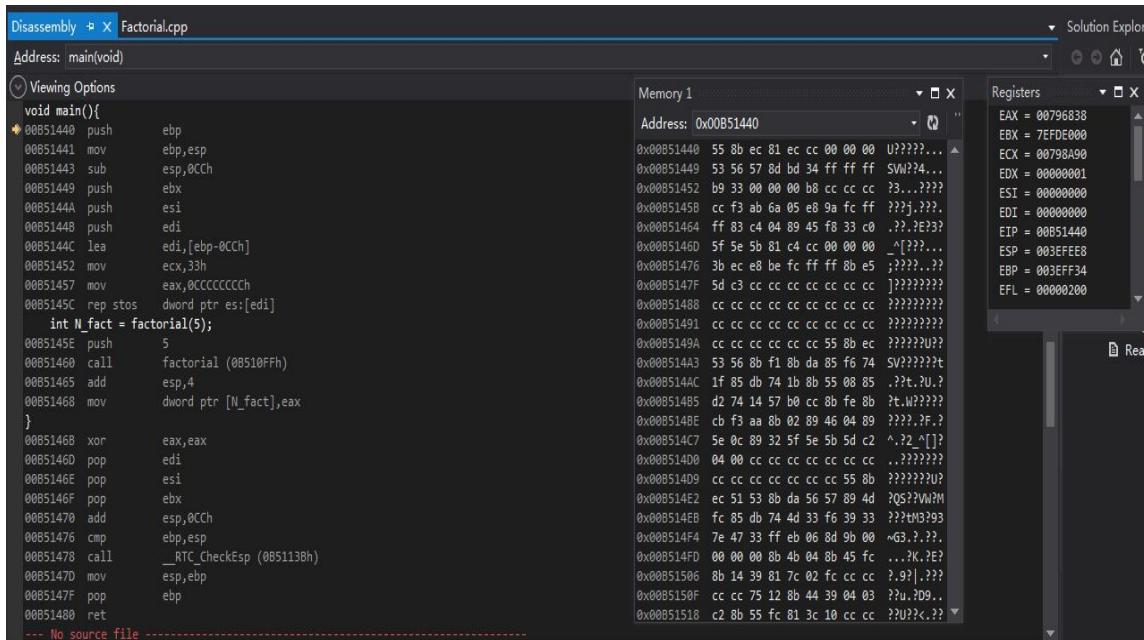
❖ 2. Part I Analysis In x86 Intel Of Microsoft's Visual Studio

The below is my factorial c++ code and debug in the Visual Studio at Windows.



```
int factorial (int N){  
    if (N==1)  
        return 1;  
    return (N*factorial(N-1));  
  
}  
  
void main(){  
    int N_fact = factorial(5);  
}
```

First we need to take look the ebp in the push process. It's EIP is 0x00B51440. The EIP register always contains the address of the next instruction to be executed. You cannot directly access or change the instruction pointer. However, instructions that control program flow, such as calls, jumps, loops, and interrupts, automatically change the instruction pointer.



Then we could know the basic memory value is 55 8b ec 81 ec cc 00 00 00. The below memory values also contain other process when the system debug this code in main function. It will lead us to the initial factorial way. That will tell the system to save the value of EBP.

The screenshot shows two panes. The left pane displays assembly code for the `main()` function:

```

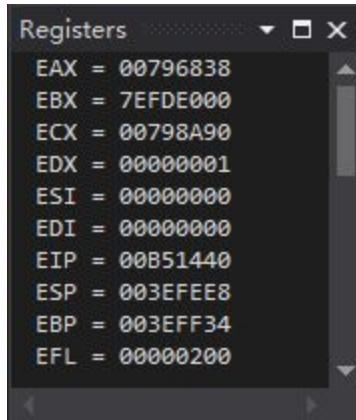
void main(){
    push    ebp
    mov     ebp,esp
    sub    esp,0CCh
    push    ebx
    push    esi
    push    edi
    lea     edi,[ebp-0CCh]
    mov     ecx,33h
    mov     eax,0CCCCCCCCh
    rep stos dword ptr es:[edi]
}

```

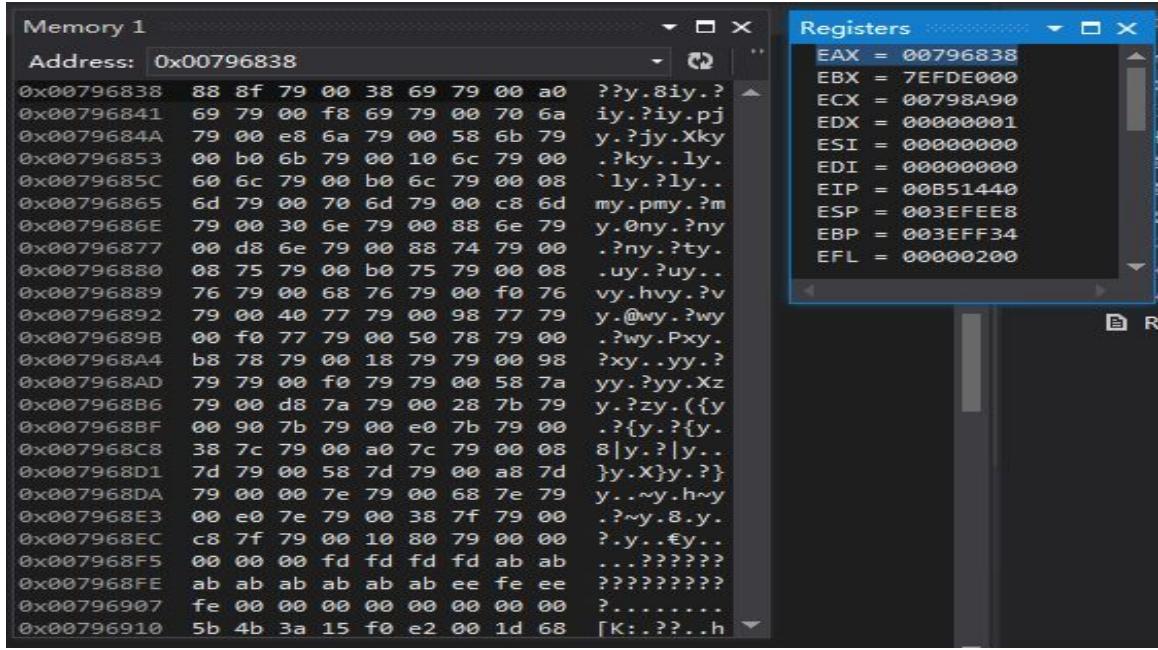
The right pane shows a memory dump starting at address 0x00B51440. The first few bytes are 55 8b ec 81 ec cc 00 00 00, followed by several bytes of question marks and underscores.

Address	Value	Content
0x00B51440	55 8b ec 81 ec cc 00 00 00	U?????...
0x00B51441	53 56 57 8d bd 34 ff ff ff	SW??4...
0x00B51452	b9 33 00 00 00 b8 cc cc cc	?3...???
0x00B5145B	cc f3 ab 6a 05 e8 9a fc ff	???j.???
0x00B51464	ff 83 c4 04 89 45 f8 33 c0	.???.?E?3?
0x00B5146D	5f 5e 5b 81 c4 cc 00 00 00	_^[???
0x00B51476	3b ec e8 be fc ff ff 8b e5	;????..??
0x00B5147F	5d c3 cc cc cc cc cc cc cc]?????????
0x00B51488	cc cc cc cc cc cc cc cc cc	???????????
0x00B51491	cc cc cc cc cc cc cc cc cc	???????????
0x00B5149A	cc cc cc cc cc 55 8b ec	??????U??

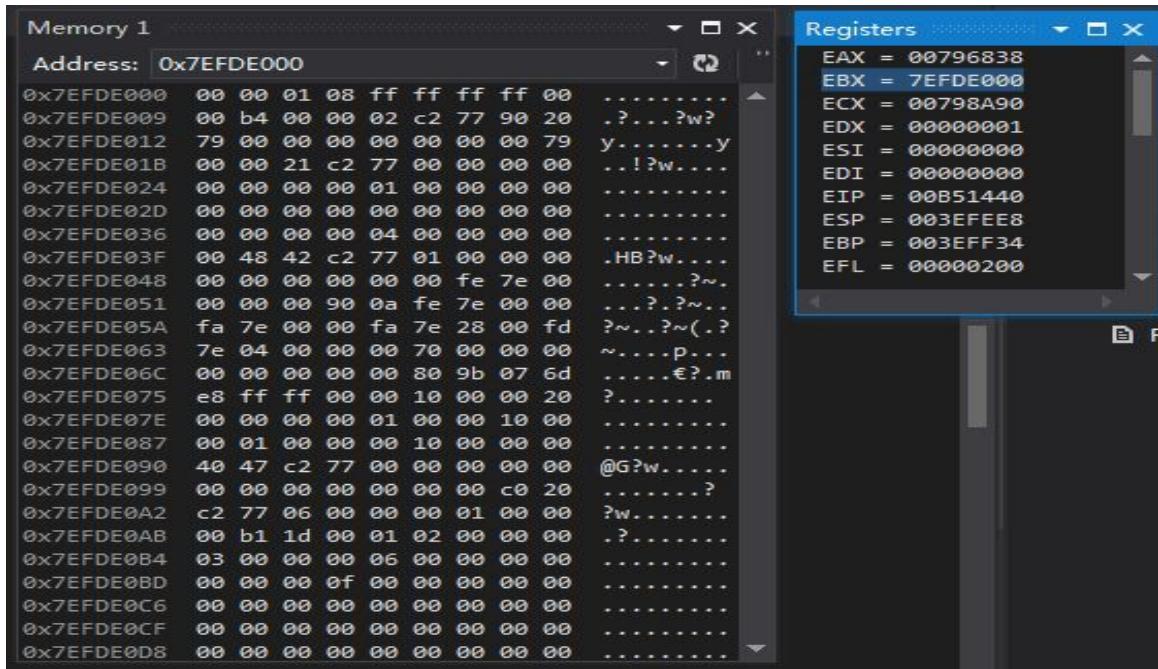
The below is the basic registers address of initial step. The EDI(Electronic Data Interchange) and the ESI(Extended Source Index) are point to 0 memory address.



The memory address of EAX is 0x00796838. Its memory value is 88 8f 79 00 38 69 79 00 a0. EAX used to be called the accumulator since it was used by a number of arithmetic operations while the system operate the math formula.



In this register, the EBX is 0x7EFDE000, the memory values are 00 00 01 08 ff ff ff ff. It will has 32 bit in the register memory. It defined as the base register, that the The base address is stored in memory addressing.



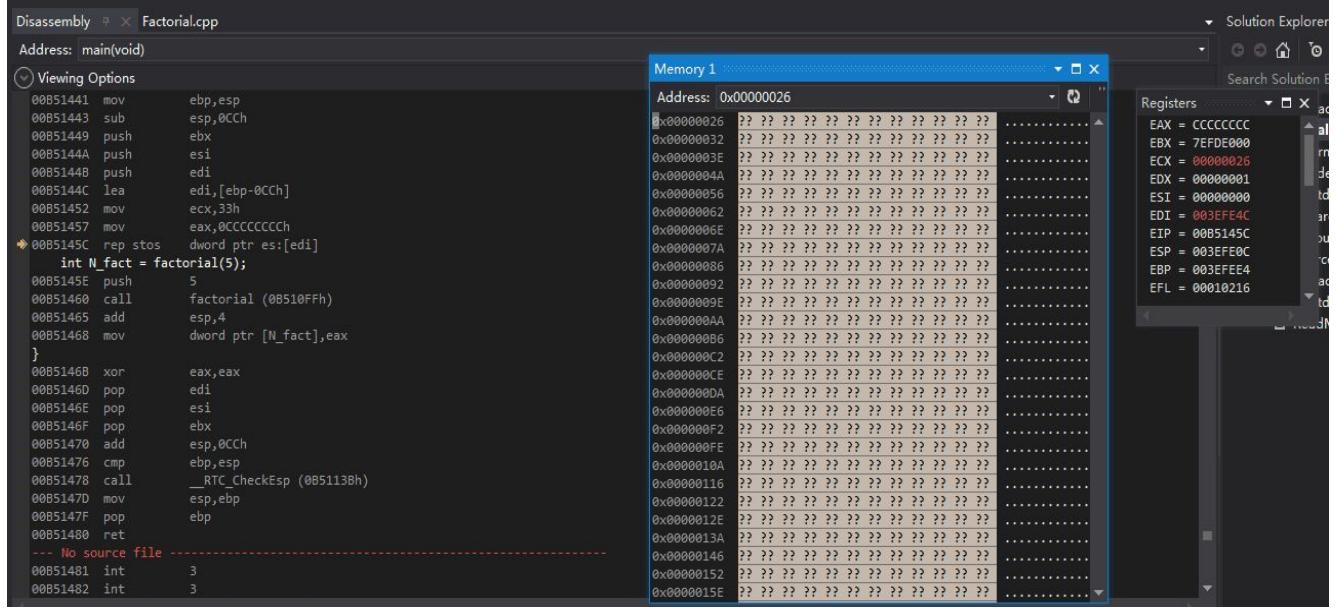
Next, we take look for the sub process. The ESP register which is the current stack pointer. EBP is the base pointer for the current stack frame. When you call a function, typically space is reserved on the stack for local variables. This space is usually referenced via EBP (all local variables and function parameters are a known constant offset from this register for the duration of the function call.) ESP, on the other hand, will change during the function call as other functions are called, or as temporary stack space is used for partial operation results. The register of ESP is 0x003EFEE4, the register of EBP is 0x003EFEE4. In this line, the “sub esp, 0CCh” means space allocated on the stack for the local variables, esb=esp-0CCH.

The screenshot shows the Microsoft Visual Studio debugger interface with the following panes:

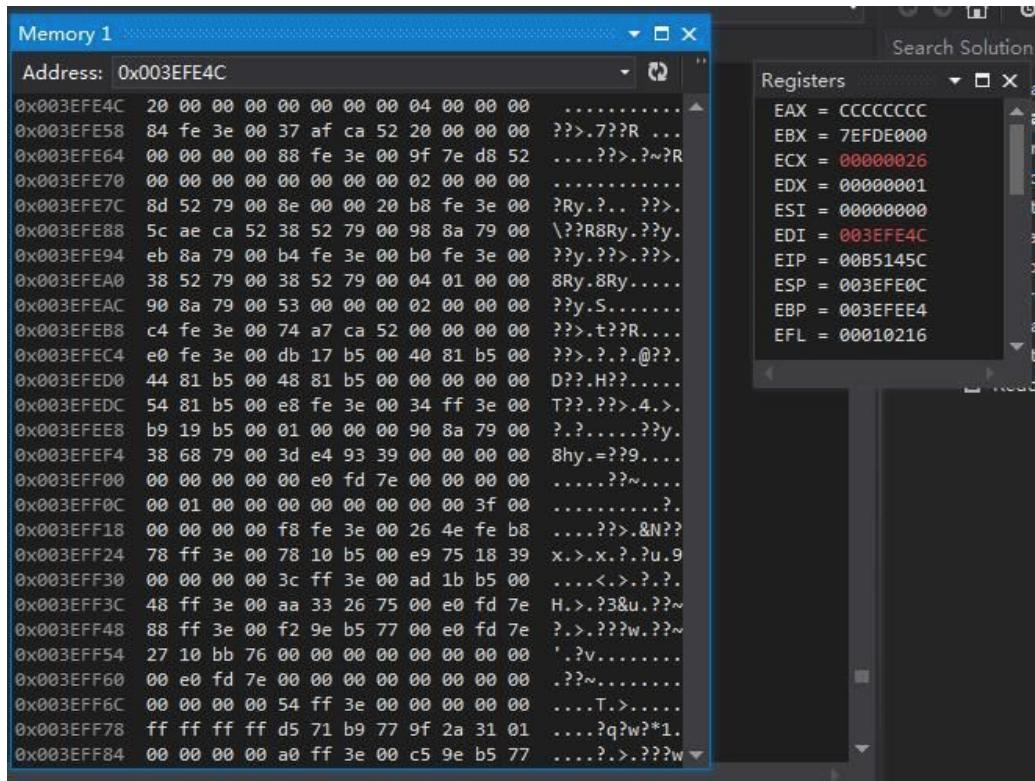
- Disassembly**: Shows the assembly code for the main() function. Key instructions include:
 - push ebp
 - mov esp,ebp
 - sub esp,0CCh
 - push ebx
 - push esi
 - push edi
 - lea edi,[ebp-0CCh]
 - mov ecx,33h
 - mov eax,0CCCCCCCCh
 - rep stos dword ptr es:[edi]
 - int N_fact = factorial(5);
 - push 5
 - call factorial (0B510FFh)
 - add esp,4
 - mov dword ptr [N_fact],eax
 - ret
- Registers**: Shows the current state of CPU registers:
 - EAX = 00796838
 - EBX = 7EFD0000
 - ECX = 00798A90
 - EDX = 00000001
 - ESI = 00000000
 - EDI = 00000000
 - EIP = 00B51443
 - ESP = 003EFEE4
 - EBP = 003EFEE4
 - EFL = 00000202
- Memory 1**: A dump of memory starting at address 0x00B51443. It shows a series of bytes, many of which are question marks or hex values like FF, FF, and 00. The memory dump starts with:


```
0x00B51443 81 ec cc 00 00 00 53 56 57 8d bd 34 ???.SW??.4
0x00B5144F ff ff ff b9 33 00 00 00 b8 cc cc cc ...??.???
0x00B5145B cc f3 ab 6a 05 e8 9a fc ff ff 83 c4 ???.j.???.??
0x00B51467 04 89 45 f8 33 c0 5f 5e 5b 81 c4 cc .?E??.^.[??
0x00B51473 00 00 00 3b ec e8 be fc ff ff 8b e5 ...;????..??
0x00B5147F 5d c3 cc ]??????????
0x00B5148B cc ???????????
0x00B51497 cc 55 8b ec ??????U??
0x00B514A3 53 56 8b f1 8b da 85 f6 74 1f 85 db SV?????t.???
0x00B514A7 74 1b 8b 55 08 85 d2 74 14 57 b0 cc t.?U.?t.W??
0x00B514B8 8b fe 8b cb f3 aa 8b 02 89 46 04 89 ??????.?F.?
0x00B514C7 5e 0c 89 32 5f 5e 5b 5d c2 04 00 cc ^.?.^.[]?.??
0x00B514D3 cc ???????????
0x00B514D9 cc 55 8b ec 51 53 8b da 56 57 89 4d ?U?S?W?M
0x00B514EB fc 85 db 74 4d 33 f6 39 33 7e 47 33 ???tM?93~G3
0x00B514F7 ff eb 8d 9b 00 00 00 00 8b 4b 04 .?.??.?K.
0x00B51503 8b 45 fc 8b 14 39 81 7c 02 fc cc cc ?E??.9?;.???
0x00B5150F cc cc 75 12 8b 44 39 04 03 c2 8b 55 ?U.?D?..?U
0x00B5151B fc 83 3c 10 cc cc cc cc 74 10 ff 74 ??.??.??.t.
0x00B51527 39 08 8b 45 04 50 e8 a5 fb ff ff 83 9.?E.P??.??
0x00B51533 c4 08 46 83 c7 03 b3 33 7c c3 b8 75 ?.F??.;3??
0x00B5153F 08 33 ff 8b c6 85 f6 74 60 8b 40 04 .?.??.?t?@.
0x00B51548 47 85 c8 75 f8 85 f6 74 54 81 3e cc G??u??.t?>?
0x00B51557 cc cc cc 75 1b 81 7e 14 cc cc cc cc ?U.?~.???.u.?
0x00B51563 75 12 81 7e 18 cc cc cc cc 75 09 81 u.?~.???.u.?
0x00B5156F 7e 1c cc cc cc cc 74 0e 8b 45 04 57 ~.???.t.?E.W
0x00B5157B 56 50 e8 3b fc ff ff 83 c4 0c 8b 46 VP?;?.??.?F?
```

The EIP is 0x00B5145C, in this step, it will repeat the command from previous. The memory values of ECX is repeat times. The memory register is 0x00000026, that shows lots of question mark in Memory window. In other words, we could say we don't know the times of repeat in factorial function.



The memory register of EDI is destination index register. This instruction copies the contents of Ax into the memory location whose address is the sum of the Bx and SI. DI stands for destination index, used as a pointer to the current character being written or compared in a string instruction. It is also available as an offset just like SI. The memory register is 0x003EFE4C, and it has memory values are 20 00 00 00 00 00 00 00 04 00 00 00. Then we could know the target string will be much smaller.



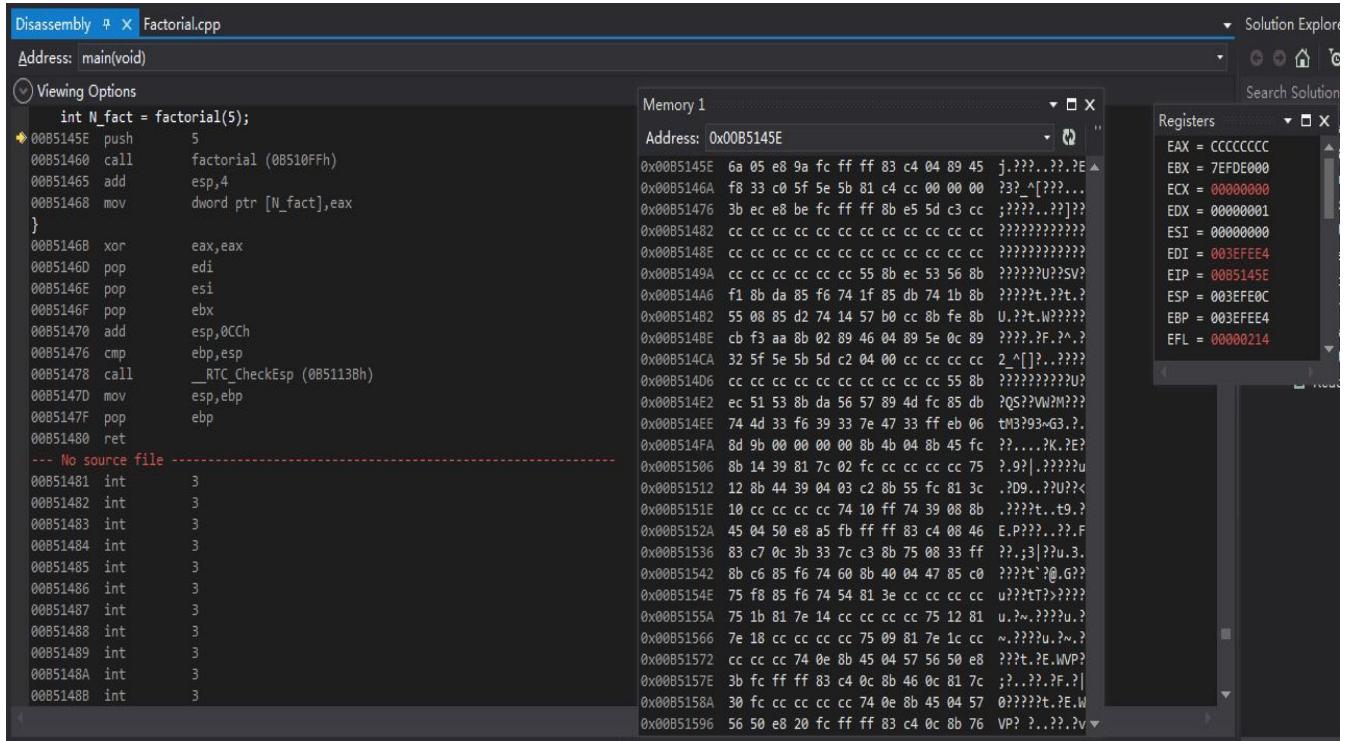
The below is the debugging process of factorial 5.

```

int N_fact = factorial(5);
00B5145E push      5
00B51460 call      factorial (0B510FFh)
00B51465 add       esp,4
00B51468 mov        dword ptr [N_fact],eax
}
00B5146B xor       eax,eax
00B5146D pop       edi
00B5146E pop       esi
00B5146F pop       ebx
00B51470 add       esp,0CCh
00B51476 cmp       ebp,esp
00B51478 call     _RTC_CheckEsp (0B5113Bh)
00B5147D mov       esp,ebp
00B5147F pop       ebp
00B51480 ret

```

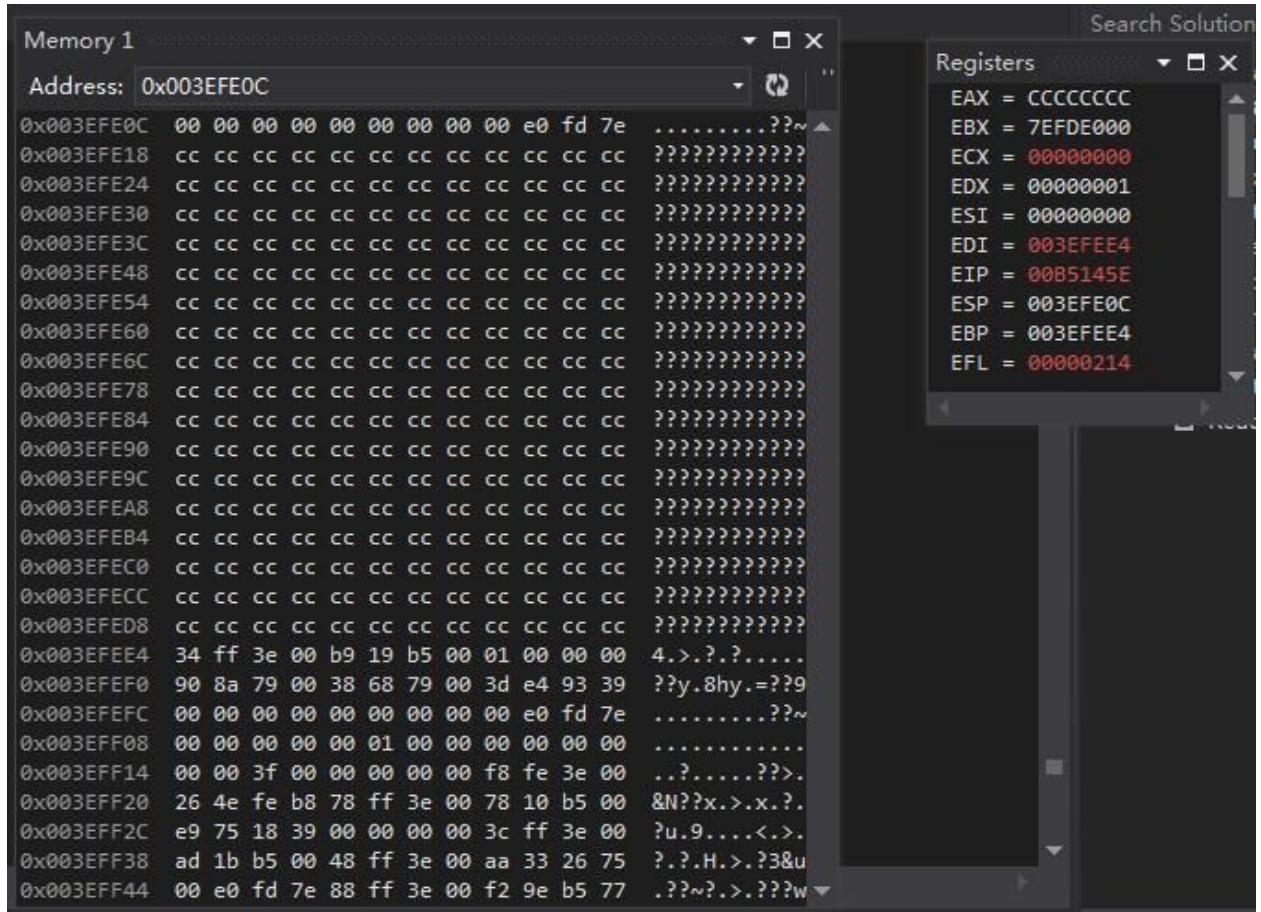
In the initial step, we need to save the value 5. Its EIP is 0x00B5145E. The EIP register always contains the address of the next instruction to be executed. You cannot directly access or change the instruction pointer. However, instructions that control program flow, such as calls, jumps, loops, and interrupts, automatically change the instruction pointer. CPU can not read the following instructions without it. The popular point on the CPU can not be executed each time the corresponding assembly instructions to complete the corresponding eip value will increase.



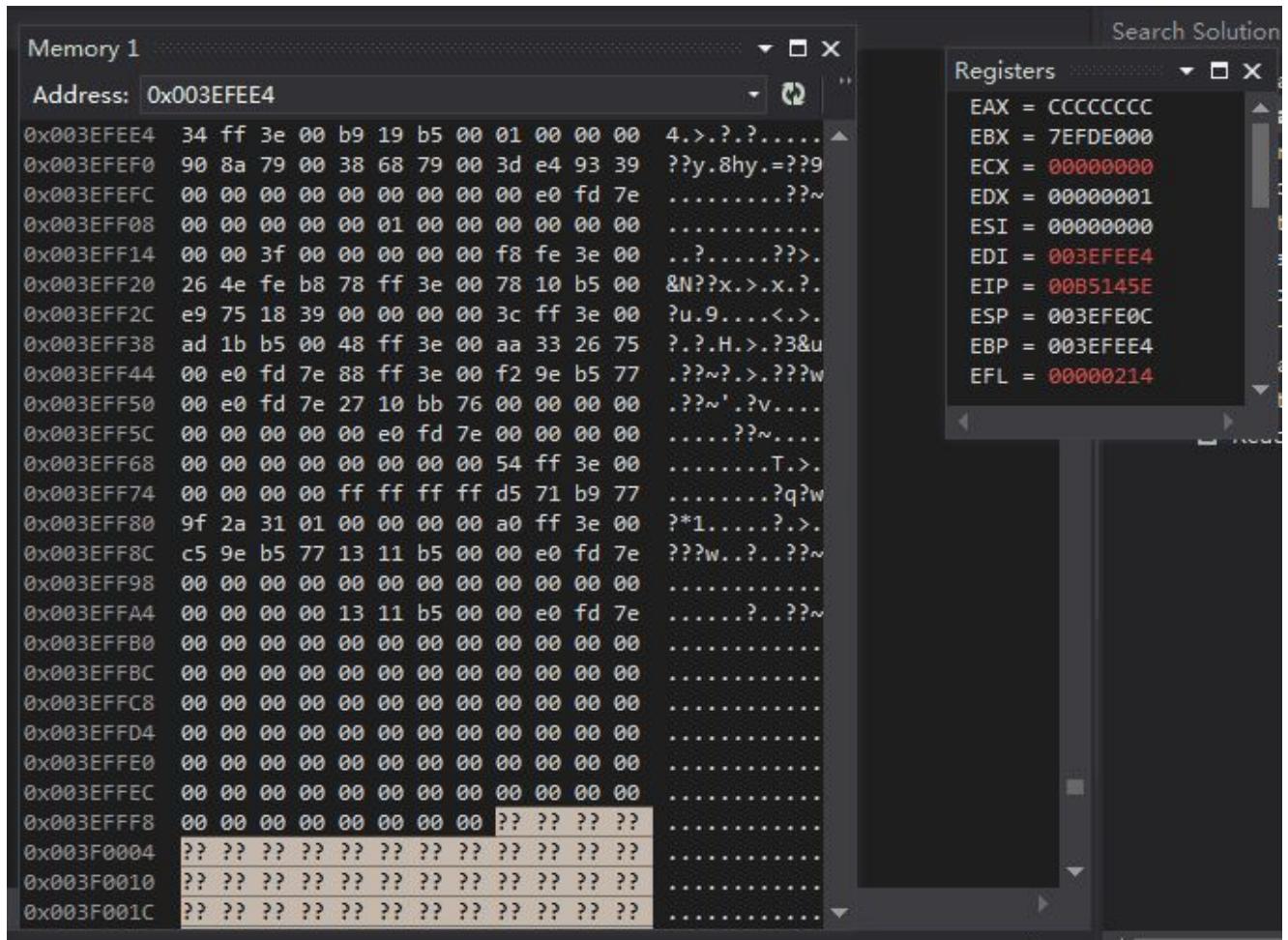
Then we need to search the register ESP. It is the extended stack pointer in the stack pointer. It has memory register is 0x003EFE0C. The esp change or not has nothing to do with ebp, ebp change or not has nothing to do with esp. The esp's point will be with the stack, the stack operation and change, and ebp will not follow the stack, the stack operation and change the

top of the stack will change with the esp point, the top of the stack changes and ebp is irrelevant.

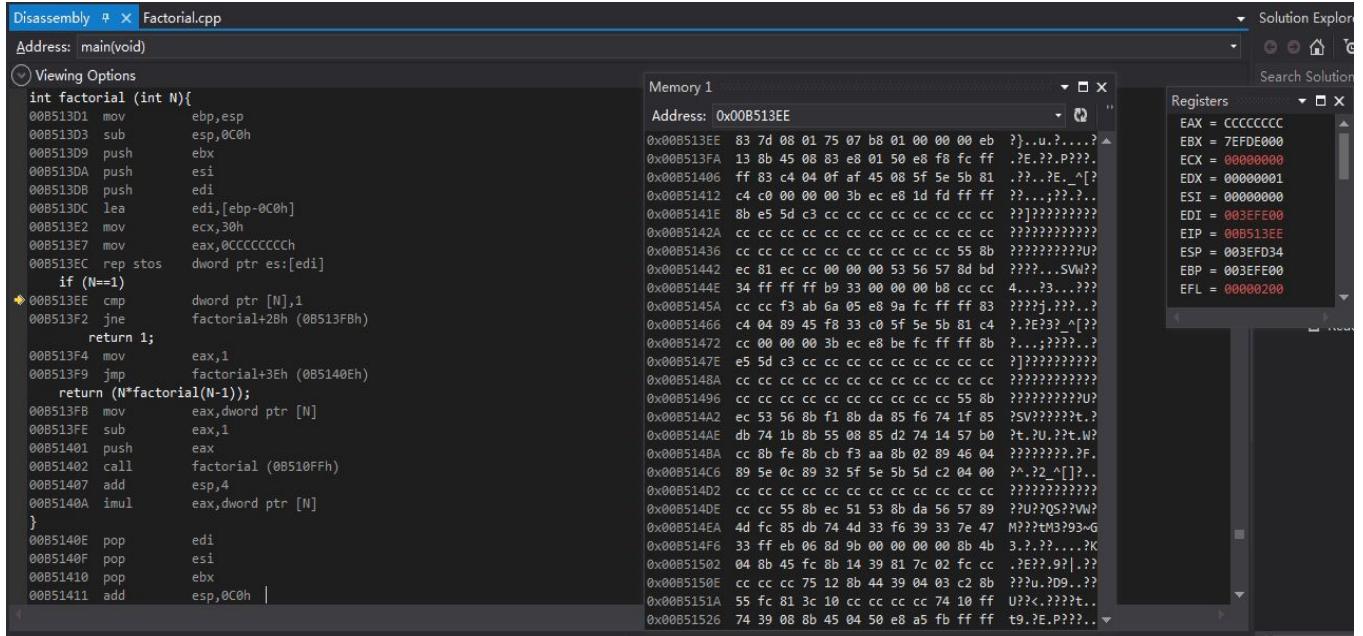
The ebp originally pointed to where or where to point, unless the instruction changes ebp value of its point to change only.



The below is the register EBP which is extended base pointer, it contains a pointer to the bottom of a stack frame at the top of the system stack. The ebp only access to a certain moment of esp, this time is to enter a function, the cpu will be assigned to the value of esp to ebp, ebp at this time can operate on the stack, such as access to function parameters, local variables, in fact the system use esp can also implement the function.

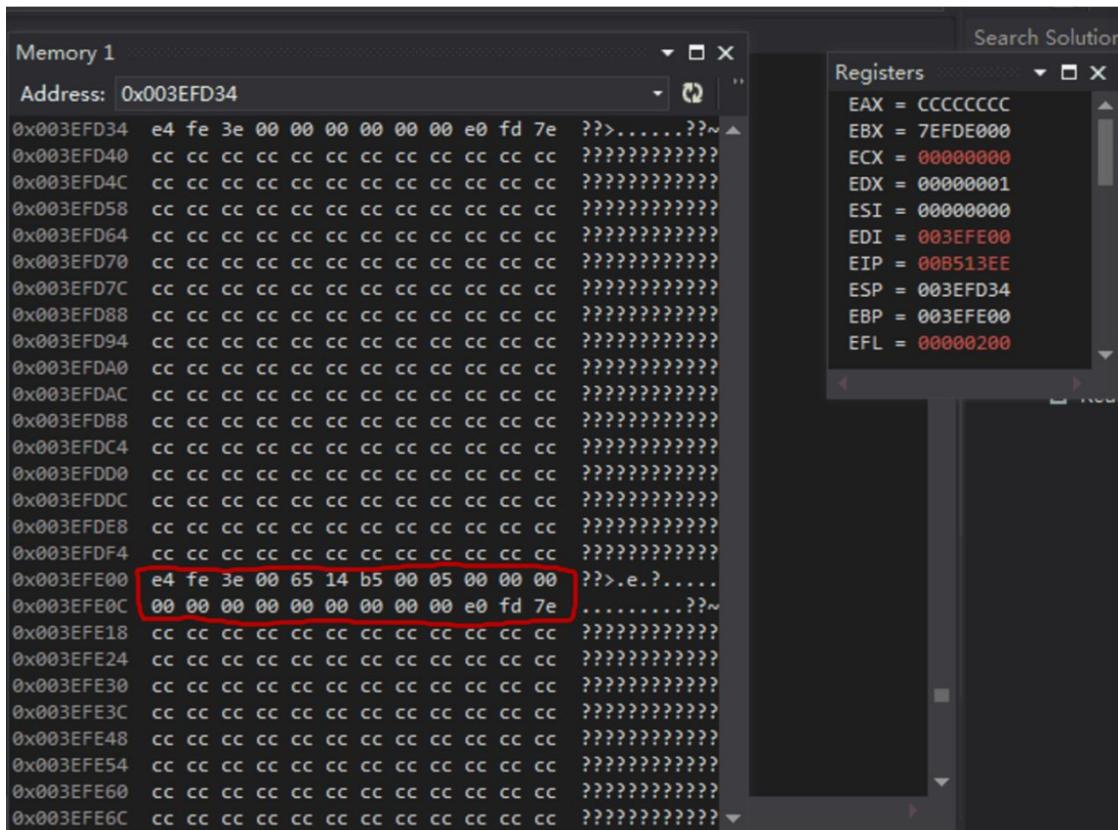


Then we debug over to inductive step of the factorial, N=1. It's memory register EIP is 0x000B513EE. First, the cmp instruction is a comparison instruction that compares the size of the first argument and the second argument. Now the first argument is dword ptr [esp] dword ptr means that this is a double word pointer, that is, the data to be addressed is a double word (4 bytes). And the data in the memory of the address offset is esp this register stored in the content. The It is where the segment is the stack, SS.

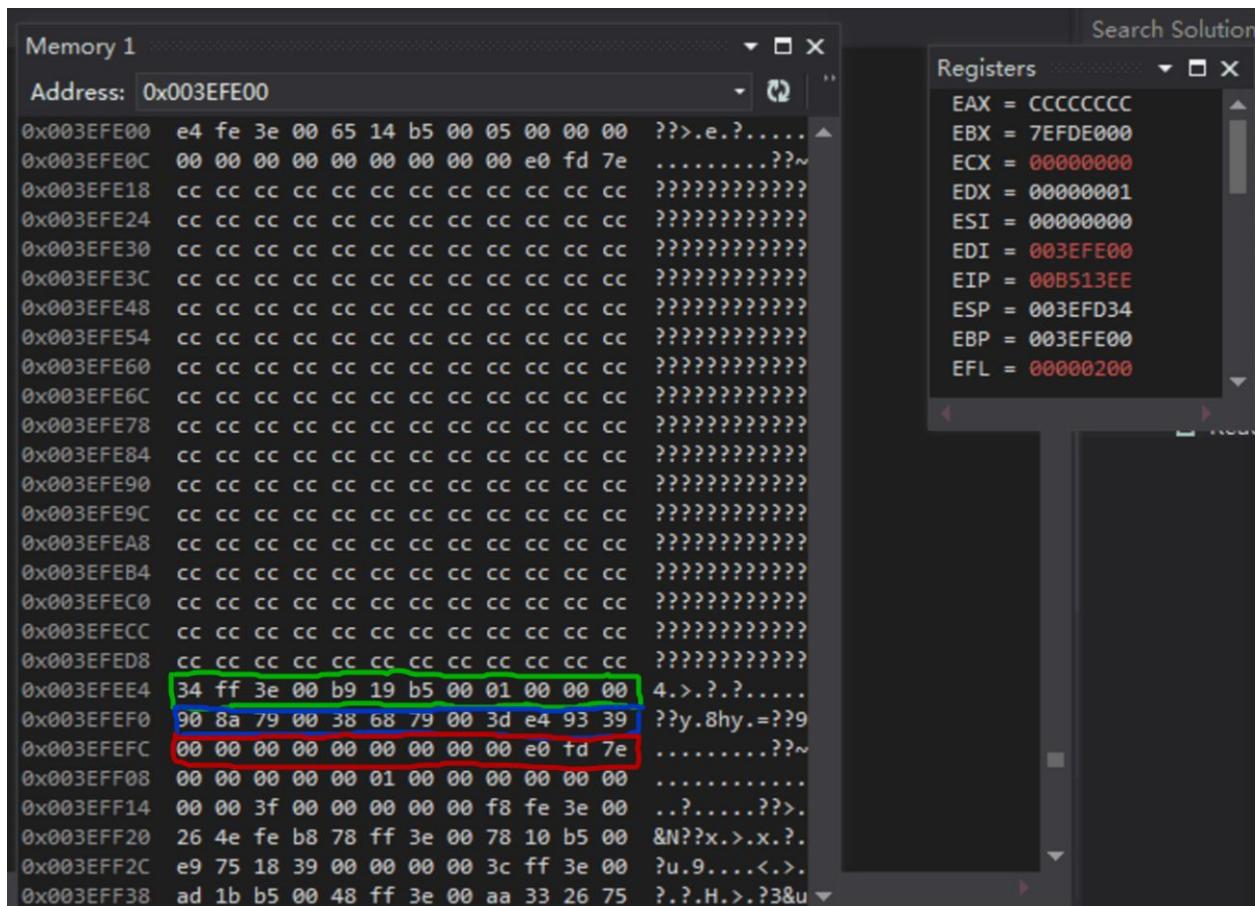


Now it is the debug memory values of register ESP. It has memory register 0x003EFD34.

The below red circle is saved ESP of main() and returning address during execution.



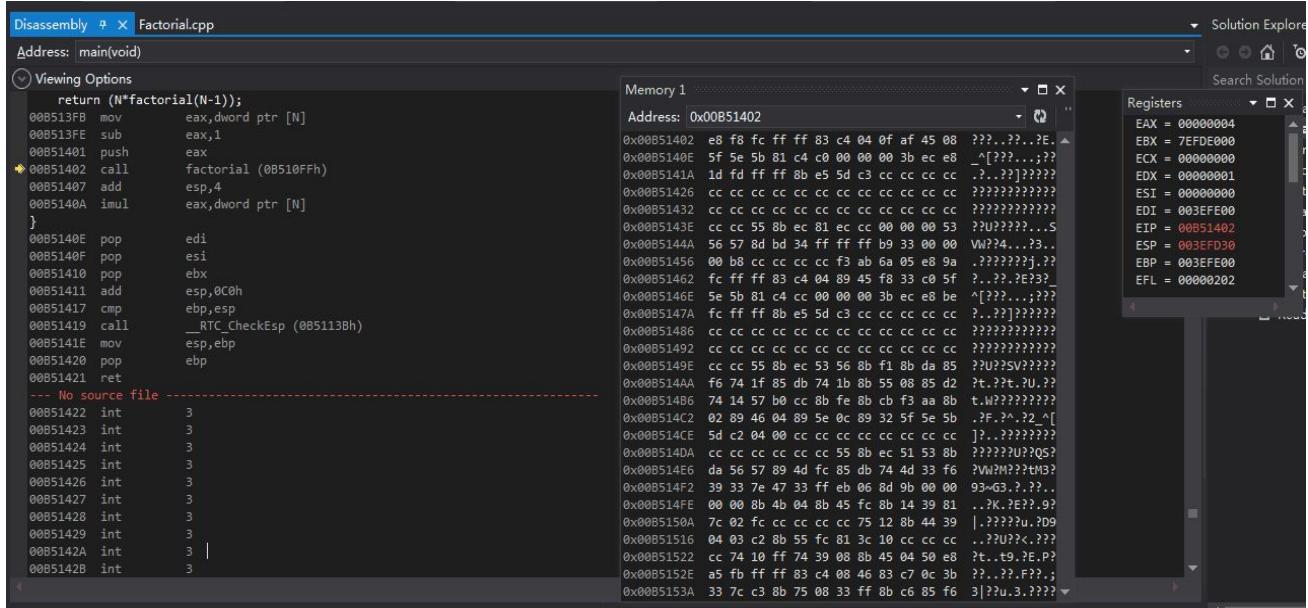
The below is the register memory of EBP which is 0x003EFE00. The green means saved EBP of main(). The blue means return address during execution of factorial(5). The red means argument during execution of factorial(5).



The red means factorial(1), it's value is 81 18 1a a5. The green means factorial(2), it's value is 78 10 05 00 89 81 3d a5. The blue means factorial(3), it's value is e0 fd 7e 69 2e ef 77. The purple means factorial(4), it's value is c5 9e 8a 77 13 11 05. The orange means factorial(5) means factorial(5), it's value is 13 11 05.

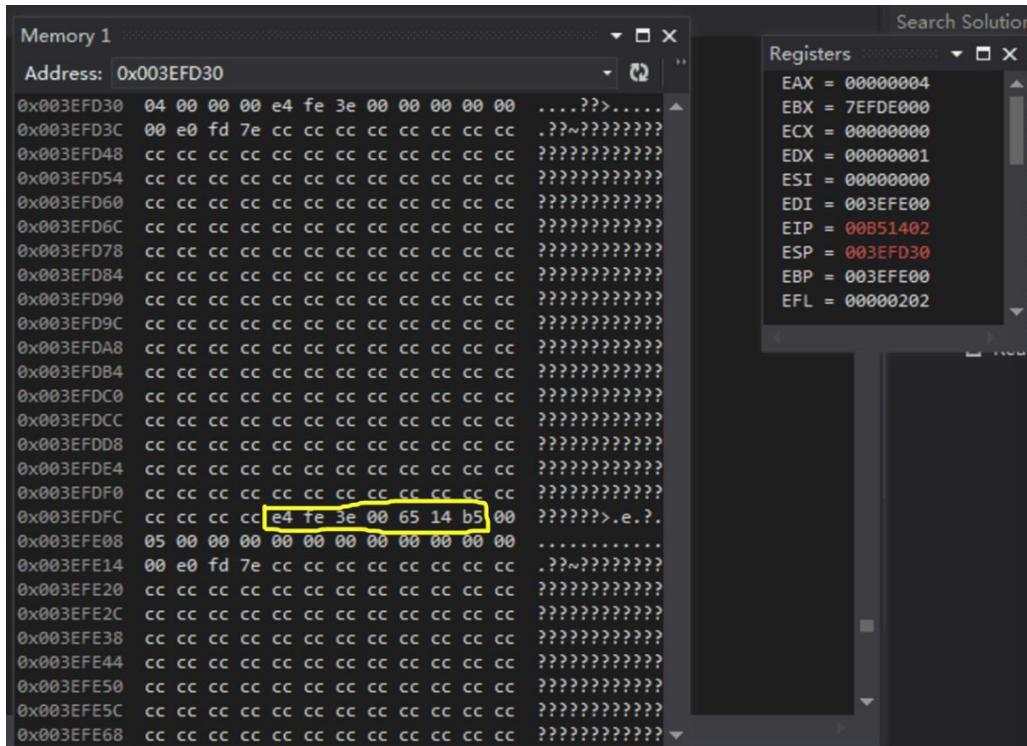
This step is going to call the factorial function. Its memory register EIP is 0x00B51402.

The ECX which is called the counter register. It is used as a loop counter and for shifts gets some interrupt values. There is no register memory and that factorial could not have any loop shift.

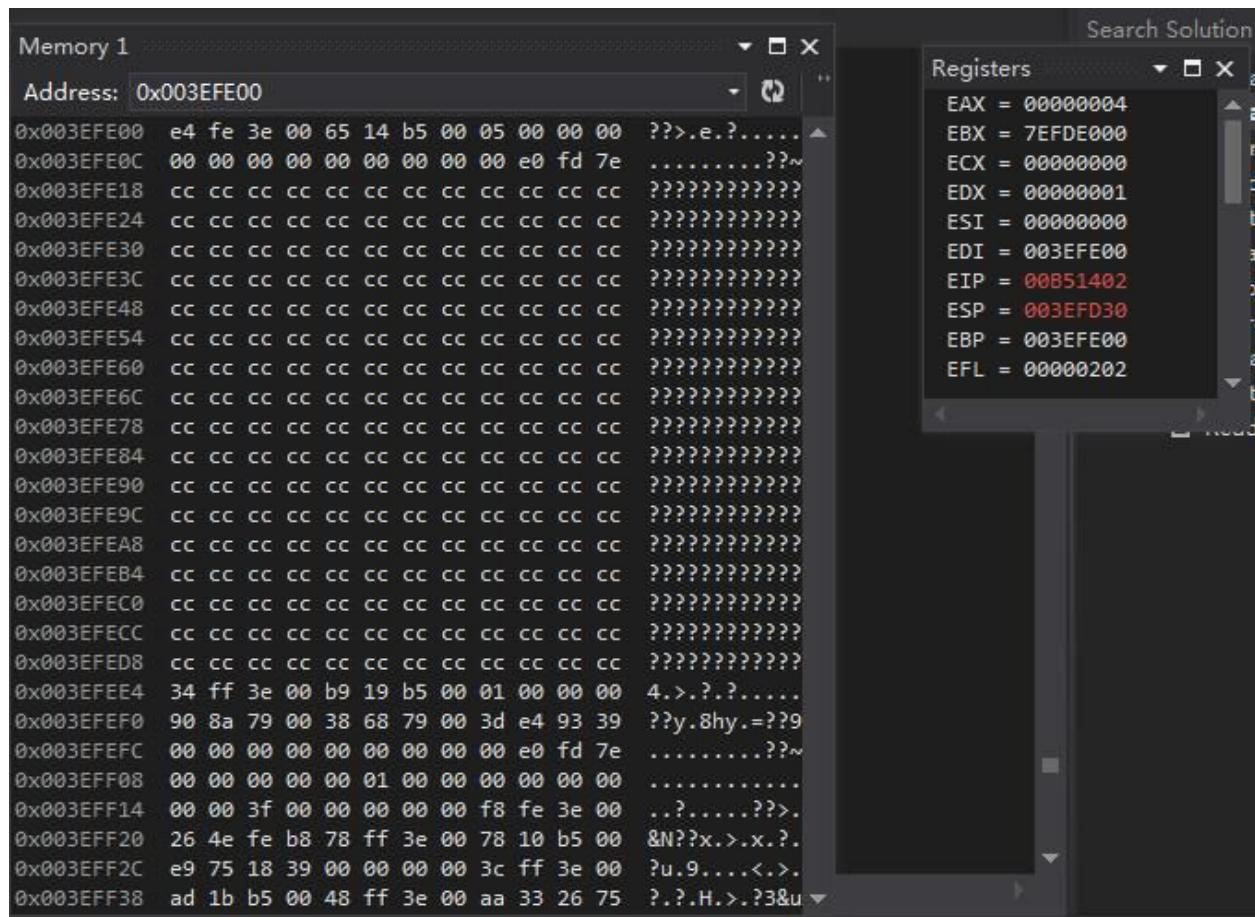


The memory register of ESP is 0x003EFD30. After the debugging, the main function call

the factorial (0B510FFh) which has the memory value in the below yellow circle.



The ESP, it is the extended stack pointer in the stack pointer. It has memory register is 0x003EFE00. The esp change or not has nothing to do with ebp, ebp change or not has nothing to do with esp. The esp's point will be with the stack, the stack operation and change, and ebp will not follow the stack, the stack operation and change the top of the stack will change with the esp point, the top of the stack changes and ebp is irrelevant.



◆ 3. Part II Analysis In MIPS on MARS Simulator

The below is my factorial assembly code. There are some keywords in this code. For example, In “jal fact” , jal is Jump and Link instructions which are similar to the jump instructions, except that they store the address of the next instruction (the one immediately after the jump) in the return address (\$ra; \$31) register. This allows a subroutine to return to the main body routine after completion.

```
factorial.asm
1 .text
2     main:
3         li    $a0, 5
4         jal   fact
5         addi $s0, $v0, 0
6         syscall
7
8     fact:
9         slti $t0, $a0, 1
10        beq   $t0, $zero, L1
11        addi $v0, $zero, 1
12        addi $sp, $sp, 8
13        jr    $ra
14
15    L1:
16        addi $sp, $sp, -8
17        sw    $ra, 4($sp)
18        sw    $a0, 0($sp)
19        addi $a0, $a0, -1
20        jal   fact
21        lw    $a0, 0($sp)
22        lw    $ra, 4($sp)
23        addi $sp, $sp, 8
24        mul   $v0, $a0, $v0
25        jr    $ra
```

In the initial step, that load immediate value 5 into destination register \$a0. The \$sp means stack pointer which has value 0x7ffffeffc, it points to last location on the stack. The pc is program counter, it shows the current address of text is 0x0040000 in \$a0.

The screenshot shows the MARS 4.5 assembly debugger interface. The assembly window displays the following code:

```

Text Segment
Bkpt Address Code Basic Source
0x00400000 0x24040005 addiu $4,$0,0x00000005 3: li $a0, 5
0x00400004 0x0c100004 jal 0x00400010 4: jal fact
0x00400008 0x20500000 addi $16,$2,0x00000000 5: addi $s0, $v0, 0
0x0040000c syscall 6: syscall
0x00400010 0x28880001 slti $8,$4,0x00000001 9: slti $t0, $a0, 1
0x00400014 0x11000003 beq $8,$0,0x00000003 10: beq $t0, $zero, L1
0x00400018 0x20020001 addi $2,$0,0x00000001 11: addi $v0, $zero, 1
0x0040001c 0x23bd0008 addi $29,$29,0x0000... 12: addi $sp, $sp, 8
0x00400020 0x03e00008 jr $31 13: jr $ra

```

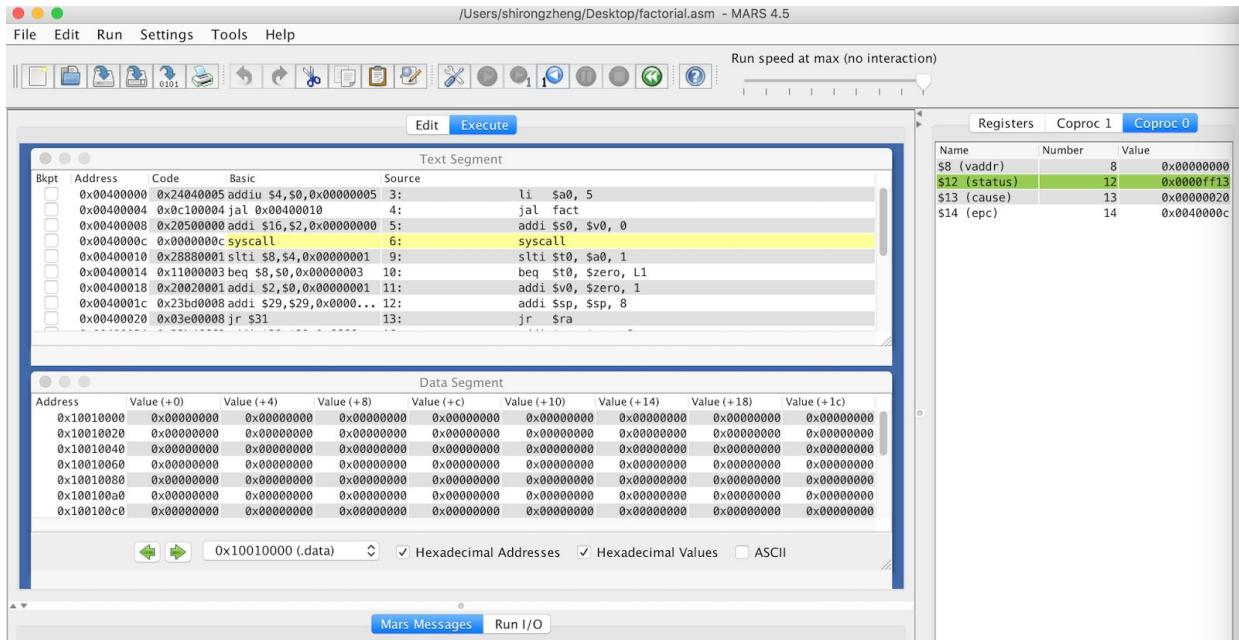
The Data Segment window shows memory starting at address 0x10010000 with all values set to 0x00000000.

The Registers window shows the following initial values:

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10000000
\$sp	29	0x7ffffeffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Mars Messages: Assemble: operation completed successfully.

In computing, a system call (syscall) is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. The fundamental interface between an application and the Linux kernel. System calls and library wrapper functions System calls are generally not invoked directly, but rather via wrapper functions in glibc (or perhaps some other library). The \$12(status) has memory value is 0x0000ff13.



A stack pointer \$sp is a small register that stores the address of the last program request in a stack. A stack is a specialized buffer which stores data from the top down. As new requests come in, they "push down" the older ones. The most recently entered request always resides at the top of the stack, and the program always takes requests from the top. It's memory values is 0x7ffffeffc.

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000078
\$v1	3	0x00000000
\$a0	4	0x00000005
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000001
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000078
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7ffffeffc
\$fp	30	0x00000000
\$ra	31	0x00400008
\$pc		0x0040000c
\$hi		0x00000000
\$lo		0x00000078

The initial value of \$a0 which is factorial(5), it is 0x00000005. Based on the current \$sp,

+0 value is 0x00400038, +4 value is 0x00000003, +8 value is 0x00400038, +c value is 0x00000004, +10 value is 0x00400038, +14 value is 0x00000005, +18 value is 0x00400008. The \$ra means return address, it will return 0x00400008.

The screenshot shows the MARS 4.5 assembly debugger interface. The top menu bar includes File, Edit, Run, Settings, Tools, and Help. Below the menu is a toolbar with various icons for file operations and simulation controls. The main window is divided into several panes:

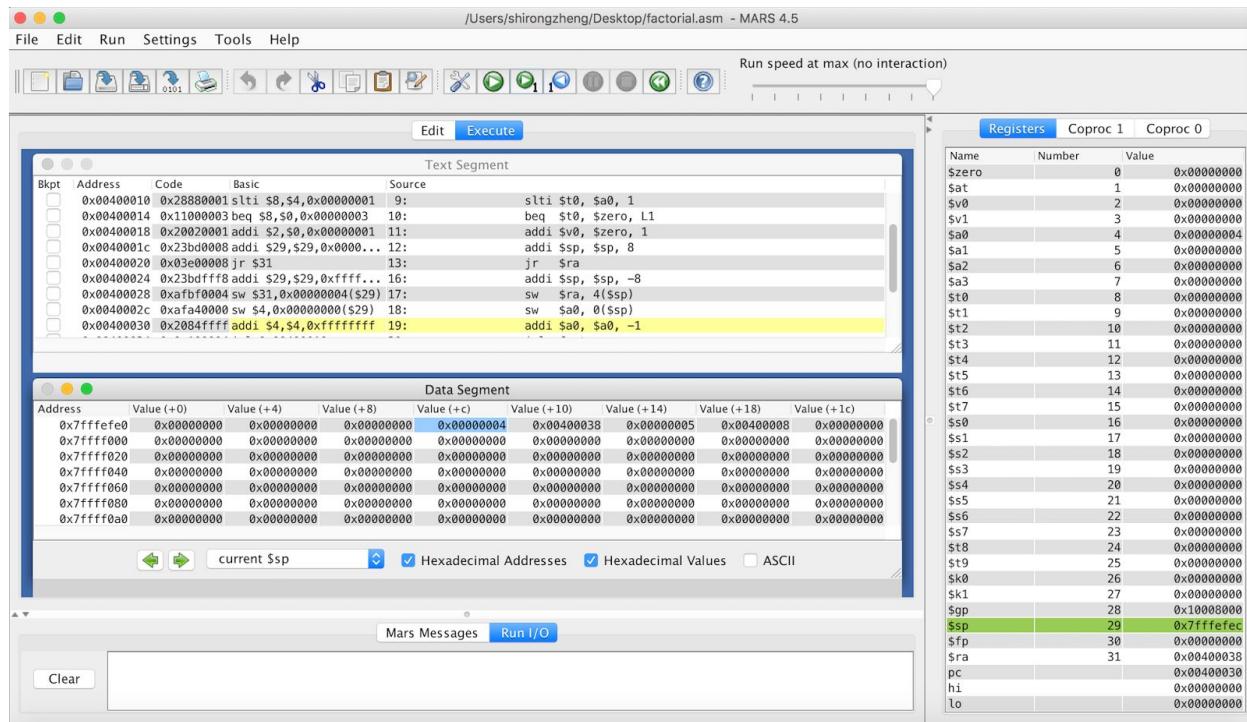
- Text Segment:** Shows assembly code with columns for Bkpt, Address, Code, Basic, and Source. A yellow highlight covers the instruction at address 0x0040000c, which is a syscall.
- Data Segment:** Shows memory starting at address 0x7ffffe00. The first few entries are:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7ffffe00	0x00400038	0x00000003	0x00400038	0x00000004	0x00400038	0x00000005	0x00400008	0x00000000
0x7fffff000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff0a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
- Registers:** A table showing the state of various registers. The \$a0 register is highlighted in green with a value of 0x00000005.

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000078
\$v1	3	0x00000000
\$a0	4	0x00000005
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000001
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000078
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10000000
\$sp	29	0x7ffffeffc
\$fp	30	0x00000000
\$ra	31	0x00400008
pc		0x00400010
hi		0x00000000
lo		0x00000078

This step is called decrement, the basic signed binary is 0xffffffff, then the factorial (n-1).

Then you could see the \$a0 become 0x00000004. The previous “addi \$sp, \$sp, -8” is subtract 8 bytes from the stack pointer , there is no override to the address of \$a0. The math way is $0x7ffffeff4 - 8 = 0x0x7fffec$.



Data Segment									
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)	
0x7fffffe0	0x00400038	0x00000003	0x00400038	0x00000004	0x00400038	0x00000005	0x00400008	0x00000000	
0x7fffff000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x7fffff020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x7fffff040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x7fffff060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x7fffff080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x7fffff0a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	

The \$14(epc) is exception program counter, It's memory value is 0x0040000c. When a procedure is called using jal, two things happen, 1. control is transferred at the address provided by the instruction, 2. the return address is saved in register \$ra. The return address can not be saved in \$ra since it may clobber a return address that has been placed in that register before the exception. The Exception Program Counter (EPC) is used to store the address of the instruction that was executing when the exception was generated.

Registers	Coproc 1	Coproc 0
Name	Number	Value
\$8 (vaddr)	8	0x00000000
\$12 (status)	12	0x0000ff13
\$13 (cause)	13	0x00000020
\$14 (epc)	14	0x0040000c

❖ 4. Part III Analysis In 64-bit Intel Processor gdb/gcc On Linux

The below is my factorial.cpp which written on VIM of Linux.

```
factorial.cpp + (~) - VIM

int factorial (int N){
    if(N==1)
        return 1;
    return (N*factorial(N-1));
}

int main(){
    int N_fact=factorial(5);
}
```

The system will set breakpoint 1 at the entry to the factorial(5) in main function.

```
+(gdb) break main
Breakpoint 1 at 0x4004f9: file factorial.cpp, line 8.
+(gdb) run
Starting program: /home/csc103/tt

Breakpoint 1, main () at factorial.cpp:8
8     int N_fact=factorial(5);
+(gdb) next 2
0x00007ffff71b9291 in __libc_start_main () from /usr/lib/libc.so.6
```

The below is debug the code and disassemble the memory address. A disassembler is the exact opposite of an assembler. Where an assembler converts code written in an assembly language into binary machine code, a disassembler reverses the process and attempts to recreate the assembly code from the binary machine code.

```
+ (gdb) disassembly
Dump of assembler code for function __libc_start_main:
0x00007ffff71b91a0 <+0>: push %r14
0x00007ffff71b91a2 <+2>: push %r13
0x00007ffff71b91a4 <+4>: push %r12
0x00007ffff71b91a6 <+6>: push %rbp
0x00007ffff71b91a7 <+7>: mov %rcx,%rbp
0x00007ffff71b91aa <+10>: push %rbx
0x00007ffff71b91ab <+11>: sub $0x90,%rsp
0x00007ffff71b91b2 <+18>: mov 0x377d87(%rip),%rax      # 0x7ffff7530f40
0x00007ffff71b91b9 <+25>: mov %rdi,0x18(%rsp)
0x00007ffff71b91be <+30>: mov %esi,0x14(%rsp)
0x00007ffff71b91c2 <+34>: mov %rdx,0x8(%rsp)
0x00007ffff71b91c7 <+39>: test %rax,%rax
0x00007ffff71b91ca <+42>: je 0x7ffff71b9298 <__libc_start_main+248>
0x00007ffff71b91d0 <+48>: mov (%rax),%eax
0x00007ffff71b91d2 <+50>: test %eax,%eax
0x00007ffff71b91d4 <+52>: sete %al
0x00007ffff71b91d7 <+55>: lea 0x377e82(%rip),%rdx      # 0x7ffff7531060 <__libc_multipl
e_libcs>
0x00007ffff71b91de <+62>: movzbl %al,%eax
0x00007ffff71b91e1 <+65>: test %r9,%r9
0x00007ffff71b91e4 <+68>: mov %eax,(%rdx)
0x00007ffff71b91e6 <+70>: je 0x7ffff71b91f4 <__libc_start_main+84>
0x00007ffff71b91e8 <+72>: xor %edx,%edx
0x00007ffff71b91ea <+74>: xor %esi,%esi
0x00007ffff71b91ec <+76>: mov %r9,%rdi
0x00007ffff71b91ef <+79>: callq 0x7ffff71cecc20 <__cxa_atexit>
0x00007ffff71b91f4 <+84>: mov 0x377c55(%rip),%rdx      # 0x7ffff7530e50
0x00007ffff71b91fb <+91>: mov (%rdx),%ebx
0x00007ffff71b91fd <+93>: and $0x2,%ebx
```

```
0x00007ffff71b9200 <+96>: jne 0x7ffff71b92d7 <__libc_start_main+311>
---Type <return> to continue, or q <return> to quit---
0x00007ffff71b9206 <+102>: test %rbp,%rbp
0x00007ffff71b9209 <+105>: je 0x7ffff71b9220 <__libc_start_main+128>
0x00007ffff71b920b <+107>: mov 0x377c8e(%rip),%rax      # 0x7ffff7530ea0
0x00007ffff71b9212 <+114>: mov 0x8(%rsp),%rsi
0x00007ffff71b9217 <+119>: mov 0x14(%rsp),%edi
0x00007ffff71b921b <+123>: mov (%rax),%rdx
0x00007ffff71b921e <+126>: callq *%rbp
0x00007ffff71b9220 <+128>: mov 0x377c29(%rip),%rax      # 0x7ffff7530e50
0x00007ffff71b9227 <+135>: mov 0x168(%rax),%r14d
0x00007ffff71b922e <+142>: test %r14d,%r14d
0x00007ffff71b9231 <+145>: jne 0x7ffff71b932b <__libc_start_main+395>
0x00007ffff71b9237 <+151>: test %ebx,%ebx
0x00007ffff71b9239 <+153>: jne 0x7ffff71b9308 <__libc_start_main+360>
0x00007ffff71b923f <+159>: lea 0x20(%rsp),%rdi
0x00007ffff71b9244 <+164>: callq 0x7ffff71cbe00 <_setjmp>
0x00007ffff71b9249 <+169>: test %eax,%eax
0x00007ffff71b924b <+171>: jne 0x7ffff71b929f <__libc_start_main+255>
0x00007ffff71b924d <+173>: mov %fs:0x300,%rax
0x00007ffff71b9256 <+182>: mov %rax,0x68(%rsp)
0x00007ffff71b925b <+187>: mov %fs:0x2f8,%rax
0x00007ffff71b9264 <+196>: mov %rax,0x70(%rsp)
0x00007ffff71b9269 <+201>: lea 0x20(%rsp),%rax
0x00007ffff71b926e <+206>: mov %rax,%fs:0x300
0x00007ffff71b9277 <+215>: mov 0x377c22(%rip),%rax      # 0x7ffff7530ea0
0x00007ffff71b927e <+222>: mov 0x8(%rsp),%rsi
0x00007ffff71b9283 <+227>: mov 0x14(%rsp),%edi
0x00007ffff71b9287 <+231>: mov (%rax),%rdx
0x00007ffff71b928a <+234>: mov 0x18(%rsp),%rax
```

```

0x00007ffff71b928f <+239>: callq  *%rax
=> 0x00007ffff71b9291 <+241>: mov    %eax,%edi
 0x00007ffff71b9293 <+243>: callq  0x7ffff71ce9d0 <exit>
---Type <return> to continue, or q <return> to quit---
0x00007ffff71b9298 <+248>: xor    %eax,%eax
0x00007ffff71b929a <+250>: jmpq   0x7ffff71b91d7 <__libc_start_main+55>
0x00007ffff71b929f <+255>: mov    0x37d40(%rip),%rax      # 0x7ffff75366b0 <__libc_pthread
_functions+400>
0x00007ffff71b92a6 <+262>: ror    $0x11,%rax
0x00007ffff71b92aa <+266>: xor    %fs:0x30,%rax
0x00007ffff71b92b3 <+275>: callq  *%rax
0x00007ffff71b92b5 <+277>: mov    0x37d3e4(%rip),%rax      # 0x7ffff75366a0 <__libc_pthread
_functions+384>
0x00007ffff71b92bc <+284>: ror    $0x11,%rax
0x00007ffff71b92c0 <+288>: xor    %fs:0x30,%rax
0x00007ffff71b92c9 <+297>: lock decl (%rax)
0x00007ffff71b92cc <+300>: sete   %dl
0x00007ffff71b92cf <+303>: test   %dl,%dl
0x00007ffff71b92d1 <+305>: je     0x7ffff71b92f3 <__libc_start_main+339>
0x00007ffff71b92d3 <+307>: xor    %eax,%eax
0x00007ffff71b92d5 <+309>: jmp    0x7ffff71b9291 <__libc_start_main+241>
0x00007ffff71b92d7 <+311>: mov    0x8(%rsp),%rax
0x00007ffff71b92dc <+316>: lea    0x140992(%rip),%rdi      # 0x7ffff72f9c75
0x00007ffff71b92e3 <+323>: mov    (%rax),%rsi
0x00007ffff71b92e6 <+326>: xor    %eax,%eax
0x00007ffff71b92e8 <+328>: callq  *0x110(%rdx)
0x00007ffff71b92ee <+334>: jmpq   0x7ffff71b9206 <__libc_start_main+102>
0x00007ffff71b92f3 <+339>: mov    $0x3c,%edx
0x00007ffff71b92f8 <+344>: nopl   0x0(%rax,%rax,1)
0x00007ffff71b9300 <+352>: xor    %edi,%edi

0x00007ffff71b9302 <+354>: mov    %edx,%eax
0x00007ffff71b9304 <+356>: syscall
0x00007ffff71b9306 <+358>: jmp    0x7ffff71b9300 <__libc_start_main+352>
0x00007ffff71b9308 <+360>: mov    0x8(%rsp),%rax
0x00007ffff71b930d <+365>: mov    0x377b3c(%rip),%rdx      # 0x7ffff7530e50
---Type <return> to continue, or q <return> to quit---
0x00007ffff71b9314 <+372>: lea    0x140974(%rip),%rdi      # 0x7ffff72f9c8f
0x00007ffff71b931b <+379>: mov    (%rax),%rsi
0x00007ffff71b931e <+382>: xor    %eax,%eax
0x00007ffff71b9320 <+384>: callq  *0x110(%rdx)
0x00007ffff71b9326 <+390>: jmpq   0x7ffff71b923f <__libc_start_main+159>
0x00007ffff71b932b <+395>: mov    0x160(%rax),%r13
0x00007ffff71b9332 <+402>: mov    0x377a97(%rip),%rax      # 0x7ffff7530dd0
0x00007ffff71b9339 <+409>: xor    %r12d,%r12d
0x00007ffff71b933c <+412>: mov    (%rax),%rbp
0x00007ffff71b933f <+415>: add    $0x470,%rbp
0x00007ffff71b9346 <+422>: mov    0x18(%r13),%rax
0x00007ffff71b934a <+426>: test   %rax,%rax
0x00007ffff71b934d <+429>: je     0x7ffff71b9354 <__libc_start_main+436>
0x00007ffff71b934f <+431>: mov    %rbp,%rdi
0x00007ffff71b9352 <+434>: callq  *%rax
0x00007ffff71b9354 <+436>: add    $0x1,%r12d
0x00007ffff71b9358 <+440>: add    $0x10,%rbp
0x00007ffff71b935c <+444>: mov    0x40(%r13),%r13
0x00007ffff71b9360 <+448>: cmp    %r12d,%r14d
0x00007ffff71b9363 <+451>: jne    0x7ffff71b9346 <__libc_start_main+422>
0x00007ffff71b9365 <+453>: jmpq   0x7ffff71b9237 <__libc_start_main+151>
End of assembler dump.

```

The “x/i \$pc” can be set to run all the time using the usual configuration mechanism. The debug system will move the value of %edi to the store place of \$eax. In the second debug command, It is pointed to by rbp, the "base pointer," and is used in combination with an offset to reference all local variables. Every time a function is called, RBP is updated to point to its stack frame. The value of rbp is 400510. The rsp point to the top of the current stack frame, then the value of rsb is fffffd70.

```
+ (gdb) x/i $pc  
=> 0x7ffff71b9291 <_libc_start_main+241>:      mov     %eax,%edi  
+(gdb) printf "rbp:%x\nrsp:%x\n", $rbp, $rsp  
rbp:400510  
rsp:fffffd70
```

The green is saved RBP of factorial(2), the blue is return address during factorial(1), the red is argument during factorial(1).

```
+ (gdb) x/12xw $rsp  
0x7fffffffdd70: 0x00011c00 0x00000000 0xfffffde48 0x00007fff  
0x7fffffffdd80: 0xfffffde58 0x00000001 0x004004f1 0x00000000  
0x7fffffffdd90: 0x00000000 0x00000000 0x4f6da5cf 0xda839b6a  
+ (gdb) p $rip  
$1 = (void (*)()) 0x7ffff71b9291 <_libc_start_main+241>
```

A program counter is a register in a computer processor that contains the address (location) of the instruction being executed at the current time. As each instruction gets fetched, the program counter increases its stored value by 1. ... Within a computer, an address is a specific location in memory or storage.

The first debug command is print /x \$rip which print program counter in hex. The second debug command is print /d \$rip which print program counter in decimal. The third debug command is print /t \$rip which print program counter in binary.

```
+ (gdb) print /x $rip  
$2 = 0x7fffff71b9291  
+ (gdb) print /d $rip  
$3 = 140737339167377  
+ (gdb) print /t $rip  
$4 = 111111111111111111110111000110111001001010010001
```

The first debug command is print /d \$rax which print contents of %rax in decimal. The second debug command is print /x \$rax which print contents of %rax in hex. The first debug command is print /t \$rax which print contents of %rax in binary. The first debug command is print /d(int) \$rax which print contents of %rax in decimal after sign-extending lower 32-bits. If there is some number below the 32-bit is 0xffffffff, then the \$8 is equal to -1.

The contents of %rax is data which is 0 in this debug system.

```
+ (gdb) print /d $rax  
$5 = 0  
+ (gdb) print /x $rax  
$6 = 0x0  
+ (gdb) print /t $rax  
$7 = 0  
+ (gdb) print /d(int)$rax  
$8 = 0
```

The first debug command is print 0x100 which is print decimal representation of 100. The value is 256. The second debug command is print /x(\$rsp+8) which is print (contents of \$rsp) +8 in hex. The original contents of \$rsp is 0x7fffffffdd70, then plus 8 is 0x7fffffffdd78. The third debug command is print *(int *) (\$rsp+8) which print integer at address %rsp + 8, then it is -8632.

```
+ (gdb) print /x $rsp  
$1 = 0x7fffffffdd70
```

```
+ (gdb) print 0x100  
$9 = 256  
+ (gdb) print /x($rsp+8)  
$10 = 0x7fffffffdd78  
+ (gdb) print *(int *) ($rsp+8)  
$11 = -8632
```

The first debug command is x/2wd \$rsp which examine two (4-byte) words starting at address in \$rsp, then print in decimal. The second debug command is x/gd \$rsp which examine (8-byte) word starting at address in \$rsp, then print in decimal. Because sum of two 4-byte words is equal to one 8-byte word, so they have same memory address and values.

```
+ (gdb) x/2wd $rsp  
0x7fffffffdd70: 72704      0  
+ (gdb) x/gd $rsp  
0x7fffffffdd70: 72704
```

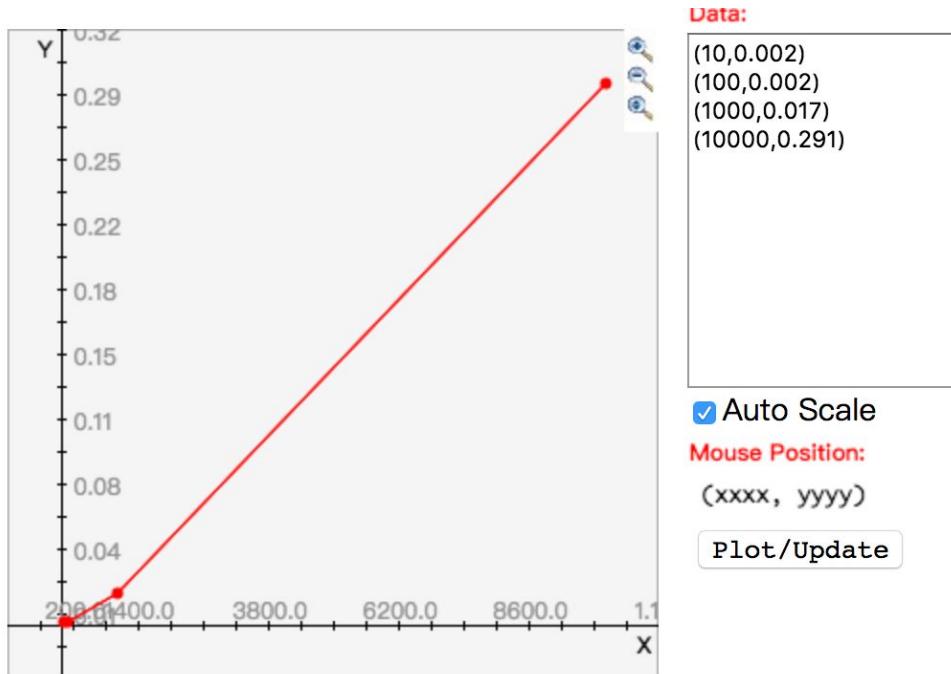
The below are register and their contents. That print the names and values of all registers except floating-point and vector registers (in the selected stack frame). GDB always considers the contents of an ordinary register as an integer when the register is examined in this way. Some machines have special registers which can hold nothing but floating point; these registers are considered to have floating point values. There is no way to refer to the contents of an ordinary register as floating point value (although you can print it as a floating point value with ‘print/f \$regname’).

```
+ (gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0xa0     160
rdx            0x7fffffffde58  140737488346712
rsi            0x7fffffffde48  140737488346696
rdi            0x1      1
rbp            0x400510 0x400510 <__libc_csu_init>
rsp            0x7fffffffdd70  0x7fffffffdd70
r8             0x400580 4195712
r9             0x7fffff7de9900  140737351948544
r10            0x7ffff7531b38  140737342806840
r11            0x0      0
r12            0x4003d0 4195280
r13            0x7fffffffde40  140737488346688
r14            0x0      0
r15            0x0      0
rip            0x7fffff71b9291  0x7fffff71b9291 <__libc_start_main+241>
eflags          0x206    [ PF IF ]
cs              0x33     51
ss              0x2b     43
ds              0x0      0
es              0x0      0
fs              0x0      0
gs              0x0      0
```

❖ 5. The Factorial Running Time

The factorial 10 has running time 0.002s, the factorial 100 has running time 0.002s, the factorial 1000 has running time 0.017s, the factorial 10000 has running time 0.291s. The line will directly going straight up, that means big factorial number cost lots of time.

```
Last login: Tue Mar 28 00:41:02 on ttys000
[ShirongdeMBP:~ shirongzheng$ cd Desktop
[ShirongdeMBP:Desktop shirongzheng$ g++ factorialTime.cpp
[ShirongdeMBP:Desktop shirongzheng$ ./a.out
factorial(10) time: 0.002
[ShirongdeMBP:Desktop shirongzheng$ g++ factorialTime.cpp
[ShirongdeMBP:Desktop shirongzheng$ ./a.out
factorial(100) time: 0.002
[ShirongdeMBP:Desktop shirongzheng$ g++ factorialTime.cpp
[ShirongdeMBP:Desktop shirongzheng$ ./a.out
factorial(1000) time: 0.017
[ShirongdeMBP:Desktop shirongzheng$ g++ factorialTime.cpp
[ShirongdeMBP:Desktop shirongzheng$ ./a.out
factorial(10000) time: 0.291
ShirongdeMBP:Desktop shirongzheng$ ]
```



❖ 6. Conclusions

In the conclusion, I learned how to debugging in MIPS, MS Visual Studio and GDB in Linux. This take home test is going to debug factorial function. We should to compare the similarity and difference of all threes. That cause each debug system, whatever the Windows X32 bit or Linux X64 bit will debugging or compiling each function. In the end, this assignment required to find out the cost time of each factorial. I spent more than 10 hours to finish all the sections. In the suggestion, it is better to do the efficiently work in less time.