Weifan Lin

Csc342 Section G

03/25/2015

Take-Home Exam

Title:

Recursive Function Factorial

Objective:

The purpose of this take-home exam was to be able to create and explain stack frame for recursive function. I compiled and ran the factorial code in .NET environment and in the MARS simulator. First, we will go through the debug mode in .NET and find the EIP, EBP, ESP, local variable and argument at that level as each Call Level is made. Next, we will go through the MARS Simulator and observe the MIPs code when factorial(n) enters the function.

Initial State before the call is made

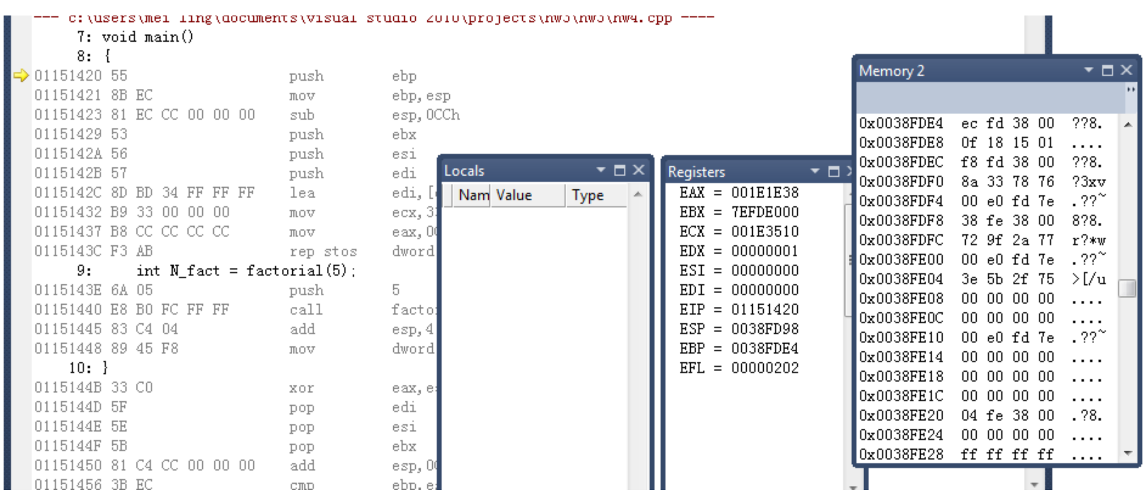


Figure 1 shows Register Frame, disassembly and stack frame at the initial state

Case 1: Num = 5;

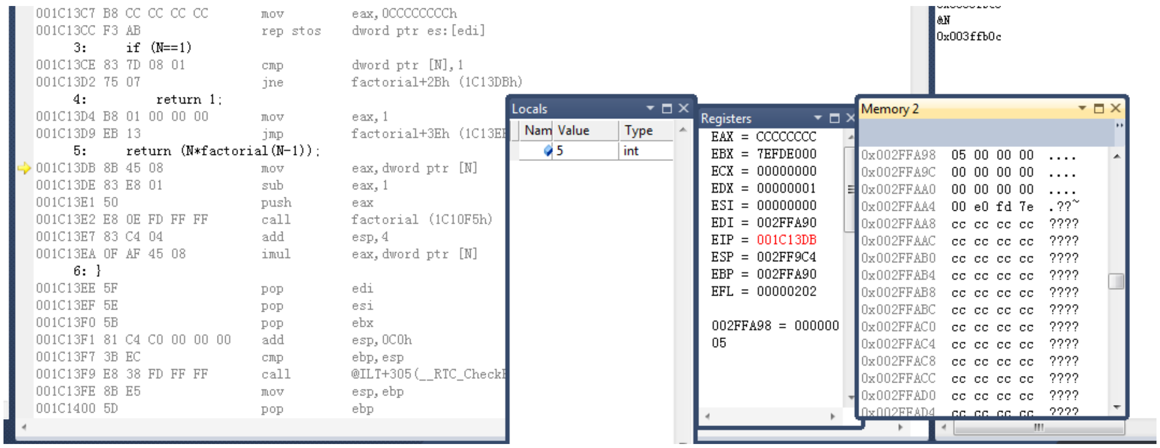


Figure 2- shows when the address at 0x 001C13E2 when first call is made

a) The argument at current level is showing the EIP equals to 0x 001C13DB meaning the next instruction is going to execute.

b) Local variable at current level has N = 5.

c) Return address at current level is store at the memory address is computed by using EIP = 0x001C13DB and since the next instruction is 0x001C13DE with length of 3 bytes. So we will add the 3rd value in stack to the EBP to find the address where **N =5 is located;**

0x002FFA90 (EBP) + 00000008 = **0x002FFA98** in figure 2

EIP = 001C13DB

ESP = 002FF9C4

EBP = 002FFA90

Case 2: N = 4:

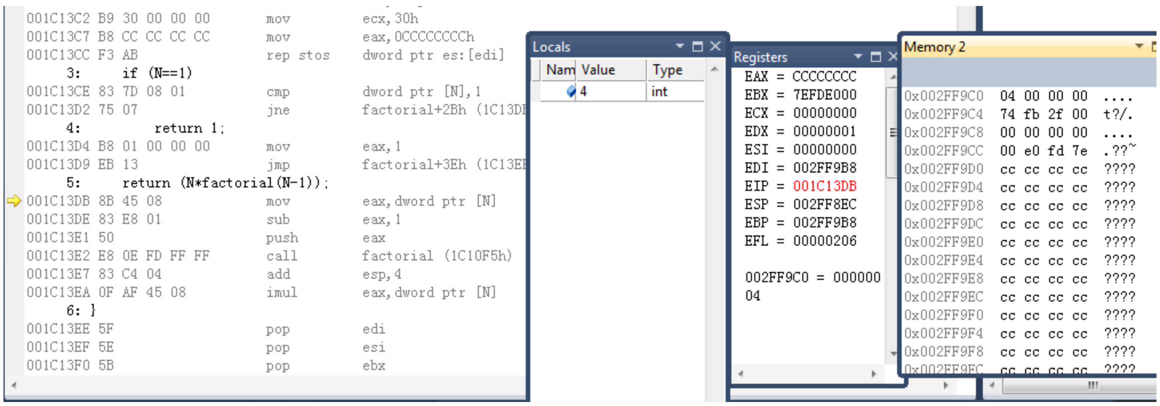


Figure 3 stack frame when EIP 0x001C13DB

a) The argument at current level is showing the EIP equals to 0x 001C13DB meaning the next instruction is going to execute.

b) Local variable at current level has N = 4.

c) Return address at current level is store at the memory address is computed by using EIP = 0x001C13DB and since the next instruction is 0x001C13DE with length of 3 bytes. So we will add the 3rd value in stack to the EBP to find the address where **N =4 is located;**

   0x002FF9B8 (EBP) + 00000008 = **0x002FF9C0** in figure 3

EIP = 001C13DB

ESP = 002FF8EC

EBP = 002FF9B8

Case 3: N = 3

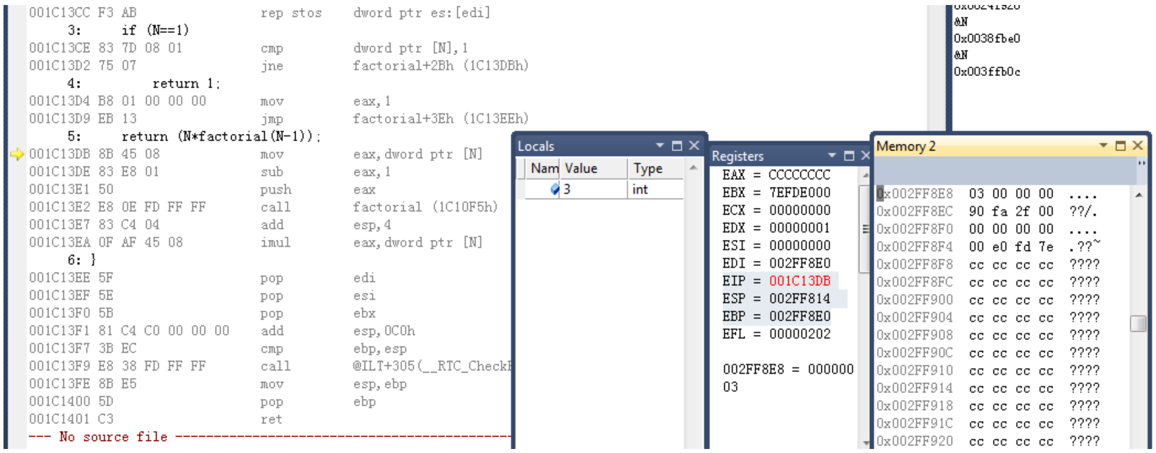


Figure 4 is showing when EBP is 0x002ff8e0

a) The argument at current level is showing the EIP equals to 0x001C13DB meaning the next instruction is going to execute.

b) Local variable at current level has N = 3.

c) Return address at current level is store at the memory address is computed by using EIP = 0x001C13DB and since the next instruction is 0x001C13DE with length of 3 bytes. So we will add the 3rd value in stack to the EBP to find the address where **N =3 is located;**

   0x002FF8E0 (EBP) + 00000008 = **0x002FF8E8** in figure 4

EIP = 0x001C13DB

ESP = 0x002FF814

EBP = 0x002FF8E0

Case 4: N = 2

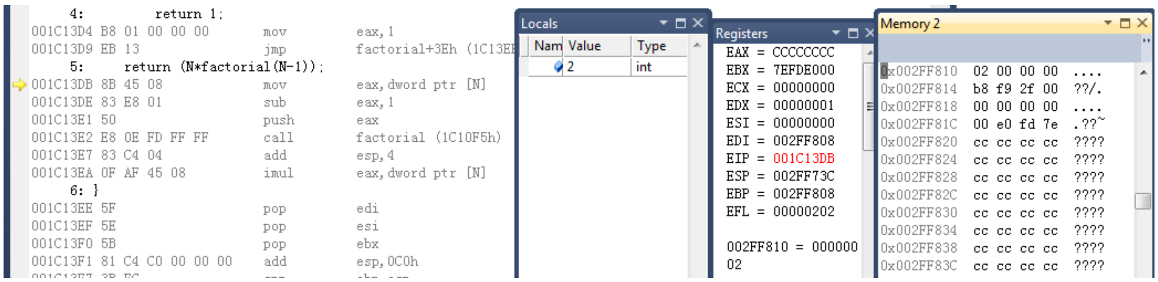


Figure 5- The stack frame when EBP = 002FF808

a) The argument at current level is showing the EIP equals to 0x001C13DB meaning the next instruction is going to execute.

b) Local variable at current level has N = 2.

c) Return address at current level is store at the memory address is computed by using EIP = 0x001C13DB and since the next instruction is 0x001C13DE with length of 3 bytes. So we will add the 3rd value in stack to the EBP to find the address where **N =2 is located;**

0x002FF808 (EBP) + 00000008 = **0x002FF810** in figure 5

EIP = 001C13DB

ESP = 002FF73C

EBP = 002FF808

Case 4: when N = 1

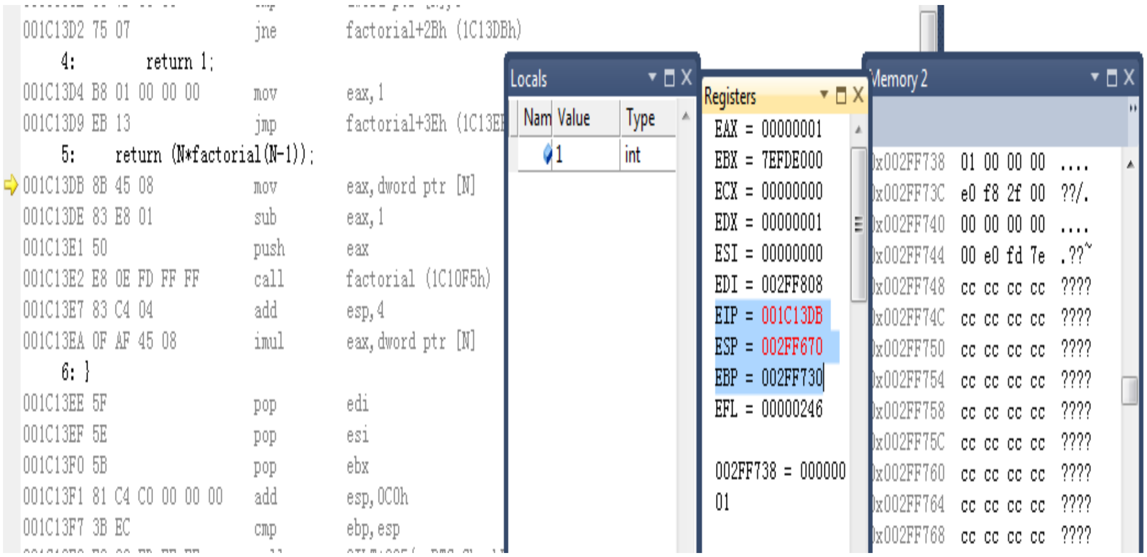


Figure 6 stack frame when EBP = 0x002FF730

a) The argument at current level is showing the EIP equals to 0x001C13DB meaning the next instruction is going to execute.

b) Local variable at current level has N =1.

c) Return address at current level is store at the memory address is computed by using EIP = 0x001C13DB and since the next instruction is 0x001C13DE with length of 3 bytes. So we will add the 3rd value in stack to the EBP to find the address where **N =1 is located;**

0x002FF730 (EBP) + 00000008 = **0x002FF738** in figure 6

EIP = 001C13DB

ESP = 002FF670

EBP = 002FF730

Stack Frame

```
Address: 0x002FF738                                              Columns:
0x002FF738  01 00 00 00 e0 f8 2f 00 00 00 00 00 00 e0 fd 7e cc cc cc cc cc
0x002FF74D  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF762  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF777  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF78C  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF7A1  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF7B6  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF7CB  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF7E0  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF7F5  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc e0 f8
0x002FF80A  2f 00 e7 13 1c 00 02 00 00 00 b8 f9 2f 00 00 00 00 00 00 e0 fd
0x002FF81F  7e cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF834  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF849  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF85E  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF873  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF888  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF89D  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF8B2  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF8C7  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF8DC  cc cc cc cc b8 f9 2f 00 e7 13 1c 00 03 00 00 00 90 fa 2f 00 00
0x002FF8F1  00 00 00 00 e0 fd 7e cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF906  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF91B  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF930  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF945  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF95A  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF96F  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF984  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF999  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF9AE  cc cc cc cc cc cc cc cc cc cc 90 fa 2f 00 e7 13 1c 00 04 00 00
0x002FF9C3  00 74 fb 2f 00 00 00 00 00 00 e0 fd 7e cc cc cc cc cc cc cc cc
0x002FF9D8  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FF9ED  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FFA02  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FFA17  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FFA2C  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FFA41  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FFA56  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FFA6B  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x002FFA80  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc 74 fb 2f 00 45
0x002FFA95  14 1c 00 05 00 00 00 00 00 00 00 00 00 00 00 00 e0 fd 7e cc cc
```

The following stack frame shows the address where each N is located in stack frame

When N is 1:   location of the address is 0x002FF738

When N is 2:   location of the address is 0x002FF810

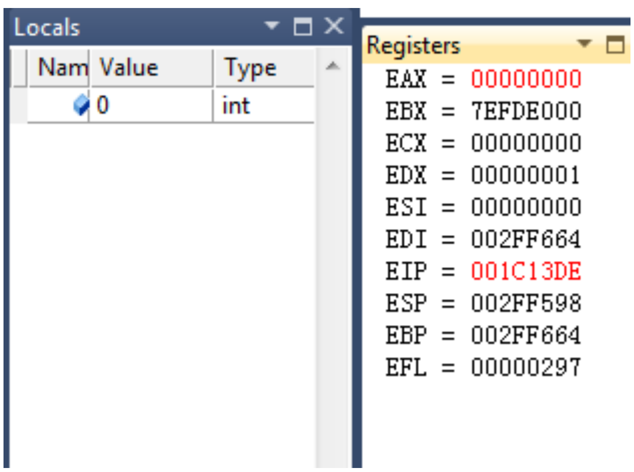
When N is 3:   location of the address is 0x002FF8E8

When N is 4:   location of the address is 0x002FF9C0

When N is 5:   location of the address is 0x002FFA98

Returning process

Register 0



We will expect to see our output is going to be 1 then 2 (2*1) then 6 (3*2) then 24 (4*6) and finally 120 (5*24) in decimal.

We will mainly look at the register EAX because it will be the address where our value will be stored in.

When N is 1:

```
Registers
EAX = 00000001
EBX = 7EFDE000
ECX = 00000000
EDX = 00000001
ESI = 00000000
EDI = 0033F6A4
EIP = 00CC13E7
ESP = 0033F5D4
EBP = 0033F6A4
EFL = 00000246
```

When N is 2:

```
Registers
EAX = 00000002
EBX = 7EFDE000
ECX = 00000000
EDX = 00000001
ESI = 00000000
EDI = 0033F6A4
EIP = 00CC13EE
ESP = 0033F5D8
EBP = 0033F6A4
EFL = 00000202
```

When N is 3:

```
Registers
EAX = 00000006
EBX = 7EFDE000
ECX = 00000000
EDX = 00000001
ESI = 00000000
EDI = 0033F77C
EIP = 00CC13EE
ESP = 0033F6B0
EBP = 0033F77C
EFL = 00000206
```

When N is 4:

```
Registers
EAX = 00000018
EBX = 7EFDE000
ECX = 00000000
EDX = 00000001
ESI = 00000000
EDI = 0033F854
EIP = 00CC13EE
ESP = 0033F788
EBP = 0033F854
EFL = 00000206
```

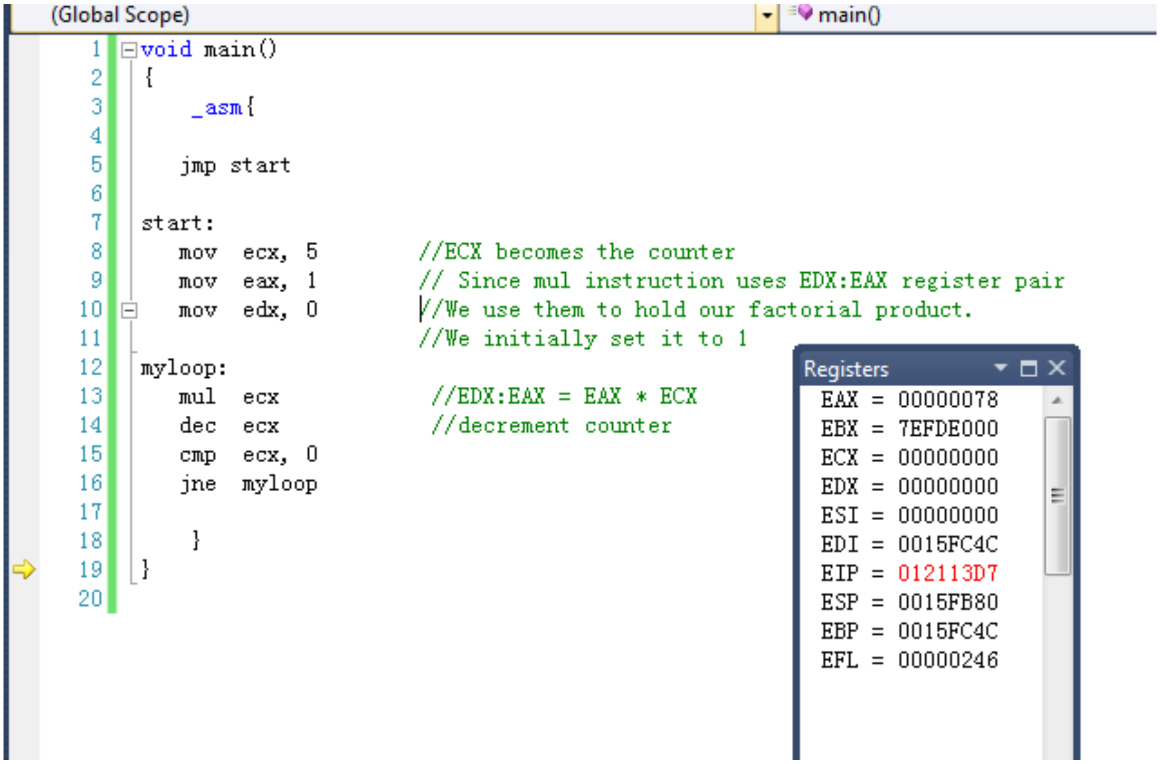
When N is 5:

```
Registers
EAX = 00000078
EBX = 7EFDE000
ECX = 00000000
EDX = 00000001
ESI = 00000000
EDI = 0033F92C
EIP = 00CC13EE
ESP = 0033F860
EBP = 0033F92C
EFL = 00000206
```

78 = 7x16 + 8 = **120 in decimal!**

Part 2: Optimize version of factorial code

Figure 1-Assembly code for Factorial and Register



The above assembly code in .Net environment shows the alternative way to write factorial without using CALL. Instead, I used “jne” which is similar to Jal in MIPS. The program is follows the following steps to compute factorial.

1) Store 5 in Ecx as the counter
2) Store 1 in Eax
3) Stores Edx as 0
4) Recursive step: multiply 5*1 and stores in Eax. Then Ecx continues to decrement by 1 and then multiply it by the previous and result in 5*4*1, then 5*4*3*1.
5) This loop ends until ECX = 0, and the result should be **5*4*3*2*1** store in EAX which is **0x78 = 120** in decimal which shows our result is correct.

Part 3: Factorial in Mars Simulator

Factorial code in MIPS

```
.data

prompt1: .asciiz        "Enter a value: "
prompt2: .asciiz        "The result is: "

.text

main:   li      $v0,4           # prompt the user for n
la      $a0,prompt1
        syscall
        li      $v0,5           #read n into v0
        syscall

        move    $a0,$v0                 #store n in a0
        jal     fact            #call the recursive procedure
move $t0,$v0            #n! is returned in $v0

        li      $v0,4           #output the result
        la      $a0,prompt2
        syscall
        li      $v0,1
        move    $a0,$t0
        syscall

        li      $v0,10          #exit
        syscall

# recursive procedure to calculate the factorial of n. n is passed in $a0, and n! is
returned in $v0
fact:   addi    $sp,$sp,-8      #adjust stack for 2 items
        sw      $ra,4($sp)      #save the return address
        sw      $a0,0($sp)      #save the argument n

        slti    $t0,$a0,1       #test for n < 1
        beq     $t0,$zero,next #if n >= 1, go to next

        addi    $v0,$zero,1     #otherwise, return 1
        addi    $sp,$sp,8       #pop 2 items off stack
        j end                   #go to the end
```

```
next:   addi    $a0,$a0,-1      #decrement n
        jal     fact            #recursive call to fact

        lw      $a0,0($sp)      #restore n from stack to $a0
        lw      $ra,4($sp)      #restore return address from stack to ra
        addi    $sp,$sp,8       #pop 2 items off stack

        mul     $v0,$a0,$v0     #return n * fact(n-1) in v0

end:    jr      $ra             #return from procedure
```

Figure 1 Recursive step from Line 23 - 37

Text Segment

Bkpt	Address	Code	Basic	Source
	0x0040003c	0x0000000c	syscall	23: syscall
	0x00400040	0x2402000a	addiu $2, $0, 0x0000000a	25: li $v0, 10 #exit
	0x00400044	0x0000000c	syscall	26: syscall
	0x00400048	0x23bdfff8	addi $29, $29, 0xfffffff8	29: fact: addi $sp, $sp, -8 #adjust stack for 2 items
	0x0040004c	0xafbf0004	sw $31, 0x00000004 ($29)	30: sw $ra, 4 ($sp) #save the return address
	0x00400050	0xafa40000	sw $4, 0x00000000 ($29)	31: sw $a0, 0 ($sp) #save the argument n
	0x00400054	0x28880001	slti $8, $4, 0x00000001	33: slti $t0, $a0, 1 #test for n < 1
	0x00400058	0x11000003	beq $8, $0, 0x00000003	34: beq $t0, $zero, next #if n >= 1, go to next
	0x0040005c	0x20020001	addi $2, $0, 0x00000001	36: addi $v0, $zero, 1 #otherwise, return 1
	0x00400060	0x23bd0008	addi $29, $29, 0x00000008	37: addi $sp, $sp, 8 #pop 2 items off stack

Figure 2 Recursive step from Line 38 to 49

Text Segment

Bkpt	Address	Code	Basic	Source
	0x0040005c	0x20020001	addi $2, $0, 0x00000001	36: addi $v0, $zero, 1 #otherwise, return 1
	0x00400060	0x23bd0008	addi $29, $29, 0x00000008	37: addi $sp, $sp, 8 #pop 2 items off stack
	0x00400064	0x08100020	j 0x00400080	38: j end #go to the end
	0x00400068	0x2084ffff	addi $4, $4, 0xffffffff	40: next: addi $a0, $a0, -1 #decrement n
	0x0040006c	0x0c100012	jal 0x00400048	41: jal fact #recursive call to fact
	0x00400070	0x8fa40000	lw $4, 0x00000000 ($29)	43: lw $a0, 0 ($sp) #restore n from stack to $a0
	0x00400074	0x8fbf0004	lw $31, 0x00000004 ($29)	44: lw $ra, 4 ($sp) #restore return address from stack to ra
	0x00400078	0x23bd0008	addi $29, $29, 0x00000008	45: addi $sp, $sp, 8 #pop 2 items off stack
	0x0040007c	0x70821002	mul $2, $4, $2	47: mul $v0, $a0, $v0 #return n * fact (n-1) in v0
	0x00400080	0x03e00008	jr $31	49: end: jr $ra #return from procedure

Figure 3 Register Table in MARS

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000004
$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	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400004
hi		0x00000000
lo		0x00000000

Explanation of the factorial code

The factorial code in assembly language begins with prompting user to enter Fact(3) for this example, and store that **3** in register $a0.

1) After that it will enter **"JAL"**, that take the value **3** to a recursive procedure. In the recursive step, it will start with adjust stack for 2 items and save return address and argument of **n** whenever **"n < 1"**.

2) Therefore, we will have each n value store in a lower address denoted by "address – 8". The recursive step will continue until the value **"n < 1"**. When **"n<1",** we will

store that value **1** in $v0.

3) After that, we will add “address +8” to pop the 2 item off the stack and restore the value in that stack to $a0 where our value **“1”** was lastly stored in the recursive step. Then pop 2 item from stack again where we stored value **“2”. Then we will do 2x1 and store value in $v0.**
4) We will continue doing this for **value “3”** as **step (3)** and becomes **2x3 =6** store in $v0.
5) Lastly, we will move value from $v0 to $a0 and output the value of $a0.

Conclusion

The central idea of this take-home exam was mainly help us better understand the procedure of the recursive call works in .Net Environment and in the MARS Simulator. The result of my finding was very surprise in the way different memory played a role at the call step. In the .NET Environment, the EBP changed each time a call is made. We will calculate location of each value in the factorial. Then when all value is done, we will see each value is stored in EAX until the final step is done. In the MARS, we see very similar event such that value is prompted then enters the recursive call until that value is less than 1. Then the program will eventually pop the value and stored in certain register, and lastly output the result in a designed register.