TAKE HOME TEST 2 FACTORIAL

CSC 343 FALL 2017

NOVEMBER, 13 2017

Jeter Gutierrez

TABLE OF CONTENTS

OBJECTIVE: The purpose of this assignment is to test how a Windows 32 bit compiler, Linux 64 bit compiler, 32 Bit Linux compiler and Mips handle recursive calls of functions. We will also be evaluating the runtime of calculating factorial of 5, 10, 100, 1000, and 1000. In order to test the factorial function on Linux 64 bit compiler we will be issuing Ubuntu operating system. In order to test 32 bit Linux compiler we will be using a raspberry pi. In order to test Windows 32 bit compiler we will be using Visual Studio. For both Linux demonstrations we will be using gcc to compile our code and gdb for testing and debugging the assembly code in order to interpret the procedure. For MIPS we will be using Mars MIPS simulator.

PART 1 FACTORIAL ON MIPS SIMULATOR

The purpose of this section is to run a FACTORIAL function on MARS MIPS simulator in order to observe what happens to the stack when we do this. We will then compare the differences between the other three sections. The goal is to understand how recursive functions work in Mips and how the stack of memory is affected.

Fact.asm

```
main:
li $a0, 5               # Setting N=5
jal factorial           # call factorial function
addi $s0, $v0, 0
syscall

factorial:
addi $sp, $sp, -8       # adjust stack for 2 times
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, L1      # if n >= 1, go to L1
addi $v0, $zero, 1      # return 1
addi $sp, $sp, 8        # pop 2 items off stack
jr $ra                  # return to caller

L1:
addi $a0, $a0, -1       # n >= 1: argument gets (n - 1)
jal factorial           # call factorial with (n - 1)
lw $a0, 0($sp)          # return from jal: restore argument n
lw $ra, 4($sp)          # restore the return address
addi $sp, $sp, 8        # adjust stack pointer to pop 2 items
mul $v0, $a0, $v0       # return n * factorial(n - 1)
jr $ra                  # return to caller
```

Figure 1: Mips assembly code for running the factorial function.

We will be running the factorial function for the input of finding the factorial of 5. The value that we will be expecting in return is 120 or 0x78.

Edit | Execute

Text Segment

Bkpt	Address	Code	Basic		
	0x00400000	0x24040005	addiu $4,$0,0x00000005	2: li $a0, 5	# Setting N=5
	0x00400004	0x0c100004	jal 0x00400010	3: jal factorial	# call factorial function
	0x00400008	0x20500000	addi $16,$2,0x00000000	4: addi $s0, $v0, 0	
	0x0040000c	0x0000000c	syscall	5: syscall	
	0x00400010	0x23bdfff8	addi $29,$29,0xffff...	8: addi $sp, $sp, -8	# adjust stack for 2 times
	0x00400014	0xafbf0004	sw $31,0x00000004($29)	9: sw $ra, 4($sp)	# save the return address
	0x00400018	0xafa40000	sw $4,0x00000000($29)	10: sw $a0, 0($sp)	# save the argument n
	0x0040001c	0x28880001	slti $8,$4,0x00000001	11: slti $t0, $a0, 1	# test for n < 1
	0x00400020	0x11000003	beq $8,$0,0x00000003	12: beq $t0, $zero, L1	# if n >= 1, go to L1
	0x00400024	0x20020001	addi $2,$0,0x00000001	13: addi $v0, $zero, 1	# return 1
	0x00400028	0x23bd0008	addi $29,$29,0x0000...	14: addi $sp, $sp, 8	# pop 2 items off stack
	0x0040002c	0x03e00008	jr $31	15: jr $ra	# return to caller
	0x00400030	0x2084ffff	addi $4,$4,0xffffffff	18: addi $a0, $a0, -1	# n >= 1: argument gets (n - 1)
	0x00400034	0x0c100004	jal 0x00400010	19: jal factorial	# call factorial with (n - 1)
	0x00400038	0x8fa40000	lw $4,0x00000000($29)	20: lw $a0, 0($sp)	# return from jal: restore argument n
	0x0040003c	0x8fbf0004	lw $31,0x00000004($29)	21: lw $ra, 4($sp)	# restore the return address
	0x00400040	0x23bd0008	addi $29,$29,0x0000...	22: addi $sp, $sp, 8	# adjust stack pointer to pop 2 items
	0x00400044	0x70821002	mul $2,$4,$2	23: mul $v0, $a0, $v0	# return n * factorial(n - 1)
	0x00400048	0x03e00008	jr $31	24: jr $ra	# return to caller

Figure 2 : Disassembly code for MIPS FACTORIAL function. We can see the immediate operations that will be run when the factorial function is called, we will be stepping through the program in detail in order to make our observations.

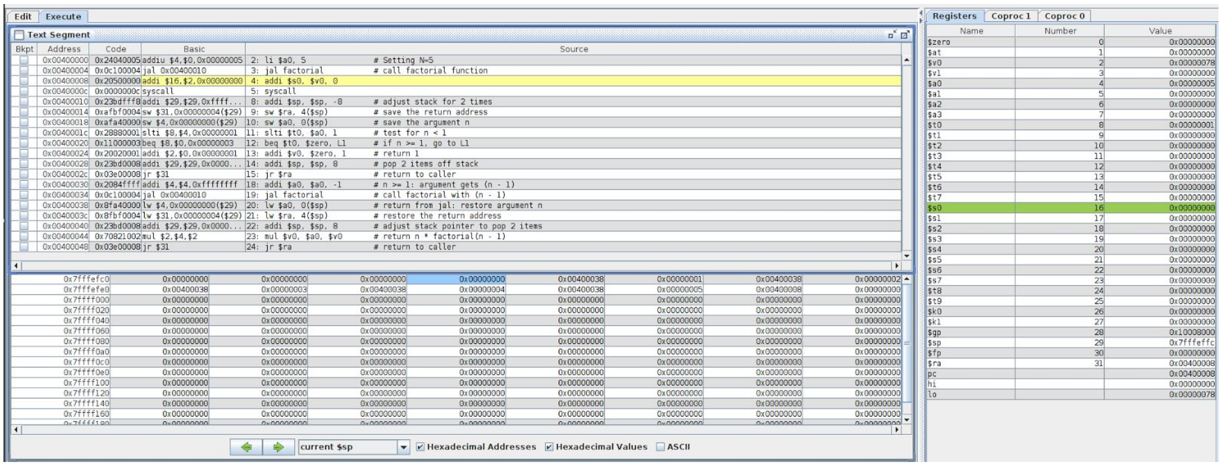

Figure 3: Disassembly code, Data segment and registers for MIPS running the FACTORIAL function. As we can see currently in green is the register where the information will be stored.

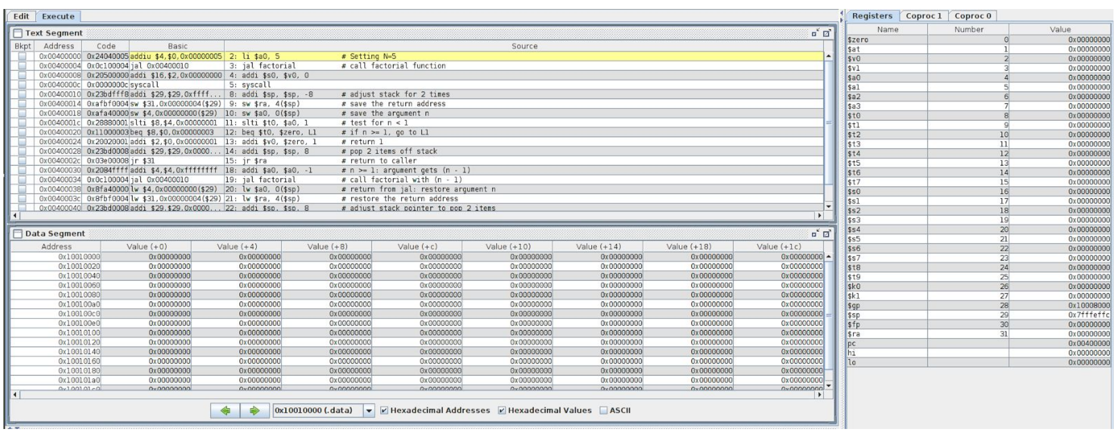

Figure 4: Disassembly code, data segment and registers for running Factorial function on MIPS mars simulator, in the current state everything in the data segment is empty or set to 0x0 we will run through the program and observe what happens in memory as we reach the end of the recursive function call once we reach the end.

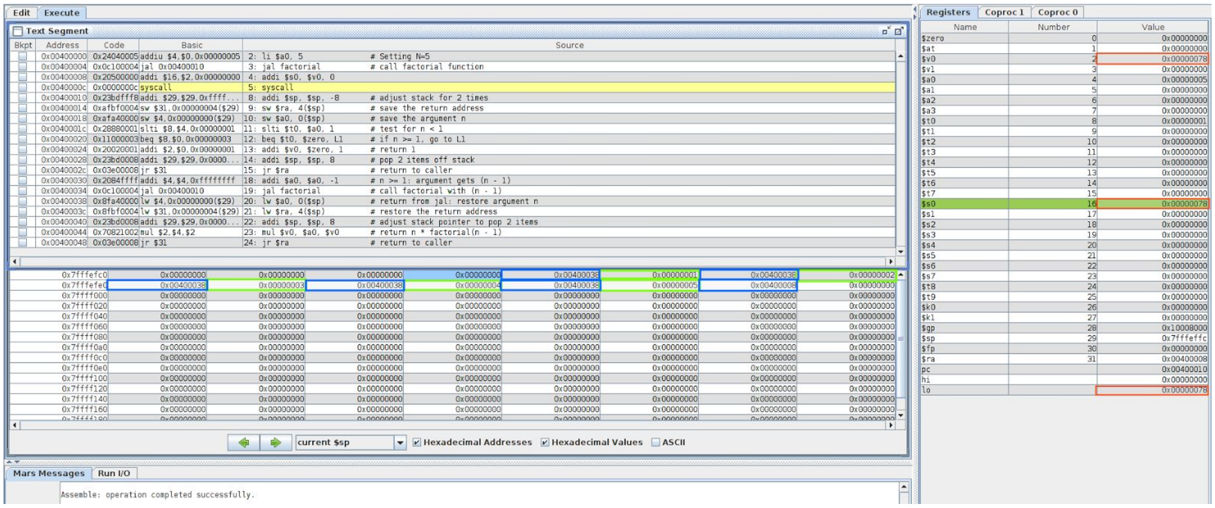

Figure 5: After running the program line by line stepping through the code we can see in the green outlines at the data segment are the values that are being used in our factorial operations, we are running factorial of 5 which means that it is running for the values 5,4,3,2,1 which are stored at a given address that can be displayed in the data segment. In the register LO we can see the value of 0x78 which is 120 in base 10, that is the value that we are looking for when we are computing the factorial of 5. In the blue outline we see the address 0x00400038 which is the address of the instruction that returns from the factorial function. This means that in the MIPS compiler the compiler uses the same return address each time that a function is called recursively. It stores the true value that we are looking for in the register LO, that is the implementation of running a factorial function on mips assembler.

PART 2 FACTORIAL ON 64 BIT LINUX COMPILER

The propose of this section is to run a FACTORIAL function on Linux 64 bit compiler using gcc and then using gdb 64 bit debugger in order to examine what happens in memory when we are using a recursive function on Linux. We would like to observe what happened to the stack and what happens in memory. We will be observing the stack, how functions are called, and how the processor or compiler is capable of keeping track of the order of a recursive program. Situations like keeping track of the addresses to store information and where to get the values for the parameters that are needed for a function call.

```
FACT5.c
int factorial(int N)
{
if(N==1)
return 1;
return(N*factorial(N-1));
}
void main(){int N_fact=factorial(5);}
```

Figure 6: C++ code for running the FACTORIAL function on Linux 64 bit compiler recursively for 5.

```
Breakpoint 2, main () at FACT5.c:7
7	void main(){int N_fact=factorial(5);}
(gdb) disassemble
Dump of assembler code for function main:
   0x0000000000400501 <+0>:	push   %rbp
   0x0000000000400502 <+1>:	mov    %rsp,%rbp
   0x0000000000400505 <+4>:	sub    $0x10,%rsp
=> 0x0000000000400509 <+8>:	mov    $0x5,%edi
   0x000000000040050e <+13>:	callq  0x4004d6 <factorial>
   0x0000000000400513 <+18>:	mov    %eax,-0x4(%rbp)
   0x0000000000400516 <+21>:	nop
   0x0000000000400517 <+22>:	leaveq
   0x0000000000400518 <+23>:	retq
End of assembler dump.
```

Figure 7: Disassembly code for Linux 64 bit compiler running the FACTORIAL function, this is the memory frame for the main function and as we can see in the red box the function callq is the function that will call the address of the FACTORIAL function directly using the addresses.

```
(gdb) disassemble
Dump of assembler code for function factorial:
   0x00000000004004d6 <+0>:     push   %rbp
   0x00000000004004d7 <+1>:     mov    %rsp,%rbp
   0x00000000004004da <+4>:     sub    $0x10,%rsp
   0x00000000004004de <+8>:     mov    %edi,-0x4(%rbp)
   0x00000000004004e1 <+11>:    cmpl   $0x1,-0x4(%rbp)
   0x00000000004004e5 <+15>:    jne    0x4004ee <factorial+24>
   0x00000000004004e7 <+17>:    mov    $0x1,%eax
   0x00000000004004ec <+22>:    jmp    0x4004ff <factorial+41>
   0x00000000004004ee <+24>:    mov    -0x4(%rbp),%eax
   0x00000000004004f1 <+27>:    sub    $0x1,%eax
   0x00000000004004f4 <+30>:    mov    %eax,%edi
   0x00000000004004f6 <+32>:    callq  0x4004d6 <factorial>
   0x00000000004004fb <+37>:    imul   -0x4(%rbp),%eax
=> 0x00000000004004ff <+41>:    leaveq
   0x0000000000400500 <+42>:    retq
End of assembler dump.
```

Figure 8: Disassembly code for Linux 64 bit compiler. This is the memory frame for the FACTORIAL function. S we can see on +32 it is calling itself. +15 is where it checks the base case of n being equal to 1. On +22 it goes into the condition that will call FACTORIAL again if n is not equal to 1. Let's keep in mind that the program is 43 bytes.

```
(gdb) x /40xw $rsp
0x7fffffffdd30: 0x00000000      0x00000000      0x00000000      0x00000001
0x7fffffffdd40: 0xffffdd60      0x00007fff      0x004004fb      0x00000000
0x7fffffffdd50: 0x00000000      0x00000000      0x00000000      0x00000002
0x7fffffffdd60: 0xffffdd80      0x00007fff      0x004004fb      0x00000000
0x7fffffffdd70: 0x00000000      0x00000000      0x00000000      0x00000003
0x7fffffffdd80: 0xffffdda0      0x00007fff      0x004004fb      0x00000000
0x7fffffffdd90: 0x00000000      0x00000000      0x00000000      0x00000004
0x7fffffffdda0: 0xffffddc0      0x00007fff      0x004004fb      0x00000000
0x7fffffffddb0: 0x00000000      0x00000000      0x00000000      0x00000005
0x7fffffffddc0: 0xffffdde0      0x00007fff      0x00400513      0x00000000
```

Figure 9: Memory stack frame for running FACTORIAL function on Linux 64 bit compiler, as we can see in the blue boxes those are the values we will be running the factorial function on each time that it will be called. In the red boxes, the one on the bottom is the return address for the main function, the rest of them are for factorial functions, this is so that when the function call is finished the compiler knows where to return to. In the green we have the return address where the value will be stored. The compiler keeps track of the entire frame as it progresses through the recursive function calls, as it makes a new call it has to allocate more space to show

where the next memory address to store is, we are looking at the stack pointer rsp in order to display the information that is stored on the stack in memory.

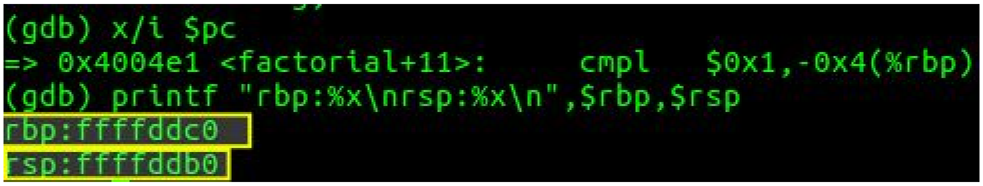

Figure 10: The addresses of the base pointer and the stack pointer are 0xffffddc0 and 0xffffddb0 which if we notice in the previous image is one of the previous addresses that are return locations for the termination of the Factorial function call.

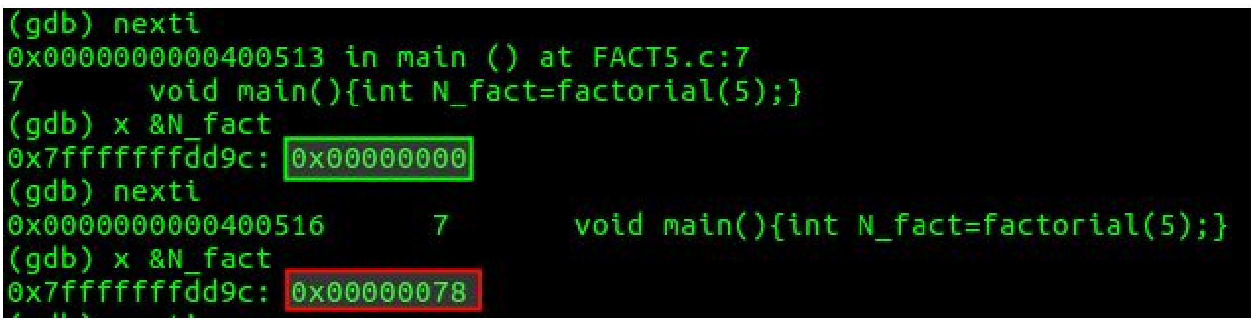

Figure 11: Displaying result of the end of the factorial function call on Linux 64 bit compiler. The value stored at the variable N_fact is 0x78 which is 120, but as we can see in the green box, the variable is not updated as we step through the recursive factorial calls, it updates only after the last time that we return from the function. That is why it is important for the compiler to keep track of the previous addresses for functions and stack frames in order to remember the structure of returning to the main function and storing the value we are looking for correctly.

As we can see in comparison to MIPS when we call the factorial function recursively on Linux 64 bit compiler it does not update the variable we are looking for as we step through the program, it only updates it at the end of the first function call, unlike MIPS that store the value in a register LO in order for us to know the value sa we are stepping through the program before it is finished.

PART 3 FACTORIAL RASPBERRY PI / 32 BIT LINUX COMPILER

The purpose of this section is to run a FACTORIAL program recursively on a 32 bit Linux operating system using gcc and gdb. We will be observing what happens to the stack, to the memory and how the program keeps track of parameters and addresses. We will be looking at the differences between this section and the other three compilers that we will be testing.

FACT5.c

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

Figure 12: C++ code for calling the FACTORIAL function on Linux 32 bit compiler using a Raspberry pi.

```
7       void main(){int N_fact=factorial(5);}
(gdb) disassemble
Dump of assembler code for function main:
   0x00010434 <+0>:     push    {r11, lr}
   0x00010438 <+4>:     add     r11, sp, #4
   0x0001043c <+8>:     sub     sp, sp, #8
=> 0x00010440 <+12>:    mov     r0, #5
   0x00010444 <+16>:    bl      0x103e8 <factorial>
   0x00010448 <+20>:    str     r0, [r11, #-8]
   0x0001044c <+24>:    sub     sp, r11, #4
   0x00010450 <+28>:    pop     {r11, pc}
End of assembler dump.
```

Figure 13: Disassembly code for running FACTORIAL function on Linux 32 bit compiler, we can see the address of calling the FACTORIAL function from within the main function, is 0x103e8 and the address for the instruction is 0x00010444.

```
(gdb) disassemble
Dump of assembler code for function factorial:
   0x000103e8 <+0>:     push    {r11, lr}
   0x000103ec <+4>:     add     r11, sp, #4
   0x000103f0 <+8>:     sub     sp, sp, #8
   0x000103f4 <+12>:    str     r0, [r11, #-8]
=> 0x000103f8 <+16>:    ldr     r3, [r11, #-8]
   0x000103fc <+20>:    cmp     r3, #1
   0x00010400 <+24>:    bne     0x1040c <factorial+36>
   0x00010404 <+28>:    mov     r3, #1
   0x00010408 <+32>:    b       0x10428 <factorial+64>
   0x0001040c <+36>:    ldr     r3, [r11, #-8]
   0x00010410 <+40>:    sub     r3, r3, #1
   0x00010414 <+44>:    mov     r0, r3
   0x00010418 <+48>:    bl      0x103e8 <factorial>
   0x0001041c <+52>:    mov     r2, r0
   0x00010420 <+56>:    ldr     r3, [r11, #-8]
   0x00010424 <+60>:    mul     r3, r3, r2
   0x00010428 <+64>:    mov     r0, r3
   0x0001042c <+68>:    sub     sp, r11, #4
   0x00010430 <+72>:    pop     {r11, pc}
End of assembler dump.
```

Figure 14: Disassembly code for the FACTORIAL function we can see here that it calls itself at +48 in the third yellow square. We can see that the way that it functions is by looking at two different conditions using bne and b. The first one is basically saying that if n is 1 we should return 1 and exit the function as well as the functions stack frame. The second condition is saying that if n is any value other than 1, we should create a stack for the function again keeping track of the previous one and using a value that is 1 less than the current n.

```
(gdb) nexti 3
0x00010418      5         return(N*factorial(N-1));
(gdb) x/i $pc
=> 0x10418 <factorial+48>:      bl      0x103e8 <factorial>
(gdb) printf "r11:%x\nsp:%x\n",$r11,$sp
r11:7effefcc
sp:7effefc0
```

Figure 15: The addresses of the base pointer and the stack pointer are 0x7effefcc and 0x7effefc0 which if we notice in the next image is one of the previous addresses that are return locations for the termination of the Factorial function call.

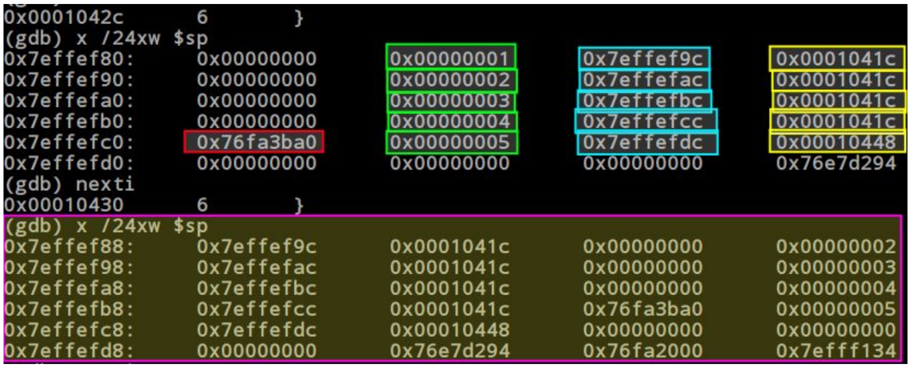

Figure 16: Stack frame from the stack pointer in memory for the Linux 32 bit compiler. As we can see in green the values that will be used for operating the FACTORIAL function at each given call of the recursive function. In the red we can see the address that will be returned at the end of the function call because it is the address of the main function stack frame. This is important and necessary because with it the compiler is capable of returning to the stack of the previous function and continue running the main program in the address 0x76fa3ba0. In the blue we can see the return address for each of the operations as the frame is called each time it remembers the previous tack in order to return to the proper function. In yellow we can see the instruction from the assembly.

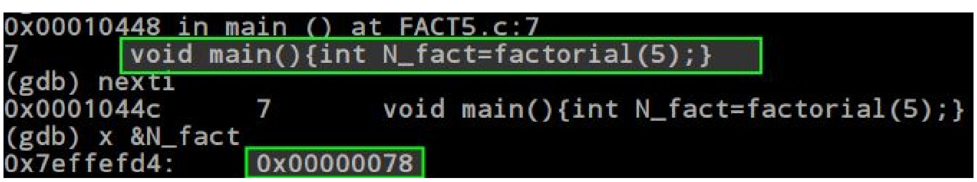

Figure 17: Result page after the final recursive factorial function sends its value to the proper return address. As we can see in this situation as well the value of N_Fact is 0x78 which is the proper expected value of 120 in decimal format. Making recursive function calls on Linux 32 bit compiler is similar to the 64 bit Linux compiler. The result is the same across the last 3 platforms and the structure is very similar in terms of using return addresses and keeping track of the

previous address of the top of the previously used stack like keeping track of the main stack frame after calling the Factorial function for the first time etc.

PART 4 FACTORIAL ON 32 BIT WINDOWS COMPILER

The purpose of this section is to test the FACTORIAL function again but on a Windows 32 bit compiler. We want to look at the differences between this section and the other three sections. We want to observe how efficient or careful this compiler is when managing recursive function calls. Instead of using MARS simulator or gcc or gdb we will be using Visual Studio as the compiling and debugging program for this example and to analyze what happens.

FACT5.c

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

Figure 18: C++ code for the FACTORIAL function that will be called on Windows 32 bit compiler using Visual studio, we will be observing what happens to the stack when the function is called and how variables are handled in memory, then we will compare them to the previous 3 sections.

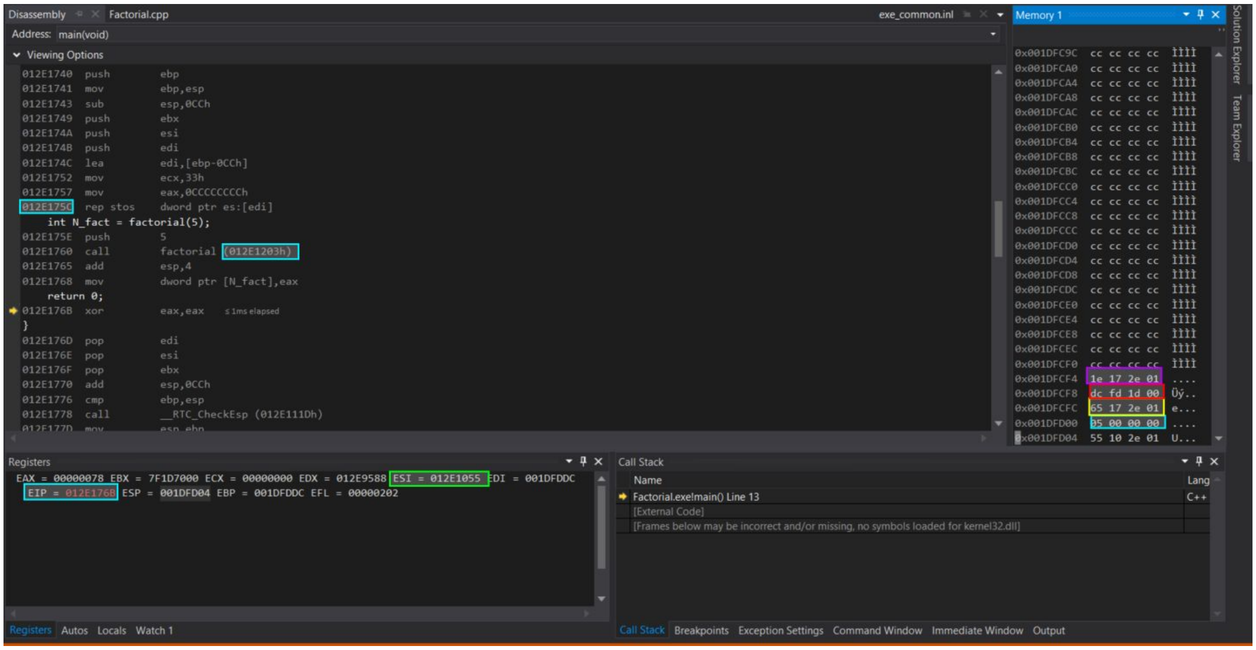

Figure 19: Disassembly windows, memory window and registers window of calling the FACTORIAL function in windows 32 bit compiler. We can see the address of the factorial function is accessed directly and is 0X012e1203. If we look to the bottom right we can see the value of the parameter for the factorial function, in blue, followed by the address of the main function stack, the return address and a new address for the function call.

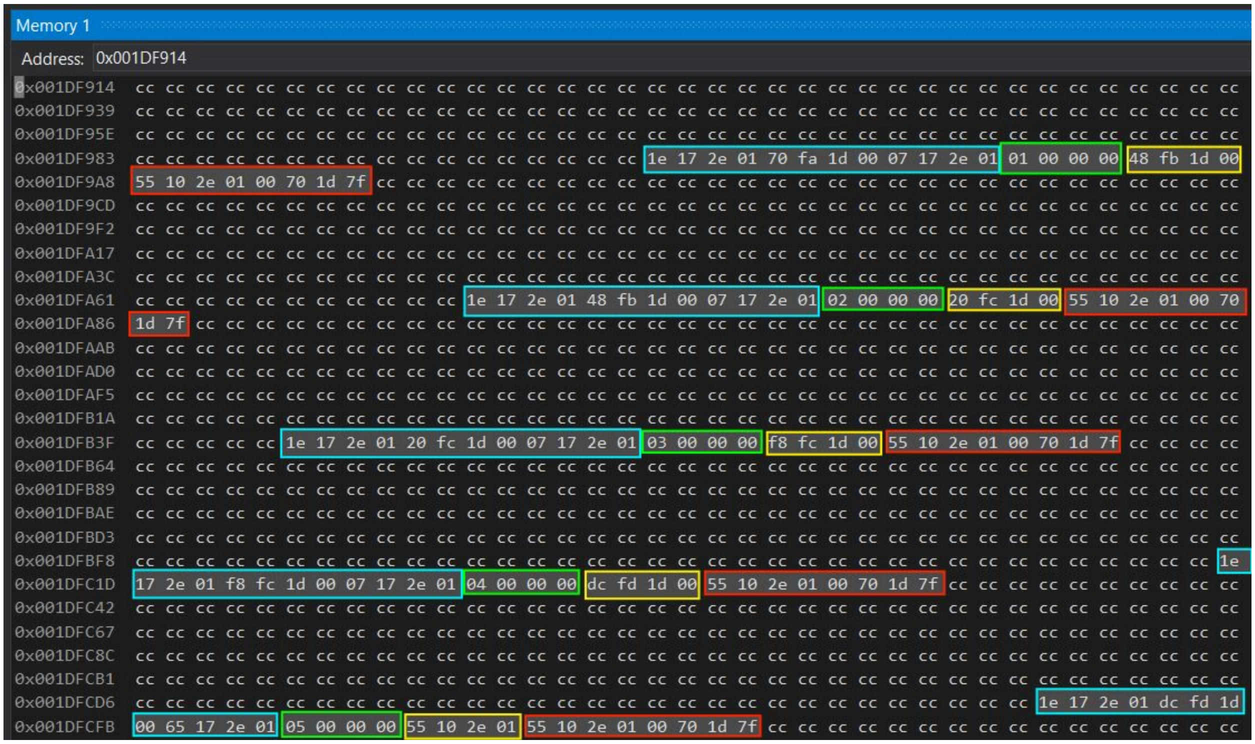

Figure 20: Memory window for Factorial function on windows 32 bit compiler on Visual studio. In green we can see each of values in Big endian in this case on this specific processor. In yellow we can see the address of the previously called function, the reason for that is because it is keeping track of where to return in the stack after the debugger is finished running through the program. In red we can see the return address to store the value being used, and in blue we can see the address of the instruction. Every index that is labeled as cc is important the values are there because windows 32 bit compile on visual studio allocates 256 bytes of data for each of the function calls in order to potentially have enough space for the performance of the memory that is going ot be needed for running the function and memory allocation as well.

PART 5 FACTORIAL RUNTIME AND STACK ANALYSIS

As we can notice the running time for running Factorial becomes a Linear Running time that is because each computation takes a specific amount of time to computer, it takes a constant time to compute however, when we are running Factorial for larger values then it depends on the how many times the function is called let's say N. That would still be a linear running time it will take the same amount of time to multiply any two numbers regardless of how many times the function is called. The only concern here is for running Factorial of 10,000 on Windows compiler, this is a problem because as we can remember when a recursive function is called Visual studio allocates 256 byte of memory for the stack of the function, doing that 10,000 time will lead in an stack overflow because the computer will run out of data. The same goes for Linux 32 bit compiler and 64 bit compiler, the only other factor is that for MIPS it might be capable of performing that operation without leading to a stack overflow because it does not allocate as much memory for the stack of a recursive call, however the number will not fit in 64 bits so we would need a different type to store the number, maybe using pointers to store such a large number.

In summary each compiler, MIPS, Linux 64, Linux 32 bit and windows 32 bit compiler handles recursive function calls differently.