Intel x86 on visual studio

Bello Melido

10/24/2023

CSC 210

Table of content

Objective 3
Introduction 3
Intel X86 on MS Visual Studio 5
Factorial c code with QueryPerformance 5
Results 6
Graph 6
Disassembly 7
Main call 7
Factorial Call From MeasureFactorial 8
First Call
mov ebp, esp sets the base pointer (EBP) to the current stack pointer (ESP), creating a new stack frame 9
Second call 11
Third call 11
Returning Calls 12
GCD 14
C code 14
Results 15
Dissasembly 15
Main code 17
First called 18
Second Called 18
Third Called 18
Forth Called 19
Fifth Called 19
Sixth Called 19
DEFINITIONS 20
How does the factorial function works? 20
How does the gcd works? 20

Objective

The aim of this test is to deepen and exhibit proficiency in recursive function calls and the management of stack frames across multiple architectures. Specifically, the test targets the MIPS instruction set architecture, the Intel x86 ISA using MS Visual Studio's 32-Bit compiler and debugger, and the Intel x86 64-bit architecture under a Linux environment utilizing the 64-bit GCC and GDB. This comprehensive examination will enable students to compare and contrast the handling of recursive functions and stack frame operations in different computing environments, providing a holistic understanding of these fundamental concepts in computer architecture and software development.

Introduction

A Stack Frame is a crucial element in the management of function calls within a program's execution. It represents a designated area of memory assigned on the stack each time a function is invoked. The creation of a new stack frame is a multi-step process beginning with the preservation of the old base pointer, followed by establishing a new base pointer. Memory allocation for the frame is then achieved by adjusting the stack pointer, allowing space for necessary data. Arguments for the function call and local variables are then positioned relative to the new base pointer. The final step involves storing the return address, which the program will revert to after the function's execution.

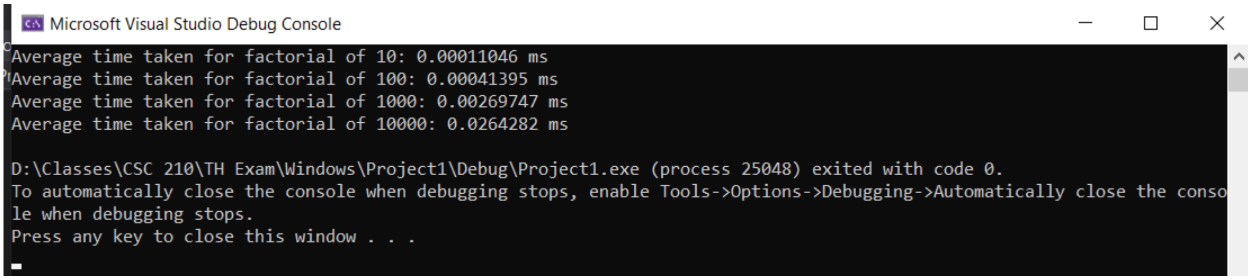
In this test, we will focus on the behavior and structure of Stack Frames, particularly in the context of recursive function calls. Recursive calls provide an excellent framework for understanding Stack Frames, as they involve repeated function calls where each call creates its own frame. This test will explore how these frames are managed and manipulated differently across MIPS, Intel x86 (via MS Visual Studio), and Intel x86 64-bit (Linux platform) architectures. Students will gain insights into the nuances of stack management in these varied environments, enhancing their understanding of both software and hardware interactions in program execution.

Intel X86 on MS Visual Studio

Factorial c code with QueryPerformance

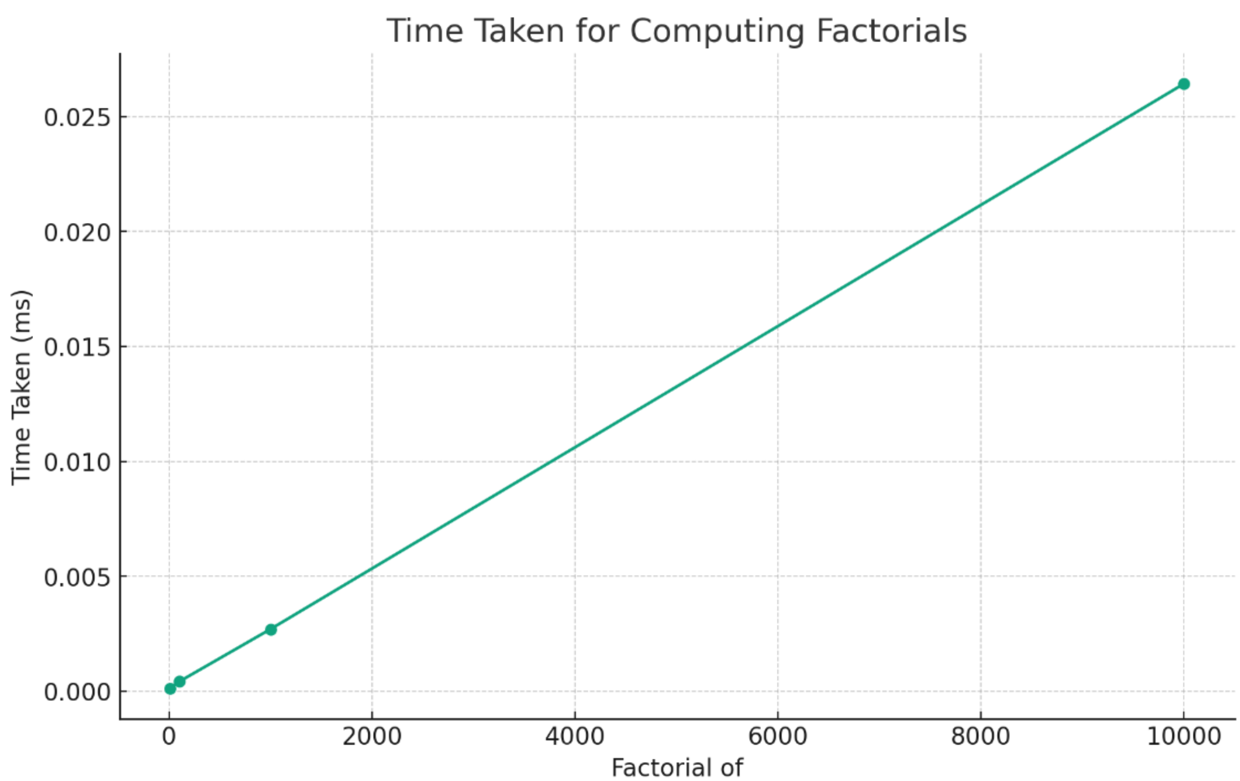
```cpp
#include <iostream>
#include <windows.h>

using namespace std;

double PCFreq = 0.0;
LONGLONG CounterStart = 0;

void StartCounter() {
    LARGE_INTEGER li;
    if (!QueryPerformanceFrequency(&li))
        cout << "QueryPerformanceFrequency failed!\n";

    PCFreq = double(li.QuadPart) / 1000.0;
    QueryPerformanceCounter(&li);
    CounterStart = li.QuadPart;
}

double GetCounter() {
    LARGE_INTEGER li;
    QueryPerformanceCounter(&li);
    return double(li.QuadPart - CounterStart) / PCFreq;
}

int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; ++i) {
        result *= i;
    }
    return result;
}

void measureFactorial(int N, int numMeasurements) {
    double totalTime = 0.0;
    for (int i = 0; i < numMeasurements; ++i) {
        StartCounter();
        factorial(N);
        totalTime += GetCounter();
    }
    double averageTime = totalTime / numMeasurements;
    cout << "Average time taken for factorial of " << N << ": " << averageTime << " ms\n";
}

int main() {
    const int numMeasurements = 10000;
    measureFactorial(10, numMeasurements);
    measureFactorial(100, numMeasurements);
    measureFactorial(1000, numMeasurements);
    measureFactorial(10000, numMeasurements);

    return 0;
}
```

Results



```
Microsoft Visual Studio Debug Console
Average time taken for factorial of 10: 0.00011046 ms
Average time taken for factorial of 100: 0.00041395 ms
Average time taken for factorial of 1000: 0.00269747 ms
Average time taken for factorial of 10000: 0.0264282 ms

D:\Classes\CSC 210\TH Exam\Windows\Project1\Debug\Project1.exe (process 25048) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Graph

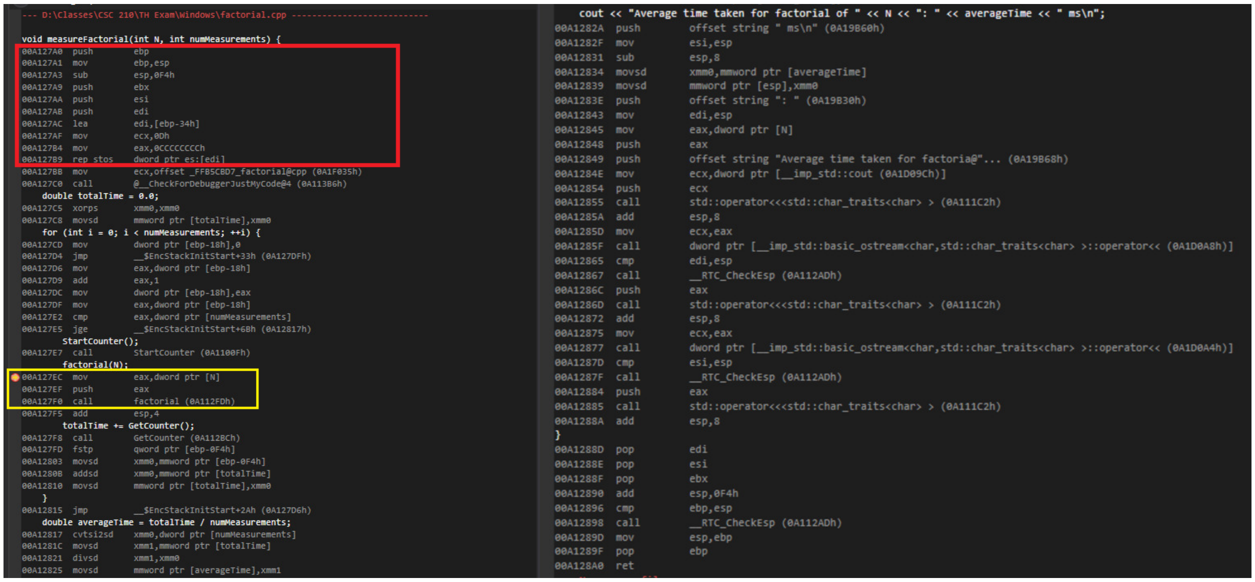


Disassembly

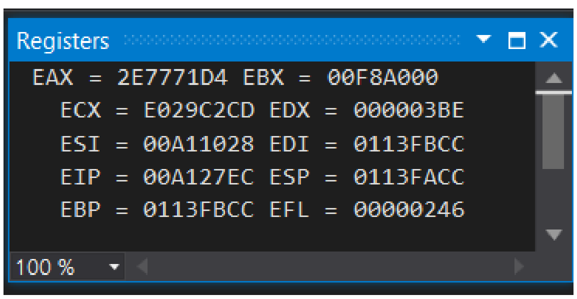
Main call

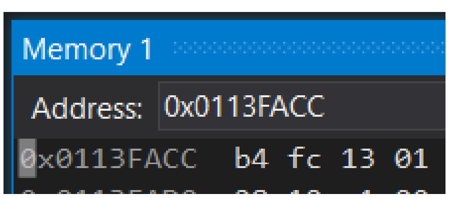
```
--- D:\Classes\CSC 210\TH Exam\Windows\factorial.cpp --------------------------

int main() {
00202900  push        ebp
00202901  mov         ebp,esp
00202903  sub         esp,0CCh
00202909  push        ebx
0020290A  push        esi
0020290B  push        edi
0020290C  lea         edi,[ebp-0Ch]
0020290F  mov         ecx,3
00202914  mov         eax,0CCCCCCCCh
00202919  rep stos    dword ptr es:[edi]
0020291B  mov         ecx,offset _FFB5CBD7_factorial@cpp (020F035h)
00202920  call        @__CheckForDebuggerJustMyCode@4 (02013B6h)
    const int numMeasurements = 10000;
00202925  mov         dword ptr [numMeasurements],2710h
    measureFactorial(10, numMeasurements);
0020292C  push        2710h
00202931  push        0Ah
00202933  call        measureFactorial (020115Eh)
00202938  add         esp,8
    measureFactorial(100, numMeasurements);
0020293B  push        2710h
00202940  push        64h
00202942  call        measureFactorial (020115Eh)
00202947  add         esp,8
    measureFactorial(1000, numMeasurements);
0020294A  push        2710h
0020294F  push        3E8h
00202954  call        measureFactorial (020115Eh)
00202959  add         esp,8
    measureFactorial(10000, numMeasurements);
0020295C  push        2710h
00202961  push        2710h
00202966  call        measureFactorial (020115Eh)
0020296B  add         esp,8

    return 0;
0020296E  xor         eax,eax
}
00202970  pop         edi
00202971  pop         esi
00202972  pop         ebx
00202973  add         esp,0CCh
00202979  cmp         ebp,esp
0020297B  call        __RTC_CheckEsp (02012ADh)
00202980  mov         esp,ebp
00202982  pop         ebp
00202983  ret
```

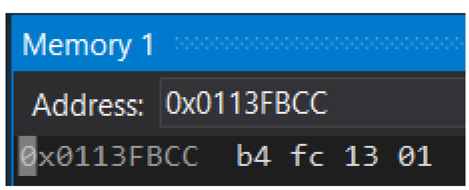
Factorial Call From MeasureFactorial



The preliminary instructions (boxed in red) initializes a new stack frame with stack pointer value 0x0113FACC and base pointer value 0x0113FBCC.







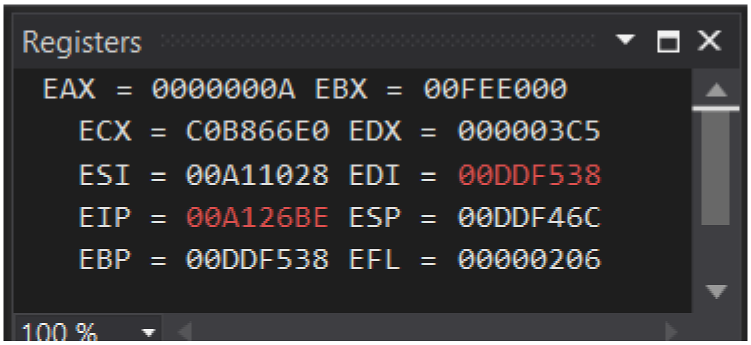
First Call

```
int factorial(int n) {
00A126B0  push        ebp    ≤ 1ms elapsed
00A126B1  mov         ebp,esp
00A126B3  sub         esp,0C0h
00A126B9  push        ebx
00A126BA  push        esi
00A126BB  push        edi
00A126BC  mov         edi,ebp
00A126BE  xor         ecx,ecx
00A126C0  mov         eax,0CCCCCCCCh
00A126C5  rep stos    dword ptr es:[edi]
00A126C7  mov         ecx,offset _FFB5CBD7_factorial@cpp (0A1F035h)
00A126CC  call        @__CheckForDebuggerJustMyCode@4 (0A113B6h)
    if (n == 1) return 1;
00A126D1  cmp         dword ptr [n],1
00A126D5  jne         __$EncStackInitStart+22h (0A126DEh)
00A126D7  mov         eax,1
00A126DC  jmp         __$EncStackInitStart+35h (0A126F1h)
    return (n * factorial(n - 1));
00A126DE  mov         eax,dword ptr [n]
00A126E1  sub         eax,1
00A126E4  push        eax
00A126E5  call        factorial (0A112FDh)
00A126EA  add         esp,4
00A126ED  imul        eax,dword ptr [n]
}
00A126F1  pop         edi
00A126F2  pop         esi
00A126F3  pop         ebx
00A126F4  add         esp,0C0h
00A126FA  cmp         ebp,esp
00A126FC  call        __RTC_CheckEsp (0A112ADh)
00A12701  mov         esp,ebp
00A12703  pop         ebp
00A12704  ret
```

mov ebp, esp sets the base pointer (EBP) to the current stack pointer (ESP), creating a new stack frame.

sub esp, 0D8h allocates space on the stack for local variables.

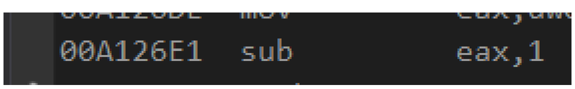
add esp, 0D8h deallocates the allocated stack space when the function exits.



EAX = 0A this is factorial of 10

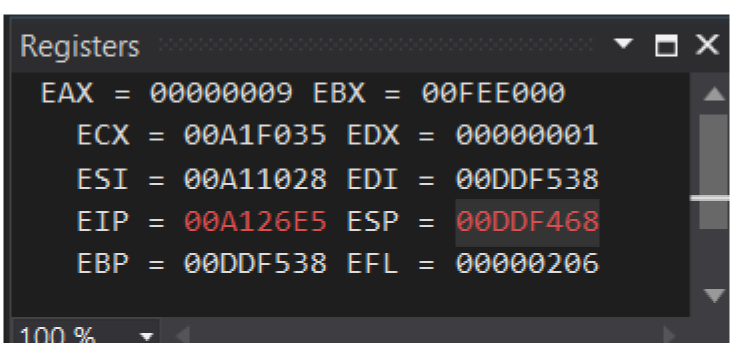
The first set of instructions (in yellow) that will initialize a new stack frame with stack pointer 0x00DDF46C and base pointer 0x00DDF538.

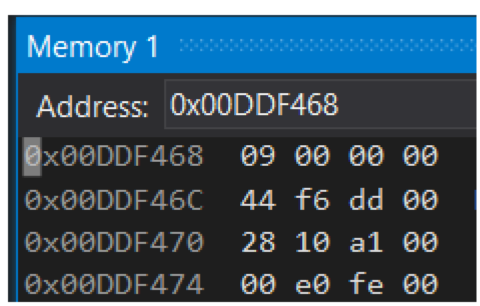
The next instructions (in green) will check if the argument value (N, in this case 10) is equal to 1 and perform a jump if they are not equal. Since they are not equal, a jump is performed. The next sequence of instructions (in red) are executed. First, the argument n is copied into register EAX, then it is decremented and pushed back onto the stack at memory location 0x00D3F6E0. This new value will be the argument used in the next function call.



Here it subtract 1 to eax. It is preparing for factorial of 9.

And this value will be push into the stack in the call factorial in the red rectangle .

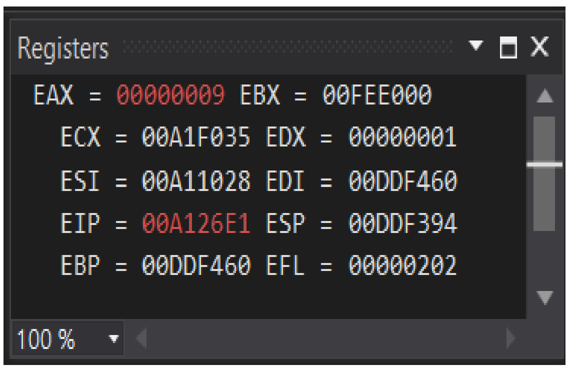


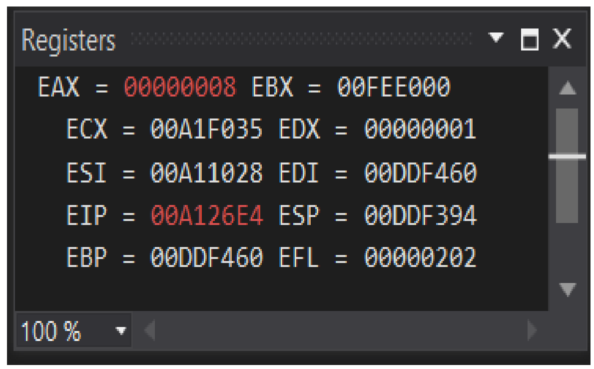


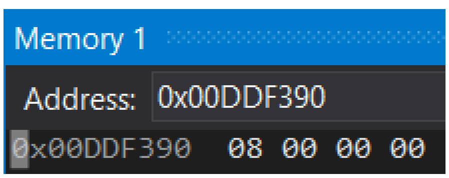
```
00A126EA  add         esp,4
```

Than, add esp, 4 instruction is used to remove the return address from the stack because it's no longer needed. And this process is repeated till factorial (1).

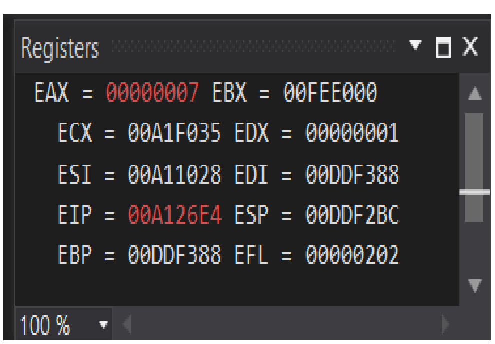
Second call

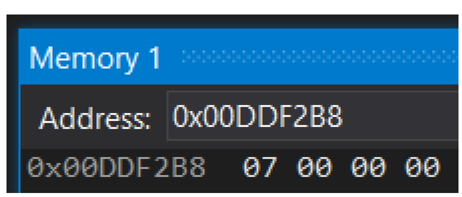






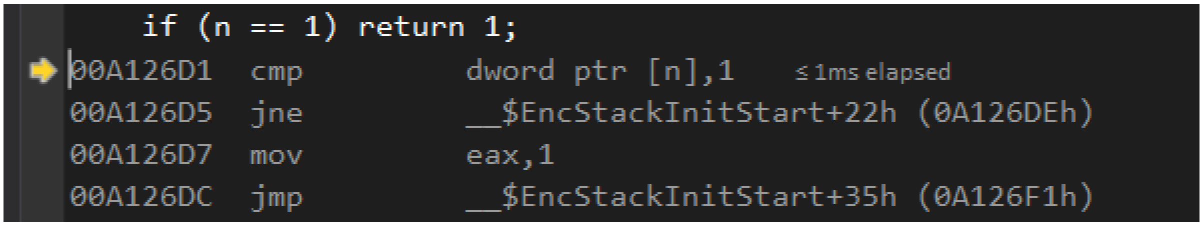
Third call



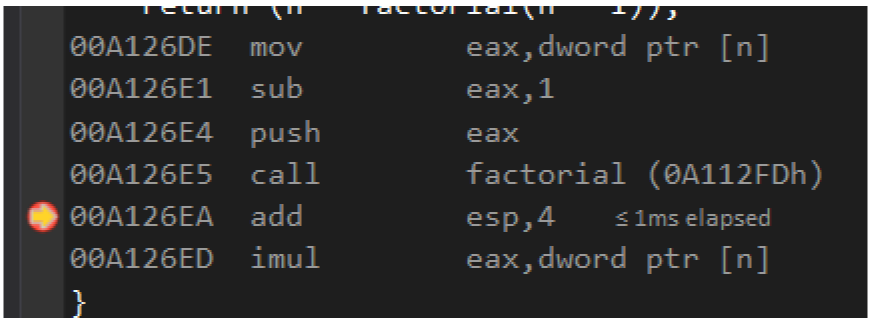


THIS IS REPEATED UPDATING ESP, EBP AND MEMORY ADDRESS VALUE TILL IT EQUALS TO 1 AND STARTS RETURNING.

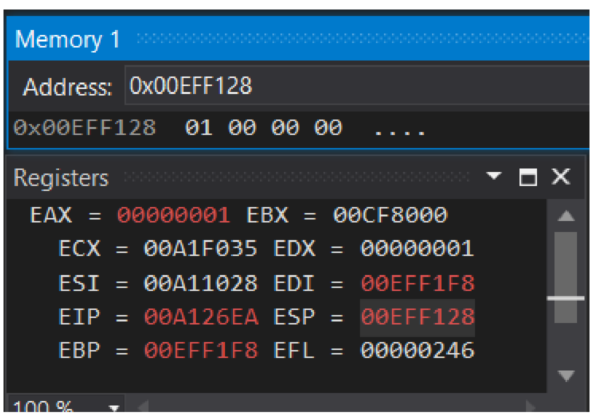
Returning Calls



After it reaches n == 1.

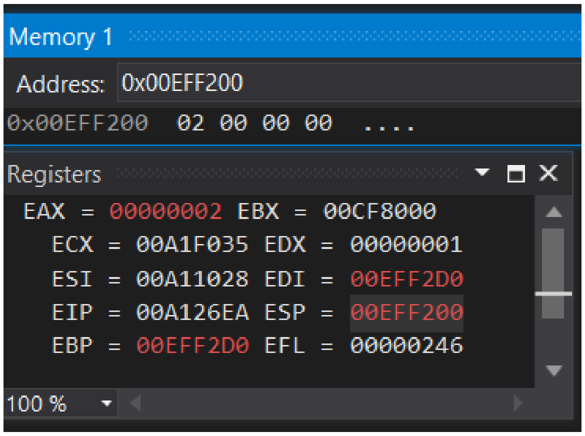


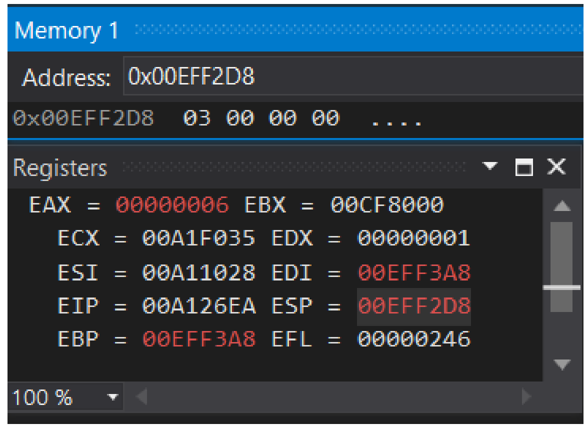
In add it starts changing EAX to the factorial of 1 and the stack pointer points to the amount of iterations.



EAX factorial of 2

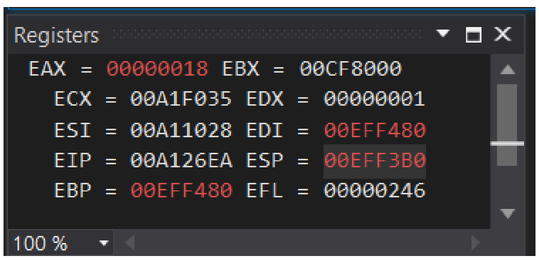
EAX factorial of 3

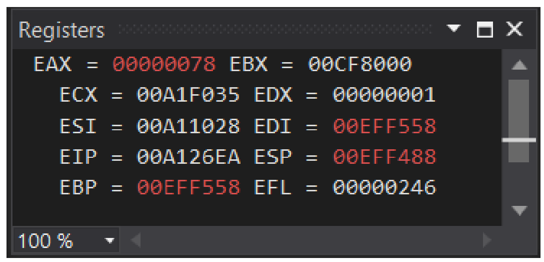




EAX factorial of 4

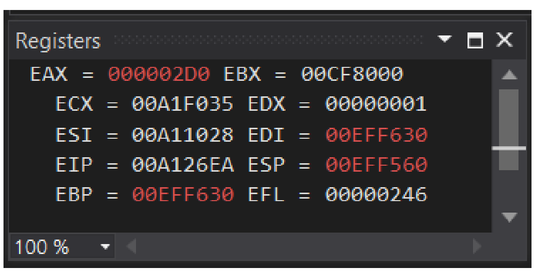
EAX factorial of 5

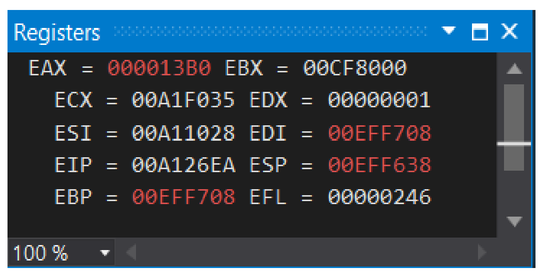




EAX factorial of 6

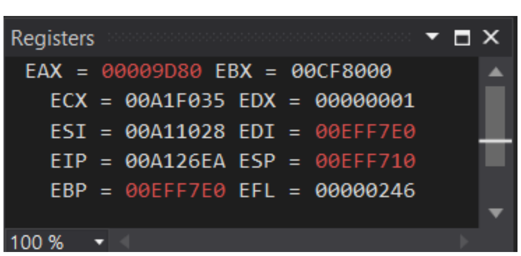
EAX factorial of 7

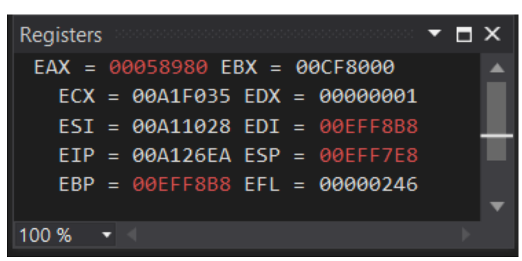




EAX factorial of 8

EAX factorial of 9



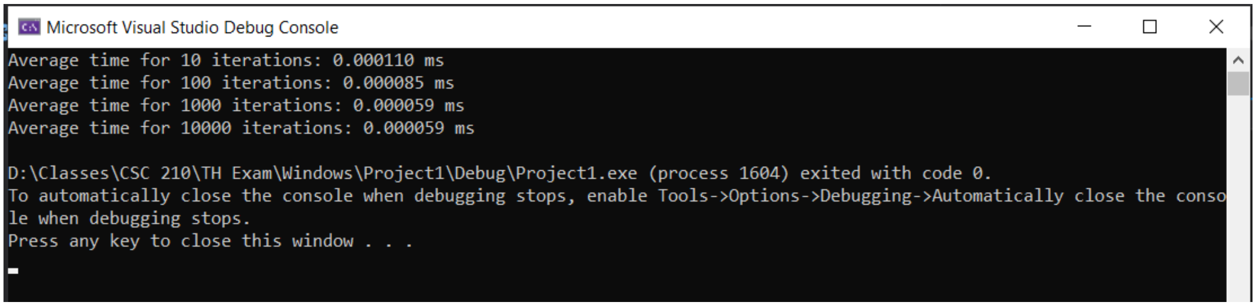


GCD

C code

```c
#include <stdio.h>
#include <windows.h>

LARGE_INTEGER frequency;        // ticks per second
LARGE_INTEGER start, end;       // ticks

int gcd_recurs(int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd_recurs(b, a % b);
}

void StartCounter() {
    QueryPerformanceFrequency(&frequency);
    QueryPerformanceCounter(&start);
}

double GetCounter() {
    QueryPerformanceCounter(&end);
    return (double)(end.QuadPart - start.QuadPart) / frequency.QuadPart;
}

void measureGCD(int a, int b, int N) {
    StartCounter();
    for (int i = 0; i < N; i++) {
        gcd_recurs(a, b);
    }
    double timeTaken = GetCounter() * 1000; // Convert to milliseconds
    printf("Average time for %d iterations: %f ms\n", N, timeTaken / N);
}

int main() {
    int a = 56;  // Example values
    int b = 98;

    measureGCD(a, b, 10);
    measureGCD(a, b, 100);
    measureGCD(a, b, 1000);
    measureGCD(a, b, 10000);

    return 0;
```

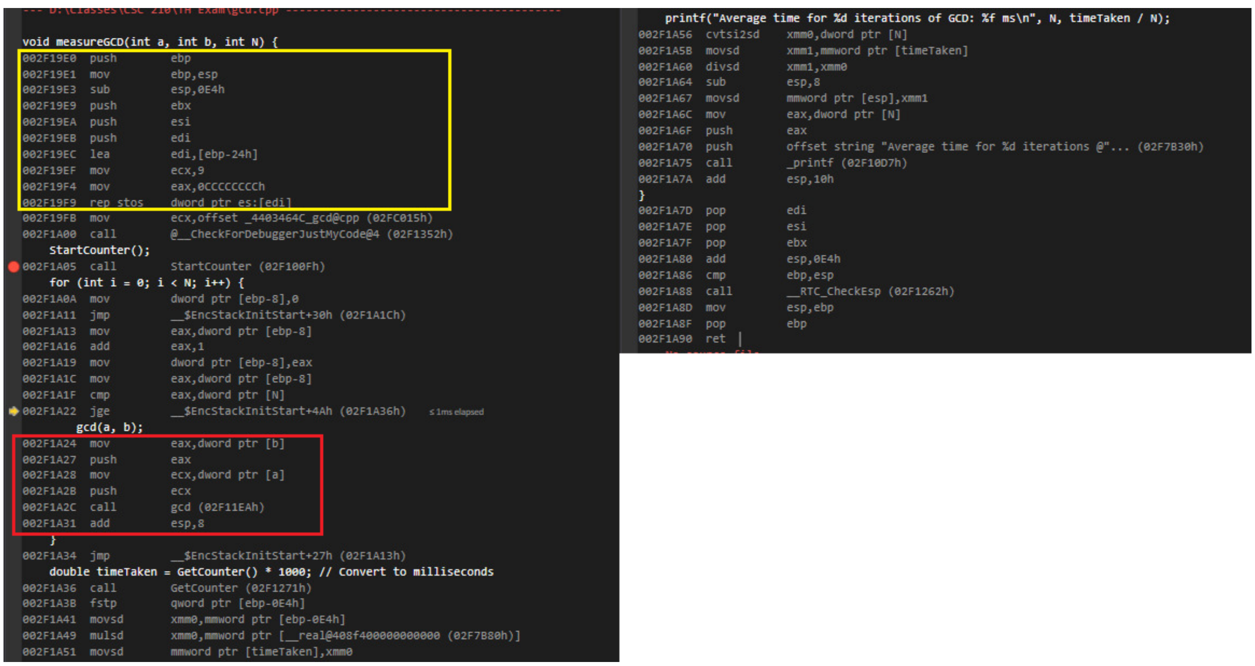
Results



```
Average time for 10 iterations: 0.000110 ms
Average time for 100 iterations: 0.000085 ms
Average time for 1000 iterations: 0.000059 ms
Average time for 10000 iterations: 0.000059 ms

D:\Classes\CSC 210\TH Exam\Windows\Project1\Debug\Project1.exe (process 1604) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

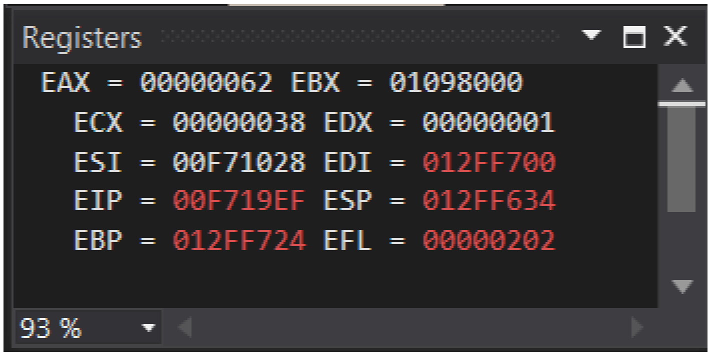
Dissasembly

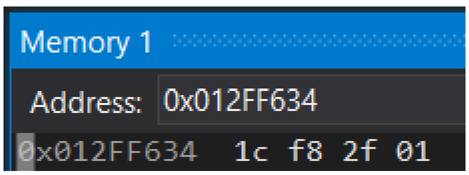
GCD call from measureGCD

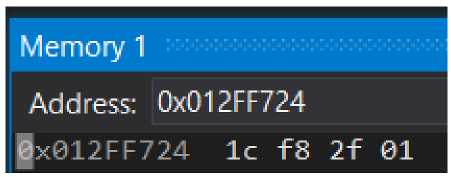


```
void measureGCD(int a, int b, int N) {
002F19E0  push        ebp
002F19E1  mov         ebp,esp
002F19E3  sub         esp,0E4h
002F19E9  push        ebx
002F19EA  push        esi
002F19EB  push        edi
002F19EC  lea         edi,[ebp-24h]
002F19EF  mov         ecx,9
002F19F4  mov         eax,0CCCCCCCCh
002F19F9  rep stos    dword ptr es:[edi]
002F19FB  mov         ecx,offset _4403464C_gcd@cpp (02FC015h)
002F1A00  call        @__CheckForDebuggerJustMyCode@4 (02F1352h)
    StartCounter();
002F1A05  call        StartCounter (02F100Fh)
    for (int i = 0; i < N; i++) {
002F1A0A  mov         dword ptr [ebp-8],0
002F1A11  jmp         __$EncStackInitStart+30h (02F1A1Ch)
002F1A13  mov         eax,dword ptr [ebp-8]
002F1A16  add         eax,1
002F1A19  mov         dword ptr [ebp-8],eax
002F1A1C  mov         eax,dword ptr [ebp-8]
002F1A1F  cmp         eax,dword ptr [N]
002F1A22  jge         __$EncStackInitStart+4Ah (02F1A36h)    ≤ 1ms elapsed
        gcd(a, b);
002F1A24  mov         eax,dword ptr [b]
002F1A27  push        eax
002F1A28  mov         ecx,dword ptr [a]
002F1A2B  push        ecx
002F1A2C  call        gcd (02F11EAh)
002F1A31  add         esp,8
    }
002F1A34  jmp         __$EncStackInitStart+27h (02F1A13h)
    double timeTaken = GetCounter() * 1000; // Convert to milliseconds
002F1A36  call        GetCounter (02F1271h)
002F1A3B  fstp        qword ptr [ebp-0E4h]
002F1A41  movsd       xmm0,mmword ptr [ebp-0E4h]
002F1A49  mulsd       xmm0,mmword ptr [__real@408f400000000000 (02F7B80h)]
002F1A51  movsd       mmword ptr [timeTaken],xmm0
    printf("Average time for %d iterations of GCD: %f ms\n", N, timeTaken / N);
002F1A56  cvtsi2sd    xmm0,dword ptr [N]
002F1A5B  movsd       xmm1,mmword ptr [timeTaken]
002F1A60  divsd       xmm1,xmm0
002F1A64  sub         esp,8
002F1A67  movsd       mmword ptr [esp],xmm1
002F1A6C  mov         eax,dword ptr [N]
002F1A6F  push        eax
002F1A70  push        offset string "Average time for %d iterations @"... (02F7B30h)
002F1A75  call        _printf (02F10D7h)
002F1A7A  add         esp,10h
}
002F1A7D  pop         edi
002F1A7E  pop         esi
002F1A7F  pop         ebx
002F1A80  add         esp,0E4h
002F1A86  cmp         ebp,esp
002F1A88  call        __RTC_CheckEsp (02F1262h)
002F1A8D  mov         esp,ebp
002F1A8F  pop         ebp
002F1A90  ret
```

The preliminary instructions (boxed in red) initializes a new stack frame with stack pointer value 0x008FF798 and base pointer value 0x008FF888.


Registers
EAX = 00000062 EBX = 01098000
ECX = 00000038 EDX = 00000001
ESI = 00F71028 EDI = 012FF700
EIP = 00F719EF ESP = 012FF634
EBP = 012FF724 EFL = 00000202
93 %


Memory 1
Address: 0x012FF634
0x012FF634 1c f8 2f 01


Memory 1
Address: 0x012FF724
0x012FF724 1c f8 2f 01

Main code

```
--- D:\Classes\CSC 210\TH Exam\gcd.cpp --------------------------------
#include <stdio.h>
#include <windows.h>

LARGE_INTEGER frequency;        // ticks per second
LARGE_INTEGER start, end;       // ticks

int gcd_recurs(int a, int b) {
00F71940  push        ebp   ≤ 1ms elapsed
00F71941  mov         ebp,esp
00F71943  sub         esp,0C0h
00F71949  push        ebx
00F7194A  push        esi
00F7194B  push        edi
00F7194C  mov         edi,ebp
00F7194E  xor         ecx,ecx
00F71950  mov         eax,0CCCCCCCCh
00F71955  rep stos    dword ptr es:[edi]
00F71957  mov         ecx,offset _4403464C_gcd@cpp (0F7C015h)
00F7195C  call        @__CheckForDebuggerJustMyCode@4 (0F71352h)
    if (b == 0)
00F71961  cmp         dword ptr [b],0
00F71965  jne         __$EncStackInitStart+22h (0F7196Eh)
        return a;
00F71967  mov         eax,dword ptr [a]
00F7196A  jmp         __$EncStackInitStart+36h (0F71982h)
00F7196C  jmp         __$EncStackInitStart+36h (0F71982h)
    else
        return gcd_recurs(b, a % b);
00F7196E  mov         eax,dword ptr [a]
00F71971  cdq
00F71972  idiv        eax,dword ptr [b]
00F71975  push        edx
00F71976  mov         eax,dword ptr [b]
00F71979  push        eax
00F7197A  call        gcd (0F713F7h)
00F7197F  add         esp,8
}
00F71982  pop         edi
00F71983  pop         esi
00F71984  pop         ebx
00F71985  add         esp,0C0h
00F7198B  cmp         ebp,esp
00F7198D  call        __RTC_CheckEsp (0F71262h)
00F71992  mov         esp,ebp
00F71994  pop         ebp
00F71995  ret
```

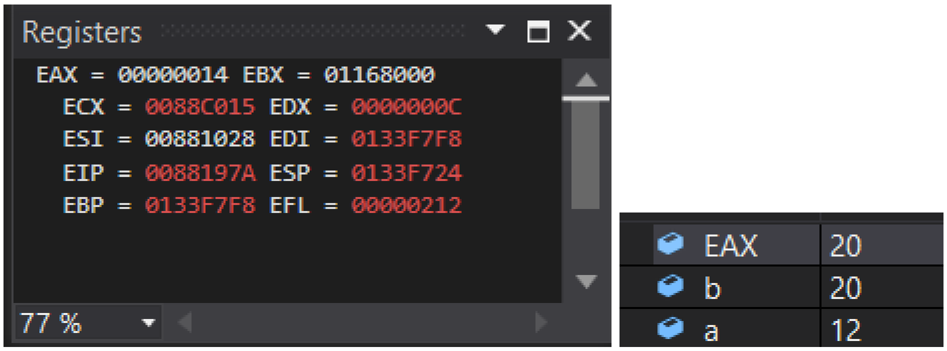
mov ebp, esp sets the base pointer (EBP) to the current stack pointer (ESP), creating a new stack frame.

sub esp, 0D8h allocates space on the stack for local variables.

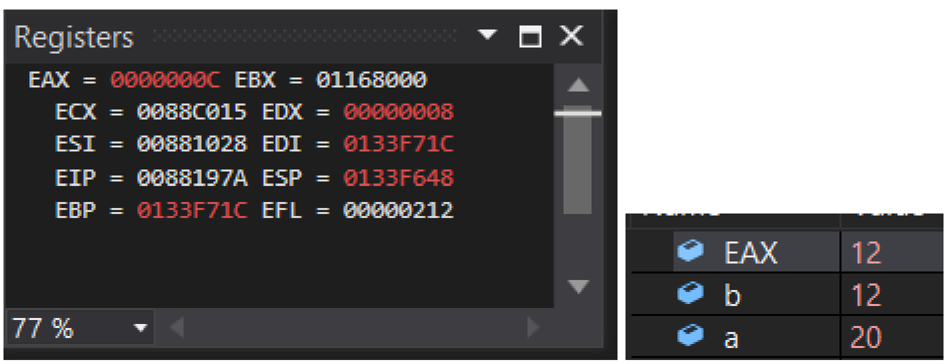
add esp, 0D8h deallocates the allocated stack space when the function exits.

This time for each called I also included a Watch to see the variables easier in decimal

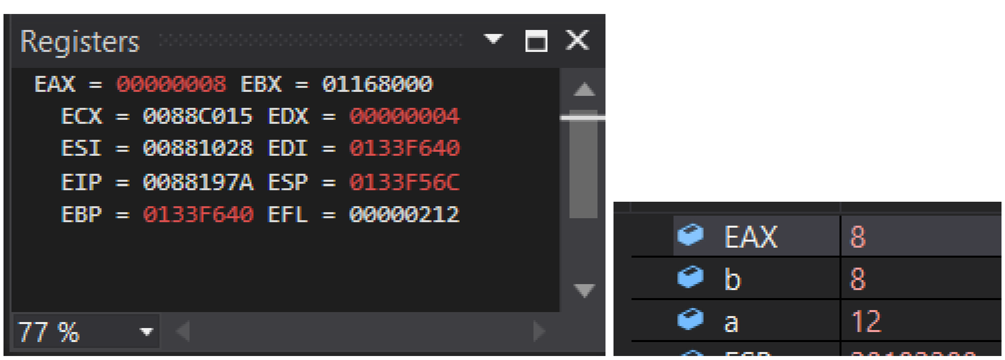
First called



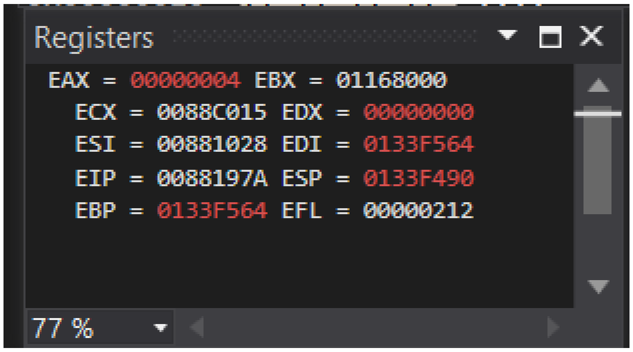
Second Called

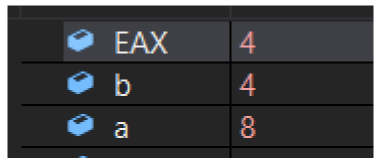


Third Called

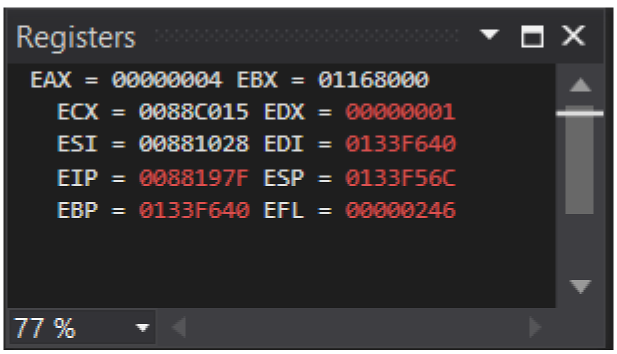


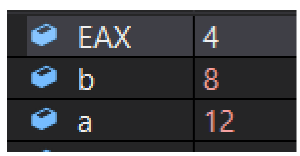
Forth Called



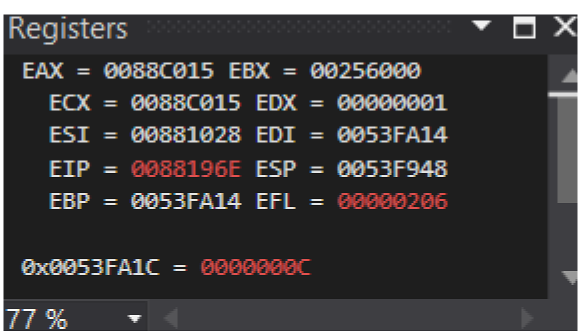


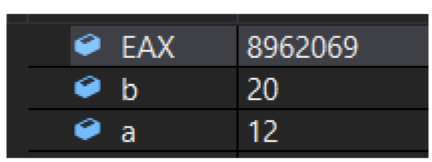
Fifth Called





Sixth Called





DEFINITIONS

How does the factorial function works?

This recursive function calculates the factorial of an integer `n`. It starts by checking if `n` is equal to 1, and if so, returns 1, which is the base case for the factorial of 1. If `n` is greater than 1, it recursively calls itself with the argument `(n - 1)` and multiplies the result by `n`. This process continues, decrementing `n` in each recursive call until `n` reaches 1, at which point the function returns the product of all the integers from `n` down to 1, effectively calculating the factorial of `n`.

How does the gcd works?

This recursive function calculates the greatest common divisor (GCD) of two integers `a` and `b` using the Euclidean algorithm. It starts by checking if `b` is equal to 0, and if so, returns `a`, which is the GCD. If `b` is not 0, it recursively calls itself with the arguments `(b, a % b)`, effectively reducing the problem to finding the GCD of `b` and the remainder of `a` divided by `b`. This process continues until `b` becomes 0, at which point the function returns the GCD of the original `a` and `b`.