

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
Dissassembly.....	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

```
factorial.cpp
% Project1
(Global Scope)

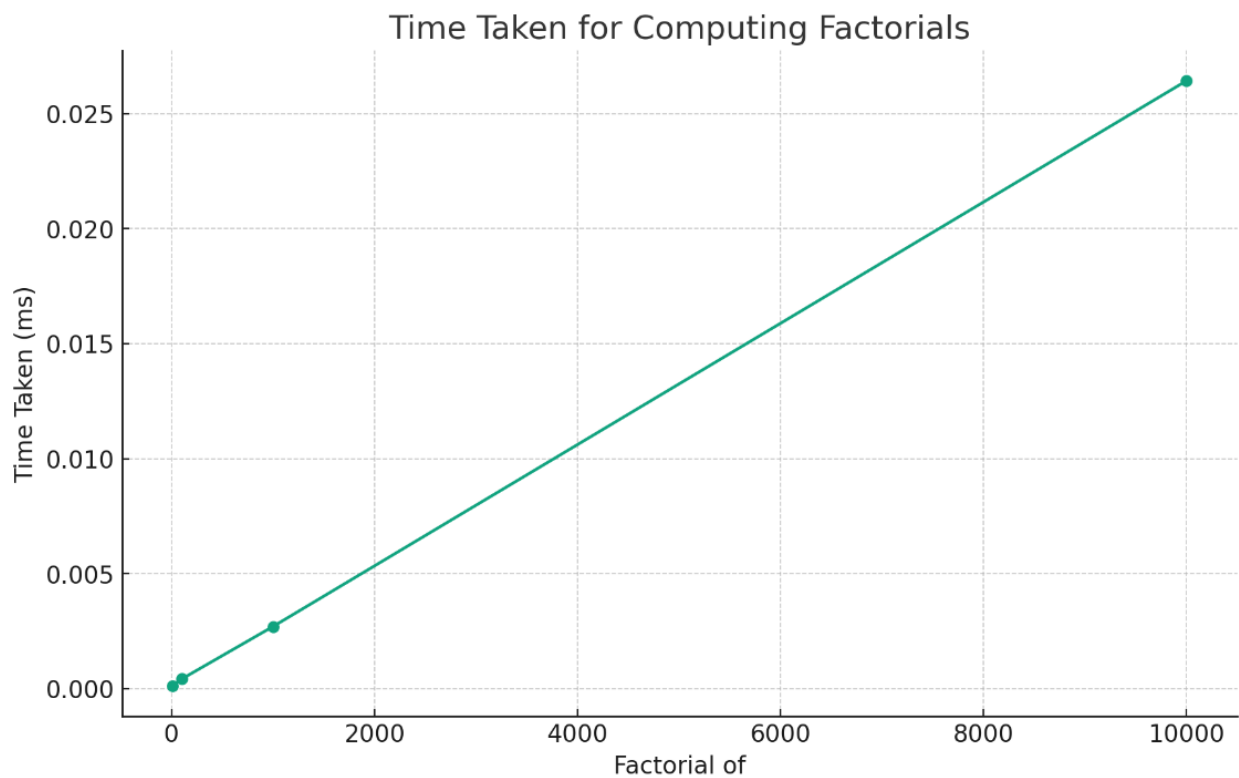
1  #include <iostream>
2  #include <windows.h>
3
4  using namespace std;
5
6  double PCFreq = 0.0;
7  LONGLONG CounterStart = 0;
8
9  void StartCounter() {
10     LARGE_INTEGER li;
11     if (!QueryPerformanceFrequency(&li))
12         cout << "QueryPerformanceFrequency failed!\n";
13
14     PCFreq = double(li.QuadPart) / 1000.0;
15     QueryPerformanceCounter(&li);
16     CounterStart = li.QuadPart;
17 }
18
19 double GetCounter() {
20     LARGE_INTEGER li;
21     QueryPerformanceCounter(&li);
22     return double(li.QuadPart - CounterStart) / PCFreq;
23 }
24
25 int factorial(int n) {
26     int result = 1;
27     for (int i = 1; i <= n; ++i) {
28         result *= i;
29     }
30     return result;
31 }
32
33 void measureFactorial(int N, int numMeasurements) {
34     double totalTime = 0.0;
35     for (int i = 0; i < numMeasurements; ++i) {
36         StartCounter();
37         factorial(N);
38         totalTime += GetCounter();
39     }
40     double averageTime = totalTime / numMeasurements;
41     cout << "Average time taken for factorial of " << N << ": " << averageTime << " ms\n";
42 }
43
44 int main() {
45     const int numMeasurements = 10000;
46     measureFactorial(10, numMeasurements);
47     measureFactorial(100, numMeasurements);
48     measureFactorial(1000, numMeasurements);
49     measureFactorial(10000, numMeasurements);
50
51     return 0;
52 }
53
```

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 console when debugging stops.
Press any key to close this window . . .
```

Graph



Disassembly

Main call

```
00202907 int 3
--- 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

```
----- D:\Classes\CS210\HW Exam\windows\Factorial.cpp -----
void measureFactorial(int N, int numMeasurements) {
00A127A8 push    ebp
00A127A1 mov     ebp,esp
00A127A3 sub     esp,0F4h
00A127A9 push    ebx
00A127AA push    esi
00A127AB push    edi
00A127AC lea     esi,[ebp-34h]
00A127AF mov     ecx,00h
00A127B4 mov     eax,0CCCCCCCCh
00A127B9 rep     stosl dword ptr esi,fid1
00A127BB mov     ecx,offset _FIBSCD07_factorial@8p (0A1F035h)
00A127C0 call    @_CheckForDebuggerJustMyCode@4 (0A11386h)
    double totalTime = 0.0;
00A127C5 xorps   xmm0,xmm0
00A127C8 movsd  mmword ptr [totalTime],xmm0
    for (int i = 0; i < numMeasurements; ++i) {
00A127CD mov     dword ptr [ebp-10h],0
    __$JncStackInitStart+33h (0A1270Fh)
00A127D6 mov     eax,dword ptr [ebp-10h]
00A127D9 add     eax,1
00A127DC mov     dword ptr [ebp-10h],eax
00A127DF mov     eax,dword ptr [ebp-10h]
00A127E2 cmp     eax,dword ptr [numMeasurements]
00A127E5 jge     __$JncStackInitStart+60h (0A12817h)
    StartCounter();
00A127E7 call    StartCounter (0A1100Fh)
    factorial(N);
00A127EC mov     eax,dword ptr [N]
00A127EF push    eax
00A127F0 call    factorial (0A112FDh)
    totalTime += GetCounter();
00A127F8 call    GetCounter (0A112BCh)
00A127FD fstp    qword ptr [ebp-0F4h]
00A12803 movsd  xmm0,mmword ptr [ebp-0F4h]
00A12808 addsd  xmm0,mmword ptr [totalTime]
00A12810 movsd  mmword ptr [totalTime],xmm0
    }
00A12815 jmp     __$JncStackInitStart+24h (0A12706h)
    double averageTime = totalTime / numMeasurements;
00A12817 cvtsi2sd xmm0,dword ptr [numMeasurements]
00A1281C movsd  xmm1,mmword ptr [totalTime]
00A12821 divsd  xmm1,xmm0
00A12825 movsd  mmword ptr [averageTime],xmm1

    cout << "Average time taken for factorial of " << N << ": " << averageTime << " ms\n";
00A1282A push    offset string " ms\n" (0A19B60h)
00A1282F mov     esi,esp
00A12831 sub     esp,8
00A12834 movsd  xmm0,mmword ptr [averageTime]
00A12839 movsd  mmword ptr [esp],xmm0
00A1283E push    offset string ": " (0A19B30h)
00A12843 mov     edi,esp
00A12845 mov     eax,dword ptr [N]
00A12848 push    eax
00A12849 push    offset string "Average time taken for factoria@"... (0A19B60h)
00A1284E mov     ecx,dword ptr [__imp_std::cout (0A1D09Ch)]
00A12854 push    ecx
00A12855 call    std::operator<<<std::char_traits<char> > (0A111C2h)
00A1285A add     esp,8
00A1285D mov     ecx,eax
00A1285F call    dword ptr [__imp_std::basic_ostream<char,std::char_traits<char> >::operator<< (0A1D0A8h)]
00A12865 cmp     edi,esp
00A12867 call    __RTC_CheckEsp (0A112ADh)
00A1286C push    eax
00A1286D call    std::operator<<<std::char_traits<char> > (0A111C2h)
00A12872 add     esp,8
00A12875 mov     ecx,eax
00A12877 call    dword ptr [__imp_std::basic_ostream<char,std::char_traits<char> >::operator<< (0A1D0A4h)]
00A1287D cmp     edi,esp
00A1287F call    __RTC_CheckEsp (0A112ADh)
00A12884 push    eax
00A12885 call    std::operator<<<std::char_traits<char> > (0A111C2h)
00A1288A add     esp,8
    }
00A1288D pop     edi
00A1288E pop     esi
00A1288F pop     ebx
00A12890 add     esp,0F4h
00A12896 cmp     ebp,esp
00A12898 call    __RTC_CheckEsp (0A112ADh)
00A1289D mov     esp,ebp
00A1289F pop     ebp
00A128A0 ret
}
```

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

Registers	
EAX = 2E7771D4	EBX = 00F8A000
ECX = E029C2CD	EDX = 000003BE
ESI = 00A11028	EDI = 0113FBCC
EIP = 00A127EC	ESP = 0113FACC
EBP = 0113FBCC	EFL = 00000246

Memory 1	
Address: 0x0113FACC	
0x0113FACC	b4 fc 13 01

Memory 1	
Address: 0x0113FBCC	
0x0113FBCC	b4 fc 13 01

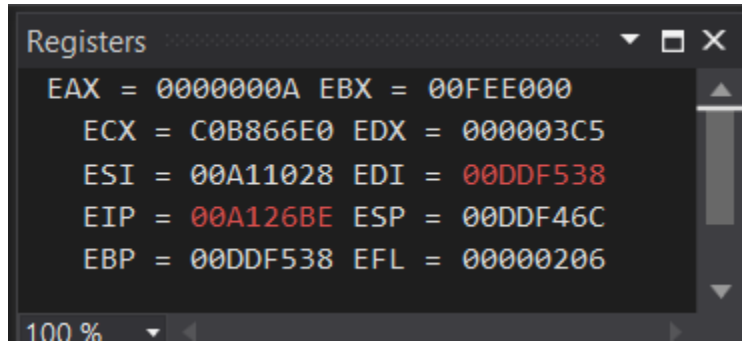
First Call

```
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.

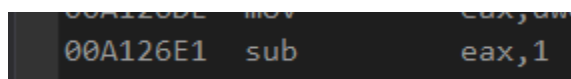


A screenshot of the 'Registers' window in a debugger. It displays the following values: EAX = 0000000A, EBX = 00FEE000, ECX = C0B866E0, EDX = 000003C5, ESI = 00A11028, EDI = 00DDF538, EIP = 00A126BE, ESP = 00DDF46C, EBP = 00DDF538, and EFL = 00000206. The values for EDI, EIP, and ESP are highlighted in red.

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.

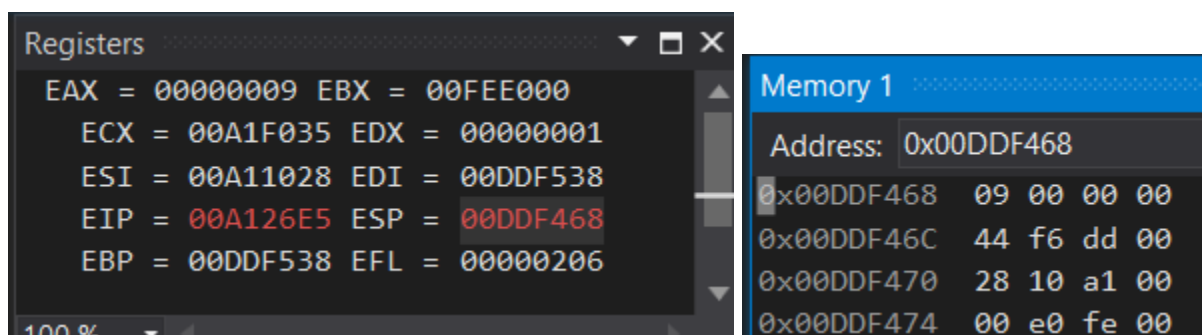
The next instructions (in green) will check if the argument value (N, in this case 10) is equal to 1 and perform a jump if they are not equal. Since they are not equal, a jump is performed. The next sequence of instructions (in red) are executed. First, the argument n is copied into register EAX, then 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.



A screenshot of the instruction window showing two assembly instructions. The first instruction is 'mov eax, [00A126E0]' and the second is 'sub eax, 1'. The second instruction is highlighted in red.

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 .



Two screenshots from a debugger. The left screenshot shows the 'Registers' window with the following values: EAX = 00000009, EBX = 00FEE000, ECX = 00A1F035, EDX = 00000001, ESI = 00A11028, EDI = 00DDF538, EIP = 00A126E5, ESP = 00DDF468, EBP = 00DDF538, and EFL = 00000206. The values for EDI, EIP, and ESP are highlighted in red. The right screenshot shows the 'Memory 1' window with the address 0x00DDF468. The memory contents are: 0x00DDF468: 09 00 00 00, 0x00DDF46C: 44 f6 dd 00, 0x00DDF470: 28 10 a1 00, and 0x00DDF474: 00 e0 fe 00.

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

Then, 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

Registers

EAX = 00000009	EBX = 00FEE000
ECX = 00A1F035	EDX = 00000001
ESI = 00A11028	EDI = 00DDF460
EIP = 00A126E1	ESP = 00DDF394
EBP = 00DDF460	EFL = 00000202

100 %

Registers

EAX = 00000008	EBX = 00FEE000
ECX = 00A1F035	EDX = 00000001
ESI = 00A11028	EDI = 00DDF460
EIP = 00A126E4	ESP = 00DDF394
EBP = 00DDF460	EFL = 00000202

100 %

Memory 1

Address: 0x00DDF390

0x00DDF390 08 00 00 00

Third call

Registers

EAX = 00000007	EBX = 00FEE000
ECX = 00A1F035	EDX = 00000001
ESI = 00A11028	EDI = 00DDF388
EIP = 00A126E4	ESP = 00DDF2BC
EBP = 00DDF388	EFL = 00000202

100 %

Memory 1

Address: 0x00DDF2B8

0x00DDF2B8 07 00 00 00

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

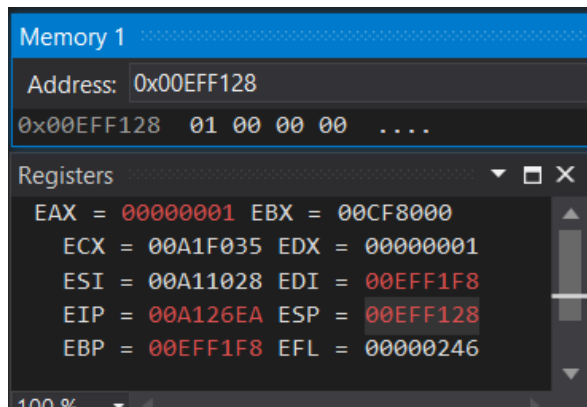
Returning Calls

```
if (n == 1) return 1;
00A126D1 cmp      dword ptr [n],1    ≤1ms elapsed
00A126D5 jne      __$EncStackInitStart+22h (0A126DEh)
00A126D7 mov      eax,1
00A126DC jmp      __$EncStackInitStart+35h (0A126F1h)
```

After it reaches `n == 1`.

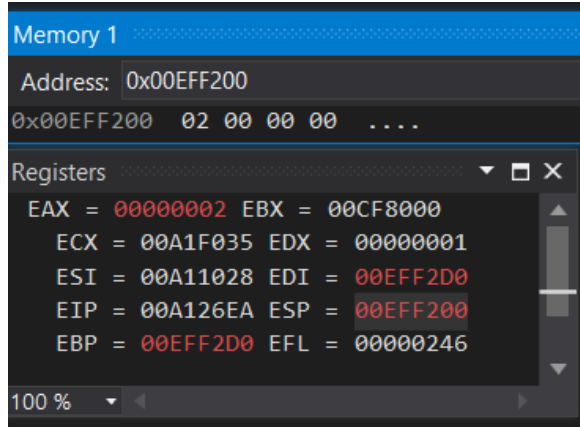
```
return (n * factorial(n - 1));
00A126DE mov      eax,dword ptr [n]
00A126E1 sub      eax,1
00A126E4 push     eax
00A126E5 call     factorial (0A112FDh)
00A126EA add      esp,4    ≤1ms elapsed
00A126ED imul     eax,dword ptr [n]
}
```

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

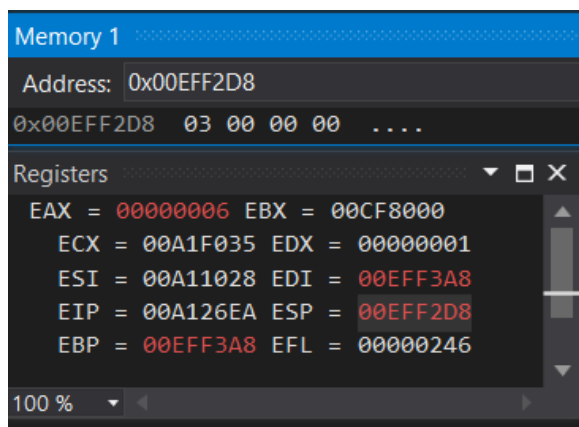


EAX factorial of 2

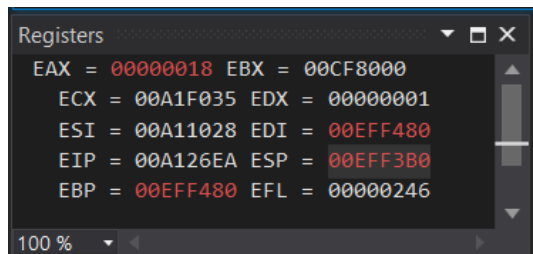
EAX factorial of 3



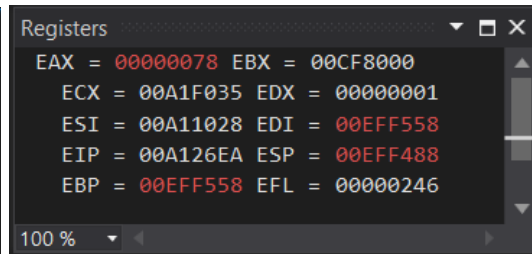
EAX factorial of 4



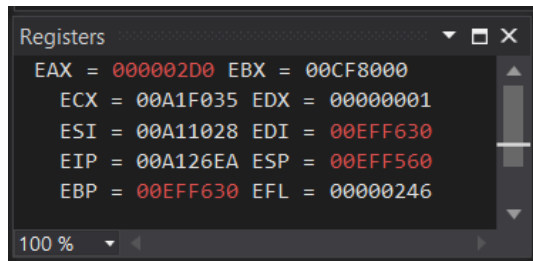
EAX factorial of 5



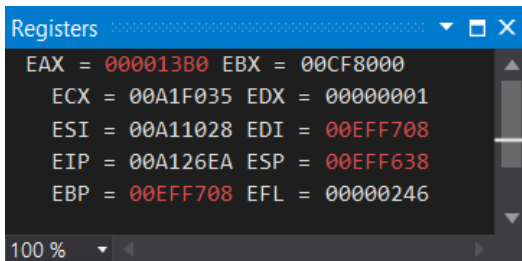
EAX factorial of 6



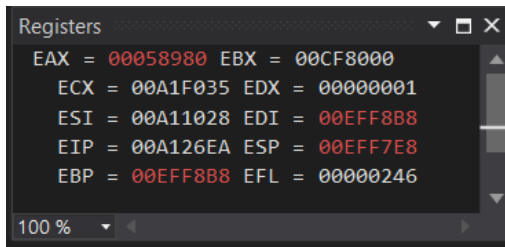
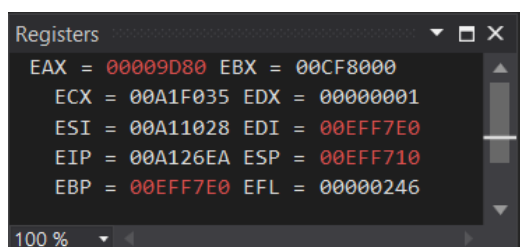
EAX factorial of 7



EAX factorial of 8



EAX factorial of 9



GCD

C code

```
gcd.cpp  ▸ ×
Project1  (Global Scope)
1  #include <stdio.h>
2  #include <windows.h>
3
4  LARGE_INTEGER frequency;    // ticks per second
5  LARGE_INTEGER start, end;    // ticks
6
7  int gcd_recurs(int a, int b) {
8      if (b == 0)
9          return a;
10     else
11         return gcd_recurs(b, a % b);
12 }
13
14 void StartCounter() {
15     QueryPerformanceFrequency(&frequency);
16     QueryPerformanceCounter(&start);
17 }
18
19 double GetCounter() {
20     QueryPerformanceCounter(&end);
21     return (double)(end.QuadPart - start.QuadPart) / frequency.QuadPart;
22 }
23
24 void measureGCD(int a, int b, int N) {
25     StartCounter();
26     for (int i = 0; i < N; i++) {
27         gcd_recurs(a, b);
28     }
29     double timeTaken = GetCounter() * 1000; // Convert to milliseconds
30     printf("Average time for %d iterations: %f ms\n", N, timeTaken / N);
31 }
32
33 int main() {
34     int a = 56; // Example values
35     int b = 98;
36
37     measureGCD(a, b, 10);
38     measureGCD(a, b, 100);
39     measureGCD(a, b, 1000);
40     measureGCD(a, b, 10000);
41
42     return 0;
}
```

Results

```
Microsoft Visual Studio Debug Console

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 console when debugging stops.
Press any key to close this window . . .
```

Dissassembly

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 esi[edi]
    002F19FB mov     ecx,offset _4403464C_gcd@cpp (02F015h)
    002F1A00 call    @_CheckForDebuggerJustMyCode@4 (02F1352h)
    StartCounter();
    002F1A05 call    StartCounter (02F108Fh)
    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,mword ptr [ebp-0E4h]
    002F1A49 mulsd   xmm0,mword ptr [__real@408f400000000000 (02F7880h)]
    002F1A51 movsd   mword ptr [timeTaken],xmm0

    printf("Average time for %d iterations of GCD: %f ms\n", N, timeTaken / N);
    002F1A56 cvtsi2sd  xmm0,dword ptr [N]
    002F1A58 movsd   xmm1,mword ptr [timeTaken]
    002F1A60 divsd   xmm1,xmm0
    002F1A64 sub     esp,8
    002F1A67 movsd   mword ptr [esp],xmm1
    002F1A6C mov     eax,dword ptr [N]
    002F1A6F push    eax
    002F1A70 push    offset string "Average time for %d iterations @"... (02F7830h)
    002F1A75 call    _printf (02F1007h)
    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,0CCCCCCC  
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

Registers

EAX = 00000014 EBX = 01168000
ECX = 0088C015 EDX = 0000000C
ESI = 00881028 EDI = 0133F7F8
EIP = 0088197A ESP = 0133F724
EBP = 0133F7F8 EFL = 00000212

77 %

EAX	20
b	20
a	12

Second Called

Registers

EAX = 0000000C EBX = 01168000
ECX = 0088C015 EDX = 00000008
ESI = 00881028 EDI = 0133F71C
EIP = 0088197A ESP = 0133F648
EBP = 0133F71C EFL = 00000212

77 %

EAX	12
b	12
a	20

Third Called

Registers

EAX = 00000008 EBX = 01168000
ECX = 0088C015 EDX = 00000004
ESI = 00881028 EDI = 0133F640
EIP = 0088197A ESP = 0133F56C
EBP = 0133F640 EFL = 00000212

77 %

EAX	8
b	8
a	12

Forth Called

Registers	
EAX = 00000004	EBX = 01168000
ECX = 0088C015	EDX = 00000000
ESI = 00881028	EDI = 0133F564
EIP = 0088197A	ESP = 0133F490
EBP = 0133F564	EFL = 00000212

EAX	4
b	4
a	8

Fifth Called

Registers	
EAX = 00000004	EBX = 01168000
ECX = 0088C015	EDX = 00000001
ESI = 00881028	EDI = 0133F640
EIP = 0088197F	ESP = 0133F56C
EBP = 0133F640	EFL = 00000246

EAX	4
b	8
a	12

Sixth Called

Registers	
EAX = 0088C015	EBX = 00256000
ECX = 0088C015	EDX = 00000001
ESI = 00881028	EDI = 0053FA14
EIP = 0088196E	ESP = 0053F948
EBP = 0053FA14	EFL = 00000206

EAX	8962069
b	20
a	12

0x0053FA1C = 0000000C

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`.