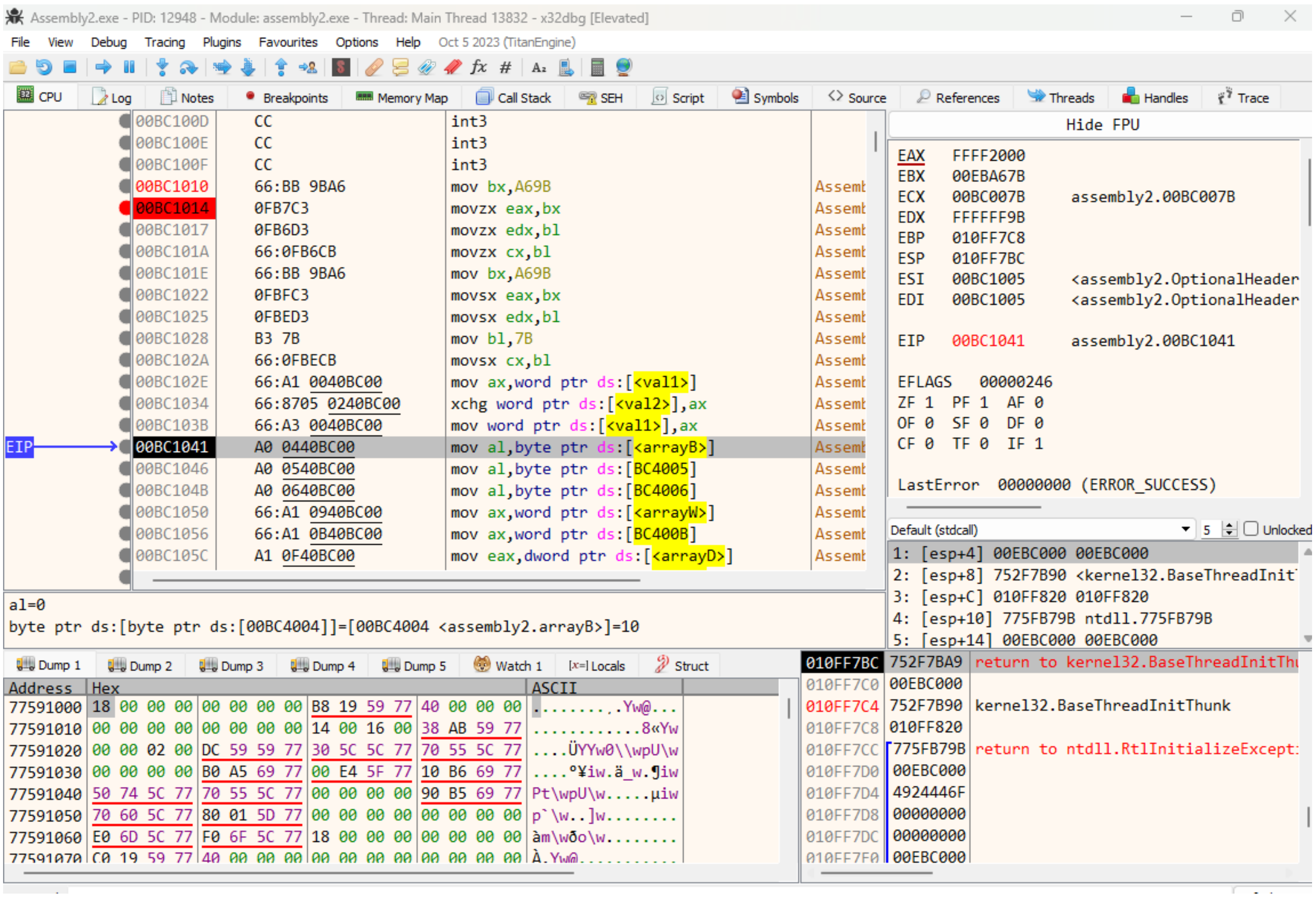
Debugging 1

Compiled the program and started debugging it in x64dbg instead of visual studio community, coz it doesn't have the registers option:


Assembly2.exe - PID: 12948 - Module: assembly2.exe - Thread: Main Thread 13832 - x32dbg [Elevated]
File View Debug Tracing Plugins Favourites Options Help Oct 5 2023 (TitanEngine)
CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads Handles Trace
Hide FPU
Dump 1 Dump 2 Dump 3 Dump 4 Dump 5 Watch 1 Locals Struct

```
     00BC100D   CC              int3
     00BC100E   CC              int3
     00BC100F   CC              int3
     00BC1010   66:BB 9BA6      mov bx,A69B                        Assemt
     00BC1014   0FB7C3          movzx eax,bx                       Assemt
     00BC1017   0FB6D3          movzx edx,bl                       Assemt
     00BC101A   66:0FB6CB       movzx cx,bl                        Assemt
     00BC101E   66:BB 9BA6      mov bx,A69B                        Assemt
     00BC1022   0FBFC3          movsx eax,bx                       Assemt
     00BC1025   0FBED3          movsx edx,bl                       Assemt
     00BC1028   B3 7B           mov bl,7B                          Assemt
     00BC102A   66:0FBECB       movsx cx,bl                        Assemt
     00BC102E   66:A1 0040BC00  mov ax,word ptr ds:[<val1>]        Assemt
     00BC1034   66:8705 0240BC00 xchg word ptr ds:[<val2>],ax      Assemt
     00BC103B   66:A3 0040BC00  mov word ptr ds:[<val1>],ax        Assemt
EIP  00BC1041   A0 0440BC00     mov al,byte ptr ds:[<arrayB>]      Assemt
     00BC1046   A0 0540BC00     mov al,byte ptr ds:[BC4005]        Assemt
     00BC104B   A0 0640BC00     mov al,byte ptr ds:[BC4006]        Assemt
     00BC1050   66:A1 0940BC00  mov ax,word ptr ds:[<arrayW>]      Assemt
     00BC1056   66:A1 0B40BC00  mov ax,word ptr ds:[BC400B]        Assemt
     00BC105C   A1 0F40BC00     mov eax,dword ptr ds:[<arrayD>]    Assemt
```

```
EAX  FFFF2000
EBX  00EBA67B
ECX  00BC007B     assembly2.00BC007B
EDX  FFFFFF9B
EBP  010FF7C8
ESP  010FF7BC
ESI  00BC1005     <assembly2.OptionalHeader
EDI  00BC1005     <assembly2.OptionalHeader

EIP  00BC1041     assembly2.00BC1041

EFLAGS   00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCESS)
```

```
Default (stdcall)   5   Unlocked
1: [esp+4] 00EBC000 00EBC000
2: [esp+8] 752F7B90 <kernel32.BaseThreadInit
3: [esp+C] 010FF820 010FF820
4: [esp+10] 775FB79B ntdll.775FB79B
5: [esp+14] 00EBC000 00EBC000
```

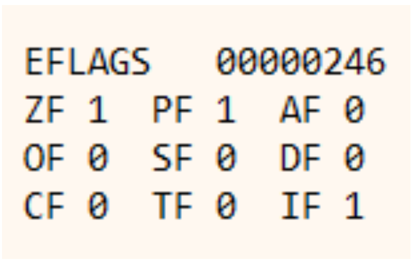
```
al=0
byte ptr ds:[byte ptr ds:[00BC4004]]=[00BC4004 <assembly2.arrayB>]=10
```

```
Address   Hex                                              ASCII
77591000  18 00 00 00 00 00 00 00 B8 19 59 77 40 00 00 00  .........Yw@...
77591010  00 00 00 00 00 00 00 00 14 00 16 00 38 AB 59 77  ............8«Yw
77591020  00 00 02 00 DC 59 59 77 30 5C 5C 77 70 55 5C 77  ....ÜYYw0\\wpU\w
77591030  00 00 00 00 B0 A5 69 77 00 E4 5F 77 10 B6 69 77  ....°¥iw.ä_w.¶iw
77591040  50 74 5C 77 70 55 5C 77 00 00 00 00 90 B5 69 77  Pt\wpU\w.....µiw
77591050  70 60 5C 77 80 01 5D 77 00 00 00 00 00 00 00 00  p`\w..]w........
77591060  E0 6D 5C 77 F0 6F 5C 77 18 00 00 00 00 00 00 00  àm\wðo\w........
77591070  C0 19 59 77 40 00 00 00 00 00 00 00 00 00 00 00  À.Yw@...........
```

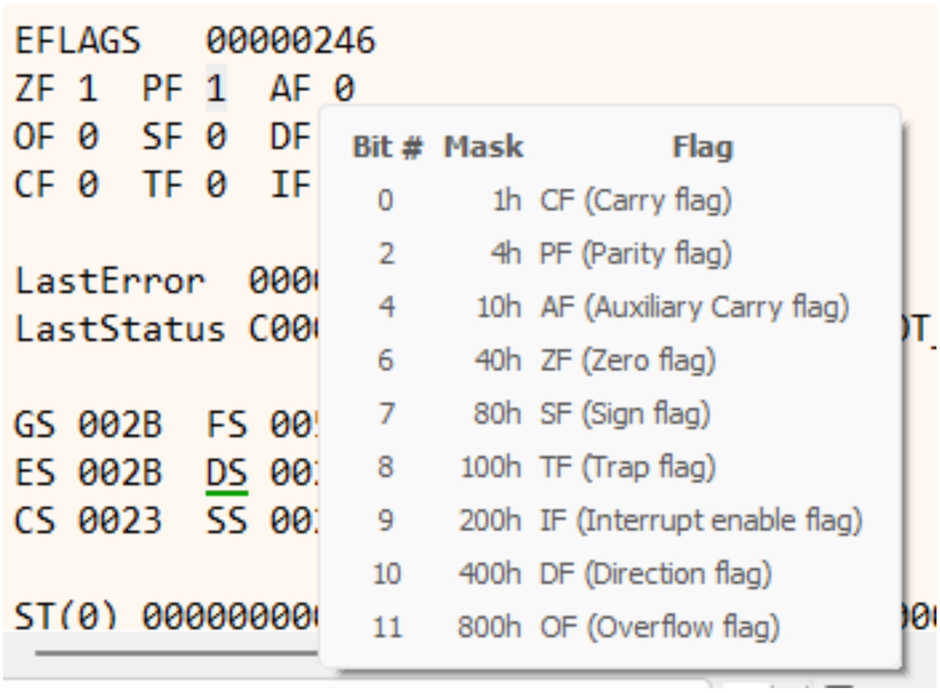
```
010FF7BC  752F7BA9  return to kernel32.BaseThreadInitThu
010FF7C0  00EBC000
010FF7C4  752F7B90  kernel32.BaseThreadInitThunk
010FF7C8  010FF820
010FF7CC  775FB79B  return to ntdll.RtlInitializeExcept:
010FF7D0  00EBC000
010FF7D4  4924446F
010FF7D8  00000000
010FF7DC  00000000
010FF7E0  00EBC000
```

You can see the registers, flag registers, memory dumps, stack etc.

Each flag is assigned a value of 0 (clear) or 1 (set). Here’s an example:



```
EFLAGS   00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1
```



```
EFLAGS   00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF
CF 0  TF 0  IF

LastError  000
LastStatus C00                    T_

GS 002B  FS 00
ES 002B  DS 00
CS 0023  SS 00

ST(0) 0000000                      0
```

Bit #	Mask	Flag
0	1h	CF (Carry flag)
2	4h	PF (Parity flag)
4	10h	AF (Auxiliary Carry flag)
6	40h	ZF (Zero flag)
7	80h	SF (Sign flag)
8	100h	TF (Trap flag)
9	200h	IF (Interrupt enable flag)
10	400h	DF (Direction flag)
11	800h	OF (Overflow flag)

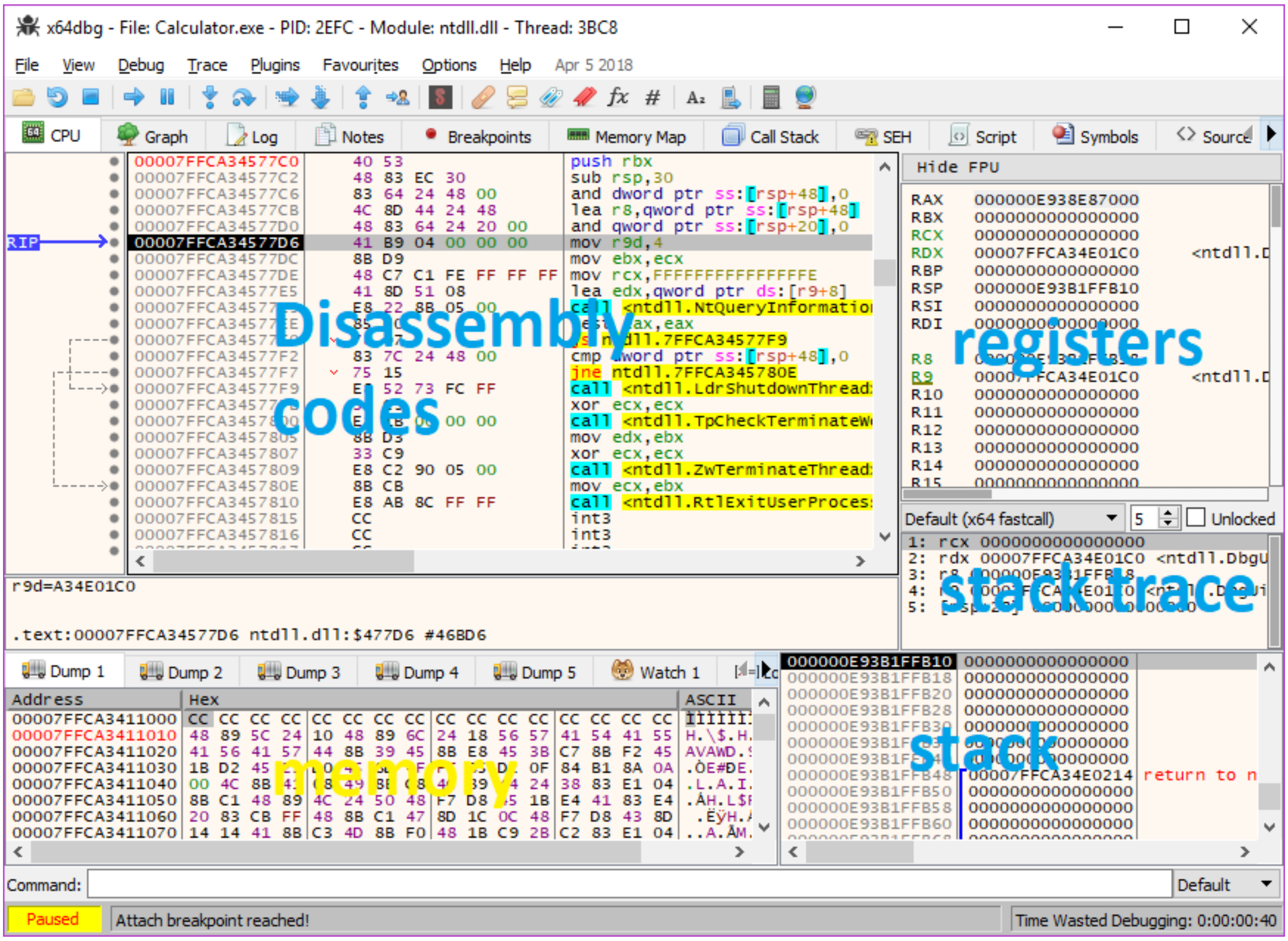
Yes, it is normal for the disassembled EXE to have more lines of code than what you wrote. This is because the compiler adds additional code to the EXE file to implement library functions, operating system calls, and other features.

For example, in the disassembly you provided, I can see that the compiler has added code to initialize the stack, call the main() function, and exit the program. The compiler has also added code to handle various errors, such as division by zero and invalid memory access.

In addition to the code that is added by the compiler, the EXE file may also contain code from libraries that are linked to the program. For example, if your program uses the C standard library, the EXE file will contain code from the standard library.

Finally, the EXE file may also contain code that is injected by other programs, such as malware or debugging tools.

If you are interested in learning more about the code in the disassembled EXE file, you can use a debugger to step through the code and inspect the registers and memory. You can also use a disassembler to generate a more detailed listing of the assembly code.


x64dbg - File: Calculator.exe - PID: 2EFC - Module: ntdll.dll - Thread: 3BC8
File View Debug Trace Plugins Favourites Options Help Apr 5 2018
CPU Graph Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source
Disassembly codes
registers
stack trace
memory
stack
Dump 1 Dump 2 Dump 3 Dump 4 Dump 5 Watch 1
Hide FPU
RIP

```
00007FFCA34577C0   40 53                 push rbx
00007FFCA34577C2   48 83 EC 30           sub rsp,30
00007FFCA34577C6   83 64 24 48 00        and dword ptr ss:[rsp+48],0
00007FFCA34577CB   4C 8D 44 24 48        lea r8,qword ptr ss:[rsp+48]
00007FFCA34577D0   48 83 64 24 20 00     and qword ptr ss:[rsp+20],0
00007FFCA34577D6   41 B9 04 00 00 00     mov r9d,4
00007FFCA34577DC   8B D9                 mov ebx,ecx
00007FFCA34577DE   48 C7 C1 FE FF FF FF  mov rcx,FFFFFFFFFFFFFFFE
00007FFCA34577E5   41 8D 51 08           lea edx,qword ptr ds:[r9+8]
00007FFCA3457[illegible]   E8 22 8B 05 00        call <ntdll.NtQueryInformatio
00007FFCA34577[illegible]E   85 [illegible]                 test eax,eax
00007FFCA34577[illegible]   [illegible]                   js ntdll.7FFCA34577F9
00007FFCA34577F2   83 7C 24 48 00        cmp dword ptr ss:[rsp+48],0
00007FFCA34577F7   75 15                 jne ntdll.7FFCA345780E
00007FFCA34577F9   E[illegible] 52 73 FC FF        call <ntdll.LdrShutdownThread
00007FFCA34577[illegible]   [illegible]                   xor ecx,ecx
00007FFCA34578[illegible]00   E[illegible] [illegible]B 0[illegible] 00 00        call <ntdll.TpCheckTerminateW
00007FFCA345780[illegible]   8B D3                 mov edx,ebx
00007FFCA3457807   33 C9                 xor ecx,ecx
00007FFCA3457809   E8 C2 90 05 00        call <ntdll.ZwTerminateThread
00007FFCA345780E   8B CB                 mov ecx,ebx
00007FFCA3457810   E8 AB 8C FF FF        call <ntdll.RtlExitUserProces
00007FFCA3457815   CC                    int3
00007FFCA3457816   CC                    int3

r9d=A34E01C0

.text:00007FFCA34577D6 ntdll.dll:$477D6 #46BD6
```

```
RAX   000000E938E87000
RBX   0000000000000000
RCX   0000000000000000
RDX   00007FFCA34E01C0      <ntdll.D
RBP   0000000000000000
RSP   000000E93B1FFB10
RSI   0000000000000000
RDI   0000000000000000

R8    [illegible]
R9    0000[illegible]FFCA34E01C0      <ntdll.D
R10   0000000000000000
R11   0000000000000000
R12   0000000000000000
R13   0000000000000000
R14   0000000000000000
R15   0000000000000000

Default (x64 fastcall)   5   Unlocked
1: rcx 0000000000000000
2: rdx 00007FFCA34E01C0 <ntdll.DbgU
3: r8 [illegible]00000E93[illegible]1FFB[illegible]8
4: [illegible] [illegible]000[illegible]FFCA[illegible]E01[illegible]0 <nt[illegible]l.D[illegible]i
5: [rsp[illegible]2[illegible]] [illegible]
```

```
Address           Hex                                               ASCII
00007FFCA3411000  CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC  ÌÌÌÌÌÌÌ
00007FFCA3411010  48 89 5C 24 10 48 89 6C 24 18 56 57 41 54 41 55  H.\$.H.
00007FFCA3411020  41 56 41 57 44 8B 39 45 8B E8 45 3B C7 8B F2 45  AVAWD.9
00007FFCA3411030  1B D2 45 [illegible] 0F 84 B1 8A 0A  .ÒE#ÐE.
00007FFCA3411040  00 4C 8B 4[illegible] [illegible]8 49 8[illegible] [illegible]8 4[illegible] [illegible]9 [illegible]4 24 38 83 E1 04  .L.A.I.
00007FFCA3411050  8B C1 48 89 4C 24 50 48 F7 D8 [illegible]5 1B E4 41 83 E4  .ÁH.L$P
00007FFCA3411060  20 83 CB FF 48 8B C1 47 8D 1C 0C 48 F7 D8 43 8D   .ËÿH.
00007FFCA3411070  14 14 41 8B C3 4D 8B F0 48 1B C9 2B C2 83 E1 04  ..A.ÃM.
```

```
000000E93B1FFB10  0000000000000000
000000E93B1FFB18  0000000000000000
000000E93B1FFB20  0000000000000000
000000E93B1FFB28  0000000000000000
000000E93B1FFB30  0000000000000000
000000E93B1F[illegible]  [illegible]0000000
000000E93B1F[illegible]  [illegible]0000000
000000E93B1FFB48  00007FFCA34E0214  return to n
000000E93B1FFB50  0000000000000000
000000E93B1FFB58  0000000000000000
000000E93B1FFB60  0000000000000000

Command:                                        Default
Paused   Attach breakpoint reached!             Time Wasted Debugging: 0:00:00:40
```

Here is a more detailed description of each window:

Call Stack: The Call Stack window shows a list of all the functions that were called to reach the current point in the program. The functions are listed in reverse order, with the most recent function call at the top of the list. To view the Call Stack window, select View > Call Stack.

Memory Map: The Memory Map window shows the layout of memory in the program, including the addresses and sizes of all loaded modules, data segments, and stacks. To view the Memory Map window, select View > Memory Map.

Command Window is a text-based window that allows you to enter and execute debugger commands. You can use the Command Window to control the debugger session, display and analyze program data, and perform various debugger actions. For example, you can use the Command Window to:

- Start and stop the debugger session.

- Set and remove breakpoints.
- Step through the program line by line.
- Inspect and modify program data.

Registers Window shows the current values of the CPU registers. The CPU registers are special memory locations that store information that is frequently used by the CPU, such as the current instruction being executed, the address of the next instruction to be executed, and the results of recent calculations. By inspecting the values of the CPU registers, you can gain insights into what the program is doing and where it is in the execution process.

Disassembly Window shows the disassembly of the machine code instructions that are being executed. Machine code is the low-level language that is actually executed by the CPU. By inspecting the disassembly, you can see exactly what instructions the CPU is executing and how they are affecting the state of the program.

Variables Window shows the values of the variables in the program. Variables are named memory locations that store data. By inspecting the values of the variables, you can see what data is being used by the program and how it is changing over time.

Threads Window shows a list of all the threads in the program. A thread is a single path of execution through a program. The Threads Window allows you to select a thread to inspect and debug. You can also use the Threads Window to start and stop threads, and to suspend and resume threads.

Breakpoints Window shows a list of all the breakpoints that have been set in the program. A breakpoint is a marker in the program that tells the debugger to stop execution at that point. Breakpoints are useful for debugging because they allow you to stop the program at specific points and inspect its state.

----

1. What are the three basic types of operands? The three basic types of operands in assembly language are:

Register operands (e.g., EAX, ECX)
Memory operands (e.g., [var1])
Immediate operands (e.g., 42)

2. (True/False): The destination operand of a MOV instruction cannot be a segment register. True. In a MOV instruction, the destination operand cannot be a segment register. Segment registers are used for memory addressing and cannot be directly overwritten using a MOV instruction.

3. (True/False): In a MOV instruction, the second operand is known as the destination operand. False. In a MOV instruction, the first operand is known as the destination operand, and the second operand is the source operand. The data is moved from the source operand to the destination operand.

4. (True/False): The EIP register cannot be the destination operand of a MOV instruction. True. The EIP (Instruction Pointer) register is used to store the address of the next instruction to be executed. It cannot be directly modified or overwritten using a MOV instruction.

5. In the operand notation used by Intel, what does reg/mem32 indicate? The operand notation **"reg/mem32"** indicates that the operand can be either a 32-bit general-purpose register (reg) or a 32-bit memory operand (mem), which can be a memory location or variable.

6. In the operand notation used by Intel, what does imm16 indicate? The operand notation **"imm16"** indicates an immediate value of 16 bits. It represents a constant value that is part of the instruction itself and is not stored in a register or memory location.