Introduction to Intel x86 Assembly, Architecture, Applications, & Alliteration

Xeno Kovah – 2009/2010
xkovah at gmail

Approved for Public Release: 10-3348. Distribution Unlimited
All materials is licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/
Additional Content/Ideas/Info Provided By:

- Jon A. Erickson, Christian Arllen, Dave Keppler
- Who suggested what, is inline with the material
About Me

• Security nerd - generalist, not specialist
• Realmz ~1996, Mac OS 8, BEQ->BNE FTW!
• x86 ~2002
• Know or have known ~5 assembly languages(x86, SPARC, ARM, PPC, 68HC12). x86 is by far the most complex.
• Routinely read assembly when debugging my own code, reading exploit code, and reverse engineering things
About You?

• Name & Department
• Why did you want to take the class?
• If you know you will be applying this knowledge, to which OS and/or development environment?
About the Class

• The intent of this class is to expose you to the most commonly generated assembly instructions, and the most frequently dealt with architecture hardware.
  – 32 bit instructions/hardware
  – Implementation of a Stack
  – Common tools

• Many things will therefore be left out or deferred to later classes.
  – Floating point instructions/hardware
  – 16/64 bit instructions/hardware
  – Complicated or rare 32 bit instructions
  – Instruction pipeline, caching hierarchy, alternate modes of operation, hw virtualization, etc
About the Class 2

• The hope is that the material covered will be provide the required background to delve deeper into areas which may have seemed daunting previously.

• Because I can't anticipate the needs of all job classes, if there are specific areas which you think would be useful to certain job types, let me know. The focus areas are currently primarily influenced by my security background, but I would like to make the class as widely applicable as possible.
Agenda

• Day 1 - Part 1 - Architecture Introduction, Windows tools
• Day 1 - Part 2 - Windows Tools & Analysis, Learning New Instructions
• Day 2 - Part 1 - Linux Tools & Analysis
• Day 2 - Part 2 - Inline Assembly, Read The Fun Manual, Choose Your Own Adventure
Miss Alaineous

• Questions: Ask 'em if you got 'em
  – If you fall behind and get lost and try to tough it out until you understand, it's more likely that you will stay lost, so ask questions ASAP.

• Browsing the web and/or checking email during class is a good way to get lost ;)

• Vote on class schedule.

• Benevolent dictator clause.

• It's called x86 because of the progression of Intel chips from 8086, 80186, 80286, etc. I just had to get that out of the way. :)}
What you're going to learn

#include <stdio.h>
int main()
{
    printf(“Hello World!\n”);
    return 0x1234;
}

Is the same as...

```
.text:00401730 main
.text:00401730     push    ebp
.text:00401731     mov     ebp, esp
.text:00401733     push    offset aHelloWorld ; "Hello world\n"
.text:00401738     call    ds:__imp__printf
.text:0040173E     add     esp, 4
.text:00401741     mov     eax, 1234h
.text:00401746     pop     ebp
.text:00401747     retn
```

Windows Visual C++ 2005, /GS (buffer overflow protection) option turned off
Disassembled with IDA Pro 4.9 Free Version
Is the same as...

08048374 <main>:

08048374: 8d 4c 24 04  lea 0x4(%esp),%ecx
08048378: 83 e4 f0  and $0xfffffffff0,%esp
0804837b: ff 71 fc  pushl -0x4(%ecx)
0804837e: 55  push %ebp
0804837f: 89 e5  mov %esp,%ebp
08048381: 51  push %ecx
08048382: 83 ec 04  sub $0x4,%esp
08048385: c7 04 24 60 84 04 08  movl $0x8048460,(%esp)
0804838c: e8 43 ff ff ff  call 80482d4 <puts@plt>
08048391: b8 2a 00 00 00  mov $0x1234,%eax
08048396: b8 2a 00 00 00  mov $0x1234,%eax
08048399: 59  add $0x4,%esp
0804839a: 5d  pop %ecx
0804839b: 5d  pop %ebp
0804839e: c3  lea -0x4(%ecx),%esp
0804839f: 90  ret

Ubuntu 8.04, GCC 4.2.4
Disassembled with “objdump -d”
Is the same as...

_main:
00001fca  pushl  %ebp
00001fcb  movl  %esp,%ebp
00001fcd  pushl  %ebx
00001fce  subl  $0x14,%esp
00001fd1  calll  0x00001fd6
00001fd6  popl  %ebx
00001fd7  leal  0x0000001a(%ebx),%eax
00001fd9  movl  %eax,(%esp)
00001fe0  calll  0x00003005 ; symbol stub for: _.puts
00001fe5  movl  $0x00001234,%eax
00001fea  addl  $0x14,%esp
00001fed  popl  %ebx
00001fee  leave
00001fef  ret

Mac OS 10.5.6, GCC 4.0.1
Disassembled from command line with “otool -tV”
But it all boils down to...

```
.text:00401000 main
.text:00401000 push offset aHelloWorld ; "Hello world\n"
.text:00401005 call ds:__imp__printf
.text:0040100B pop ecx
.text:0040100C mov eax, 1234h
.text:00401011 ret
```

Windows Visual C++ 2005, /GS (buffer overflow protection) option turned off
Optimize for minimum size (/O1) turned on
Disassembled with IDA Pro 4.9 Free Version
Take Heart!

- By one measure, only 14 assembly instructions account for 90% of code!
- I think that knowing about 20-30 (not counting variations) is good enough that you will have the check the manual very infrequently
- You've already seen 11 instructions, just in the hello world variations!
Refresher - Data Types

In C:
- char
- short
- int/long
- double/long long

?->
- long double?

Figure 4-1. Fundamental Data Types
Refresher - Alt. Radices
Decimal, Binary, Hexidecimal

If you don't know this, you must memorize tonight

<table>
<thead>
<tr>
<th>Decimal (base 10)</th>
<th>Binary (base 2)</th>
<th>Hex (base 16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0000b</td>
<td>0x00</td>
</tr>
<tr>
<td>01</td>
<td>0001b</td>
<td>0x01</td>
</tr>
<tr>
<td>02</td>
<td>0010b</td>
<td>0x02</td>
</tr>
<tr>
<td>03</td>
<td>0011b</td>
<td>0x03</td>
</tr>
<tr>
<td>04</td>
<td>0100b</td>
<td>0x04</td>
</tr>
<tr>
<td>05</td>
<td>0101b</td>
<td>0x05</td>
</tr>
<tr>
<td>06</td>
<td>0110b</td>
<td>0x06</td>
</tr>
<tr>
<td>07</td>
<td>0111b</td>
<td>0x07</td>
</tr>
<tr>
<td>08</td>
<td>1000b</td>
<td>0x08</td>
</tr>
<tr>
<td>09</td>
<td>1001b</td>
<td>0x09</td>
</tr>
<tr>
<td>10</td>
<td>1010b</td>
<td>0x0A</td>
</tr>
<tr>
<td>11</td>
<td>1011b</td>
<td>0x0B</td>
</tr>
<tr>
<td>12</td>
<td>1100b</td>
<td>0x0C</td>
</tr>
<tr>
<td>13</td>
<td>1101b</td>
<td>0x0D</td>
</tr>
<tr>
<td>14</td>
<td>1110b</td>
<td>0x0E</td>
</tr>
<tr>
<td>15</td>
<td>1111b</td>
<td>0x0F</td>
</tr>
</tbody>
</table>
Refresher - Negative Numbers

- “one's complement” = flip all bits. 0->1, 1->0
- “two's complement” = one's complement + 1
- Negative numbers are defined as the “two's complement” of the positive number

<table>
<thead>
<tr>
<th>Number</th>
<th>One's Comp.</th>
<th>Two's Comp. (negative)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001b : 0x01</td>
<td>11111110b : 0xFE</td>
<td>11111111b : 0xFF : -1</td>
</tr>
<tr>
<td>00000100b : 0x04</td>
<td>11111011b : 0xFB</td>
<td>11111100b : 0xFC : -4</td>
</tr>
<tr>
<td>00011010b : 0x1A</td>
<td>11001011b : 0xE5</td>
<td>11001100b : 0xE6 : -26</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>10110000b : 0xB0 : -?</td>
</tr>
</tbody>
</table>

- 0x01 to 0x7F positive byte, 0x80 to 0xFF negative byte
- 0x00000001 to 0xffffffff positive dword
- 0x80000000 to 0xffffffff negative dword
Architecture - CISC vs. RISC

• Intel is CISC - Complex Instruction Set Computer
  – Many very special purpose instructions that you will never see, and a given compiler may never use - just need to know how to use the manual
  – Variable-length instructions, between 1 and 16(?) bytes long.
    • 16 is max len in theory, I don't know if it can happen in practice

• Other major architectures are typically RISC - Reduced Instruction Set Computer
  – Typically more registers, less and fixed-size instructions
  – Examples: PowerPC, ARM, SPARC, MIPS
Architecture - Endian

• Endianness comes from Jonathan Swift's *Gulliver's Travels*. It doesn't matter which way you eat your eggs :)

• Little Endian - 0x12345678 stored in RAM “little end” first. The least significant byte of a word or larger is stored in the lowest address. E.g. 0x78563412
  – Intel is Little Endian

• Big Endian - 0x12345678 stored as is.
  – Network traffic is Big Endian
  – Most everyone else you've heard of (PowerPC, ARM, SPARC, MIPS) is either Big Endian by default or can be configured as either (Bi-Endian)

Book p. 163
Book for this class is “Professional Assembly Language” by Blum
Endianess pictures

**Big Endian (Others)**

- **Register**
  - FE
  - ED
  - FA
  - CE

- **High Memory Addresses**
  - 00
  - 00
  - CE
  - FA
  - ED
  - FE

**Little Endian (Intel)**

- **Register**
  - FE
  - ED
  - FA
  - CE

- **Low Memory Addresses**
  - CE
  - FA
  - ED
  - FE
  - 0x0
  - 0x1
  - 0x2
  - 0x3
  - 0x4
  - 0x5
Architecture - Registers

• Registers are small memory storage areas built into the processor (still volatile memory)
• 8 "general purpose" registers + the instruction pointer which points at the next instruction to execute
  – But two of the 8 are not that general
• On x86-32, registers are 32 bits long
• On x86-64, they're 64 bits
These are Intel's suggestions to compiler developers (and assembly handcoders). Registers don't have to be used these ways, but if you see them being used like this, you'll know why. But I simplified some descriptions. I also color coded as **GREEN** for the ones which we will actually see in *this* class (as opposed to future ones), and **RED** for not.

- **EAX** – Stores function return values
- **EBX** – Base pointer to the data section
- **ECX** – Counter for string and loop operations
- **EDX** – I/O pointer
Architecture - Registers

Conventions 2

- **ESI** – Source pointer for string operations
- **EDI** – Destination pointer for string operations
- **ESP** – Stack pointer
- **EBP** – Stack frame base pointer
- **EIP** - Pointer to next instruction to execute ("instruction pointer")
Architecture - Registers
Conventions 3

• Caller-save registers - eax, edx, ecx
  – If the caller has anything in the registers that it cares about, the caller is in charge of saving the value before a call to a subroutine, and restoring the value after the call returns
  – Put another way - the callee can (and is highly likely to) modify values in caller-save registers

• Callee-save registers - ebp, ebx, esi, edi
  – If the callee needs to use more registers than are saved by the caller, the callee is responsible for making sure the values are stored/restored
  – Put another way - the callee must be a good citizen and not modify registers which the caller didn't save, unless the callee itself saves and restores the existing values
Architecture - Registers - 8/16/32 bit addressing

<table>
<thead>
<tr>
<th>8/16/32bit general purpose registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>AX</th>
<th>AL</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AH</td>
<td>AL</td>
</tr>
<tr>
<td>ECX</td>
<td>CX</td>
<td>CL</td>
</tr>
<tr>
<td>reserved</td>
<td>CH</td>
<td>CL</td>
</tr>
<tr>
<td>EDX</td>
<td>DX</td>
<td>DL</td>
</tr>
<tr>
<td>reserved</td>
<td>DH</td>
<td>DL</td>
</tr>
<tr>
<td>EBX</td>
<td>BX</td>
<td>BL</td>
</tr>
<tr>
<td>reserved</td>
<td>BH</td>
<td>BL</td>
</tr>
</tbody>
</table>

http://www.sandpile.org/ia32/reg.htm
Architecture - Registers - 8/16/32 bit addressing 2

http://www.sandpile.org/ia32/reg.htm
Architecture - EFLAGS

- EFLAGS register holds many single bit flags. Will only ask you to remember the following for now.
  - Zero Flag (ZF) - Set if the result of some instruction is zero; cleared otherwise.
  - Sign Flag (SF) - Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)
Your first x86 instruction: NOP

- NOP - No Operation! No registers, no values, no nothin'!
- Just there to pad/align bytes, or to delay time
- Bad guys use it to make simple exploits more reliable. But that's another class ;)

1
Extra! Extra!
Late-breaking NOP news!

• Amaze those who know x86 by citing this interesting bit of trivia:
  • “The one-byte NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.”
    – I had never looked in the manual for NOP apparently :)
• Every other person I had told this to had never heard it either.
• Thanks to Jon Erickson for cluing me in to this.
• XCHG instruction is not officially in this class. But if I hadn't just told you what it does, I bet you would have guessed right anyway.

XCHG in book p. 112
The Stack

• The stack is a conceptual area of main memory (RAM) which is designated by the OS when a program is started.
  – Different OS start it at different addresses by convention

• A stack is a Last-In-First-Out (LIFO/FILO) data structure where data is "pushed" on to the top of the stack and "popped" off the top.

• By convention the stack grows toward lower memory addresses. Adding something to the stack means the top of the stack is now at a lower memory address.
The Stack 2

• As already mentioned, esp points to the top of the stack, the lowest address which is being used
  – While data will exist at addresses beyond the top of the stack, it is considered undefined
• The stack keeps track of which functions were called before the current one, it holds local variables and is frequently used to pass arguments to the next function to be called.
• A firm understanding of what is happening on the stack is *essential* to understanding a program's operation.
PUSH - Push Word, Doubleword or Quadword onto the Stack

- For our purposes, it will always be a DWORD (4 bytes).
  - Can either be an immediate (a numeric constant), or the value in a register
- The push instruction automatically decrements the stack pointer, esp, by 4.
push eax

Stack Before

Stack After

Registers Before

<table>
<thead>
<tr>
<th>eax</th>
<th>0x00000003</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td>0x0012FF8C</td>
</tr>
</tbody>
</table>

Registers After

<table>
<thead>
<tr>
<th>eax</th>
<th>0x00000003</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td>0x0012FF88</td>
</tr>
</tbody>
</table>
POP - Pop a Value from the Stack

- Take a DWORD off the stack, put it in a register, and increment esp by 4

Book p. 120
pop eax

Stack Before

Stack After

Registers Before

<table>
<thead>
<tr>
<th>eax</th>
<th>0xFFFFFFFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td>0x0012FF88</td>
</tr>
</tbody>
</table>

Registers After

<table>
<thead>
<tr>
<th>eax</th>
<th>0x00000003</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td>0x0012FF8C</td>
</tr>
</tbody>
</table>

Stack Before

<table>
<thead>
<tr>
<th>0x0012FF90</th>
<th>0x0012FF8C</th>
<th>0x0012FF88</th>
<th>0x0012FF84</th>
<th>0x0012FF80</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000001</td>
<td>0x00000002</td>
<td>0x00000003</td>
<td>undefined</td>
<td>undefined</td>
</tr>
</tbody>
</table>

Stack After

<table>
<thead>
<tr>
<th>0x0012FF90</th>
<th>0x0012FF8C</th>
<th>0x0012FF88</th>
<th>0x0012FF84</th>
<th>0x0012FF80</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000001</td>
<td>0x00000002</td>
<td>undefined</td>
<td>undefined</td>
<td>undefined</td>
</tr>
</tbody>
</table>
Calling Conventions

• How code calls a subroutine is compiler-dependent and configurable. But there are a few conventions.
• We will only deal with the “cdecl” and “stdcall” conventions.
• More info at
  – http://www.programmersheaven.com/2/Calling-conventions
Calling Conventions - cdecl

• “C declaration” - most common calling convention
• Function parameters pushed onto stack right to left
• Saves the old stack frame pointer and sets up a new stack frame
• eax or edx:eax returns the result for primitive data types
• Caller is responsible for cleaning up the stack
Calling Conventions - stdcall

• I typically only see this convention used by Microsoft C++ code - e.g. Win32 API
• Function parameters pushed onto stack right to left
• Saves the old stack frame pointer and sets up a new stack frame
• eax or edx:eax returns the result for primitive data types
• Callee responsible for cleaning up any stack parameters it takes
• Aside: typical MS, “If I call my new way of doing stuff 'standard' it must be true!”
CALL - Call Procedure

• CALL's job is to transfer control to a different function, in a way that control can later be resumed where it left off.

• First it pushes the address of the next instruction onto the stack
  – For use by RET for when the procedure is done

• Then it changes eip to the address given in the instruction

• Destination address can be specified in multiple ways
  – Absolute address
  – Relative address (relative to the end of the instruction)
RET - Return from Procedure

- Two forms
  - Pop the top of the stack into eip (remember pop increments stack pointer)
    - In this form, the instruction is just written as “ret”
    - Typically used by cdecl functions
  - Pop the top of the stack into eip and add a constant number of bytes to esp
    - In this form, the instruction is written as “ret 0x8”, or “ret 0x20”, etc
    - Typically used by stdcall functions

Kinda book p. 133
MOV - Move

• Can move:
  – register to register
  – memory to register, register to memory
  – immediate to register, immediate to memory

• Never memory to memory!

• Memory addresses are given in r/m32 form talked about later

Book p. 97
General Stack Frame Operation

We are going to pretend that main() is the very first function being executed in a program. This is what its stack looks like to start with (assuming it has any local variables).

Local Variables

<table>
<thead>
<tr>
<th>stack bottom</th>
</tr>
</thead>
<tbody>
<tr>
<td>main() frame</td>
</tr>
<tr>
<td>undef</td>
</tr>
<tr>
<td>undef</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>stack top</td>
</tr>
</tbody>
</table>

Book p. 306
General Stack Frame Operation 2

When main() decides to call a subroutine, main() becomes “the caller”. We will assume main() has some registers it would like to remain the same, so it will save them. We will also assume that the callee function takes some input arguments.

<table>
<thead>
<tr>
<th>Local Variables</th>
<th>stack bottom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caller-Save Registers</td>
<td>main() frame</td>
</tr>
<tr>
<td>Arguments to Pass to Callee</td>
<td>undefined</td>
</tr>
<tr>
<td></td>
<td>undefined</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

stack top
General Stack Frame Operation 3

When `main()` actually issues the CALL instruction, the return address gets saved onto the stack, and because the next instruction after the call will be the beginning of the called function, we consider the frame to have changed to the callee.
General Stack Frame Operation 4

When foo() starts, the frame pointer (ebp) still points to main()'s frame. So the first thing it does is to save the old frame pointer on the stack and set the new value to point to its own frame.

<table>
<thead>
<tr>
<th>Local Variables</th>
<th>stack bottom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caller-Save Registers</td>
<td>main() frame</td>
</tr>
<tr>
<td>Arguments to Pass to Callee</td>
<td>foo()'s frame</td>
</tr>
<tr>
<td>Caller's saved return address</td>
<td>undef</td>
</tr>
<tr>
<td>Saved Frame Pointer</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>stack top</td>
</tr>
</tbody>
</table>
Next, we'll assume the callee foo() would like to use all the registers, and must therefore save the callee-save registers. Then it will allocate space for its local variables.

<table>
<thead>
<tr>
<th>Local Variables</th>
<th>stack bottom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caller-Save Registers</td>
<td>main() frame</td>
</tr>
<tr>
<td>Arguments to Pass to Callee</td>
<td>foo()'s frame</td>
</tr>
<tr>
<td>Caller's saved return address</td>
<td>undef</td>
</tr>
<tr>
<td>Saved Frame Pointer</td>
<td>...</td>
</tr>
<tr>
<td>Callee-Save Registers</td>
<td>stack top</td>
</tr>
<tr>
<td>Local Variables</td>
<td></td>
</tr>
</tbody>
</table>
General Stack Frame Operation 6

At this point, foo() decides it wants to call bar(). It is still the callee-of-main(), but it will now be the caller-of-bar. So it saves any caller-save registers that it needs to. It then puts the function arguments on the stack as well.
General Stack Frame Layout

Every part of the stack frame is technically optional (that is, you can hand code asm without following the conventions.) But compilers generate code which uses portions if they are needed. Which pieces are used can sometimes be manipulated with compiler options. (E.g. omit frame pointers, changing calling convention to pass arguments in registers, etc.)
Stack Frames are a Linked List!

The ebp in the current frame points at the saved ebp of the previous frame.
Example1.c

The stack frames in this example will be very simple. Only saved frame pointer (ebp) and saved return addresses (eip).

//Example1 - using the stack
//to call subroutines
//New instructions:
//push, pop, call, ret, mov
int sub(){
    return 0xbeef;
}

int main(){
    sub();
    return 0xf00d;
}
Example1.c 1:
EIP = 00401010, but no instruction yet executed

Key:
- □ executed instruction,
- ○ modified value
- ✗ start value

<table>
<thead>
<tr>
<th>eax</th>
<th>0x003435C0 ✗</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FFB8 ✗</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF6C ✗</td>
</tr>
</tbody>
</table>

sub:
- 00401000 push ebp
- 00401001 mov ebp,esp
- 00401003 mov eax,0BEEFh
- 00401008 pop ebp
- 00401009 ret

main:
- 00401010 push ebp
- 00401011 mov ebp,esp
- 00401013 call sub (401000h)
- 00401018 mov eax,0F00Dh
- 0040101D pop ebp
- 0040101E ret
Example1.c 2

<table>
<thead>
<tr>
<th></th>
<th>eax</th>
<th>ebp</th>
<th>esp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x003435C0</td>
<td>0x0012FFB8</td>
<td>0x0012FF68</td>
</tr>
</tbody>
</table>

Key:
- □ executed instruction,
- ▼ modified value,
- ♠ start value

```
sub:
00401000  push        ebp
00401001  mov         ebp,esp
00401003  mov         eax,0BEEFh
00401008  pop         ebp
00401009  ret
main:
00401010  push        ebp □
00401011  mov         ebp,esp
00401013  call        sub (401000h)
00401018  mov         eax,0F00Dh
0040101D  pop         ebp
0040101E  ret
```
Example1.c

<table>
<thead>
<tr>
<th>eax</th>
<th>0x003435C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FF68</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF68</td>
</tr>
</tbody>
</table>

Key:
- ✗ executed instruction,
- ♣ modified value
- ⨁ start value

sub:
- 00401000 push ebp
- 00401001 mov ebp,esp
- 00401003 mov eax,0BEEFh
- 00401008 pop ebp
- 00401009 ret

Main:
- 00401010 push ebp
- 00401011 mov ebp,esp ✗
- 00401013 call sub (401000h)
- 00401018 mov eax,0F00Dh
- 0040101D pop ebp
- 0040101E ret
Example1.c 4

**Key:**
- ✗ executed instruction,
- ☑ modified value
- ✗ start value

<table>
<thead>
<tr>
<th>eax</th>
<th>0x003435C0 ✗</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FF68</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF64 ☑</td>
</tr>
</tbody>
</table>

**sub:**
- 00401000 push ebp
- 00401001 mov ebp,esp
- 00401003 mov eax,0BEEFh
- 00401008 pop ebp
- 00401009 ret

**main:**
- 00401010 push ebp
- 00401011 mov ebp,esp
- 00401013 call sub (401000h) ✗
- 00401018 mov eax,0F00Dh
- 0040101D pop ebp
- 0040101E ret
Example1.c 5

<table>
<thead>
<tr>
<th>eax</th>
<th>0x003435C0 ━━ x</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FF68</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF60  Mr</td>
</tr>
</tbody>
</table>

Key:

 executed instruction,  
Mr modified value  
☆ start value  

sub:
00401000 push ebp  Mr
00401001 mov ebp,esp
00401003 mov eax,0BEEFh
00401008 pop ebp
00401009 ret

main:
00401010 push ebp
00401011 mov ebp,esp
00401013 call sub (401000h)
00401018 mov eax,0F00Dh
0040101D pop ebp
0040101E ret
Example1.c 6

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x003435C0 ※</td>
</tr>
<tr>
<td>ebp</td>
<td>0x0012FF60 ▶</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF60</td>
</tr>
</tbody>
</table>

Key:
※ executed instruction,
▶ modified value
※ start value

sub:
00401000 push ebp
00401001 mov ebp,esp ※
00401003 mov eax,0BEEFh
00401008 pop ebp
00401009 ret

main:
00401010 push ebp
00401011 mov ebp,esp
00401013 call sub (401000h)
00401018 mov eax,0F00Dh
0040101D pop ebp
0040101E ret
Example1.c 6
STACK FRAME TIME OUT

```
sub
push    ebp
mov     ebp, esp
mov     eax, 0BEEFh
pop     ebp
retn

main
push    ebp
mov     ebp, esp
call    _sub
mov     eax, 0F00Dh
pop     ebp
retn
```

"Function-before-main"'s frame

main's frame
(saved frame pointer and saved return address)

sub's frame
(only saved frame pointer, because it doesn't call anything else, and doesn't have local variables)

```
  0x0012FFC
  0x0012FF8
  0x0012FF4
  0x0012FF0
  0x0012F5C
  0x0012F58
  0x004012E8
  0x0012FFB8
  0x00401018
  0x0012FF68
  undef
  undef
```

58
Example1.c 7

<table>
<thead>
<tr>
<th>eax</th>
<th>0x0000BEEF</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FF60</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF60</td>
</tr>
</tbody>
</table>

Key:
- • executed instruction,
- # modified value
- * start value

sub:
- 00401000 push ebp
- 00401001 mov ebp, esp
- 00401003 mov eax, 0BEEFh *
- 00401008 pop ebp
- 00401009 ret

main:
- 00401010 push ebp
- 00401011 mov ebp, esp
- 00401013 call sub (401000h)
- 00401018 mov eax, 0F00Dh
- 0040101D pop ebp
- 0040101E ret

0x0012FF6C 0x004012E8 *
0x0012FF68
0x0012FF64
0x0012FF60
0x0012FF60
0x0012FF60
0x0012FF60
0x0012FF60
0x0012FF60
0x0012FF60
0x0012FF60
0x0012FF60
0x0012FF60
0x0012FF60
Example1.c 8

<table>
<thead>
<tr>
<th>eax</th>
<th>0x0000BEEF</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FF68</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF64</td>
</tr>
</tbody>
</table>

**Key:**
- □ executed instruction,
- ![modified value](image) modified value
- * start value

**sub:**
- 00401000  push  ebp
- 00401001  mov  ebp,esp
- 00401003  mov  eax,0BEEFh
- 00401008  pop  ebp  □
- 00401009  ret

**main:**
- 00401010  push  ebp
- 00401011  mov  ebp,esp
- 00401013  call  sub (401000h)
- 00401018  mov  eax,0F00Dh
- 0040101D  pop  ebp
- 0040101E  ret
Example1.c 9

<table>
<thead>
<tr>
<th>eax</th>
<th>0x0000BEEF</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FF68</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF68 ▼</td>
</tr>
</tbody>
</table>

### Key:
- □ executed instruction
- ▼ modified value
- ★ start value

sub:
- `00401000` push ebp
- `00401001` mov ebp,esp
- `00401003` mov eax,0BEEFh
- `00401008` pop ebp
- `00401009` ret □

main:
- `00401010` push ebp
- `00401011` mov ebp,esp
- `00401013` call sub (401000h)
- `00401018` mov eax,0F00Dh
- `0040101D` pop ebp
- `0040101E` ret
Example 1.c 9

<table>
<thead>
<tr>
<th>Key:</th>
<th>Executed instruction,</th>
<th>Modified value</th>
<th>Start value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### sub:
```
0x0012FF6C  push  ebp
0x0012FF68  mov  ebp,esp
0x0012FF6C  mov  eax,0BEEFh
0x0012FF68  pop  ebp
0x0012FF6C  ret
```

### main:
```
0x0012FF58  push  ebp
0x0012FF5C  mov  ebp,esp
0x0012FF58  call  sub (401000h)
0x0012FF5C  mov  eax,0F00Dh  ✗
0x0012FF58  pop  ebp
0x0012FF5C  ret
```

### Table:

<table>
<thead>
<tr>
<th>eax</th>
<th>0x0000F00D ✷</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FF68</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF68</td>
</tr>
</tbody>
</table>
Example1.c 10

<table>
<thead>
<tr>
<th>eax</th>
<th>0x0000F00D</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FFB8</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF6C</td>
</tr>
</tbody>
</table>

### sub:
- BF100000 push ebp
- BF100001 mov ebp,esp
- BF100003 mov eax,0BEEFH
- BF100008 pop ebp
- BF100009 ret

### main:
- BF101010 push ebp
- BF101011 mov ebp,esp
- BF101013 call sub (401000h)
- BF101018 mov eax,0F00Dh
- BF10101D pop ebp √
- BF10101E ret

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012FF6C</td>
<td>0x004012E8 √</td>
</tr>
<tr>
<td>0x0012FF68</td>
<td>undef</td>
</tr>
<tr>
<td>0x0012FF64</td>
<td>undef</td>
</tr>
<tr>
<td>0x0012FF60</td>
<td>undef</td>
</tr>
<tr>
<td>0x0012FF5C</td>
<td>undef</td>
</tr>
<tr>
<td>0x0012FF58</td>
<td>undef</td>
</tr>
</tbody>
</table>

**Key:**
- √ executed instruction,
- √ modified value
- √ start value

---

63
Example1.c 11

<table>
<thead>
<tr>
<th>eax</th>
<th>0x000000F0D</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FFB8</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF70</td>
</tr>
</tbody>
</table>

**Key:**
- 🔴 executed instruction
- 🔄 modified value
- ✴️ start value

**sub:**
- 00401000 push ebp
- 00401001 mov ebp, esp
- 00401003 mov eax, 0BEEFH
- 00401008 pop ebp
- 00401009 ret

**main:**
- 00401010 push ebp
- 00401011 mov ebp, esp
- 00401013 call sub (401000h)
- 00401018 mov eax, 0F00Dh
- 0040101D pop ebp
- 0040101E ret 🔴

Execution would continue at the value ret removed from the stack: 0x004012E8
Example1 Notes

• `sub()` is deadcode - its return value is not used for anything, and `main` always returns 0xF00D. If optimizations are turned on in the compiler, it would remove `sub()`

• Because there are no input parameters to `sub()`, there is no difference whether we compile as cdecl vs stdcall calling conventions
Let's do that in a tool

- Visual C++ 2008 Express Edition (which I will shorthand as “VisualStudio” or VS)
- Standard Windows development environment
- Available for free, but missing some features that pro developers might want
- Can't move applications to other systems without installing the “redistributable libraries”
Finding the Visual Studio solution file
Creating a new project - 1
Creating a new project - 2
Creating a new project - 3
Adding files to the project

Example1
- Header Files
- Resource Files
- Source Files

Add New Item - Example1

Categories:
- Visual C++
  - UI
  - Code
  - Property Sheets

Templates:
- Visual Studio installed templates
  - Windows Form
  - Header File (.h)
  - Component (.cpp)
  - Property Sheet (.vsprops)

My Templates
- Search Online Templates...

Creates a file containing C++ source code

Name: Example1.c
Location: c:\Documents and Settings\Administrator\My Documents\Visual Studio 2008\Projects\IntroToAsm
Setting project properties - 1
Unfortunately the debug information format alters the code which gets generated too much, making it not as simple as I would like for this class.
This would just add extra complexity to the asm which we don't want for now.
Different options can be set for release vs debug builds.

It's all just a wrapper to set command line options.

Click this to change which config set is active.
C++ has more complicated compiler-generated code, and while our stuff is simple enough that the compiler probably wouldn't do anything different, it's good to do this just to be safe.
Another thing where I found out the hard way that it will increase the asm complexity.
Building the project - 1
Building the project - 2

Information about whether the build succeeded will be here. If it fails, a separate error tab will open up.
Setting breakpoints & start debugger

Click to the left of the line to break at.
Showing assembly

Right click: Only available while debugging
Note that it knows the ebp register is going to be used in this instruction.
Showing registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x003435c0</td>
<td>unsigned</td>
</tr>
<tr>
<td>ebx</td>
<td>0x7ffdf000</td>
<td>unsigned</td>
</tr>
<tr>
<td>ecx</td>
<td>0x00343250</td>
<td>unsigned</td>
</tr>
<tr>
<td>edx</td>
<td>0x00000001</td>
<td>unsigned</td>
</tr>
<tr>
<td>edi</td>
<td>0x07abf9d4</td>
<td>unsigned</td>
</tr>
<tr>
<td>esi</td>
<td>0x00000000</td>
<td>unsigned</td>
</tr>
</tbody>
</table>

Here you can enter register names or variable names
Watching the stack change - 1
Watching the stack change - 2

Set address to esp (will always be the top of the stack)

Right click on the body of the data in the window and make sure everything's set like this

Set to 1

Click “Reevaluate Automatically” so that it will change the display as esp changes
Going through Example1.c in Visual Studio

sub:
  push    ebp
  mov     ebp,esp
  mov     eax,0BEEFh
  pop     ebp
  ret

main:
  push    ebp
  mov     ebp,esp
  call    sub
  mov     eax,0F00Dh
  pop     ebp
  ret
#include <stdlib.h>

int sub(int x, int y){
    return 2*x+y;
}

int main(int argc, char ** argv){
    int a;
    a = atoi(argv[1]);
    return sub(argc,a);
}
"r/m32" Addressing Forms

• Anywhere you see an r/m32 it means it could be taking a value either from a register, or a memory address.
• I'm just calling these “r/m32 forms” because anywhere you see “r/m32” in the manual, the instruction can be a variation of the below forms.
• In Intel syntax, most of the time square brackets [] means to treat the value within as a memory address, and fetch the value at that address (like dereferencing a pointer)
  – mov eax, ebx
  – mov eax, [ebx]
  – mov eax, [ebx+ecx*X] (X=1, 2, 4, 8)
  – mov eax, [ebx+ecx*X+Y] (Y= one byte, 0-255 or 4 bytes, 0-2^32-1)
• Most complicated form is: [base + index*scale + disp]

More info: Intel v2a, Section 2.1.5 page 2-4
in particular Tables 2-2 and 2-3
LEA - Load Effective Address

- Frequently used with pointer arithmetic, sometimes for just arithmetic in general
- Uses the r/m32 form but **is the exception to the rule** that the square brackets [ ] syntax means dereference ("value at")
- Example: ebx = 0x2, edx = 0x1000
  - lea eax, [edx+ebx*2]
  - eax = 0x1004, not the value at 0x1004

Not covered in book
ADD and SUB

• Adds or Subtracts, just as expected
• Destination operand can be r/m32 or register
• Source operand can be r/m32 or register or immediate
• No source \texttt{and} destination as r/m32s, because that could allow for memory to memory transfer, which isn't allowed on x86
• Evaluates the operation as if it were on signed AND unsigned data, and sets flags as appropriate. Instructions modify OF, SF, ZF, AF, PF, and CF flags
  • add esp, 8
  • sub eax, [ebx*2]

### Example2.c - 1

<table>
<thead>
<tr>
<th>eax</th>
<th>0xcafe ✹</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecx</td>
<td>0xbabe ✹</td>
</tr>
<tr>
<td>edx</td>
<td>0xfeed ✹</td>
</tr>
<tr>
<td>ebp</td>
<td>0x0012FF50 ✹</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF24 ⬤</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addr after “call _main” ✹</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012FF24 ⬤</td>
</tr>
</tbody>
</table>

#### Key:
- **executed instruction** ✹, **modified value** ⬤, **arbitrary example start value** ✹
Example2.c - 2

.text:00000000 _sub:
    push    ebp
.text:00000001     mov     ebp, esp
.text:00000003     mov     eax, [ebp+8]
.text:00000006     mov     ecx, [ebp+0Ch]
.text:00000009     lea     eax, [ecx+eax*2]
.text:0000000C     pop     ebp
.text:0000000D     retn
.text:00000010 _main:
    push    ebp
.text:00000011     mov     ebp, esp
.text:00000013     push    ecx
.text:00000014     mov     eax, [ebp+0Ch]
.text:00000017     mov     ecx, [eax+4]
.text:0000001B     push    ecx
.text:0000001B     call    dword ptr ds:__imp__atoi
.text:00000021     add     esp, 4
.text:00000024     mov     [ebp-4], eax
.text:00000027     mov     edx, [ebp-4]
.text:0000002A     push    edx
.text:0000002B     mov     eax, [ebp-4]
.text:0000002E     push    eax
.text:0000002F     call    _sub
.text:00000034     add     esp, 8
.text:00000037     mov     esp, ebp
.text:00000039     pop     ebp
.text:0000003A     retn

| eax     | 0xcafe |
| ecx     | 0xbabe |
| edx     | 0xfeed |
| ebp     | 0x0012FF24 |
| esp     | 0x0012FF24 |

0x0012FF00 (char ** argv)
0x0012FF0C 0x12FB0
0x0012FF20 0x02 (int argc)
0x0012FF24 Addr after “call _main”
0x0012FF28 0x0012FF50 (saved ebp)
0x0012FF30 0x0012FF20
0x0012FF34 0x0012FF1C
0x0012FF38 0x0012FF1E
0x0012FF40 0x0012FF20
0x0012FF44 0x0012FF20
0x0012FF48 0x0012FF20
0x0012FF4C 0x0012FF20
0x0012FF50 0x0012FF20
0x0012FF54 0x0012FF20
0x0012FF58 0x0012FF20
0x0012FF5C 0x0012FF20
0x0012FF60 0x0012FF20
0x0012FF64 0x0012FF20
93
Example2.c - 3

Caller-save, or space for local var? This time it turns out to be space for local var since there is no corresponding pop, and the address is used later to refer to the value we know is stored in a.
Example2.c - 4

Getting the base of the argv char * array (aka argv[0])

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x12FFB0</td>
</tr>
<tr>
<td>ecx</td>
<td>0xbabe</td>
</tr>
<tr>
<td>edx</td>
<td>0xfeed</td>
</tr>
<tr>
<td>ebp</td>
<td>0x0012FF24</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF20</td>
</tr>
</tbody>
</table>

0x12FFB0 (char ** argv)
0x02 (int argc)
Addr after “call _main”
0x0012FF50 (saved ebp)
0xbabe (int a)
undef
undef
undef
undef
0x0012FF2C
0x0012FF28
0x0012FF24
0x0012FF20
0x0012FF1C
0x0012FF18
0x0012FF14
0x0012FF10
0x0012FF0C
0x0012FF30
0x0012FF2C
0x0012FF28
0x0012FF24
0x0012FF20
0x0012FF1C
0x0012FF18
0x0012FF14
0x0012FF10
0x0012FF0C
0x0012FF1C
0x0012FF18
0x0012FF14
0x0012FF10
0x0012FF0C
0x0012FF1C
0x0012FF18
0x0012FF14
0x0012FF10
0x0012FF0C
Example 2 - 5

```
.text:00000000 _sub:         push    ebp
.text:00000001                 mov     ebp, esp
.text:00000003                 mov     eax, [ebp+8]
.text:00000006                 mov     ecx, [ebp+0Ch]
.text:00000009                 lea     eax, [ecx+eax*2]
.text:0000000C                 pop     ebp
.text:0000000D                 retn

.text:00000010 _main:      push    ebp
.text:00000011                 mov     ebp, esp
.text:00000013                 push    ecx
.text:00000014                 mov     eax, [ebp+0Ch]
.text:00000017                 mov     ecx, [eax+4]
.text:0000001A                 push    ecx
.text:0000001B                 call    dword ptr ds:__imp__atoi
.text:00000021                 add     esp, 4
.text:00000024                 mov     [ebp-4], eax
.text:00000028                 add     esp, 4
.text:0000002c                 add     esp, 4
.text:0000002f                 mov     edx, [ebp-4]
.text:00000032                 push    edx
.text:00000035                 mov     eax, [ebp-4]
.text:00000038                 push    eax
.text:0000003b                 call    _sub
.text:0000003e                 add     esp, 8
.text:00000042                 mov     esp, ebp
.text:00000045                 pop     ebp
.text:00000046                 ret
```

Getting the char * at argv[1]
(I chose 0x12FFD4 arbitrarily since it's out of the stack scope we're currently looking at)

<table>
<thead>
<tr>
<th>Segment</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x12FFB0</td>
<td>char ** argv</td>
</tr>
<tr>
<td>ecx</td>
<td>0x12FFD4 (arbitrary)</td>
<td></td>
</tr>
<tr>
<td>edx</td>
<td>0xfeed</td>
<td></td>
</tr>
<tr>
<td>ebp</td>
<td>0x0012FF24</td>
<td></td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF20</td>
<td></td>
</tr>
</tbody>
</table>

- 0x12FFB0: 0x12FFB0 (char ** argv)
- 0x12FFD4: 0x12FFD4 (arbitrary)
- 0x02: 0x2 (int argc)
- 0x0012FF28: Addr after “call _main”
- 0x0012FF24: 0x0012FF50 (saved ebp)
- 0x0012FF20: 0xbabe (int a)
- 0x0012FF1C: undefined
- 0x0012FF18: undefined
- 0x0012FF14: undefined
- 0x0012FF10: undefined
- 0x0012FF0C: undefined
Example 2 - 6

Saving some slides...
This will push the address of the string at argv[1] (0x12FFD4). atoi() will read the string and turn it into an int, put that int in eax, and return. Then the adding 4 to esp will negate the having pushed the input parameter and make 0x12FF1C undefined again (this is indicative of cdecl).

| eax     | 0x10000 (arbitrary) |
| ecx     | 0x12FFFD4 |
| edx     | 0xfeed |
| ebp     | 0x0012FF24 |
| esp     | 0x0012FF20 |

<table>
<thead>
<tr>
<th>Addr after “call _main”</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012FF30 0x12FFB0 (char ** argv)</td>
</tr>
<tr>
<td>0x0012FF2C 0x2 (int argc)</td>
</tr>
<tr>
<td>0x0012FF28 0x0012FF50 (saved ebp)</td>
</tr>
<tr>
<td>0x0012FF24 0x0012FF20 Oxbabe (int a)</td>
</tr>
<tr>
<td>0x0012FF1C undef</td>
</tr>
<tr>
<td>0x0012FF18 undef</td>
</tr>
<tr>
<td>0x0012FF14 undef</td>
</tr>
<tr>
<td>0x0012FF10 undef</td>
</tr>
<tr>
<td>0x0012FF0C undef</td>
</tr>
</tbody>
</table>
First setting “a” equal to the return value. Then pushing “a” as the second parameter in sub(). We can see an obvious optimization would have been to replace the last two instructions with “push eax”.

<table>
<thead>
<tr>
<th>eax</th>
<th>0x100</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecx</td>
<td>0x12FFD4</td>
</tr>
<tr>
<td>edx</td>
<td>0x100 (int a)</td>
</tr>
<tr>
<td>ebp</td>
<td>0x00012FF24</td>
</tr>
<tr>
<td>esp</td>
<td>0x00012FF1C (saved ebp)</td>
</tr>
<tr>
<td>addr after “call _main”</td>
<td>0x0012FF50</td>
</tr>
<tr>
<td>push ebp</td>
<td>0x0012FF30</td>
</tr>
<tr>
<td>mov ebp, esp</td>
<td>0x0012FF28</td>
</tr>
<tr>
<td>mov ecx, [ebp+0Ch]</td>
<td>0x0012FF24</td>
</tr>
<tr>
<td>add esp, 4</td>
<td>0x0012FF20</td>
</tr>
<tr>
<td>push ecx</td>
<td>0x0012FF2C</td>
</tr>
<tr>
<td>call dword ptr ds:__imp__atoi</td>
<td>0x0012FF18</td>
</tr>
<tr>
<td>add esp, 4</td>
<td>0x0012FF1C</td>
</tr>
<tr>
<td>mov [ebp-4], eax</td>
<td>0x0012FF14</td>
</tr>
<tr>
<td>mov edx, [ebp-4]</td>
<td>0x0012FF10</td>
</tr>
<tr>
<td>push edx</td>
<td>0x0012FF0C</td>
</tr>
<tr>
<td>mov eax, [ebp+8]</td>
<td></td>
</tr>
</tbody>
</table>
Example 2 - 8

```assembly
.text:00000000 _sub:
    push ebp
    mov ebp, esp
    mov eax, [ebp+8]
    mov ecx, [ebp+0Ch]
    lea eax, [ecx+eax*2]
    pop ebp
    ret

.text:00000010 _main:
    push ebp
    mov ebp, esp
    push ecx
    mov eax, [ebp+0Ch]
    mov ecx, [eax+4]
    push ecx
    call dword ptr ds:__imp__atoi
    add esp, 4
    mov [ebp-4], eax
    mov edx, [ebp-4]
    push edx
    mov [ebp-4], eax
    push [ebp-4]
    call _sub
    add esp, 8
    mov esp, ebp
    pop ebp
    ret
```

<table>
<thead>
<tr>
<th>Func</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x2</td>
</tr>
<tr>
<td>ecx</td>
<td>0x12FFD4</td>
</tr>
<tr>
<td>edx</td>
<td>0x100</td>
</tr>
<tr>
<td>ebp</td>
<td>0x0012FF24</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF18</td>
</tr>
</tbody>
</table>

Pushing argc as the first parameter (int x) to sub()
Example2 - 9

.text:00000000 _sub:        push    ebp
.text:00000001                 mov     ebp, esp
.text:00000003                 mov     eax, [ebp+8]
.text:00000006                 mov     ecx, [ebp+0Ch]
.text:00000009                 lea     eax, [ecx+eax*2]
.text:0000000C                 pop     ebp
.text:0000000D                 retn
.text:00000010 _main:     push    ebp
.text:00000011                 mov     ebp, esp
.text:00000013                 push    ecx
.text:00000014                 mov     eax, [ebp+0Ch]
.text:00000017                 mov     ecx, [eax+4]
.text:0000001A                 push    ecx
.text:0000001B                 call    dword ptr ds:__imp__atoi
.text:00000021                 add     esp, 4
.text:00000024                 mov     [ebp-4], eax
.text:00000027                 mov     edx, [ebp-4]
.text:0000002A                 push    edx
.text:0000002B                 mov     eax, [ebp+8]
.text:0000002E                 push    eax
.text:0000002F                 call    _sub
.text:00000034                 add     esp, 8
.text:00000037                 mov     esp, ebp
.text:00000039                 pop     ebp
.text:0000003A                 retn

| eax     | 0x2   |
| ecx     | 0x12FFD4 |
| edx     | 0x100  |
| ebp     | 0x0012FF24 |
| esp     | 0x0012FF14 |

| 0x0012FF30 | 0x12FFB0 (char ** argv) |
| 0x0012FF2C | 0x2 (int argc)          |
| 0x0012FF28 | Addr after “call _main” |
| 0x0012FF24 | 0x0012FF50 (saved ebp)  |
| 0x0012FF20 | 0x100 (int a)           |
| 0x0012FF1C | 0x100 (int y)           |
| 0x0012FF18 | 0x2 (int x)             |
| 0x0012FF14 | 0x000000034             |
| 0x0012FF10 | undef                   |
| 0x0012FF0C | undef                   |
Example2 - 10

.text:00000000 _sub:     push    ebp
.text:00000001         mov     ebp, esp
.text:00000003         mov     eax, [ebp+8]
.text:00000006         mov     ecx, [ebp+0Ch]
.text:00000009         lea     eax, [ecx+eax*2]
.text:0000000C         pop     ebp
.text:0000000D         ret
.text:00000010 _main:    push    ebp
.text:00000011         mov     ebp, esp
.text:00000013         push    ecx
.text:00000014         mov     eax, [ebp+0Ch]
.text:00000017         mov     ecx, [eax+4]
.text:00000018         push    ecx
.text:0000001B         call    dword ptr ds:__imp__atoi
.text:00000021         add     esp, 4
.text:00000024         mov     [ebp-4], eax
.text:00000027         mov     edx, [ebp-4]
.text:0000002A         push    edx
.text:0000002B         mov     eax, [ebp-8]
.text:0000002E         push    eax
.text:0000002F         call    _sub
.text:00000034         add     esp, 8
.text:00000037         mov     esp, ebp
.text:00000039         pop     ebp
.text:0000003A         ret

| eax  | 0x2   |
| ecx  | 0x12FFD4 |
| edx  | 0x100  |
| ebp  | 0x0012FF10 |
| esp  | 0x0012FF10 |

0x0012FF00 (char ** argv)
0x0012FF2C 0x2 (int argc)
0x0012FF28 Addr after “call _main”
0x0012FF24 0x0012FF50 (saved ebp)
0x0012FF20 0x100 (int a)
0x0012FF1C 0x100 (int y)
0x0012FF18 0x2 (int x)
0x0012FF14 0x00000034
0x0012FF10 0x0012FF24 (saved ebp)
0x0012FF0C undef

101
Example 2 - 11

Move “x” into eax, and “y” into ecx.

```
.text:00000000 _sub:
    push   ebp
    mov    ebp, esp
    mov    eax, [ebp+8] ☐
    mov    ecx, [ebp+0Ch] ☐
    lea    eax, [ecx+eax*2]
    pop    ebp
    ret

.text:00000010 _main:
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, [ebp+0Ch]
    mov    ecx, [eax+4]
    push   ecx
    call   dword ptr ds:__imp__atoi
    add    esp, 4
    mov    [ebp-4], eax
    mov    edx, [ebp-4]
    push   edx
    mov    eax, [ebp-8]
    push   eax
    call   _sub
    add    esp, 8
    mov    esp, ebp
    pop    ebp
    retn
```

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012FF0C</td>
<td>undef</td>
<td></td>
</tr>
<tr>
<td>0x0012FF10</td>
<td>0x00000034</td>
<td>(saved ebp)</td>
</tr>
<tr>
<td>0x0012FF14</td>
<td>0x02</td>
<td>(int x)</td>
</tr>
<tr>
<td>0x0012FF18</td>
<td>0x100</td>
<td>(int y)</td>
</tr>
<tr>
<td>0x0012FF20</td>
<td>0x100</td>
<td>(int a)</td>
</tr>
<tr>
<td>0x0012FF24</td>
<td>0x00012FF50 (saved ebp)</td>
<td>Addr after “call _main”</td>
</tr>
<tr>
<td>0x0012FF28</td>
<td>Addr</td>
<td></td>
</tr>
<tr>
<td>0x0012FF30</td>
<td>0x12FFB0 (char ** argv)</td>
<td></td>
</tr>
<tr>
<td>0x12FFB0</td>
<td>0x2</td>
<td>(no value change)</td>
</tr>
</tbody>
</table>
Example 2 - 12

Set the return value (eax) to \(2 \times x + y\).

Note: neither pointer arith, nor an “address” which was loaded. Just an efficient way to do a calculation.

```
.text:00010000 _sub:        push    ebp
                         mov     ebp, esp
                         mov     eax, [ebp+8]
                         mov     ecx, [ebp+0Ch]
                         lea     eax, [ecx+eax*2]  \\
                          pop      ebp
                         ret
                         push    ebp
                         mov     ebp, esp
                         push    ecx
                         mov     eax, [ebp+0Ch]
                         mov     ecx, [eax+4]
                         push    ecx
                         call    dword ptr ds:__imp__atoi
                         add     esp, 4
                         mov     [ebp-4], eax
                         mov     edx, [ebp-4]
                         push    edx
                         mov     eax, [ebp-4]
                         push    eax
                         call    _sub
                         add     esp, 8
                         mov     esp, ebp
                         pop      ebp
                         ret
```

| eax    | 0x104 |
| ecx    | 0x100 |
| edx    | 0x100 |
| ebp    | 0x0012FF10 |
| esp    | 0x0012FF10 |

<table>
<thead>
<tr>
<th>Addr after “call _main”</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12FFB0 (char ** argv)</td>
</tr>
<tr>
<td>0x02 (int argc)</td>
</tr>
<tr>
<td>0x0012FF28</td>
</tr>
<tr>
<td>0x0012FF50 (saved ebp)</td>
</tr>
<tr>
<td>0x100 (int a)</td>
</tr>
<tr>
<td>0x100 (int y)</td>
</tr>
<tr>
<td>0x2 (int x)</td>
</tr>
<tr>
<td>0x00000034</td>
</tr>
<tr>
<td>0x0012FF24 (saved ebp)</td>
</tr>
<tr>
<td>undef</td>
</tr>
</tbody>
</table>
Example2 - 13

```
.text:00000000 _sub:       push    ebp
.text:00000001           mov     ebp, esp
.text:00000003           mov     eax, [ebp+8]
.text:00000006           mov     ecx, [ebp+0Ch]
.text:00000009           lea     eax, [ecx+eax*2]
.text:0000000C      pop     ebp  
.text:0000000D     retn
.text:00000010 _main:     push    ebp
.text:00000011           mov     ebp, esp
.text:00000013           push    ecx
.text:00000014           mov     eax, [ebp+0Ch]
.text:00000017           mov     ecx, [eax+4]
.text:0000001A           push    ecx
.text:0000001B   call    dword ptr ds:___imp__atoi
.text:00000021       add     esp, 4
.text:00000024           mov     [ebp-4], eax
.text:00000027           mov     edx, [ebp-4]
.text:0000002A           push    edx
.text:0000002B           mov     eax, [ebp-4]
.text:0000002E           push    eax
.text:0000002F   call    _sub
.text:00000034       add     esp, 8
.text:00000037           mov     esp, ebp
.text:00000039           pop     ebp
.text:0000003A     retn
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12FB0</td>
<td>(char ** argv)</td>
</tr>
<tr>
<td>0x0012FF20</td>
<td>0x104</td>
</tr>
<tr>
<td>0x0012FF24</td>
<td>0x100</td>
</tr>
<tr>
<td>0x0012FF28</td>
<td>Addr after “call _main”</td>
</tr>
<tr>
<td>0x0012FF30</td>
<td>0x100</td>
</tr>
<tr>
<td>0x0012FF45</td>
<td>0x02 (int argc)</td>
</tr>
<tr>
<td>0x0012FF50</td>
<td>0x0012FF24</td>
</tr>
<tr>
<td>0x0012FF50</td>
<td>0x0012FF50 (saved ebp)</td>
</tr>
<tr>
<td>0x0012FF60</td>
<td>0x100</td>
</tr>
<tr>
<td>0x0012FF68</td>
<td>0x100</td>
</tr>
<tr>
<td>0x0012FF6B</td>
<td>0x02 (int x)</td>
</tr>
<tr>
<td>0x0012FF70</td>
<td>0x0012FF1C</td>
</tr>
<tr>
<td>0x0012FF78</td>
<td>0x0012FF10</td>
</tr>
<tr>
<td>0x0012FF80</td>
<td>undefined</td>
</tr>
<tr>
<td>0x0012FF88</td>
<td>undefined</td>
</tr>
<tr>
<td>0x0012FF90</td>
<td>undefined</td>
</tr>
<tr>
<td>0x0012FF98</td>
<td>undefined</td>
</tr>
<tr>
<td>0x0012FFA0</td>
<td>undefined</td>
</tr>
<tr>
<td>0x0012FFB0</td>
<td>undefined</td>
</tr>
<tr>
<td>0x0012FFC0</td>
<td>undefined</td>
</tr>
<tr>
<td>0x0012FFD0</td>
<td>undefined</td>
</tr>
<tr>
<td>0x0012FFE0</td>
<td>undefined</td>
</tr>
<tr>
<td>0x0012FFF0</td>
<td>undefined</td>
</tr>
</tbody>
</table>
Example2 - 14

.text:00000000  _sub:     push    ebp
.text:00000001 mov ebp, esp
.text:00000003 mov eax, [ebp+8]
.text:00000006 mov ecx, [ebp+0Ch]
.text:00000009 lea eax, [ecx+eax*2]
.text:0000000C pop ebp
.text:0000000D retn

.text:00000010  _main:     push    ebp
.text:00000011 mov ebp, esp
.text:00000013 push ecx
.text:00000014 mov eax, [ebp+0Ch]
.text:00000017 mov ecx, [eax+4]
.text:0000001A push ecx
.text:0000001B call dword ptr ds:__imp__atoi
.text:00000021 add esp, 4
.text:00000024 mov [ebp-4], eax
.text:00000027 mov edx, [ebp-4]
.text:0000002A push edx
.text:0000002B mov eax, [ebp-4]
.text:0000002E push eax
.text:0000002F call _sub
.text:00000034 add esp, 8
.text:00000037 mov esp, ebp
.text:00000039 pop ebp
.text:0000003A retn

<table>
<thead>
<tr>
<th>eax</th>
<th>0x104</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecx</td>
<td>0x100</td>
</tr>
<tr>
<td>edx</td>
<td>0x100</td>
</tr>
<tr>
<td>ebp</td>
<td>0x0012FF24</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF18</td>
</tr>
</tbody>
</table>

0x12FFB0 (char ** argv)
0x02 (int argc)
Addr after “call _main”
0x0012FF28
0x0012FF24
0x0012FF50 (saved ebp)
0x0012FF20
0x100 (int a)
0x0012FF1C
0x100 (int y)
0x0012FF18
0x02 (int x)
0x0012FF14
0x0012FF10
0x0012FF0C
undef
undef
### Example2 - 15

```assembly
.text:00000000 _sub:    push ebp
.text:00000001     mov ebp, esp
.text:00000003     mov eax, [ebp+8]
.text:00000006     mov ecx, [ebp+0Ch]
.text:00000009     lea eax, [ecx+eax*2]
.text:0000000C     pop ebp
.text:0000000D     retn
.text:0000000F _main:    push ebp
.text:00000011     mov ebp, esp
.text:00000013     push ecx
.text:00000014     mov eax, [ebp+0Ch]
.text:00000017     mov ecx, [eax+4]
.text:0000001A     push ecx
.text:0000001B     call dword ptr ds:__imp__atoi
.text:00000021     add esp, 4
.text:00000024     mov [ebp-4], eax
.text:00000027     mov edx, [ebp-4]
.text:0000002A     push edx
.text:0000002B     mov eax, [ebp+8]
.text:0000002E     push eax
.text:0000002F     call __sub
.text:00000034     add esp, 8
.text:00000037     mov esp, ebp
.text:00000039     pop ebp
.text:0000003A     retn
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12FFB0</td>
<td>char ** argv</td>
</tr>
<tr>
<td>0x02</td>
<td>int argc</td>
</tr>
<tr>
<td>Addr after “call _main”</td>
<td></td>
</tr>
<tr>
<td>0x0012FF50</td>
<td>saved ebp</td>
</tr>
<tr>
<td>0x100</td>
<td>int a</td>
</tr>
<tr>
<td>0x0012FF1C</td>
<td>undef</td>
</tr>
<tr>
<td>0x0012FF18</td>
<td>undef</td>
</tr>
<tr>
<td>0x0012FF14</td>
<td>undef</td>
</tr>
<tr>
<td>0x0012FF10</td>
<td>undef</td>
</tr>
<tr>
<td>0x0012FF0C</td>
<td>undef</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x104</td>
</tr>
<tr>
<td>ecx</td>
<td>0x100</td>
</tr>
<tr>
<td>edx</td>
<td>0x100</td>
</tr>
<tr>
<td>ebp</td>
<td>0x0012FF24</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF20</td>
</tr>
</tbody>
</table>

106
Example 2 - 16

```assembly
.text:00000000  _sub:    push    ebp
.text:00000001  mov     ebp, esp
.text:00000003  mov     eax, [ebp+8]
.text:00000006  mov     ecx, [ebp+0Ch]
.text:00000009  lea     eax, [ecx+eax*2]
.text:0000000C  pop     ebp
.text:0000000D  retn
.text:00000010  _main:  push    ebp
.text:00000011  mov     ebp, esp
.text:00000013  push    ecx
.text:00000014  mov     eax, [ebp+0Ch]
.text:00000017  mov     ecx, [eax+4]
.text:0000001A  push    ecx
.text:0000001B  call    dword ptr ds:__imp__atoi
.text:00000021  add     esp, 4
.text:00000024  mov     [ebp-4], eax
.text:00000027  mov     edx, [ebp-4]
.text:0000002A  push    edx
.text:0000002B  mov     eax, [ebp+8]
.text:0000002E  push    eax
.text:0000002F  call    _sub
.text:00000034  add     esp, 8
.text:00000037  mov     esp, ebp
.text:00000039  pop     ebp
.text:0000003A  retn
```

Table:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012FF0C</td>
<td>0x11FFB0 (char ** argv)</td>
</tr>
<tr>
<td>0x0012FF2C</td>
<td>0x2 (int argc)</td>
</tr>
<tr>
<td>0x0012FF28</td>
<td>Addr after &quot;call _main&quot;</td>
</tr>
<tr>
<td>0x0012FF24</td>
<td>0x0012FF50 (saved ebp)</td>
</tr>
<tr>
<td>0x0012FF20</td>
<td>undefined</td>
</tr>
<tr>
<td>0x0012FF1C</td>
<td>undefined</td>
</tr>
<tr>
<td>0x0012FF18</td>
<td>undefined</td>
</tr>
<tr>
<td>0x0012FF14</td>
<td>undefined</td>
</tr>
<tr>
<td>0x0012FF10</td>
<td>undefined</td>
</tr>
<tr>
<td>0x0012FF0C</td>
<td>undefined</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>$0x104</td>
</tr>
<tr>
<td>ecx</td>
<td>$0x100</td>
</tr>
<tr>
<td>edx</td>
<td>$0x100</td>
</tr>
<tr>
<td>ebp</td>
<td>0x0012FF24</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF24</td>
</tr>
</tbody>
</table>

107
Example2 - 17

.text:00000000 _sub:    push ebp
.text:00000001           mov ebp, esp
.text:00000003           mov eax, [ebp+8]
.text:00000006           mov ecx, [ebp+0Ch]
.text:00000009           lea eax, [ecx+eax*2]
.text:0000000C           pop ebp
.text:0000000D           ret

.text:00000010 _main:    push ebp
.text:00000011           mov ebp, esp
.text:00000013           push ecx
.text:00000014           mov eax, [ebp+0Ch]
.text:00000017           mov ecx, [eax+4]
.text:0000001A           mov ecx
.text:0000001B           call dword ptr ds:__imp__atoi
.text:00000021           add esp, 4
.text:00000024           mov [ebp-4], eax
.text:00000027           mov edx, [ebp-4]
.text:0000002A           push edx
.text:0000002B           mov eax, [ebp-8]
.text:0000002E           push eax
.text:0000003F           call _sub
.text:00000034           add esp, 8
.text:00000037           mov esp, ebp
.text:00000039           pop ebp
.text:0000003A           ret
Going through Example2.c in Visual Studio

sub:
    push    ebp
    mov     ebp,esp
    mov     eax,0BEEFh
    pop     ebp
    ret

main:
    push    ebp
    mov     ebp,esp
    call    sub
    mov     eax,0F00Dh
    pop     ebp
    ret
Changing active project
Setting command line arguments

Command Arguments
The command line arguments to pass to the application.
Instructions we now know (9)

• NOP
• PUSH/POP
• CALL/RET
• MOV
• LEA
• ADD/SUB
Back to Hello World

.text:00401730 main
.text:00401730 push ebp
.text:00401731 mov ebp, esp
.text:00401733 push offset aHelloWorld ; "Hello world\n"
.text:00401738 call ds:_imp__printf
.text:0040173E add esp, 4
.text:00401741 mov eax, 1234h
.text:00401746 pop ebp
.text:00401747 retn

Are we all comfortable with this now?

Windows Visual C++ 2005, /GS (buffer overflow protection) option turned off
Disassembled with IDA Pro 4.9 Free Version