Intro to Assembly. x86

@Its_EZBB

Agenda

- Refresher
- Calling Convention
- Register
- Addresses r/m32
- Instructions
- Control Flow
- Arguments
- Size
- Program

Refresher

- Data types
- C
- High vs Low
  - Top half of the memory
  - Lower half of the memory

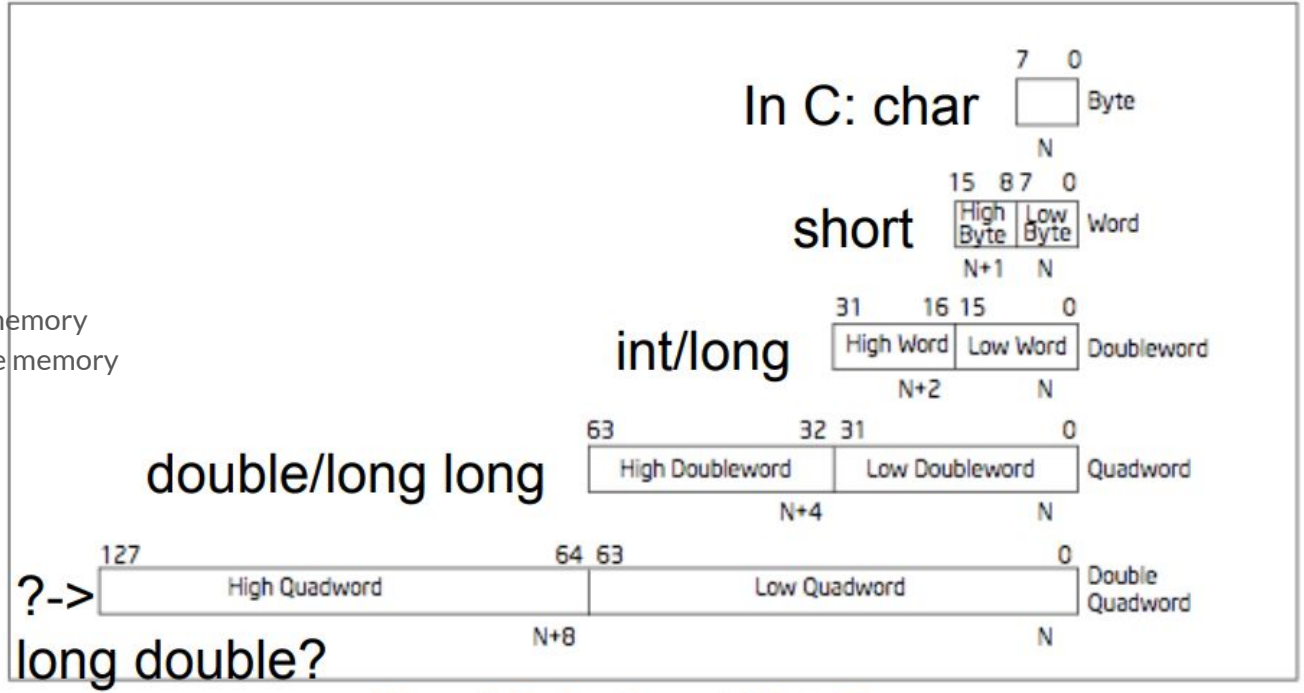


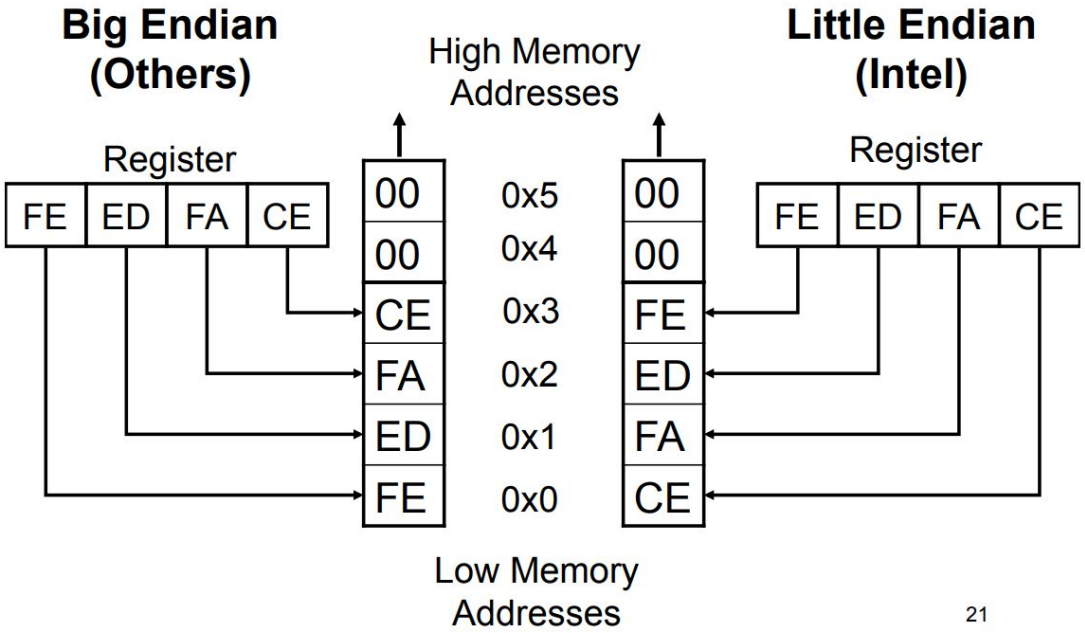
Figure 4-1. Fundamental Data Types

Refresher

- Hex
- Negative number
  - 2's complement
    - Flip all bit
    - Then add 1
  - Most significant bit - left most
    - 1 = negative
    - 0 = positive

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

Refresher - Endian



Calling Convention

- How code call a functions:
  - There are a few routine.
- Caller - Function that made the call
- Callee - Function that is being called

Calling Convention

- Calling conventions:
  - cdecl
    - Caller set up the stack - parameters
    - Then call the function
    - **Caller clean up the stack afterwards.**
  - Stdcall
    - Caller set up the stack - parameters
    - Then call the function
    - **Callee clean up the stack afterwards.**

Register

- Register?
  - A very fast memory used to store something
- Typical store useful data while the program is running
  - EX:
    - Stack pointers
    - Variables
    - Counters
    - Return value or address

Registers

- There are 16 registers - conventions
  - EAX - Store return value
  - EIP - Store address of next instruction.
  - ESP - Stack pointer
  - EBP - Store stack frame base pointer
  - ESI - Pointer for string operations
  - EDI - Destination pointer for string operations
  - ECX - Counter for string and loop operations
  - .....

Note: This DOES NOT mean programs FOLLOW it. Look at the code and see what value does it holds.

Register Convention

- Caller saved registers
  - Saved by the caller function
    - EAX, ECX, EDX
    - EAX usually get modified by callee
      - Aka getting the return value
- Callee saved registers
  - Saved by the callee function.
    - EBP, EBX, ESI, EDI

****Basically, if either the caller have something in the register that they care about, save it before or call. While the callee, not to change the registers that callers didn't save, unless it saves and restores it later.

Addressing - r/m32 form

- Intel - You will be the “[“ and “]” brackets
  - => Treat the value as a **memory address**.
  - => And fetch the value at that **address.**

- 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)

Look at the value located at ebx instead of ebx

- Most complicated form is: `[base + index*scale + disp]`

Instructions

We will mostly look at this for the slide!!!

In x86, there are 2 syntaxes:

- Intel
  - int source , destination
- AT&T
  - int %destination , %source

Example:

Intel Syntax	AT&T Syntax
`mov al,bl`	`movb %bl,%al`
`mov ax,bx`	`movw %bx,%ax`
`mov eax,ebx`	`movl %ebx,%eax`
`mov eax, dword ptr [ebx]`	`movl (%ebx),%eax`

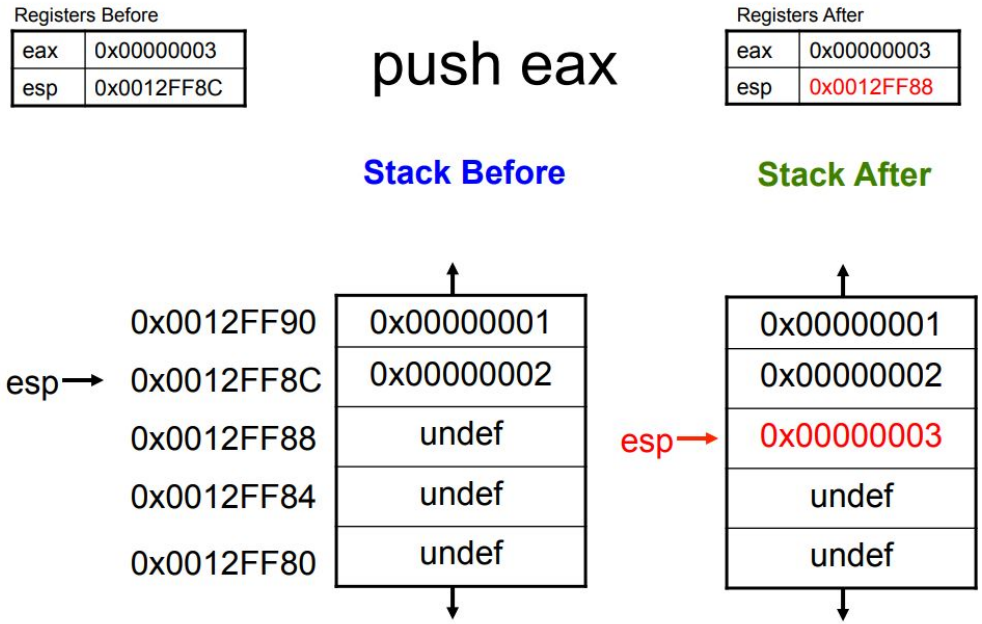
Instructions

- **NOP**
    - Do nothing
        - No operation
        - No registers
        - No values
    - There to pad and align bytes/ OR delay time
    - EXPLOIT!!! :)
        - You will see this a lot for alignment.

Note: Underneath, it just exchange eax with itself.

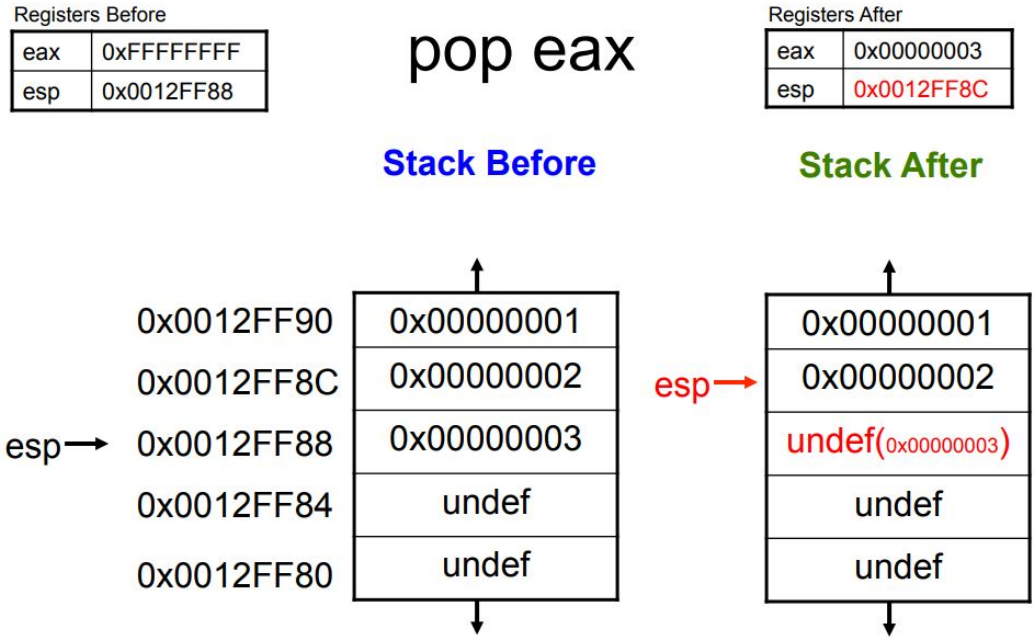
Instructions

- **PUSH**
  - Push value onto the stack
  - Decrements esp by 4
    - Move down = stack grow



Instructions

- POP
  - Take value off stack
  - Increment esp by 4
    - Move up = stack shrinks



Instructions

- **MOV**
  - Move
  - Can move:
    - Register to register
    - Memory to register; register to memory
    - Immediate to register; immediate to memory
  - **NEVER** memory to memory

Examples

`mov eax, ebx` — copy the value in ebx into eax

`mov byte ptr [var], 5` — store the value 5 into the byte at location var

Instructions

- **LEA**
  - Load effective address
  - Place the **ADDRESS** into specified register.
  - **MATH**

Example: ebx = 0x2, edx = 0x1000

– lea eax, [edx+ebx*2]

– eax = 0x1004, not the value at 0x1004

Examples

`lea edi, [ebx+4*esi]` — the quantity EBX+4*ESI is placed in EDI.

`lea eax, [var]` — the value in *var* is placed in EAX.

`lea eax, [val]` — the value *val* is placed in EAX.

Instructions

- **ADD** and **SUB**
  - Add and Subtracts.
  - Destination
    - Can be address or register
  - Source
    - Can be address or register or immediate
  - Source AND destination **CANNOT** be addresses.
    - No memory to memory

- add esp, 8
- sub eax, [ebx*2]

Instructions

- **CALL**
  - Transfer control to a different function
  - Steps:
    - Push address of next instruction to stack
      - So it knows where to go back
      - After executing the new function
    - Change EIP to address of new function

Instructions

- **RET**
  - Return
  - Go back to the instruction that **CALL** save on stack
  - There are 2 forms
    - Depends on what calling convention is using.
    - cdecl
      - Pop top of stack -> eip
      - Written as “ret”
    - stdcall
      - Pop top of stack -> eip, **then** add a constant of bytes to esp
      - Written as “ret 0x12”

Instructions

- **CMP**
  - Compare 2 operands
  - Set appropriate flags
    - Hold single bit - 0/1

Notes: Flags can also be set in arithmetic instructions such as Add, sub, ...

Table 1 - Common Flags

Symbol	Bit	Name	Set if...
CF	0	Carry	Operation generated a carry or borrow
PF	2	Parity	Last byte has even number of 1's, else 0
AF	4	Adjust	Denotes Binary Coded Decimal in-byte carry
ZF	6	Zero	Result was 0
SF	7	Sign	Most significant bit of result is 1
OF	11	Overflow	Overflow on signed operation
DF	10	Direction	Direction string instructions operate (increment or decrement)
ID	21	Identification	Changeability denotes presence of CPUID instruction

Instructions

- **JMP**
  - Jump to an address
  - Jump there NO MATTER WHAT
  - UNCONDITIONAL JUMP

Instructions ****

- CONDITIONAL JUMPS
  - Jump base on conditions
    - Jump if equal
    - Jump if greater
    - Jump if smaller
    - Etc…
  - How?
    - Check flags

```
CMP ebx,10
JLE there
```

http://unixwiz.net/techtips/x86-jumps.html

Instruction	Description	signed-ness	Flags	short jump opcodes	near jump opcodes
JO	Jump if overflow		OF = 1	70	0F 80
JNO	Jump if not overflow		OF = 0	71	0F 81
JS	Jump if sign		SF = 1	78	0F 88
JNS	Jump if not sign		SF = 0	79	0F 89
JE JZ	Jump if equal Jump if zero		ZF = 1	74	0F 84
JNE JNZ	Jump if not equal Jump if not zero		ZF = 0	75	0F 85
JB JNAE JC	Jump if below Jump if not above or equal Jump if carry	unsigned	CF = 1	72	0F 82
JNB JAE JNC	Jump if not below Jump if above or equal Jump if not carry	unsigned	CF = 0	73	0F 83
JBE JNA	Jump if below or equal Jump if not above	unsigned	CF = 1 or ZF = 1	76	0F 86
JA JNBE	Jump if above Jump if not below or equal	unsigned	CF = 0 and ZF = 0	77	0F 87
JL JNGE	Jump if less Jump if not greater or equal	signed	SF <> OF	7C	0F 8C
JGE JNL	Jump if greater or equal Jump if not less	signed	SF = OF	7D	0F 8D
JLE JNG	Jump if less or equal Jump if not greater	signed	ZF = 1 or SF <> OF	7E	0F 8E
JG JNLE	Jump if greater Jump if not less or equal	signed	ZF = 0 and SF = OF	7F	0F 8F
JP JPE	Jump if parity Jump if parity even		PF = 1	7A	0F 8A
JNP JPO	Jump if not parity Jump if parity odd		PF = 0	7B	0F 8B
JCXZ JECXZ	Jump if %CX register is 0 Jump if %ECX register is 0		%CX = 0 %ECX = 0	E3	

Control Flow

- Instructions such as
  - JMP
  - CALL
  - RET
  - CONDITIONAL JUMPS
- Will change the program flow.

Arguments

- Functions takes arguments
  - Where does x86 save the arguments?
    - Registers
      - edi, esi, edx, ecx, r8, r9
      - (1 , 2 , 3 , 4 , 5 , 6)
    - Stack
      - Right to left = 1st to last saved
      - Will see in a bit!

Size

- In x86, it use things such as:
  - BYTE PTR - 1 byte
  - WORD PTR - 2 bytes
  - DWORD PTR -4 bytes

For example:

```
mov BYTE PTR [ebx], 2      ; Move 2 into the single byte at the address stored in EBX.
mov WORD PTR [ebx], 2      ; Move the 16-bit integer representation of 2 into the 2 bytes starting at the address in EBX.
mov DWORD PTR [ebx], 2     ; Move the 32-bit integer representation of 2 into the 4 bytes starting at the address in EBX.
```

Program!!!

- Let’s look at a program

Example2.c with Input parameters and Local Variables

```c
#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);
}
```

```
.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
```

Example2.c - 1

```
.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	0xcafe ⌘
ecx	0xbabe ⌘
edx	0xfeed ⌘
ebp	0x0012FF50 ⌘
esp	0x0012FF24 ♍

0x0012FF30	0x12FFB0 (char ** argv)⌘
0x0012FF2C	0x2 (int argc) ⌘
0x0012FF28	Addr after "call _main" ⌘
0x0012FF24	0x0012FF50(saved ebp)♍
0x0012FF20	undef
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

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: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	0xcafe
ecx	0xbabe
edx	0xfeed
ebp	0x0012FF24 ♍
esp	0x0012FF24

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	undef
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2.c - 3

```
.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 ☒
                         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
```

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.

eax	0xcafe
ecx	0xbabe
edx	0xfeed
ebp	0x0012FF24
esp	0x0012FF20 ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0xbabe (int a) ♍
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2.c - 4

```
.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]
                        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
.text:0000002F          call   _sub
.text:00000034          add    esp, 8
.text:00000037          mov    esp, ebp
.text:00000039          pop    ebp
.text:0000003A          retn
```

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

eax	0x12FFB0
ecx	0xbabe
edx	0xfeed
ebp	0x0012FF24
esp	0x0012FF20

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0xbabe (int a)
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 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]
                         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
```

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

eax	0x12FFB0
ecx	0x12FFD4 (arbitrary)
edx	0xfeed
ebp	0x0012FF24
esp	0x0012FF20

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after "call _main"
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0xbabe (int a)
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 6

```
.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]
                       pop    ebp
                       retn
                       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
```

Saving some slides…
This will push the address of the string at argv[1] (0x12FFD4). atoi() will read the string and turn in 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	0x100♍ (arbitrary⌘)
ecx	0x12FFD4
edx	0xfeed
ebp	0x0012FF24
esp	0x0012FF20

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after "call _main"
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0xbabe (int a)
0x0012FF1C	undef ♍
0x0012FF18	undef ♍
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 7

```
.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
                         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
.text:0000003A           retn
```

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".

eax	0x100
ecx	0x12FFD4
edx	0x100 ♍
ebp	0x0012FF24
esp	0x0012FF1C ♍

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	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 8

```
.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
                        mov    eax, [ebp+8] ☒
                        push   eax ☒
                        call   _sub
                        add    esp, 8
                        mov    esp, ebp
                        pop    ebp
.text:0000003A          retn
```

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

eax	0x2 ♍
ecx	0x12FFD4
edx	0x100
ebp	0x0012FF24
esp	0x0012FF18 ♍

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	undef
0x0012FF10	undef
0x0012FF0C	undef

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	0x00000034 ♍
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          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	0x0012FF10 ♍
esp	0x0012FF10 ♍

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	0x00000034
0x0012FF10	0x0012FF24(saved ebp)♍
0x0012FF0C	undef

Example2 - 11

```
.text:00000000 _sub:    push   ebp
.text:00000001          mov    ebp, esp
                        mov    eax, [ebp+8] ☒
                        mov    ecx, [ebp+0Ch] ☒
                        lea    eax, [ecx+eax*2]
                        pop    ebp
                        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
```

Move “x” into eax, and “y” into ecx.

eax	0x2 ♍ (no value change)
ecx	0x100 ♍
edx	0x100
ebp	0x0012FF10
esp	0x0012FF10

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	0x00000034
0x0012FF10	0x0012FF24 (saved ebp)
0x0012FF0C	undef

Example2 - 12

```
.text:00000000 _sub:   push   ebp
.text:00000001         mov    ebp, esp
.text:00000003         mov    eax, [ebp+8]
                       mov    ecx, [ebp+0Ch]
                       lea    eax, [ecx+eax*2]
                       pop    ebp
                       retn
                       push   ebp
                       mov    ebp, esp
                       push   ecx
                       mov    eax, [ebp+0Ch]
                       mov    ecx, [eax+4]
                       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
```

Set the return value (eax) to 2*x + y. Note: neither pointer arith, nor an "address" which was loaded. Just an afficient way to do a calculation.

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

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	0x00000034
0x0012FF10	0x0012FF24 (saved ebp)
0x0012FF0C	undef

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+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	0x104
ecx	0x100
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	0x00000034
0x0012FF10	undef ♍
0x0012FF0C	undef

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+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	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF24
esp	0x0012FF18 ♍

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	undef ♍
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 15

```
.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	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF24
esp	0x0012FF20 ♍

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

Example2 - 16

```
.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	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF24
esp	0x0012FF24 ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	undef ♍
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

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           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	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF50 ♍
esp	0x0012FF28 ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	undef ♍
0x0012FF20	undef
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

That’s it, thanks folks!!!!