

(Re)introduction to x86 Assembly

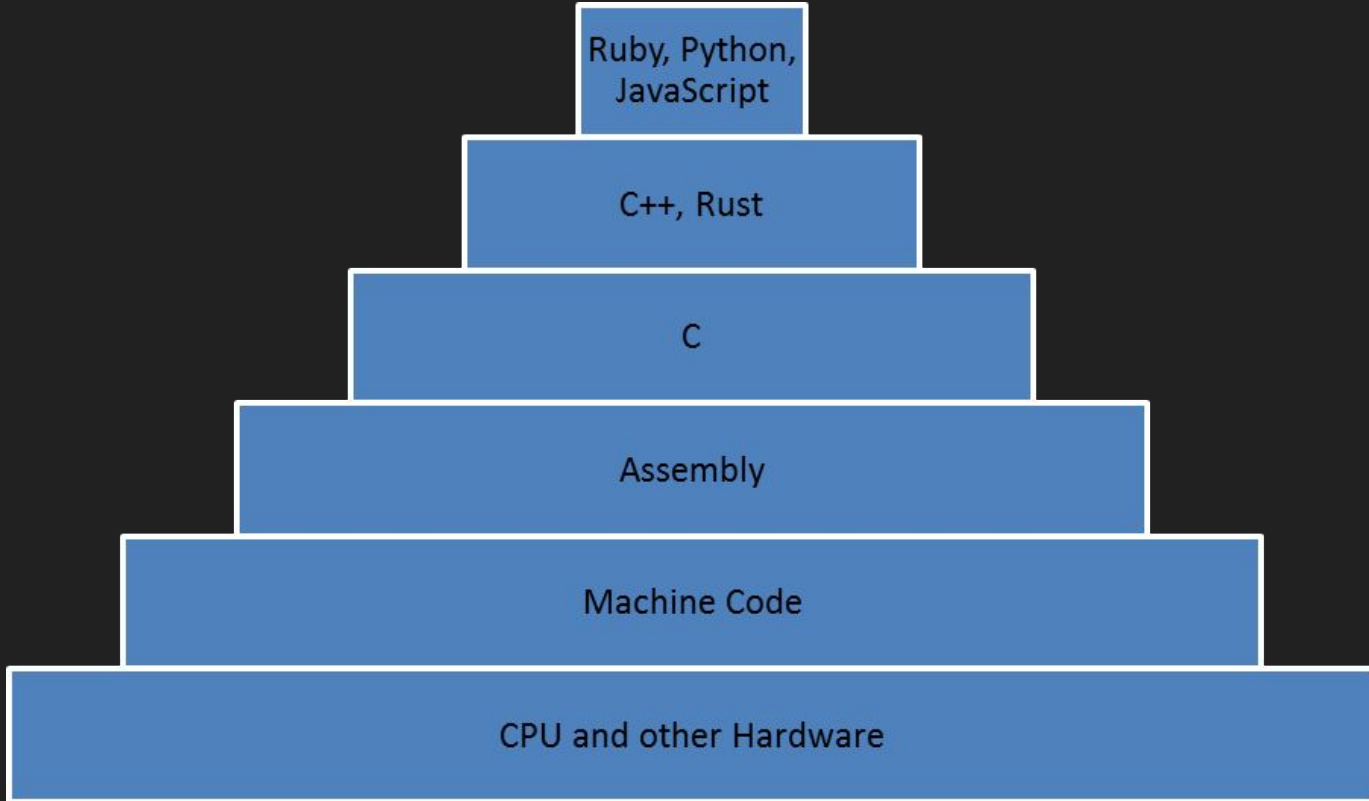
MST - 1/26

What is Assembly?

What and why

- Lowest level language we're concerned about directly writing
- Translates almost 1:1 into machine code for the CPU
- We focus on 32-bit **x86 assembly** in this class, common on Intel and AMD processors
 - Many other variants including ARM, RISC, AVR, etc
- Understanding assembly will help you learn what your computer is really doing
- A program we're trying to exploit may only be available as assembly
- Initial learning curve can be difficult, but stick with it

What is Assembly?



Basic Syntax

x86 Key Concepts

- Understanding assembly is crucial to low-level exploitation
- **Instructions** are the main unit of code
- May take multiple instructions to perform one higher-level “operation”
- Many instructions use **registers**, which are built-in named temporary variables
- Two main forms: **Intel** and **AT&T** syntax

Intel: `mov eax, 12345`

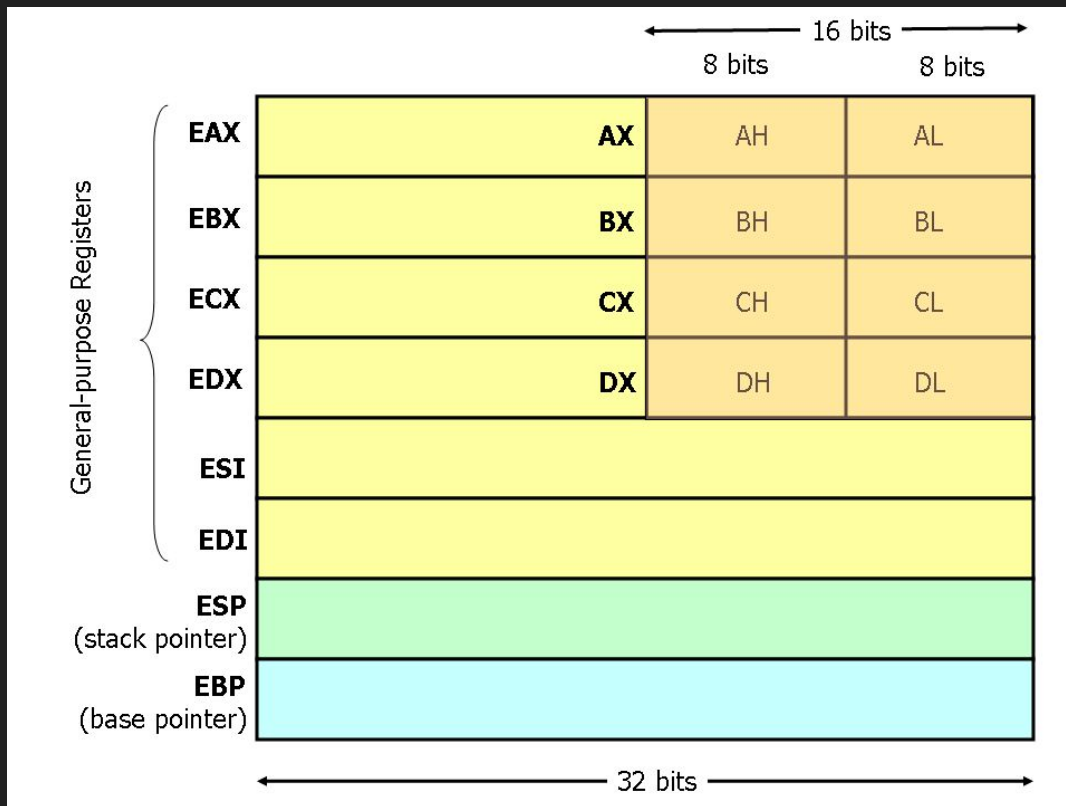
AT&T: `movl $12345, %eax`

```
000000000040057d <main>:
55                               push    rbp
48 89 e5                         mov     rbp, rsp
48 83 ec 10                       sub     rsp, 0x10
bf 44 06 40 00                   mov     edi, 0x400644
e8 c1 fe ff ff                   call    400450 <puts@plt>
c7 45 fc 05 00 00 00             mov     DWORD PTR [rbp-0x4], 0x5
83 45 fc 01                       add     DWORD PTR [rbp-0x4], 0x1
8b 45 fc                         mov     eax, DWORD PTR [rbp-0x4]
89 c6                             mov     esi, eax
bf 50 06 40 00                   mov     edi, 0x400650
b8 00 00 00 00                   mov     eax, 0x0
e8 b2 fe ff ff                   call    400460 <printf@plt>
b8 00 00 00 00                   mov     eax, 0x0
c9                               leave   %eax
c3                               ret
```

```
$ gcc hello.c
$ ./a.out
Hello World
6
```

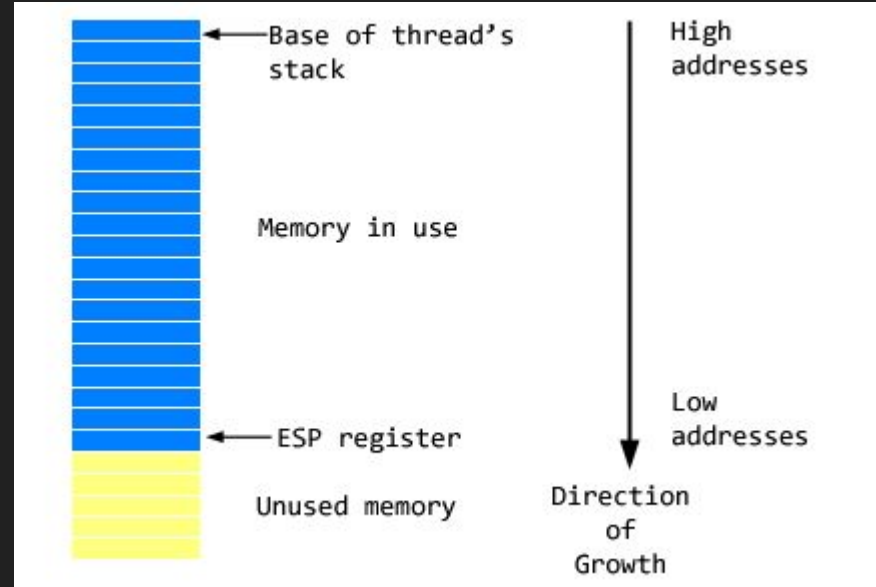
Registers

- **EAX, EBX, ECX, EDX**
 - Common general purpose registers
- **ESP**
 - Points to the “top” of the current stack frame
- **EBP**
 - Stack base pointer, points to the “bottom” of the current stack frame
- **EIP**
 - Points to the location of the current instruction in memory
- **EFLAGS**
 - Contains **flag bits** (zero flag, carry flag, sign flag, etc)



The Stack

- LIFO data structure used for local variables and function calls
- A **stack frame** is the portion of the stack used by the current function
- **ESP** stores the lowest address, which is the top of the stack
- **EBP** stores the highest address, which is the bottom of the current stack frame

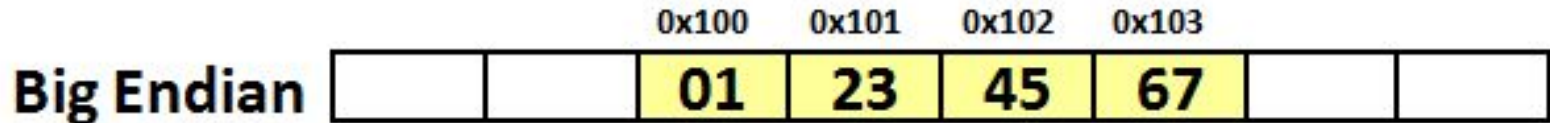


Side note: Hexadecimal and Binary

- Both hex and binary are widely used in assembly and low-level exploitation
- Addresses are almost always expressed in hexadecimal
 - 134524739 = 0x0804af43
- Useful to know ASCII representation of common characters in hex as well
 - 'A' = 0x41, 'B' = 0x42,
- Understanding binary operations quickly can also be helpful
 - AND, OR, XOR, NOT, etc

Side note: Little Endian vs Big Endian

- x86 is a **little endian** architecture, which refers to the order of bytes in memory for multi-byte data
- This matters only if you access specific bytes of a larger data type - otherwise, it will be transparently handled for you



Common Instructions

Arithmetic Instructions

- **add** `eax, 0x123`
 - `eax = eax + 0x123`
- **sub** `eax, 0x456`
 - `eax = eax - 0x456`
- **and** `eax, ebx`
 - `eax = eax & ebx`
- **not** `ecx`
 - `ecx = ~ecx`
- **inc** `edx`
 - `edx++`
- **shl** `ecx, al`
 - `ecx = ecx << al`
- **mul** `ecx`
 - `(edx:eax) = ecx * eax`
- **div** `ecx`
 - `eax = (edx:eax) / ecx`
 - `edx = (edx:eax) % ecx`

MOV Instructions

- `mov eax, ebx`
 - `eax = ebx`
- `mov eax, 123`
 - `eax = 123`
- `mov eax, DWORD PTR [0x123456]`
 - `eax = *(0x123456)`
- `mov eax, DWORD PTR [edx+esi*4]`
 - `eax = *(edx + esi * 4)`
- `lea esi, [ebp - 16]`
 - `esi = ebp - 16`
 - Commonly used for pointer manipulations

Addressing Modes

`[ebx]`

`[ebx - 4]`

`[ebx + eax]`

`[ebx + eax*4]`

`[ebx-eax]` ; Invalid

(Can only add registers)

`[ebx + eax + ecx]` ; Invalid

(Can only use 2 registers)

Size Directives

BYTE PTR `[edx]`

WORD PTR `[edx]`

DWORD PTR `[edx]`

Control Flow Instructions

- `jmp eax`
 - Unconditional jump
- `jz $my_location`
 - Jump if zero flag set
- `jnz $my_location`
 - Jump if zero flag not set
- `jg $my_location`
 - Jump if greater
- `jb $my_location`
 - Jump if below

EFLAGS register

- Stores bit flags to indicate the results of operations
 - **Carry Flag**
 - **Zero Flag**
 - **Sign Flag**
- Implicitly set after certain instructions

```
mov eax, 5
```

```
cmp eax, 4
```

```
jg $cool_place
```

```
mov eax, 5
```

```
cmp eax, 5
```

```
jz $also_cool
```

PUSH, POP, CALL, RET

- Mainly used for calling functions and saving registers
- `push ebp`
 - Decrements `ESP` and puts `EBP` on the top of the stack
- `pop ebp`
 - Loads the top of the stack into `EBP` and increments `ESP`
- `call 400460 <printf@plt>`
 - Pushes the address of the next instruction to the stack and jumps to `400460`
- `ret`
 - Pulls the top of the stack into `EIP` and continues execution

Calling Conventions

- When compilers generate x86 from C, they use the **cdecl** calling convention
- cdecl specifies what to do with registers and the stack
 - Which registers have to be saved before calling a function?
 - How are arguments passed?
 - How do we handle saving and updating **EBP** and **ESP**?
 - Which function should handle cleanup?
- Since all relevant C compilers for x86 use cdecl, their code can interoperate

cdecl

- Function arguments are pushed to the stack in reverse order before the call
- **EBP** is set to the top of the stack at the beginning of the function
- **EBP** is restored at the bottom of a function
- Caller is responsible for
 - Cleaning up the stack after a call
 - Saving **EAX**, **ECX**, and **EDX** registers
- Callee is responsible for
 - Putting the return value in **EAX**
 - Preserving all other registers

callee(1, 2, 3);

caller:

```
. . .  
; push call arguments  
push    3  
push    2  
push    1  
; call subroutine 'callee'  
call    callee  
; remove arguments from frame  
add     esp, 12  
; use subroutine result  
cmp     eax, 6  
. . .
```

callee:

```
; make new call frame  
push    ebp  
mov     ebp, esp  
; use arguments to set result  
mov     eax, [ebp+0x8]  
mov     edx, [ebp+0xc]  
mov     ecx, [ebp+0x10]  
add     eax, edx  
add     eax, ecx  
; restore old call frame  
pop     ebp  
; return to caller  
ret
```

Calling Conventions

- You **don't** have to use `cdecl` when writing in assembly...
- ...but you won't be able to call 32-bit C library functions!
- Also, it's good to get practice for later when we are reading compiler-generated assembly

Basic x86 example

What does this code do?

my_function:

push ebp

mov ebp, esp

mov eax, [ebp+0x8]

mov edx, [ebp+0xc]

body:

mov ecx, [eax]

cmp ecx, edx

jz found

cmp ecx, 0

jz notfound

inc eax

jmp body

found:

mov ecx, [ebp+0x4]

sub eax, ecx

jmp done

notfound:

xor eax, eax

done:

pop ebp

ret

What does this code do?

```
my_function:
    push ebp ; function prologue
    mov ebp, esp
    mov eax, [ebp+0x8]
    mov edx, [ebp+0xc]
body: ; loop body
    mov ecx, [eax]
    cmp ecx, edx ; what is ecx? edx?
    jz found
    cmp ecx, 0
    jz notfound
    inc eax
    jmp body
```

```
found: ; success case
    mov ecx, [ebp+0x4]
    sub eax, ecx
    jmp done
notfound: ; error case
    xor eax, eax
done: ; function epilogue
    pop ebp
    ret
```

x86 -> C

```
my_function:
    push ebp
    mov ebp, esp
    mov eax, [ebp+0x8]
    mov edx, [ebp+0xc]
body:
    mov ecx, [eax]
    cmp ecx, edx
    jz found
    cmp ecx, 0
    jz notfound
    inc eax
    jmp body
( rest omitted )
```

```
int my_function(char *d, char ch)
{
    int x = 0;
    while (d[x] != 0) {
        if (d[x] == ch)
            return x;
        x++;
    }
    return 0;
}
```

Creating Executables

Using GCC as an assembler

```
$ gcc -m32 -o simple simple.s
```

```
$ ./simple
```

```
$ echo $?
1
```

```
.intel_syntax noprefix
```

```
.globl main
```

```
main:
```

```
mov eax, 1
```

```
ret
```

Calling C functions

```
$ gcc -m32 -o hello hello.s
```

```
$ ./hello
```

```
Hello World!
```

```
.intel_syntax noprefix
```

```
.data
```

```
message:
```

```
.string "Hello World!"
```

```
.text
```

```
.globl main
```

```
main:
```

```
mov ecx, offset message
```

```
push ecx
```

```
call puts
```

```
pop ecx
```

```
mov eax, 0
```

```
ret
```

Print an argument

```
$ gcc -m32 -o arg arg.s
```

```
$ ./arg ayyy
```

```
ayyy
```

```
.intel_syntax noprefix
```

```
.text
```

```
.globl main
```

```
main:
```

```
push ebp
```

```
mov ebp, esp
```

```
mov ecx, [ebp+0xc]
```

```
mov ecx, [ecx+0x4]
```

```
push ecx
```

```
call puts
```

```
pop ecx
```

```
mov eax, 0
```

```
pop ebp
```

```
ret
```

Homework

Homework: Print out a file backwards

- Write a script that accepts a filename, reads in the file, and prints it out backwards

```
$ cat hello_world.txt
Hello World!
$ ./your_program hello_world.txt
!dlrow olleH
```

- Write code only in 32-bit x86 assembly, but no other restrictions
- Calling C functions is allowed
 - But you'll get extra credit if you don't use any!

Homework grading

- Submit your assembly code file to cm7bv@virginia.edu with the subject “MST Assignment 1 - <YOUR_UVA_ID>”
 - eg: “MST Assignment 1 - cm7bv”
- Also, include a brief (1-paragraph) description of what you did and how it went

Homework tips

- You will need to use dynamic memory management (with malloc/free) handle command line arguments
- Don't overthink it or worry about performance. Only consider correctness
- Become familiar with [man pages](#), they are invaluable
- Syscalls may be easier than the C standard library in many cases
- Don't be afraid to ask questions either via Slack or email!

Good Resources

Resources

https://en.wikibooks.org/wiki/X86_Assembly

<https://www.nayuki.io/page/a-fundamental-introduction-to-x86-assembly-programming> (Uses AT&T syntax)

<https://www.cs.virginia.edu/~evans/cs216/guides/x86.html>

General bash and terminal practice: <http://overthewire.org/wargames/bandit/>

x86 system calls (pay attention to 32 vs 64 bit):

https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux