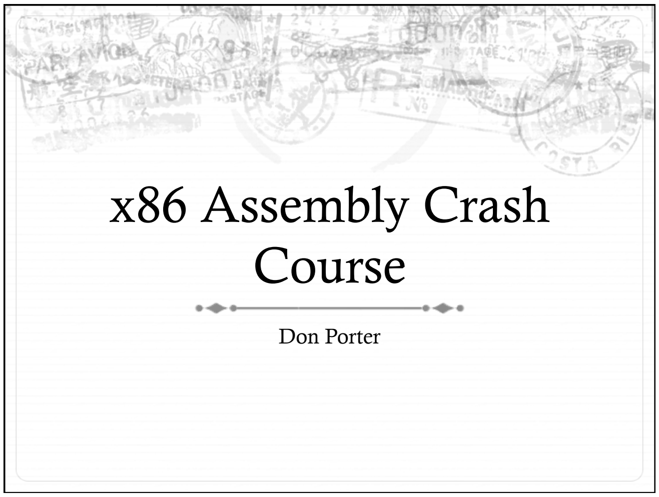
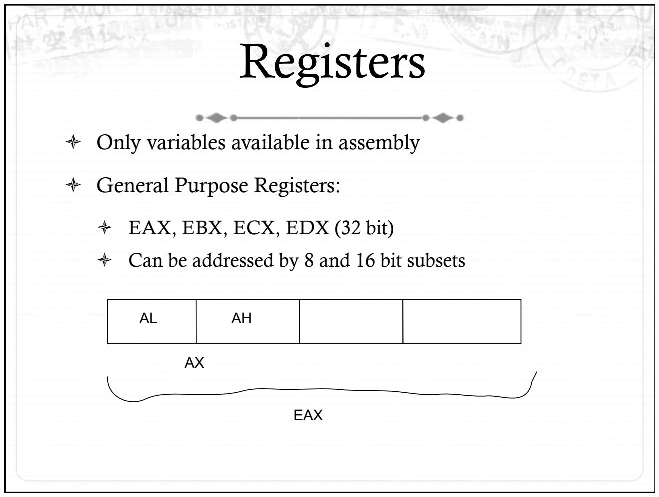


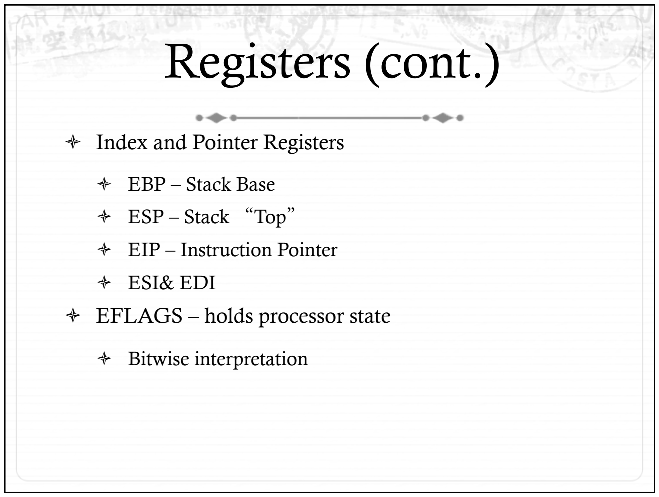
x86 Assembly Crash Course

Don Porter



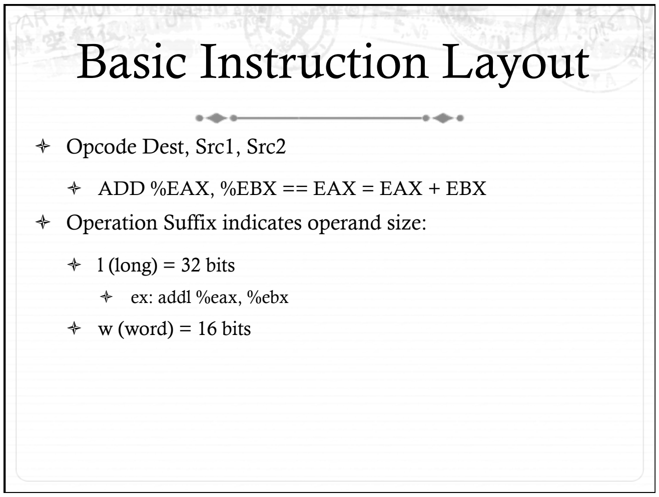
Registers

- Only variables available in assembly
- General Purpose Registers:
  - EAX, EBX, ECX, EDX (32 bit)
  - Can be addressed by 8 and 16 bit subsets



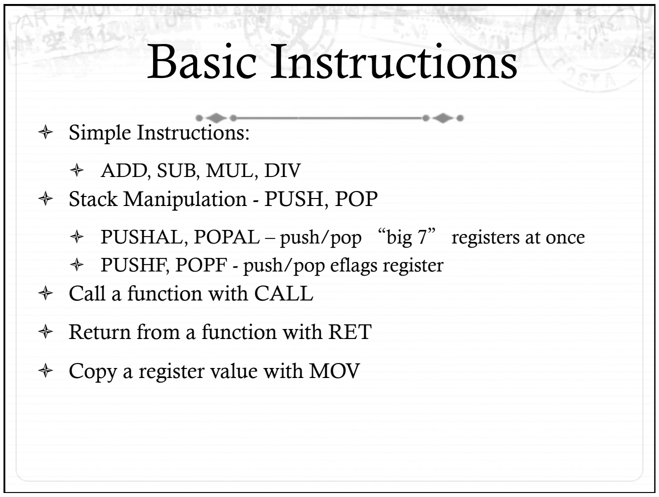
Registers (cont.)

- Index and Pointer Registers
  - EBP – Stack Base
  - ESP – Stack "Top"
  - EIP – Instruction Pointer
  - ESI& EDI
- EFLAGS – holds processor state
  - Bitwise interpretation



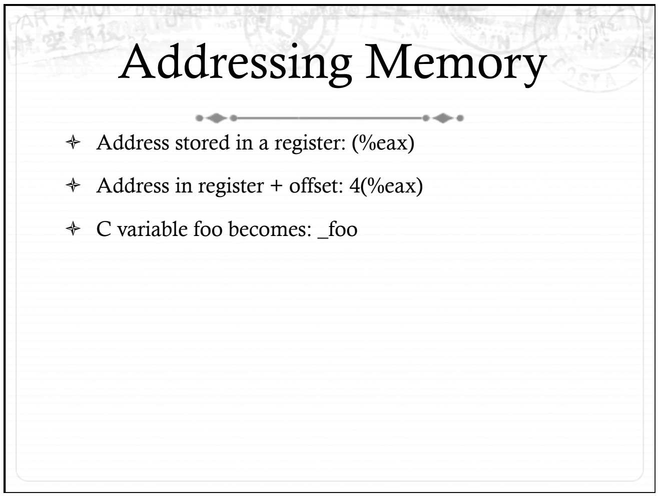
Basic Instruction Layout

- Opcode Dest, Src1, Src2
  - ADD %EAX, %EBX == EAX = EAX + EBX
- Operation Suffix indicates operand size:
  - l (long) = 32 bits
    - ex: addl %eax, %ebx
  - w (word) = 16 bits



Basic Instructions

- Simple Instructions:
  - ADD, SUB, MUL, DIV
- Stack Manipulation - PUSH, POP
  - PUSHAL, POPAL – push/pop "big 7" registers at once
  - PUSHF, POPF - push/pop eflags register
- Call a function with CALL
- Return from a function with RET
- Copy a register value with MOV



Addressing Memory

- Address stored in a register: (%eax)
- Address in register + offset: 4(%eax)
- C variable foo becomes: _foo

Next: Inline assembly

- But first, a bit of very helpful background on compilers

Detour: Compiler Intro

- Parse high-level source code
- Convert to intermediate form (often SSA)
  - Convert all variables into infinite, logical registers
- Optimize! Optimize! Optimize! (heavy thinking here)
- Map logical registers onto architectural registers
  - A.k.a. register assignment
- Emit machine code

Example (high-level lang)

```
x = 0;
y = x + 1;
// x = x * y
asm (“imul %eax, %ebx”: “=a”(x) : “a”(x), “b”(y));
y = y + x;
```

Example (Convert to pseudo-SSA)

```
x_0 = 0;
y_0 = x_0 + 1;
// x = x * y
asm (“imul %eax, %ebx”: “=a”(x_1) : “a”(x_0), “b”(y_0));
y_1 = y_0 + x_1;
```

Assembly treated as black box, except input/output params

Every assignment treated like a new variable

Example (Assign Registers)

```
x_0 = 0;
y_0 = x_0 + 1;
// x = x * y
asm (“imul %eax, %ebx”:
     “=a”(x_1) :
     “a”(x_0), “b”(y_0));
y_1 = y_0 + x_1;
```

```
%edx= 0;
%ecx = %edx + 1;
%eax = %edx; // “a”(x_0),
%ebx = %ecx // “b”(y_0)
“imul %eax, %ebx”
%edx = %ecx + %eax;
```

Reuse edx. No longer live

“=a”(x_1)

Key points

- Compiler treats your assembly code mostly as a black box
- You specify what input variables should be in which registers
  - Compiler adds code to move variables around as needed
- You specify what output variables are in which registers
  - Compiler factors this into register assignment after the assembly
- Note that parameters are copy-by-value
  - In the previous example, if you don’t specify an output back to x, the output will be ignored
  - Treated as x_1 vs. x_0

For completeness

- Compilers are really smart. Seriously.
- In reality, a register assignment phase would probably work backwards from input constraints on inline assembly
  - I didn't do this in the previous slide for the purposes of illustration
  - Not always possible to avoid moving registers around or saving values before inline assembly

Example (More Sophisticated)

```
x_0 = 0;

y_0 = x_0 + 1;

// x = x * y

asm (“imul %eax, %ebx”:

    “=a”(x_1) :

    “a”(x_0), “b”(y_0));

y_1 = y_0 + x_1;
```

```
%eax= 0; // “a”(x_0),

%ebx = %eax + 1;
  // “b”(y_0)

“imul %eax, %ebx”

%ecx = %ebx + %eax;
```

Inlined Assembly

```
… // c code

asm ( “assembly code” \

        output registers : \

        input registers :\

        clobbered registers );
```

Think of this as a separate function; inputs/outputs must be explicit

What is a clobbered register?

A Concrete Example

```
asm volatile ("movl %0, %%edx\r\n” \
              movl %1, %%ecx\r\n” \
              movl %2, %%ebx\r\n” \
              movl %3, %%eax\r\n” \
              xchg %%bx, %%bx \r\n” \
              : /*no output*/ \
              : "g"(addr), "g"(name), \
                "g"(len), "g"(105) \
              : "eax", "ebx", "ecx", "edx");
```

%0 – not a real register; compiler will slot in

g = Let the compiler assign the register

These registers will be trashed (but not input/output)

Clobbered Registers

- Suppose %edx is not an input or output parameter to your inline assembly
- The compiler may store some unrelated variable in this registers **before** your assembly, and then try to use it **after** the assembly
- Clobber registers tell the compiler to save this value (e.g., by pushing it on the stack), and restore it later if needed
  - Compiler does sophisticated liveness analysis to figure out whether this is necessary

A More Efficient Version

```
asm volatile (xchg %%bx, %%bx " \
              : /*no output*/ \
              : “d"(addr), “c"(name), \
                “b"(len), “a"(105) );
```

- Notice:
  - Clobber registers only needed if not in input/output
  - If we want arguments in specific registers, no need to move them/waste time bouncing between registers
  - If you don't care, good to give the compiler some options