Homework

- In-line Assembly Code
- Machine Language
- Program Efficiency Tricks
- Reading
  - PAL, pp 3-6, 361-367
  - Practice Exam 1

In-line Assembly Code

- The gcc compiler allows you to put assembly instructions in-line between your C statements
- This is a lot trickier way to integrate assembly code with C than writing C callable functions
- You need to know a lot more about how the compiler locates variables, uses registers, and manages the stack.
- Does execute faster than a function call to the equivalent amount of assembly code

In-line Assembly Code

- Example Function with in-line assembly code

```
int foobar(int x, int *y)
{
        int i, *j;
        asm("pushl %eax");
        i = x;
        j = &i;
        *y = *j;
        asm("popl %eax");
        return 0;
}
```

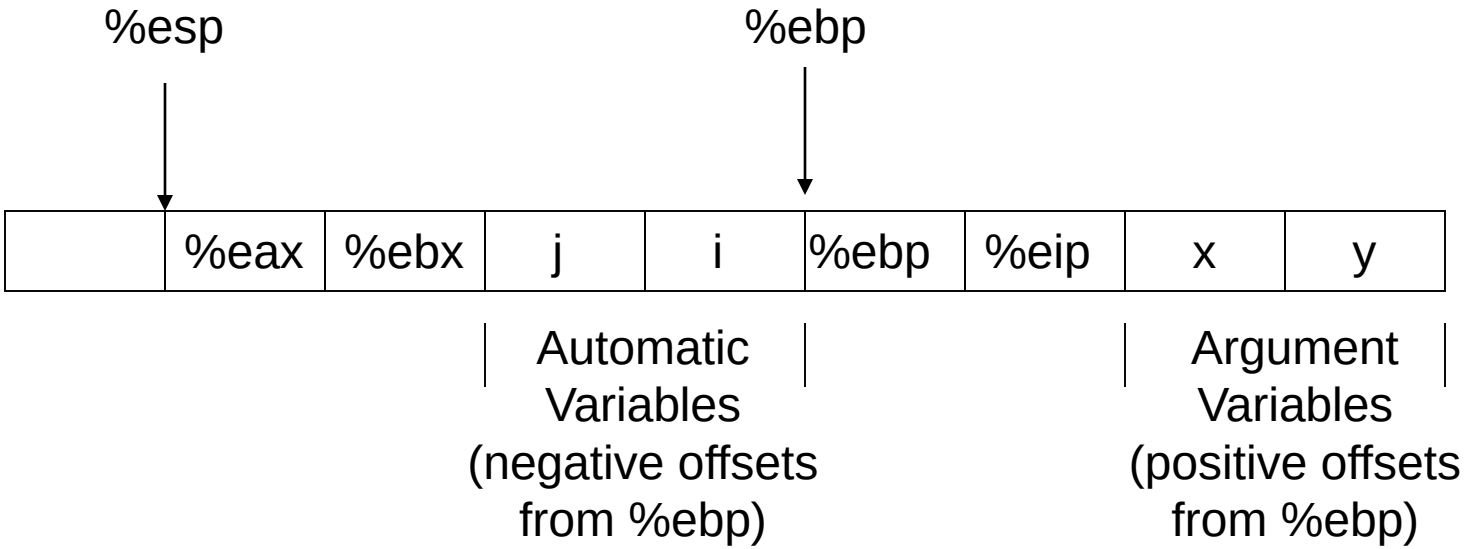
In-line Assembly Code

- Resulting .s file at entry point:

```
_foobar:
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp        # space for automatic variables
        pushl %ebx          # will use %ebx for pointers
#APP
        pushl %eax
#NO_APP
…
```

In-Line Assembly Code

- State of the Stack at maximum depth:



In-line Assembly Code

- Body of function logic

```
movl 8(%ebp),%eax        # i = x;
movl %eax,-4(%ebp)
leal -4(%ebp),%ebx       # j = &i;
movl %ebx,-8(%ebp)
movl 12(%ebp),%eax       # *y = *j;
movl -8(%ebp),%edx
movl (%edx),%ecx
movl %ecx,(%eax)
```

In-line Assembly Code

- Resulting .s file at return point:

```
#APP
        popl %eax
#NO_APP
        xorl %eax,%eax        # clear %eax for return 0;
        jmp L1
        .align 2,0x90
L1:
        movl -12(%ebp),%ebx
        leave                 # translates to instructions below
#       movl  %ebp, %esp
#       popl   %ebp
        ret
```

Machine Language

- Lets look at our disassembly (i386-objdump) of tiny.lnx

```
u18(8)% disas --full tiny.lnx
tiny.lnx:     file format a.out-i386-linux
Contents of section .text:
0000 b8080000 0083c003 a3000200 00cc9090  ................
Contents of section .data:
Disassembly of section .text:
00000000 <tiny.opc-100100> movl   $0x8,%eax
00000005 <tiny.opc-1000fb> addl   $0x3,%eax
00000008 <tiny.opc-1000f8> movl   %eax,0x200
0000000d <tiny.opc-1000f3> int3
0000000e <tiny.opc-1000f2> nop
0000000f <tiny.opc-1000f1> Address 0x10 is out of bounds
```

Machine Language

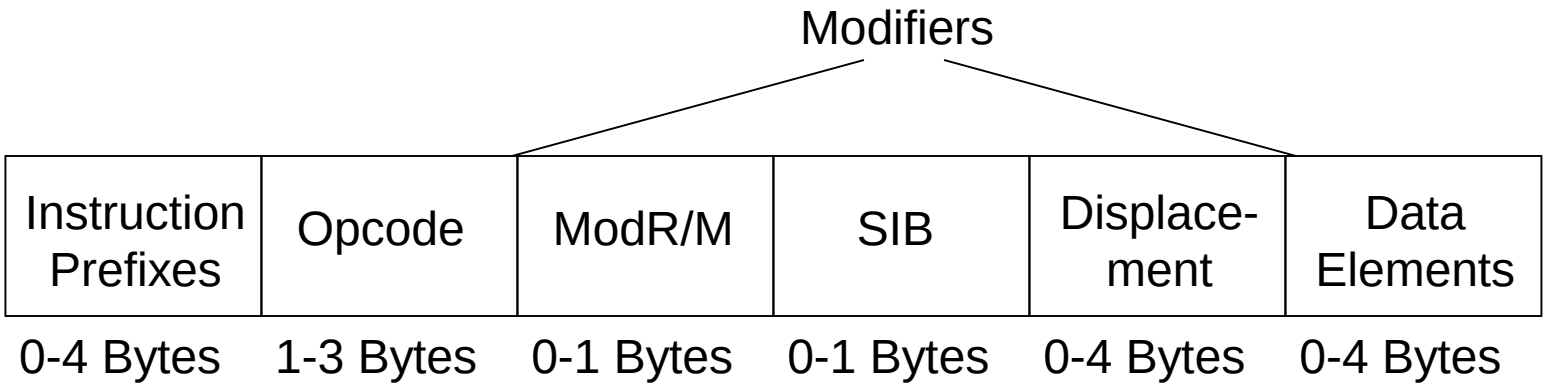
- Another way to show the same data (used to be “tiny.lis” file)

```
# comments from the source code
00000000    b8 08 00 00 00    movl   $0x8,%eax        # comments
00000005    83 c0 03          addl   $0x3,%eax        # comments
00000008    a3 00 02 00 00    movl   %eax,0x200       # comments
0000000d    cc                int3                   # comments
0000000e    90                nop                    (filler)
0000000f    90                nop                    (filler)
```

- How do we understand the hex code values?
- We need to understand the machine language coding rules!
  - Built up using various required and optional binary fields
  - Each field has a unique location, size in bits, and meaning for code values

Machine Language

- The i386 byte by byte binary instruction format is shown in the figure with fields:
  - Optional instruction prefix
  - Operation code (op code)
  - Optional Modifier
  - Optional Data Elements



Machine Language

- Binary Machine Coding for Some Sample Instructions

	Opcode	ModR/M	Data	Total
movl reg, reg	10001001	11sssddd	none	2
movl idata, reg	10111ddd		idata	5
addl reg, reg	00000001	11sssddd	none	2
addl idata, reg	10000001	11000ddd*	idata	6
subl reg, reg	00101001	11sssddd	none	2
subl idata, reg	10000001	11101ddd*	idata	6
incl reg	01000ddd			1
decl reg	01001ddd			1

Machine Language

- ModR/M 3-Bit Register Codes (for sss or ddd)

%eax	000	%esp	100
%ecx	001	%ebp	101
%edx	010	%esi	110
%ebx	011	%edi	111

- * Optimization:

  For some instructions involving %eax, there is a shorter machine code available (hence prefer %eax)

Machine Language

- Examples from tiny.lnx:

```
b8 08 00 00 00      movl   $0x8,%eax
b8                  = 1011 1ddd with ddd = 000 for %eax
    08 00 00 00     = immediate data for 0x00000008

83 c0 03            addl   $0x3,%eax (See Note)
83                  = opcode
    c0              = 11000ddd with ddd = 000 for %eax
        03          = short version of immediate data value
```

Note: Shorter than 81 cx 03 00 00 00 (x != 0) → optimized

Machine Language

- Why should you care about machine language?
- Generating optimal code for performance!!
- Example:

  b8 00 00 00 00 movl $0, %eax # clear %eax

  Generates five bytes

  31 c0 xorl %eax, %eax # clear %eax

  Generates two bytes
- Two bytes uses less program memory and is faster to fetch and execute than five bytes!!

void exchange(int *x, int *y)

/* A C callable assembly language function to exchange two int values via pointers.

The function prototype in C is:

    extern void exchange(int *x, int *y);

The function uses the pointer arguments to exchange the values of the two int variables x and y. You can write your assembly code with or (more efficiently) without a stack frame as you wish.

*/

// Here is a C version of the exchange function for reference:

```
void exchange(int *x, int *y)
{
   int dummy = *x;    // three way move
   *x = *y;
   *y = dummy;
}
```

void exchange(int *x, int *y)

```
_exchange:
    push %ebp              # set up stack frame
    movl %esp, %ebp
    subl $4, %esp          # allocate dummy automatic variable for 3 way move

    movl 8(%ebp), %ecx     # get argument (pointer to x)
    movl 12(%ebp), %edx    # get argument (pointer to y)

    movl (%ecx), %eax      # three way move
    movl %eax, -4(%ebp)
    movl (%edx), %eax
    movl %eax, (%ecx)
    movl -4(%ebp), %eax
    movl %eax, (%edx)

    movl %ebp, %esp        # remove auto variable from stack
    popl %ebp              # restore previous stack frame
    ret                    # void return - so %eax is immaterial
```

void exchange(int *x, int *y)

/* Without a stack frame using a register variable (%ebx) instead of an automatic variable for three way move:
*/

```
_exchange:
    pushl %ebx              # save ebx to use for 3 way move
    movl 8(%esp), %ecx      # get argument (pointer to x)
    movl 12(%esp), %edx     # get argument (pointer to y)

    movl (%ecx), %ebx       # three way move
    movl (%edx), %eax
    movl %eax, (%ecx)
    movl %ebx, (%edx)

    popl %ebx               # restore ebx
    ret                     # void return - so %eax is immaterial
```

void exchange(int *x, int *y)

```
/* Without a stack frame and the wizard's version of the three way move:
   (Saves 4 bytes on the stack and 2 bytes of program memory.)
*/
_exchange:
    movl 4(%esp), %ecx          # get argument (pointer to x)
    movl 8(%esp), %edx          # get argument (pointer to y)
                                # three way move
                                #  x          y          %eax     Data Bus
    movl (%ecx), %eax           #  x          y          x        1R
    xorl (%edx), %eax           #  x          y          x^y      1R
    xorl %eax, (%ecx)           #  x^x^y(=y)  y          x^y      1W
    xorl %eax, (%edx)           #  y          x^y^y(=x)  x^y      1W

    ret                         # void return - so %eax is immaterial
```

void exchange(int *x, int *y)

- Normal 3 way move w/o a stack frame:

- Contents of section .text:
- 0000 538b4c24 088b5424 0c8b198b 02890189 S.L$..T$........
- 0010 1a5bc390 c3 = ret 90 = nop (filler) .[..

- Wizard's version:

- Contents of section .text:
- 0000 8b4c2404 8b542408 8b013302 31013102 .L$..T$...3.1.1.
- 0010 c3909090 c3 = ret 90 = nop (filler) ....

void exchange(int *x, int *y)

- Two Students previously gave a “thinking out of the box” version:
- (Same as the wizard’s version for program memory usage, but uses 8 more bytes on the stack and 4 more bus cycles to memory)
-
- _exchange:

```
        movl 4(%esp), %ecx      # get argument (pointer to x)
        movl 8(%esp), %edx      # get argument (pointer to y)
                                # three way move        Data Bus Cycles
        pushl (%ecx)                                    # 1R + 1W
        pushl (%edx)                                    # 1R + 1W
        popl (%ecx)                                     # 1R + 1W
        popl (%edx)                                     # 1R + 1W

        ret                     # void return - so %eax is immaterial
```