CS354: Machine Organization and Programming

Lecture 8
Monday the September 21th 2015

Section 2
Instructor: Leo Arulraj

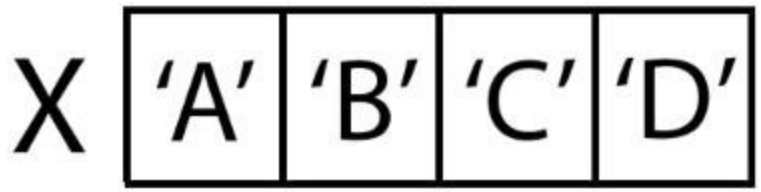
Class Announcements

1. Urmish's office hours from 9-10 AM was cancelled. Alternated office hours details soon.
2. If you need alternate Midterm 1 email me with your name and the reason.
3. Questions about Prog.Assign. 1?
4. Details about hands-on intro to C Program. relevant to Prog.Assign. 1 soon.

Lecture Overview

- IEEE Floating Point
- ISA history and intro
- Assembly Intro, Disassembly
- IA32 Registers
- IA32 Operand forms
- IA32 Data Movement Instructions

- Assume a set of 4 chars. are in an integer-sized variable (X).
- Assume an instruction exists to print out the character all the way to the right…




```
putc X
```
(prints D)

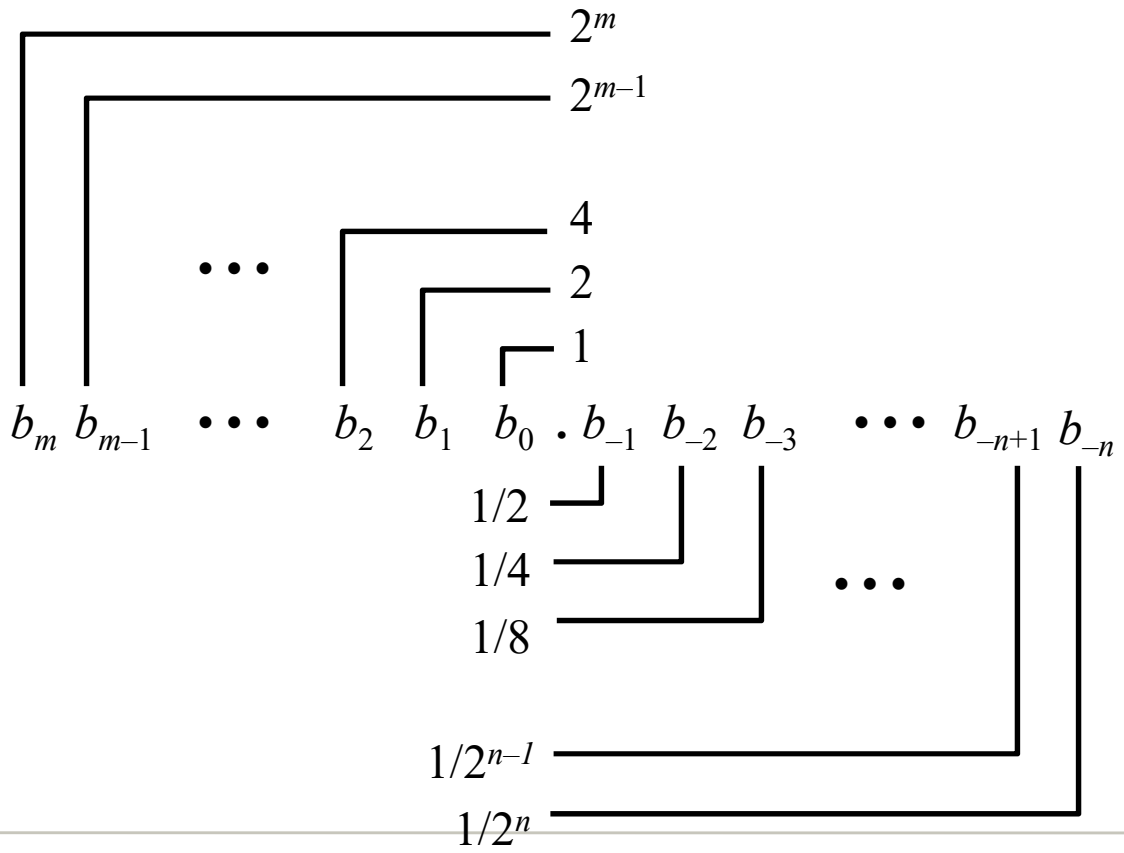
- Invent instructions, and write code to print ABCD, without changing X.

Karen's solution

```
rotl X, 8 bits
putc X    # A
rotl X, 8 bits
putc X    # B
rotl X, 8 bits
putc X    # C
rotl X, 8 bits
putc X    # D
```

Floating Point (Won't be on exam)

1. Fractional Binary Notation



IEEE Floating Point (Won't be on exam)

- Limitations with binary Notation:
  - Can only exactly represent numbers of the form $x/2^k$
  - Just one setting of binary point within the *w* bits
- IEEE Standard 754
  - Established in 1985 as uniform standard for floating point arithmetic
  - Nice standards for rounding, overflow, underflow

IEEE Floating Point (Won't be on exam)

- Numerical Form:

$$(-1)^s \; M \; 2^E$$

  - ***S*** is sign bit : negative or positive
  - **Significand *M*** normally a fractional value in range [0.0,2.0).
  - **Exponent *E*** weights value by power of two

- Encoding
  - MSB s is sign bit ***s***
  - exp field encodes ***E*** (but is not equal to E)
  - frac field encodes ***M*** (but is not equal to M)

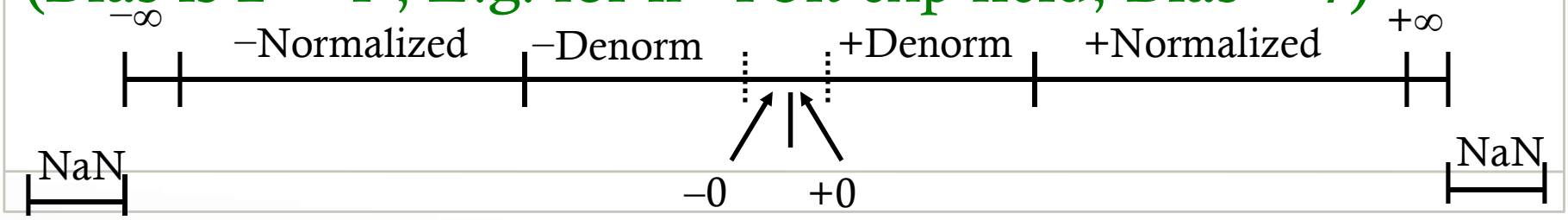
IEEE Floating Point (Won't be on exam)

1. Normalized values: Exp neither all zeroes nor all ones)
   E = Exp – Bias and M = 1 + f

2. Denormalized values: Exp is all zeroes
   E = 1 – Bias and M = f

3. Special values: Exp is all ones : Inf, NaN

(Bias is $2^{k-1}-1$, E.g. for k=4 bit exp field, Bias = 7)



IEEE Floating Point (Won't be on exam)

- Single precision: 32 bits

1 8-bits 23-bits

- Double precision: 64 bits

1 11-bits 52-bits

- Extended precision: 80 bits (Intel only)

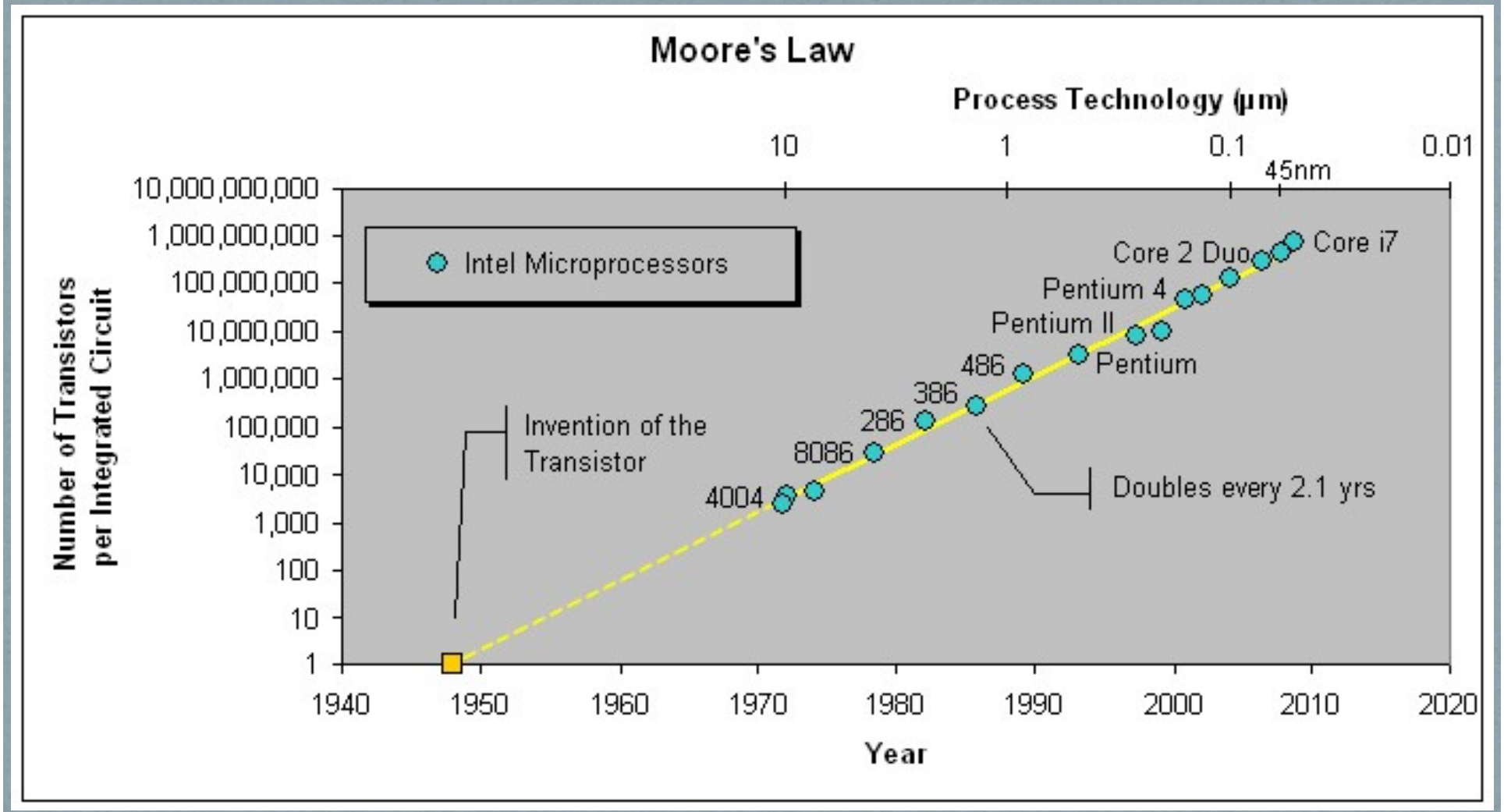
1 15-bits 63 or 64-bits

IEEE Floating Point (Won't be on exam)

Description	Bit representation	e	E	f	M	V
Zero	0 0000 000	0	-6	0	0	0
Smallest positive	0 0000 001	0	-6	1/8	1/8	1/512
	0 0000 010	0	-6	2/8	2/8	2/512
	0 0000 011	0	-6	3/8	3/8	3/512
	0 0000 110	0	-6	6/8	6/8	6/512
Largest denorm.	0 0000 111	0	-6	7/8	7/8	7/512
Smallest norm.	0 0001 000	1	-6	0	8/8	8/512
	0 0001 001	1	-6	1/8	9/8	9/512
	0 0110 110	6	-1	6/8	14/8	14/16
	0 0110 111	6	-1	7/8	15/8	15/16
One	0 0111 000	7	0	0	8/8	1
	0 0111 001	7	0	1/8	9/8	9/8
	0 0111 010	7	0	2/8	10/8	10/8
	0 1110 110	14	7	6/8	14/8	224
Largest norm.	0 1110 111	14	7	7/8	15/8	240
Infinity	0 1111 000	-	-	-	-	+∞

Focus: x86 architecture

- 1960s: CISC
  System/360(IBM),B5000(Burroughs), Motorola 68000
- 1970s: Large Scale Integration
  8008,8080,8086 (Intel), PDP-11,VAX(DEC)
- 1980s: RISC, Instruction Level Parallelism, Pipelining
  80286, 80386,80486(Intel), Motorola 68020
- 1990s - today: Multi-threading , Multi-Core, Open source processors
  Pentium, Pentium Pro, Intel Core(Intel), Athlon series(AMD)


Moore's Law
Process Technology (µm)
10
1
0.1
0.01
45nm
10,000,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
Number of Transistors per Integrated Circuit
Intel Microprocessors
Core i7
Core 2 Duo
Pentium 4
Pentium II
Pentium
486
386
286
8086
4004
Invention of the Transistor
Doubles every 2.1 yrs
1940
1950
1960
1970
1980
1990
2000
2010
2020
Year

entire architecture on 1 slide:

32- bit architecture

2-address instruction set

CISC (not RISC, load/store)

8 registers (depending on how we count)

uses condition codes for control instructions

IA 32

Assembly Programmer's view

- **Programmer-Visible State**
  - PC: Program counter
    - Address of next instruction
    - Called "EIP" (IA32) or "RIP" (x86-64)
  - Register file
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching
- **Memory**
  - Byte addressable array
  - Code, user data, heap, (some) OS data
  - Includes stack used to support procedures

Assembly Language

- Why learn assembly ?
  - Preferred for low level tasks: boot loaders, system calls
  - Less overhead than with HLL code
  - Helpful while debugging
  - Can access some new features of processor only through assembly until compilers add support.
  - One of the oldest tools in Programmers toolbox
- We will use the ATT syntax and not the Intel syntax.
  Consider:

```
Intel:     mov eax,1;       mov ebx,0ffh ;        int 80h
AT&T:   movl $1,%eax;  movl $0xff,%ebx;   int $0x80
```

Characteristics of Assembly Programs: Data Types

1. Integer data of 1,2,or 4 bytes (data values, addresses)
2. Floating point data of 4,8,or 10 bytes
3. No aggregate types such as arrays or structures (Just contiguously allocated bytes in memory)

Characteristics of Assembly Programs: Operations

1. Arithmetic operations on memory or registers
2. Transfer data between memory and registers: Load and Store
3. Transfer control: Unconditional jumps, Conditional branches

Example Assembly Program

```
.include "defines.h“
.data
hw:
    .string "hello world\n“
.text
.globl main
main:
    movl    $SYS_write,%eax
    movl    $1,%ebx
    movl    $hw,%ecx
    movl    $12,%edx
    int $0x80
    movl    $SYS_exit,%eax
    xorl    %ebx,%ebx
    int $0x80
    ret
```

Generating Assembly Code from C

Example C Program and its assembly

```
#include <stdio.h>
int a = 10,b =20;
int main(){
    int t = a;
    a =b;
    b =t;
    printf("%d %d\n",a,b);
    return 0;
}
```

Disassembly of Executables

1. objdump –S
2. gdb and then disassembly command
3. Compile with –g for source code info:

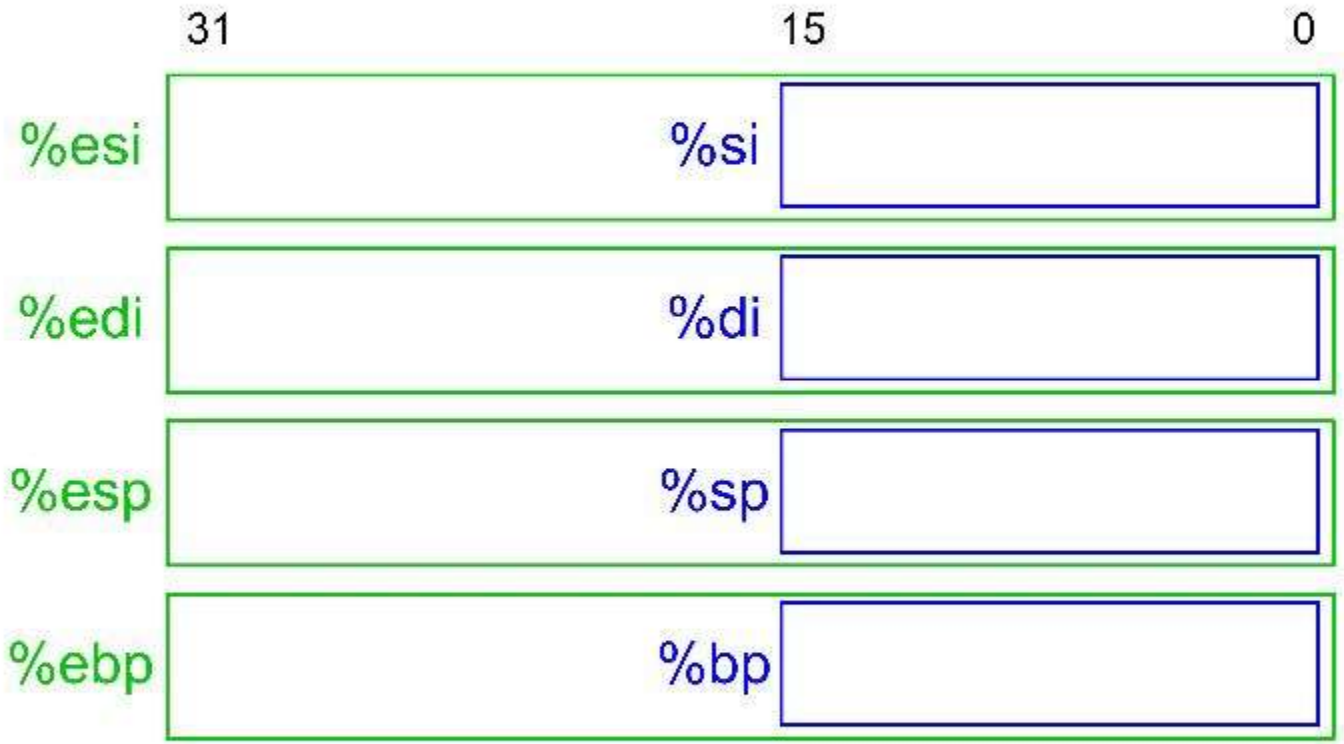
   man gcc says :

   ```
   -g Produce debugging information in the
   operating system's native format
   (stabs, COFF, XCOFF, or DWARF 2).
   GDB can work with this debugging information.
   ```

Registers


31
15
8 7
0
%eax
%ax
%ah
%al
%ecx
%cx
%ch
%cl
%edx
%dx
%dh
%dl
%ebx
%bx
%bh
%bl
"double word"
"word"

4 More Registers



Registers

1. %esp, %ebp : stack pointer, base pointer
2. %eip : instruction pointer
3. x86-64 : %rax, %rbx etc. (64 bits)

What to do when there are not enough registers?

Answer: Store temporarily in memory.

On to the instruction set. Our coverage will be of a small subset.

Classify instructions:

data movement

arithmetic

logical (and shift)

control

Operands

Syntax	Addressing mode name	Effect
`$Imm`	immediate	value in machine code
`%R`	register	value in register `R`
`Imm`	absolute	address given by `Imm`
`(%R)`	register direct (incorrect in textbook)	address in `%R`
`Imm(%R)`	base displacement	address is `Imm + %R`

Some more operand formats in IA32

(E_b, E_i)	$M[R[E_b] + R[E_i]]$	Indexed
$Imm(E_b, E_i)$	$M[Imm + R[E_b] + R[E_i]]$	Indexed
$(, E_i, s)$	$M[R[E_i] \cdot s]$	Scaled indexed
$Imm(, E_i, s)$	$M[Imm + R[E_i] \cdot s]$	Scaled indexed
(E_b, E_i, s)	$M[R[E_b] + R[E_i] \cdot s]$	Scaled indexed
$Imm(E_b, E_i, s)$	$M[Imm + R[E_b] + R[E_i] \cdot s]$	Scaled indexed

Cannot do memory to memory transfer with a single instruction

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Operand	Value
%eax	
0x104	
$0x108	
(%eax)	
4(%eax)	
9(%eax,%edx)	
260(%ecx,%edx)	
0xFC(,%ecx,4)	
(%eax,%edx,4)	

Operand	Value	Comment
%eax	0x100	Register
0x104	0xAB	Absolute address
$0x108	0x108	Immediate
(%eax)	0xFF	Address 0x100
4(%eax)	0xAB	Address 0x104
9(%eax,%edx)	0x11	Address 0x10C
260(%ecx,%edx)	0x13	Address 0x108
0xFC(,%ecx,4)	0xFF	Address 0x100
(%eax,%edx,4)	0x11	Address 0x10C

Data Movement Instructions

`movb`
`movw`
`movl`
	S, D	nondestructive copy of S to D

`movsbw`
`movsbl`
`movswl`
	S, D	sign-extended, nondestructive copy of S to D
		byte to word
		byte to double word
		word to double word

`movzbw`
`movzbl`
`movswl`
	S, D	zero-extended, nondestructive copy of S to D
		byte to word
		byte to double word
		word to double word

`pushl`	S	push double word S onto *the* stack
`popl`	D	pop double word off *the* stack into D

Five possible combination of Source and Destination Types

movl $0x4050,%eax	Immediate--Register, 4 bytes
movw %bp,%sp	Register--Register, 2 bytes
movb (%edi,%ecx),%ah	Memory--Register, 1 byte
movb $-17,(%esp)	Immediate--Memory, 1 byte
movl %eax,-12(%ebp)	Register--Memory, 4 bytes