Systems I

Machine-Level Programming I: Introduction

Topics

- Assembly Programmer's Execution Model
- Accessing Information
  - Registers

IA32 Processors

Totally Dominate General Purpose CPU Market

Evolutionary Design

- Starting in 1978 with 8086
- Added more features as time goes on
- Still support old features, although obsolete

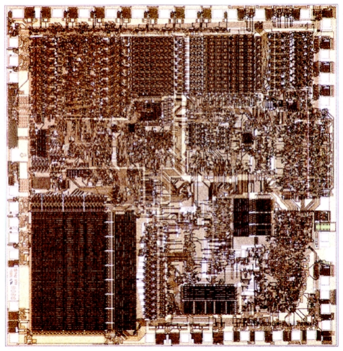
Complex Instruction Set Computer (CISC)

- Many different instructions with many different formats
  - But, only small subset encountered with Linux programs
- Hard to match performance of Reduced Instruction Set Computers (RISC)
- But, Intel has done just that!

X86 Evolution: Programmer's View

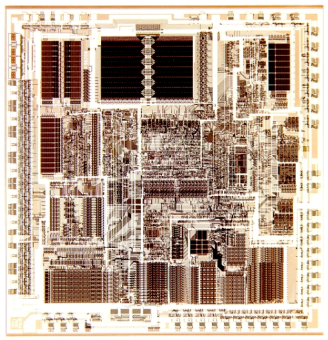
Name	Date	Transistors
8086	1978	29K

- 16-bit processor. Basis for IBM PC & DOS
- Limited to 1MB address space. DOS only gives you 640K



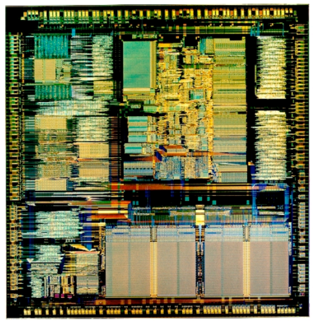
Name	Date	Transistors
80286	1982	134K

- Added elaborate, but not very useful, addressing scheme
- Basis for IBM PC-AT and Windows



Name	Date	Transistors
386	1985	275K

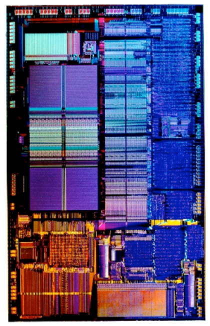
- Extended to 32 bits. Added "flat addressing"
- Capable of running Unix
- Linux/gcc uses no instructions introduced in later models



X86 Evolution: Programmer's View

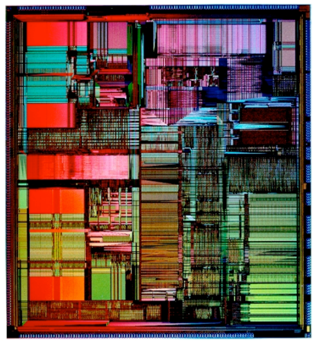
Name	Date	Transistors
486	1989	1.9M

- Added on-chip floating-point unit



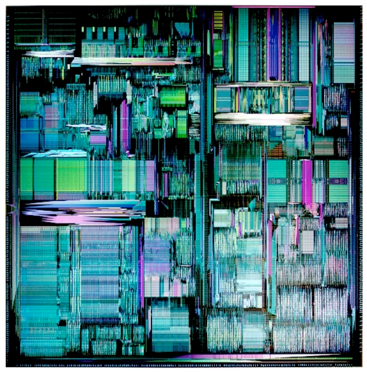
Pentium	1993	3.1M
Pentium/MMX	1997	4.5M

- Added special collection of instructions for operating on 64-bit vectors of 1, 2, or 4 byte integer data



PentiumPro	1995	6.5M

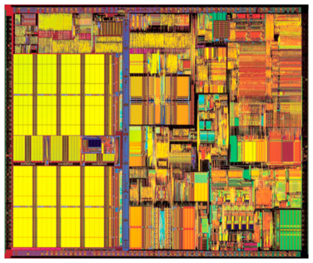
- Added conditional move instructions
- Hardware can execute instructions out of order



X86 Evolution: Programmer's View

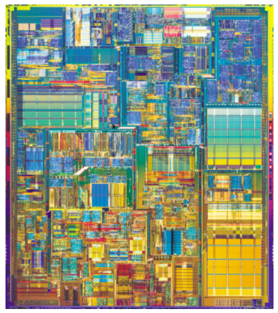
Name	Date	Transistors
Pentium III	1999	8.2M

- Added "streaming SIMD" instructions for operating on 128-bit vectors of 1, 2, or 4 byte integer or floating point data



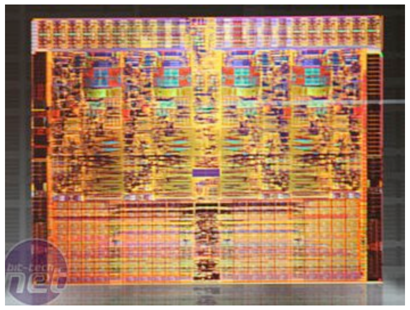
Name	Date	Transistors
Pentium 4	2001	42M

- Added 8-byte formats and 144 new instructions for streaming SIMD mode
- "Superpipelined" with very fast clocks



Name	Date	Transistors
"Nehalem"	2009	700M+

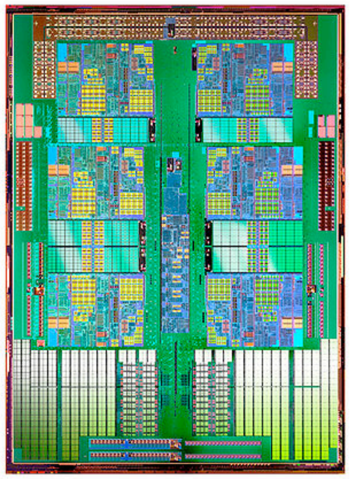
- 4 Cores on the same chip
- 8MB+ of on-chip memory



X86 Evolution: Clones

Advanced Micro Devices (AMD)

- Historically
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper
- Recently
  - Drove 64-bit extensions to IA32 architecture
  - Acquired ATI (graphics chip company)
  - Increasing core counts too
  - 6-core Opteron (Istanbul) 2009



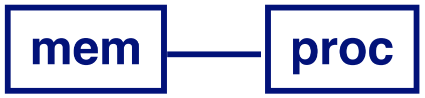
Variety of x86 chips in different markets

- Embedded/low power (Atom, Neo)
- Desktop/laptop
- Server
- Supercomputer

Abstract and Concrete Machine Models

Machine Models

C



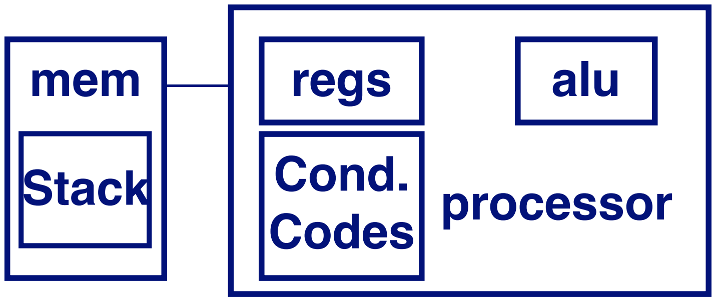
Data

1) `char`
2) `int`, `float`
3) `double`
4) `struct`, array
5) pointer

Control

1) loops
2) conditionals
3) `switch`
4) Proc. call
5) Proc. return

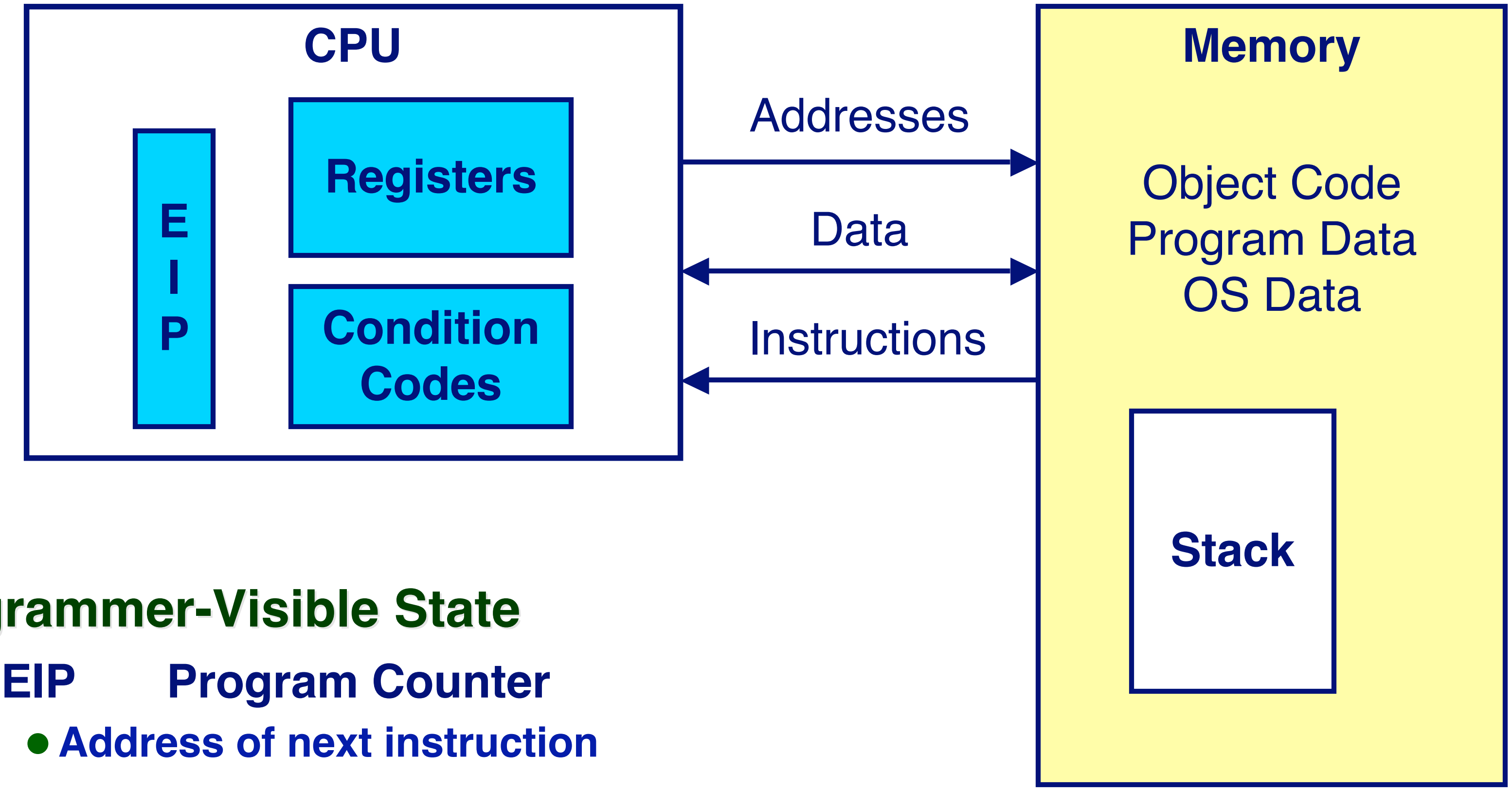
Assembly



1) byte
2) 2-byte word
3) 4-byte long word
4) contiguous byte allocation
5) address of initial byte

3) branch/jump
4) `call`
5) `ret`

Assembly Programmer's View



Programmer-Visible State

- EIP Program Counter
  - Address of next instruction
- 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, (some) OS data
  - Includes stack used to support procedures

By the *architecture* of a system, I mean the complete and detailed specification of the user interface. … As Blaauw has said, "Where architecture tells *what* happens, implementation tells *how* it is made to happen."

***The Mythical Man-Month*, Brooks, pg 45**

Slide adapted from M.J. Irwin, 2005

Instruction Set Architecture Principles

Contract between programmer and the hardware

- Defines visible state of the system
- Defines how state changes in response to instructions

Programmer: ISA is model of how a program will execute

Hardware Designer: ISA is formal definition of the correct way to execute a program

- With a stable ISA, SW doesn't care what the HW looks like under the covers
  - Hardware implementations can change (drastically)
  - As long as the HW implements the same ISA, all prior SW will still run
- Example: x86 ISA has spanned many chips
  - Instructions have been added but the SW of prior chips still runs on later chips

ISA specification

- The binary encodings of the instruction set

Instruction Set Architecture

Contract between programmer and the hardware

- Defines visible state of the system
- Defines how state changes in response to instructions

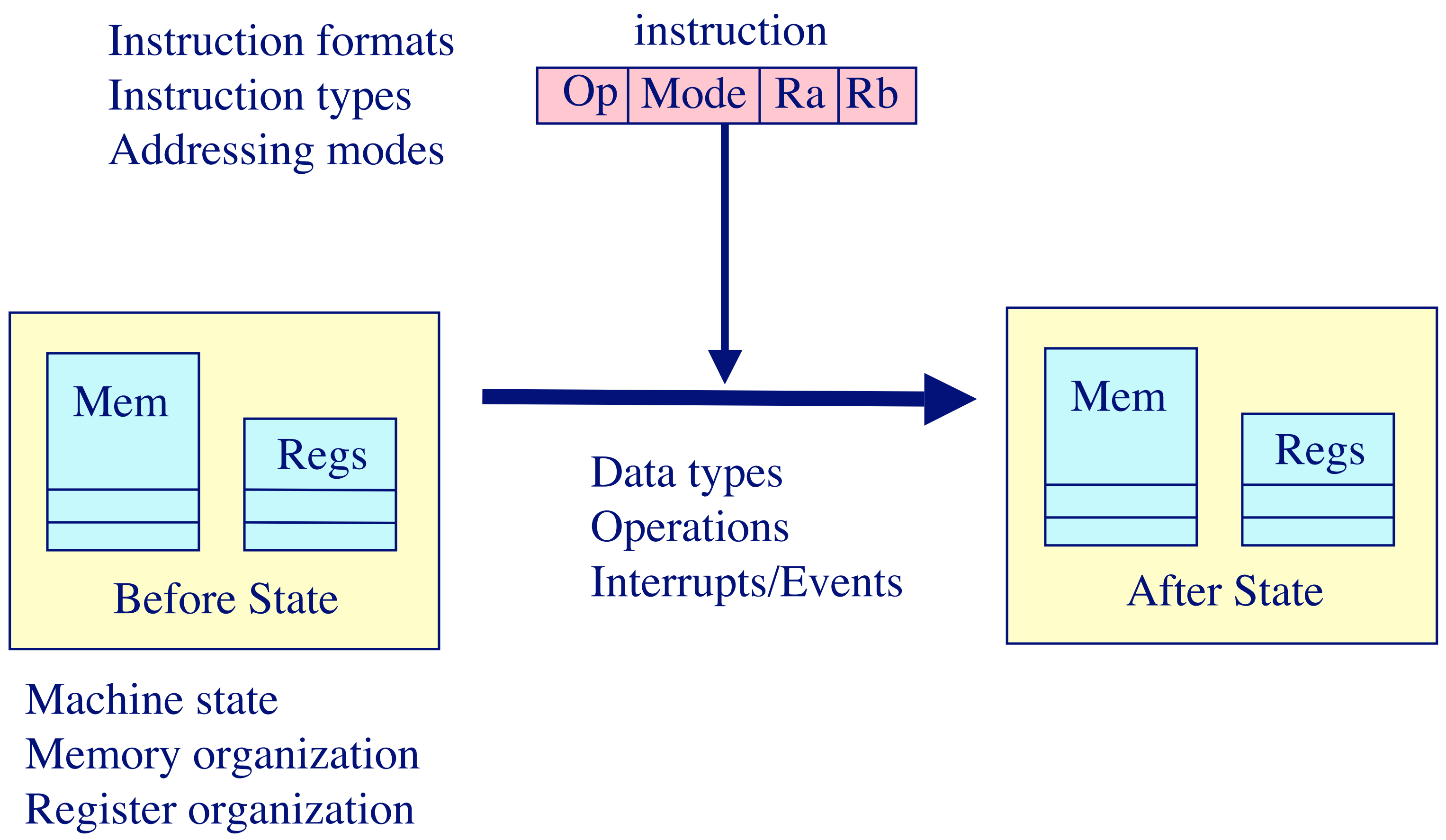
Programmer: ISA is model of how a program will execute

Hardware Designer: ISA is formal definition of the correct way to execute a program

ISA specification

- The binary encodings of the instruction set

ISA Basics


Instruction formats
Instruction types
Addressing modes
instruction
Op
Mode
Ra
Rb
Mem
Regs
Before State
Data types
Operations
Interrupts/Events
Mem
Regs
After State
Machine state
Memory organization
Register organization

Architecture vs. Implementation

Architecture: defines what a computer system does in response to a program and a set of data

- Programmer visible elements of computer system

Implementation: defines how a computer does it

- Sequence of steps to complete operations
- Time to execute each operation
- Hidden “bookkeeping” functions

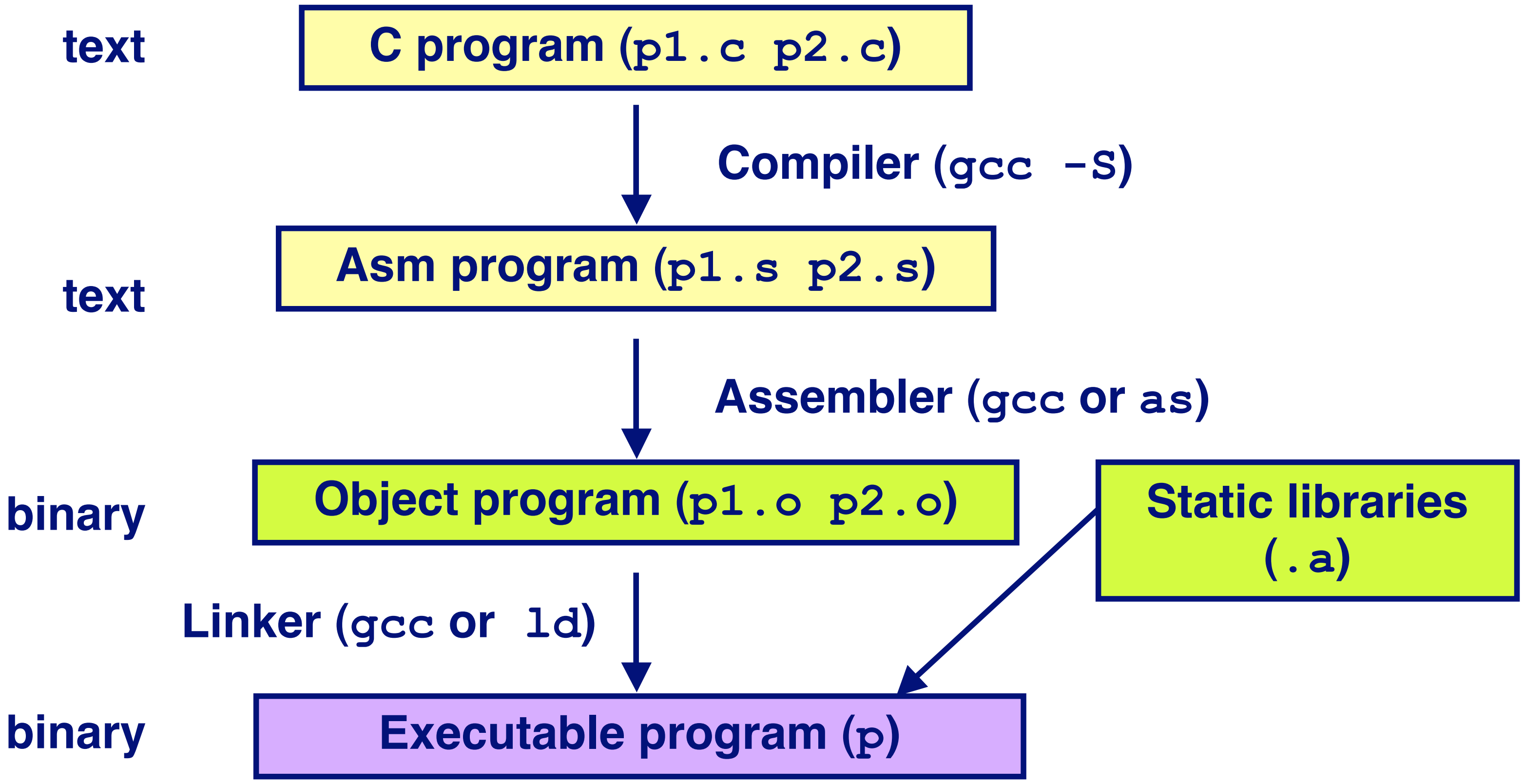
Examples

Architecture or Implementation?

- Number of GP registers
- Width of memory bus
- Binary representation of the instruction
  `sub r4,r2,#27`
- Number of cycles to execute FP instruction
- How condition code bits are set on a move instruction
- Size of the instruction cache
- Type of FP format

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -O p1.c p2.c -o p`
  - Use optimizations (`-O`)
  - Put resulting binary in file `p`



Compiling Into Assembly

C Code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Generated Assembly

```
_sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Obtain with command

```
gcc -O -S code.c
```

Produces file `code.s`

Assembly Characteristics

Minimal Data Types

- "Integer" data of 1, 2, or 4 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

Primitive Operations

- Perform arithmetic function on register or memory data
- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

Object Code

Code for `sum`

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

- Total of 13 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address `0x401040`

Assembler

- Translates `.s` into `.o`
- Binary encoding of each instruction
- Nearly-complete image of executable code
- Missing linkages between code in different files

Linker

- Resolves references between files
- Combines with static run-time libraries
  - E.g., code for `malloc`, `printf`
- Some libraries are *dynamically linked*
  - Linking occurs when program begins execution

Machine Instruction Example

```
int t = x+y;
```

C Code

- Add two signed integers

```
    addl 8(%ebp),%eax
```

Similar to expression
`x += y`

Assembly

- Add 2 4-byte integers
  - "Long" words in GCC parlance
  - Same instruction whether signed or unsigned
- Operands:

`x:`	Register	`%eax`
`y:`	Memory	M[`%ebp+8`]
`t:`	Register	`%eax`

  » Return function value in `%eax`

```
0x401046:     03 45 08
```

Object Code

- 3-byte instruction
- Stored at address `0x401046`

Disassembling Object Code

Disassembled

```
00401040 <_sum>:
   0:      55                push   %ebp
   1:      89 e5             mov    %esp,%ebp
   3:      8b 45 0c          mov    0xc(%ebp),%eax
   6:      03 45 08          add    0x8(%ebp),%eax
   9:      89 ec             mov    %ebp,%esp
   b:      5d                pop    %ebp
   c:      c3                ret
   d:      8d 76 00          lea    0x0(%esi),%esi
```

Disassembler

```
objdump -d p
```

- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either `a.out` (complete executable) or `.o` file

Alternate Disassembly

Object

```
0x401040:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

Disassembled

```
0x401040 <sum>:      push   %ebp
0x401041 <sum+1>:    mov    %esp,%ebp
0x401043 <sum+3>:    mov    0xc(%ebp),%eax
0x401046 <sum+6>:    add    0x8(%ebp),%eax
0x401049 <sum+9>:    mov    %ebp,%esp
0x40104b <sum+11>:   pop    %ebp
0x40104c <sum+12>:   ret
0x40104d <sum+13>:   lea    0x0(%esi),%esi
```

Within gdb Debugger

```
gdb p
disassemble sum
```

- Disassemble procedure

```
x/13b sum
```

- Examine the 13 bytes starting at `sum`

What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:     file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55             push   %ebp
30001001:  8b ec          mov    %esp,%ebp
30001003:  6a ff          push   $0xffffffff
30001005:  68 90 10 00 30 push   $0x30001090
3000100a:  68 91 dc 4c 30 push   $0x304cdc91
```

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

Whose Assembler?

Intel/Microsoft Format

```
lea   eax,[ecx+ecx*2]
sub   esp,8
cmp   dword ptr [ebp-8],0
mov   eax,dword ptr [eax*4+100h]
```

GAS/Gnu Format

```
leal (%ecx,%ecx,2),%eax
subl $8,%esp
cmpl $0,-8(%ebp)
movl $0x100(,%eax,4),%eax
```

Intel/Microsoft Differs from GAS

- Operands listed in opposite order

  `mov` Dest, Src `movl` Src, Dest

- Constants not preceded by ‘$’, Denote hex with ‘h’ at end

  `100h` `$0x100`

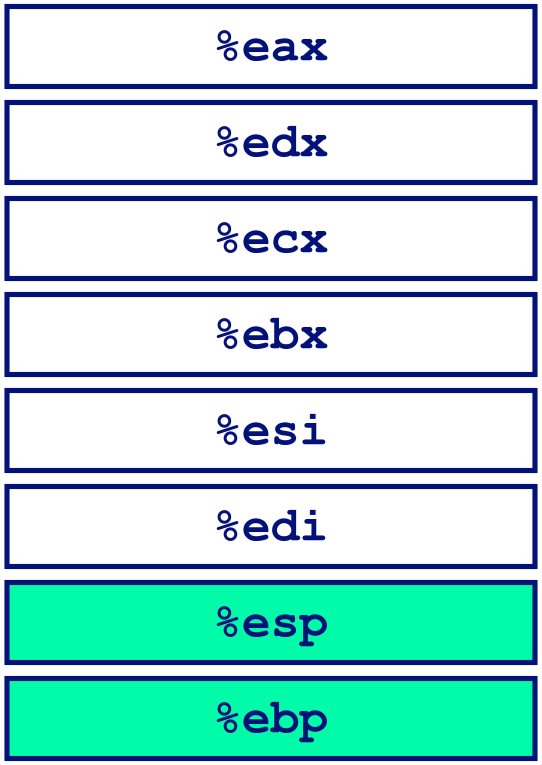
- Operand size indicated by operands rather than operator suffix

  `sub` `subl`

- Addressing format shows effective address computation

  `[eax*4+100h]` `$0x100(,%eax,4)`

Moving Data



Moving Data

`movl` ***Source*,*Dest***:

- Move 4-byte ("long") word
- Lots of these in typical code

Operand Types

- Immediate: Constant integer data
  - Like C constant, but prefixed with '`$`'
  - E.g., `$0x400`, `$-533`
  - Encoded with 1, 2, or 4 bytes
- Register: One of 8 integer registers
  - But `%esp` and `%ebp` reserved for special use
  - Others have special uses for particular instructions
- Memory: 4 consecutive bytes of memory
  - Various "address modes"

`movl` Operand Combinations

	Source	Destination		C Analog
`movl`	*Imm*	*Reg*	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax),%edx`	`temp = *p;`

- Cannot do memory-memory transfers with single instruction

Simple Addressing Modes

Normal	**(R)**	**Mem[Reg[R]]**

- Register R specifies memory address

```
movl (%ecx),%eax
```

Displacement	**D(R)**	**Mem[Reg[R]+D]**

- Register R specifies start of memory region
- Constant displacement D specifies offset

```
movl 8(%ebp),%edx
```

Summary

Today

- ISA/processor evolution (for x86)
- Programmer machine models
- Introduction to ISA and usage

Next time

- Memory access
- Arithmetic operations
- C pointers and Addresses