

CS 33

Machine Programming (1)

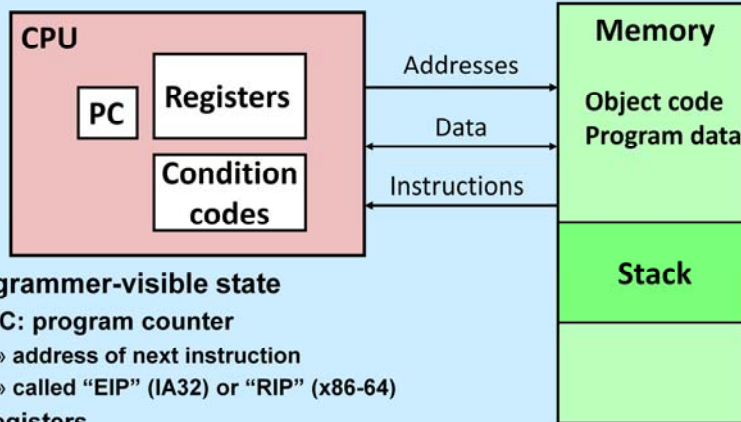
Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective,” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.

Today: Machine Programming I: Basics

- **C, assembly, machine code**
- **Assembly basics: registers, operands, move, addressing modes**
- **Intro to x86-64**

Supplied by CMU.

Assembly Programmer's View



- **Programmer-visible state**

- **PC: program counter**
 - » address of next instruction
 - » called “EIP” (IA32) or “RIP” (x86-64)
- **registers**
 - » quickly accessed program data
- **condition codes**
 - » store status information about most recent arithmetic operation
 - » used for conditional branching

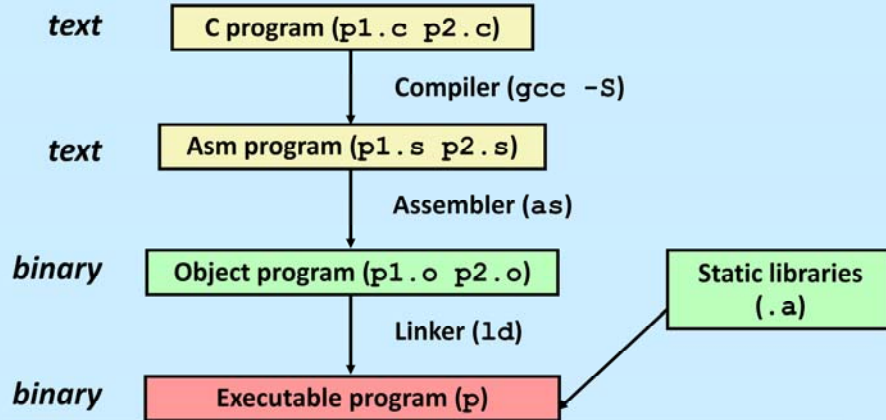
- **memory**

- » byte-addressable array
- » code, data
- » includes stack used to support procedures

Supplied by CMU.

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -O1 p1.c p2.c -o p`
 - » use basic optimizations (`-O1`)
 - » put resulting binary in file `p`



Supplied by CMU.

Note that normally one does not ask `gcc` to produce assembler code, but instead it compiles C code directly into machine code (producing an object file). Note that the `gcc` command actually invokes a script; the compiler (also known as `gcc`) compiles code into either assembler code or machine code; if necessary, the assembler (`as`) assembles assembler code into a object code. The linker (`ld`) links together multiple object files (containing object code) into an executable program.

Compiling Into Assembly

C code

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

Generated IA32 assembly

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

Obtain with command

```
/usr/bin/gcc -O1 -S code.c
```

Produces file code.s

Assembly Characteristics: 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

Supplied by CMU.

Assembly Characteristics: 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

Supplied by CMU.

Object Code

Code for `sum`

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

- Total of 11 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 `printf`
- some libraries are *dynamically linked*
 - » linking occurs when program begins execution

Supplied by CMU.

Disassembling Object Code

Disassembled

```
080483c4 <sum>:  
80483c4: 55      push    %ebp  
80483c5: 89 e5    mov     %esp,%ebp  
80483c7: 8b 45 0c  mov     0xc(%ebp),%eax  
80483ca: 03 45 08  add     0x8(%ebp),%eax  
80483cd: 5d      pop     %ebp  
80483ce: c3      ret
```

- **Disassembler**

`objdump -d <file>`

- useful tool for examining object code
- analyzes bit pattern of series of instructions
- produces approximate rendition of assembly code
- can be run on either executable or object (.o) file

Supplied by CMU.

Alternate Disassembly

Object

0x401040:

0x55
0x89
0xe5
0x8b
0x45
0x0c
0x03
0x45
0x08
0x5d
0xc3

Disassembled

Dump of assembler code for function sum:

```
0x080483c4 <sum+0>:    push    %ebp
0x080483c5 <sum+1>:    mov     %esp,%ebp
0x080483c7 <sum+3>:    mov     0xc(%ebp),%eax
0x080483ca <sum+6>:    add     0x8(%ebp),%eax
0x080483cd <sum+9>:    pop     %ebp
0x080483ce <sum+10>:   ret
```

- **Within gdb debugger**

`gdb <file>`

`disassemble sum`

– disassemble procedure

`x/11xb sum`

– examine the 11 bytes starting at sum

Supplied by CMU.

Machine-Instruction Example

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

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

Similar to expression:

```
x += y
```

Sort of like:

```
int eax;
```

```
int *ebp;
```

```
eax += ebp[2];
```

```
0x80483ca: 03 45 08
```

- **C code**

- add two signed integers

- **Assembly**

- add two 4-byte integers
 - » ints in C parlance
 - » same instruction whether signed or unsigned

- **Operands:**

x: register %eax

y: memory M[%ebp+8]

t: register %eax

- function return value in %eax

- **Object code**

- 3-byte instruction
- stored at address 0x80483ca

Supplied by CMU.

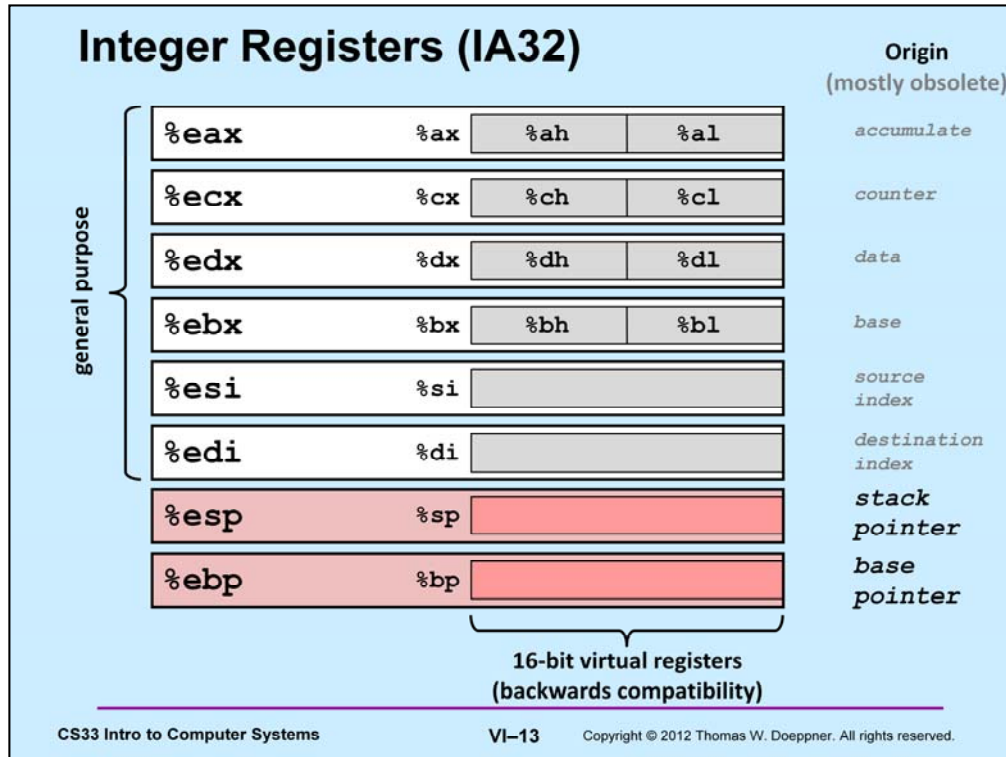
The “M” in “M[%ebp+8]” is the conceptual memory array. In the case of this instruction, we are adding 8 to the contents of register ebp to produce an address, which is treated as an index into the “memory array.”

Note that some assemblers (in particular, those of Intel and Microsoft) place the operands in the opposite order. Thus the example of the slide would be “addl %eax,8(%ebp)”. The order we use is that used by gcc.

Today: Machine Programming I: Basics

- C, assembly, machine code
- **Assembly basics: registers, operands, move**
- Intro to x86-64

Supplied by CMU.



Supplied by CMU.

Moving Data: IA32

- Moving data

`movl source, dest`

- Operand types

- **Immediate:** constant integer data

- » example: `$0x400`, `$-533`
 - » like C constant, but prefixed with `'$'`
 - » encoded with 1, 2, or 4 bytes

- **Register:** one of 8 integer registers

- » example: `%eax`, `%edx`
 - » but `%esp` and `%ebp` reserved for special use
 - » others have special uses for particular instructions

- **Memory:** 4 consecutive bytes of memory at address given by register(s)

- » simplest example: `(%eax)`
 - » various other “address modes”

<code>%eax</code>
<code>%ecx</code>
<code>%edx</code>
<code>%ebx</code>
<code>%esi</code>
<code>%edi</code>
<code>%esp</code>
<code>%ebp</code>

Supplied by CMU.

Note that though `esp` and `ebp` have special uses, they may also be used in both source and destination operands.

movl Operand Combinations

	Source	Dest	Src, Dest	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 transfer with a single instruction

Supplied by CMU.

Simple Memory 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
```

Supplied by CMU.

If one thinks of there being an array of registers, then “Reg[R]” selects register “R” from this array.

Using Simple Addressing Modes

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

swap:

```
pushl %ebp
movl  %esp,%ebp
pushl %ebx
```

} Set
Up

```
movl  8(%ebp), %edx
movl  12(%ebp), %ecx
movl  (%edx), %ebx
movl  (%ecx), %eax
movl  %eax, (%edx)
movl  %ebx, (%ecx)
```

} Body

```
popl  %ebx
popl  %ebp
ret
```

} Finish

Supplied by CMU.

Using Simple Addressing Modes

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

swap:

```
pushl %ebp
movl  %esp,%ebp
pushl %ebx
```

} Set
Up

```
movl 8(%ebp), %edx
movl 12(%ebp), %ecx
movl (%edx), %ebx
movl (%ecx), %eax
movl %eax, (%edx)
movl %ebx, (%ecx)
```

} Body

```
popl  %ebx
popl  %ebp
ret
```

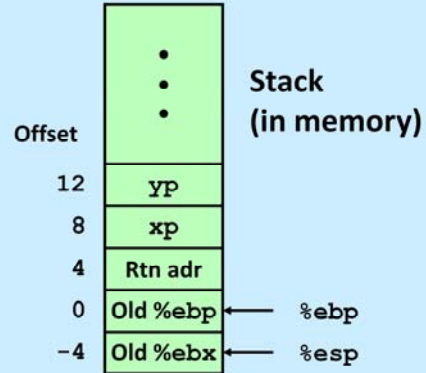
} Finish

Understanding Swap

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Register	Value
%edx	xp
%ecx	yp
%ebx	t0
%eax	t1

```
movl 8(%ebp), %edx    # edx = xp
movl 12(%ebp), %ecx   # ecx = yp
movl (%edx), %ebx     # ebx = *xp (t0)
movl (%ecx), %eax     # eax = *yp (t1)
movl %eax, (%edx)     # *xp = t1
movl %ebx, (%ecx)     # *yp = t0
```



Supplied by CMU.

Understanding Swap

%eax	
%edx	
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

		Address
	123	0x124
	456	0x120
		0x11c
		0x118
		0x114
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%ebp	0	
	-4	
		0x100

```

movl 8(%ebp), %edx # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
    
```

Supplied by CMU.

Understanding Swap

%eax	
%edx	0x124
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

		Address
		123 0x124
		456 0x120
		0x11c
		0x118
Offset		0x114
yp	12	0x120 0x110
xp	8	0x124 0x10c
	4	Rtn adr 0x108
%ebp	→ 0	0x104
	-4	0x100

```

movl 8(%ebp), %edx # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
    
```

Supplied by CMU.

Understanding Swap

%eax	
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

		Address	
		123	0x124
		456	0x120
			0x11c
			0x118
	Offset		0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	→ 0		0x104
	-4		0x100

```
movl 8(%ebp), %edx # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx # ebx = *xp (t0)
movl (%ecx), %eax # eax = *yp (t1)
movl %eax, (%edx) # *xp = t1
movl %ebx, (%ecx) # *yp = t0
```

Understanding Swap

%eax	
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

		Offset	Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	→ 0		0x104
	-4		0x100

```
movl 8(%ebp), %edx # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx # ebx = *xp (t0)
movl (%ecx), %eax # eax = *yp (t1)
movl %eax, (%edx) # *xp = t1
movl %ebx, (%ecx) # *yp = t0
```

Supplied by CMU.

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

		Offset	Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	→ 0		0x104
	-4		0x100

CS33 Intro to Computer Systems

Supplied by CMU.

Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

		Address
		456
		0x124
		0x11c
		0x118
		0x114
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%ebp	→ 0	0x108
	-4	0x104
		0x100

```

movl 8(%ebp), %edx # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx) # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
    
```

Supplied by CMU.

Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

		Address
		456
		0x124
		123
		0x11c
		0x118
		0x114
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%ebp	→ 0	0x108
	-4	0x104
		0x100

```

movl 8(%ebp), %edx # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
    
```

Supplied by CMU.

Complete Memory-Addressing Modes

- Most general form

$D(Rb, Ri, S)$ $Mem[Reg[Rb] + S * Reg[Ri] + D]$

- D: constant “displacement”
- Rb: base register: any of 8 integer registers
- Ri: index register: any, except for %esp
 - » unlikely you’d use %ebp either
- S: scale: 1, 2, 4, or 8

- Special cases

(Rb, Ri) $Mem[Reg[Rb] + Reg[Ri]]$

$D(Rb, Ri)$ $Mem[Reg[Rb] + Reg[Ri] + D]$

(Rb, Ri, S) $Mem[Reg[Rb] + S * Reg[Ri]]$

Address-Computation Examples

<code>%edx</code>	<code>0xf000</code>
<code>%ecx</code>	<code>0x0100</code>

Expression	Address Computation	Address
<code>0x8(%edx)</code>	$0xf000 + 0x8$	<code>0xf008</code>
<code>(%edx,%ecx)</code>	$0xf000 + 0x0100$	<code>0xf100</code>
<code>(%edx,%ecx,4)</code>	$0xf000 + 4*0x0100$	<code>0xf400</code>
<code>0x80(,%edx,2)</code>	$2*0xf000 + 0x80$	<code>0x1e080</code>

Address-Computation Instruction

- **leal *src, dest***
 - *src* is address mode expression
 - set *dest* to address denoted by expression
- **Uses**
 - computing addresses without a memory reference
 - » e.g., translation of `p = &x[i];`
 - computing arithmetic expressions of the form $x + k \cdot y$
 - » $k = 1, 2, 4, \text{ or } 8$
- **Example**

```
int mul12(int x)
{
    return x*12;
}
```

Converted to ASM by compiler:

```
movl 8(%ebp), %eax    # get arg
leal (%eax,%eax,2), %eax # t <- x+x*2
sall $2, %eax         # return t<<2
```

Today: Machine Programming I: Basics

- C, assembly, machine code
- Assembly basics: registers, operands, move, addressing modes
- **Intro to x86-64**

Supplied by CMU.

Data Representations: IA32 + x86-64

- Sizes of C objects (in bytes)

<i>C Data Type</i>	<i>Generic 32-bit</i>	<i>Intel IA32</i>	<i>x86-64</i>
unsigned	4	4	4
int	4	4	4
long int	4	4	8
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
long double	8	10/12	16
char * <i>or any other pointer</i>	4	4	8

Supplied by CMU.

x86-64 Integer Registers

<code>%rax</code>	<code>%eax</code>	<code>%r8</code>	<code>%r8d</code>
<code>%rbx</code>	<code>%ebx</code>	<code>%r9</code>	<code>%r9d</code>
<code>%rcx</code>	<code>%ecx</code>	<code>%r10</code>	<code>%r10d</code>
<code>%rdx</code>	<code>%edx</code>	<code>%r11</code>	<code>%r11d</code>
<code>%rsi</code>	<code>%esi</code>	<code>%r12</code>	<code>%r12d</code>
<code>%rdi</code>	<code>%edi</code>	<code>%r13</code>	<code>%r13d</code>
<code>%rsp</code>	<code>%esp</code>	<code>%r14</code>	<code>%r14d</code>
<code>%rbp</code>	<code>%ebp</code>	<code>%r15</code>	<code>%r15d</code>

- Extend existing registers. Add 8 new ones.
- Make `%ebp/%rbp` general purpose

Supplied by CMU.

Instructions

- Long word `l` (4 Bytes) ↔ Quad word `q` (8 Bytes)
- New instructions:
 - `movl` → `movq`
 - `addl` → `addq`
 - `sall` → `salq`
 - etc.
- 32-bit instructions that generate 32-bit results
 - set higher order bits of destination register to 0
 - example: `addl`

Supplied by CMU.

32-bit code for swap

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

swap:

```
pushl %ebp
movl  %esp,%ebp
pushl %ebx
```

} Set
Up

```
movl  8(%ebp), %edx
movl  12(%ebp), %ecx
movl  (%edx), %ebx
movl  (%ecx), %eax
movl  %eax, (%edx)
movl  %ebx, (%ecx)
```

} Body

```
popl  %ebx
popl  %ebp
ret
```

} Finish

Supplied by CMU.

64-bit code for swap

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

swap:

```
movl    (%rdi), %edx
movl    (%rsi), %eax
movl    %eax, (%rdi)
movl    %edx, (%rsi)
```

ret

} Set
Up

} Body

} Finish

- **Arguments passed in registers (why useful?)**
 - first (xp) in %rdi, second (yp) in %rsi
 - 64-bit pointers
- **No stack operations required**
- **32-bit data**
 - data held in registers %eax and %edx
 - movl operation

Supplied by CMU.

64-bit code for long int swap

<pre>void swap(long *xp, long *yp) { long t0 = *xp; long t1 = *yp; *xp = t1; *yp = t0; }</pre>	<pre>swap_1: movq (%rdi), %rdx movq (%rsi), %rax movq %rax, (%rdi) movq %rdx, (%rsi) ret</pre>	<p>} Set Up</p> <p>} Body</p> <p>} Finish</p>
--	--	---

- **64-bit data**

- data held in registers `%rax` and `%rdx`
- `movq` operation
 - » “q” stands for quad-word

How Many Instructions are There?

- We cover ~29
- Implemented by Intel:
 - 80 in original 8086 architecture
 - 7 added with 80186
 - 17 added with 80286
 - 33 added with 386
 - 6 added with 486
 - 6 added with Pentium
 - 1 added with Pentium MMX
 - 4 added with Pentium Pro
 - 8 added with SSE
 - 8 added with SSE2
 - 2 added with SSE3
 - 14 added with x86-64
 - 10 added with VT-x
 - 2 added with SSE4a
- Total: 198
- Doesn't count:
 - floating-point instructions
 - SIMD instructions
 - AMD-added instructions
 - undocumented instructions

The source for this is http://en.wikipedia.org/wiki/X86_instruction_listings, viewed on 9/18/2012, and also depends upon my ability to count.