

Machine-Level Programming I – Introduction

CSCI 224 / ECE 317: Computer Architecture

Instructor:

Prof. Jason Fritts

Slides adapted from Bryant & O'Hallaron's slides

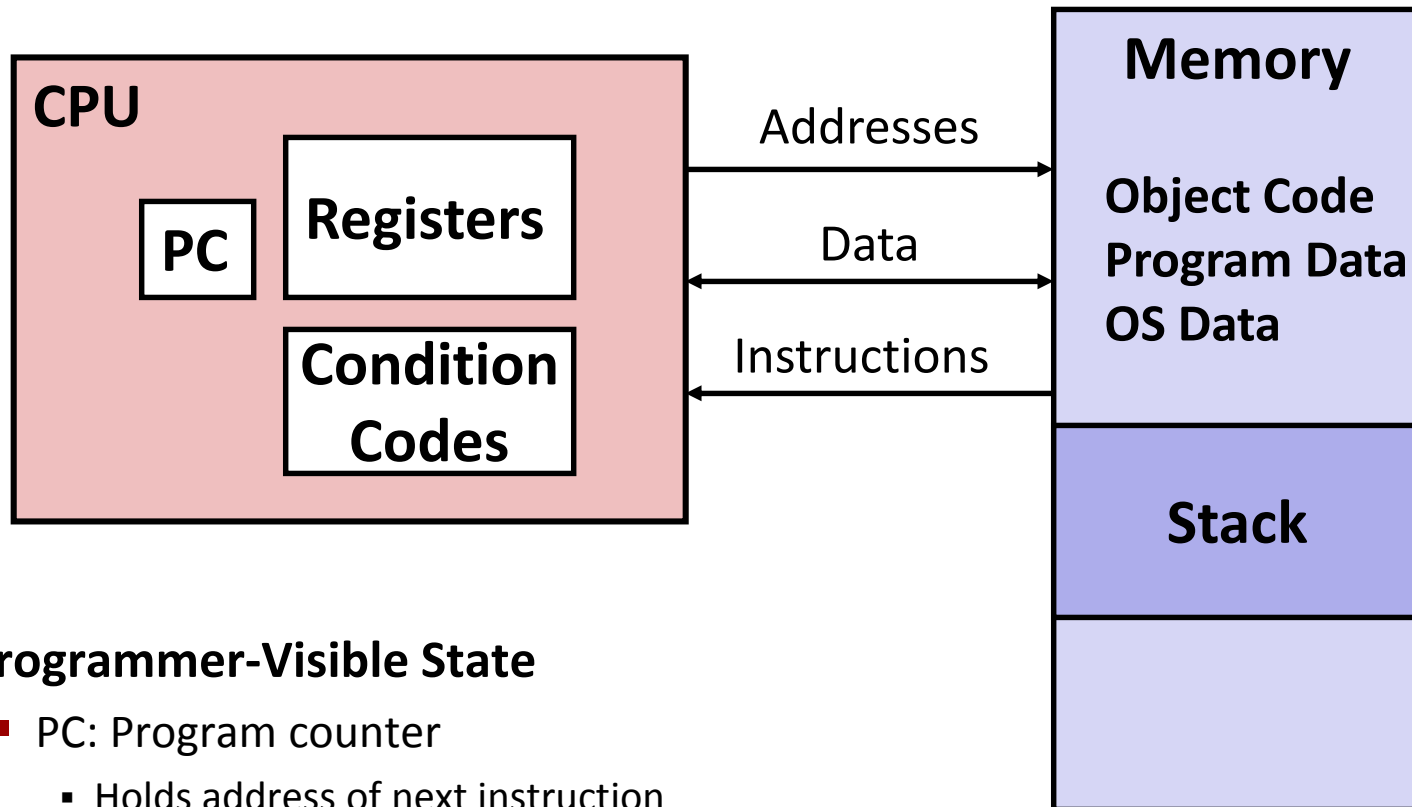
Machine Programming I – Basics

- **Instruction Set Architecture**
 - Software Architecture vs. Hardware Architecture
 - Common Architecture Classifications
- **The Intel x86 ISA – History and Microarchitectures**
- **Dive into C, Assembly, and Machine code**
- **The Intel x86 Assembly Basics:**
 - Registers and Operands
 - `mov` instruction
- **Intro to x86-64**
 - AMD was first!

Hardware vs. Software Architecture

- There are two parts to the computer architecture of a processor:
 - *Software architecture*
 - commonly know as the *Architecture* or *Instruction Set Architecture (ISA)*
 - *Hardware architecture*
 - commonly know as the *Microarchitecture*
- The *(software) architecture* includes all aspects of the design that are visible to programmers
- The *microarchitecture* refers to one specific implementation of a software architecture
 - e.g. number of cores, processor frequency, cache sizes, instructions supported, etc.
 - the set of all independent hardware architectures for a given software architecture is known as the *processor family*
 - e.g. the Intel x86 family

Assembly Programmer's View



■ Programmer-Visible State

- PC: Program counter
 - Holds address of next instruction
- Register file
 - Temp storage for program data
- Condition codes
 - Store status info about recent operation
 - Used for conditional branching

■ Memory

- Byte addressable array
- Code, user data, (some) OS data
- Includes stack used to support procedures

Separation of hardware and software

- The reason for the separation of the (software) **architecture** from the **microarchitecture** (hardware) is backwards compatibility
- **Backwards compatibility ensures:**
 - software written on older processors will run on newer processors (of the same ISA)
 - processor families can always utilize the latest technology by creating new hardware architectures (for the same ISA)
- However, new **microarchitectures** often add to the (software) **architecture**, so software written on newer processors may not run on older processors

Parts of the Software Architecture

- **There are 4 parts to the (software) architecture**
 - **instruction set**
 - the set of available instructions and the rules for using them
 - **register file organization**
 - the number, size, and rules for using registers
 - **memory organization & addressing**
 - the organization of the memory and the rules for accessing data
 - **operating modes**
 - the various modes of execution for the processor
 - there are usually at least two modes:
 - *user* mode (for general use)
 - *system* mode (allows access to privileged instructions and memory)

Software Architecture: Instruction Set

■ The *Instruction Set* defines

- the set of available instructions
- fundamental nature of the instructions
 - simple and fast
 - complex and concise
- instruction formats
 - define the rules for using the instructions
- the width (in bits) of the datapath
 - this defines the fundamental size of data in the CPU, including:
 - the size (number of bits) for the data buses in the CPU
 - the number of bits per register in the register file
 - the width of the processing units
 - the number of address bits for accessing memory

Software Architecture: Instruction Set

■ There are 9 fundamental categories of instructions

- arithmetic
 - these instructions perform integer arithmetic, such as add, subtract, multiply, and negate
 - Note: integer division is commonly done in software
- logical
 - these instructions perform Boolean logic (AND, OR, NOT, etc.)
- relational
 - these instructions perform comparisons, including
==, !=, <, >, <=, >=
 - some ISAs perform comparisons in the conditional branches
- control
 - these instructions enable changes in control flow, both for decision making and modularity
 - the set of control instructions includes:
 - conditional branches
 - unconditional jumps
 - procedure calls and returns

Software Architecture: Instruction Set

- memory
 - these instructions allow data to be read from or written to memory
- floating-point
 - these instruction perform real-number operations, including add, subtract, multiply, division, comparisons, and conversions
- shifts
 - these instructions allow bits to be shifted or rotated left or right
- bit manipulation
 - these instructions allow data bits to be set or cleared
 - some ISAs do not provide these, since they can be done via logic instructions
- system instructions
 - specialized instructions for system control purposes, such as
 - STOP or HALT (stop execution)
 - cache hints
 - interrupt handling
 - some of these instructions are privileged, requiring system mode

Software Architecture: Register File

- **The *Register File* is a small, fast temporary storage area in the processor's CPU**
 - it serves as the primary place for holding data values currently being operated upon by the CPU
- **The organization of the register file determines**
 - the number of registers
 - a large number of registers is desirable, but having too many will negatively impact processor speed
 - the number of bits per register
 - this is equivalent to the width of the datapath
 - the purpose of each register
 - ideally, most registers should be general-purpose
 - however, some registers serve specific purposes

Purpose of Register File

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
 - More instructions to be executed
- Compilers store values in registers whenever possible
 - Only spill to memory for less frequently used variables
 - Register optimization is important!

Software Architecture: Memory

- **The *Memory Organization & Addressing* defines**
 - **how memory is organized in the architecture**
 - where data and program memory are unified or separate
 - the amount of addressable memory
 - usually determined by the datapath width
 - the number of bytes per address
 - most processors are *byte-addressable*, so each byte has a unique address
 - whether it employs virtual memory, or just physical memory
 - virtual memory is usually required in complex computer systems, like desktops, laptops, servers, tablets, smart phones, etc.
 - simpler systems use embedded processors with only physical memory
 - **rules identifying how instructions access data in memory**
 - what instructions may access memory (usually only loads, stores)
 - what addressing modes are supported
 - the ordering and alignment rules for multi-byte primitive data types

Software Architecture: Operating Modes

- ***Operating Modes*** define the processor's modes of execution
- **The ISA typically supports at least two operating modes**
 - **user mode**
 - this is the mode of execution for typical use
 - **system mode**
 - allows access to privileged instructions and memory
 - aside from interrupt and exception handling, system mode is typically only available to system programmers and administrators
- Processors also generally have hardware testing modes, but these are usually part of the microarchitecture, not the (software) architecture

Machine Programming I – Basics

■ Instruction Set Architecture

- Software Architecture vs. Hardware Architecture
- Common Architecture Classifications

■ The Intel x86 ISA – History and Microarchitectures

■ Dive into C, Assembly, and Machine code

■ The Intel x86 Assembly Basics:

- Registers
- Operands
- `mov` instruction

■ Intro to x86-64

- AMD was first!

Common Architecture (ISA) Classifications:

■ Concise vs. Fast: CISC vs. RISC

■ *CISC – Complex Instruction Set Computers*

- complex instructions targeting efficient program representation
- variable-length instructions
- versatile addressing modes
- specialized instructions and registers implement complex tasks
- NOT optimized for speed – tend to be SLOW

■ *RISC – Reduced Instruction Set Computers*

- small set of simple instructions targeting high speed implementation
- fixed-length instructions
- simple addressing modes
- many general-purpose registers
- leads to FAST hardware implementations
- but less memory efficient

Classifications: Unified vs. Separate Memory

■ von Neumann vs. Harvard architecture

- relates to whether program and data in unified or separate memory
- *von Neumann* architecture
 - program and data are stored in the same unified memory space
 - requires only one physical memory
 - allows self-modifying code
 - however, code and data must share the same memory bus
 - used by most general-purpose processors (e.g. Intel x86)
- *Harvard* architecture
 - program and data are stored in separate memory spaces
 - requires separate physical memory
 - code and data do not share same bus, giving higher bandwidths
 - often used by digital signal processors for data-intensive applications

Classifications: Performance vs. Specificity

■ Microprocessor vs. Microcontroller

■ *Microprocessor*

- processors designed for high-performance and flexibility in personal computers and other general purpose applications
- architectures target high performance through a combination of high speed and parallelism
- processor chip contains only CPU(s) and cache
- no peripherals included on-chip

■ *Microcontroller*

- processors designed for specific purposes in embedded systems
- only need performance sufficient to needs of that application
- processor chip generally includes:
 - a simple CPU
 - modest amounts of RAM and (Flash) ROM
 - appropriate peripherals needed for specific application
- also often need to meet low power and/or real-time requirements

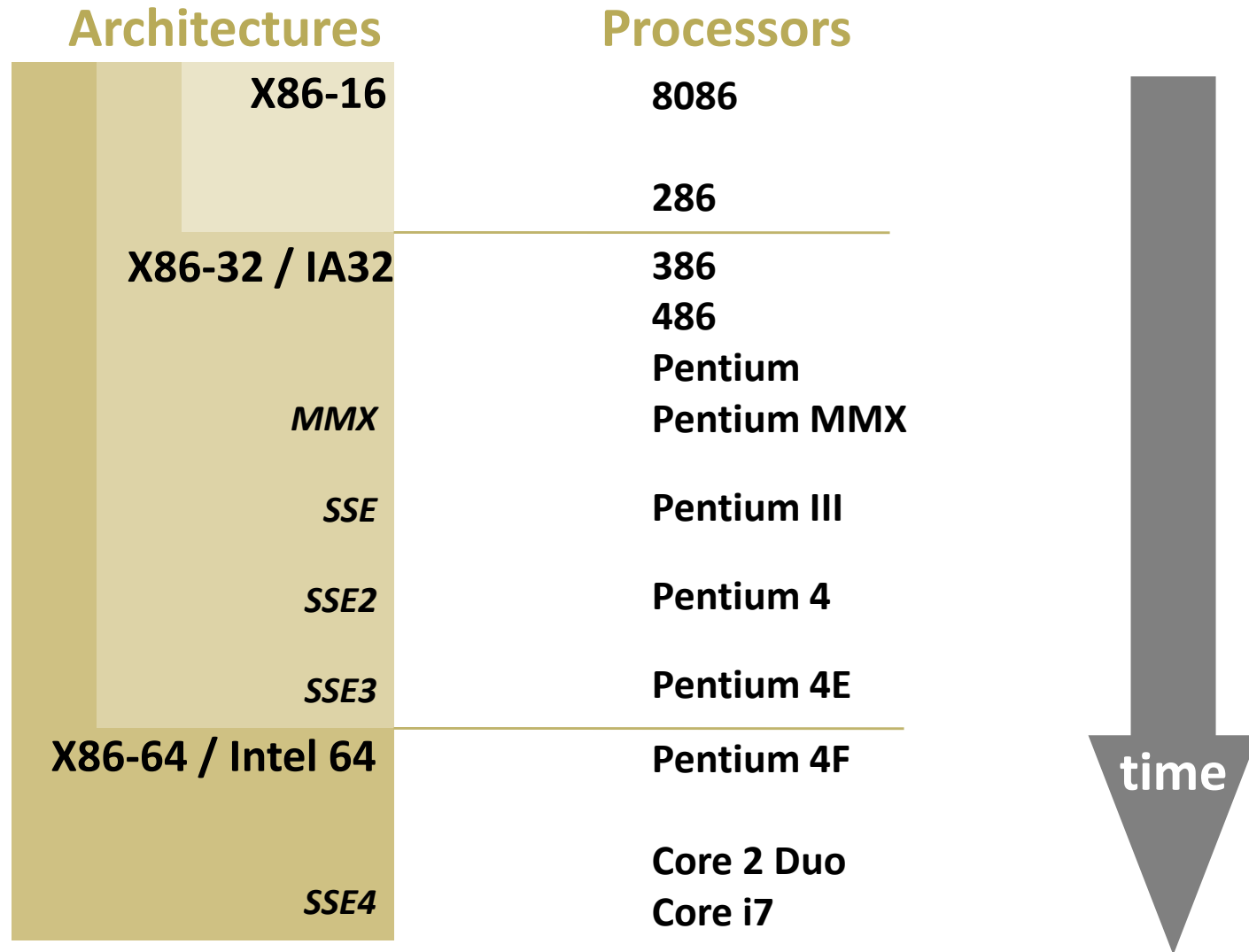
Machine Programming I – Basics

- **Instruction Set Architecture**
 - Software Architecture vs. Hardware Architecture
 - Common Architecture Classifications
- **The Intel x86 ISA – History and Microarchitectures**
- **Dive into C, Assembly, and Machine code**
- **The Intel x86 Assembly Basics:**
 - Registers
 - Operands
 - `mov` instruction
- **Intro to x86-64**
 - AMD was first!

Intel x86 Processors

- **The main software architecture for Intel is the x86 ISA**
 - also known as IA-32
 - for 64-bit processors, it is known as x86-64
- **Totally dominate laptop/desktop/server market**
- **Evolutionary design**
 - Backwards compatible back to 8086, introduced in 1978
 - Added more features as time goes on
- **Complex instruction set computer (CISC)**
 - Many different instructions with many different formats
 - *but, only small subset used in Linux programs*

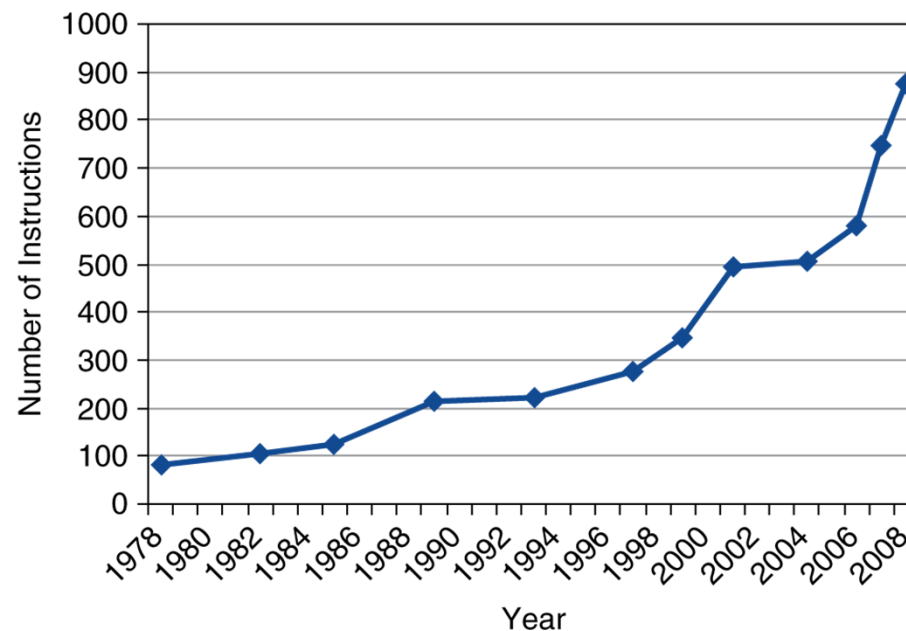
Intel x86 Family: Many Microarchitectures



IA: often redefined as latest Intel architecture

Software architecture can grow

- **Backward compatibility does not mean instruction set is fixed**
 - new instructions and functionality can be added to the software architecture over time
- **Intel added additional features over time**
 - Instructions to support multimedia operations (*MMX, SSE*)
 - SIMD parallelism – same operation done across multiple data
 - Instructions enabling more efficient conditional operations



x86 instruction set

Intel x86: Milestones & Trends

<u><i>Name</i></u>	<u><i>Date</i></u>	<u><i>Transistors</i></u>	<u><i>MHz</i></u>
■ 8086	1978	29K	5-10
■ First 16-bit processor. Basis for IBM PC & DOS ■ 1MB address space			
■ 386	1985	275K	16-33
■ First 32 bit processor, referred to as IA32 ■ Added “flat addressing”			
■ Pentium	1993	3.1M	50-75
■ Pentium II	1996	7.5M	233-300
■ Pentium III	1999	9.5-21M	450-800
■ Pentium 4F	2004	169M	3200-3800
■ First 64-bit processor ■ Got very hot			
■ Core i7	2008	731M	2667-3333

But IA-32 is CISC? How does it get speed?

- Hard to match RISC performance, but Intel has done just that!

....In terms of speed; less so for power

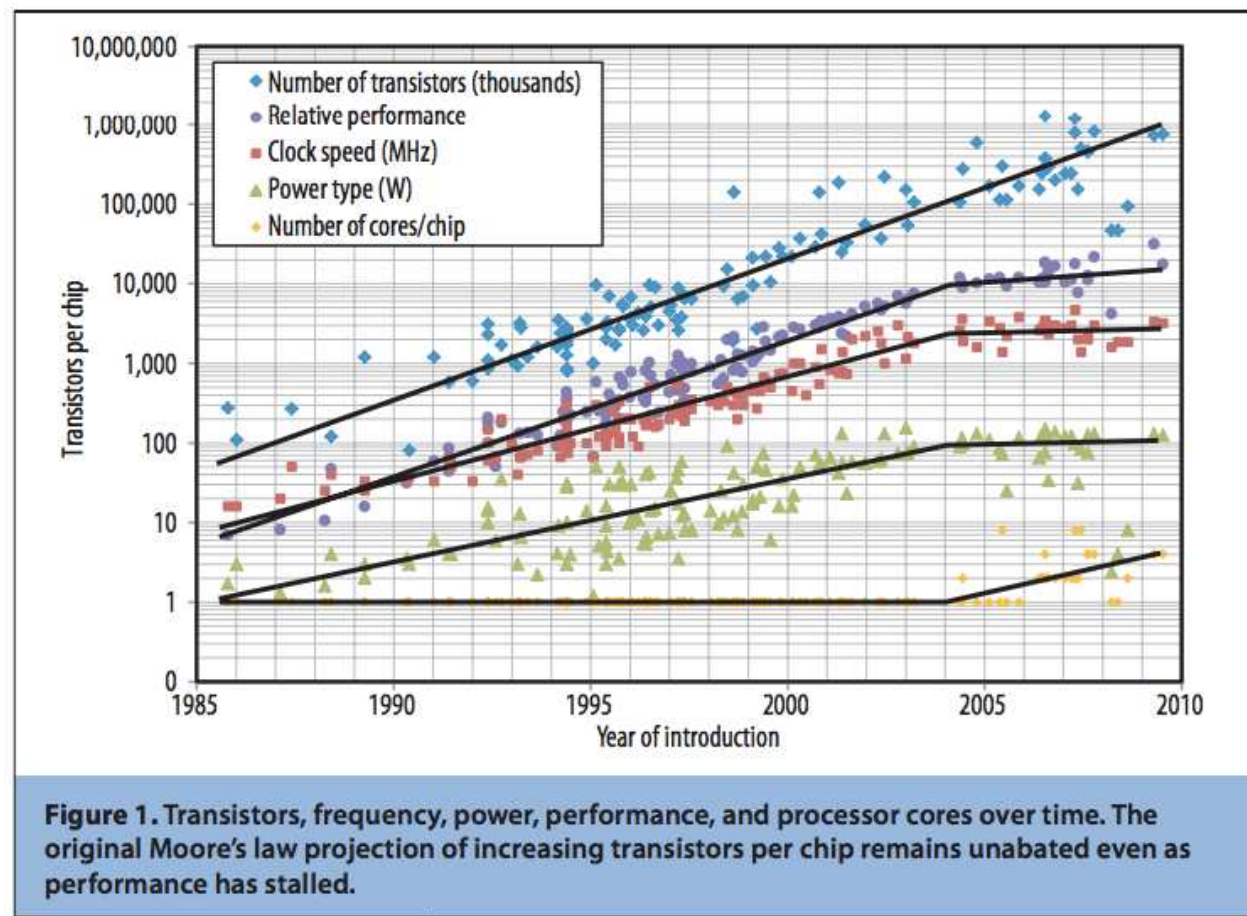
- CISC instruction set makes implementation difficult

- Hardware translates instructions to simpler *micro-operations*
 - simple instructions: 1-to-1
 - complex instructions: 1-to-many
- Micro-engine similar to RISC
- Market share makes this economically viable

- Comparable performance to RISC

- Compilers avoid CISC instructions

Processor Trends



- Number of transistors has continued to double every 2 years
- In 2004 – we hit the *Power Wall*
 - Processor clock speeds started to leveled off

Machine Programming I – Basics

- **Instruction Set Architecture**

Software Architecture (“Architecture” or “ISA”)

vs.

Hardware Architecture (“Microarchitecture”)

- **The Intel x86 ISA – History and Microarchitectures**

- **Dive into C, Assembly, and Machine code**

- **The Intel x86 Assembly Basics:**

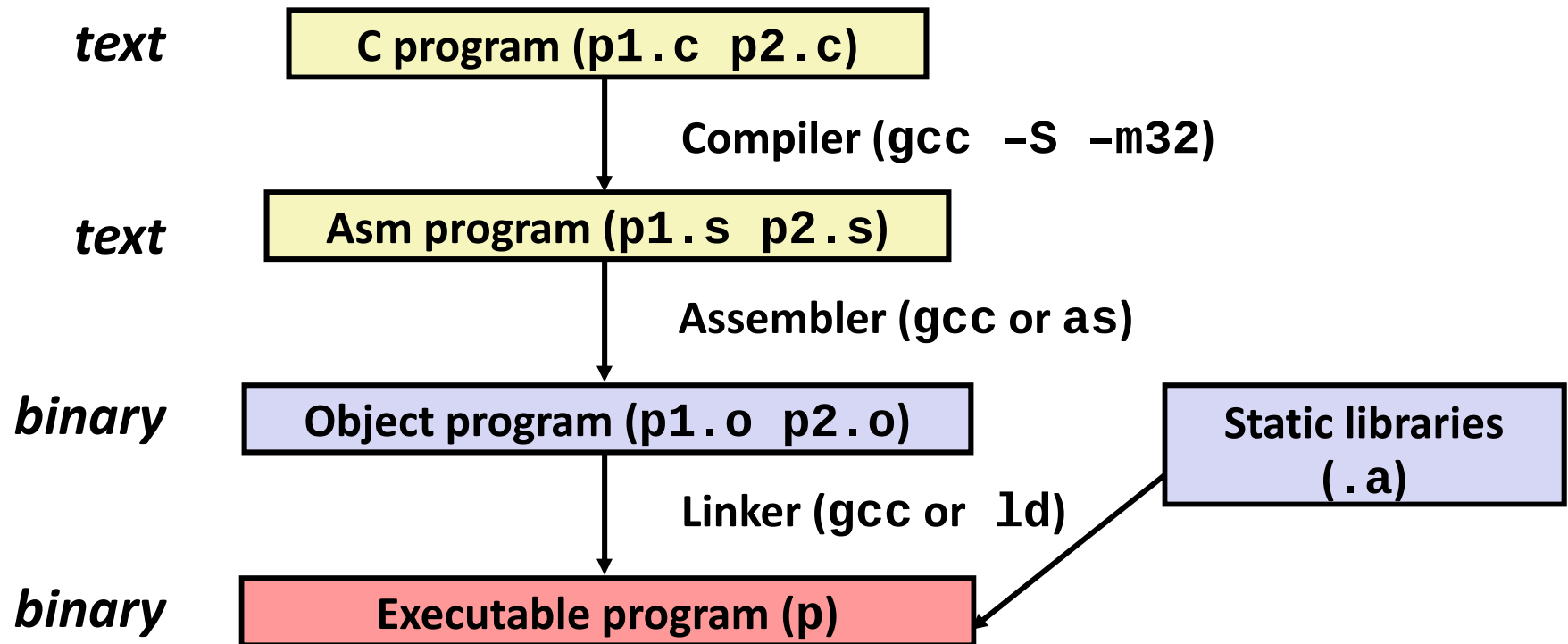
- Registers
- Operands
- `mov` instruction

- **Intro to x86-64**

- AMD was first!

Turning C into Object Code

- Code in files **p1.c p2.c**
- Compile with command: **gcc -O1 -m32 p1.c p2.c -o p**
 - Use basic optimizations (**-O1**)
 - Put resulting binary in file **p**
 - On 64-bit machines, specify 32-bit x86 code (**-m32**)

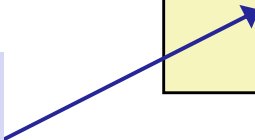


Compiling Into Assembly

C Code

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

Some compilers use
instruction "leave"



Generated IA32 Assembly

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

Obtain with command:

```
gcc -O1 -S -m32 code.c
```

-S specifies compile to assembly (vs object) code, and
produces file **code.s**

Assembly Characteristics: Simple Types

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

Assembly Characteristics: Operations

■ Perform some operation on register or memory data

- arithmetic
- logical
- bit shift or manipulation
- comparison (relational)

■ 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

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 **malloc**, **printf**
- Some libraries are *dynamically linked*
 - Linking occurs when program begins execution

Machine Instruction Example

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

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

Similar to expression:

```
x += y
```

More precisely:

```
int eax;
```

```
int *ebp;
```

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

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

■ C Code

- Add two signed integers

■ 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**

■ Object Code

- 3-byte instruction
- Stored at address **0x80483ca**

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 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

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 p`

`disassemble sum`

■ Disassemble procedure

`x/11xb sum`

■ Examine the 11 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

Machine Programming I – Basics

- **Instruction Set Architecture**
 - Software Architecture vs. Hardware Architecture
 - Common Architecture Classifications
- **The Intel x86 ISA – History and Microarchitectures**
- **Dive into C, Assembly, and Machine code**
- **The Intel x86 Assembly Basics:**
 - Common instructions
 - Registers, Operands, and **mov** instruction
 - Addressing modes
- **Intro to x86-64**
 - AMD was first!

Typical Instructions in Intel x86

■ Arithmetic

- add, sub, neg, imul, div, inc, dec, leal, ...

■ Logical (bit-wise Boolean)

- and, or, xor, not

■ Relational

- cmp, test, sete, ...

■ Control

- je, jle, jg, jb, jmp, call, ret, ...

■ Moves & Memory Access

- mov, push, pop, movswl, movzbl, cmov, ...
- *nearly all x86 instructions can access memory*

■ Shifts

- shr, sar, shl, sal (same as shl)

■ Floating-point

- fld, fadd, fsub, fxch, addsd, movss, cvt..., ucom...
- *float-point change completely with x86-64*

CISC Instructions: Variable-Length

a. JE EIP + displacement

4	4	8
JE	Condi- tion	Displacement

b. CALL

8	32
CALL	Offset

c. MOV EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	r/m Postbyte	Displacement

d. PUSH ESI

5	3
PUSH	Reg

e. ADD EAX, #6765

4	3	1	32
ADD	Reg	w	Immediate

f. TEST EDX, #42

7	1	8	32
TEST	w	Postbyte	Immediate

Machine Programming I – Basics

- **Instruction Set Architecture**

Software Architecture (“Architecture” or “ISA”)

vs.

Hardware Architecture (“Microarchitecture”)

- **The Intel x86 ISA – History and Microarchitectures**

- **Dive into C, Assembly, and Machine code**

- **The Intel x86 Assembly Basics:**

- Common instructions
- Registers, Operands, and **mov** instruction
- Addressing modes

- **Intro to x86-64**

- AMD was first!

Integer Registers (IA32)

general purpose	%eax	%ax	%ah	%al	Origin (mostly obsolete) <i>accumulate</i>
	%ecx	%cx	%ch	%cl	<i>counter</i>
	%edx	%dx	%dh	%dl	<i>data</i>
	%ebx	%bx	%bh	%bl	<i>base</i>
	%esi	%si			<i>source index</i>
	%edi	%di			<i>destination index</i>
	%esp	%sp			<i>stack pointer</i>
	%ebp	%bp			<i>base pointer</i>

16-bit virtual registers
(backwards compatibility)

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 in particular situations
- **Memory:** 4 consecutive bytes of memory at address given by register
 - simplest example: **(%eax)**
 - various other “address modes”

%eax

%ecx

%edx

%ebx

%esi

%edi

%esp

%ebp

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

Machine Programming I – Basics

- **Instruction Set Architecture**

Software Architecture (“Architecture” or “ISA”)

vs.

Hardware Architecture (“Microarchitecture”)

- **The Intel x86 ISA – History and Microarchitectures**

- **Dive into C, Assembly, and Machine code**

- **The Intel x86 Assembly Basics:**

- Common instructions
- Registers, Operands, and **mov** instruction
- Addressing modes

- **Intro to x86-64**

- AMD was first!

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

Using Simple Addressing Modes

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

swap:

pushl %ebx

} Set Up

movl 8(%esp), %edx
 movl 12(%esp), %eax
 movl (%edx), %ecx
 movl (%eax), %ebx
 movl %ebx, (%edx)
 movl %ecx, (%eax)

} Body

popl %ebx
 ret

} Finish

Using Simple Addressing Modes

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

swap:

pushl %ebx

} **Set Up**

movl 8(%esp), %edx
movl 12(%esp), %eax
movl (%edx), %ecx
movl (%eax), %ebx
movl %ebx, (%edx)
movl %ecx, (%eax)

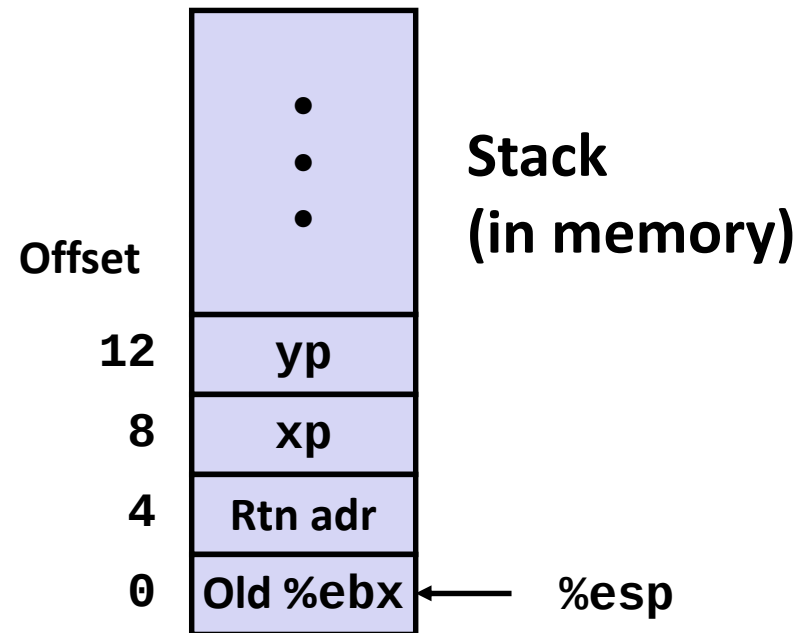
} **Body**

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

Understanding Swap

%eax	
%edx	
%ecx	
%ebx	
%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
%esp	→ 0		0x104

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

Understanding Swap

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

		Address
		0x124
		0x120
		0x11c
		0x118
		0x114
yp	12	0x120
xp	8	0x10c
	4	Rtn adr
%esp	0	0x104

```

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

```

Understanding Swap

%eax	0x120
%edx	0x124
%ecx	
%ebx	
%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
%esp	→ 0		0x104

```

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

```

Understanding Swap

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

		Address
		0x124
		0x120
		0x11c
		0x118
		0x114
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%esp	0	0x104

```

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

```

Understanding Swap

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

		Address
		0x124
		0x120
		0x11c
		0x118
		0x114
yp	12	0x110
xp	8	0x10c
	4	0x108
%esp	0	0x104

```

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

```

Understanding Swap

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

		Address
	Offset	
		0x124
		0x120
		0x11c
		0x118
		0x114
yp	12	0x110
xp	8	0x10c
	4	Rtn adr
%esp	0	0x104

```

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

```

Understanding Swap

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

		Address
		456
		0x124
		123
		0x120
		0x11c
		0x118
		0x114
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%esp	0	0x104

```

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

```

Complete Memory Addressing Modes

■ Most General Form

$$D(Rb, Ri, S) \qquad \text{Mem}[\text{Reg}[Rb] + S * \text{Reg}[Ri] + D]$$

- *D*: Constant “displacement” 1, 2, or 4 bytes
- *Rb*: Base register: Any of 8 integer registers
- *Ri*: Index register: Any, except for **%esp** (likely not **%ebp** either)
- *S*: Scale: 1, 2, 4, or 8 (*why these numbers?*)

■ Special Cases

$$\begin{array}{ll} (Rb, Ri) & \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]] \\ D(Rb, Ri) & \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D] \\ (Rb, Ri, S) & \text{Mem}[\text{Reg}[Rb] + S * \text{Reg}[Ri]] \end{array}$$

Basic x86 Addressing Modes

- **Memory addressing modes**
 - **Address in register**
 - **Address = R_{base} + displacement**
 - **Address = R_{base} + R_{index}**
 - **Address = R_{base} + $2^{\text{scale}} \times R_{\text{index}}$ (scale = 0, 1, 2, or 3)**
 - **Address = R_{base} + $2^{\text{scale}} \times R_{\text{index}}$ + displacement**

Machine Programming I – Basics

- **Instruction Set Architecture**
 - Software Architecture vs. Hardware Architecture
 - Common Architecture Classifications
- **The Intel x86 ISA – History and Microarchitectures**
- **Dive into C, Assembly, and Machine code**
- **The Intel x86 Assembly Basics:**
 - Common instructions
 - Registers, Operands, and **mov** instruction
 - Addressing modes
- **Intro to x86-64**
 - AMD was first!

AMD created first 64-bit version of x86

■ Historically

- AMD has followed just behind Intel
- A little bit slower, a lot cheaper

■ 2003, developed 64-bit version of x86: **x86-64**

- Recruited top circuit designers from DEC and other diminishing companies
- Built Opteron: tough competitor to Pentium 4

Intel's 64-Bit

- **Intel Attempted Radical Shift from IA32 to IA64**
 - Totally different architecture (*Itanium*)
 - Executes IA32 code only as legacy
 - Performance disappointing
- **2003: AMD Stepped in with Evolutionary Solution**
 - Originally called **x86-64** (now called **AMD64**)
- **2004: Intel Announces their 64-bit extension to IA32**
 - Originally called **EMT64** (now called **Intel 64**)
 - Almost identical to **x86-64**!
- **Collectively known as **x86-64****
 - minor differences between the two

Data Representations: IA32 vs. x86-64

■ Sizes of C Objects (in bytes)

<u><i>C Data Type</i></u>	<u><i>Intel IA32</i></u>	<u><i>x86-64</i></u>
▪ unsigned	4	4
▪ int	4	4
▪ long int	4	8
▪ char	1	1
▪ short	2	2
▪ float	4	4
▪ double	8	8
▪ long double	10/12	16
▪ pointer (e.g. char *)	4	8

x86-64 Integer Registers

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

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

New Instructions for 64-bit Operands

- Long word **l** (4 Bytes) $\leftrightarrow$ Quad word **q** (8 Bytes)

- New instructions:
 - `movl` $\rightarrow$ `movq`
 - `addl` $\rightarrow$ `addq`
 - `sall` $\rightarrow$ `salq`
 - etc.

- 32-bit instructions that generate 32-bit results
 - Set higher order bits of destination register to 0
 - Example: `addl`

32-bit code for `int` swap

swap:

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

`pushl %ebx`

} Set
Up

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

} Body

`popl %ebx`
`ret`

} Finish

64-bit code for `int` swap

swap:

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

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

ret

} Set Up

} Body

} Finish

- **Operands passed in registers (why useful?)**
 - First input arg (**xp**) in **%rdi**, second input arg (**yp**) in **%rsi**
 - 64-bit pointers
- **No stack operations required**
- **32-bit `int`s held temporarily in **%eax** and **%edx****

64-bit code for `long int swap`

`swap_1:`

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

```
movq    (%rdi), %rdx
movq    (%rsi), %rax
movq    %rax, (%rdi)
movq    %rdx, (%rsi)
```

`ret`

} Set Up

} Body

} Finish

■ 64-bit `long ints`

- Pass input arguments in registers `%rax` and `%rdx`
- **`movq`** operation
 - “q” stands for quad-word

Machine Programming I – Basics

- **Instruction Set Architecture**
 - Software Architecture vs. Hardware Architecture
 - Common Architecture Classifications
- **The Intel x86 ISA – History and Microarchitectures**
- **Dive into C, Assembly, and Machine code**
- **The Intel x86 Assembly Basics:**
 - Common instructions
 - Registers, Operands, and **mov** instruction
 - Addressing modes
- **Intro to x86-64**
 - AMD was first!

Machine Programming I – Summary

■ Instruction Set Architecture

- Many different varieties and features of processor architectures
- Separation of (software) Architecture and Microarchitecture is key for backwards compatibility

■ The Intel x86 ISA – History and Microarchitectures

- Evolutionary design leads to many quirks and artifacts

■ Dive into C, Assembly, and Machine code

- Compiler must transform statements, expressions, procedures into low-level instruction sequences

■ The Intel x86 Assembly Basics:

- The x86 move instructions cover wide range of data movement forms

■ Intro to x86-64

- A major departure from the style of code seen in IA32

Stored Program Computers

- **Instructions represented in binary, just like data**
- **Instructions and data stored in memory**
- **Programs can operate on programs**
 - e.g., compilers, linkers, ...
- **Binary compatibility allows compiled programs to work on different computers**
 - Standardized ISAs

Basic x86 Registers

Name	31	0	Use
EAX			GPR 0
ECX			GPR 1
EDX			GPR 2
EBX			GPR 3
ESP			GPR 4
EBP			GPR 5
ESI			GPR 6
EDI			GPR 7
	CS		Code segment pointer
	SS		Stack segment pointer (top of stack)
	DS		Data segment pointer 0
	ES		Data segment pointer 1
	FS		Data segment pointer 2
	GS		Data segment pointer 3
EIP			Instruction pointer (PC)
EFLAGS			Condition codes