

Machine-Level Programming I: Basics

15-213/18-213: Introduction to Computer Systems

5th Lecture, Sep. 15, 2015

Instructors:

Randal E. Bryant and David R. O'Hallaron

Today: Machine Programming I: Basics

- **History of Intel processors and architectures**
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

Intel x86 Processors

- **Dominate laptop/desktop/server market**

- **Evolutionary design**
 - Backwards compatible up until 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 encountered with Linux programs
 - Hard to match performance of Reduced Instruction Set Computers (RISC)
 - But, Intel has done just that!
 - In terms of speed. Less so for low power.

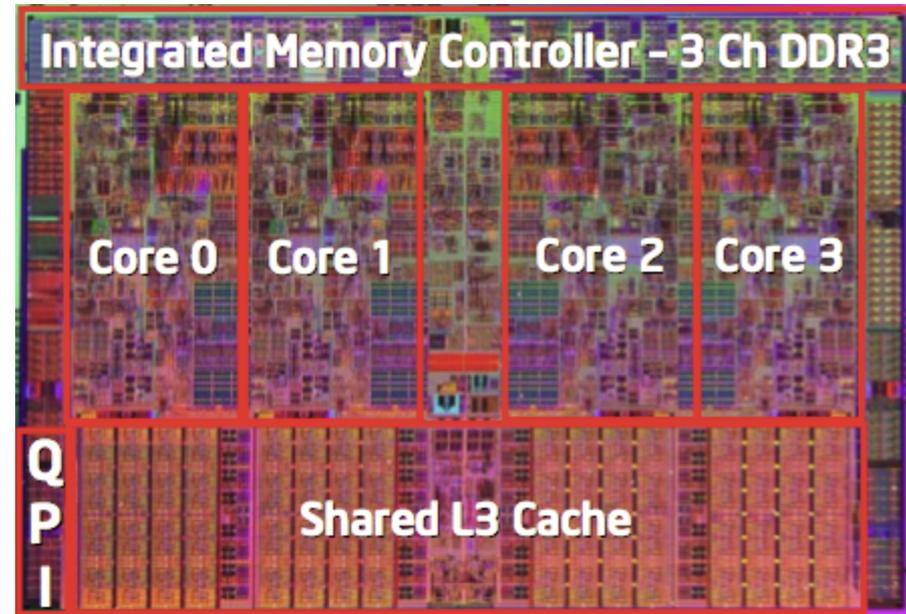
Intel x86 Evolution: Milestones

<i>Name</i>	<i>Date</i>	<i>Transistors</i>	<i>MHz</i>
■ 8086	1978	29K	5-10
<ul style="list-style-type: none"> ▪ First 16-bit Intel processor. Basis for IBM PC & DOS ▪ 1MB address space 			
■ 386	1985	275K	16-33
<ul style="list-style-type: none"> ▪ First 32 bit Intel processor , referred to as IA32 ▪ Added “flat addressing”, capable of running Unix 			
■ Pentium 4E	2004	125M	2800-3800
<ul style="list-style-type: none"> ▪ First 64-bit Intel x86 processor, referred to as x86-64 			
■ Core 2	2006	291M	1060-3500
<ul style="list-style-type: none"> ▪ First multi-core Intel processor 			
■ Core i7	2008	731M	1700-3900
<ul style="list-style-type: none"> ▪ Four cores (our shark machines) 			

Intel x86 Processors, cont.

■ Machine Evolution

■ 386	1985	0.3M
■ Pentium	1993	3.1M
■ Pentium/MMX	1997	4.5M
■ PentiumPro	1995	6.5M
■ Pentium III	1999	8.2M
■ Pentium 4	2001	42M
■ Core 2 Duo	2006	291M
■ Core i7	2008	731M



■ Added Features

- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 bits to 64 bits
- More cores

2015 State of the Art

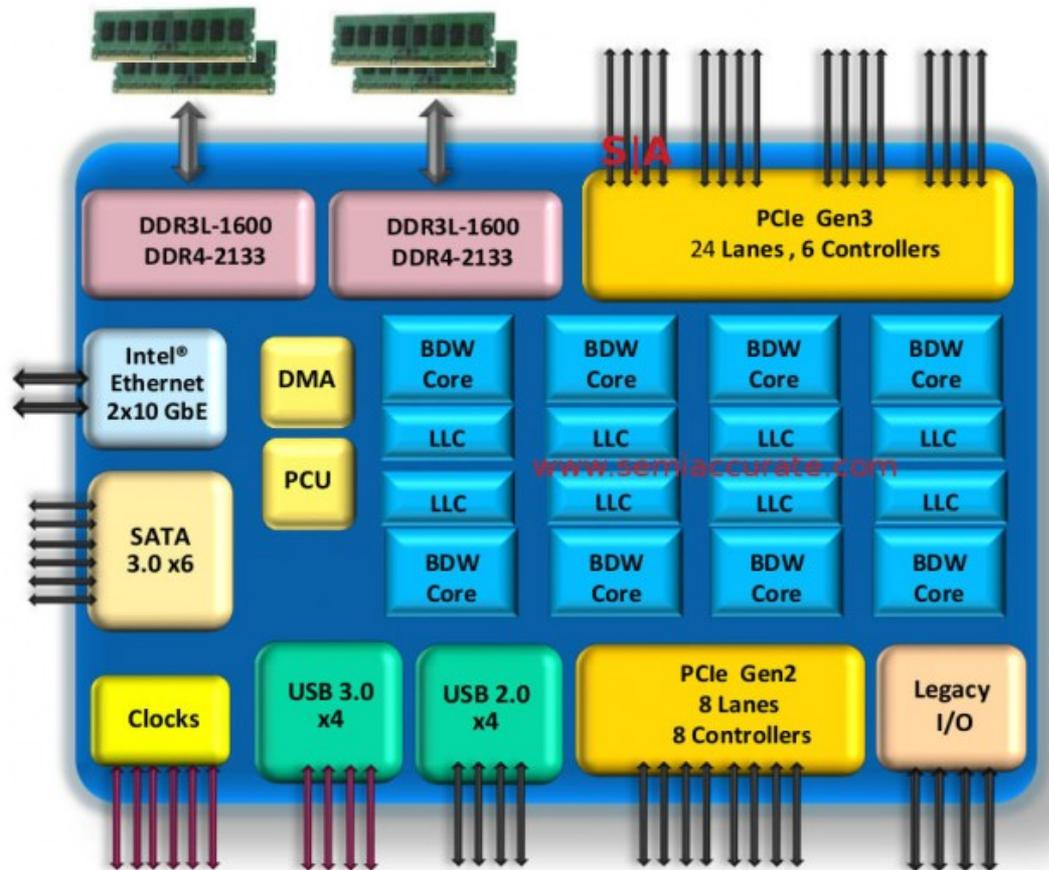
- Core i7 Broadwell 2015

■ Desktop Model

- 4 cores
- Integrated graphics
- 3.3-3.8 GHz
- 65W

■ Server Model

- 8 cores
- Integrated I/O
- 2-2.6 GHz
- 45W



x86 Clones: Advanced Micro Devices (AMD)

■ Historically

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

■ Then

- Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
- Built Opteron: tough competitor to Pentium 4
- Developed x86-64, their own extension to 64 bits

■ Recent Years

- Intel got its act together
 - Leads the world in semiconductor technology
- AMD has fallen behind
 - Relies on external semiconductor manufacturer

Intel's 64-Bit History

- **2001: Intel Attempts Radical Shift from IA32 to IA64**
 - Totally different architecture (Itanium)
 - Executes IA32 code only as legacy
 - Performance disappointing
- **2003: AMD Steps in with Evolutionary Solution**
 - x86-64 (now called “AMD64”)
- **Intel Felt Obligated to Focus on IA64**
 - Hard to admit mistake or that AMD is better
- **2004: Intel Announces EM64T extension to IA32**
 - Extended Memory 64-bit Technology
 - Almost identical to x86-64!
- **All but low-end x86 processors support x86-64**
 - But, lots of code still runs in 32-bit mode

Our Coverage

■ IA32

- The traditional x86
- For 15/18-213: RIP, Summer 2015

■ x86-64

- The standard
- `shark> gcc hello.c`
- `shark> gcc -m64 hello.c`

■ Presentation

- Book covers x86-64
- Web aside on IA32
- We will only cover x86-64

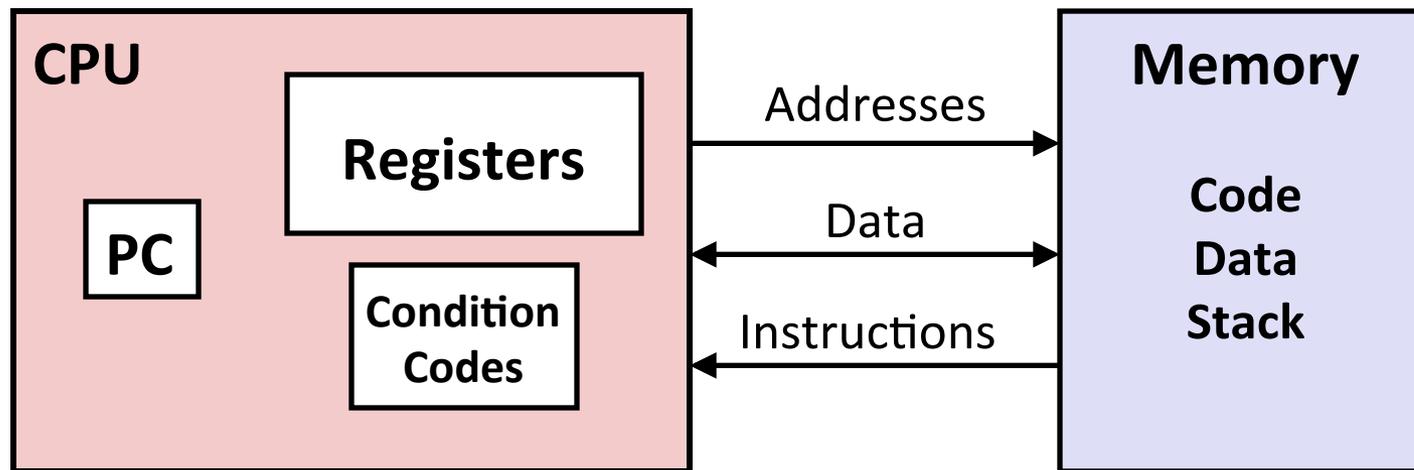
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- **C, assembly, machine code**
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

Definitions

- **Architecture:** (also ISA: instruction set architecture) The parts of a processor design that one needs to understand or write assembly/machine code.
 - Examples: instruction set specification, registers.
- **Microarchitecture:** Implementation of the architecture.
 - Examples: cache sizes and core frequency.
- **Code Forms:**
 - **Machine Code:** The byte-level programs that a processor executes
 - **Assembly Code:** A text representation of machine code
- **Example ISAs:**
 - Intel: x86, IA32, Itanium, x86-64
 - ARM: Used in almost all mobile phones

Assembly/Machine Code View

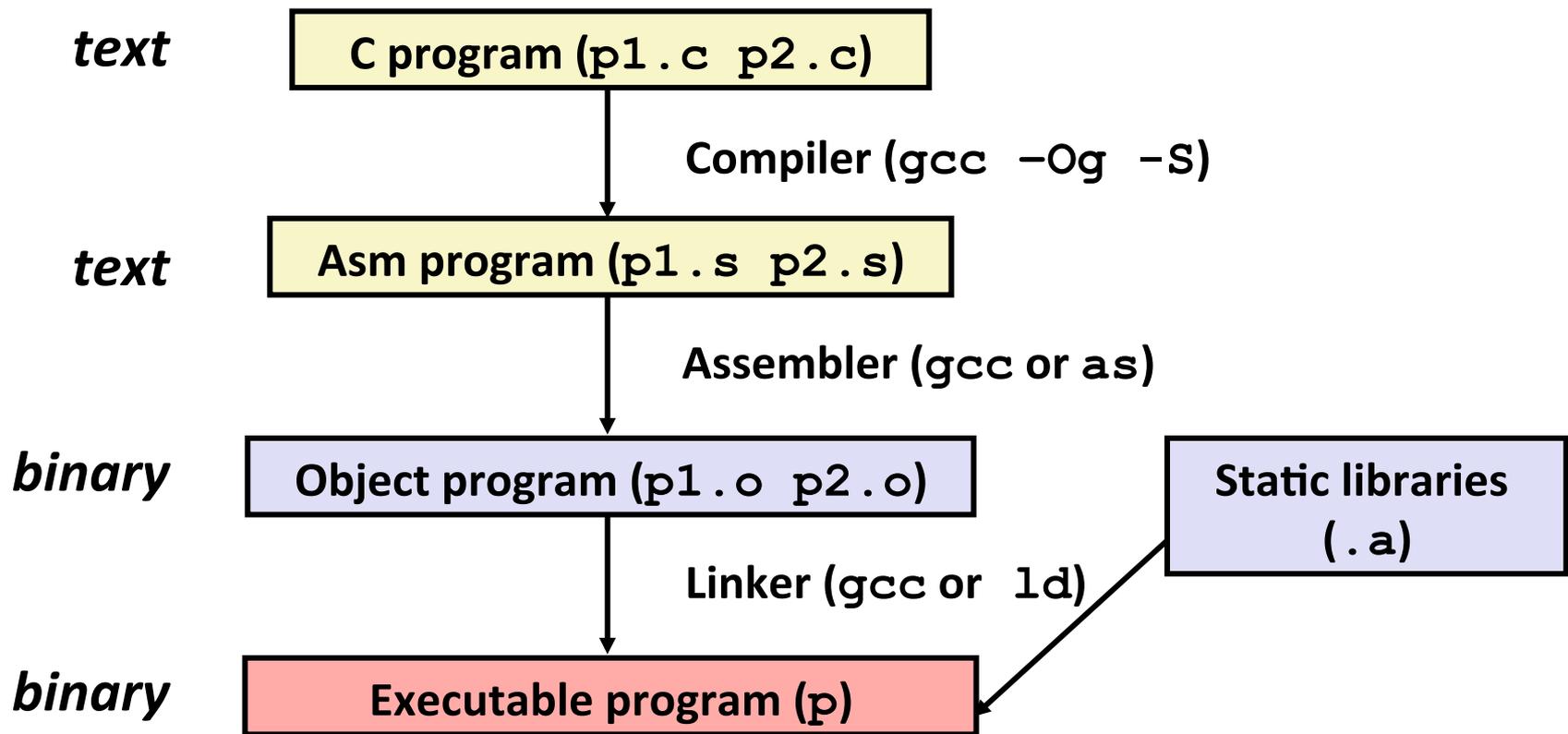


Programmer-Visible State

- **PC: Program counter**
 - Address of next instruction
 - Called “RIP” (x86-64)
- **Register file**
 - Heavily used program data
- **Condition codes**
 - Store status information about most recent arithmetic or logical operation
 - Used for conditional branching
- **Memory**
 - Byte addressable array
 - Code and user data
 - Stack to support procedures

Turning C into Object Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
 - Use basic optimizations (`-Og`) [New to recent versions of GCC]
 - Put resulting binary in file `p`



Compiling Into Assembly

C Code (sum.c)

```
long plus(long x, long y);  
  
void sumstore(long x, long y,  
              long *dest)  
{  
    long t = plus(x, y);  
    *dest = t;  
}
```

Generated x86-64 Assembly

```
sumstore:  
    pushq    %rbx  
    movq    %rdx, %rbx  
    call    plus  
    movq    %rax, (%rbx)  
    popq    %rbx  
    ret
```

Obtain (on shark machine) with command

```
gcc -Og -S sum.c
```

Produces file `sum.s`

Warning: Will get very different results on non-Shark machines (Andrew Linux, Mac OS-X, ...) due to different versions of gcc and different compiler settings.

Assembly Characteristics: Data Types

- **“Integer” data of 1, 2, 4, or 8 bytes**
 - Data values
 - Addresses (untyped pointers)
- **Floating point data of 4, 8, or 10 bytes**
- **Code: Byte sequences encoding series of instructions**
- **No aggregate types such as arrays or structures**
 - Just contiguously allocated bytes in memory

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

Object Code

Code for `sumstore`

0x0400595:

0x53

0x48

0x89

0xd3

0xe8

0xf2

0xff

0xff

0xff

0x48

0x89

0x03

0x5b

0xc3

- **Total of 14 bytes**
- **Each instruction 1, 3, or 5 bytes**
- **Starts at address 0x0400595**

■ 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

```
*dest = t;
```

```
movq %rax, (%rbx)
```

```
0x40059e: 48 89 03
```

■ C Code

- Store value `t` where designated by `dest`

■ Assembly

- Move 8-byte value to memory
 - Quad words in x86-64 parlance
- Operands:
 - `t`: Register `%rax`
 - `dest`: Register `%rbx`
 - `*dest`: Memory `M[%rbx]`

■ Object Code

- 3-byte instruction
- Stored at address `0x40059e`

Disassembling Object Code

Disassembled

```
0000000000400595 <sumstore>:  
400595: 53                push   %rbx  
400596: 48 89 d3         mov    %rdx,%rbx  
400599: e8 f2 ff ff ff   callq 400590 <plus>  
40059e: 48 89 03         mov    %rax, (%rbx)  
4005a1: 5b                pop    %rbx  
4005a2: c3                retq
```

■ Disassembler

```
objdump -d sum
```

- 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

0x0400595:

0x53

0x48

0x89

0xd3

0xe8

0xf2

0xff

0xff

0xff

0x48

0x89

0x03

0x5b

0xc3

Disassembled

```
Dump of assembler code for function sumstore:
0x0000000000400595 <+0>: push    %rbx
0x0000000000400596 <+1>: mov     %rdx,%rbx
0x0000000000400599 <+4>: callq  0x400590 <plus>
0x000000000040059e <+9>: mov     %rax, (%rbx)
0x00000000004005a1 <+12>: pop    %rbx
0x00000000004005a2 <+13>: retq
```

■ Within gdb Debugger

```
gdb sum
```

```
disassemble sumstore
```

- Disassemble procedure

```
x/14xb sumstore
```

- Examine the 14 bytes starting at `sumstore`

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:
```

```
30001001:
```

```
30001003:
```

```
30001005:
```

```
3000100a:
```

**Reverse engineering forbidden by
Microsoft End User License Agreement**

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

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- **Assembly Basics: Registers, operands, move**
- Arithmetic & logical operations

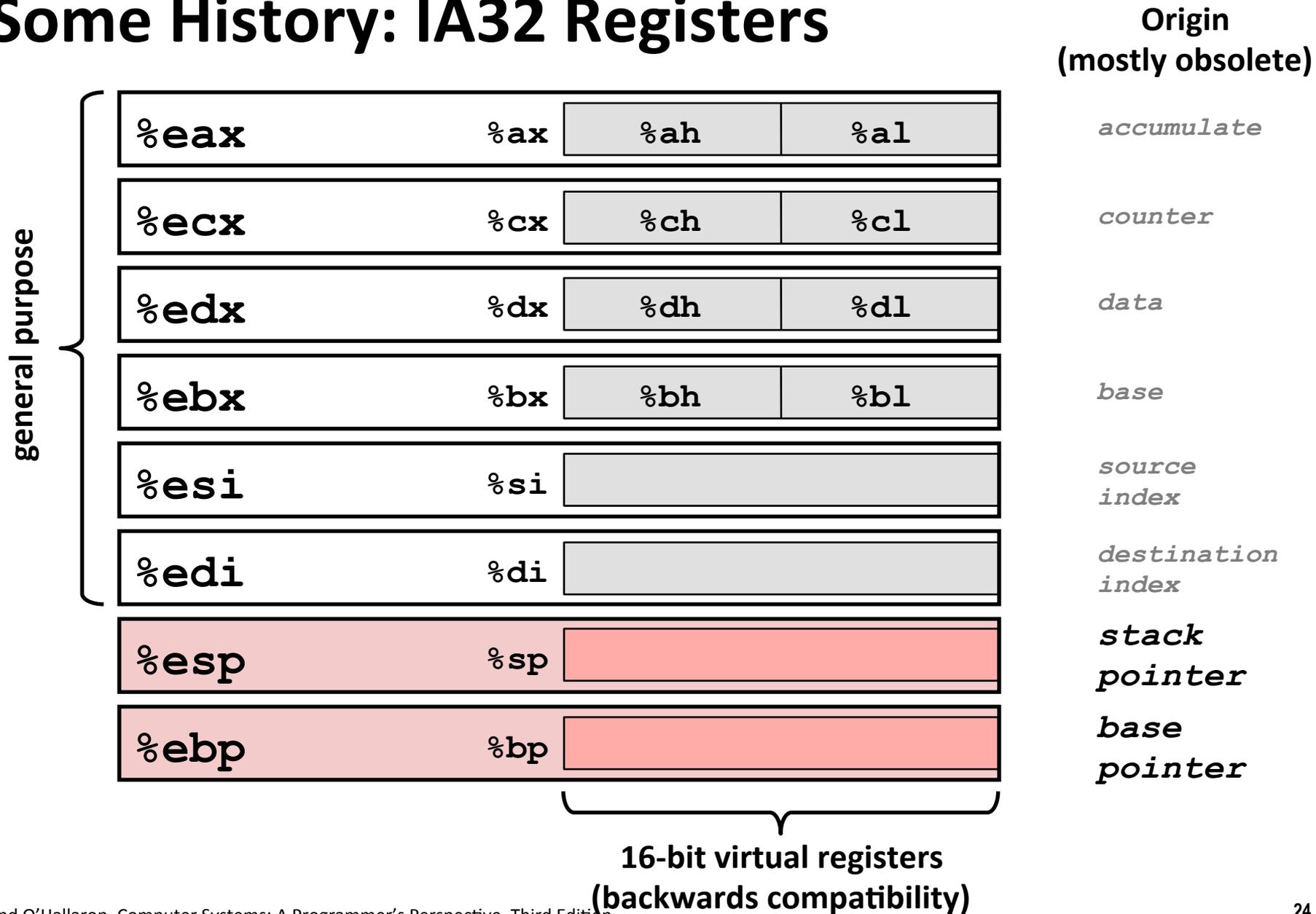
x86-64 Integer Registers

<code>%rax</code>	<code>%eax</code>
<code>%rbx</code>	<code>%ebx</code>
<code>%rcx</code>	<code>%ecx</code>
<code>%rdx</code>	<code>%edx</code>
<code>%rsi</code>	<code>%esi</code>
<code>%rdi</code>	<code>%edi</code>
<code>%rsp</code>	<code>%esp</code>
<code>%rbp</code>	<code>%ebp</code>

<code>%r8</code>	<code>%r8d</code>
<code>%r9</code>	<code>%r9d</code>
<code>%r10</code>	<code>%r10d</code>
<code>%r11</code>	<code>%r11d</code>
<code>%r12</code>	<code>%r12d</code>
<code>%r13</code>	<code>%r13d</code>
<code>%r14</code>	<code>%r14d</code>
<code>%r15</code>	<code>%r15d</code>

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

Some History: IA32 Registers



Moving Data

■ Moving Data

`movq 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 16 integer registers
 - Example: `%rax`, `%r13`
 - But `%rsp` reserved for special use
 - Others have special uses for particular instructions
- **Memory:** 8 consecutive bytes of memory at address given by register
 - Simplest example: `(%rax)`
 - Various other “address modes”

`%rax`

`%rcx`

`%rdx`

`%rbx`

`%rsi`

`%rdi`

`%rsp`

`%rbp`

`%rN`

movl Operand Combinations

	Source	Dest	Src, Dest	C Analog
movq	Imm	Reg	movq \$0x4, %rax	temp = 0x4;
		Mem	movq \$-147, (%rax)	*p = -147;
	Reg	Reg	movq %rax, %rdx	temp2 = temp1;
		Mem	movq %rax, (%rdx)	*p = temp;
	Mem	Reg	movq (%rax), %rdx	temp = *p;

Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes

■ Normal (R) Mem[Reg[R]]

- Register R specifies memory address
- Aha! Pointer dereferencing in C

```
movq (%rcx), %rax
```

■ Displacement D(R) Mem[Reg[R]+D]

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

```
movq 8(%rbp), %rdx
```

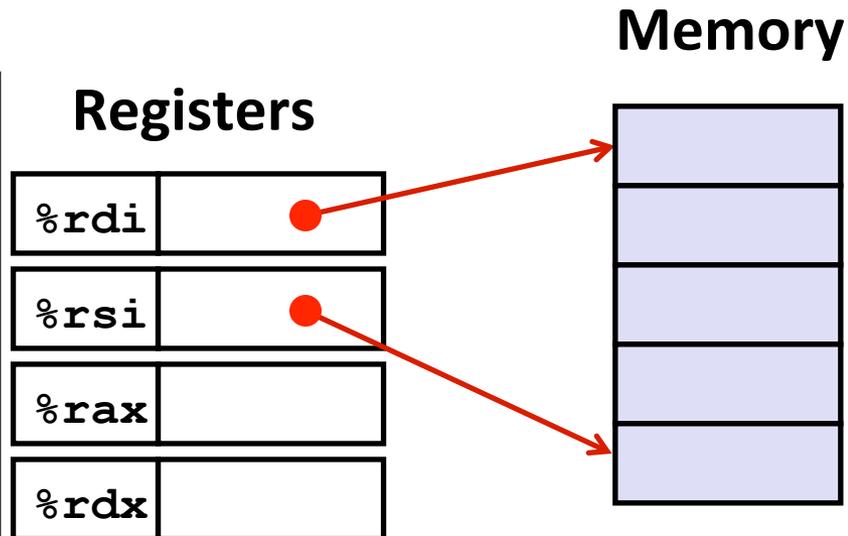
Example of Simple Addressing Modes

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

```
swap:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```

Understanding Swap()

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



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)   # *xp = t1
movq    %rax, (%rsi)   # *yp = t0
ret
```

Understanding Swap()

Registers

<code>%rdi</code>	<code>0x120</code>
<code>%rsi</code>	<code>0x100</code>
<code>%rax</code>	
<code>%rdx</code>	

Memory

	Address
123	<code>0x120</code>
	<code>0x118</code>
	<code>0x110</code>
	<code>0x108</code>
456	<code>0x100</code>

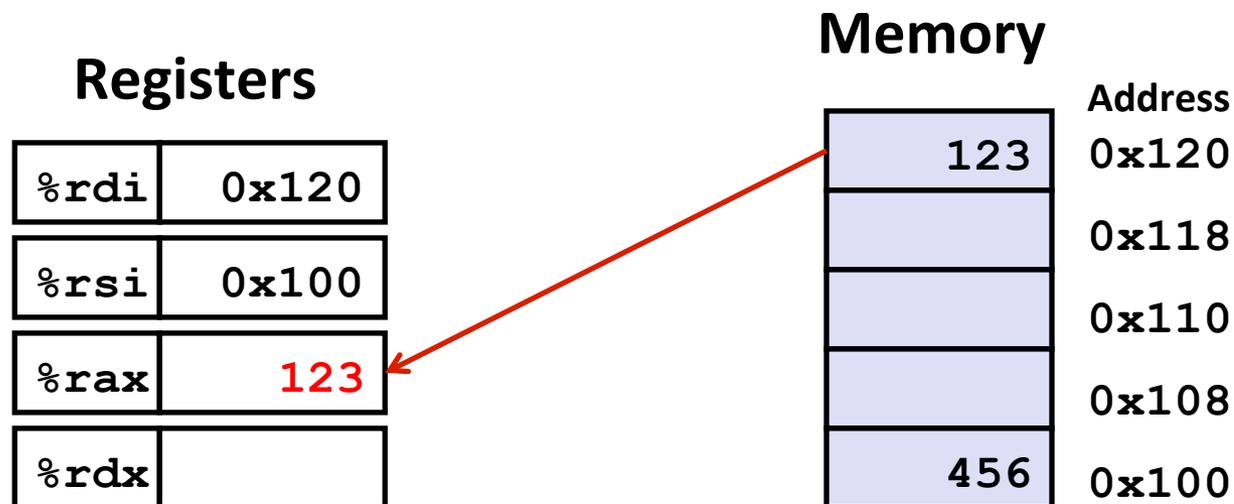
`swap:`

```

movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret

```

Understanding Swap()



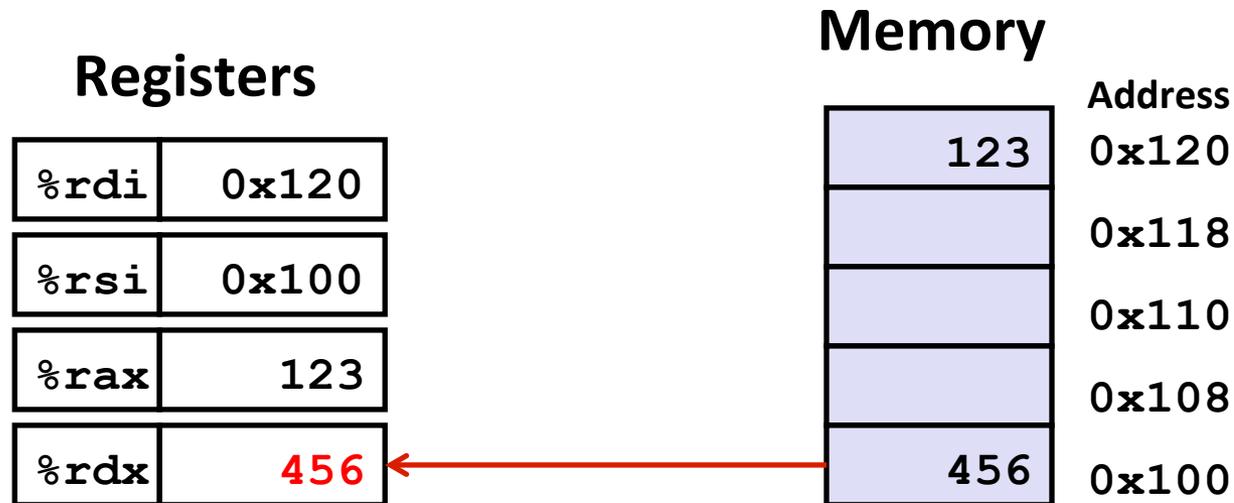
swap:

```

movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret

```

Understanding Swap()



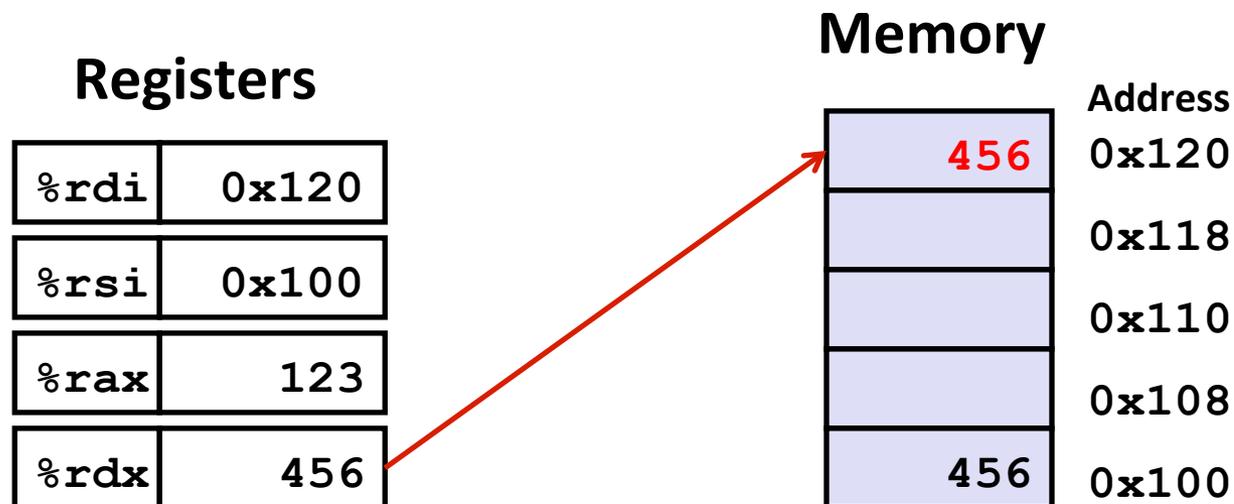
swap:

```

movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret

```

Understanding Swap()



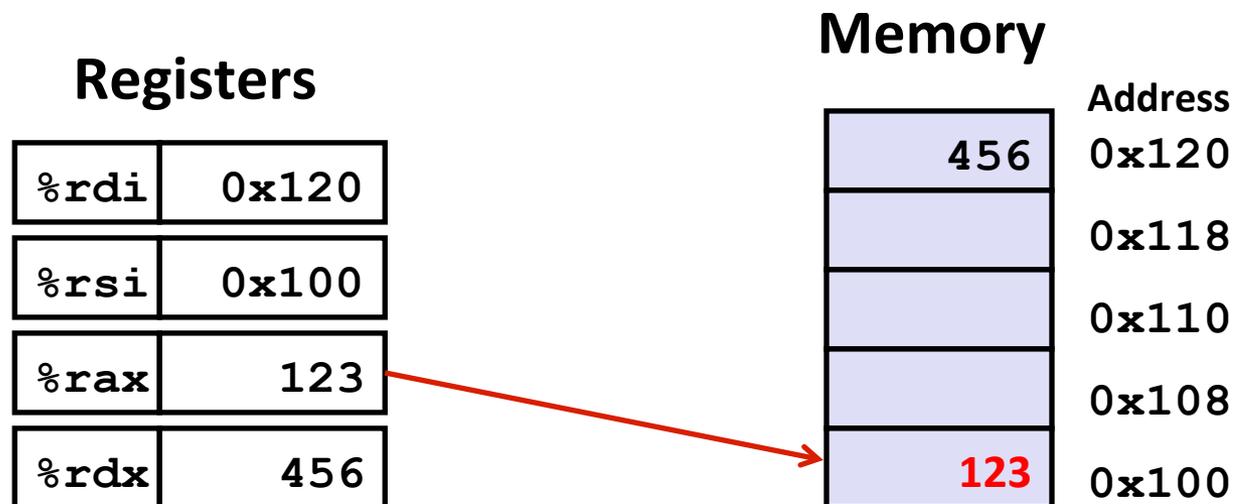
swap:

```

movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret

```

Understanding Swap()



swap:

```

movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret

```

Simple Memory Addressing Modes

■ Normal (R) Mem[Reg[R]]

- Register R specifies memory address
- Aha! Pointer dereferencing in C

```
movq (%rcx), %rax
```

■ Displacement D(R) Mem[Reg[R]+D]

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

```
movq 8(%rbp), %rdx
```

Complete Memory Addressing Modes

■ Most General Form

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

- D: Constant “displacement” 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for `%rsp`
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

■ 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>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x0100</code>

Expression	Address Computation	Address
<code>0x8 (%rdx)</code>	<code>0xf000 + 0x8</code>	<code>0xf008</code>
<code>(%rdx, %rcx)</code>	<code>0xf000 + 0x100</code>	<code>0xf100</code>
<code>(%rdx, %rcx, 4)</code>	<code>0xf000 + 4*0x100</code>	<code>0xf400</code>
<code>0x80(, %rdx, 2)</code>	<code>2*0xf000 + 0x80</code>	<code>0x1e080</code>

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- **Arithmetic & logical operations**

Address Computation Instruction

■ `leaq Src, Dst`

- Src is address mode expression
- Set Dst 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 $x + k*y$
 - $k = 1, 2, 4, \text{ or } 8$

■ Example

```
long m12(long x)
{
    return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax           # return t<<2
```

Some Arithmetic Operations

■ Two Operand Instructions:

Format	Computation		
<code>addq</code>	<code>Src, Dest</code>	<code>Dest = Dest + Src</code>	
<code>subq</code>	<code>Src, Dest</code>	<code>Dest = Dest - Src</code>	
<code>imulq</code>	<code>Src, Dest</code>	<code>Dest = Dest * Src</code>	
<code>salq</code>	<code>Src, Dest</code>	<code>Dest = Dest << Src</code>	Also called <code>shlq</code>
<code>sarq</code>	<code>Src, Dest</code>	<code>Dest = Dest >> Src</code>	Arithmetic
<code>shrq</code>	<code>Src, Dest</code>	<code>Dest = Dest >> Src</code>	Logical
<code>xorq</code>	<code>Src, Dest</code>	<code>Dest = Dest ^ Src</code>	
<code>andq</code>	<code>Src, Dest</code>	<code>Dest = Dest & Src</code>	
<code>orq</code>	<code>Src, Dest</code>	<code>Dest = Dest Src</code>	

■ Watch out for argument order!

■ No distinction between signed and unsigned int (why?)

Some Arithmetic Operations

■ One Operand Instructions

`incq` `Dest` `Dest = Dest + 1`

`decq` `Dest` `Dest = Dest - 1`

`negq` `Dest` `Dest = - Dest`

`notq` `Dest` `Dest = ~Dest`

■ See book for more instructions

Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

```
arith:
    leaq    (%rdi,%rsi), %rax
    addq    %rdx, %rax
    leaq    (%rsi,%rsi,2), %rdx
    salq    $4, %rdx
    leaq    4(%rdi,%rdx), %rcx
    imulq   %rcx, %rax
    ret
```

Interesting Instructions

- **leaq**: address computation
- **salq**: shift
- **imulq**: multiplication
 - But, only used once

Understanding Arithmetic Expression

Example

```

long arith
(long x, long y, long z)
{
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}

```

```

arith:
    leaq    (%rdi,%rsi), %rax    # t1
    addq    %rdx, %rax          # t2
    leaq    (%rsi,%rsi,2), %rdx
    salq    $4, %rdx           # t4
    leaq    4(%rdi,%rdx), %rcx  # t5
    imulq   %rcx, %rax         # rval
    ret

```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	t1, t2, rval
%rdx	t4
%rcx	t5

Machine Programming I: Summary

- **History of Intel processors and architectures**
 - Evolutionary design leads to many quirks and artifacts
- **C, assembly, machine code**
 - New forms of visible state: program counter, registers, ...
 - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- **Assembly Basics: Registers, operands, move**
 - The x86-64 move instructions cover wide range of data movement forms
- **Arithmetic**
 - C compiler will figure out different instruction combinations to carry out computation