

COMP2310/COMP6310

Systems, Networks, & Concurrency

Convener: Prof John Taylor

Machine-Level Programming I: Basics

Acknowledgement of material: With changes suited to ANU needs, the slides are obtained from Carnegie Mellon University: <https://www.cs.cmu.edu/~213/>

Today: Machine Programming I: Basics

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

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
 - Now 3 volumes, about 5,000 pages of documentation
- **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.

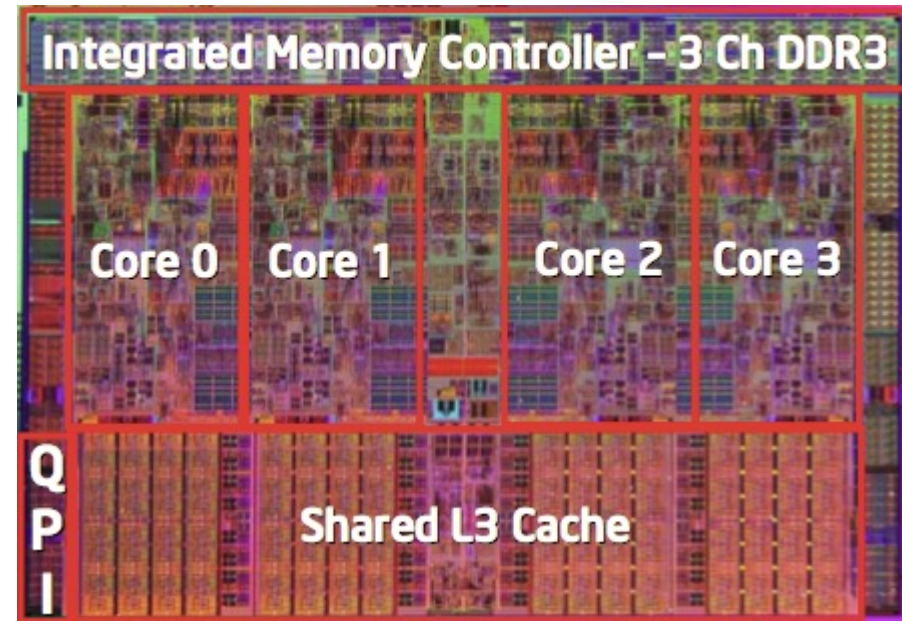
Intel x86 Evolution: Milestones

<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	1060-3333
<ul style="list-style-type: none">▪ First multi-core Intel processor			
■ Core i7	2008	731M	1600-4400
<ul style="list-style-type: none">▪ Four cores			

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	2000	42M
■ Core 2 Duo	2006	291M
■ Core i7	2008	731M
■ Core i7 Skylake	2015	1.9B



■ Added Features

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

Intel x86 Processors, cont.

■ Past Generations

Process technology

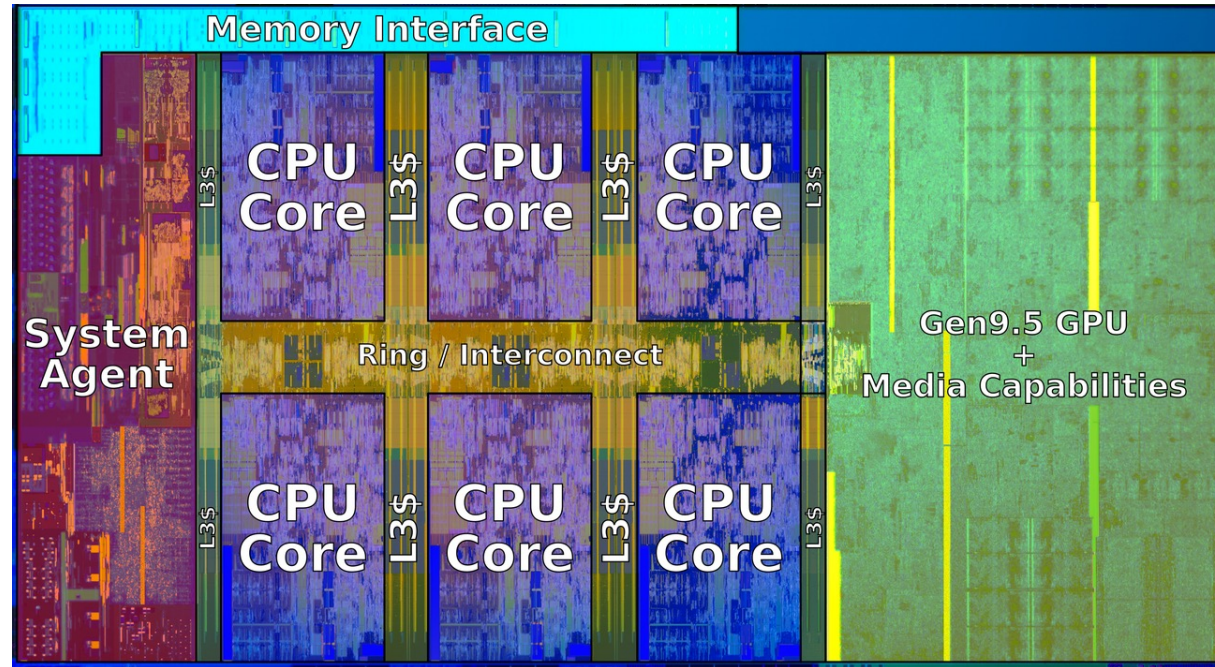
■ 1 st Pentium Pro	1995	600 nm
■ 1 st Pentium III	1999	250 nm
■ 1 st Pentium 4	2000	180 nm
■ 1 st Core 2 Duo	2006	65 nm

■ Recent & Upcoming Generations

1.	Nehalem	2008	45 nm
2.	Sandy Bridge	2011	32 nm
3.	Ivy Bridge	2012	22 nm
4.	Haswell	2013	22 nm
5.	Broadwell	2014	14 nm
6.	Skylake	2015	14 nm
7.	Kaby Lake	2016	14 nm
8.	Coffee Lake	2017	14 nm
9.	Cannon Lake	2018	10 nm
10.	Ice Lake	2019	10 nm
11.	Tiger Lake	2020	10 nm
12.	Alder Lake	2022	“intel 7” (10nm+++)
13.	Raptor Lake	2023	“intel 7” (10nm+++)

**Process technology dimension
= width of narrowest wires
(10 nm ≈ 100 atoms wide)**

2018 State of the Art: Coffee Lake



■ Mobile Model: Core i7

- 2.2-3.2 GHz
- 45 W

■ Desktop Model: Core i7

- Integrated graphics
- 2.4-4.0 GHz
- 35-95 W

■ Server Model: Xeon E

- Integrated graphics
- Multi-socket enabled
- 3.3-3.8 GHz
- 80-95 W

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
 - 1995-2011: Lead semiconductor “fab” in world
 - 2018: #2 largest by \$\$ (#1 is Samsung)
 - 2019: reclaimed #1
- AMD fell behind: Spun off GlobalFoundries
- 2019-20: Pulled ahead! Used TSMC for part of fab
- 2022: Intel re-took the lead

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

■ x86-64

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

■ Presentation

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

Today: Machine Programming I: Basics

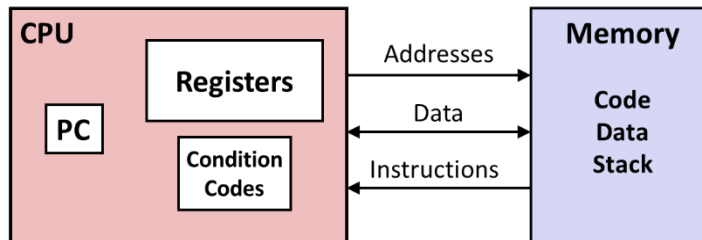
- History of Intel processors and architectures
- **Assembly Basics: Registers, operands, move**
- Arithmetic & logical operations
- C, assembly, machine code

Levels of Abstraction

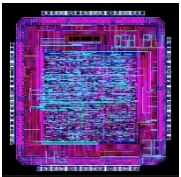
C programmer

```
#include <stdio.h>
int main(){
    int i, n = 10, t1 = 0, t2 = 1, nxt;
    for (i = 1; i <= n; ++i){
        printf("%d, ", t1);
        nxt = t1 + t2;
        t1 = t2;
        t2 = nxt; }
    return 0; }
```

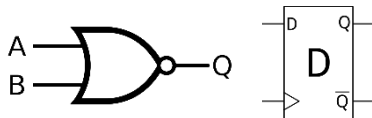
Assembly programmer



Computer Designer



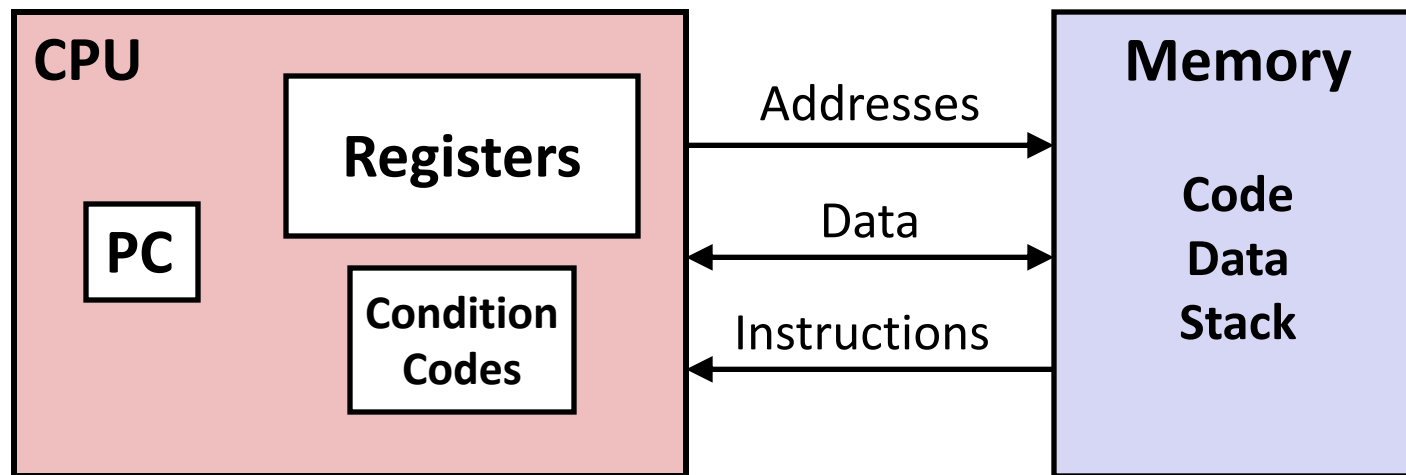
Gates, clocks, circuit layout, ...



Definitions

- **Architecture:** (also ISA: instruction set architecture) The parts of a processor design that one needs to understand for writing 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
 - RISC V: New open-source ISA

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

Assembly: 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**
- **(SIMD vector data types of 8, 16, 32 or 64 bytes)**
- **Code: Byte sequences encoding series of instructions**
- **No aggregate types such as arrays or structures**
 - Just contiguously allocated bytes in memory

Assembly: Data Types

- “Integer” data of 1, 2, 4, or 8 bytes

- Data values
- Addresses (untyped pointers)

Register names



```
addq %rbx, %rax
```

is

rax += rbx

These are 64-bit registers, so we know this is a 64-bit add

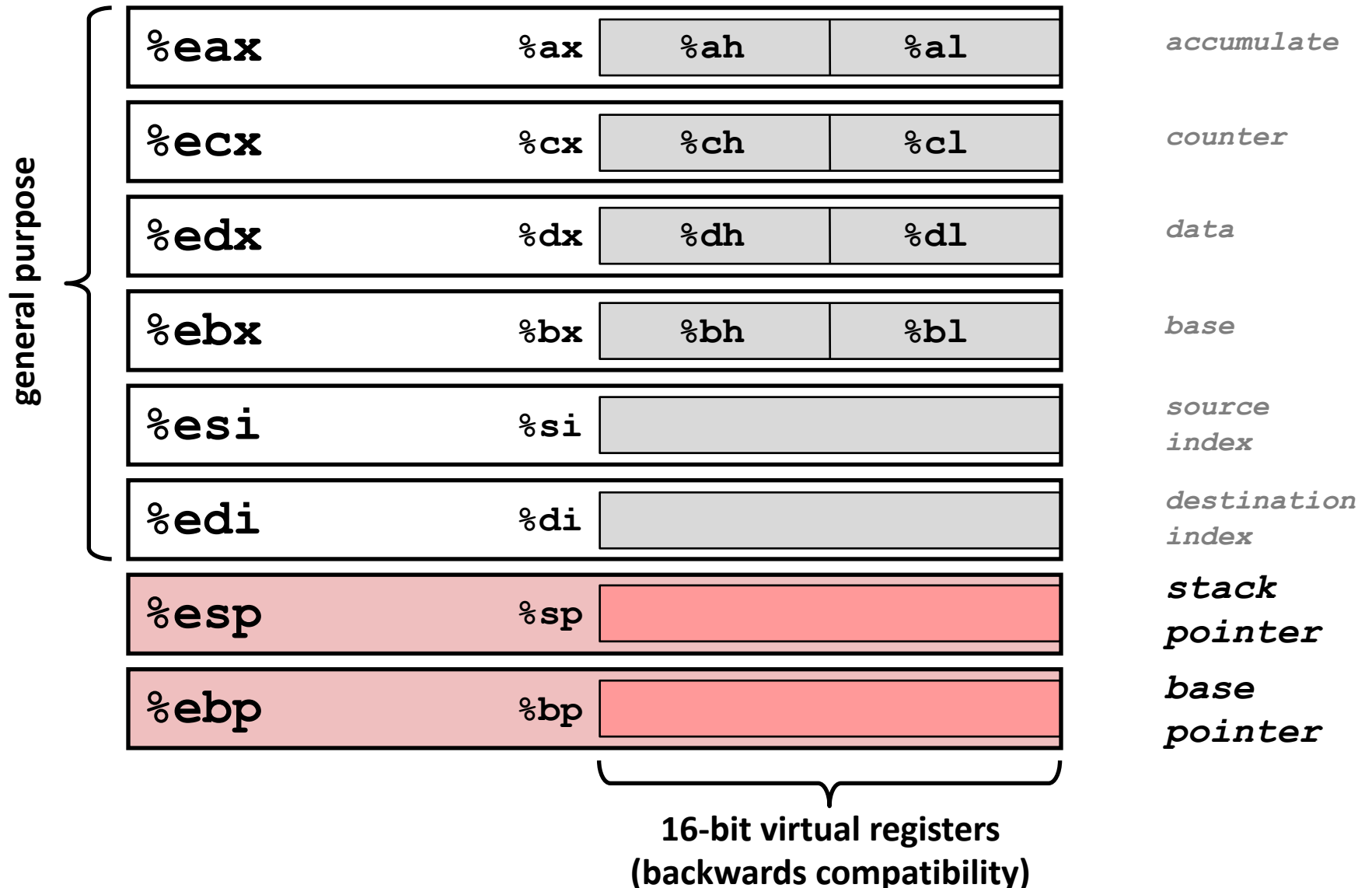
x86-64 Integer Registers

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

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

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- Not part of memory (or cache)

Some History: IA32 Registers



Assembly: Operations

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

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 “addressing modes”

`%rax`

`%rcx`

`%rdx`

`%rbx`

`%rsi`

`%rdi`

`%rsp`

`%rbp`

`%rN`

Warning: Intel docs use
`mov Dest, Source`

movq 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
```

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]]

Example of Simple Addressing Modes

```
void  
whatAmI (<type> a, <type> b)  
{  
    ???  
}
```

`%rdi`

`%rsi`

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

Interlude: Pointers

Pointers: Introduction

- A pointer is a variable that contains the address of another variable

```
int A = 19;  
int B = 10;  
int C = 8;  
int D = 17;  
....  
int *P;  
P = &B;  
//unary operator & gives  
    the address of a  
    variable
```

Address	Data	Variable
•	•	•
•	•	•
•	•	•
00000010	00000004	P
0000000C	17	D
00000008	8	C
00000004	10	B
00000000	19	A

← 4 Bytes →

Pointers: Introduction

- Can use the pointer to access the value stored in a memory location

```
int A = 19;  
int B = 10;  
int C = 8;  
int D = 17;  
....  
int *P;  
P = &B;  
*P = 1;  
//dereferencing or  
// indirection operator  
//that accesses the value  
// stored at address in P
```

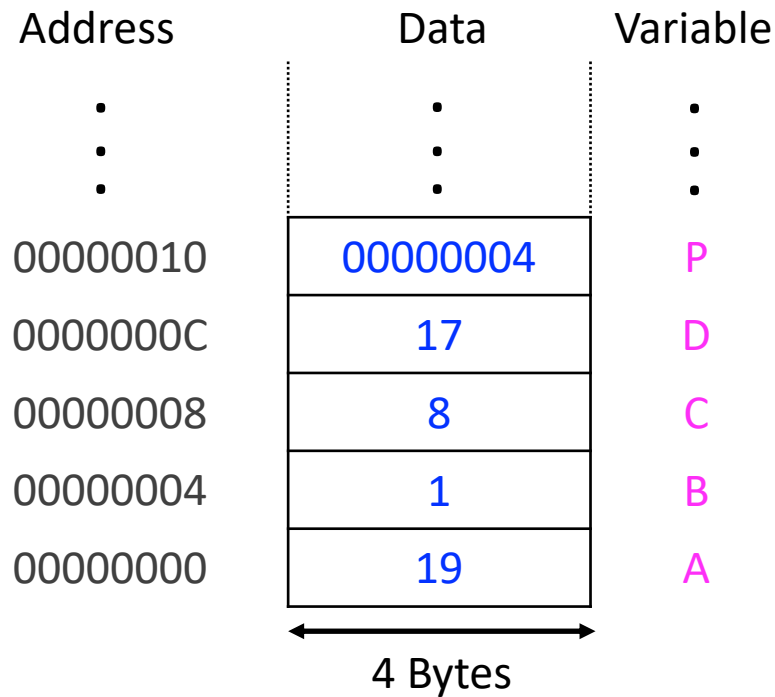
Address	Data	Variable
•	•	•
•	•	•
•	•	•
00000010	00000004	P
0000000C	17	D
00000008	8	C
00000004	1	B
00000000	19	A

← 4 Bytes →

Pointers: Example

- A pointer is 4-bytes on a 32-bit system and 8-bytes on a 64-bits system & it can be stored on the stack or data segment like ordinary variables

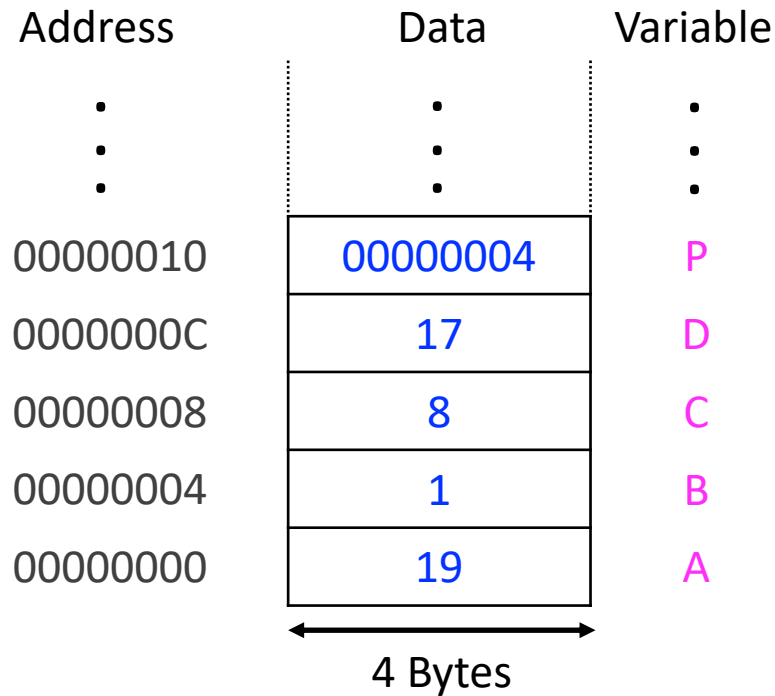
```
int A = 19;
int B = 1;
int C = 8;
int D = 17;
....
int *P = &B;
char *Q = &B;
// Both P and Q contain
// 00000004
printf("%i\n", *P); ??
printf("%i\n", *Q); ??
```



Answer

- `printf("%i\n", *P);` Output is always 1
- `printf("%i\n", *Q);` Big Endian: 0, Little Endian: 1

```
int A = 19;
int B = 1;
int C = 8;
int D = 17;
....
int *P = &B;
char *Q = &B;
// Both P and Q contain
    00000004
printf("%i\n", *P); ??
printf("%i\n", *Q); ??
```



Pointers: Their *Nature*

- A pointer points to a memory location
- Its content is a memory address
- It wears “datatype glasses”
 - Wherever it points, it sees through these glasses
- The variable stored at some memory address can be interpreted (via the dereferencing operator `*`) as character or integer or float, depending on the type of the pointer



Pointers: Their *Nature*

CHAPTER 5: Pointers and Arrays

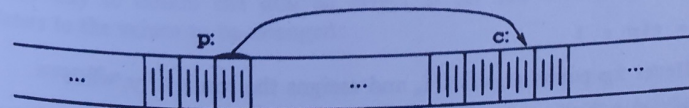
A pointer is a variable that contains the address of a variable. Pointers are much used in C, partly because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways. Pointers and arrays are closely related; this chapter also explores this relationship and shows how to exploit it.

Pointers have been lumped with the `goto` statement as a marvelous way to create impossible-to-understand programs. This is certainly true when they are used carelessly, and it is easy to create pointers that point somewhere unexpected. With discipline, however, pointers can also be used to achieve clarity and simplicity. This is the aspect that we will try to illustrate.

The main change in ANSI C is to make explicit the rules about how pointers can be manipulated, in effect mandating what good programmers already practice and good compilers already enforce. In addition, the type `void *` (pointer to void) replaces `char *` as the proper type for a generic pointer.

5.1 Pointers and Addresses

Let us begin with a simplified picture of how memory is organized. A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups. One common situation is that any byte can be a `char`, a pair of one-byte cells can be treated as a short integer, and four adjacent bytes form a `long`. A pointer is a group of cells (often two or four) that can hold an address. So if `c` is a `char` and `p` is a pointer that points to it, we could represent the situation this way:



The unary operator `&` gives the address of an object, so the statement

Pointer Translation in Action

```
*dest = t;
```

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

0x40059e: 48 89 03

0100 1 0 0 0 10001011 00 000 011
REX W R X B MOV r->x Mod R M

C

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]**

Machine

3 bytes at address **0x40059e**

Compact representation of the assembly instruction

(Relatively) easy for hardware to interpret

Back to Simple Addressing Modes

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
```

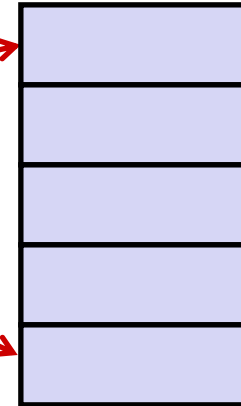
Understanding Swap()

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

Registers

%rdi	
%rsi	
%rax	
%rdx	

Memory



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

%rdi	0x120
%rsi	0x100
%rax	
%rdx	

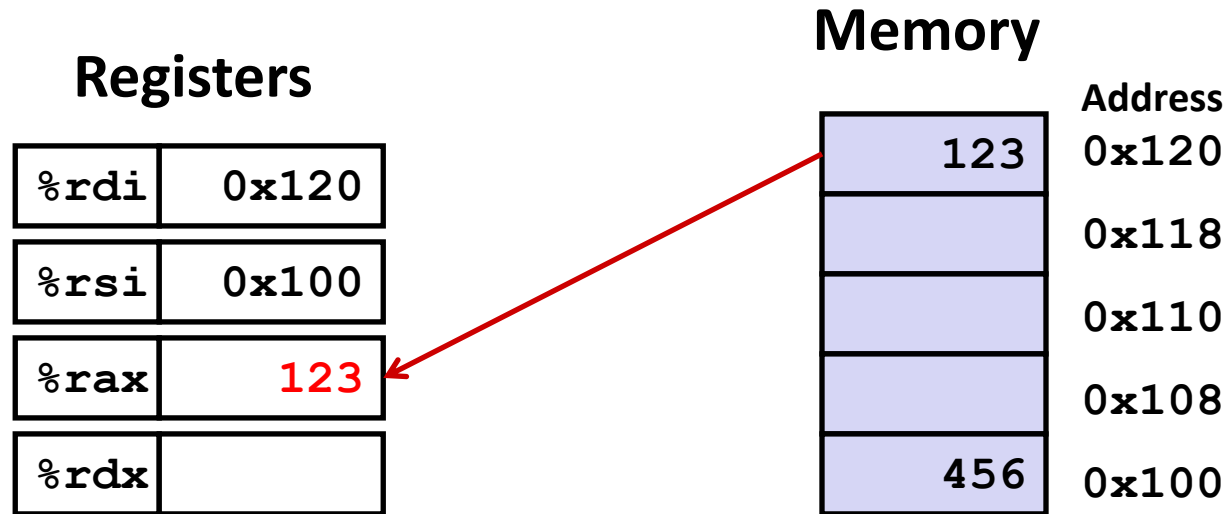
Memory

Address
0x120
123
0x118
0x110
0x108
0x100
456

swap:

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

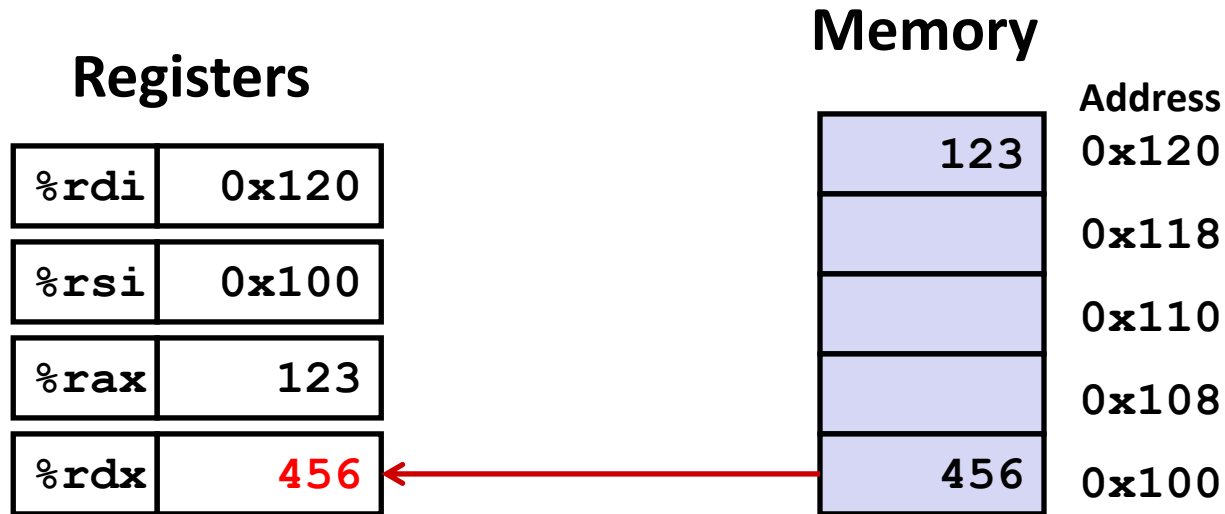
Understanding Swap()



swap:

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

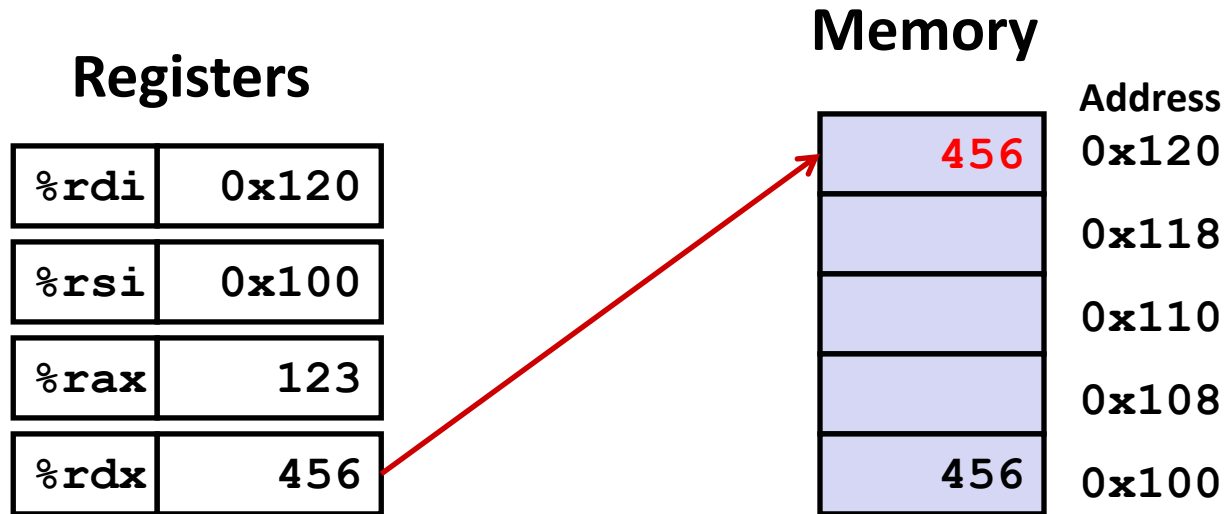
Understanding Swap()



swap:

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

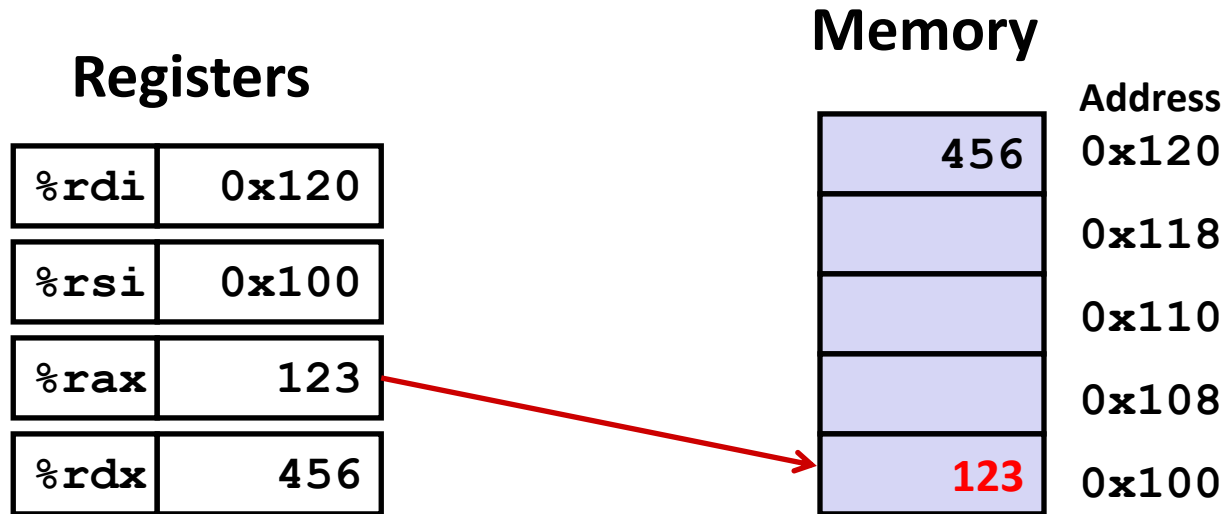
Understanding Swap()



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
```

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

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?*)

Expression	Address Computation	Address
0x8 (%rdx)		
(%rdx, %rcx)		
(%rdx, %rcx, 4)		
0x80 (, %rdx, 2)		

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
- Assembly Basics: Registers, operands, move
- **Arithmetic & logical operations**
- C, assembly, machine code

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+2*x
salq $2, %rax               # return t<<2
```

Some Arithmetic Operations

■ Two Operand Instructions:

Format	Computation		
<code>addq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} + \text{Src}$	
<code>subq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} - \text{Src}$	
<code>imulq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} * \text{Src}$	
<code>salq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} \ll \text{Src}$	Also called <code>shlq</code>
<code>sarq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} \gg \text{Src}$	Arithmetic
<code>shrq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} \gg \text{Src}$	Logical
<code>xorq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} \wedge \text{Src}$	
<code>andq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} \& \text{Src}$	
<code>orq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} \text{Src}$	

- Watch out for argument order! *Src, Dest*
(Warning: Intel docs use “op *Dest, Src*”)
- No distinction between signed and unsigned int (why?)

Some Arithmetic Operations

■ One Operand Instructions

<code>incq</code>	<code>Dest</code>	$\text{Dest} = \text{Dest} + 1$
<code>decq</code>	<code>Dest</code>	$\text{Dest} = \text{Dest} - 1$
<code>negq</code>	<code>Dest</code>	$\text{Dest} = -\text{Dest}$
<code>notq</code>	<code>Dest</code>	$\text{Dest} = \sim\text{Dest}$

■ 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
```

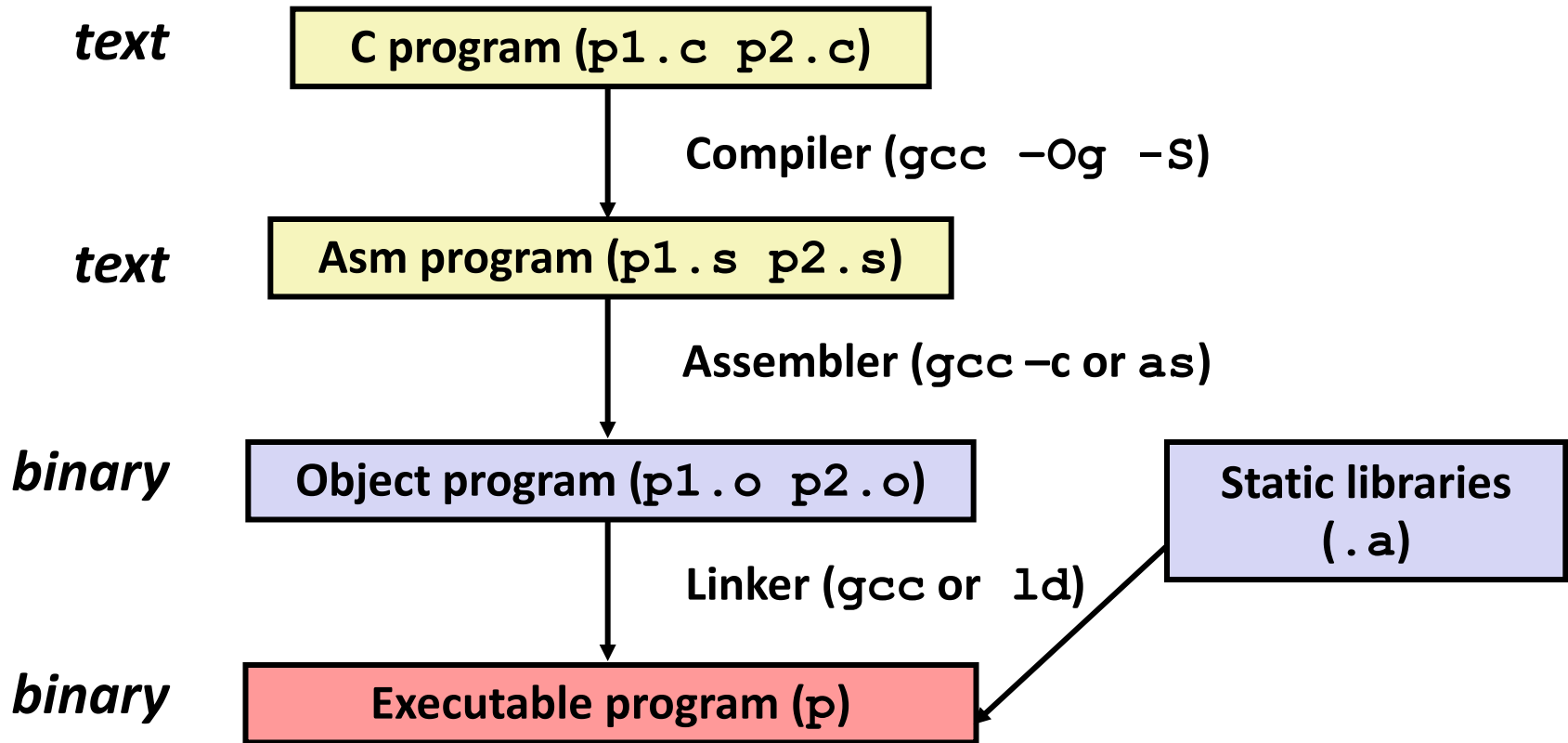
Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z , t4
%rax	t1 , t2 , rval
%rcx	t5

Today: Machine Programming I: Basics

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

Turning C into Object Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
 - Use debugging-friendly optimizations (`-Og`)
 - 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 with command

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

Produces file `sum.s`

What it really looks like

```
.globl  sumstore
.type   sumstore, @function
sumstore:
.LFB35:
    .cfi_startproc
pushq   %rbx
.cfi_def_cfa_offset 16
.cfi_offset 3, -16
movq    %rdx, %rbx
call    plus
movq    %rax, (%rbx)
popq    %rbx
.cfi_def_cfa_offset 8
ret
.cfi_endproc
.LFE35:
.size   sumstore, .-sumstore
```

What it really looks like

Things that look weird
and are preceded by a ‘
are generally directives.

```
.globl  sumstore
.type   sumstore, @function
sumstore:
.LFB35:
    .cfi_startproc
    pushq   %rbx
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    movq    %rdx, %rbx
    call    plus
    movq    %rax, (%rbx)
    popq    %rbx
    .cfi_def_cfa_offset 8
    ret
    .cfi_endproc
.LFE35:
.size     sumstore, .-sumstore
```

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

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

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

- Disassemble procedure

```
gdb sum
```

```
disassemble sumstore
```

Alternate Disassembly

Object
Code

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

- Disassemble procedure

`gdb sum`

`disassemble sumstore`

- Examine the 14 bytes starting at `sumstore`

`x/14xb 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

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