

# **Introduction to Computer Systems**

Cheng Jin

# Course Information

- Email
  - jc@fudan.edu.cn
- Office
  - 江湾校区交叉二号楼D5013
- TA

# Rules

- Attendance
- Quite vs Talktive
- Portrait Photo
  - 240 (width) x 320 (height)
  - 学号\_姓名.jpg

# Why this course?

- You are going to learn:
- How to avoid strange numerical errors
- How to optimize your C code
- How the compiler implements procedure calls
- How to recognize and avoid nasty errors
- How to write your own dynamic storage allocation package
- How to...

# Why this course?

From: torvalds@klaava.Helsinki.FI (**Linus Benedict Torvalds**)

Newsgroups: comp.os.minix

Subject: What would you like to see most in minix?

Summary: small poll for my new operating system

Date: **25 Aug 91 20:57:08 GMT**



Hello everybody out there using minix –

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since April, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

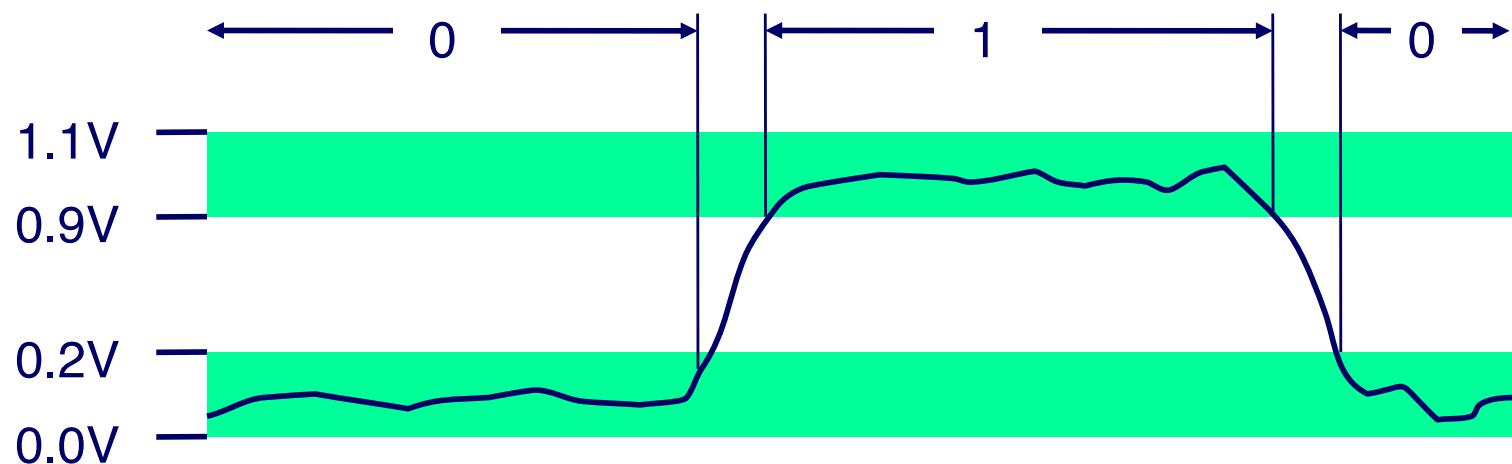
# **Bits, Bytes, and Integers**

# Bits, Bytes, and Integers

- **Representing information as bits**
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- **Floating Point**
- **Representations in memory, pointers, strings**

# Everything is bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
  - Computers determine what to do (instructions)
  - ... and represent and manipulate numbers, sets, strings, etc...
- Why bits? Electronic Implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



# For example, can count in binary

## ■ Base 2 Number Representation

- Represent  $15213_{10}$  as  $11101101101101_2$
- Represent  $1.20_{10}$  as  $1.0011001100110011[0011]\dots_2$
- Represent  $1.5213 \times 10^4$  as  $1.1101101101101_2 \times 2^{13}$

# Encoding Byte Values

## ■ Byte = 8 bits

- Binary  $00000000_2$  to  $11111111_2$
- Decimal:  $0_{10}$  to  $255_{10}$
- Hexadecimal  $00_{16}$  to  $FF_{16}$ 
  - Base 16 number representation
  - Use characters '0' to '9' and 'A' to 'F'
  - Write  $FA1D37B_{16}$  in C as
    - `0xFA1D37B`
    - `0xfa1d37b`

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

# Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
<b>char</b>	1	1	1
<b>short</b>	2	2	2
<b>int</b>	4	4	4
<b>long</b>	4	8	8
<b>float</b>	4	4	4
<b>double</b>	8	8	8
<b>long double</b>	-	-	10/16
pointer	4	8	8

# Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Floating Point
- Representations in memory, pointers, strings

# Boolean Algebra

- Developed by George Boole in 19th Century

- Algebraic representation of logic
  - Encode “True” as 1 and “False” as 0

And

- $A \& B = 1$  when both  $A=1$  and  $B=1$

&	0	1
0	0	0
1	0	1

Or

- $A | B = 1$  when either  $A=1$  or  $B=1$

	0	1
0	0	1
1	1	1

Not

- $\sim A = 1$  when  $A=0$

$\sim$	
0	1
1	0

Exclusive-Or (Xor)

- $A ^ B = 1$  when either  $A=1$  or  $B=1$ , but not both

$^$	0	1
0	0	1
1	1	0

# General Boolean Algebras

- Operate on Bit Vectors
  - Operations applied bitwise

$$\begin{array}{rcl} \begin{array}{c} 01101001 \\ \& 01010101 \end{array} & \begin{array}{c} 01101001 \\ | \quad 01010101 \end{array} & \begin{array}{c} 01101001 \\ ^ \quad 01010101 \end{array} \\ \hline \begin{array}{c} 01000001 \\ 0111101 \end{array} & \begin{array}{c} 0111101 \\ 00111100 \end{array} & \begin{array}{c} 10101010 \\ 00111100 \end{array} \end{array}$$

- All of the Properties of Boolean Algebra Apply

# Example: Representing & Manipulating Sets

## ■ Representation

- Width w bit vector represents subsets of  $\{0, \dots, w-1\}$
- $a_j = 1$  if  $j \in A$

- 01101001       $\{0, 3, 5, 6\}$

- ~~76543210~~

- 01010101       $\{0, 2, 4, 6\}$

- ~~76543210~~

## ■ Operations

■ & Intersection	01000001	$\{0, 6\}$
■   Union	01111101	$\{0, 2, 3, 4, 5, 6\}$
■ ^ Symmetric difference	00111100	$\{2, 3, 4, 5\}$
■ ~ Complement	10101010	$\{1, 3, 5, 7\}$

# Bit-Level Operations in C

## ■ Operations &, |, ~, ^ Available in C

- Apply to any “integral” data type
  - long, int, short, char, unsigned
- View arguments as bit vectors
- Arguments applied bit-wise

## ■ Examples (Char data type)

- $\sim 0x41 \rightarrow 0xBE$ 
  - $\sim 01000001_2 \rightarrow 10111110_2$
- $\sim 0x00 \rightarrow 0xFF$ 
  - $\sim 00000000_2 \rightarrow 11111111_2$
- $0x69 \& 0x55 \rightarrow 0x41$ 
  - $01101001_2 \& 01010101_2 \rightarrow 01000001_2$
- $0x69 | 0x55 \rightarrow 0x7D$ 
  - $01101001_2 | 01010101_2 \rightarrow 01111101_2$

# Contrast: Logic Operations in C

## ■ Contrast to Logical Operators

- `&&`, `||`, `!`
  - View 0 as “False”
  - Anything nonzero as “True”
  - Always return 0 or 1
  - Early termination

## ■ Examples (char data type)

- `!0x41` → `0x00`
- `!0x00` → `0x01`
- `!!0x41` → `0x01`
- `0x69 && 0x55` → `0x01`
- `0x69 || 0x55` → `0x01`
- `p && *p` (avoids null pointer access)

# Contrast: Logic Operations in C

## ■ Contrast to Logical Operators

- `&&`, `||`, `!`
  - View 0 as “False”
  - Anything nonzero is “True”
  - Always short-circuits
  - Early return

## ■ Examples

- `!0x41`
- `!0x00`
- `!!0x41`
- `0x69 && 0x55` → `0x01`
- `0x69 || 0x55` → `0x01`
- `p && *p` (avoids null pointer access)

**Watch out for `&&` vs. `&` (and `||` vs. `|`)...  
one of the more common oopsies in  
C programming**

# Shift Operations

- **Left Shift:**  $x \ll y$ 
  - Shift bit-vector  $x$  left  $y$  positions
    - Throw away extra bits on left
    - Fill with 0's on right
- **Right Shift:**  $x \gg y$ 
  - Shift bit-vector  $x$  right  $y$  positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- **Undefined Behavior**
  - Shift amount  $< 0$  or  $\geq$  word size

<b>Argument x</b>	01100010
$\ll 3$	00010000
<b>Log. <math>\gg 2</math></b>	00011000
<b>Arith. <math>\gg 2</math></b>	00011000

<b>Argument x</b>	10100010
$\ll 3$	00010000
<b>Log. <math>\gg 2</math></b>	00101000
<b>Arith. <math>\gg 2</math></b>	11101000

# Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - **Representation: unsigned and signed**
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Floating Point
- Representations in memory, pointers, strings

# Encoding Integers

## Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

## Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Sign  
Bit

### ■ C short 2 bytes long

```
short int x = 15213;
short int y = -15213;
```

### ■ Sign Bit

- For 2's complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

# Two-complement Encoding Example (Cont.)

x =	15213:	00111011	01101101
y =	-15213:	11000100	10010011

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum	15213		-15213	

# Numeric Ranges

## ■ Unsigned Values

- $UMin = 0$   
000...0
- $UMax = 2^w - 1$   
111...1

## ■ Two's Complement Values

- $TMin = -2^{w-1}$   
100...0
- $TMax = 2^{w-1} - 1$   
011...1

## ■ Other Values

- Minus 1  
111...1

## Values for $W = 16$

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

# Values for Different Word Sizes

	W			
	8	16	32	64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

## ■ Observations

- $|TMin| = TMax + 1$ 
  - Asymmetric range
- $UMax = 2 * TMax + 1$

## ■ C Programming

- `#include <limits.h>`
- Declares constants, e.g.,
  - `ULONG_MAX`
  - `LONG_MAX`
  - `LONG_MIN`
- Values platform specific

# Unsigned & Signed Numeric Values

## ■ Equivalence

- Same encodings for nonnegative values

## ■ Uniqueness

- Every bit pattern represents unique integer value
- Each representable integer has unique bit encoding

## ■ $\Rightarrow$ Can Invert Mappings

- $U2B(x) = B2U^{-1}(x)$ 
  - Bit pattern for unsigned integer
- $T2B(x) = B2T^{-1}(x)$ 
  - Bit pattern for two's comp integer

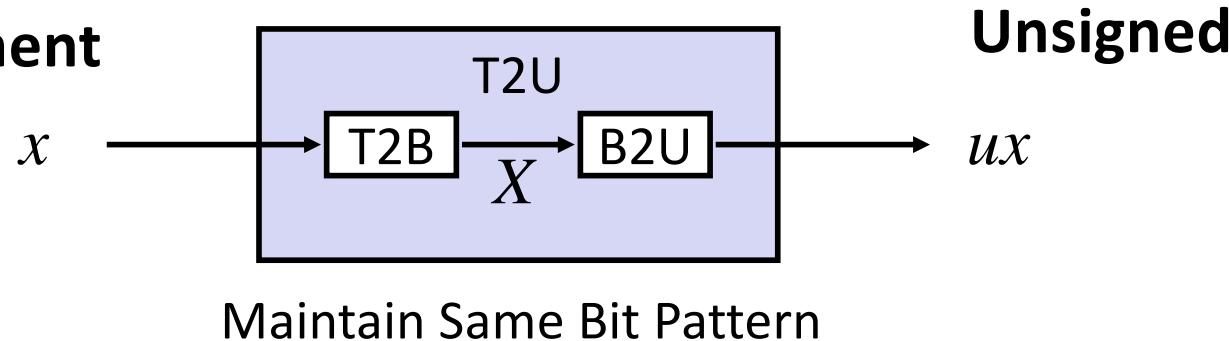
$X$	$B2U(X)$	$B2T(X)$
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

# Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - **Conversion, casting**
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Floating Point
- Representations in memory, pointers, strings

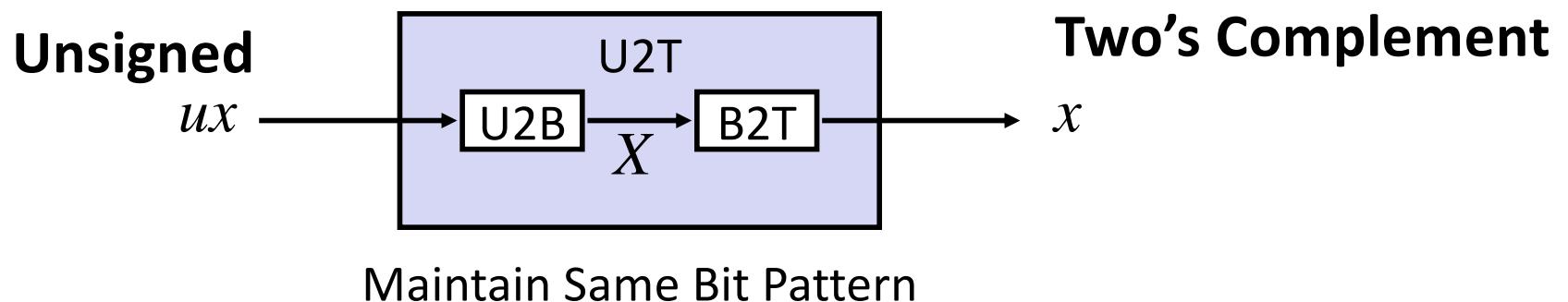
# Mapping Between Signed & Unsigned

Two's Complement



Unsigned

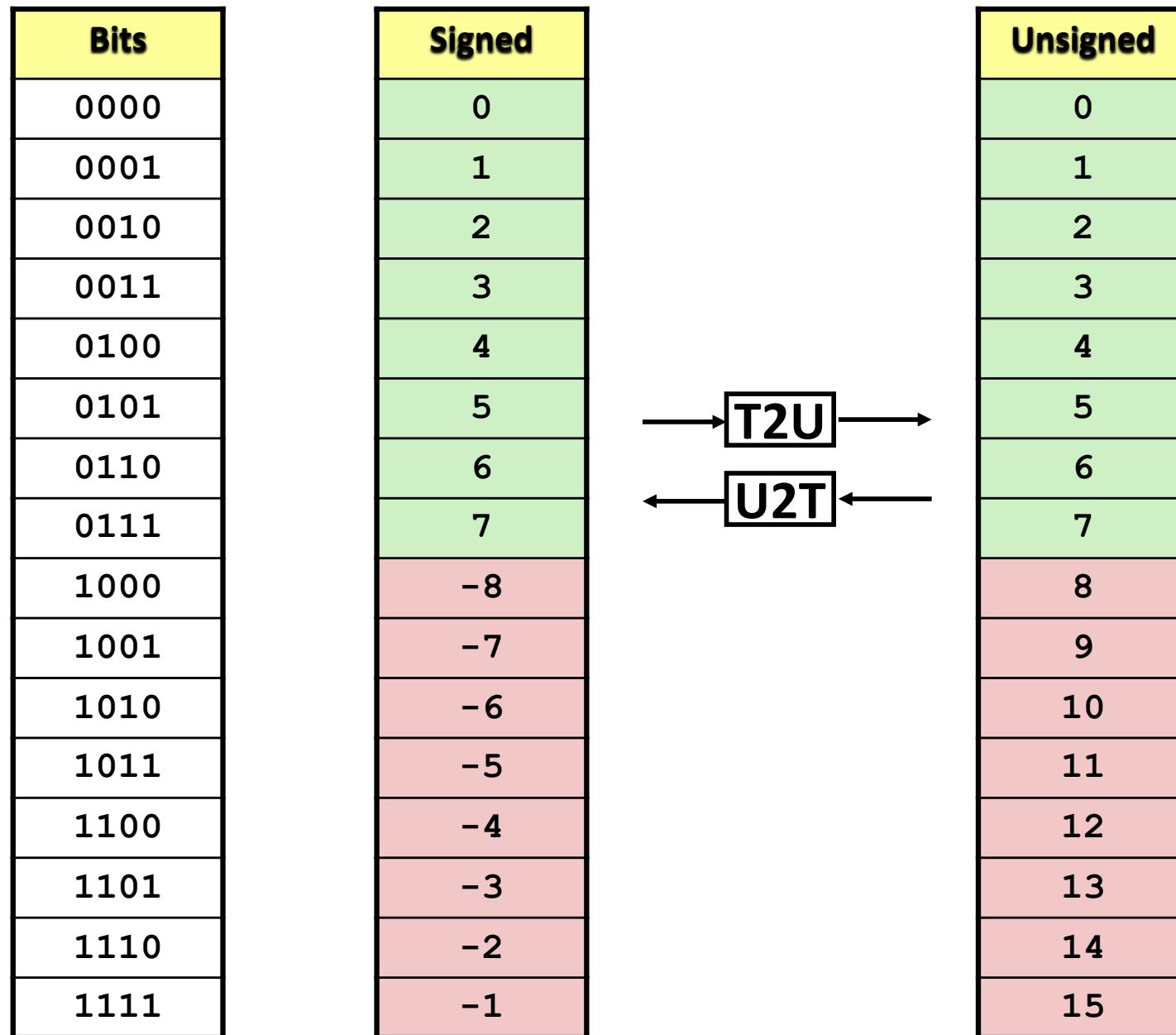
Unsigned



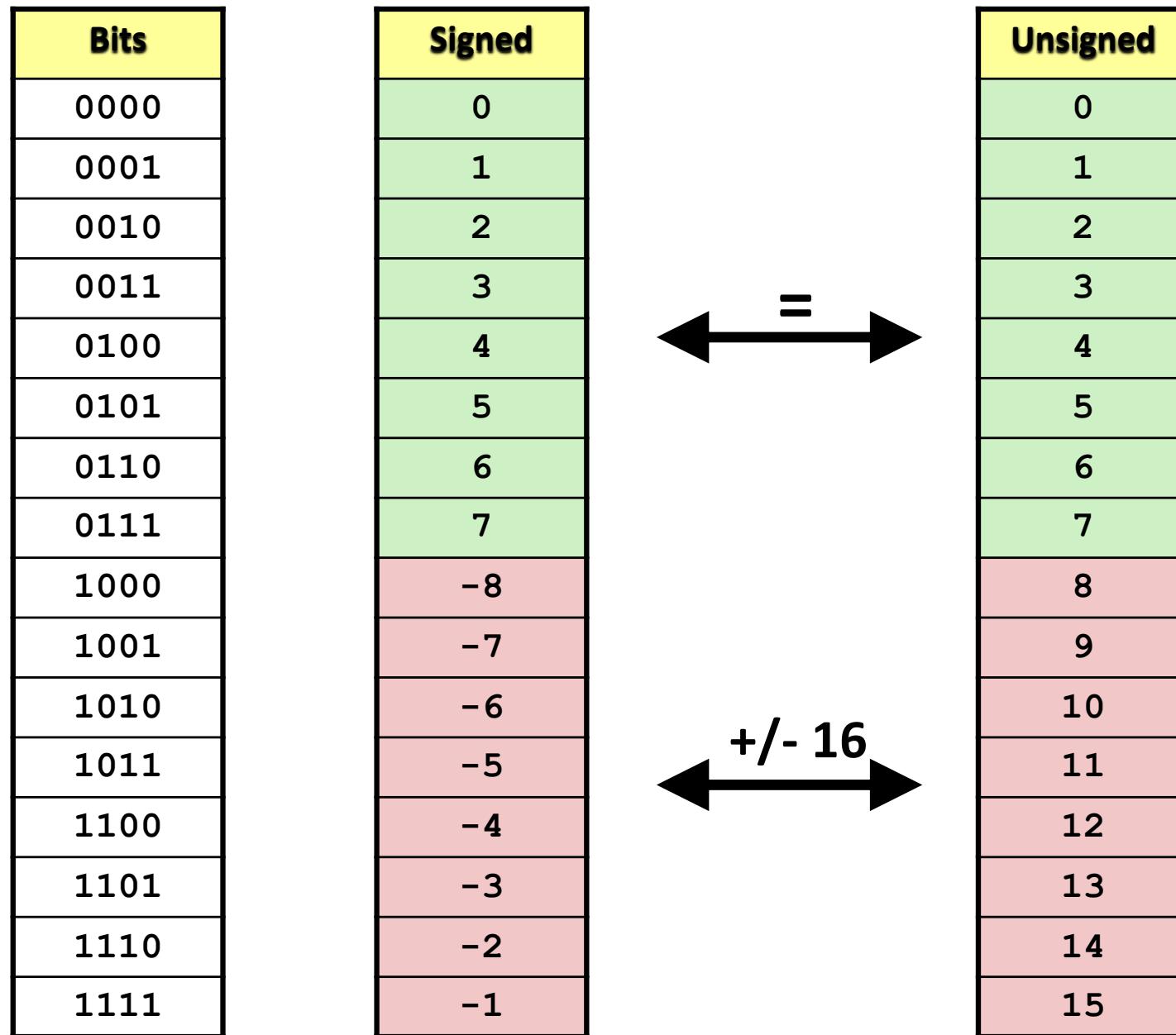
Two's Complement

- Mappings between unsigned and two's complement numbers:  
**Keep bit representations and reinterpret**

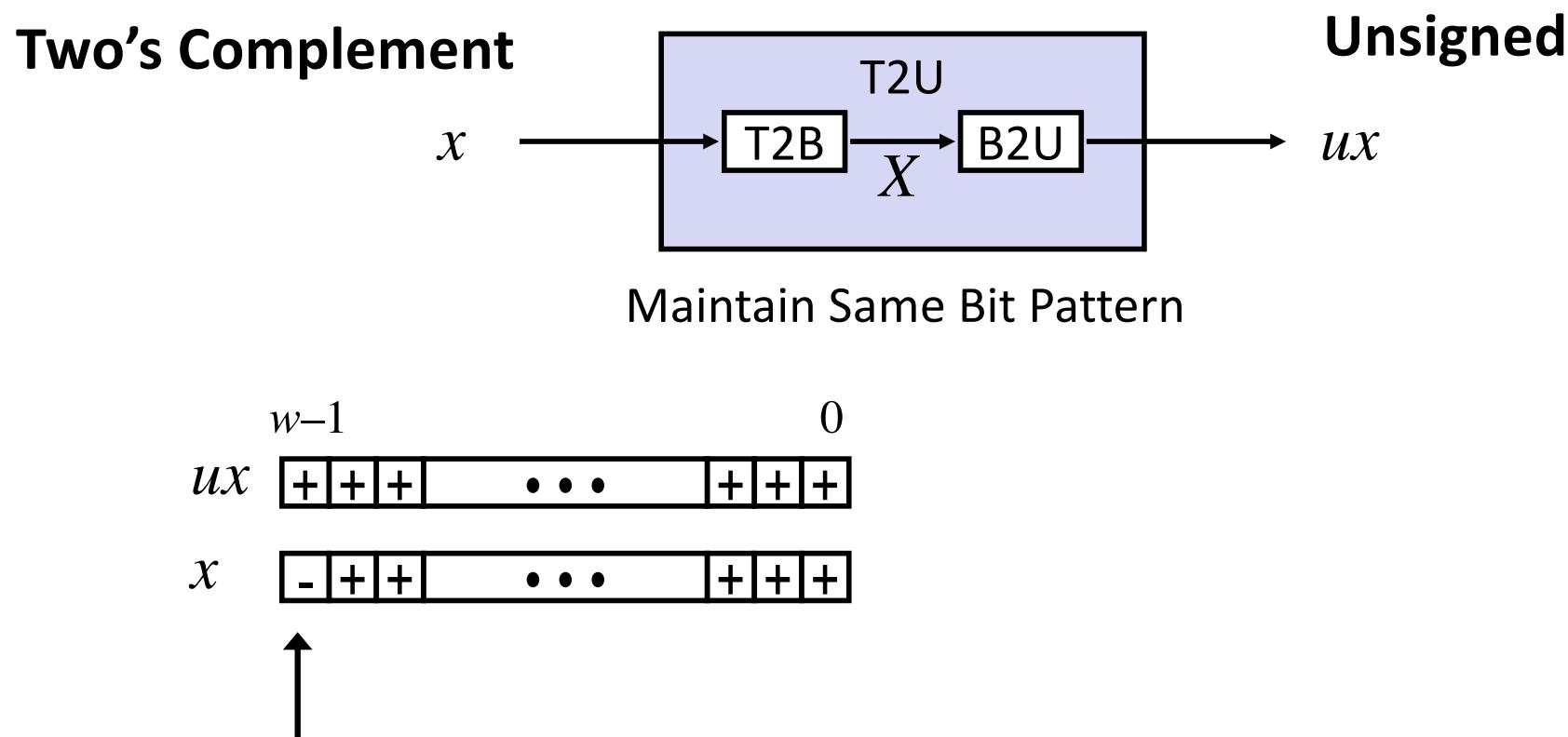
# Mapping Signed ↔ Unsigned



# Mapping Signed ↔ Unsigned



# Relation between Signed & Unsigned



## Large negative weight

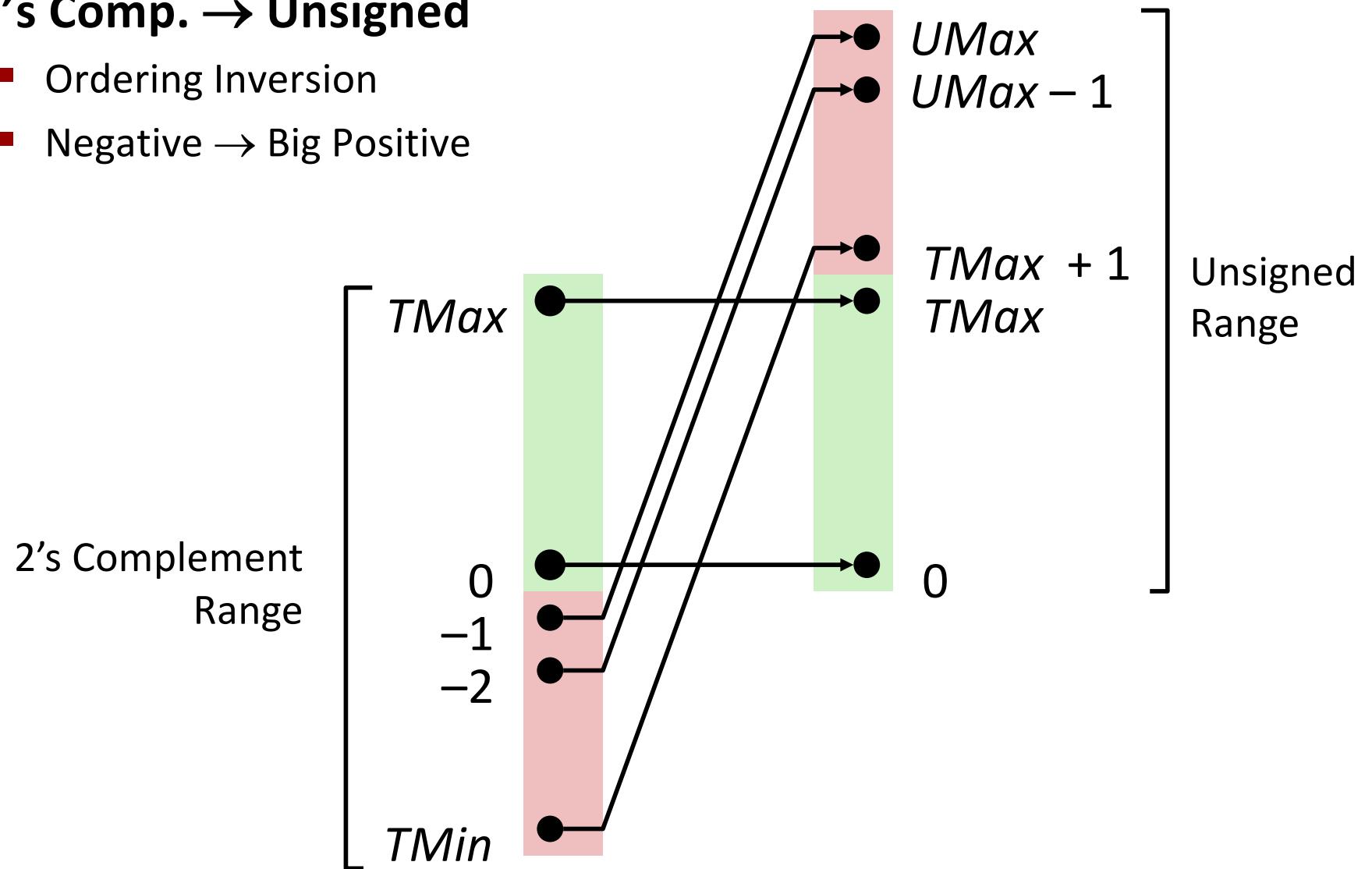
*becomes*

# Large positive weight

# Conversion Visualized

## ■ 2's Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive



# Signed vs. Unsigned in C

## ■ Constants

- By default are considered to be signed integers
- Unsigned if have “U” as suffix

**0U, 4294967259U**

## ■ Casting

- Explicit casting between signed & unsigned same as U2T and T2U

```
int tx, ty;  
unsigned ux, uy;  
tx = (int) ux;  
uy = (unsigned) ty;
```

- Implicit casting also occurs via assignments and procedure calls

```
tx = ux;  
uy = ty;
```

# Casting Surprises

## ■ Expression Evaluation

- If there is a mix of unsigned and signed in single expression,  
*signed values implicitly cast to unsigned*
- Including comparison operations `<`, `>`, `==`, `<=`, `>=`
- Examples for  $W = 32$ : **TMIN = -2,147,483,648** , **TMAX = 2,147,483,647**

■ Constant <sub>1</sub>	Constant <sub>2</sub>	Relation	Evaluation
0	0U	<code>==</code>	unsigned
-1	0	<code>&lt;</code>	signed
-1	0U	<code>&gt;</code>	unsigned
2147483647	-2147483647-1	<code>&gt;</code>	signed
2147483647U	-2147483647-1	<code>&lt;</code>	unsigned
-1	-2	<code>&gt;</code>	signed
(unsigned)-1	-2	<code>&gt;</code>	unsigned
2147483647	2147483648U	<code>&lt;</code>	unsigned
2147483647	(int) 2147483648U	<code>&gt;</code>	signed

# Summary

## Casting Signed $\leftrightarrow$ Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting  $2^w$
  
- Expression containing signed and unsigned int
  - int is cast to unsigned!!

# Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - **Expanding, truncating**
  - Addition, negation, multiplication, shifting
  - Summary
- Floating Point
- Representations in memory, pointers, strings

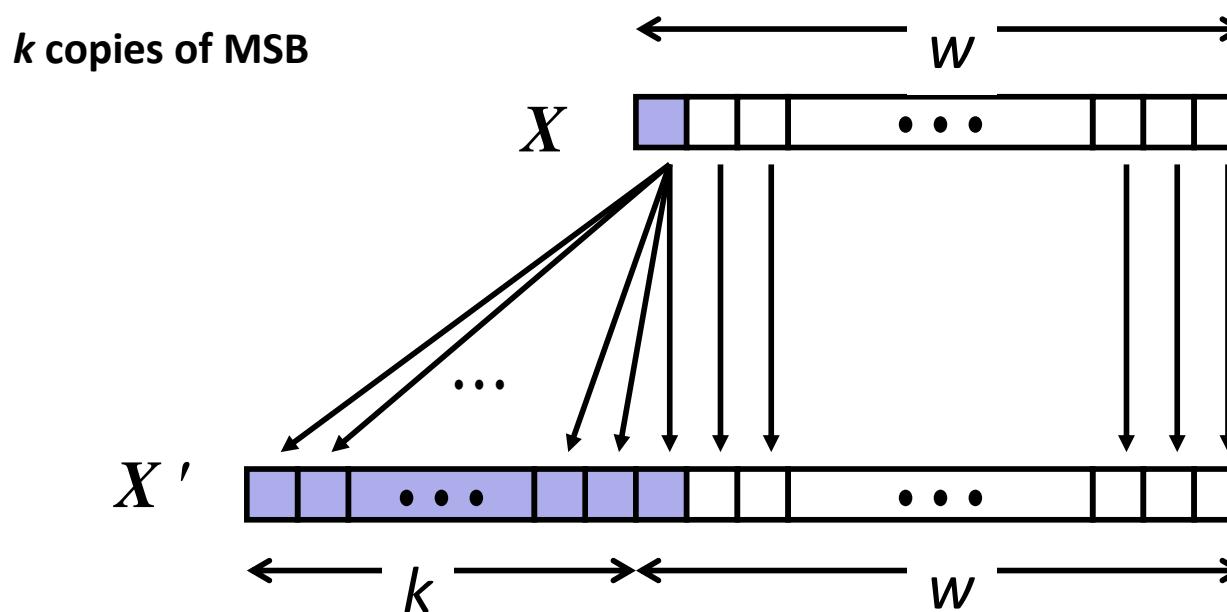
# Sign Extension

## ■ Task:

- Given  $w$ -bit signed integer  $x$
- Convert it to  $w+k$ -bit integer with same value

## ■ Rule:

- Make  $k$  copies of sign bit:
- $X' = \underbrace{x_{w-1}, \dots, x_{w-1}}_k, x_{w-1}, x_{w-2}, \dots, x_0$



# Sign Extension Example

```
short int x = 15213;
int     ix = (int) x;
short int y = -15213;
int     iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- Converting from smaller to larger integer data type
- C automatically performs sign extension

# Summary: Expanding, Truncating: Basic Rules

- **Expanding (e.g., short int to int)**
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result
  
- **Truncating (e.g., unsigned to unsigned short)**
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small numbers yields expected behavior

# Bits, Bytes, and Integers

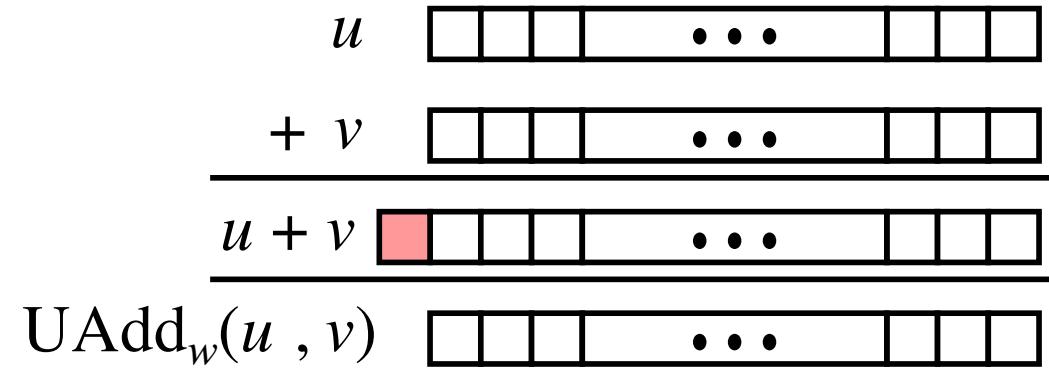
- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - **Addition, negation, multiplication, shifting**
  - Summary
- Floating Point
- Representations in memory, pointers, strings
- Summary

# Unsigned Addition

Operands:  $w$  bits

True Sum:  $w+1$  bits

Discard Carry:  $w$  bits



## ■ Standard Addition Function

- Ignores carry output

## ■ Implements Modular Arithmetic

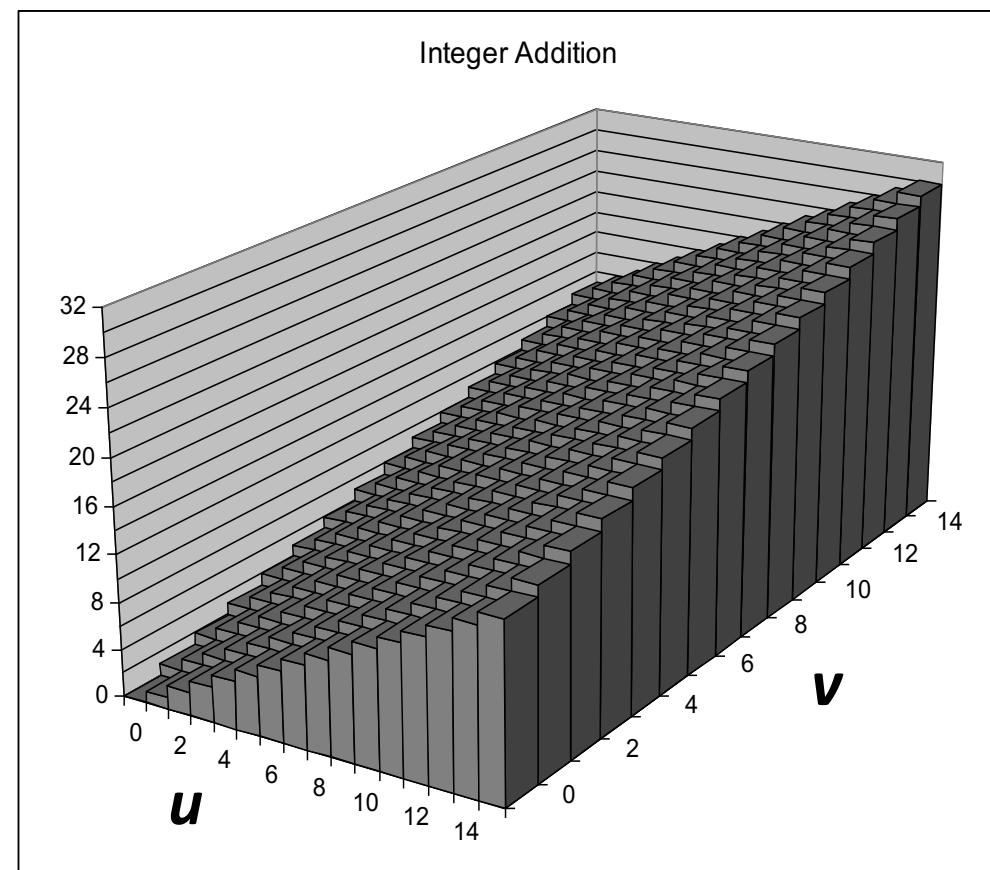
$$s = \text{UAdd}_w(u, v) = u + v \bmod 2^w$$

# Visualizing (Mathematical) Integer Addition

## ■ Integer Addition

- 4-bit integers  $u, v$
- Compute true sum  
 $\text{Add}_4(u, v)$
- Values increase linearly  
with  $u$  and  $v$
- Forms planar surface

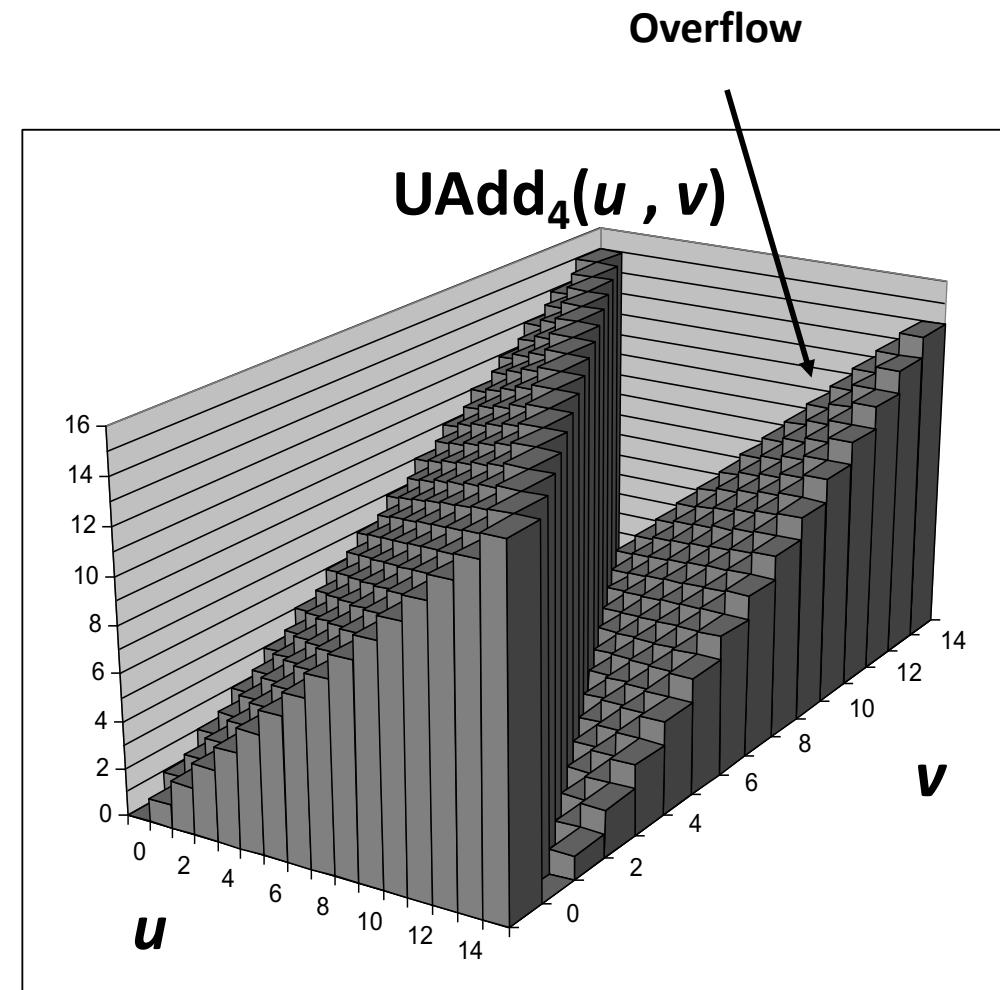
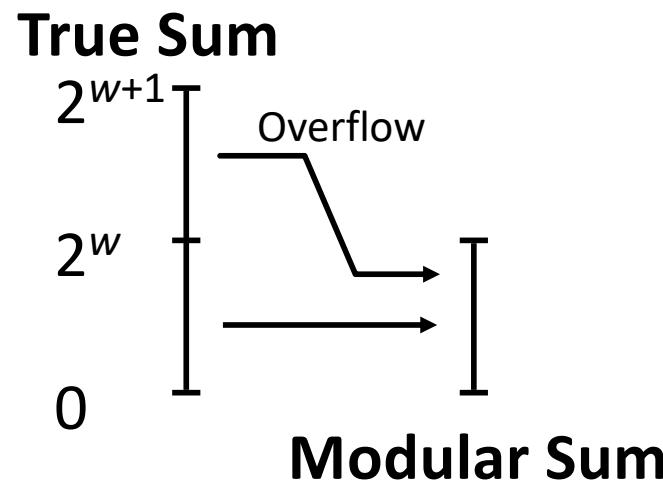
$\text{Add}_4(u, v)$



# Visualizing Unsigned Addition

## Wraps Around

- If true sum  $\geq 2^w$
- At most once



# Two's Complement Addition

## Operands: $w$ bits

## True Sum: $w+1$ bits

## Discard Carry: $w$ bits

$$\begin{array}{r}
 u \quad \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} \dots \boxed{\phantom{0}} \boxed{\phantom{0}} \\
 + \quad v \quad \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} \dots \boxed{\phantom{0}} \boxed{\phantom{0}} \\
 \hline
 u + v \quad \boxed{\textcolor{red}{1}} \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} \dots \boxed{\phantom{0}} \boxed{\phantom{0}} \\
 \hline
 \text{dd}_w(u, v) \quad \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} \dots \boxed{\phantom{0}} \boxed{\phantom{0}}
 \end{array}$$

## ■ TAdd and UAdd have Identical Bit-Level Behavior

- #### ▪ Signed vs. unsigned addition in C:

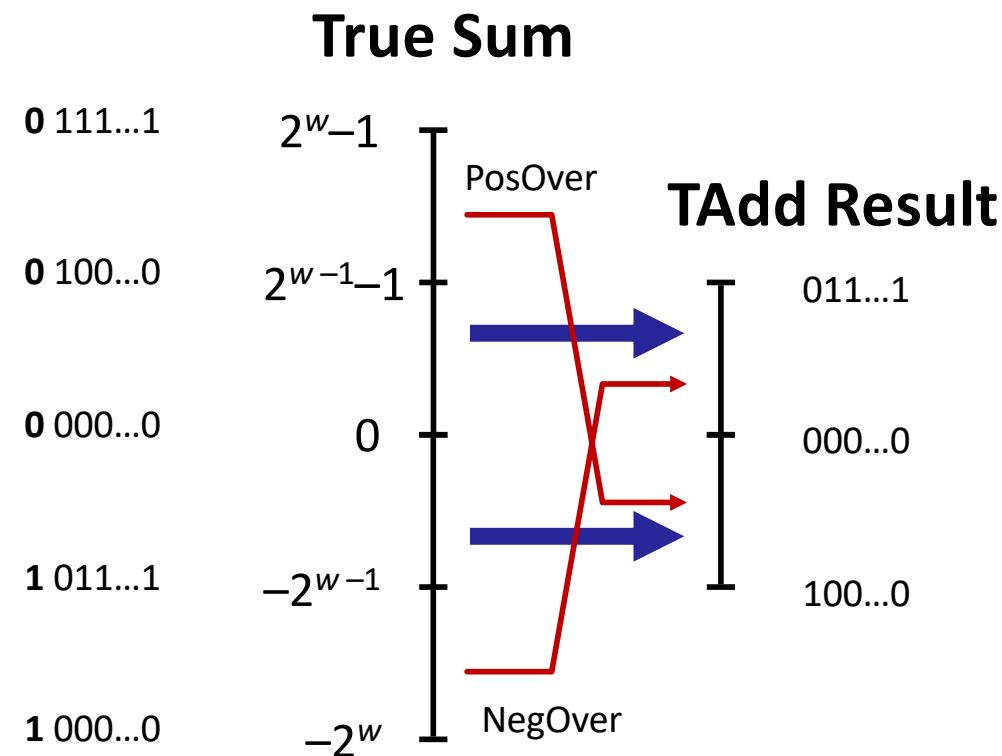
```
int s, t, u, v;  
s = (int) ((un  
t = u + v
```

- Will give  $s = t$

# TAdd Overflow

## ■ Functionality

- True sum requires  $w+1$  bits
- Drop off MSB
- Treat remaining bits as 2's comp. integer



# Visualizing 2's Complement Addition

## ■ Values

- 4-bit two's comp.
- Range from -8 to +7

## ■ Wraps Around

- If sum  $\geq 2^{w-1}$ 
  - Becomes negative
  - At most once
- If sum  $< -2^{w-1}$ 
  - Becomes positive
  - At most once

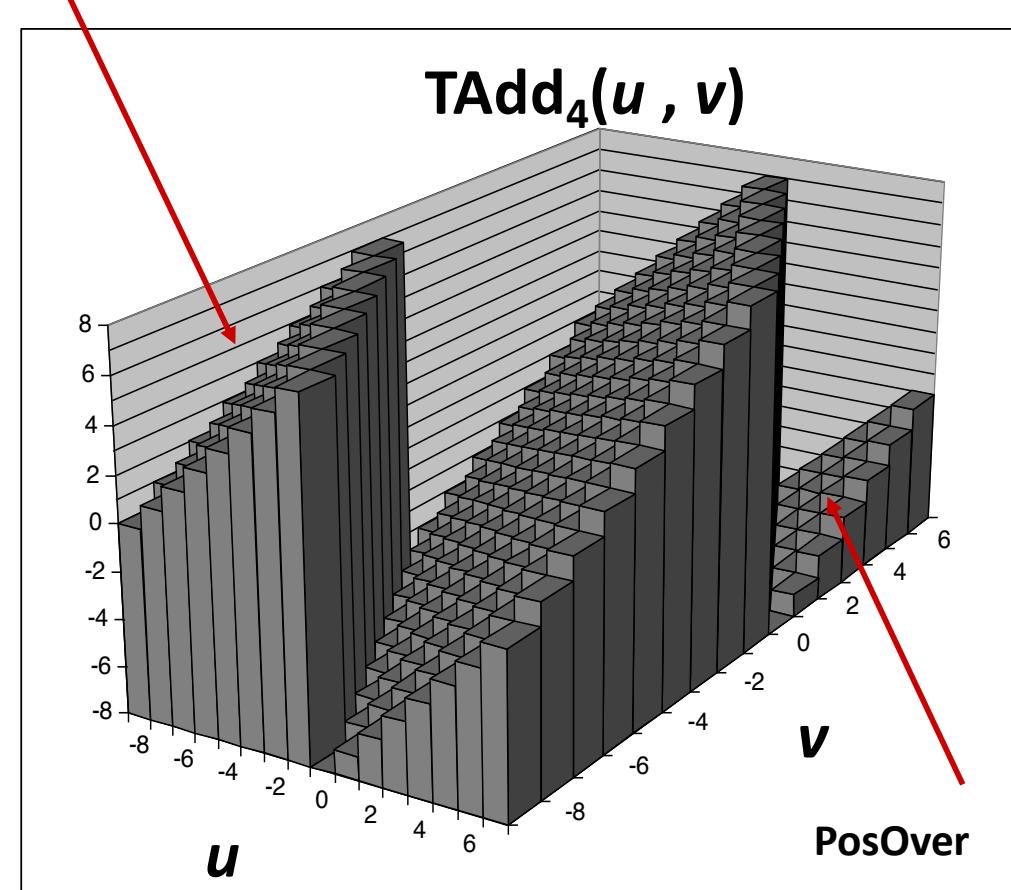
NegOver

TAdd<sub>4</sub>( $u$ ,  $v$ )

$u$

PosOver

$v$

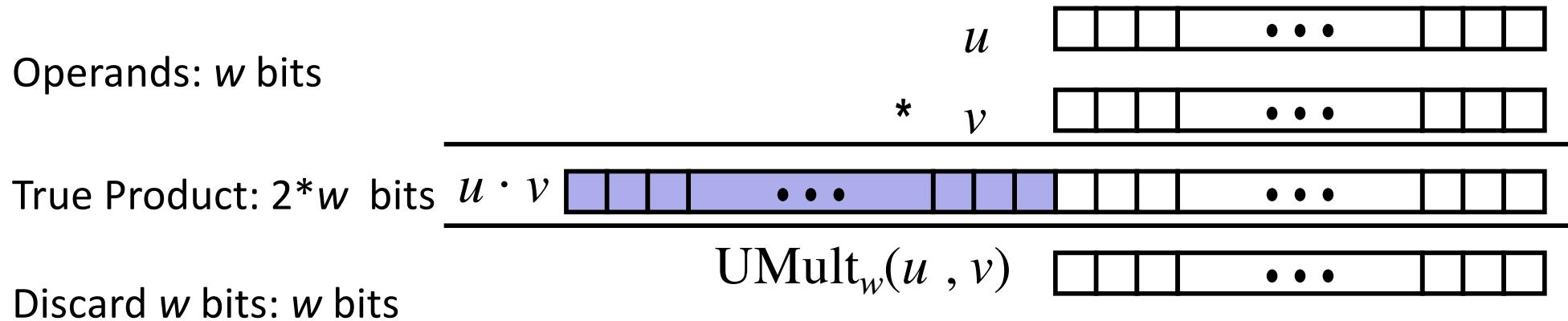


# Multiplication

- **Goal: Computing Product of  $w$ -bit numbers  $x, y$** 
  - Either signed or unsigned
- **But, exact results can be bigger than  $w$  bits**
  - Unsigned: up to  $2w$  bits
    - Result range:  $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to  $2w-1$  bits
    - Result range:  $x * y \geq (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max (positive): Up to  $2w$  bits, but only for  $(TMin_w)^2$ 
    - Result range:  $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$
- **So, maintaining exact results...**
  - would need to keep expanding word size with each product computed
  - is done in software, if needed
    - e.g., by “arbitrary precision” arithmetic packages

# Unsigned Multiplication in C

Operands:  $w$  bits



- **Standard Multiplication Function**

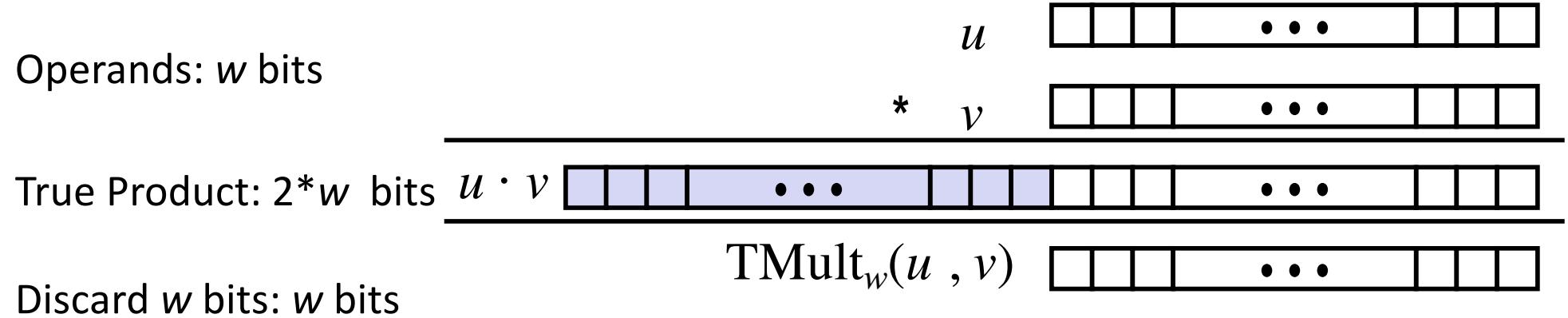
- Ignores high order  $w$  bits

- **Implements Modular Arithmetic**

$$\text{UMult}_w(u , v) = u \cdot v \bmod 2^w$$

# Signed Multiplication in C

Operands:  $w$  bits



## ■ Standard Multiplication Function

- Ignores high order  $w$  bits
- Some of which are different for signed vs. unsigned multiplication
- Lower bits are the same

# Power-of-2 Multiply with Shift

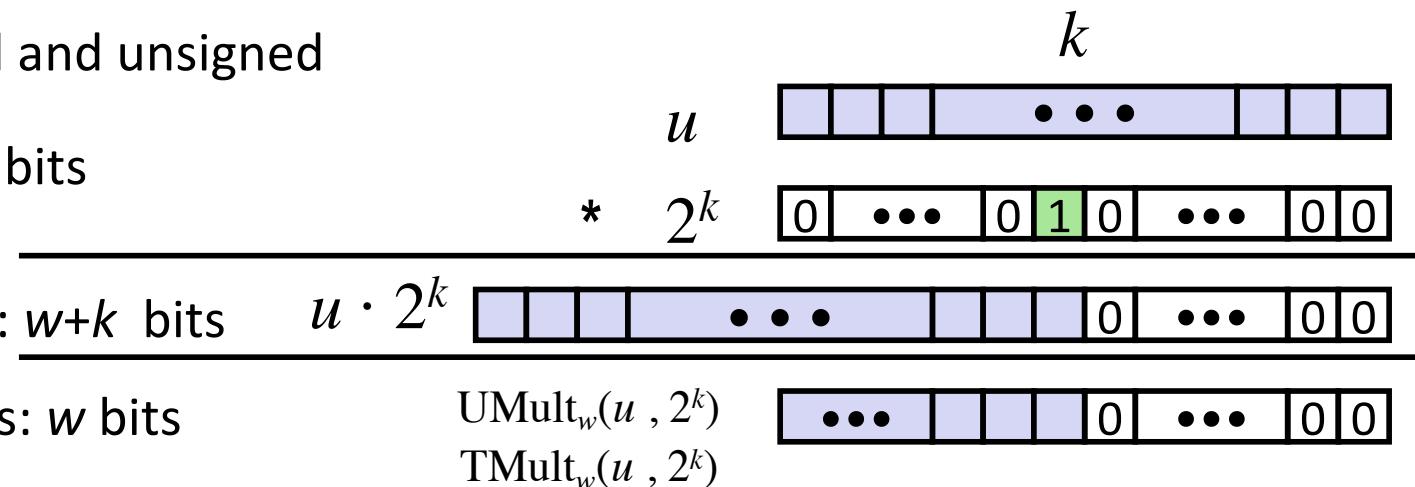
## ■ Operation

- $u \ll k$  gives  $u * 2^k$
- Both signed and unsigned

Operands:  $w$  bits

True Product:  $w+k$  bits

Discard  $k$  bits:  $w$  bits



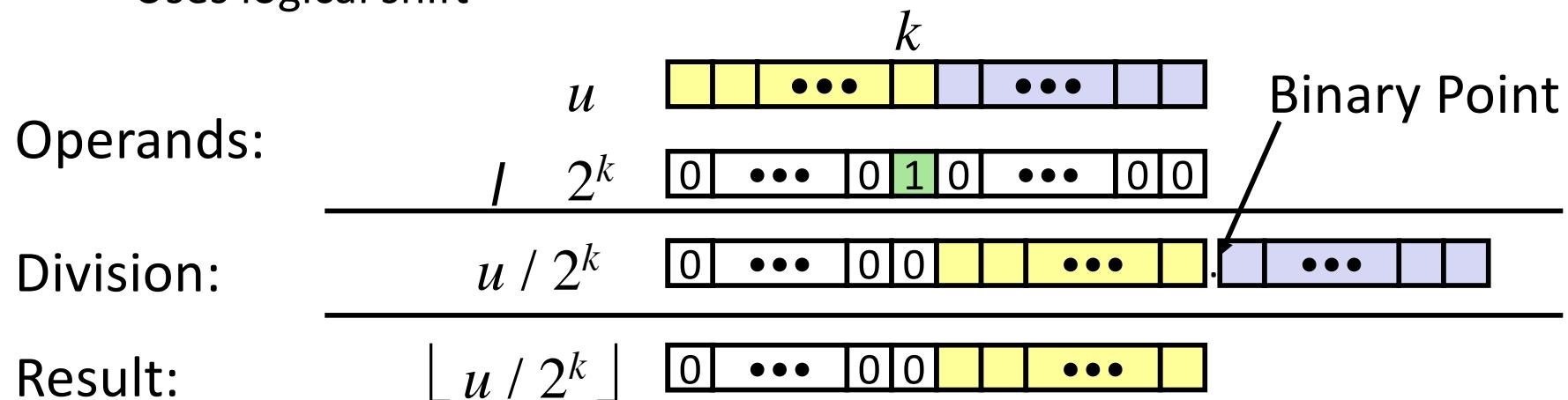
## ■ Examples

- $u \ll 3 == u * 8$
- $(u \ll 5) - (u \ll 3) == u * 24$
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically

# Unsigned Power-of-2 Divide with Shift

## ■ Quotient of Unsigned by Power of 2

- $u \gg k$  gives  $\lfloor u / 2^k \rfloor$
- Uses logical shift



	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	00011101 10110110
x >> 4	950.8125	950	03 B6	00000011 10110110
x >> 8	59.4257813	59	00 3B	00000000 00111011

# Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - **Summary**
- Representations in memory, pointers, strings

# Arithmetic: Basic Rules

## ■ Addition:

- Unsigned/signed: Normal addition followed by truncate, same operation on bit level
- Unsigned: addition mod  $2^w$ 
  - Mathematical addition + possible subtraction of  $2^w$
- Signed: modified addition mod  $2^w$  (result in proper range)
  - Mathematical addition + possible addition or subtraction of  $2^w$

## ■ Multiplication:

- Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
- Unsigned: multiplication mod  $2^w$
- Signed: modified multiplication mod  $2^w$  (result in proper range)

# Why Should I Use Unsigned?

## ■ *Don't use without understanding implications*

- Easy to make mistakes

```
unsigned i;  
for (i = cnt-2; i >= 0; i--)  
    a[i] += a[i+1];
```

- Can be very subtle

```
#define DELTA sizeof(int)  
int i;  
for (i = CNT; i-DELTA >= 0; i-= DELTA)  
    . . .
```

# Counting Down with Unsigned

- Proper way to use unsigned as loop index

```
unsigned i;  
for (i = cnt-2; i < cnt; i--)  
    a[i] += a[i+1];
```

- See Robert Seacord, *Secure Coding in C and C++*

- C Standard guarantees that unsigned addition will behave like modular arithmetic
  - $0 - 1 \rightarrow UMax$

- Even better

```
size_t i;  
for (i = cnt-2; i < cnt; i--)  
    a[i] += a[i+1];
```

- Data type **size\_t** defined as unsigned value with length = word size
- Code will work even if **cnt** = *UMax*
- What if **cnt** is signed and < 0?

# Why Should I Use Unsigned? (cont.)

- ***Do Use When Performing Modular Arithmetic***
  - Multiprecision arithmetic
- ***Do Use When Using Bits to Represent Sets***
  - Logical right shift, no sign extension

# Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Floating Point
- Representations in memory, pointers, strings

# IEEE Floating-Point Representation

## ■ Numeric form

- $V=(-1)^s \times M \times 2^E$ 
  - Sign bit **s** determines whether number is negative or positive
  - Significand **M** normally a fractional value in range [1.0,2.0)
  - Exponent **E** weights value by power of two

# IEEE Floating-Point Representation

## ■ Encoding



- **s** is sign bit
- **exp** field encodes  $E$
- **frac** field encodes  $M$

## ■ Sizes

- Single precision (32 bits): 8 exp bits, 23 frac bits
- Double precision (64 bits): 11 exp bits, 52 frac bits

# Normalized Values

- Condition
  - $\text{exp} \neq 000\dots0$  and  $\text{exp} \neq 111\dots1$
- Exponent coded as *biased value*
  - $E = Exp - Bias$ 
    - $Exp$  : unsigned value denoted by **exp**
    - $Bias$  : Bias value
      - Single precision: **127** ( $Exp: 1\dots254, E: -126\dots127$ )
      - Double precision: **1023** ( $Exp: 1\dots2046, E: -1022\dots1023$ )
      - In general:  $Bias = 2^{k-1} - 1$ , where  $k$  is the number of exponent bits

# Normalized Values

- Significand coded with implied leading 1
  - $m = 1.\text{xxxx...x}_2$ 
    - xxxx...x: bits of `frac`
    - Minimum when 000...0 ( $M = 1.0$ )
    - Maximum when 111...1 ( $M = 2.0 - \varepsilon$ )
    - Get extra leading bit for “free”

# Normalized Encoding Examples

- Value: 12345 (Hex: 0x3039)
- Binary bits: 11000000111001
- Fraction representation: 1.1000000111001 $\times 2^{13}$
- M: 100000011100100000000000
- E: 10001100 (140)
- Binary Encoding
  - 0100 0110 0100 0000 1110 0100 0000 0000
  - 4      6      4      0      E      4      0      0

# Denormalized Values

## ■ Condition

- $\text{exp} = 000\dots0$

## ■ Values

- Exponent Value:  $E = 1 - \text{Bias}$
- Significant Value  $m = 0.\text{xxx}\dots\text{x}_2$ 
  - $\text{xxx}\dots\text{x}$ : bits of  $\text{frac}$

# Denormalized Values

## ■ Cases

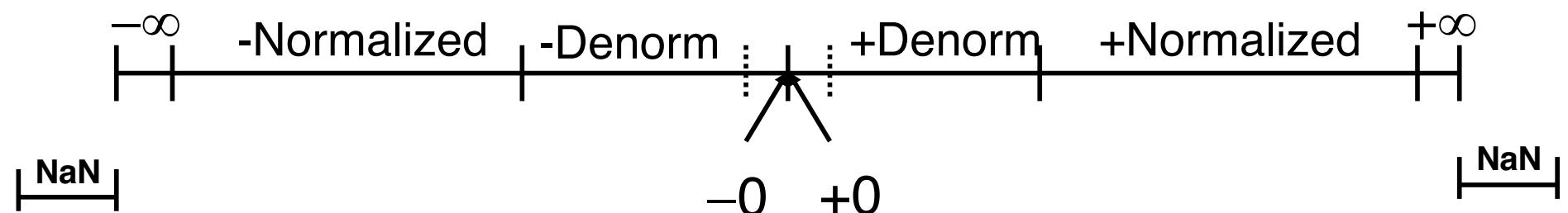
- $\text{exp} = 000\dots0, \text{frac} = 000\dots0$ 
  - Represents value 0
  - Note that have distinct values +0 and -0
- $\text{exp} = 000\dots0, \text{frac} \neq 000\dots0$ 
  - Numbers very close to 0.0
  - Lose precision as get smaller
  - “Gradual underflow”

# Special Values

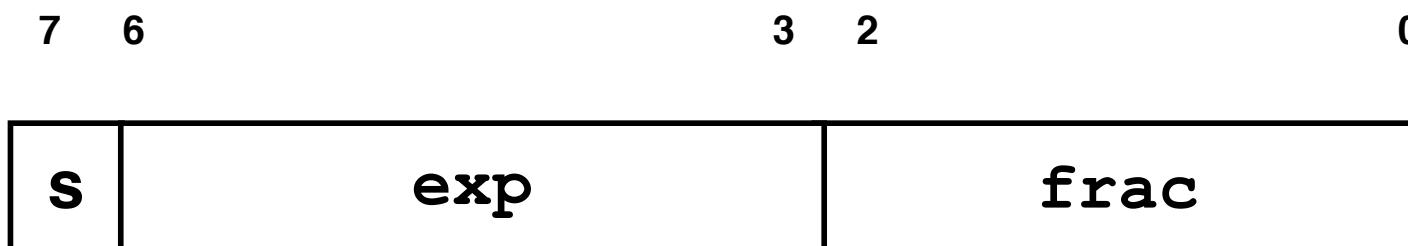
## ■ Condition

- s=0, exp = 111...1, frac=000...0                       $+\infty$
- s=1, exp = 111...1, frac=000...0                       $-\infty$
- exp = 111...1    NaN

# Summary of Real Number Encodings



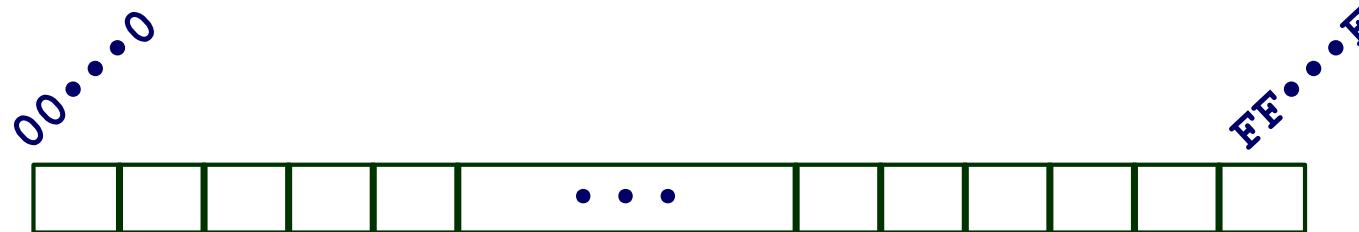
# 8-bit Floating-Point Representations



# Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Floating Point
- Representations in memory, pointers, strings

# Byte-Oriented Memory Organization



## ■ Programs refer to data by address

- Conceptually, envision it as a very large array of bytes
  - In reality, it's not, but can think of it that way
- An address is like an index into that array
  - and, a pointer variable stores an address

## ■ Note: system provides private address spaces to each “process”

- Think of a process as a program being executed
- So, a program can clobber its own data, but not that of others

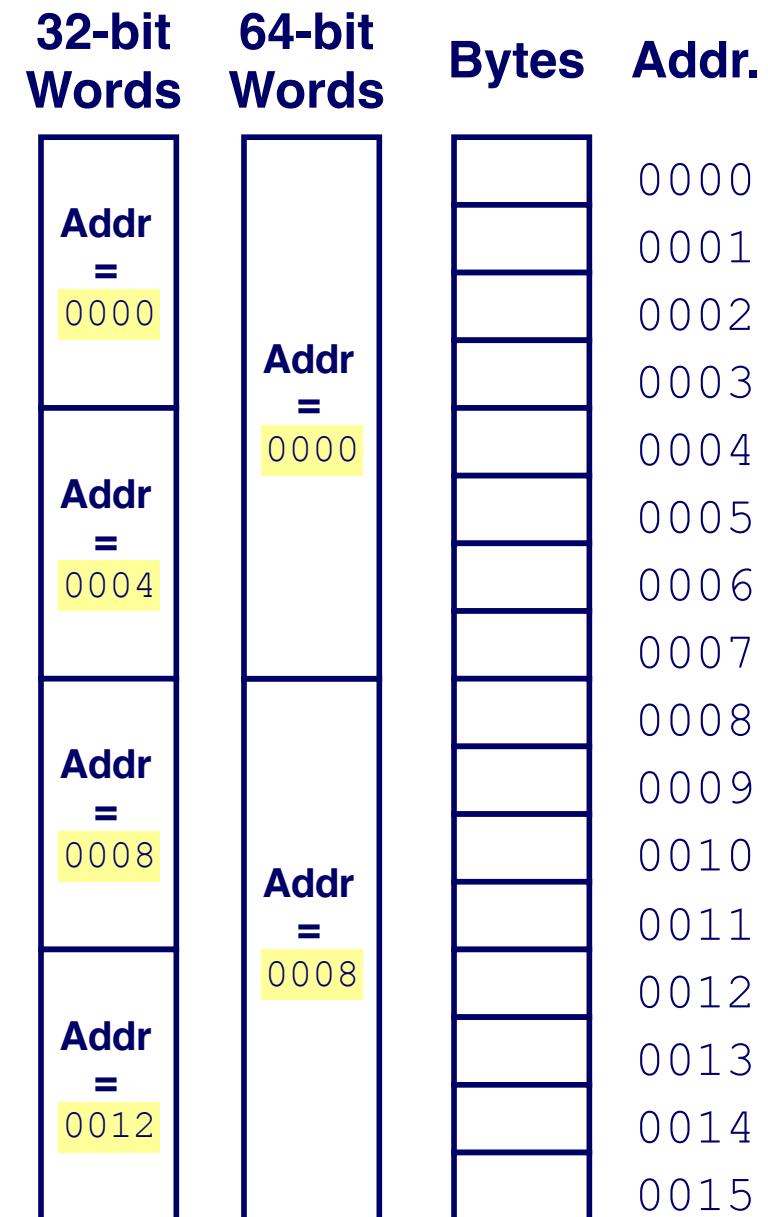
# Machine Words

- Any given computer has a “Word Size”
  - Nominal size of integer-valued data
    - and of addresses
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB ( $2^{32}$  bytes)
  - Increasingly, machines have 64-bit word size
    - Potentially, could have 18 EB (exabytes) of addressable memory
    - That's  $18.4 \times 10^{18}$
  - Machines still support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

# Word-Oriented Memory Organization

## ■ Addresses Specify Byte Locations

- Address of first byte in word
- Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



# Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
<b>char</b>	1	1	1
<b>short</b>	2	2	2
<b>int</b>	4	4	4
<b>long</b>	4	8	8
<b>float</b>	4	4	4
<b>double</b>	8	8	8
<b>long double</b>	-	-	10/16
pointer	4	8	8

# Byte Ordering

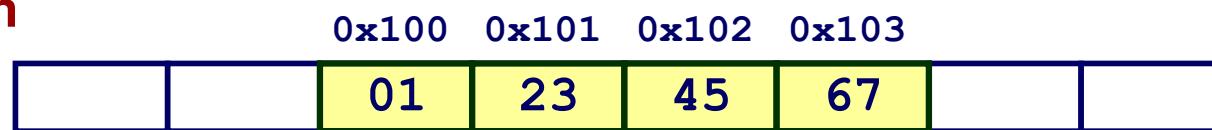
- So, how are the bytes within a multi-byte word ordered in memory?
- Conventions
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - Little Endian: x86, ARM processors running Android, iOS, and Windows
    - Least significant byte has lowest address

# Byte Ordering Example

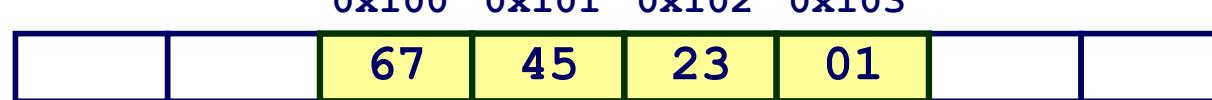
## ■ Example

- Variable x has 4-byte value of 0x01234567
- Address given by &x is 0x100

### BigEndian



### Little Endian



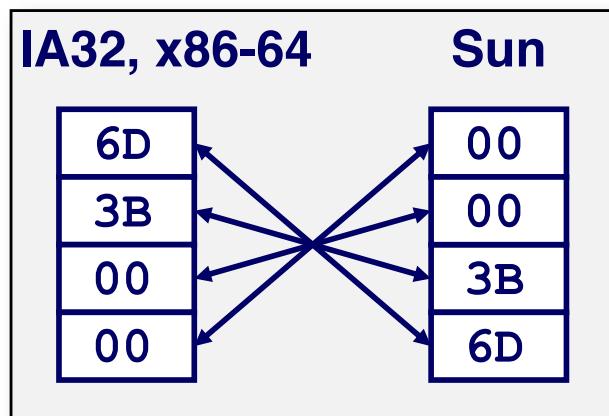
# Representing Integers

Decimal: 15213

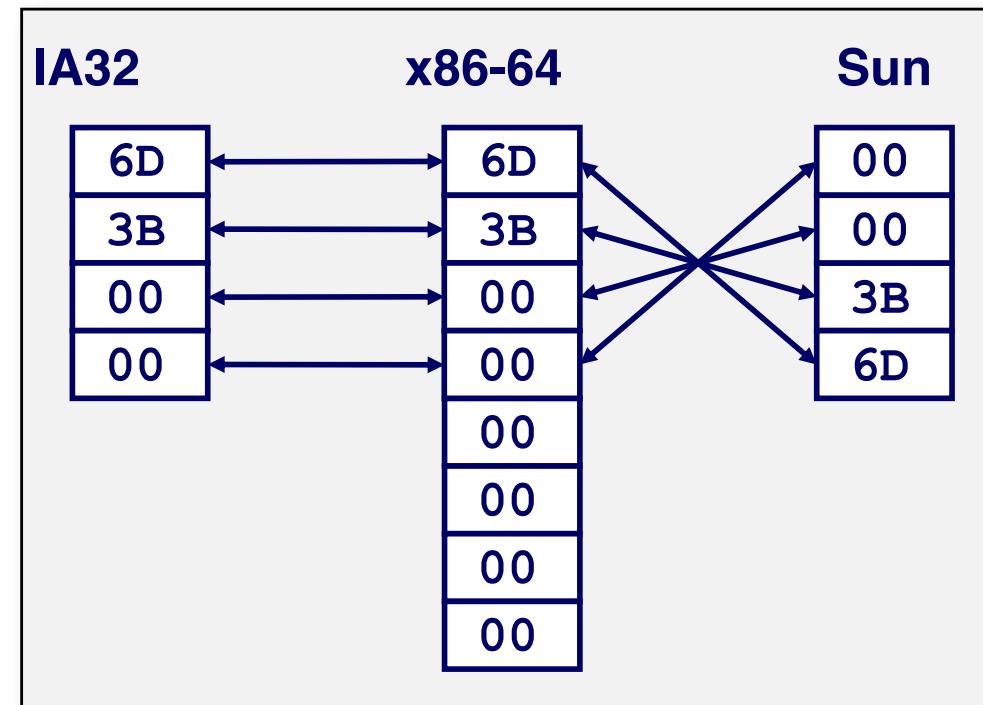
Binary: 0011 1011 0110 1101

Hex: 3 B 6 D

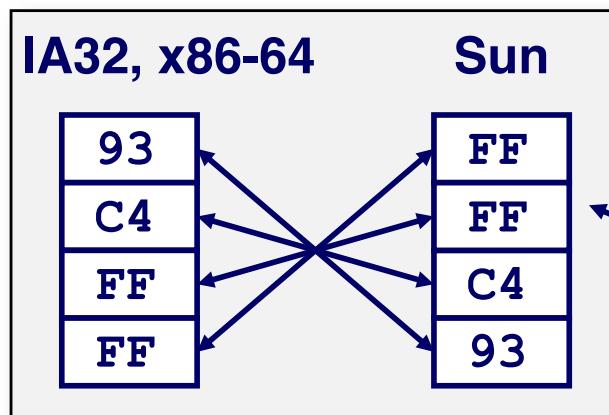
```
int A = 15213;
```



```
long int C = 15213;
```



```
int B = -15213;
```



Two's complement representation

# Examining Data Representations

## ■ Code to Print Byte Representation of Data

- Casting pointer to unsigned char \* allows treatment as a byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len) {
    size_t i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```

### Printf directives:

%p: Print pointer  
%x: Print Hexadecimal

# **show\_bytes Execution Example**

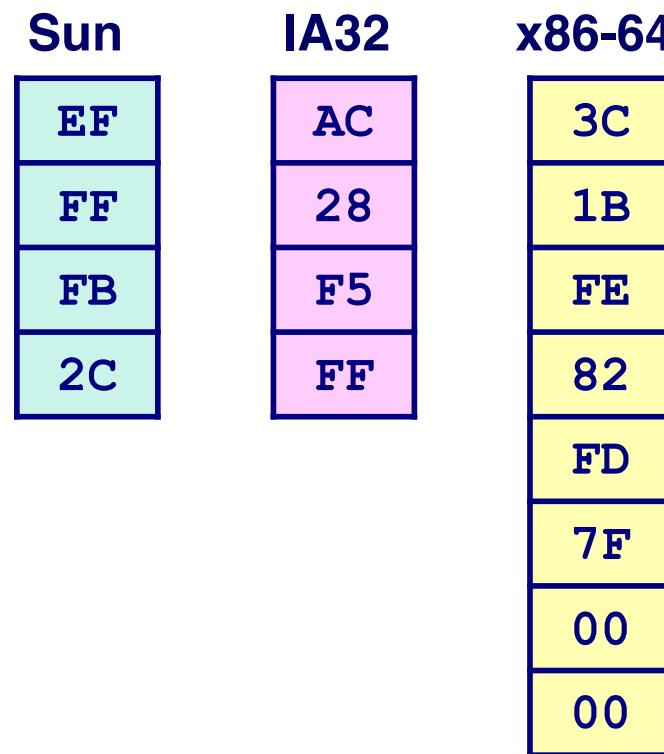
```
int a = 15213;  
printf("int a = 15213;\\n");  
show_bytes((pointer) &a, sizeof(int));
```

## **Result (Linux x86-64):**

```
int a = 15213;  
0x7fffb7f71dbc      6d  
0x7fffb7f71dbd      3b  
0x7fffb7f71dbe      00  
0x7fffb7f71dbf      00
```

# Representing Pointers

```
int B = -15213;  
int *P = &B;
```



Different compilers & machines assign different locations to objects  
Even get different results each time run program

# Representing Strings

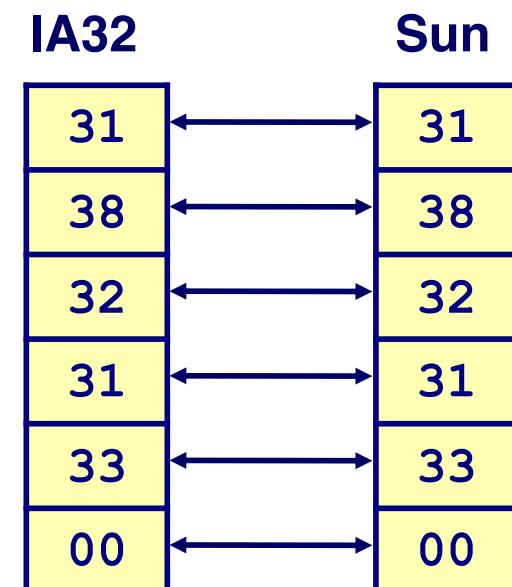
## ■ Strings in C

- Represented by array of characters
- Each character encoded in ASCII format
  - Standard 7-bit encoding of character set
  - Character “0” has code 0x30
    - Digit  $i$  has code  $0x30+i$
- String should be null-terminated
  - Final character = 0

## ■ Compatibility

- Byte ordering not an issue

```
char S[6] = "18213";
```



# Integer C Puzzles

## Initialization

```
int x = foo();  
  
int y = bar();  
  
unsigned ux = x;  
  
unsigned uy = y;
```

- $x < 0$    $((x*2) < 0)$
- $ux \geq 0$
- $x \& 7 == 7$    $(x << 30) < 0$
- $ux > -1$
- $x > y$    $-x < -y$
- $x * x \geq 0$
- $x > 0 \&\& y > 0$    $x + y > 0$
- $x \geq 0$    $-x \leq 0$
- $x \leq 0$    $-x \geq 0$
- $(x | -x) \gg 31 == -1$
- $ux \gg 3 == ux/8$
- $x \gg 3 == x/8$
- $x \& (x-1) != 0$

# Machine-Level Programming I: Basics

# 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.

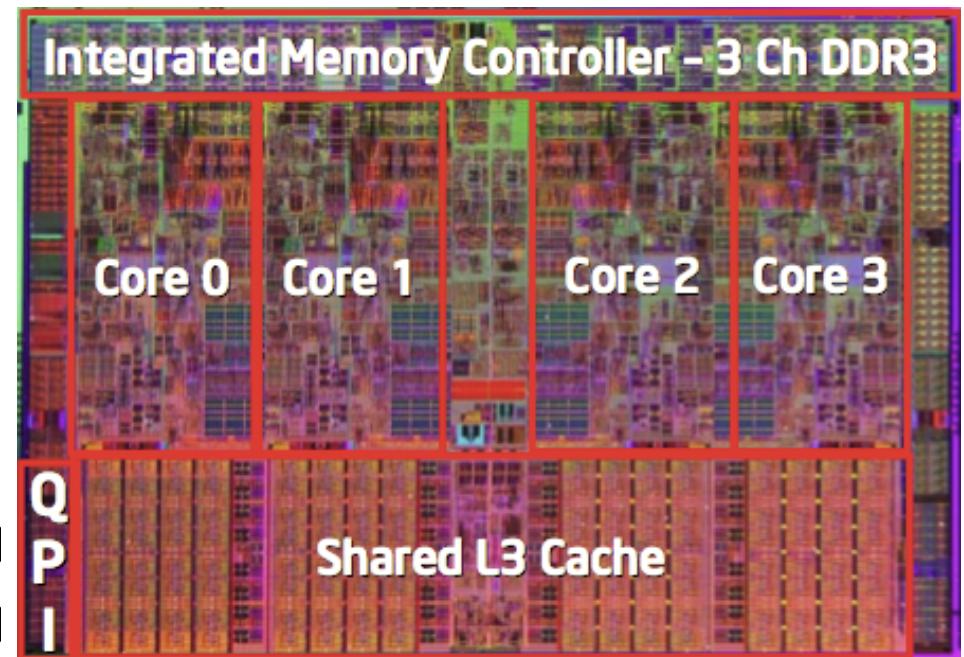
# Intel x86 Evolution: Milestones

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

# 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

# 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

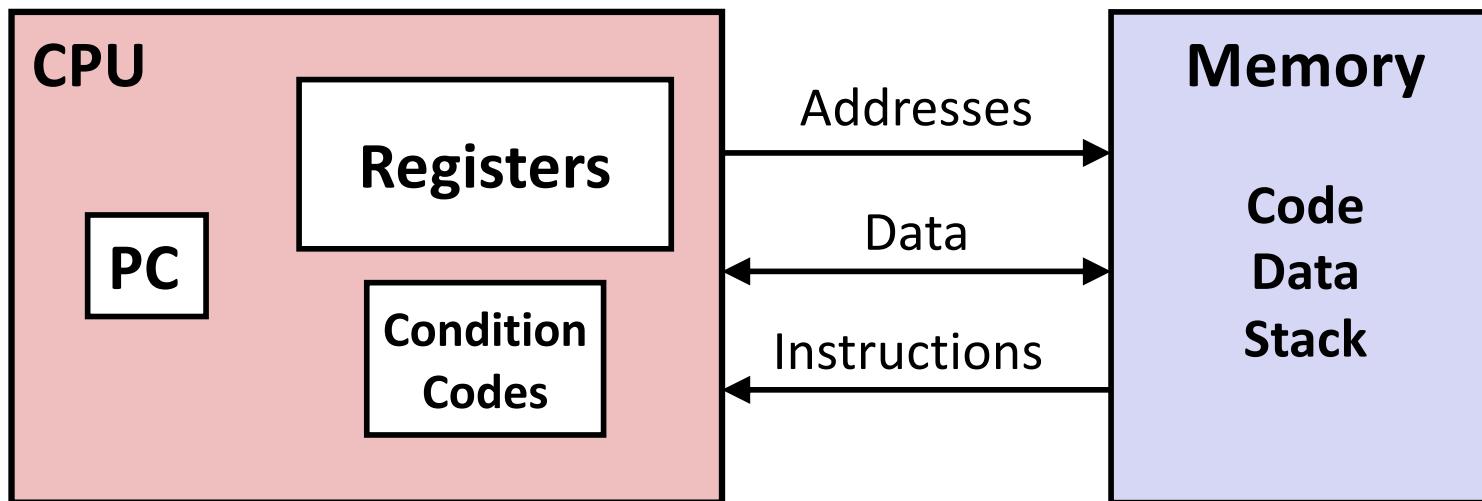
# Machine Programming I: Basics

- 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

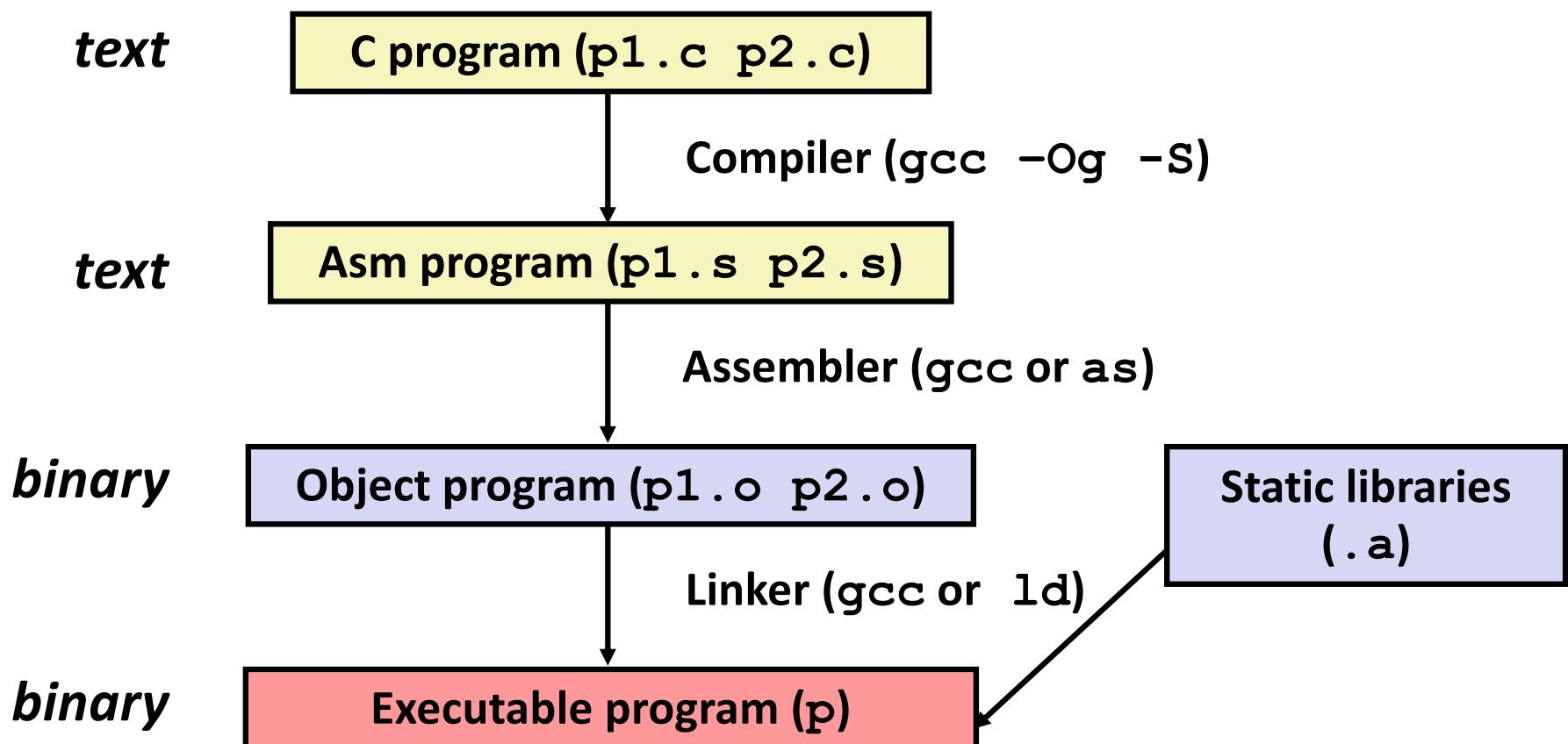


## Programmer-Visible State

- **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 with command

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

Produces file sum.s

**Warning:** Will get very different results on different 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

## ■ 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

## Code for `sumstore`

0x0400595:

0x53  
0x48  
0x89  
0xd3  
0xe8  
0xf2  
0xff  
0xff

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

0x03  
0x5b  
0xc3

# Machine Instruction Example

## ■ C Code

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

```
*dest = t;
```

## ■ Assembly

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

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

## ■ Object Code

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

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

# 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

# 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

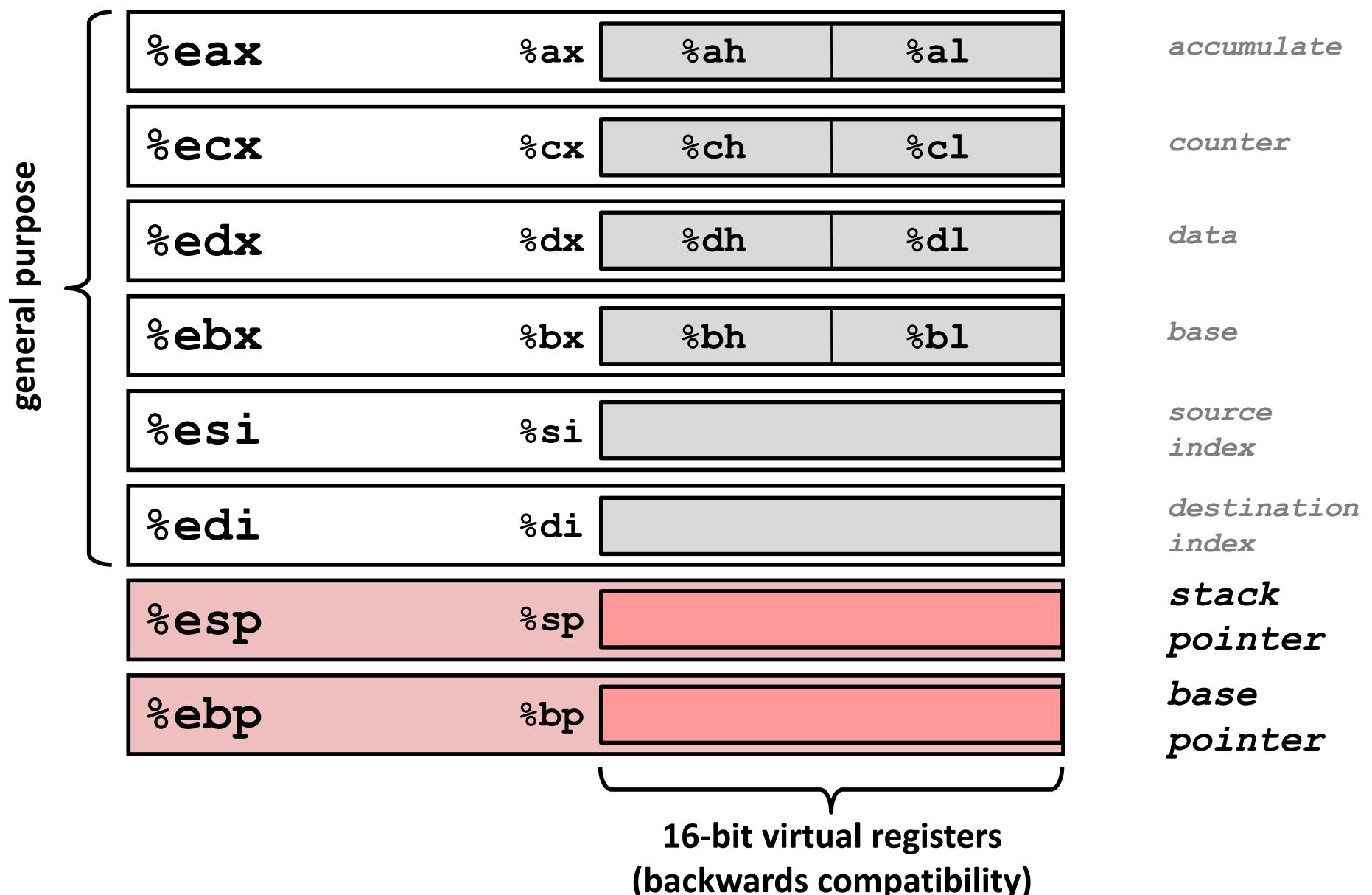
%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

%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)

# Some History: IA32 Registers

Origin  
(mostly obsolete)



# 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`

# movq Operand Combinations

*Cannot do memory-memory transfer with a single instruction*

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

# 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

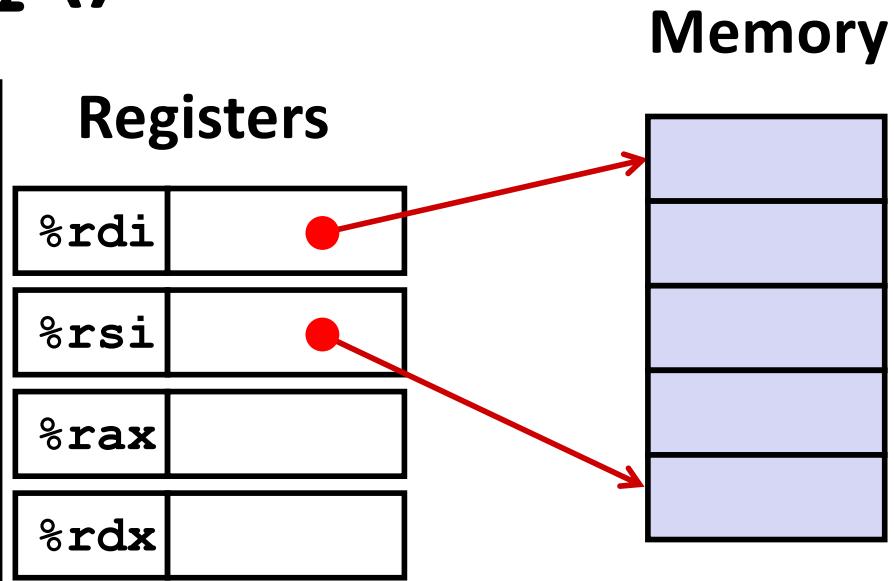
```
void swap
    (long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

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:

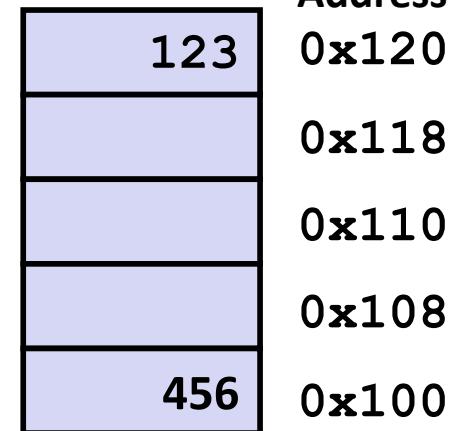
```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

# Understanding Swap()

Registers

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

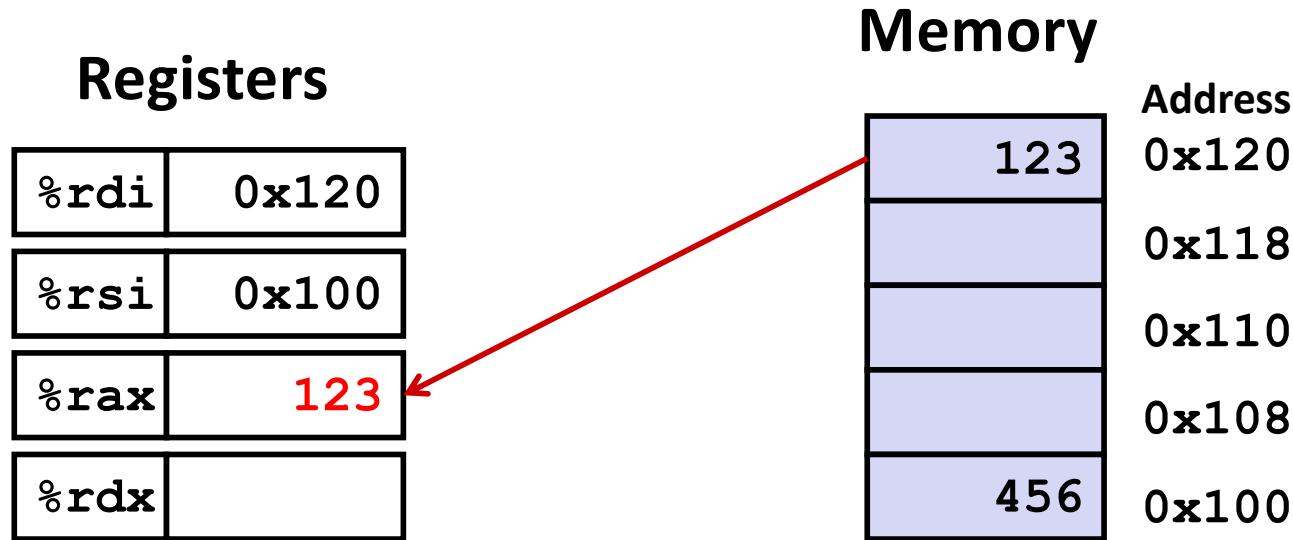
Memory



swap:

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

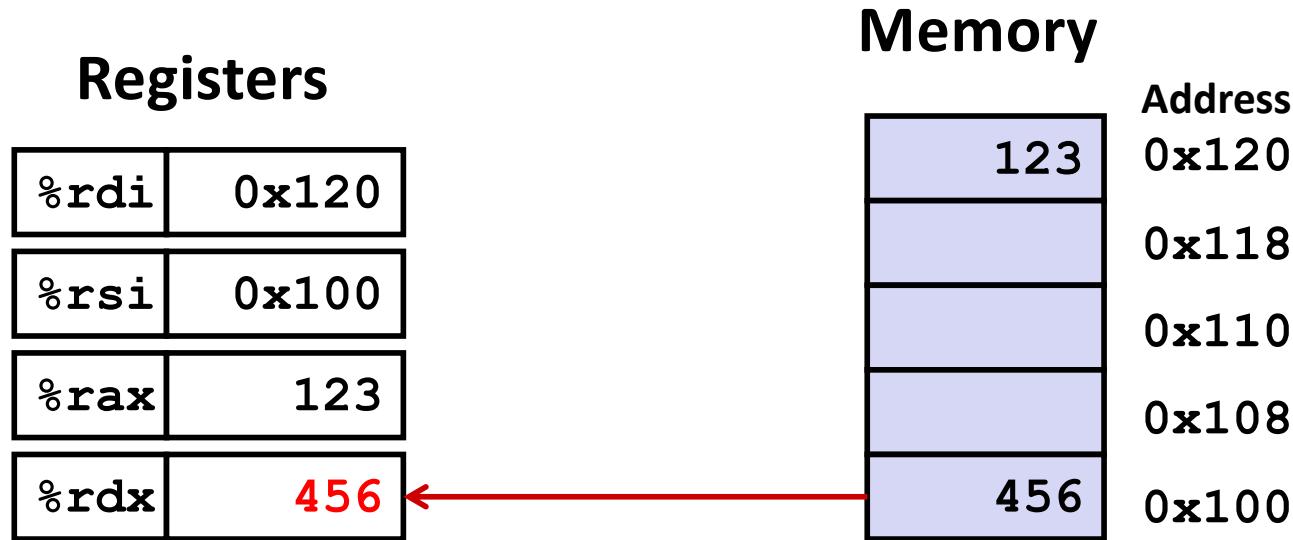
# Understanding Swap()



swap:

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

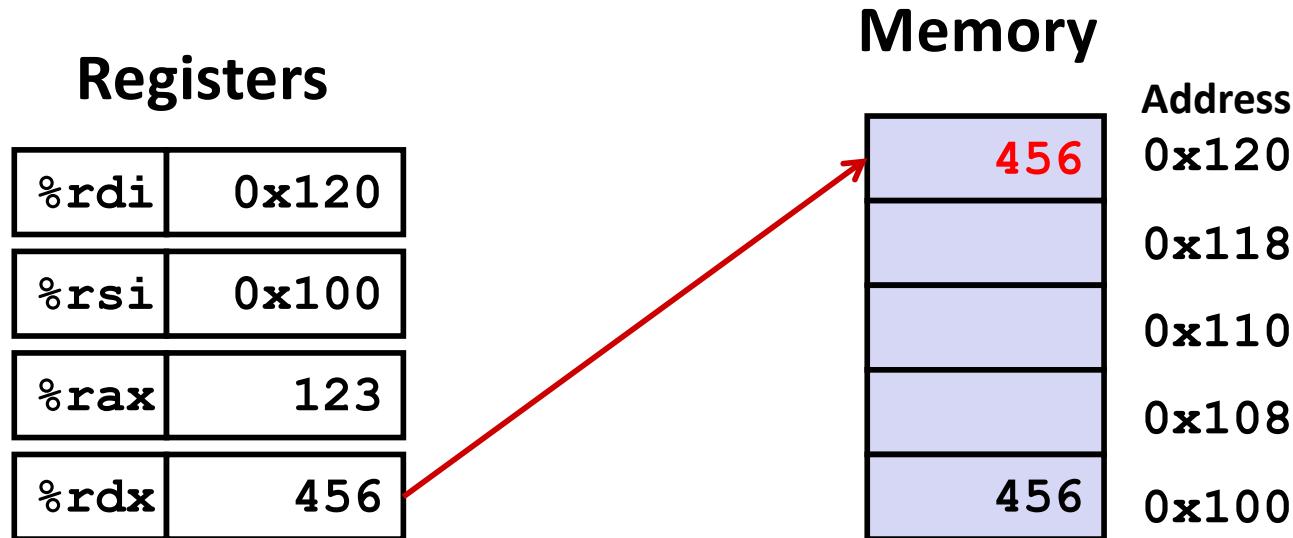
# Understanding Swap()



swap:

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

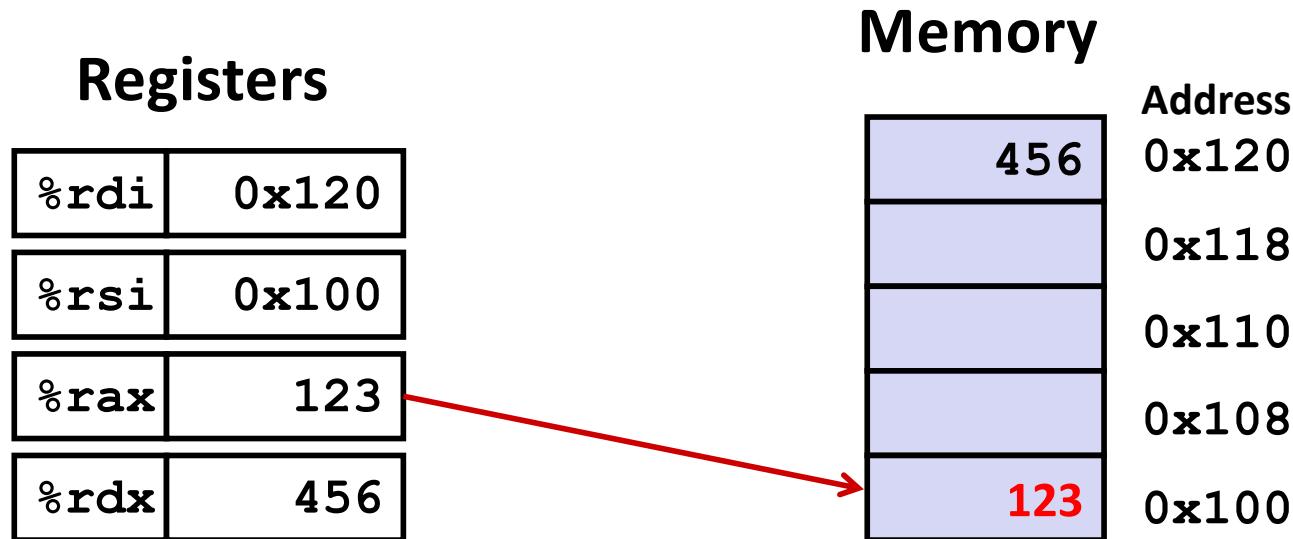
# Understanding Swap()



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

%rdx	0xf000
%rcx	0x0100

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

# 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		
addq	Src,Dest	Dest = Dest + Src	
subq	Src,Dest	Dest = Dest – Src	
imulq	Src,Dest	Dest = Dest * Src	
salq	Src,Dest	Dest = Dest << Src	Also called shlq
sarq	Src,Dest	Dest = Dest >> Src	Arithmetic
shrq	Src,Dest	Dest = Dest >> Src	Logical
xorq	Src,Dest	Dest = Dest ^ Src	
andq	Src,Dest	Dest = Dest & Src	
orq	Src,Dest	Dest = Dest   Src	

- 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

# **Machine-Level Programming II: Control**

# Control

- **Control: Condition codes**
- Conditional branches
- Loops
- Switch Statements

# Processor State (x86-64, Partial)

## ■ Information about currently executing program

- Temporary data ( `%rax`, ... )
- Location of runtime stack ( `%rsp` )
- Location of current code control point ( `%rip`, ... )
- Status of recent tests ( CF, ZF, SF, OF )

Current stack top

Registers

<code>%rax</code>
<code>%rbx</code>
<code>%rcx</code>
<code>%rdx</code>
<code>%rsi</code>
<code>%rdi</code>
<code>%rsp</code>
<code>%rbp</code>

`%rip`

<code>%r8</code>
<code>%r9</code>
<code>%r10</code>
<code>%r11</code>
<code>%r12</code>
<code>%r13</code>
<code>%r14</code>
<code>%r15</code>

Instruction pointer

CF

ZF

SF

OF

Condition codes

# Condition Codes (Implicit Setting)

## ■ Single bit registers

- CF Carry Flag (for unsigned) SF Sign Flag (for signed)
- ZF Zero Flag OF Overflow Flag (for signed)

## ■ Implicitly set (think of it as side effect) by arithmetic operations

Example: **addq Src,Dest**  $\leftrightarrow t = a+b$

CF set if carry out from most significant bit (unsigned overflow)

ZF set if  $t == 0$

SF set if  $t < 0$  (as signed)

OF set if two's-complement (signed) overflow

$(a>0 \ \&\& \ b>0 \ \&\& \ t<0) \ || \ (a<0 \ \&\& \ b<0 \ \&\& \ t>=0)$

## ■ Not set by **leaq** instruction

# Condition Codes (Explicit Setting: Compare)

## ■ Explicit Setting by Compare Instruction

- **cmpq** Src2, Src1
- **cmpq b, a** like computing  $a-b$  without setting destination
- **CF set** if carry out from most significant bit (used for unsigned comparisons)
- **ZF set** if  $a == b$
- **SF set** if  $(a-b) < 0$  (as signed)
- **OF set** if two's-complement (signed) overflow  
$$(a>0 \ \&\& \ b<0 \ \&\& \ (a-b)<0) \ \|\ (a<0 \ \&\& \ b>0 \ \&\& \ (a-b)>0)$$

# Condition Codes (Explicit Setting: Test)

## ■ Explicit Setting by Test instruction

- **testq Src2, Src1**
  - **testq b, a** like computing **a&b** without setting destination
- Sets condition codes based on value of Src1 & Src2
- Useful to have one of the operands be a mask
- ZF set when **a&b == 0**
- SF set when **a&b < 0**

# Reading Condition Codes

## ■ SetX Instructions

- Set low-order byte of destination to 0 or 1 based on combinations of condition codes
- Does not alter remaining 7 bytes

SetX	Condition	Description
<b>sete</b>	<b>ZF</b>	Equal / Zero
<b>setne</b>	<b>~ZF</b>	Not Equal / Not Zero
<b>sets</b>	<b>SF</b>	Negative
<b>setns</b>	<b>~SF</b>	Nonnegative
<b>setg</b>	<b>~(SF^OF) &amp; ~ZF</b>	Greater (Signed)
<b>setge</b>	<b>~(SF^OF)</b>	Greater or Equal (Signed)
<b>setl</b>	<b>(SF^OF)</b>	Less (Signed)
<b>setle</b>	<b>(SF^OF)   ZF</b>	Less or Equal (Signed)
<b>seta</b>	<b>~CF &amp; ~ZF</b>	Above (unsigned)
<b>setb</b>	<b>CF</b>	Below (unsigned)

# x86-64 Integer Registers

%rax	%al	
%rbx	%bl	
%rcx	%cl	
%rdx	%dl	
%rsi	%sil	
%rdi	%dil	
%rsp	%spl	
%rbp	%bpl	
%r8		%r8b
%r9		%r9b
%r10		%r10b
%r11		%r11b
%r12		%r12b
%r13		%r13b
%r14		%r14b
%r15		%r15b

- Can reference low-order byte

# Reading Condition Codes (Cont.)

## ■ SetX Instructions:

- Set single byte based on combination of condition codes

## ■ One of addressable byte registers

- Does not alter remaining bytes
- Typically use **movzbl** to finish job
  - 32-bit instructions also set upper 32 bits to

```
int gt (long x, long y)
{
    return x > y;
}
```

Register	Use(s)
%rdi	Argument <b>x</b>
%rsi	Argument <b>y</b>
%rax	Return value

```
cmpq    %rsi, %rdi    # Compare x:y
setg    %al             # Set when >
movzbl  %al, %eax      # Zero rest of %eax
ret
```

# Control

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

# Jumping

## ■ jX Instructions

- Jump to different part of code depending on condition codes

jX	Condition	Description
<b>jmp</b>	<b>1</b>	Unconditional
<b>je</b>	<b>ZF</b>	Equal / Zero
<b>jne</b>	<b>~ZF</b>	Not Equal / Not Zero
<b>js</b>	<b>SF</b>	Negative
<b>jns</b>	<b>~SF</b>	Nonnegative
<b>jg</b>	<b>~(SF^OF) &amp; ~ZF</b>	Greater (Signed)
<b>jge</b>	<b>~(SF^OF)</b>	Greater or Equal (Signed)
<b>jl</b>	<b>(SF^OF)</b>	Less (Signed)
<b>jle</b>	<b>(SF^OF)   ZF</b>	Less or Equal (Signed)
<b>ja</b>	<b>~CF &amp; ~ZF</b>	Above (unsigned)
<b>jb</b>	<b>CF</b>	Below (unsigned)

# Conditional Branch Example (Old Style)

## ■ Generation

```
shark> gcc -Og -S -fno-if-conversion control.c
```

```
long absdiff
    (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

absdiff:

```
    cmpq    %rsi, %rdi  # x:y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:      # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

# Expressing with Goto Code

- C allows goto statement
- Jump to position designated by label

```
long absdiff
    (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j
    (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
Else:
    result = y-x;
Done:
    return result;
}
```

# General Conditional Expression Translation (Using Branches)

## C Code

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

## Goto Version

```
ntest = !Test;
if (ntest) goto Else;
val = Then_Expr;
goto Done;
Else:
    val = Else_Expr;
Done:
    . . .
```

- Create separate code regions for then & else expressions
- Execute appropriate one

# Using Conditional Moves

## ■ Conditional Move Instructions

- Instruction supports:  
if (Test) Dest  $\leftarrow$  Src
- Supported in post-1995 x86 processors
- GCC tries to use them
  - But, only when known to be safe

## ■ Why?

- Branches are very disruptive to instruction flow through pipelines
- Conditional moves do not require control transfer

### C Code

```
val = Test  
? Then_Expr  
: Else_Expr;
```

### Goto Version

```
result = Then_Expr;  
eval = Else_Expr;  
nt = !Test;  
if (nt) result = eval;  
return result;
```

# Conditional Move Example

```
long absdiff
    (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

```
absdiff:
    movq    %rdi, %rax    # x
    subq    %rsi, %rax    # result = x-y
    movq    %rsi, %rdx
    subq    %rdi, %rdx    # eval = y-x
    cmpq    %rsi, %rdi    # x:y
    cmovle %rdx, %rax    # if <=, result = eval
    ret
```

# Bad Cases for Conditional Move

## Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

## Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

## Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

# Control

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

# “Do-While” Loop Example

## C Code

```
long pcount_do
(unsigned long x) {
    long result = 0;
    do {
        result += x & 0x1;
        x >>= 1;
    } while (x);
    return result;
}
```

## Goto Version

```
long pcount_goto
(unsigned long x) {
    long result = 0;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

- Count number of 1's in argument **x** (“popcount”)
- Use conditional branch to either continue looping or to exit loop

# “Do-While” Loop Compilation

## Goto Version

```
long pcount_goto
(unsigned long x) {
    long result = 0;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

Register	Use(s)
%rdi	Argument <b>x</b>
%rax	<b>result</b>

```
        movq    $0, %rax      #  result = 0
.L2:
        movq    %rdi, %rdx
        andq    $1, %rdx      #  t = x & 0x1
        addq    %rdx, %rax    #  result += t
        shrq    %rdi          #  x >>= 1
        jne     .L2           #  if (x) goto loop
        ret
```

# General “Do-While” Translation

## C Code

```
do  
    Body  
    while (Test) ;
```

■ **Body:**

```
{  
    Statement1;  
    Statement2;  
    ...  
    Statementn;  
}
```

## Goto Version

```
loop:  
    Body  
    if (Test)  
        goto loop
```

# General “While” Translation #1

- “Jump-to-middle” translation
- Used with -Og

While version

```
while (Test)
  Body
```



Goto Version

```
goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```

# While Loop Example #1

## C Code

```
long pcount_while
(unsigned long x) {
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

## Jump to Middle

```
long pcount_goto_jtm
(unsigned long x) {
    long result = 0;
    goto test;
loop:
    result += x & 0x1;
    x >>= 1;
test:
    if(x) goto loop;
    return result;
}
```

- Compare to do-while version of function
- Initial goto starts loop at test

# General “While” Translation #2

## While version

```
while (Test)
```

*Body*

- “Do-while” conversion
- Used with -O1

## Do-While Version

```
if (!Test)
    goto done;
do
    Body
    while (Test);
done:
```

## Goto Version

```
if (!Test)
    goto done;
loop:
    Body
    if (Test)
        goto loop;
done:
```

# While Loop Example #2

## C Code

```
long pcount_while
(unsigned long x) {
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

## Do-While Version

```
long pcount_goto_dw
(unsigned long x) {
    long result = 0;
    if (!x) goto done;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
done:
    return result;
}
```

- Compare to do-while version of function
- Initial conditional guards entrance to loop

## Jump to Middle

```
long pcount_goto_jtm
(unsigned long x) {
    long result = 0;
    goto test;
loop:
    result += x & 0x1;
    x >>= 1;
test:
    if(x) goto loop;
    return result;
}
```

## Do-While Version

```
long pcount_goto_dw
(unsigned long x) {
    long result = 0;
    if (!x) goto done;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
done:
    return result;
}
```

# “For” Loop Form

## General Form

```
for (Init; Test; Update)  
    Body
```

```
#define WSIZE 8*sizeof(int)  
long pcount_for  
(unsigned long x)  
{  
    size_t i;  
    long result = 0;  
    for (i = 0; i < WSIZE; i++)  
    {  
        unsigned bit =  
            (x >> i) & 0x1;  
        result += bit;  
    }  
    return result;  
}
```

### Init

```
i = 0
```

### Test

```
i < WSIZE
```

### Update

```
i++
```

### Body

```
{  
    unsigned bit =  
        (x >> i) & 0x1;  
    result += bit;  
}
```

# “For” Loop → While Loop

## For Version

```
for (Init; Test; Update)
```

*Body*



## While Version

```
Init;
```

```
while (Test) {
```

*Body*

*Update*;

```
}
```

# For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{  
    unsigned bit =  
        (x >> i) & 0x1;  
    result += bit;  
}
```

```
long pcount_for_while  
    (unsigned long x)  
{  
    size_t i;  
    long result = 0;  
    i = 0;  
    while (i < WSIZE)  
    {  
        unsigned bit =  
            (x >> i) & 0x1;  
        result += bit;  
        i++;  
    }  
    return result;  
}
```

# “For” Loop Do-While Conversion

## C Code

```
long pcount_for
(unsigned long x)
{
    size_t i;
    long result = 0;
    for (i = 0; i < WSIZE; i++)
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

## Goto Version

```
long pcount_for_goto_dw
(unsigned long x) {
    size_t i;
    long result = 0;
    i = 0;
    if (! (i < WSIZE)) Init
        goto done; ! Test
loop:
{
    unsigned bit =
        (x >> i) & 0x1; Body
    result += bit;
}
i++; Update
if (i < WSIZE) Test
    goto loop;
done:
    return result;
}
```

- Initial test can be optimized away

# Control

- Control: Condition codes
- Conditional branches
- Loops
- **Switch Statements**

```
long switch_eg
    (long x, long y, long z)
{
    long w = 1;
    switch(x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
            /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

# Switch Statement Example

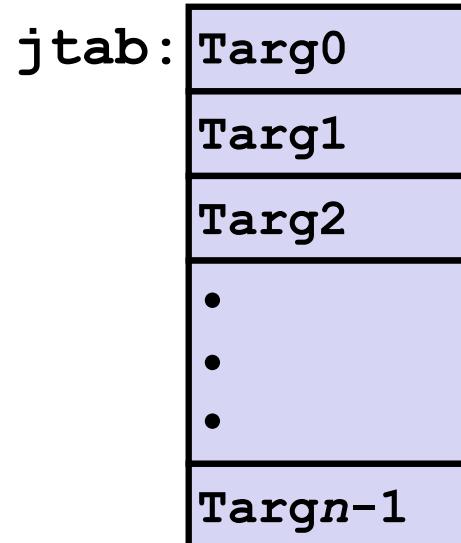
- **Multiple case labels**
  - Here: 5 & 6
- **Fall through cases**
  - Here: 2
- **Missing cases**
  - Here: 4

# Jump Table Structure

## Switch Form

```
switch(x) {  
    case val_0:  
        Block 0  
    case val_1:  
        Block 1  
    . . .  
    case val_{n-1}:  
        Block n-1  
}
```

## Jump Table



## Jump Targets

Targ0:

Code Block 0

Targ1:

Code Block 1

Targ2:

Code Block 2

•  
•  
•

Targ{n-1}:

Code Block n-1

## Translation (Extended C)

```
goto *JTab[x];
```

# Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

Setup:

switch\_eg:

```
    movq    %rdx, %rcx
    cmpq    $6, %rdi    # x:6
    ja      .L8
    jmp    * .L4(,%rdi,8)
```

What range of values  
takes default?

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Note that w not  
initialized here

# Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi      # x:6
    ja      .L8          # Use default
Indirect   jmp    * .L4(,%rdi,8) # goto *JTab[x]
jump
```

Jump table

```
.section .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
```

# Assembly Setup Explanation

## ■ Table Structure

- Each target requires 8 bytes
- Base address at `.L4`

## ■ Jumping

- Direct: `jmp .L8`
- Jump target is denoted by label `.L8`
- Indirect: `jmp * .L4(,%rdi,8)`
- Start of jump table: `.L4`
- Must scale by factor of 8 (addresses are 8 bytes)
- Fetch target from effective Address `.L4 + x*8`
  - Only for  $0 \leq x \leq 6$

## Jump table

```
.section    .rodata
.align 8
.L4:
.quad      .L8    # x = 0
.quad      .L3    # x = 1
.quad      .L5    # x = 2
.quad      .L9    # x = 3
.quad      .L8    # x = 4
.quad      .L7    # x = 5
.quad      .L7    # x = 6
```

# Jump Table

## Jump table

```
.section    .rodata
.align 8
.L4:
.quad      .L8    # x = 0
.quad      .L3    # x = 1
.quad      .L5    # x = 2
.quad      .L9    # x = 3
.quad      .L8    # x = 4
.quad      .L7    # x = 5
.quad      .L7    # x = 6
```

```
switch(x) {
    case 1:          // .L3
        w = y*z;
        break;
    case 2:          // .L5
        w = y/z;
        /* Fall Through */
    case 3:          // .L9
        w += z;
        break;
    case 5:
    case 6:          // .L7
        w -= z;
        break;
    default:         // .L8
        w = 2;
}
```

# Code Blocks ( $x == 1$ )

```
switch(x) {  
    case 1:          // .L3  
        w = y*z;  
        break;  
    . . .  
}
```

```
.L3:  
    movq    %rsi, %rax # y  
    imulq   %rdx, %rax # y*z  
    ret
```

Register	Use(s)
%rdi	Argument <b>x</b>
%rsi	Argument <b>y</b>
%rdx	Argument <b>z</b>
%rax	Return value

# Handling Fall-Through

```
long w = 1;  
.  
.  
switch(x) {  
.  
. . .  
case 2:  
    w = y/z;  
    /* Fall Through */  
case 3:  
    w += z;  
    break;  
.  
. . .  
}
```

```
case 2:  
    w = y/z;  
    goto merge;
```

```
case 3:  
    w = 1;  
  
merge:  
    w += z;
```

# Code Blocks ( $x == 2$ , $x == 3$ )

```
long w = 1;  
.  
.  
.  
switch(x) {  
.  
.  
.  
case 2:  
    w = y/z;  
    /* Fall Through */  
case 3:  
    w += z;  
    break;  
.  
.  
}
```

```
.L5:          # Case 2  
    movq    %rsi, %rax  
    cqto  
    idivq   %rcx      # y/z  
    jmp     .L6        # goto merge  
.L9:          # Case 3  
    movq    $1, %rax  # w = 1  
.L6:          # merge:  
    addq    %rcx, %rax # w += z  
    ret
```

Register	Use(s)
%rdi	Argument <b>x</b>
%rsi	Argument <b>y</b>
%rdx	Argument <b>z</b>
%rax	Return value

# Code Blocks ( $x == 5$ , $x == 6$ , default)

```
switch(x) {  
    . . .  
    case 5: // .L7  
    case 6: // .L7  
        w -= z;  
        break;  
    default: // .L8  
        w = 2;  
}
```

```
.L7:                      # Case 5,6  
    movq $1, %rax      # w = 1  
    subq %rdx, %rax   # w -= z  
    ret  
.L8:                      # Default:  
    movq $2, %rax      # 2  
    ret
```

Register	Use(s)
%rdi	Argument <b>x</b>
%rsi	Argument <b>y</b>
%rdx	Argument <b>z</b>
%rax	Return value

# Summarizing

## ■ C Control

- if-then-else
- do-while
- while, for
- switch

## ■ Assembler Control

- Conditional jump
- Conditional move
- Indirect jump (via jump tables)
- Compiler generates code sequence to implement more complex control

## ■ Standard Techniques

- Loops converted to do-while or jump-to-middle form
- Large switch statements use jump tables
- Sparse switch statements may use decision trees (if-elseif-elseif-else)

# Summary

## ■ Control

- Control: Condition codes
- Conditional branches & conditional moves
- Loops
- Switch statements

# **Machine-Level Programming III: Procedures**

# Mechanisms in Procedures

## ■ Passing control

- To beginning of procedure code
- Back to return point

## ■ Passing data

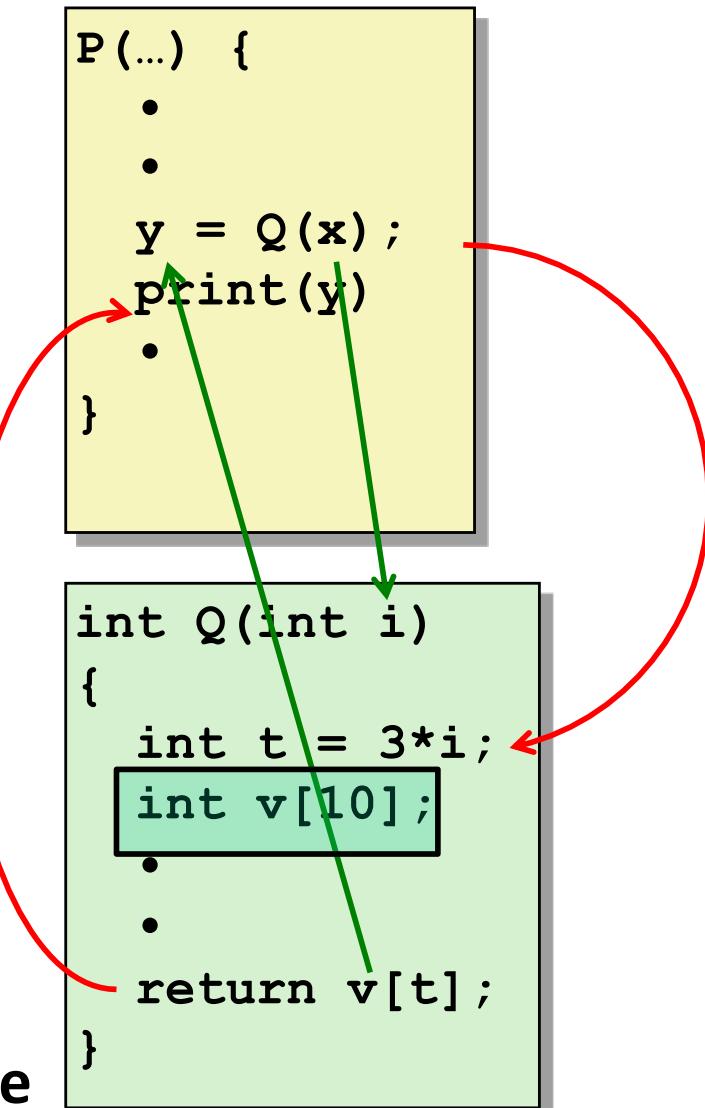
- Procedure arguments
- Return value

## ■ Memory management

- Allocate during procedure execution
- Deallocate upon return

## ■ Mechanisms all implemented with machine instructions

## ■ x86-64 implementation of a procedure uses only those mechanisms required



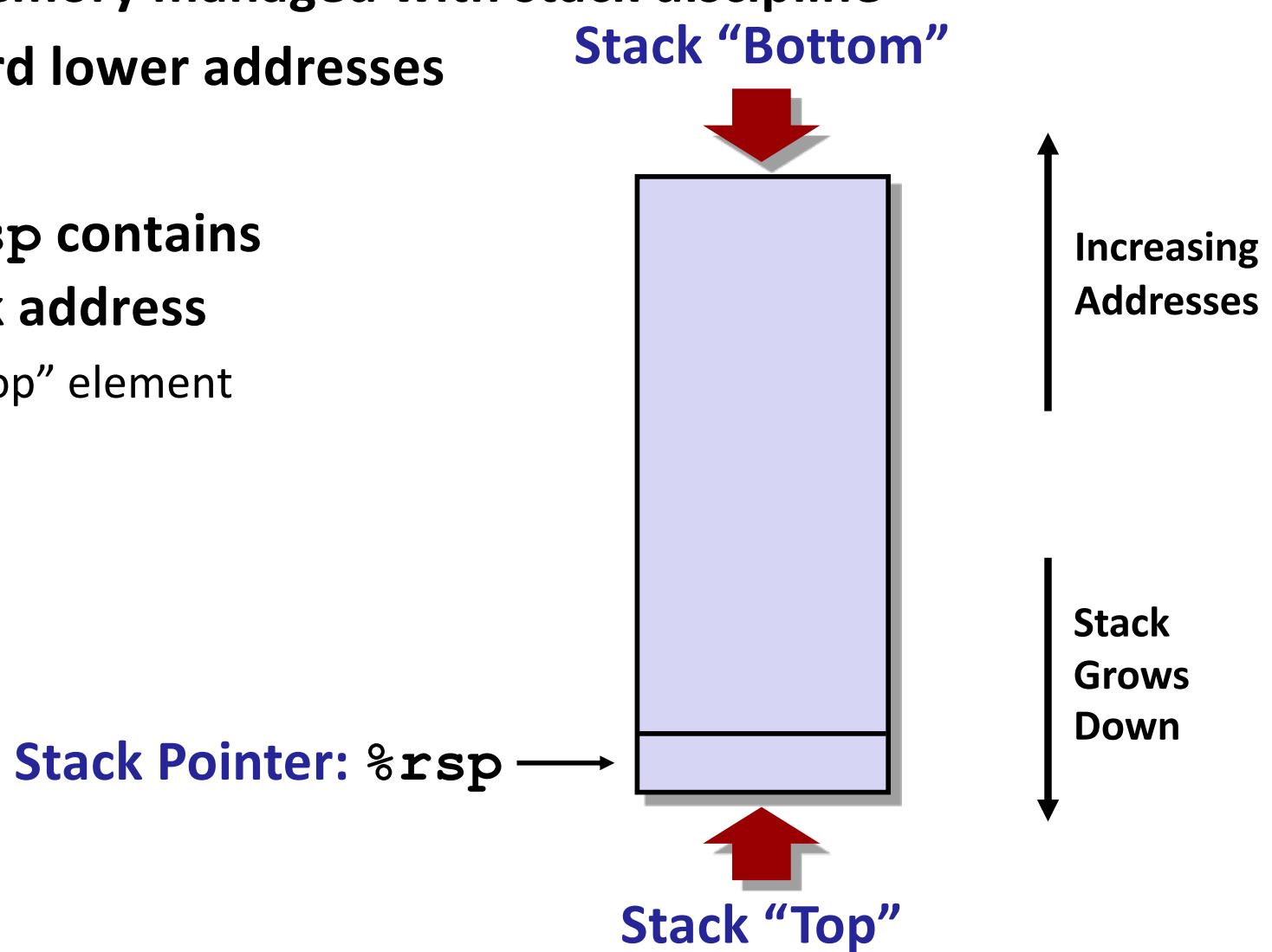
# Today

## ■ Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

# x86-64 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%rsp` contains lowest stack address
  - address of “top” element



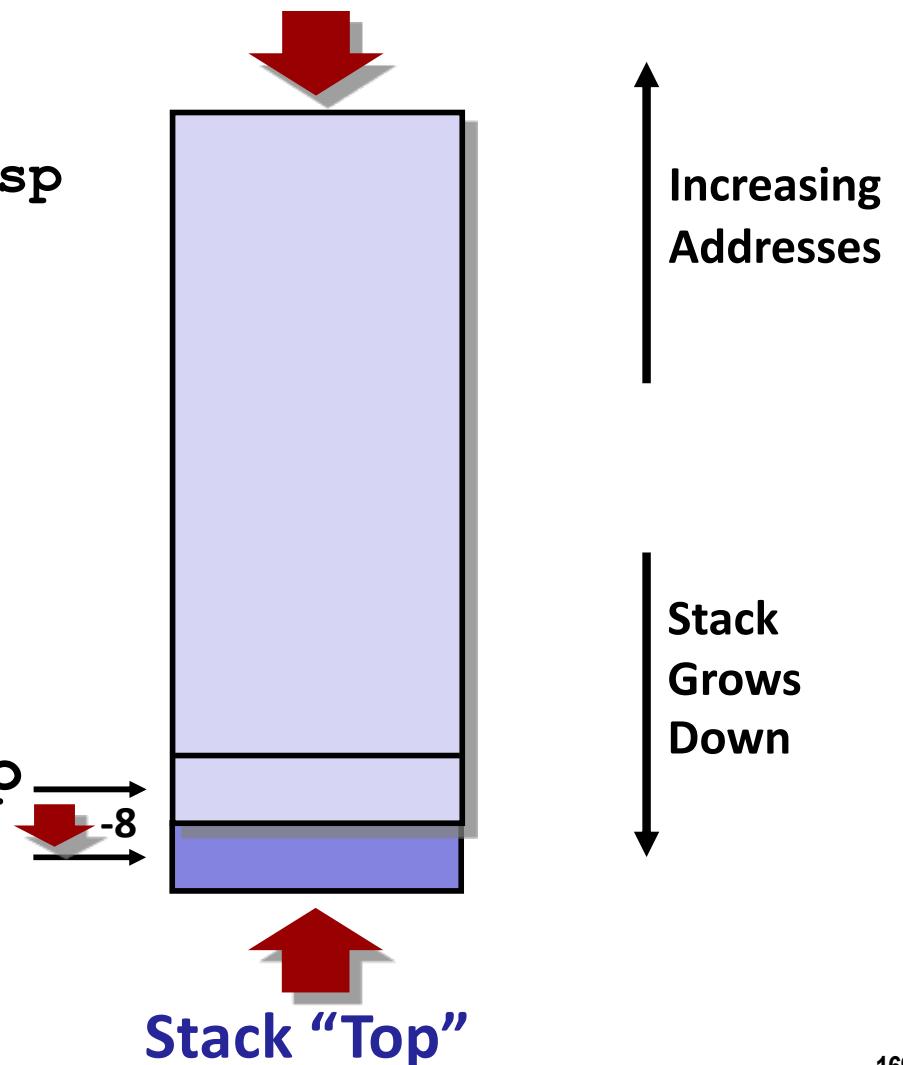
# x86-64 Stack: Push

## ■ `pushq Src`

- Fetch operand at Src
- Decrement `%rsp` by 8
- Write operand at address given by `%rsp`

Stack Pointer: `%rsp`

Stack “Bottom”



# x86-64 Stack: Pop

## ■ `popq Dest`

- Read value at address given by `%rsp`
- Increment `%rsp` by 8
- Store value at Dest (must be register)

Stack Pointer: `%rsp`

Stack “Bottom”



# Today

## ■ Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

# Code Examples

```
void multstore  
    (long x, long y, long *dest)  
{  
    long t = mult2(x, y);  
    *dest = t;  
}
```

0000000000400540 <multstore>:

```
400540: push    %rbx          # Save %rbx  
400541: mov     %rdx,%rbx    # Save dest  
400544: callq   400550 <mult2>  # mult2(x,y)  
400549: mov     %rax,(%rbx)   # Save at dest  
40054c: pop     %rbx          # Restore %rbx  
40054d: retq               # Return
```

```
long mult2  
    (long a, long b)  
{  
    long s = a * b;  
    return s;  
}
```

0000000000400550 <mult2>:

```
400550: mov     %rdi,%rax      # a  
400553: imul   %rsi,%rax      # a * b  
400557: retq               # Return
```

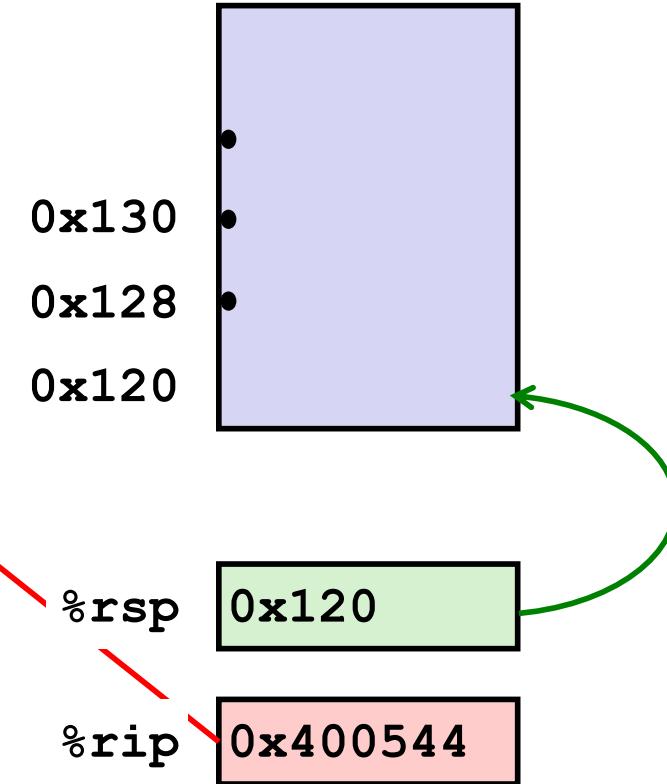
# Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call:** `call label`
  - Push return address on stack
  - Jump to label
- **Return address:**
  - Address of the next instruction right after call
  - Example from disassembly
- **Procedure return:** `ret`
  - Pop address from stack
  - Jump to address

# Control Flow Example #1

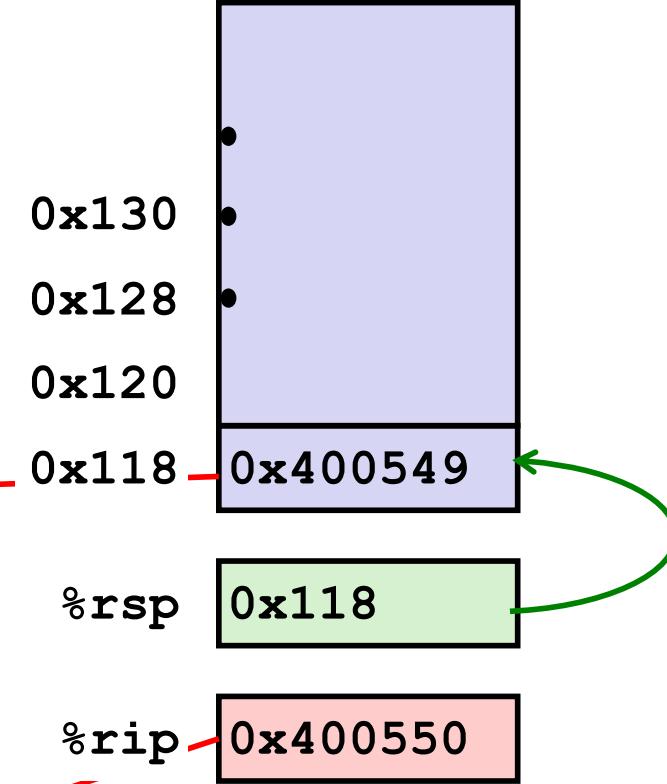
```
0000000000400540 <multstore>:  
.  
. .  
400544: callq 400550 <mult2>  
400549: mov    %rax, (%rbx)  
. .  
.
```

```
0000000000400550 <mult2>:  
400550: mov    %rdi,%rax  
. .  
400557: retq
```



# Control Flow Example #2

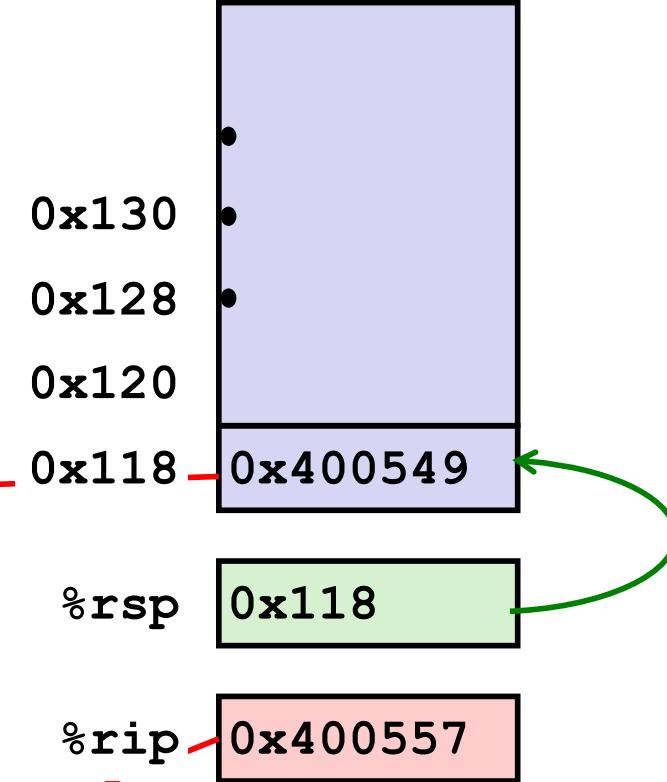
```
0000000000400540 <multstore>:  
•  
•  
400544: callq 400550 <mult2>  
400549: mov    %rax, (%rbx) ←
```



```
0000000000400550 <mult2>:  
400550: mov    %rdi,%rax ←  
•  
•  
400557: retq
```

# Control Flow Example #3

```
0000000000400540 <multstore>:  
  •  
  •  
 400544: callq  400550 <mult2>  
 400549: mov     %rax, (%rbx) ←
```



```
0000000000400550 <mult2>:  
 400550: mov     %rdi,%rax  
  •  
  •  
 400557: retq ←
```

# Control Flow Example #4

```
0000000000400540 <multstore>:
```

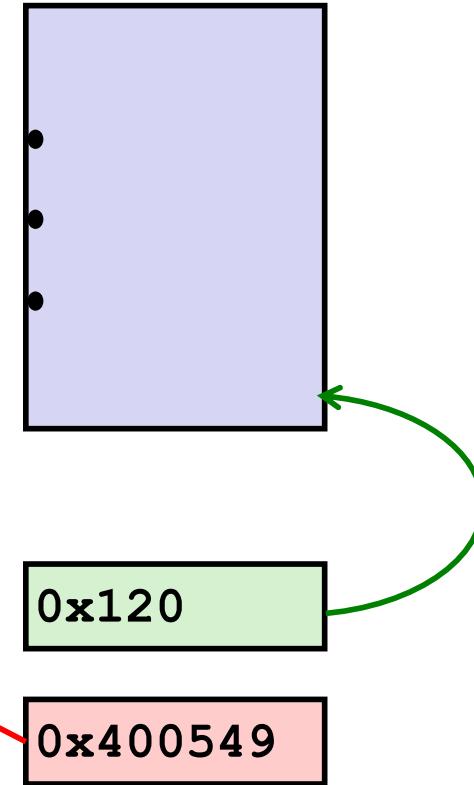
```
•  
•  
400544: callq 400550 <mult2>  
400549: mov    %rax, (%rbx) ←  
•  
•
```

```
0000000000400550 <mult2>:
```

```
400550: mov    %rdi,%rax  
•  
•  
400557: retq
```

0x130  
0x128  
0x120

%rsp 0x120  
%rip 0x400549



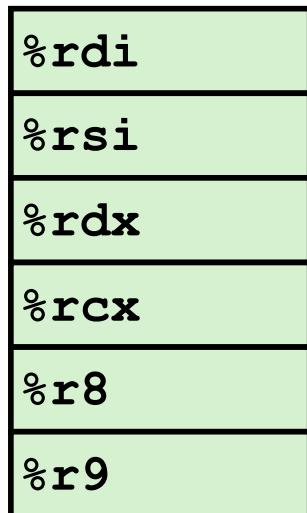
# Today

## ■ Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - **Passing data**
  - Managing local data
- Illustrations of Recursion & Pointers

# Procedure Data Flow

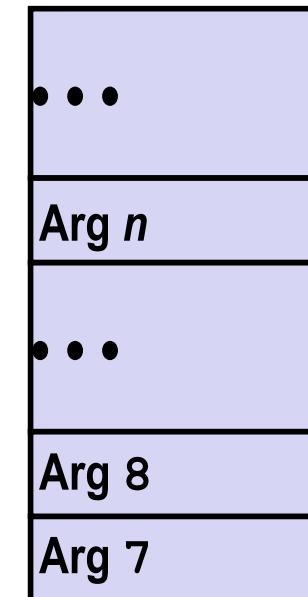
- First 6 arguments
- Registers



- Return value



- Stack



- Only allocate stack space when needed

# Data Flow Examples

```
void multstore  
    (long x, long y, long *dest)  
{  
    long t = mult2(x, y);  
    *dest = t;  
}
```

```
0000000000400540 <multstore>:  
    # x in %rdi, y in %rsi, dest in %rdx  
    ...  
    400541: mov    %rdx,%rbx          # Save dest  
    400544: callq   400550 <mult2>    # mult2(x,y)  
    # t in %rax  
    400549: mov    %rax,(%rbx)       # Save at dest  
    ...
```

```
long mult2  
    (long a, long b)  
{  
    long s = a * b;  
    return s;  
}
```

```
0000000000400550 <mult2>:  
    # a in %rdi, b in %rsi  
    400550: mov    %rdi,%rax          # a  
    400553: imul   %rsi,%rax          # a * b  
    # s in %rax  
    400557: retq   # Return
```

# Today

## ■ Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

# Stack-Based Languages

## ■ Languages that support recursion

- e.g., C, Pascal, Java
- Code must be “Reentrant”
  - Multiple simultaneous instantiations of single procedure
- Need some place to store state of each instantiation
  - Arguments
  - Local variables
  - Return pointer

## ■ Stack discipline

- State for given procedure needed for limited time
  - From when called to when return
- Callee returns before caller does

## ■ Stack allocated in **Frames**

- state for single procedure instantiation

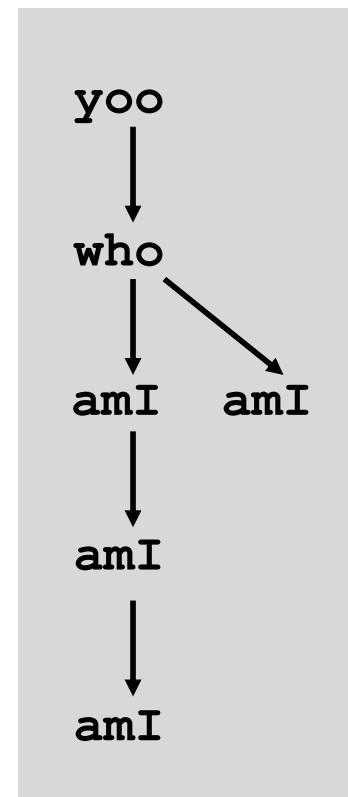
# Call Chain Example

```
yoo (...)  
{  
    •  
    •  
    who () ;  
    •  
    •  
}
```

```
who (...)  
{  
    • • •  
    amI () ;  
    • • •  
    amI () ;  
    • • •  
}
```

```
amI (...)  
{  
    •  
    •  
    amI () ;  
    •  
    •  
}
```

Example  
Call Chain



Procedure amI () is recursive

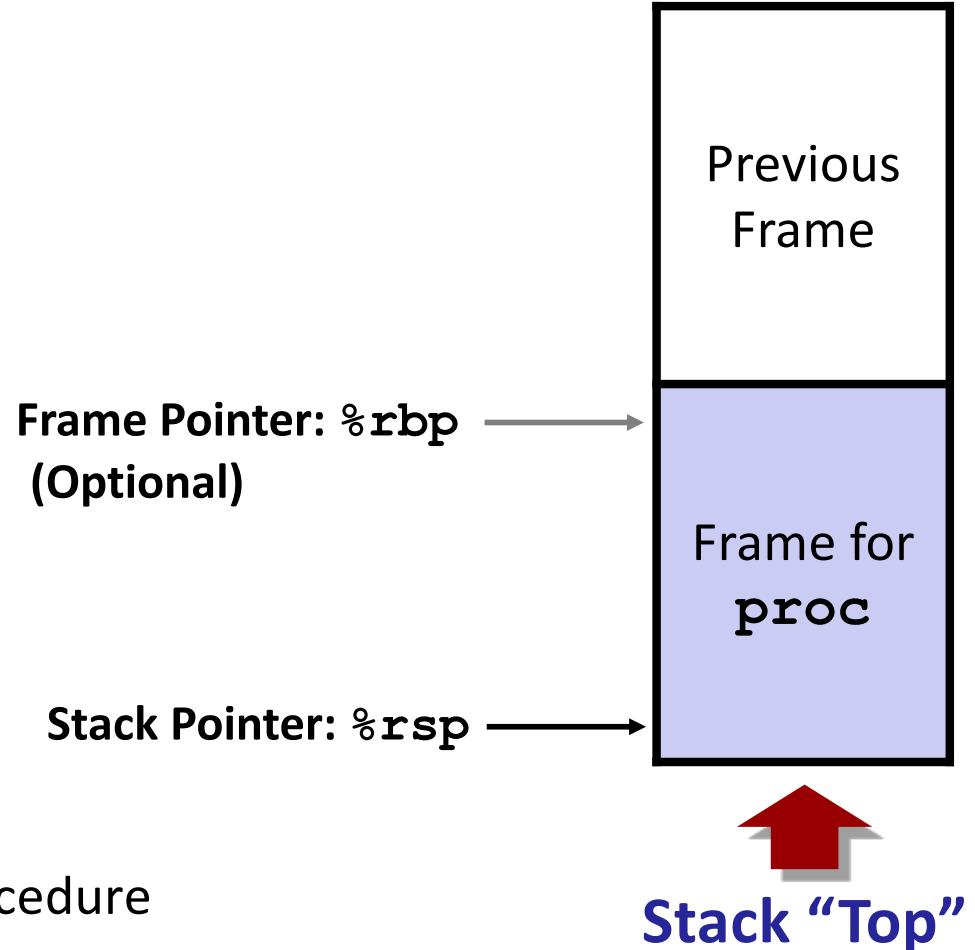
# Stack Frames

## ■ Contents

- Return information
- Local storage (if needed)
- Temporary space (if needed)

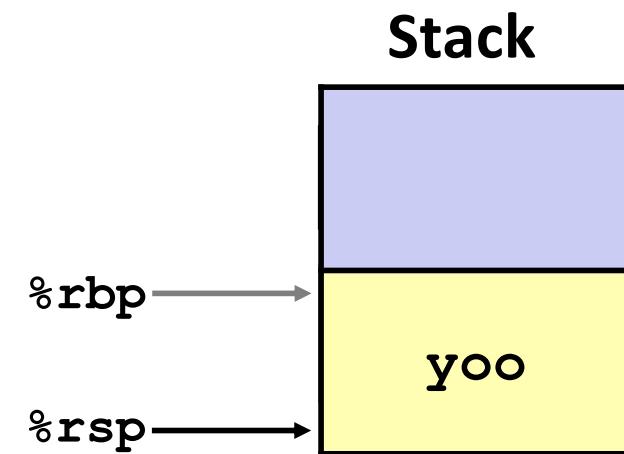
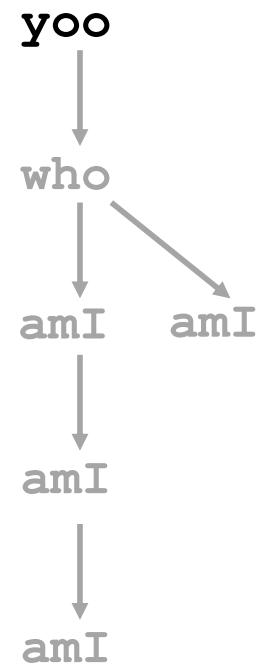
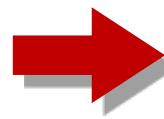
## ■ Management

- Space allocated when enter procedure
  - “Set-up” code
  - Includes push by **call** instruction
- Deallocated when return
  - “Finish” code
  - Includes pop by **ret** instruction



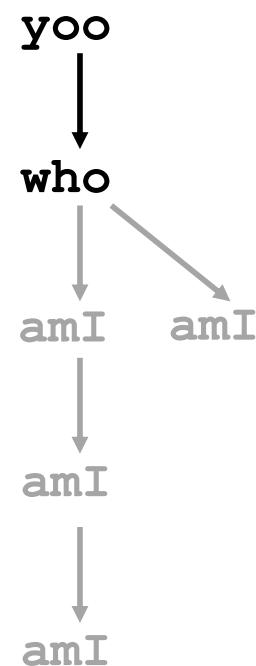
# Example

```
yoo (...)  
{  
    •  
    •  
    who () ;  
    •  
    •  
}
```

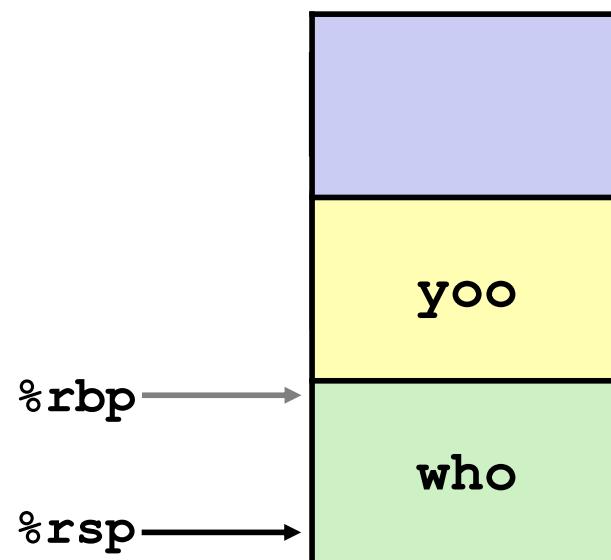


# Example

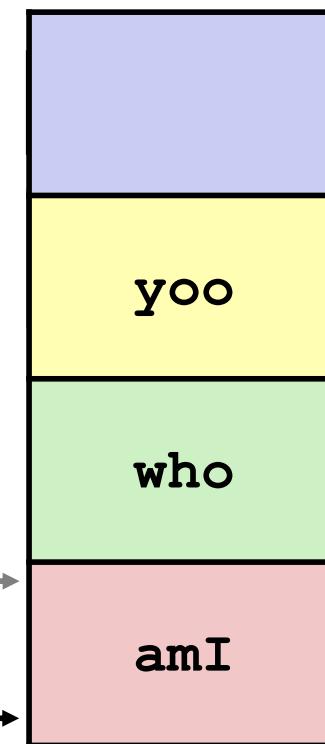
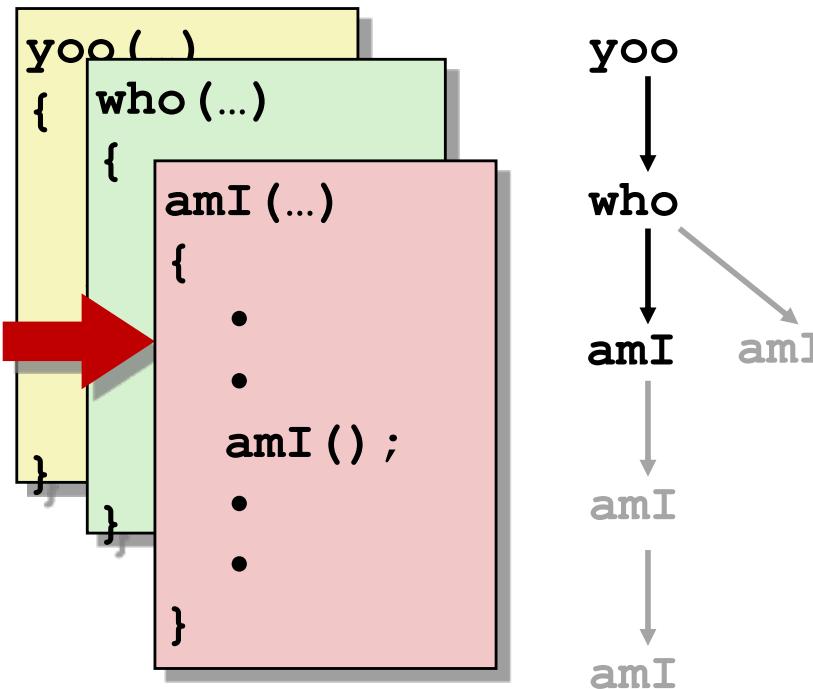
```
yoo( )  
{ who (...)  
{  
    • • •  
    amI () ;  
    • • •  
    amI () ;  
    • • •  
}  
}
```



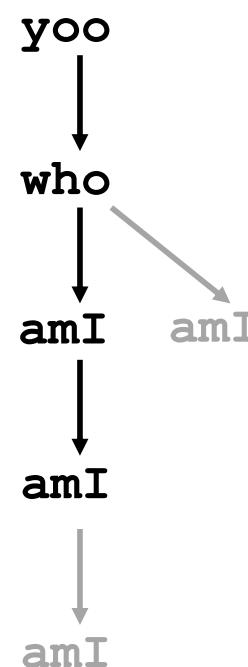
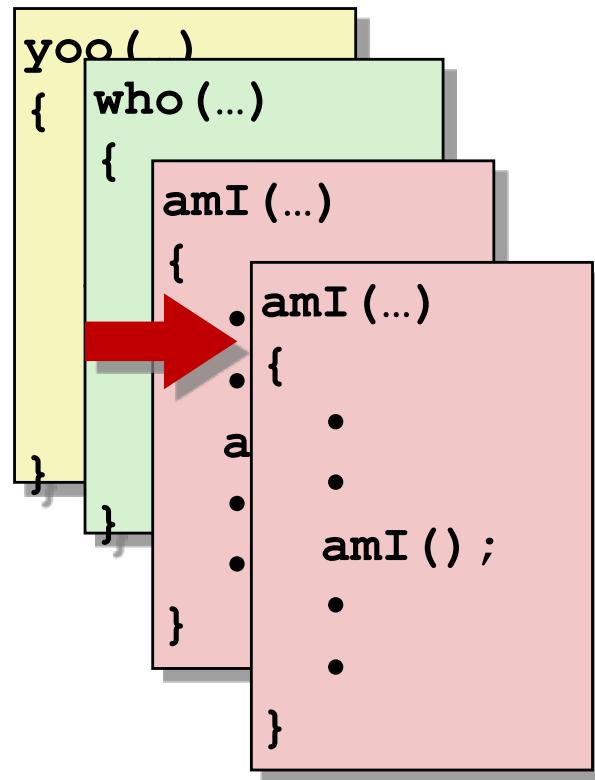
# Stack



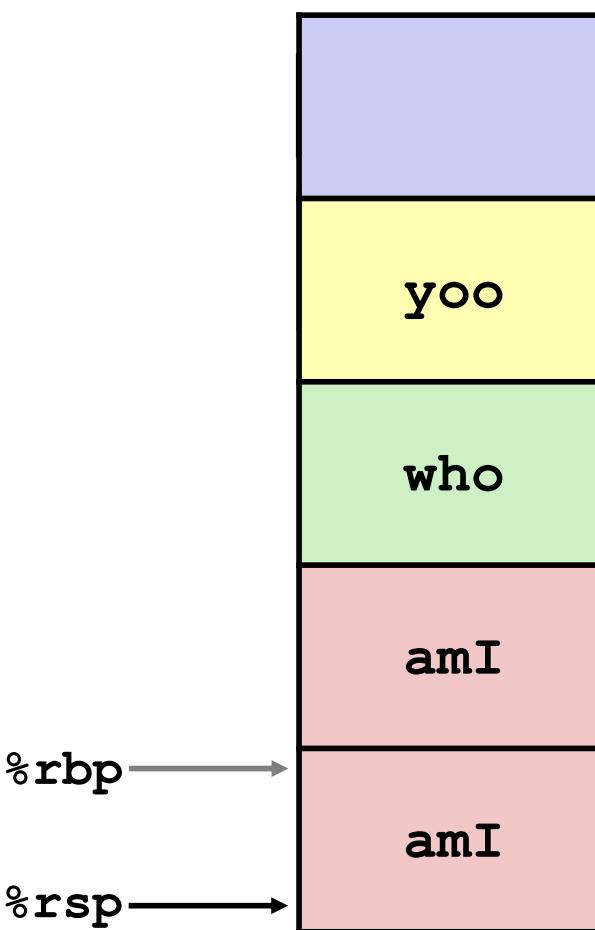
# Example



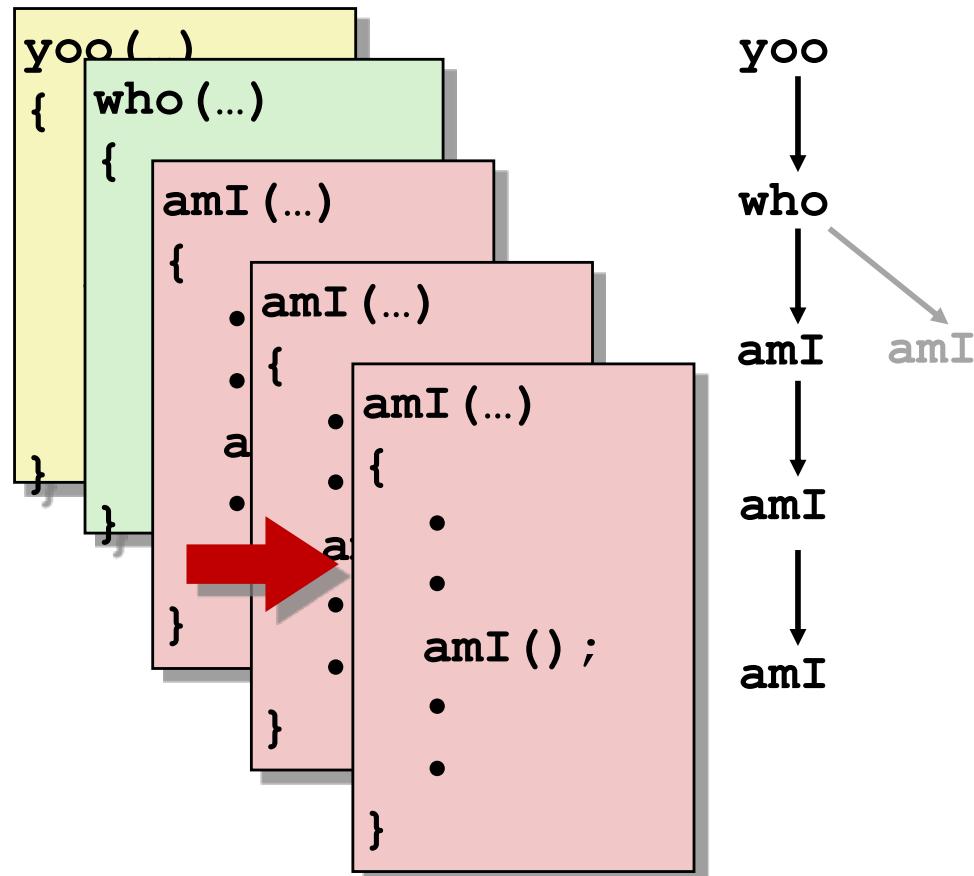
# Example



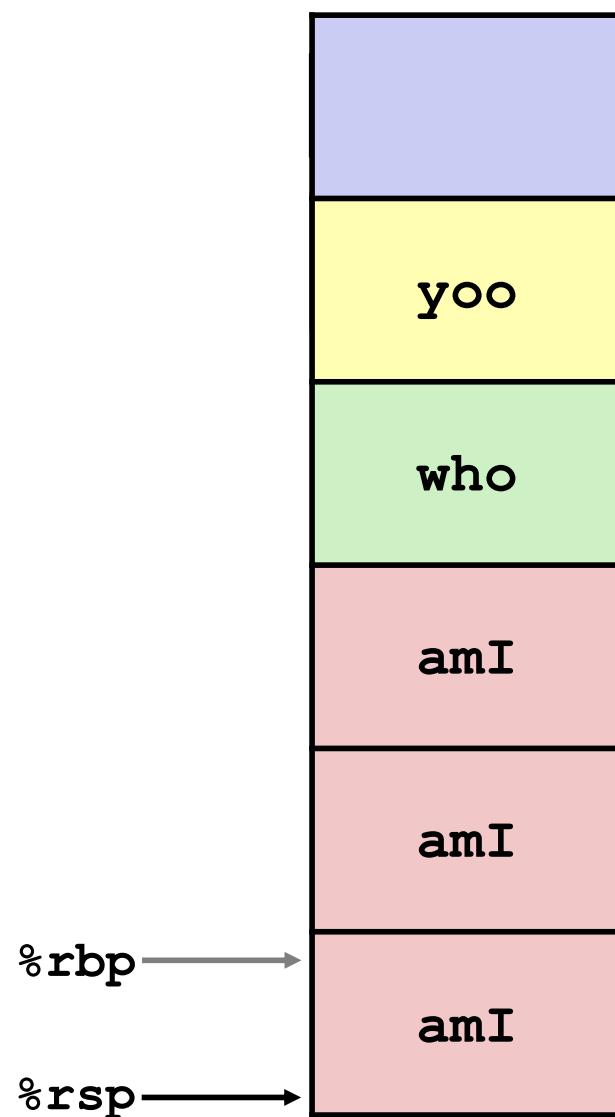
Stack



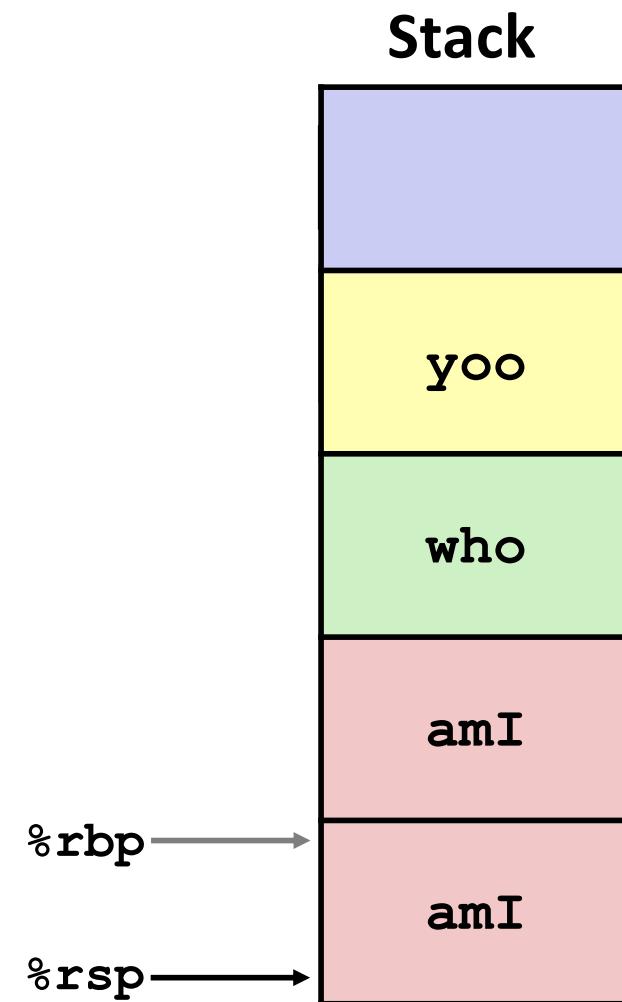
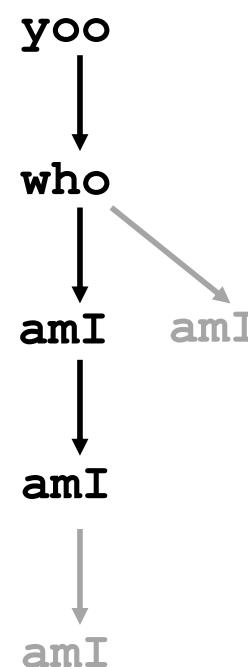
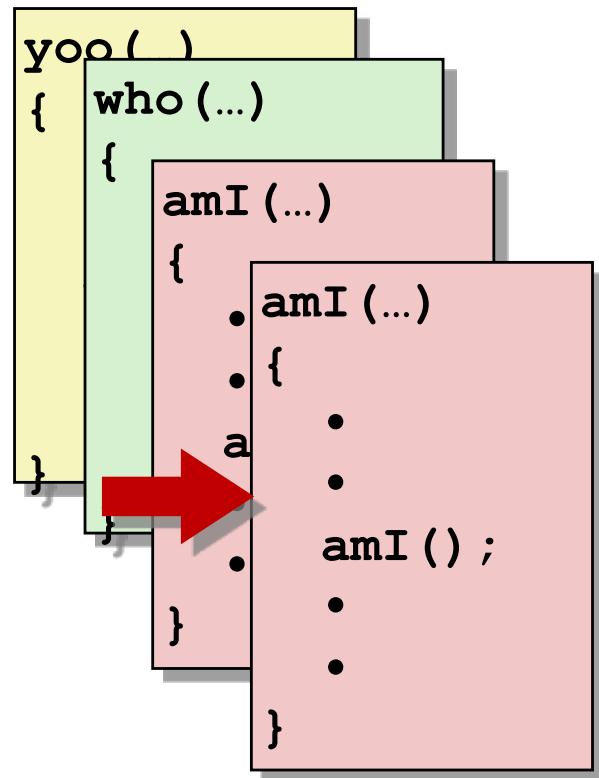
# Example



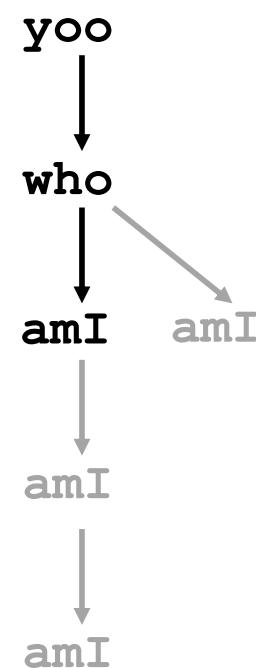
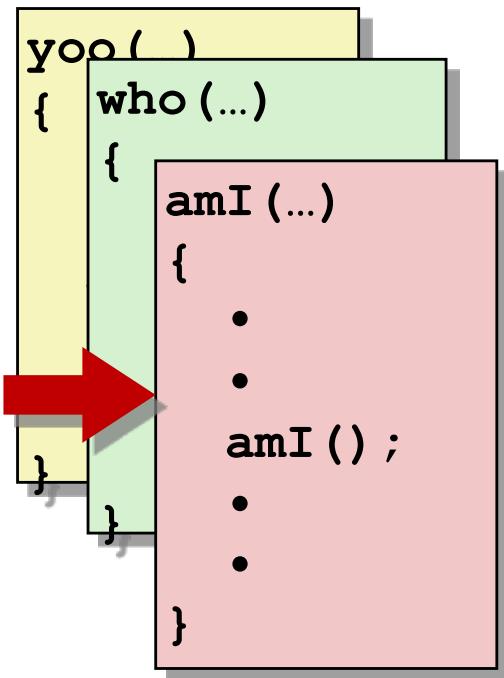
Stack



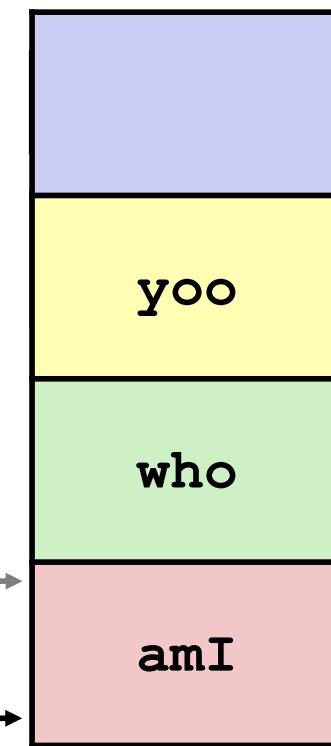
# Example



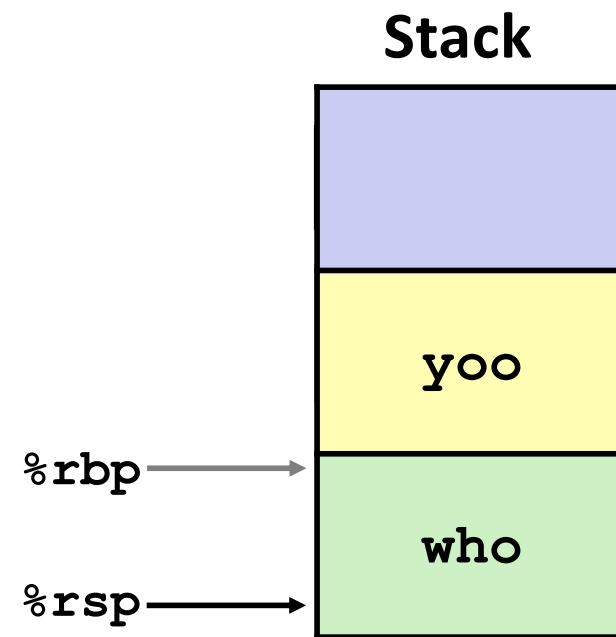
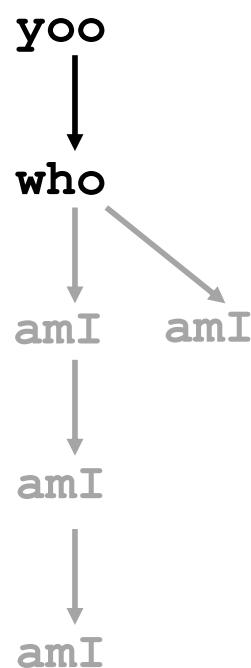
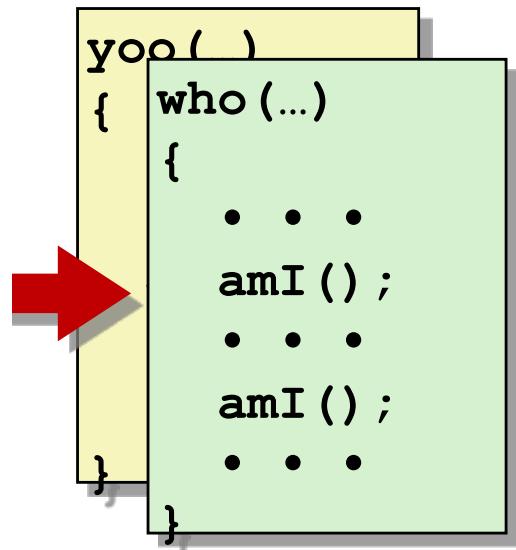
# Example



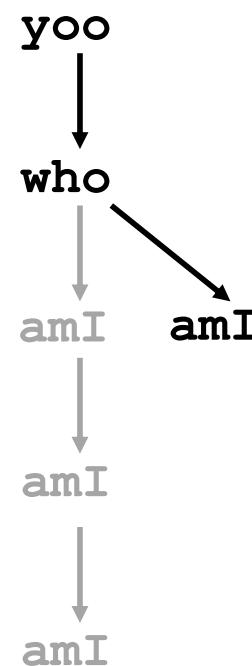
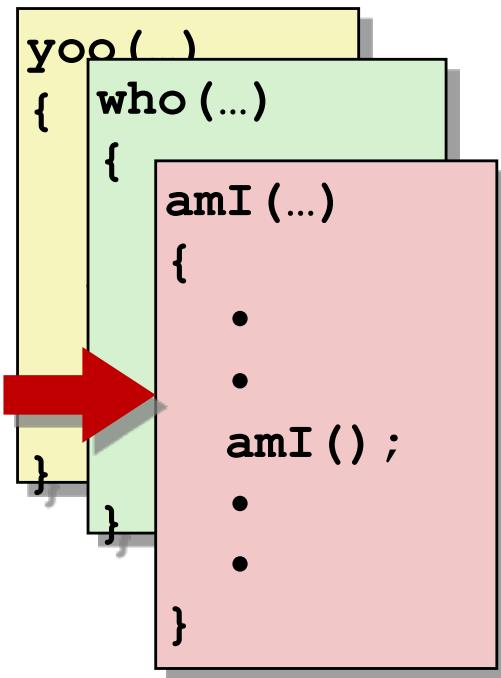
# Stack



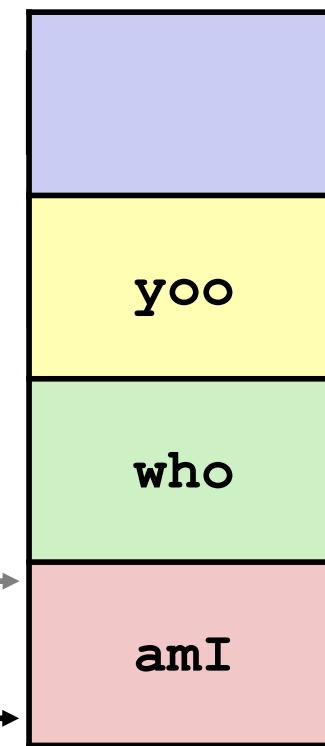
# Example



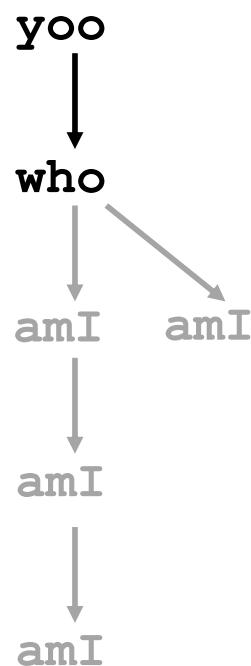
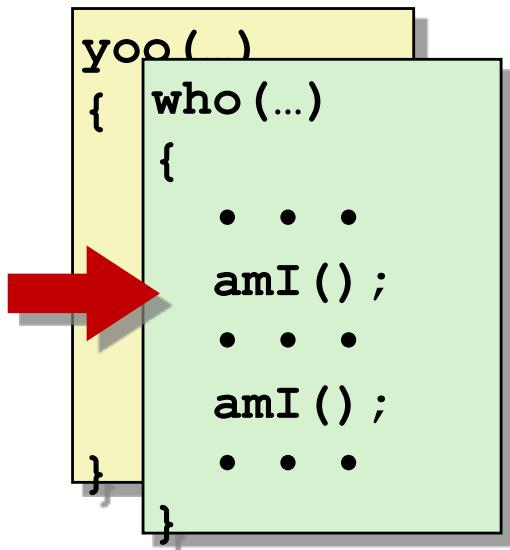
# Example



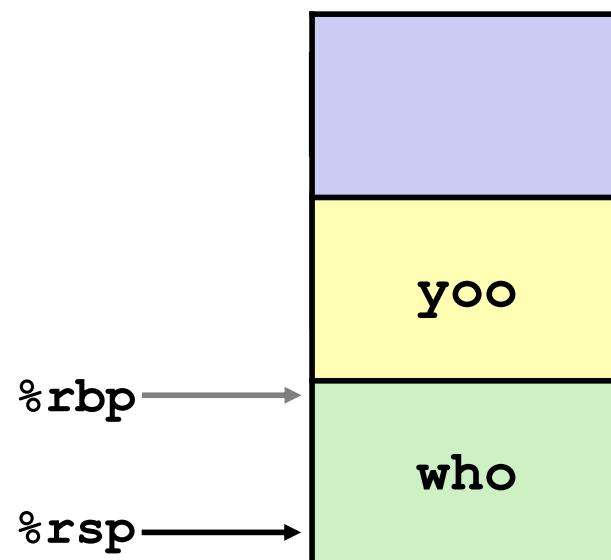
Stack



# Example

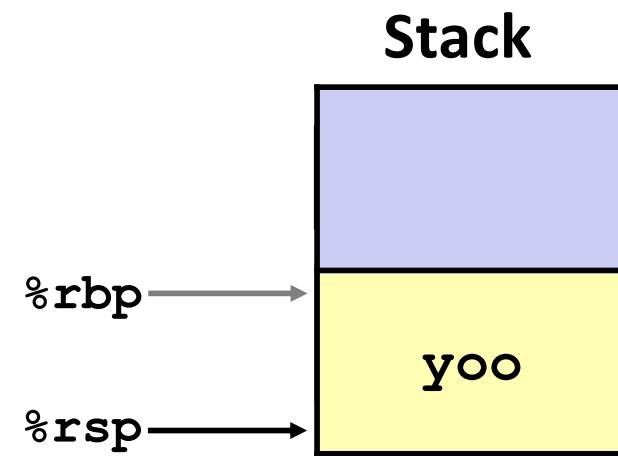
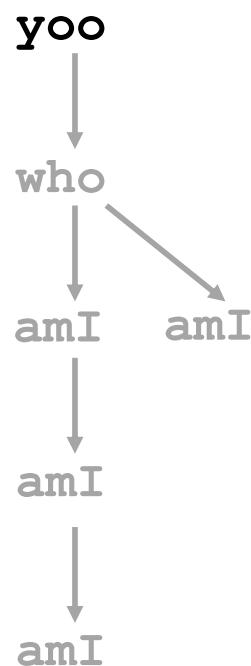


Stack



# Example

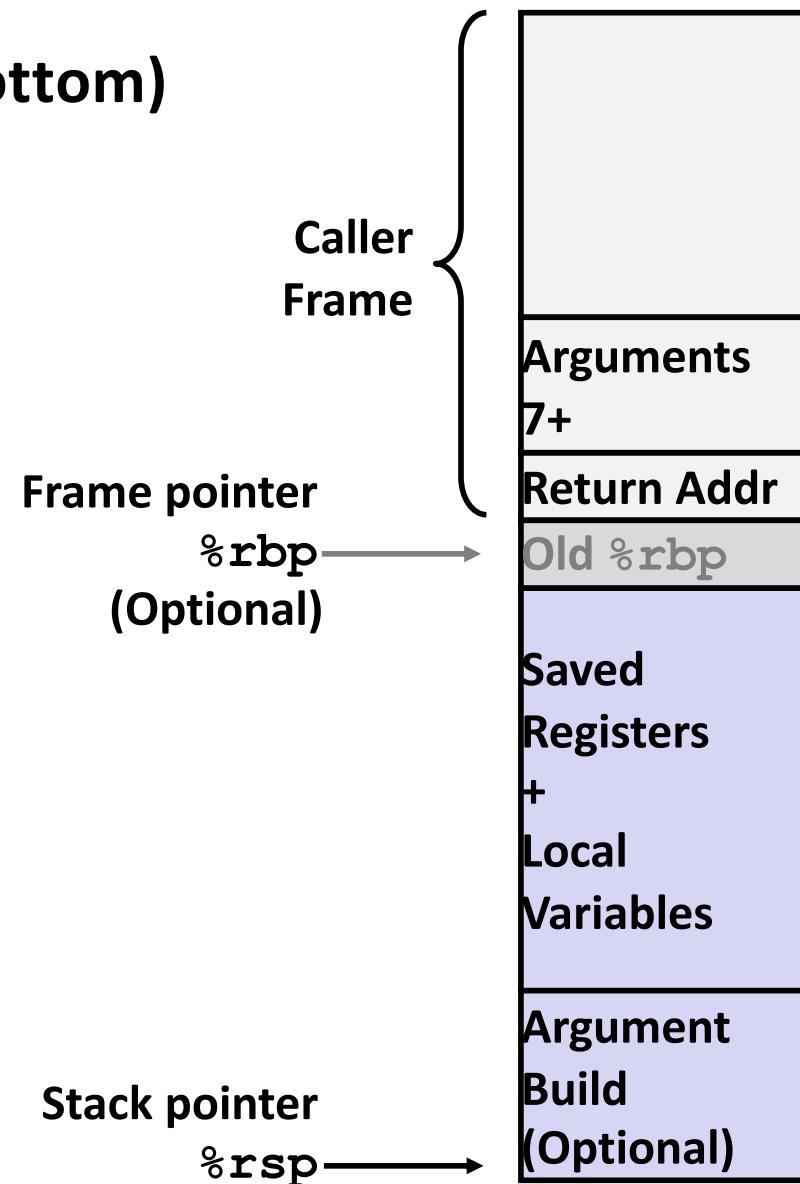
```
yoo (...)  
{  
    •  
    •  
    who () ;  
    •  
    •  
}
```



# x86-64/Linux Stack Frame

## ■ Current Stack Frame (“Top” to Bottom)

- “Argument build:”  
Parameters for function about to call
- Local variables  
If can’t keep in registers
- Saved register context
- Old frame pointer (optional)



## ■ Caller Stack Frame

- Return address
  - Pushed by **call** instruction
- Arguments for this call

# Example: incr

```
long incr(long *p, long val) {  
    long x = *p;  
    long y = x + val;  
    *p = y;  
    return x;  
}
```

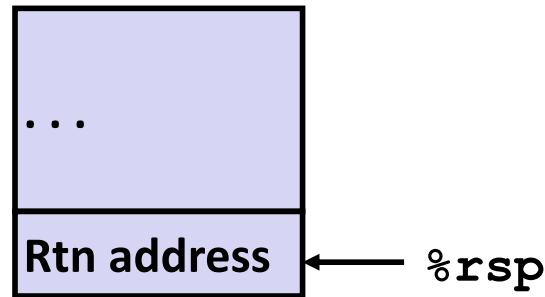
```
incr:  
    movq    (%rdi), %rax  
    addq    %rax, %rsi  
    movq    %rsi, (%rdi)  
    ret
```

Register	Use(s)
%rdi	Argument <b>p</b>
%rsi	Argument <b>val</b> , <b>y</b>
%rax	<b>x</b> , Return value

# Example: Calling incr #1

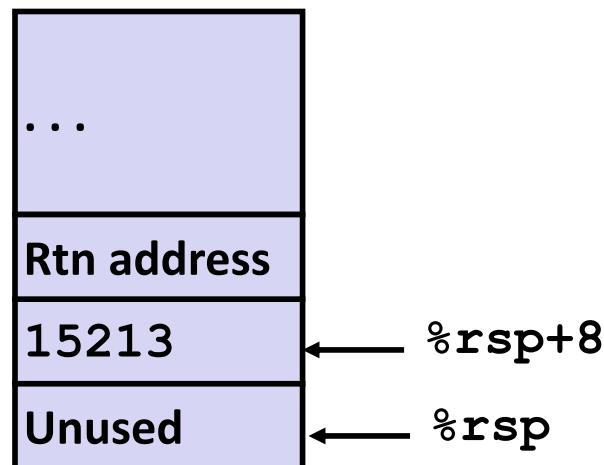
```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

Initial Stack Structure



```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movq    $3000, %rsi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Resulting Stack Structure

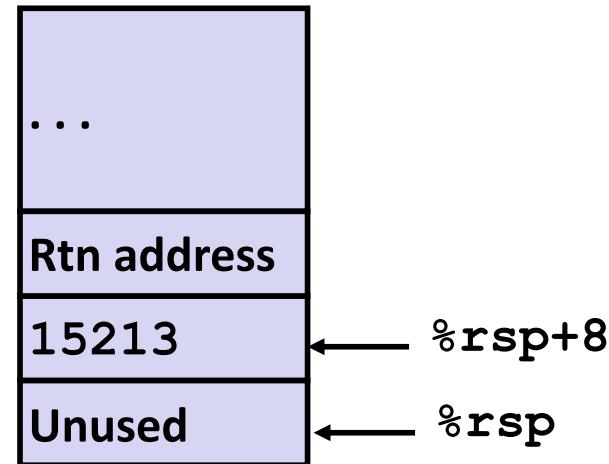


# Example: Calling incr #2

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movq    $3000, %rsi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Stack Structure



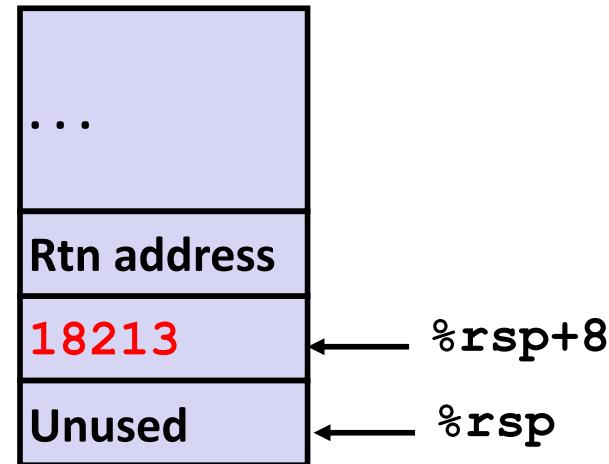
Register	Use(s)
%rdi	&v1
%rsi	3000

# Example: Calling incr #3

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movq    $3000, %rsi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Stack Structure

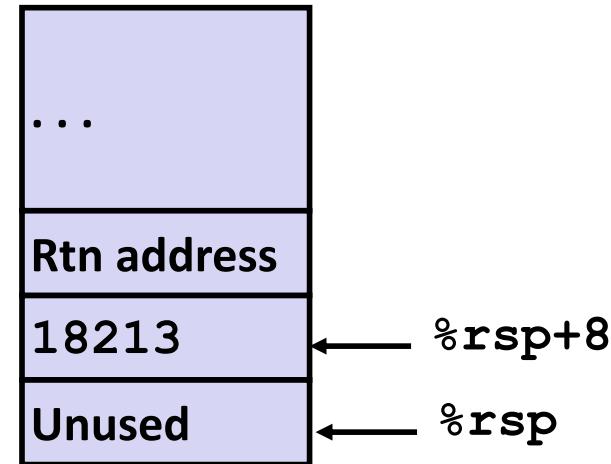


Register	Use(s)
%rdi	&v1
%rsi	3000

# Example: Calling incr #4

Stack Structure

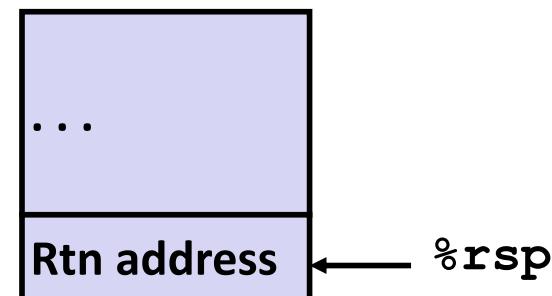
```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```



```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movq    $3000, %rsi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Register	Use(s)
%rax	Return value

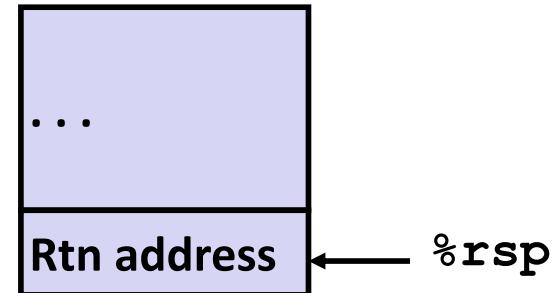
Updated Stack Structure



# Example: Calling incr #5

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

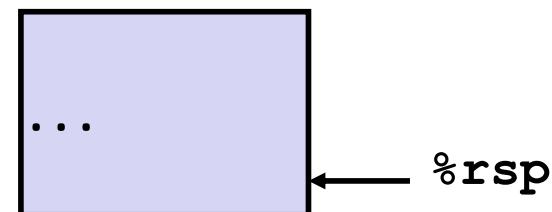
Updated Stack Structure



```
call_incr:  
subq    $16, %rsp  
movq    $15213, 8(%rsp)  
movq    $3000, %rsi  
leaq    8(%rsp), %rdi  
call    incr  
addq    8(%rsp), %rax  
addq    $16, %rsp  
ret
```

Register	Use(s)
%rax	Return value

Final Stack Structure



# Register Saving Conventions

- When procedure **yoo** calls **who**:
  - **yoo** is the **caller**
  - **who** is the **callee**
- Can register be used for temporary storage?

```
yoo:
```

```
• • •  
    movq $15213, %rdx  
    call who  
    addq %rdx, %rax  
• • •  
    ret
```

```
who:
```

```
• • •  
    subq $18213, %rdx  
• • •  
    ret
```

- Contents of register **%rdx** overwritten by **who**
- This could be trouble → something should be done!
  - Need some coordination

# Register Saving Conventions

- When procedure **yoo** calls **who**:
  - **yoo** is the **caller**
  - **who** is the **callee**
- Can register be used for temporary storage?
- Conventions
  - “Caller Saved”
    - Caller saves temporary values in its frame before the call
  - “Callee Saved”
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller

# x86-64 Linux Register Usage #1

## ■ **%rax**

- Return value
- Also caller-saved
- Can be modified by procedure

## ■ **%rdi, ..., %r9**

- Arguments
- Also caller-saved
- Can be modified by procedure

## ■ **%r10, %r11**

- Caller-saved
- Can be modified by procedure

Return value

**%rax**

**%rdi**

**%rsi**

**%rdx**

**%rcx**

**%r8**

**%r9**

**%r10**

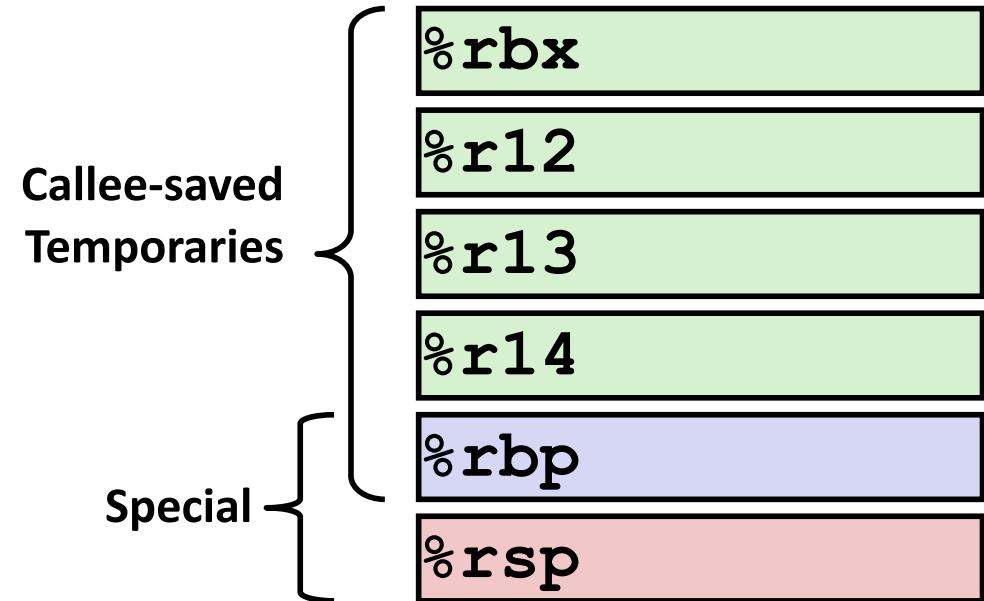
**%r11**

Arguments

Caller-saved  
temporaries

# x86-64 Linux Register Usage #2

- **%rbx, %r12, %r13, %r14**
  - Callee-saved
  - Callee must save & restore
- **%rbp**
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
  - Can mix & match
- **%rsp**
  - Special form of callee save
  - Restored to original value upon exit from procedure



# Revisiting swap

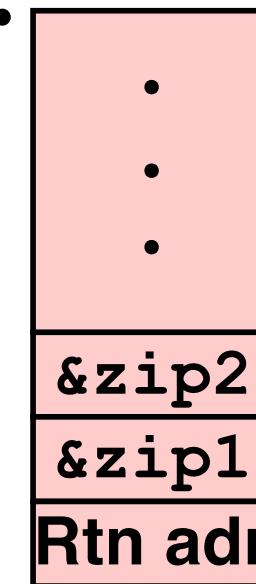
```
int zip1 = 15213;
int zip2 = 91125;

void call_swap()
{
    swap(&zip1, &zip2);
}
```

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

## Calling swap from call\_swap

```
call_swap:
    • • •
    pushq $zip2 # Global
Var
    pushq $zip1 # Global
Var
    call swap
    • • •
```



**Resulting Stack**

# Revisiting swap

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

swap:

```
pushq %rbp  
movq %rsp,%rbp  
pushq %rbx
```

} Set Up

```
movq 24(%rbp),%rcx  
movq 16(%rbp),%rdx  
movq (%rcx),%rax  
movq (%rdx),%rbx  
movq %rax,(%rdx)  
movq %rbx,(%rcx)
```

} Body

```
movq -8(%rbp),%rbx  
movq %rbp,%rsp  
popq %rbp  
ret
```

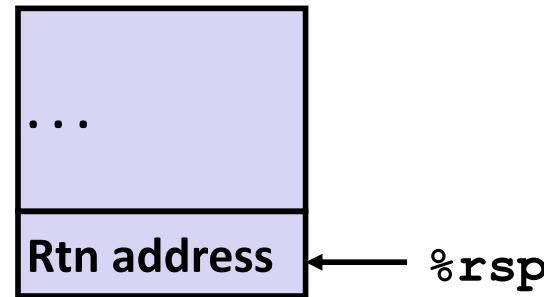
} Finish

# Callee-Saved Example #1

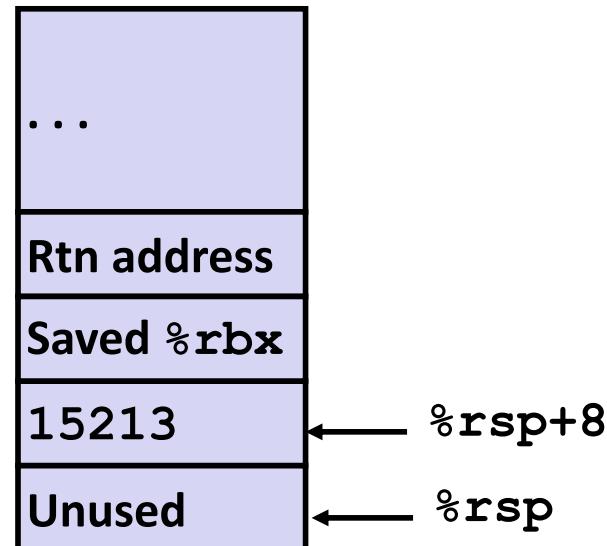
```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

```
call_incr2:  
    pushq  %rbx  
    subq    $16, %rsp  
    movq    %rdi, %rbx  
    movq    $15213, 8(%rsp)  
    movq    $3000, %rsi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    %rbx, %rax  
    addq    $16, %rsp  
    popq    %rbx  
    ret
```

Initial Stack Structure



Resulting Stack Structure

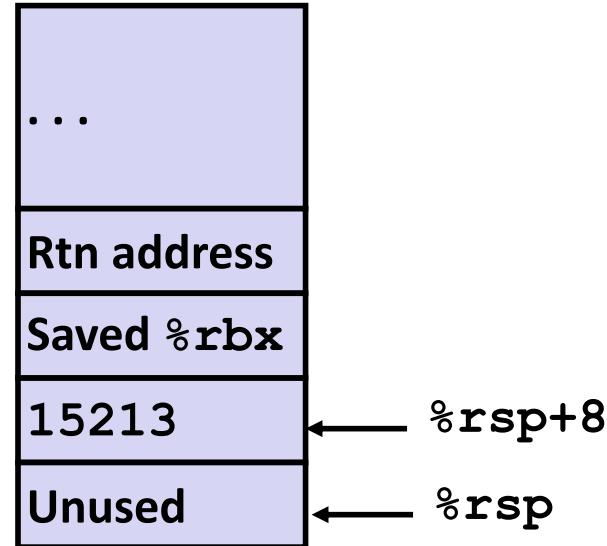


# Callee-Saved Example #2

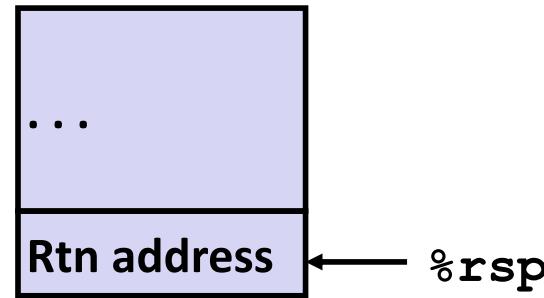
Resulting Stack Structure

```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

```
call_incr2:  
    pushq  %rbx  
    subq    $16, %rsp  
    movq    %rdi, %rbx  
    movq    $15213, 8(%rsp)  
    movq    $3000, %rsi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    %rbx, %rax  
    addq    $16, %rsp  
    popq    %rbx  
    ret
```



Pre-return Stack Structure



# Today

## ■ Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

# Recursive Function

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

# Recursive Function

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

pcount\_r:

```
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq   $1, %rbx
    shrq   %rdi # (by 1)
    call    pcount_r
    addq    %rbx, %rax
    popq   %rbx
.L6:
    ret
```

Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

# Recursive Function Terminal Case

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

pcount\_r:

```
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq    %rdi # (by 1)
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
```

.L6:

ret

Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

# Recursive Function Register Save

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

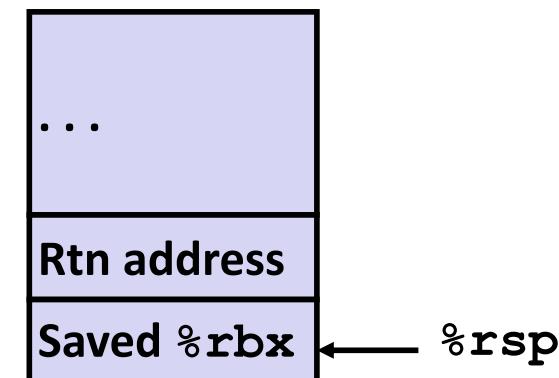
pcount\_r:

```
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq   $1, %rbx
    shrq    %rdi # (by 1)
    call    pcount_r
    addq    %rbx, %rax
    popq   %rbx
```

.L6:

ret

Register	Use(s)	Type
%rdi	x	Argument



# Recursive Function Call Setup

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

pcount\_r:

```
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq    %rdi # (by 1)
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
```

.L6:

ret

Register	Use(s)	Type
%rdi	x >> 1	Rec. argument
%rbx	x & 1	Callee-saved

# Recursive Function Call

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

pcount\_r:

```
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq    %rdi # (by 1)
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
```

.L6:

ret

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Recursive call return value	

# Recursive Function Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

pcount\_r:

```
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq    %rdi # (by 1)
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
```

.L6:

ret

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

# Recursive Function Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

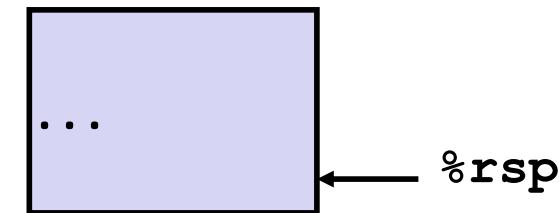
pcount\_r:

```
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq    %rdi # (by 1)
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
```

.L6:

ret

Register	Use(s)	Type
%rax	Return value	Return value



# Observations About Recursion

## ■ Handled Without Special Consideration

- Stack frames mean that each function call has private storage
  - Saved registers & local variables
  - Saved return pointer
- Register saving conventions prevent one function call from corrupting another's data
  - Unless the C code explicitly does so (e.g., buffer overflow in Lecture 9)
- Stack discipline follows call / return pattern
  - If P calls Q, then Q returns before P
  - Last-In, First-Out

## ■ Also works for mutual recursion

- P calls Q; Q calls P

# x86-64 Procedure Summary

## ■ Important Points

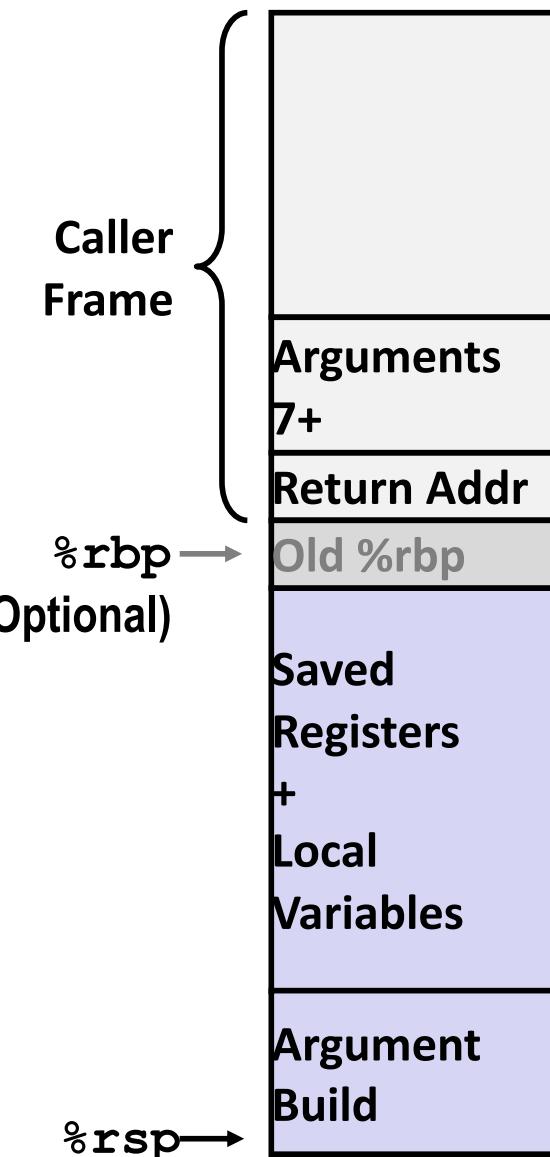
- Stack is the right data structure for procedure call / return
  - If P calls Q, then Q returns before P

## ■ Recursion (& mutual recursion) handled by normal calling conventions

- Can safely store values in local stack frame and in callee-saved registers
- Put function arguments at top of stack
- Result return in `%rax`

## ■ Pointers are addresses of values

- On stack or global



# **Machine-Level Programming IV: Data**

# Today

## ■ Arrays

- One-dimensional
- Multi-dimensional (nested)
- Multi-level

## ■ Structures

- Allocation
- Access
- Alignment

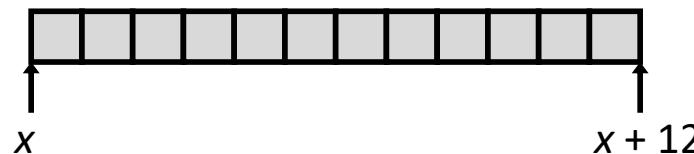
# Array Allocation

## ■ Basic Principle

$T \ A[L];$

- Array of data type  $T$  and length  $L$
- Contiguously allocated region of  $L * \text{sizeof}(T)$  bytes in memory

`char string[12];`



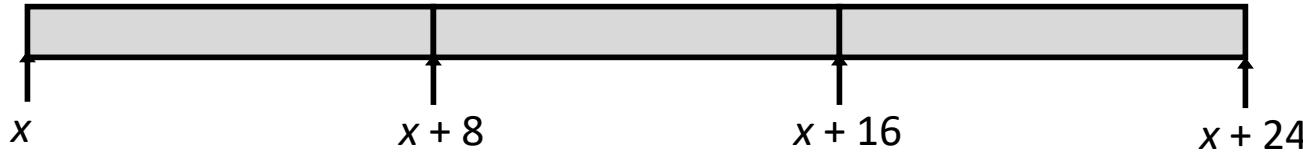
`int val[5];`



`double a[3];`



`char *p[3];`

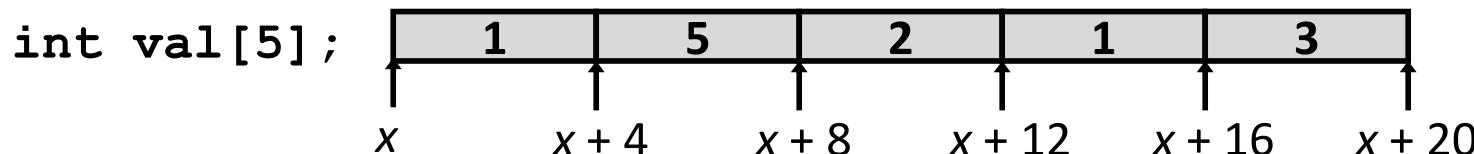


# Array Access

## ■ Basic Principle

$T \mathbf{A}[L];$

- Array of data type  $T$  and length  $L$
- Identifier **A** can be used as a pointer to array element 0: Type  $T^*$

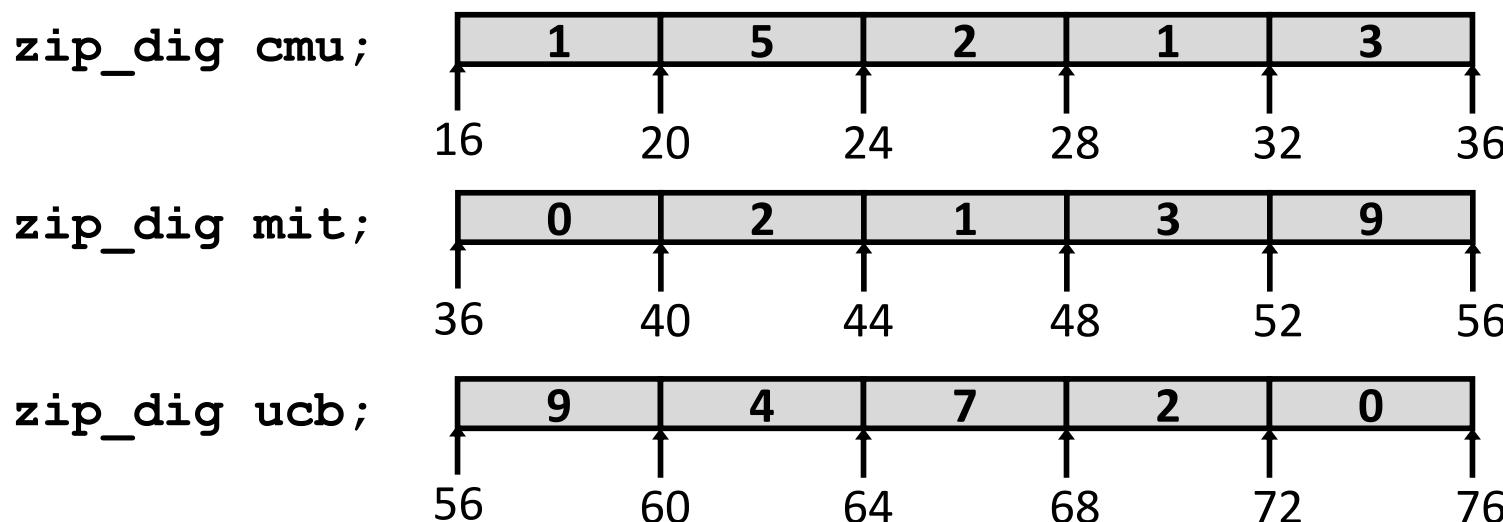


■ Reference	Type	Value
<code>val[4]</code>	<code>int</code>	3
<code>val</code>	<code>int *</code>	$x$
<code>val+1</code>	<code>int *</code>	$x + 4$
<code>&amp;val[2]</code>	<code>int *</code>	$x + 8$
<code>val[5]</code>	<code>int</code>	??
<code>* (val+1)</code>	<code>int</code>	5
<code>val + i</code>	<code>int *</code>	$x + 4i$

# Array Example

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

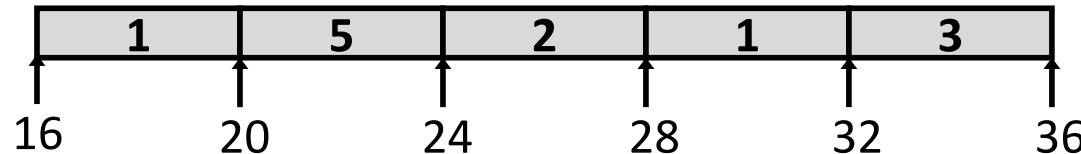
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



- Declaration “`zip_dig cmu`” equivalent to “`int cmu[5]`”
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

# Array Accessing Example

```
zip_dig cmu;
```



```
int get_digit
    (zip_dig z, int digit)
{
    return z[digit];
}
```

IA32

```
# %rdi = z
# %rsi = digit
movl (%rdi,%rsi,4), %eax # z[digit]
```

- Register **%rdi** contains starting address of array
- Register **%rsi** contains array index
- Desired digit at  $\%rdi + 4 * \%rsi$
- Use memory reference  $(\%rdi, \%rsi, 4)$

# Array Loop Example

```
void zincr(zip_dig z) {
    size_t i;
    for (i = 0; i < ZLEN; i++)
        z[i]++;
}
```

```
# %rdi = z
movq    $0, %rax          #   i = 0
jmp     .L3                #   goto middle
.L4:               # loop:
    addl    $1, (%rdi,%rax,4) #   z[i]++
    addq    $1, %rax          #   i++
.L3:               # middle
    cmpq    $4, %rax          #   i:4
    jbe     .L4                #   if <=, goto loop
rep; ret
```

# Multidimensional (Nested) Arrays

## ■ Declaration

$T \ A[R][C];$

- 2D array of data type  $T$
- $R$  rows,  $C$  columns
- Type  $T$  element requires  $K$  bytes

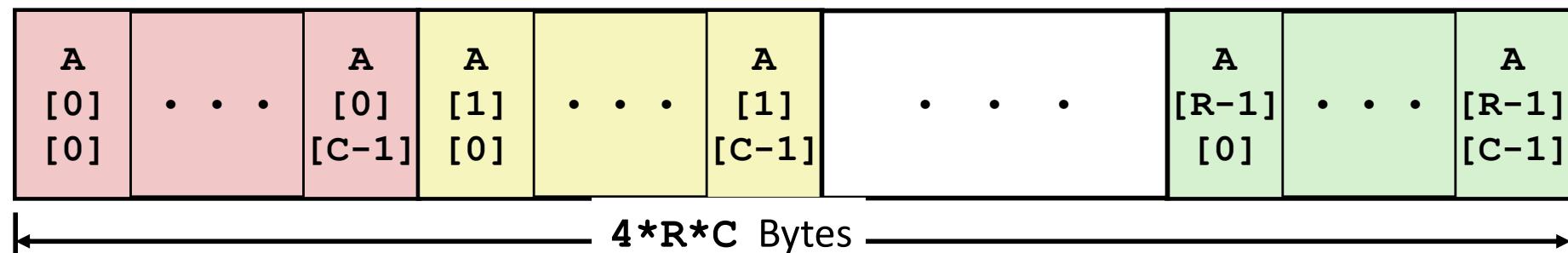
## ■ Array Size

- $R * C * K$  bytes

## ■ Arrangement

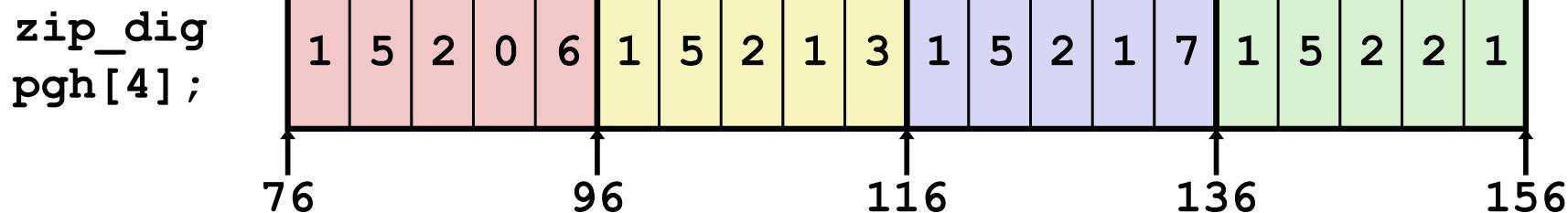
- Row-Major Ordering

`int A[R][C];`



# Nested Array Example

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
{{1, 5, 2, 0, 6},
 {1, 5, 2, 1, 3 },
 {1, 5, 2, 1, 7 },
 {1, 5, 2, 2, 1 }};
```



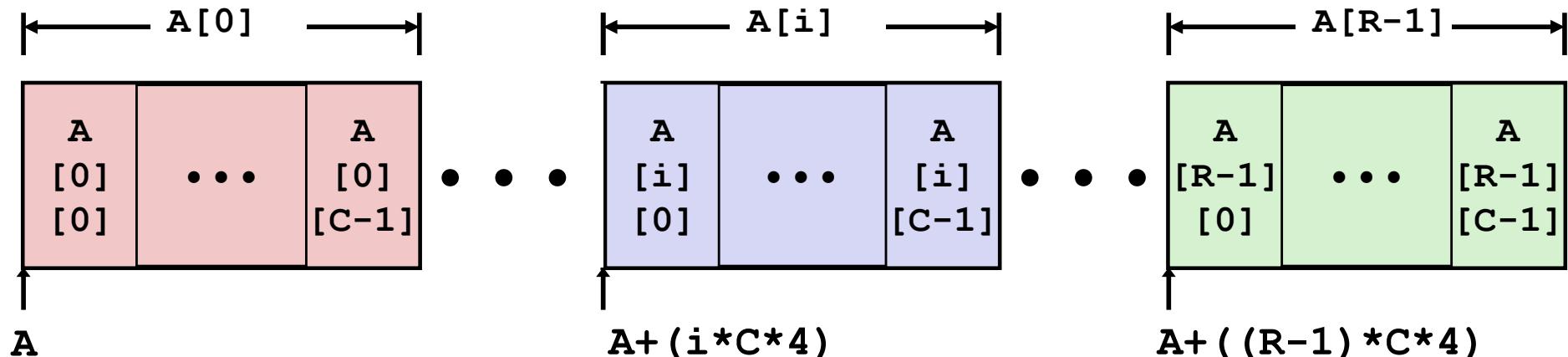
- “`zip_dig pgh[4]`” equivalent to “`int pgh[4][5]`”
  - Variable `pgh`: array of 4 elements, allocated contiguously
  - Each element is an array of 5 `int`'s, allocated contiguously
- “Row-Major” ordering of all elements in memory

# Nested Array Row Access

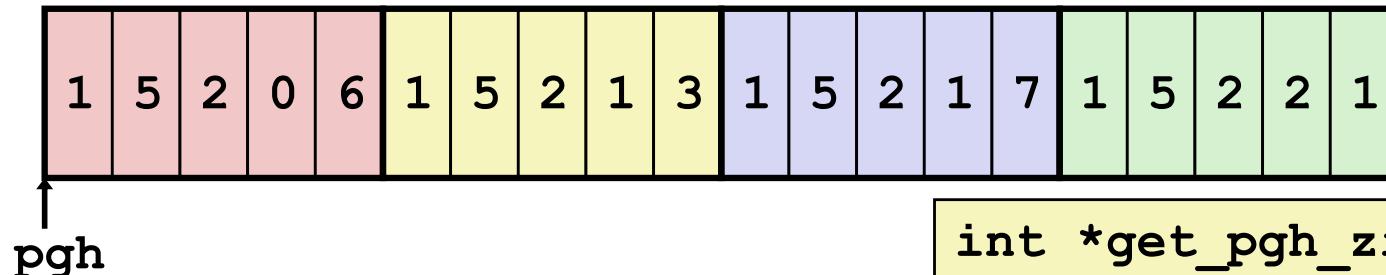
## ■ Row Vectors

- $\mathbf{A}[i]$  is array of  $C$  elements
- Each element of type  $T$  requires  $K$  bytes
- Starting address  $\mathbf{A} + i * (C * K)$

```
int A[R][C];
```



# Nested Array Row Access Code



```
int *get_pgh_zip(int index)
{
    return pgh[index];
}
```

```
# %rdi = index
leaq (%rdi,%rdi,4),%rax # 5 * index
leaq pgh(,%rax,4),%rax # pgh + (20 * index)
```

## ■ Row Vector

- `pgh[index]` is array of 5 `int`'s
- Starting address `pgh+20*index`

## ■ Machine Code

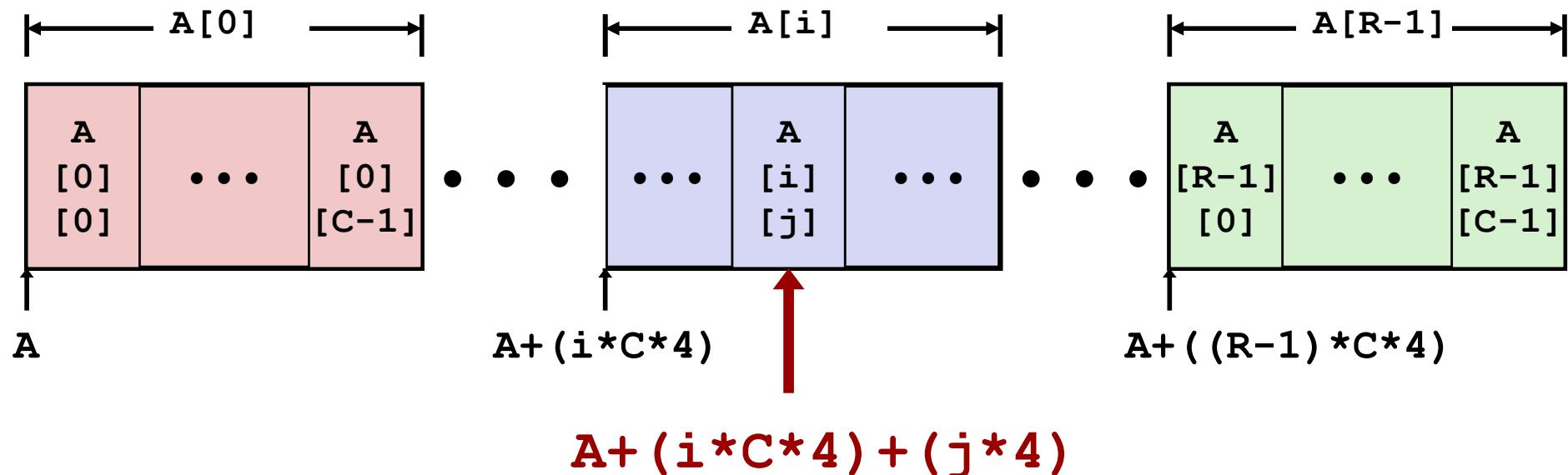
- Computes and returns address
- Compute as `pgh + 4*(index+4*index)`

# Nested Array Element Access

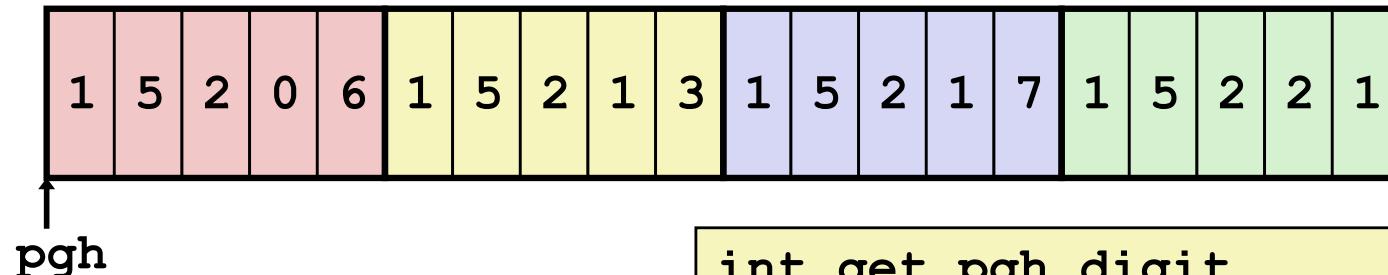
## ■ Array Elements

- $A[i][j]$  is element of type  $T$ , which requires  $K$  bytes
- Address  $A + i * (C * K) + j * K = A + (i * C + j) * K$

```
int A[R][C];
```



# Nested Array Element Access Code



```
int get_pgh_digit
    (int index, int dig)
{
    return pgh[index][dig];
}
```

```
leaq (%rdi,%rdi,4), %rax      # 5*index
addl %rax, %rsi                # 5*index+dig
movl pgh(,%rsi,4), %eax       # M[pgh + 4*(5*index+dig)]
```

## ■ Array Elements

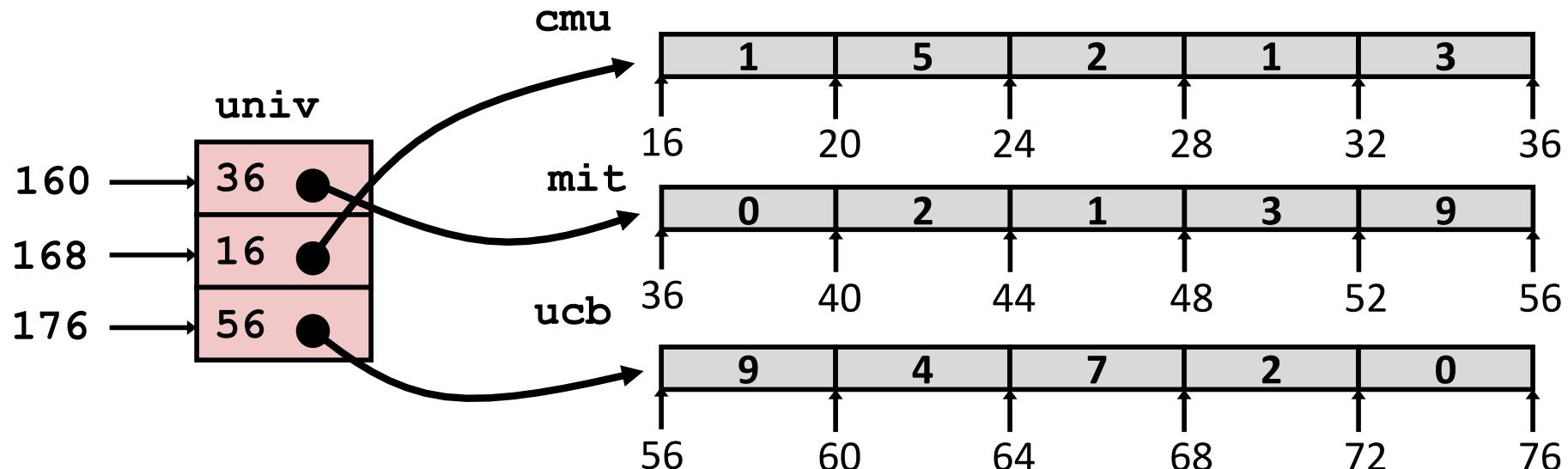
- `pgh[index][dig]` is `int`
- Address: `pgh + 20*index + 4*dig`
  - = `pgh + 4*(5*index + dig)`

# Multi-Level Array Example

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 8 bytes
- Each pointer points to array of `int`'s

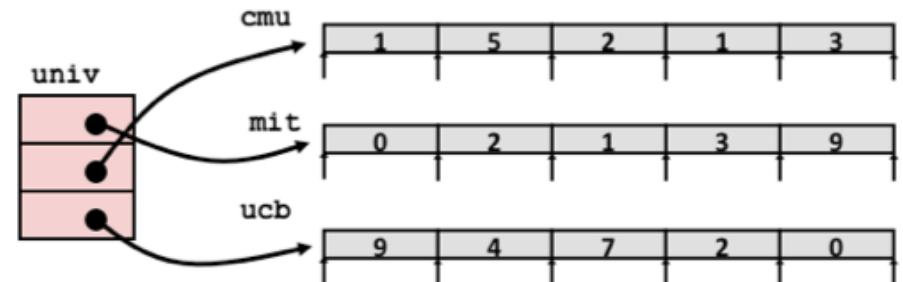
```
zip_dig cmu = { 1, 5, 2, 1, 3 };  
zip_dig mit = { 0, 2, 1, 3, 9 };  
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

```
#define UCOUNT 3  
int *univ[UCOUNT] = {mit, cmu, ucb};
```



# Element Access in Multi-Level Array

```
int get_univ_digit  
    (size_t index, size_t digit)  
{  
    return univ[index][digit];  
}
```



```
salq    $2, %rsi          # 4*digit  
addq    univ(,%rdi,8), %rsi # p = univ[index] + 4*digit  
movl    (%rsi), %eax       # return *p  
ret
```

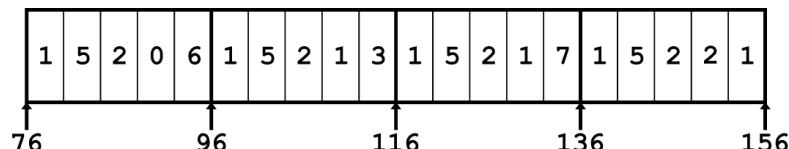
## ■ Computation

- Element access **Mem[Mem[univ+8\*index]+4\*digit]**
- Must do two memory reads
  - First get pointer to row array
  - Then access element within array

# Array Element Accesses

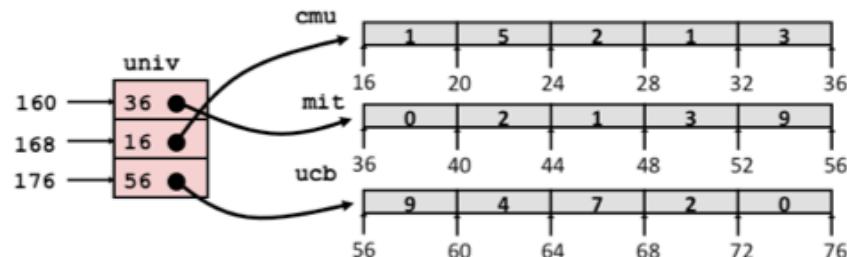
Nested array

```
int get_pgh_digit
    (size_t index, size_t digit)
{
    return pgh[index][digit];
}
```



Multi-level array

```
int get_univ_digit
    (size_t index, size_t digit)
{
    return univ[index][digit];
}
```



Accesses looks similar in C, but address computations very different:

`Mem[pgh+20*index+4*digit]`

`Mem[Mem[univ+8*index]+4*digit]`

# NxN Matrix Code

## ■ Fixed dimensions

- Know value of N at compile time

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele(fix_matrix a,
            size_t i, size_t j)
{
    return a[i][j];
}
```

## ■ Variable dimensions, explicit indexing

- Traditional way to implement dynamic arrays

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele(size_t n, int *a,
            size_t i, size_t j)
{
    return a[IDX(n,i,j)];
}
```

## ■ Variable dimensions, implicit indexing

- Now supported by gcc

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n],
            size_t i, size_t j) {
    return a[i][j];
}
```

# 16 X 16 Matrix Access

## ■ Array Elements

- Address  $\mathbf{A} + i * (C * K) + j * K$
- C = 16, K = 4

```
/* Get element a[i][j] */  
int fix_ele(fix_matrix a, size_t i, size_t j) {  
    return a[i][j];  
}
```

```
# a in %rdi, i in %rsi, j in %rdx  
salq    $6, %rsi          # 64*i  
addq    %rsi, %rdi         # a + 64*i  
movl    (%rdi,%rdx,4), %eax # M[a + 64*i + 4*j]  
ret
```

# $n \times n$ Matrix Access

## ■ Array Elements

- Address  $\mathbf{A} + i * (C * K) + j * K$
- $C = n, K = 4$
- Must perform integer multiplication

```
/* Get element a[i][j] */  
int var_ele(size_t n, int a[n][n], size_t i, size_t j)  
{  
    return a[i][j];  
}
```

```
# n in %rdi, a in %rsi, i in %rdx, j in %rcx  
imulq  %rdx, %rdi          # n*i  
leaq   (%rsi,%rdi,4), %rax # a + 4*n*i  
movl   (%rax,%rcx,4), %eax # a + 4*n*i + 4*j  
ret
```

# Today

## ■ Arrays

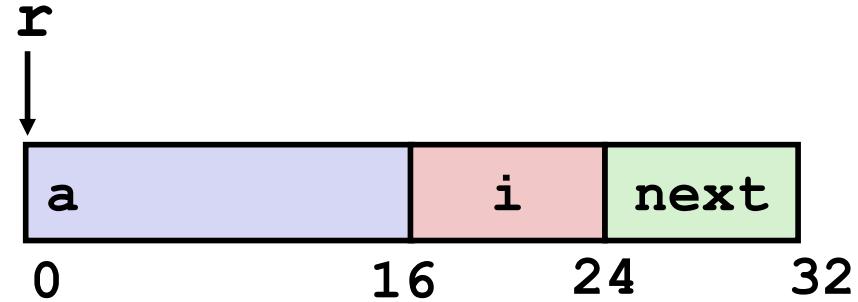
- One-dimensional
- Multi-dimensional (nested)
- Multi-level

## ■ Structures

- Allocation
- Access
- Alignment

# Structure Representation

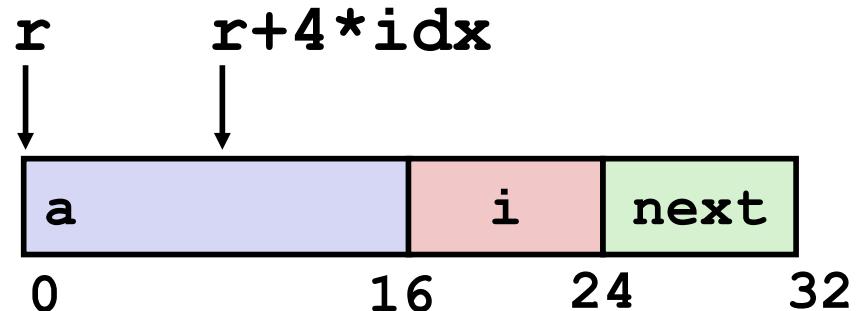
```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec *next;  
};
```



- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration
  - Even if another ordering could yield a more compact representation
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

# Generating Pointer to Structure Member

```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec *next;  
};
```



- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Compute as `r + 4*idx`

```
int *get_ap  
(struct rec *r, size_t idx)  
{  
    return &r->a[idx];  
}
```

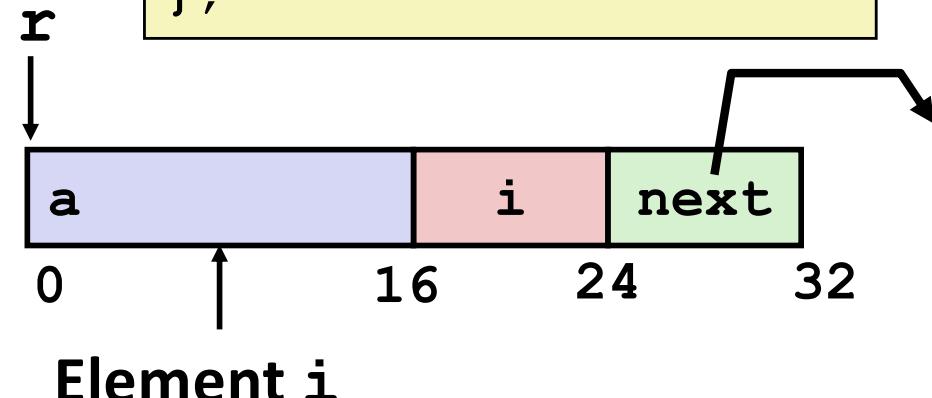
```
# r in %rdi, idx in %rsi  
leaq (%rdi,%rsi,4), %rax  
ret
```

# Following Linked List

## ■ C Code

```
void set_val
    (struct rec *r, int val)
{
    while (r) {
        int i = r->i;
        r->a[i] = val;
        r = r->next;
    }
}
```

```
struct rec {
    int a[4];
    int i;
    struct rec *next;
};
```



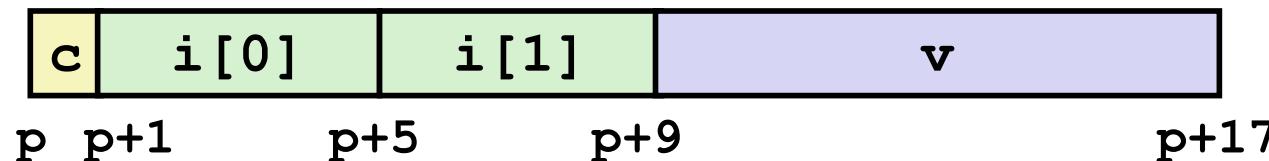
Element i

Register	Value
%rdi	r
%rsi	val

```
.L11:                                # loop:
    movslq  16(%rdi), %rax      #   i = M[r+16]
    movl    %esi, (%rdi,%rax,4) #   M[r+4*i] = val
    movq    24(%rdi), %rdi      #   r = M[r+24]
    testq   %rdi, %rdi         #   Test r
    jne     .L11                #   if !=0 goto loop
```

# Structures & Alignment

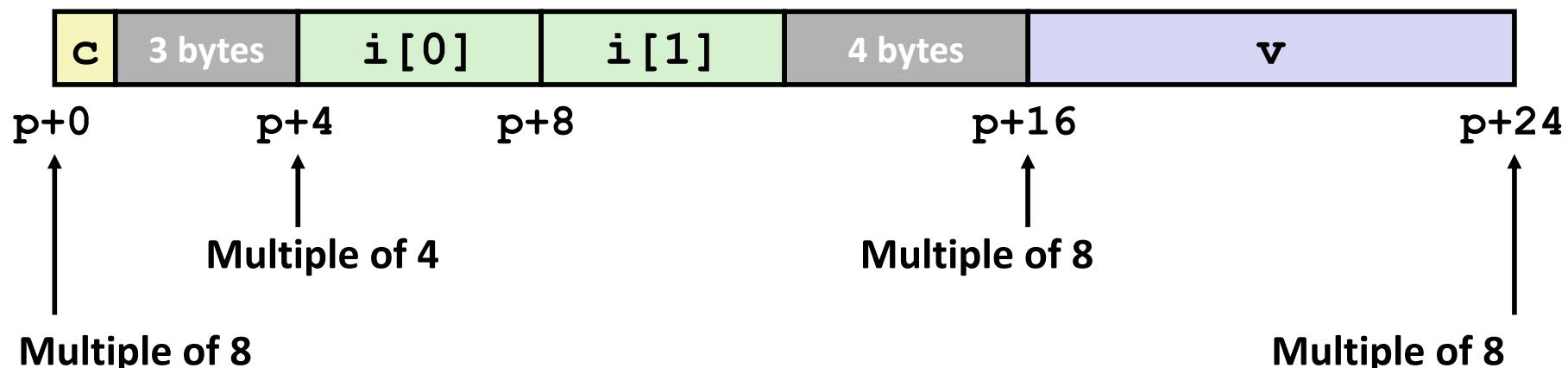
## ■ Unaligned Data



```
struct S1 {  
    char c;  
    int i[2];  
    double v;  
} *p;
```

## ■ Aligned Data

- Primitive data type requires K bytes
- Address must be multiple of K



# Alignment Principles

## ■ Aligned Data

- Primitive data type requires K bytes
- Address must be multiple of K
- Required on some machines; advised on x86-64

## ■ Motivation for Aligning Data

- Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
  - Inefficient to load or store datum that spans quad word boundaries
  - Virtual memory trickier when datum spans 2 pages

## ■ Compiler

- Inserts gaps in structure to ensure correct alignment of fields

# Specific Cases of Alignment (x86-64)

- **1 byte: char, ...**
  - no restrictions on address
- **2 bytes: short, ...**
  - lowest 1 bit of address must be  $0_2$
- **4 bytes: int, float, ...**
  - lowest 2 bits of address must be  $00_2$
- **8 bytes: double, long, char \*, ...**
  - lowest 3 bits of address must be  $000_2$
- **16 bytes: long double (GCC on Linux)**
  - lowest 4 bits of address must be  $0000_2$

# Satisfying Alignment with Structures

## ■ Within structure:

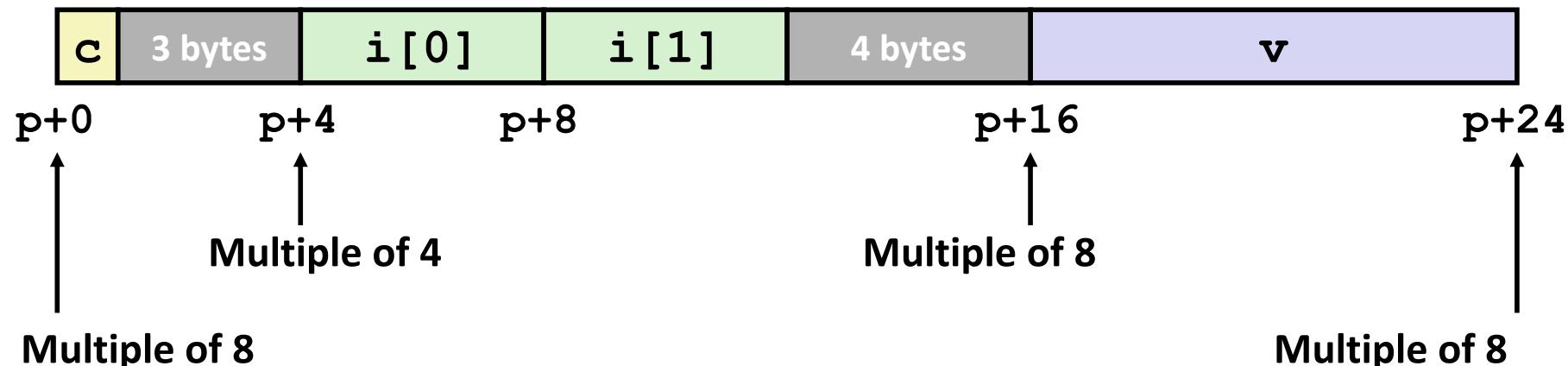
- Must satisfy each element's alignment requirement

## ■ Overall structure placement

- Each structure has alignment requirement K
  - $K = \text{Largest alignment of any element}$
- Initial address & structure length must be multiples of K

## ■ Example:

- $K = 8$ , due to **double** element

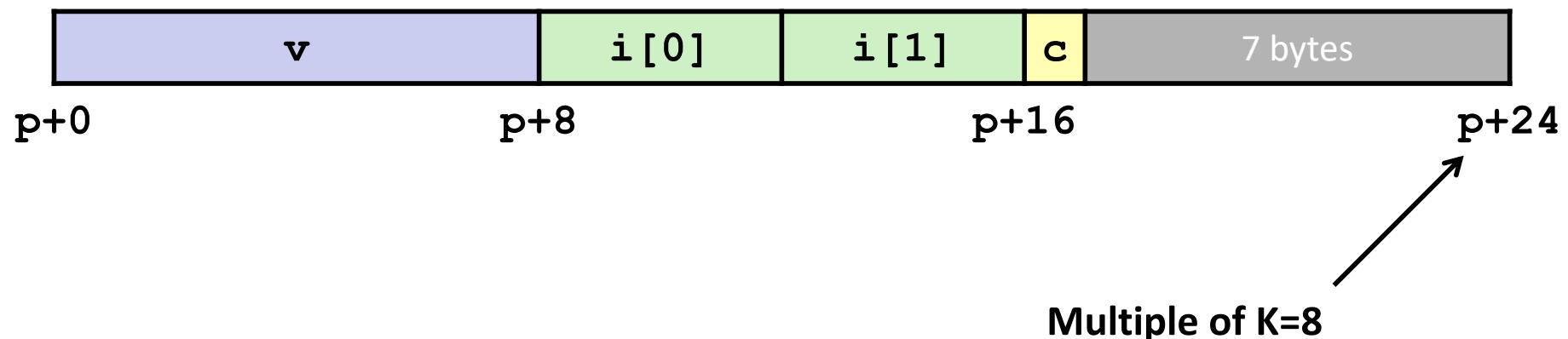


```
struct S1 {  
    char c;  
    int i[2];  
    double v;  
} *p;
```

# Meeting Overall Alignment Requirement

- For largest alignment requirement K
- Overall structure must be multiple of K

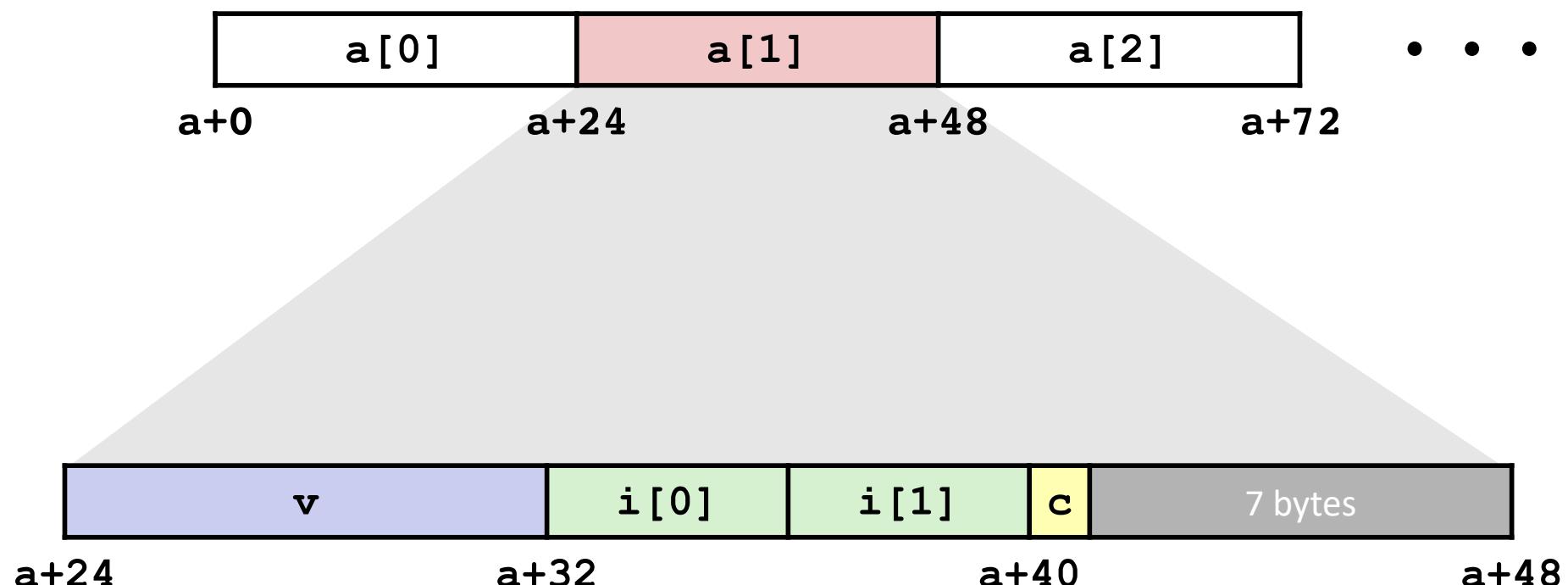
```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} *p;
```



# Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

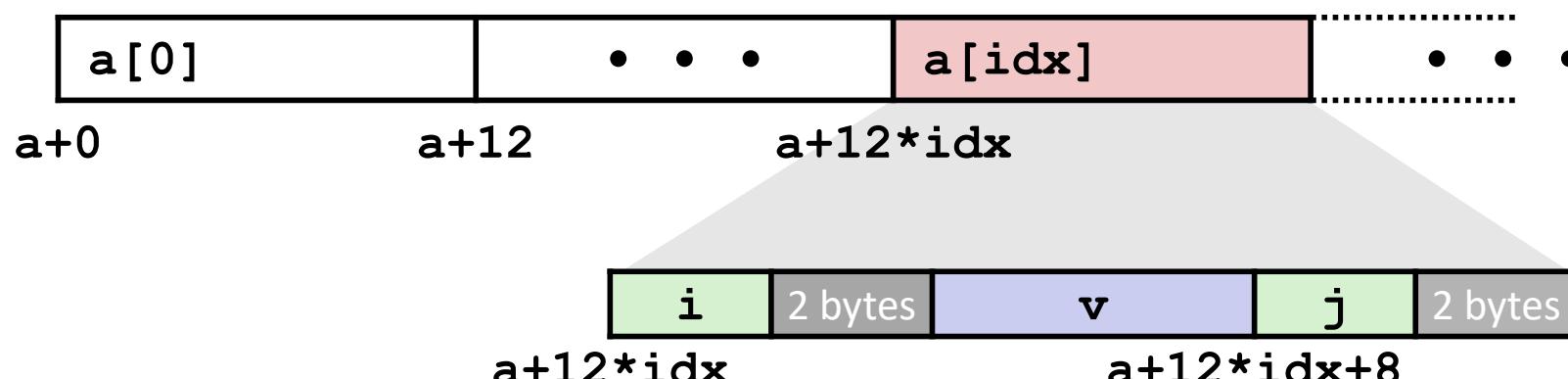
```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} a[10];
```



# Accessing Array Elements

- Compute array offset  $12 * \text{idx}$ 
  - `sizeof(S3)`, including alignment spacers
- Element `j` is at offset 8 within structure
- Assembler gives offset `a+8`
  - Resolved during linking

```
struct S3 {  
    short i;  
    float v;  
    short j;  
} a[10];
```



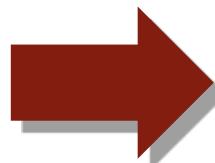
```
short get_j(int idx)  
{  
    return a[idx].j;  
}
```

```
# %rdi = idx  
leaq (%rdi,%rdi,2),%rax # 3*idx  
movzwl a+8(%rax,4),%eax
```

# Saving Space

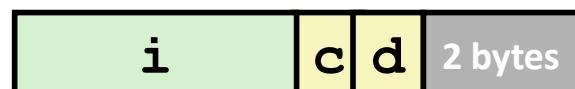
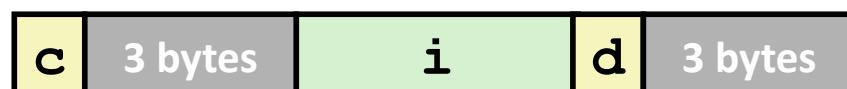
- Put large data types first

```
struct S4 {  
    char c;  
    int i;  
    char d;  
} *p;
```



```
struct S5 {  
    int i;  
    char c;  
    char d;  
} *p;
```

- Effect (K=4)



# Summary

## ■ Arrays

- Elements packed into contiguous region of memory
- Use index arithmetic to locate individual elements

## ■ Structures

- Elements packed into single region of memory
- Access using offsets determined by compiler
- Possible require internal and external padding to ensure alignment

## ■ Combinations

- Can nest structure and array code arbitrarily

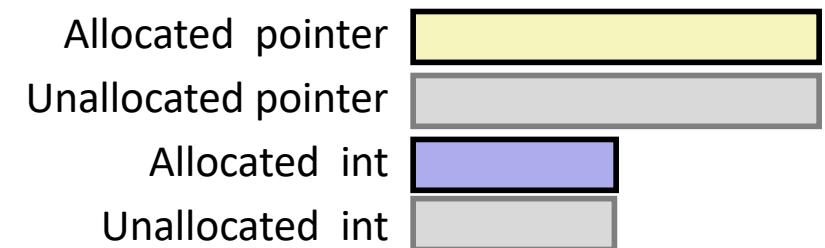
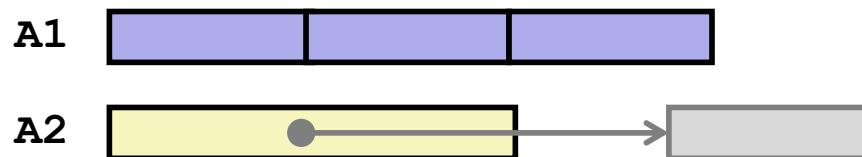
# Understanding Pointers & Arrays #1

Decl	<i>An</i>			<i>*An</i>		
	Cmp	Bad	Size	Cmp	Bad	Size
<code>int A1[3]</code>						
<code>int *A2</code>						

- **Cmp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by sizeof**

# Understanding Pointers & Arrays #1

Decl	<i>An</i>			<i>*An</i>		
	Cmp	Bad	Size	Cmp	Bad	Size
<code>int A1[3]</code>	Y	N	12	Y	N	4
<code>int *A2</code>	Y	N	8	Y	Y	4



- **Cmp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by sizeof**

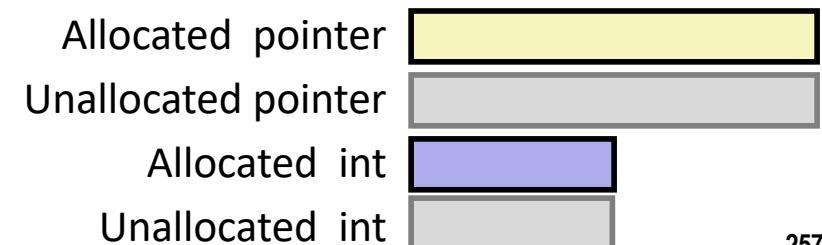
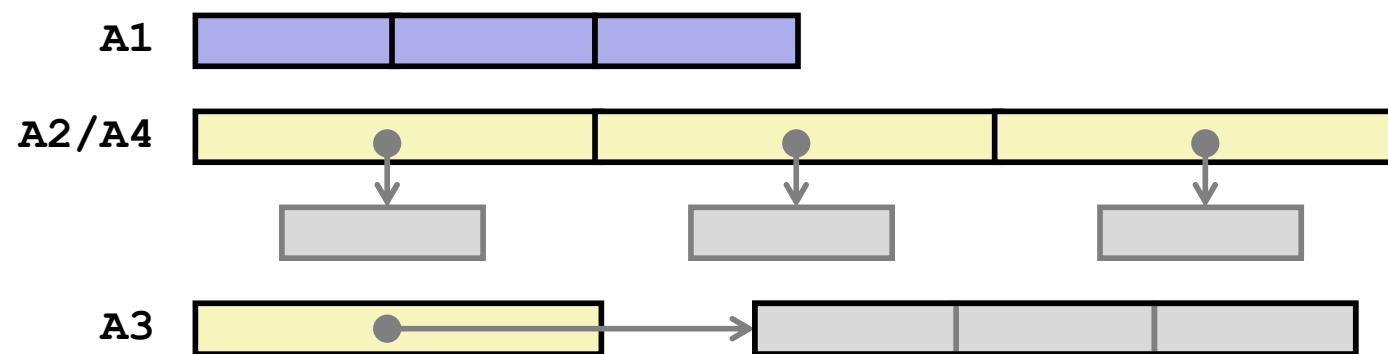
# Understanding Pointers & Arrays #2

Decl	<i>An</i>			<i>*An</i>			<i>**An</i>		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
<code>int A1[3]</code>									
<code>int *A2[3]</code>									
<code>int (*A3)[3]</code>									
<code>int (*A4[3])</code>									

- **Cmp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

# Understanding Pointers & Arrays #2

Decl	<i>An</i>			<i>*An</i>			<i>**An</i>		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
<code>int A1[3]</code>	Y	N	12	Y	N	4	N	-	-
<code>int *A2[3]</code>	Y	N	24	Y	N	8	Y	Y	4
<code>int (*A3)[3]</code>	Y	N	8	Y	Y	12	Y	Y	4
<code>int (*A4[3])</code>	Y	N	24	Y	N	8	Y	Y	4

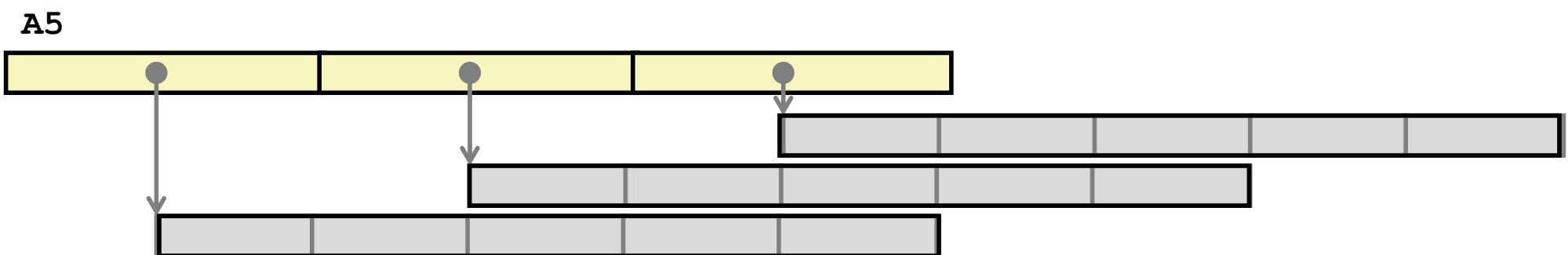
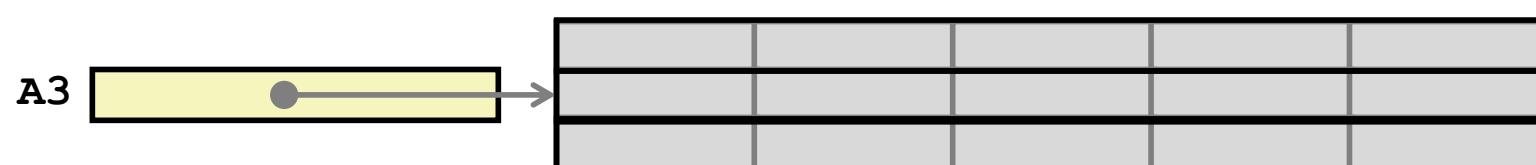
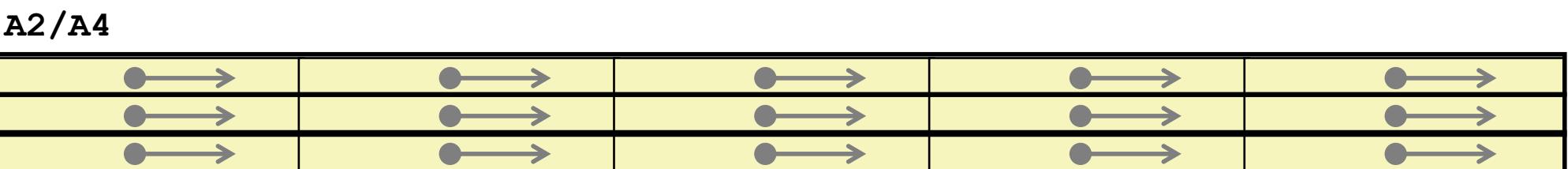
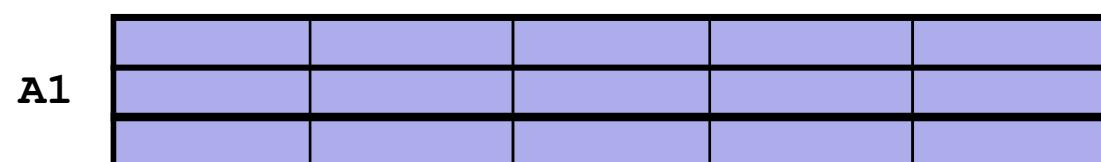
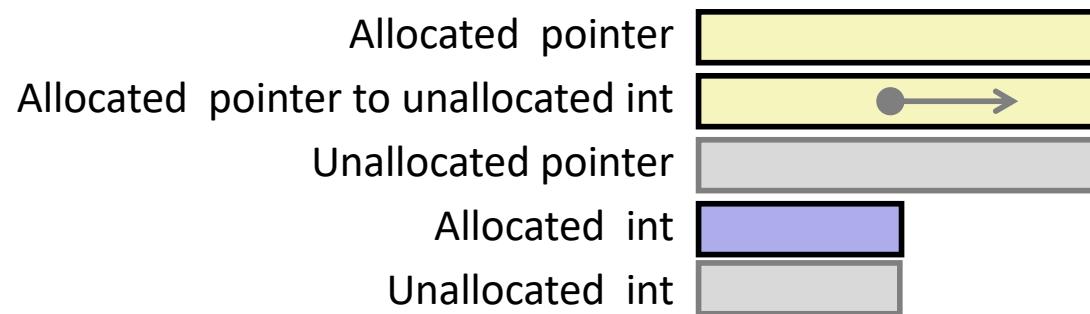


# Understanding Pointers & Arrays #3

Decl	An			*An			**An		
	Cm p	Bad	Size	Cm p	Bad	Size	Cm p	Bad	Size
<code>int A1[3][5]</code>									
<code>int *A2[3][5]</code>									
<code>int (*A3)[3][5]</code>									
<code>int *(A4[3][5])</code>									
<code>int (*A5[3])[5]</code>									

- **Cmp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by sizeof**

Decl	***An		
	Cm p	Bad	Size
<code>int A1[3][5]</code>			
<code>int *A2[3][5]</code>			
<code>int (*A3)[3][5]</code>			
<code>int *(A4[3][5])</code>			
<code>int (*A5[3])[5]</code>			



## Declaration

```
int A1[3][5]
```

```
int *A2[3][5]
```

```
int (*A3)[3][5]
```

```
int *(A4[3][5])
```

```
int (*A5[3])[5]
```

# Understanding Pointers & Arrays #3

Decl	An			*An			**An		
	Cm p	Bad	Size	Cm p	Bad	Size	Cm p	Bad	Size
int A1[3][5]	Y	N	60	Y	N	20	Y	N	4
int *A2[3][5]	Y	N	120	Y	N	40	Y	N	8
int (*A3)[3][5]	Y	N	8	Y	Y	60	Y	Y	20
int *(A4[3][5])	Y	N	120	Y	N	40	Y	N	8
int (*A5[3])[5]	Y	N	24	Y	N	8	Y	Y	20

- Cmp: Compiles (Y/N)
- Bad: Possible bad pointer reference (Y/N)
- Size: Value returned by sizeof

Decl	***An		
	Cm p	Bad	Size
int A1[3][5]	N	-	-
int *A2[3][5]	Y	Y	4
int (*A3)[3][5]	Y	Y	4
int *(A4[3][5])	Y	Y	4
int (*A5[3])[5]	Y	Y	4

# **Machine-Level Programming V: Advanced Topics**

# Today

- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection
- **Unions**

# x86-64 Linux Memory Layout

*not drawn to scale*

## ■ Stack

- Runtime stack (8MB limit)
- E. g., local variables

## ■ Heap

- Dynamically allocated as needed
- When call `malloc()`, `calloc()`, `new`

## ■ Data

- Statically allocated data
- E.g., global vars, `static` vars, string constants

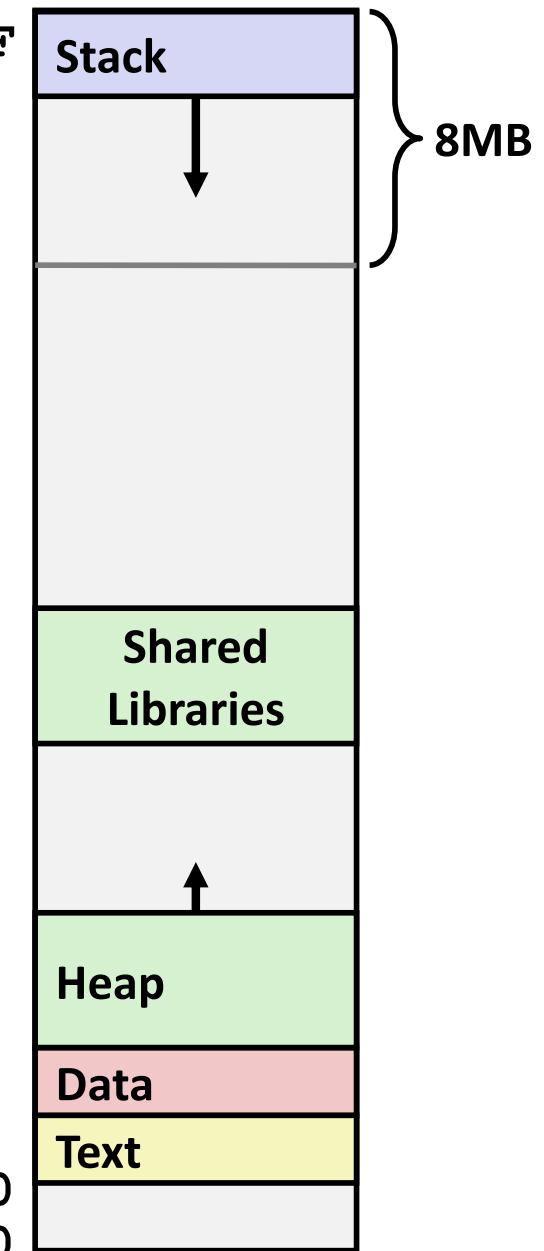
## ■ Text / Shared Libraries

- Executable machine instructions
- Read-only

Hex Address



400000  
000000



*not drawn to scale*

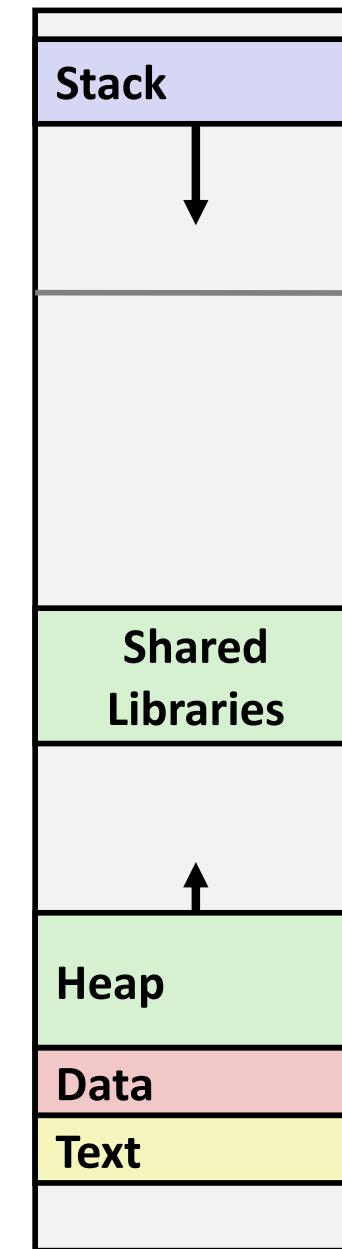
# Memory Allocation Example

```
char big_array[1L<<24]; /* 16 MB */
char huge_array[1L<<31]; /* 2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8); /* 256 B */
    p3 = malloc(1L << 32); /* 4 GB */
    p4 = malloc(1L << 8); /* 256 B */
    /* Some print statements ... */
}
```

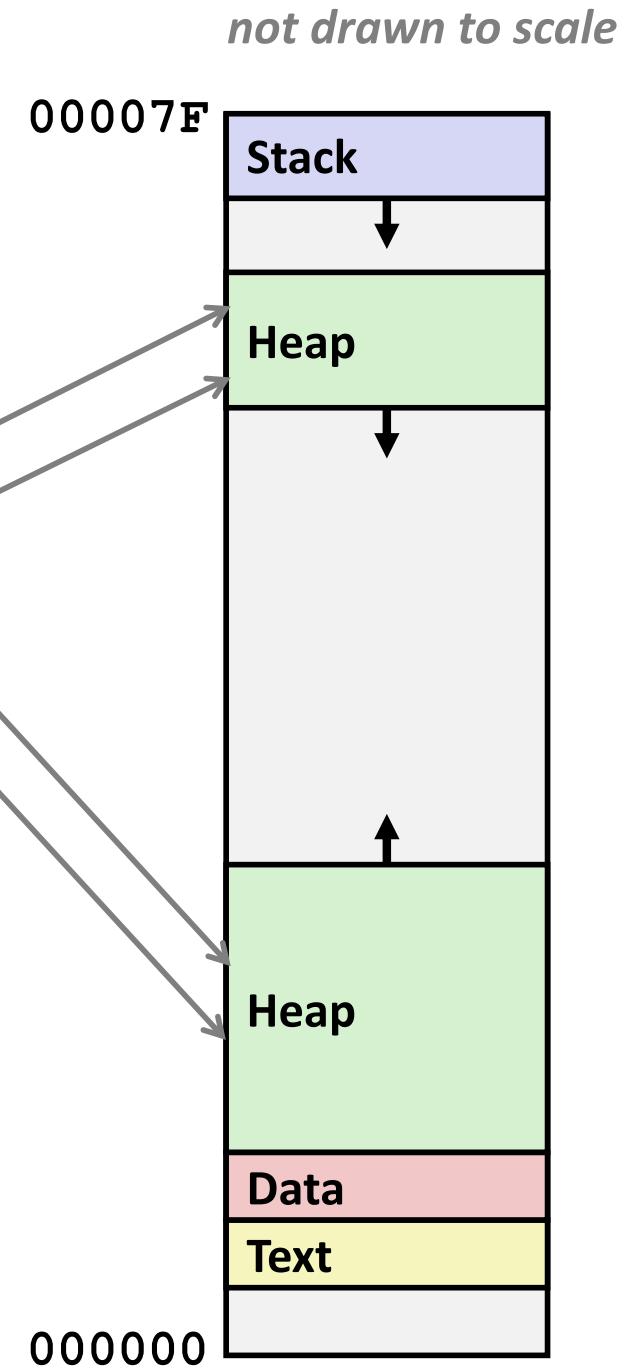


*Where does everything go?*

# x86-64 Example Addresses

*address range  $\sim 2^{47}$*

local	0x00007ffe4d3be87c
p1	0x00007f7262a1e010
p3	0x00007f7162a1d010
p4	0x000000008359d120
p2	0x000000008359d010
big_array	0x0000000080601060
huge_array	0x0000000000601060
main()	0x000000000040060c
useless()	0x0000000000400590



# Today

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
- Unions

# Recall: Memory Referencing Bug Example

```
typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i) {
    volatile struct_t s;
    s.d = 3.14;
    s.a[i] = 1073741824; /* Possibly out of bounds */
    return s.d;
}
```

<b>fun(0)</b>	<b>→</b>	<b>3.14</b>
<b>fun(1)</b>	<b>→</b>	<b>3.14</b>
<b>fun(2)</b>	<b>→</b>	<b>3.1399998664856</b>
<b>fun(3)</b>	<b>→</b>	<b>2.00000061035156</b>
<b>fun(4)</b>	<b>→</b>	<b>3.14</b>
<b>fun(6)</b>	<b>→</b>	<b>Segmentation fault</b>

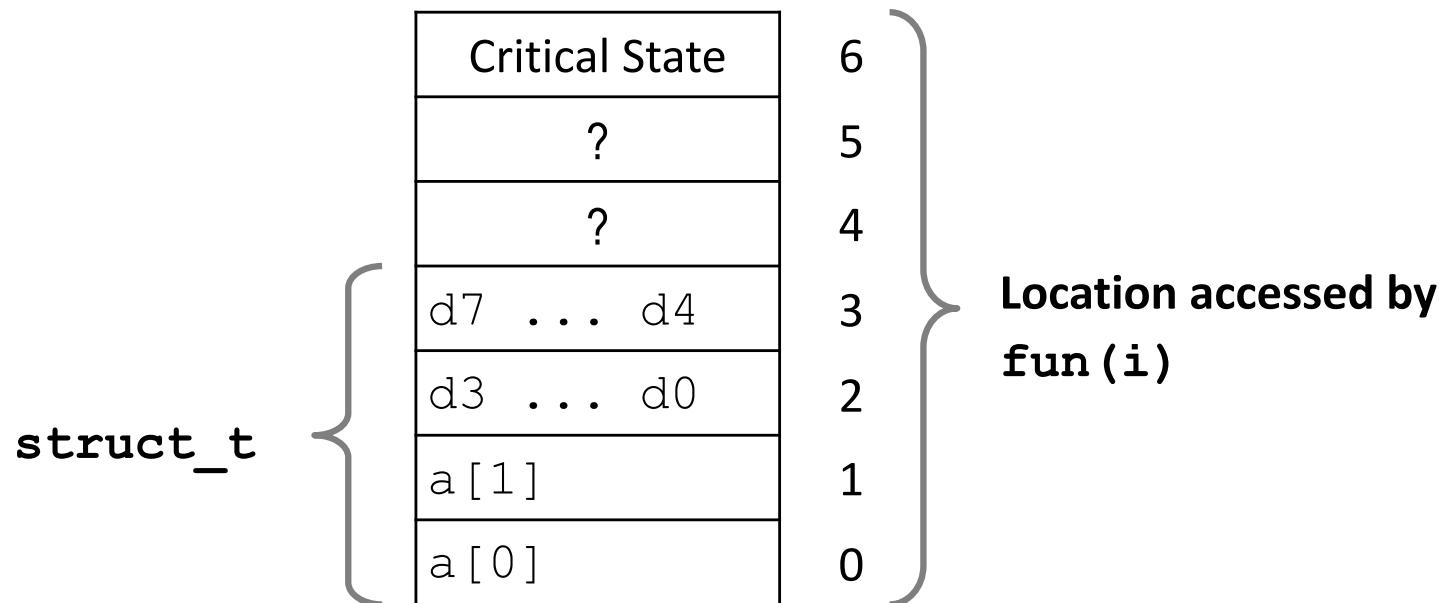
- Result is system specific

# Memory Referencing Bug Example

```
typedef struct {  
    int a[2];  
    double d;  
} struct_t;
```

fun(0)	→	3.14
fun(1)	→	3.14
fun(2)	→	3.1399998664856
fun(3)	→	2.00000061035156
fun(4)	→	3.14
fun(6)	→	Segmentation fault

## Explanation:



# Such problems are a BIG deal

- **Generally called a “buffer overflow”**
  - when exceeding the memory size allocated for an array
- **Why a big deal?**
  - It's the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance
- **Most common form**
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing

# String Library Code

## ■ Implementation of Unix function `gets()`

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

- No way to specify limit on number of characters to read
- Similar problems with other library functions
  - `strcpy`, `strcat`: Copy strings of arbitrary length
  - `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification

# Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

←btw, how big  
is big enough?

```
void call_echo() {
    echo();
}
```

```
unix> ./bufdemo-nsp
Type a string: 012345678901234567890123
012345678901234567890123
```

```
unix> ./bufdemo-nsp
Type a string: 0123456789012345678901234
Segmentation Fault
```

# Buffer Overflow Disassembly

echo:

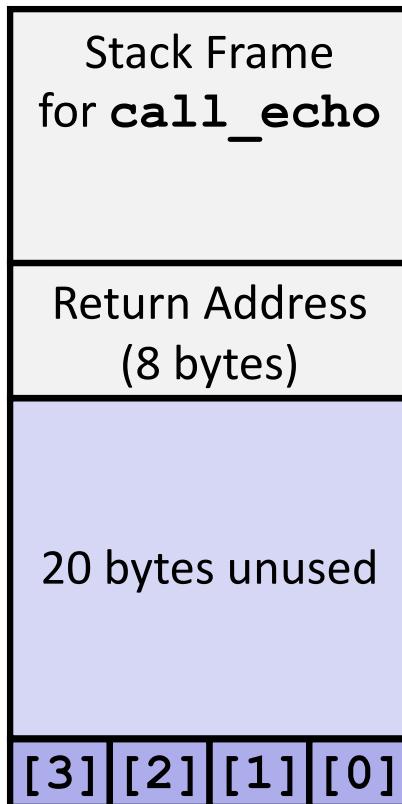
```
00000000004006cf <echo>:  
4006cf: 48 83 ec 18          sub    $0x18,%rsp  
4006d3: 48 89 e7          mov    %rsp,%rdi  
4006d6: e8 a5 ff ff ff      callq   400680 <gets>  
4006db: 48 89 e7          mov    %rsp,%rdi  
4006de: e8 3d fe ff ff      callq   400520 <puts@plt>  
4006e3: 48 83 c4 18          add    $0x18,%rsp  
4006e7: c3                  retq
```

call\_echo:

```
4006e8: 48 83 ec 08          sub    $0x8,%rsp  
4006ec: b8 00 00 00 00      mov    $0x0,%eax  
4006f1: e8 d9 ff ff ff      callq   4006cf <echo>  
4006f6: 48 83 c4 08          add    $0x8,%rsp  
4006fa: c3                  retq
```

# Buffer Overflow Stack

*Before call to gets*



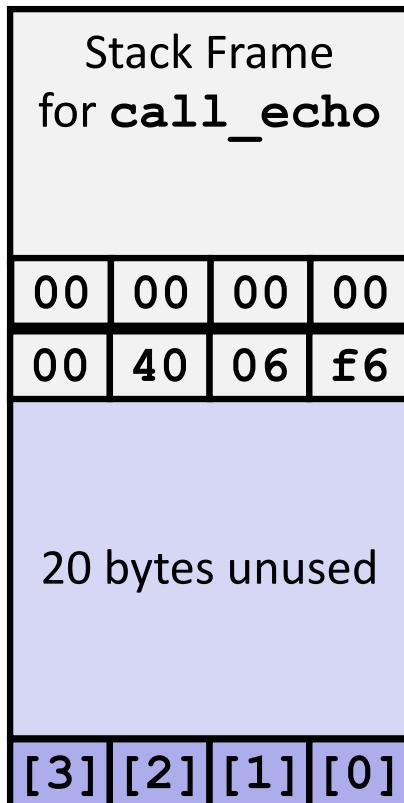
```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

`buf ← %rsp`

```
echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    . . .
```

# Buffer Overflow Stack Example

*Before call to gets*



```
void echo()
{
    char buf[4];
    gets(buf);
    ...
}
```

```
echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...
}
```

call\_echo:

```
...
4006f1: callq 4006cf <echo>
4006f6: add    $0x8,%rsp
...
```

# Buffer Overflow Stack Example #1

*After call to gets*

Stack Frame for <code>call_echo</code>			
00	00	00	00
00	40	06	f6
00	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

```
void echo()
{
    char buf[4];
    gets(buf);
    ...
}
```

```
echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...
.
```

call\_echo:

```
...
4006f1: callq 4006cf <echo>
4006f6: add    $0x8,%rsp
...
```

`buf`  $\leftarrow$  `%rsp`

```
unix>./bufdemo-nsp
Type a string: 01234567890123456789012
01234567890123456789012
```

Overflowed buffer, but did not corrupt state

# Buffer Overflow Stack Example #2

*After call to gets*

Stack Frame for <code>call_echo</code>			
00	00	00	00
00	40	00	34
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

```
void echo()
{
    char buf[4];
    gets(buf);
    ...
}
```

```
echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...
.
```

call\_echo:

```
...
4006f1: callq 4006cf <echo>
4006f6: add    $0x8,%rsp
...
```

`buf`  $\leftarrow$  `%rsp`

```
unix>./bufdemo-nsp
Type a string: 0123456789012345678901234
Segmentation Fault
```

Overflowed buffer and corrupted return pointer

# Buffer Overflow Stack Example #3

*After call to gets*

Stack Frame for <code>call_echo</code>			
00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

```
void echo()
{
    char buf[4];
    gets(buf);
    ...
}
```

```
echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...
.
```

call\_echo:

```
...
4006f1: callq 4006cf <echo>
4006f6: add    $0x8,%rsp
...
```

`buf`  $\leftarrow$  `%rsp`

```
unix>./bufdemo-nsp
Type a string: 012345678901234567890123
012345678901234567890123
```

Overflowed buffer, corrupted return pointer, but program seems to work!

# Buffer Overflow Stack Example #3 Explained

*After call to gets*

Stack Frame for <code>call_echo</code>			
00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

`buf ← %rsp`

`register_tm_clones:`

```
...  
400600: mov    %rsp,%rbp  
400603: mov    %rax,%rdx  
400606: shr    $0x3f,%rdx  
40060a: add    %rdx,%rax  
40060d: sar    %rax  
400610: jne    400614  
400612: pop    %rbp  
400613: retq
```

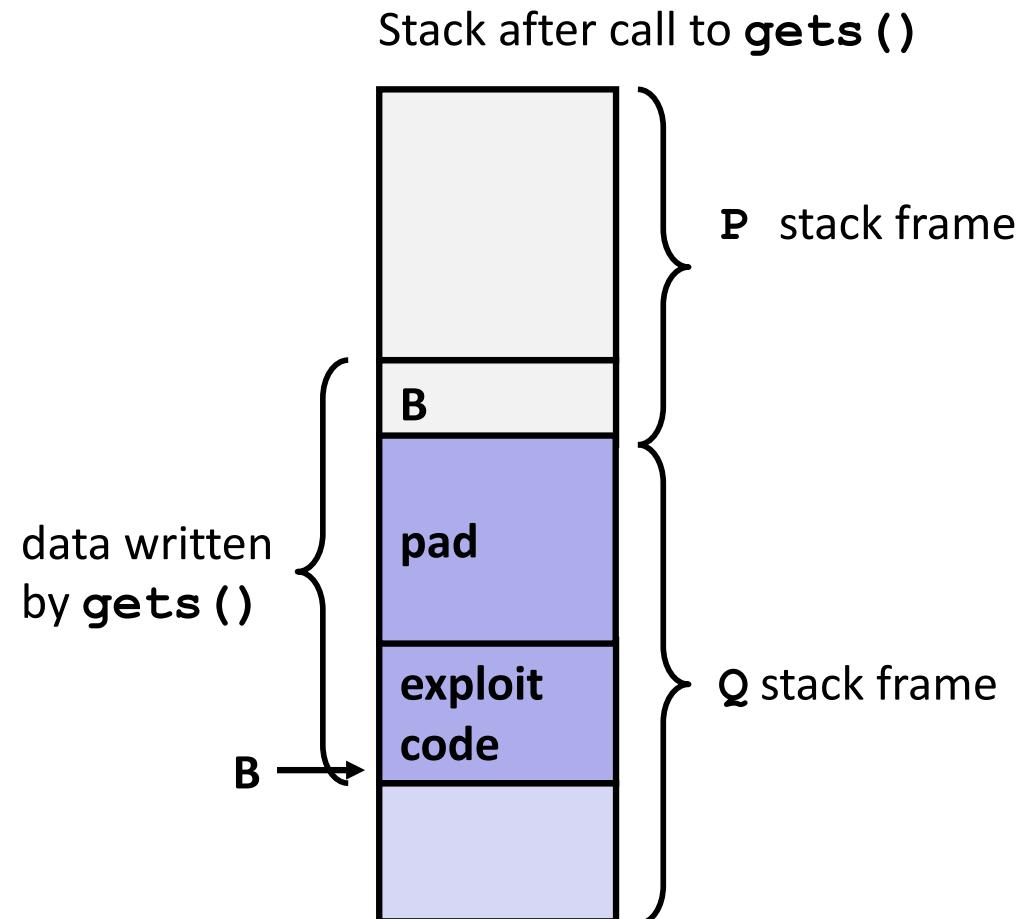
“Returns” to unrelated code  
Lots of things happen, without modifying critical state  
Eventually executes `retq` back to `main`

# Code Injection Attacks

```
void P() {  
    Q();  
    ...  
}
```

return address  
A

```
int Q() {  
    char buf[64];  
    gets(buf);  
    ...  
    return ...;  
}
```



- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When Q executes `ret`, will jump to exploit code

# Exploits Based on Buffer Overflows

- *Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines*
- Distressingly common in real programs
  - Programmers keep making the same mistakes 😞
  - Recent measures make these attacks much more difficult
- Examples across the decades
  - Original “Internet worm” (1988)
  - “IM wars” (1999)
  - Twilight hack on Wii (2000s)
  - ... and many, many more
- You will learn some of the tricks in attacklab
  - Hopefully to convince you to never leave such holes in your programs!!

# Example: the original Internet worm (1988)

## ■ Exploited a few vulnerabilities to spread

- Early versions of the finger server (fingerd) used `gets()` to read the argument sent by the client:
  - `finger droh@cs.cmu.edu`
- Worm attacked fingerd server by sending phony argument:
  - `finger "exploit-code padding new-return-address"`
  - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

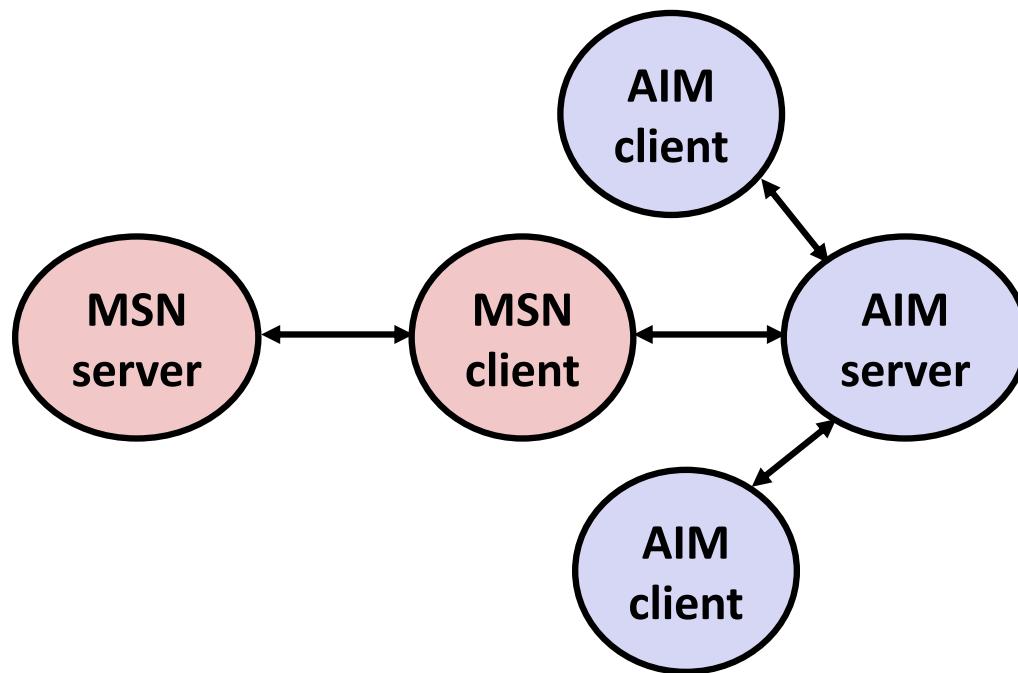
## ■ Once on a machine, scanned for other machines to attack

- invaded ~6000 computers in hours (10% of the Internet ☺)
  - see June 1989 article in *Comm. of the ACM*
- the young author of the worm was prosecuted...
- and CERT was formed... still homed at CMU

# Example 2: IM War

## ■ July, 1999

- Microsoft launches MSN Messenger (instant messaging system).
- Messenger clients can access popular AOL Instant Messaging Service (AIM) servers



# IM War (cont.)

## ■ August 1999

- Mysteriously, Messenger clients can no longer access AIM servers
- Microsoft and AOL begin the IM war:
  - AOL changes server to disallow Messenger clients
  - Microsoft makes changes to clients to defeat AOL changes
  - At least 13 such skirmishes
- What was really happening?
  - AOL had discovered a buffer overflow bug in their own AIM clients
  - They exploited it to detect and block Microsoft: the exploit code returned a 4-byte signature (the bytes at some location in the AIM client) to server
  - When Microsoft changed code to match signature, AOL changed signature location

Date: Wed, 11 Aug 1999 11:30:57 -0700 (PDT)  
From: Phil Bucking <philbucking@yahoo.com>  
Subject: AOL exploiting buffer overrun bug in their own software!  
To: rms@pharlap.com

Mr. Smith,

I am writing you because I have discovered something that I think you might find interesting because you are an Internet security expert with experience in this area. I have also tried to contact AOL but received no response.

I am a developer who has been working on a revolutionary new instant messaging client that should be released later this year.

...

It appears that the AIM client has a buffer overrun bug. By itself this might not be the end of the world, as MS surely has had its share. But AOL is now \*exploiting their own buffer overrun bug\* to help in its efforts to block MS Instant Messenger.

....

Since you have significant credibility with the press I hope that you can use this information to help inform people that behind AOL's friendly exterior they are nefario

Sincerely,  
Phil Bucking  
Founder, Bucking Consulting  
philbucking@yahoo.com

***It was later determined that this email originated from within Microsoft!***

# CIH (Chernobyl Virus)

- First wave of attack: 1999.4.26
- Second wave: 2000.4.26
- 60 Million Computers were infected including 15% of all Korean Computers
- CIH 2.0: 80% of the work was done when he was arrested



5	0E	00	B7	HH	ZH	0E	58	11	70000000000000000000000000000000
2	E4	88	00	91	E2	FE	E9	'Q0=200e a0=	
3	C6	00	20	E2	FE	B4	0U0*	YÁ~ E9 ô=	
F	D6	33	DB	B7	80	53	óê fÃF=2> i3=àCS		
8	53	51	51	51	68	01	âý,h	àSQQQh@	
C	AC	00	00	00	CD	20	± 0AQQI=Kiy%	=	
4	05	FE	46	4D	EB	EE	♦ ▶ fâ~†t±=FMù-		
8	08	88	01	C6	00	80	0^>äFMÇùöêææä Ç		
7	87	D5	EC	0C	44	97	êæ@Juç¹'ùç¹ý?Dü		
A	66	27	53	00	01	00	c¹'ùç¹-+ :f'S @		
0	40	00	43	49	48	20	h 0 A 0 2 0	CIH	
0	00	00	00	00	00	00	v1.2 TTIT		
0	00	00	00	00	00	00			

# Aside: Worms and Viruses

- **Worm: A program that**
  - Can run by itself
  - Can propagate a fully working version of itself to other computers
- **Virus: Code that**
  - Adds itself to other programs
  - Does not run independently
- **Both are (usually) designed to spread among computers and to wreak havoc**

# What to do about buffer overflow attacks

# 1. Avoid Overflow Vulnerabilities in Code (!)

```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- For example, use library routines that limit string lengths
  - **fgets** instead of **gets**
  - **strncpy** instead of **strcpy**
  - Don't use **scanf** with **%s** conversion specification
    - Use **fgets** to read the string
    - Or use **%ns** where **n** is a suitable integer

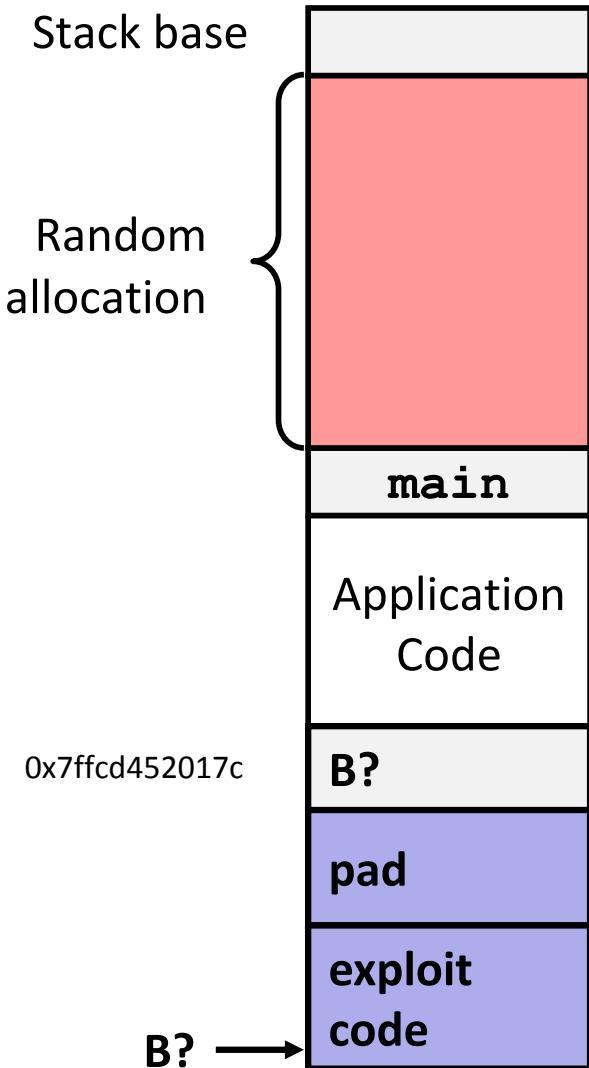
## 2. System-Level Protections can help

### ■ Randomized stack offsets

- At start of program, allocate random amount of space on stack
- Shifts stack addresses for entire program
- Makes it difficult for hacker to predict beginning of inserted code
- E.g.: 5 executions of memory allocation code

local            0x7ffe4d3be87c    0x7fff75a4f9fc    0x7feadb7c80c    0x7ffeaea2fdac    0x7ffcd452017c

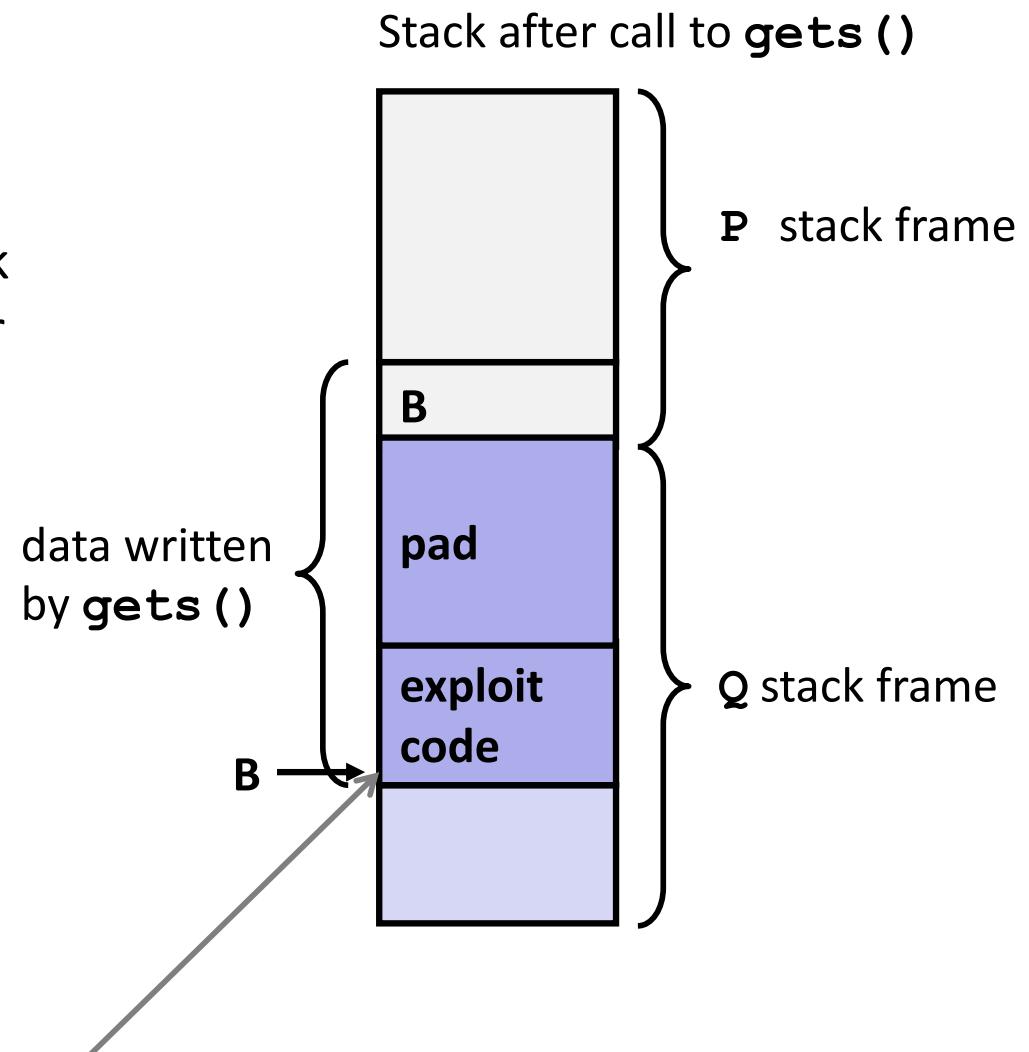
- Stack repositioned each time program executes



## 2. System-Level Protections can help

### ■ Nonexecutable code segments

- In traditional x86, can mark region of memory as either “read-only” or “writeable”
  - Can execute anything readable
- X86-64 added explicit “execute” permission
- Stack marked as non-executable



Any attempt to execute this code will fail

# 3. Stack Canaries can help

## ■ Idea

- Place special value (“canary”) on stack just beyond buffer
- Check for corruption before exiting function

## ■ GCC Implementation

- **-fstack-protector**
- Now the default (disabled earlier)

```
unix> ./bufdemo-sp
Type a string: 0123456
0123456
```

```
unix> ./bufdemo-sp
Type a string: 01234567
*** stack smashing detected ***
```

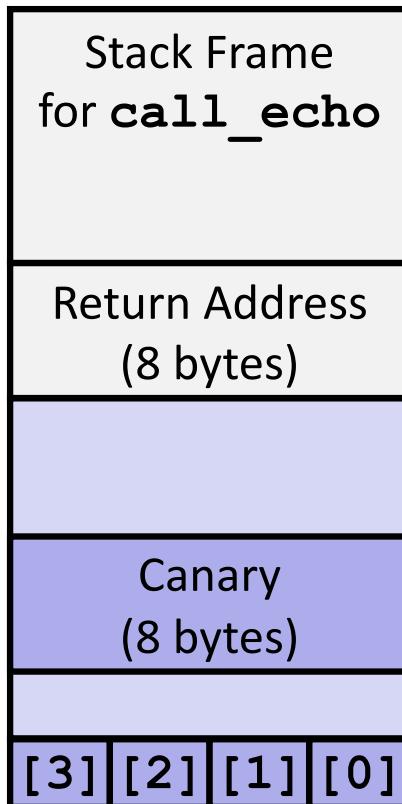
# Protected Buffer Disassembly

echo:

```
40072f: sub    $0x18,%rsp
400733: mov    %fs:0x28,%rax
40073c: mov    %rax,0x8(%rsp)
400741: xor    %eax,%eax
400743: mov    %rsp,%rdi
400746: callq  4006e0 <gets>
40074b: mov    %rsp,%rdi
40074e: callq  400570 <puts@plt>
400753: mov    0x8(%rsp),%rax
400758: xor    %fs:0x28,%rax
400761: je    400768 <echo+0x39>
400763: callq  400580 <__stack_chk_fail@plt>
400768: add    $0x18,%rsp
40076c: retq
```

# Setting Up Canary

*Before call to gets*

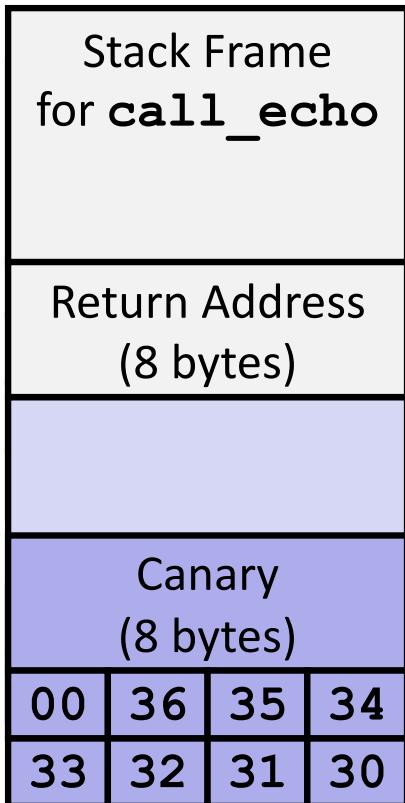


```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movq    %fs:40, %rax # Get canary
    movq    %rax, 8(%rsp) # Place on stack
    xorl    %eax, %eax   # Erase canary
    . . .
```

# Checking Canary

*After call to gets*



```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

**Input: 0123456**

`buf ← %rsp`

```
echo:
    .
    .
    .
    movq    8(%rsp), %rax      # Retrieve from stack
    xorq    %fs:40, %rax      # Compare to canary
    je     .L6                  # If same, OK
    call    __stack_chk_fail   # FAIL
.L6:    .
    .
```

# Return-Oriented Programming Attacks

## ■ Challenge (for hackers)

- Stack randomization makes it hard to predict buffer location
- Marking stack nonexecutable makes it hard to insert binary code

## ■ Alternative Strategy

- Use existing code
  - E.g., library code from stdlib
- String together fragments to achieve overall desired outcome
- *Does not overcome stack canaries*

## ■ Construct program from *gadgets*

- Sequence of instructions ending in `ret`
  - Encoded by single byte `0xc3`
- Code positions fixed from run to run
- Code is executable

# Gadget Example #1

```
long ab_plus_c  
    (long a, long b, long c)  
{  
    return a*b + c;  
}
```

```
00000000004004d0 <ab_plus_c>:  
4004d0: 48 0f af fe imul %rsi,%rdi  
4004d4: 48 8d 04 17 lea (%rdi,%rdx,1),%rax  
4004d8: c3 retq
```

$\text{rax} \leftarrow \text{rdi} + \text{rdx}$

Gadget address = 0x4004d4

- Use tail end of existing functions

# Gadget Example #2

```
void setval(unsigned *p) {  
    *p = 3347663060u;  
}
```

<setval>:

4004d9:	c7 07 d4	48 89 c7	movl \$0xc78948d4, (%rdi)
4004df:	c3		retq

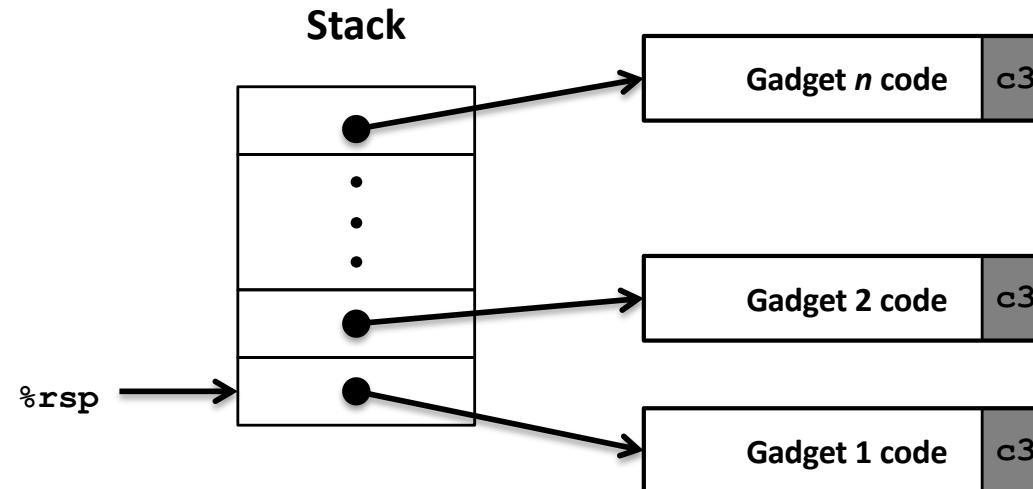
Encodes movq %rax, %rdi

rdi ← rex

Gadget address = 0x4004dc

- Repurpose byte codes

# ROP Execution



- Trigger with `ret` instruction
  - Will start executing Gadget 1
- Final `ret` in each gadget will start next one

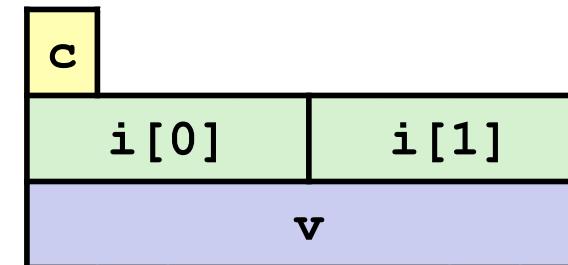
# Today

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
- Unions

# Union Allocation

- Allocate according to largest element
- Can only use one field at a time

```
union U1 {  
    char c;  
    int i[2];  
    double v;  
} *up;
```



```
struct S1 {  
    char c;  
    int i[2];  
    double v;  
} *sp;
```

up+0      up+4      up+8



sp+0      sp+4      sp+8      sp+16      sp+24

# Using Union to Access Bit Patterns

```
typedef union {
    float f;
    unsigned u;
} bit_float_t;
```



```
float bit2float(unsigned u)
{
    bit_float_t arg;
    arg.u = u;
    return arg.f;
}
```

Same as (float) u ?

```
unsigned float2bit(float f)
{
    bit_float_t arg;
    arg.f = f;
    return arg.u;
}
```

Same as (unsigned) f ?

# Byte Ordering Revisited

## ■ Idea

- Short/long/quad words stored in memory as 2/4/8 consecutive bytes
- Which byte is most (least) significant?
- Can cause problems when exchanging binary data between machines

## ■ Big Endian

- Most significant byte has lowest address
- Sparc

## ■ Little Endian

- Least significant byte has lowest address
- Intel x86, ARM Android and IOS

## ■ Bi Endian

- Can be configured either way
- ARM

# Byte Ordering Example

```
union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned long l[1];
} dw;
```

**32-bit**

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]	s[1]		s[2]	s[3]			
i[0]		i[1]					
l[0]							

**64-bit**

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]	s[1]		s[2]	s[3]			
i[0]		i[1]					
l[0]							

# Byte Ordering Example (Cont).

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
       dw.c[0], dw.c[1], dw.c[2], dw.c[3],
       dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

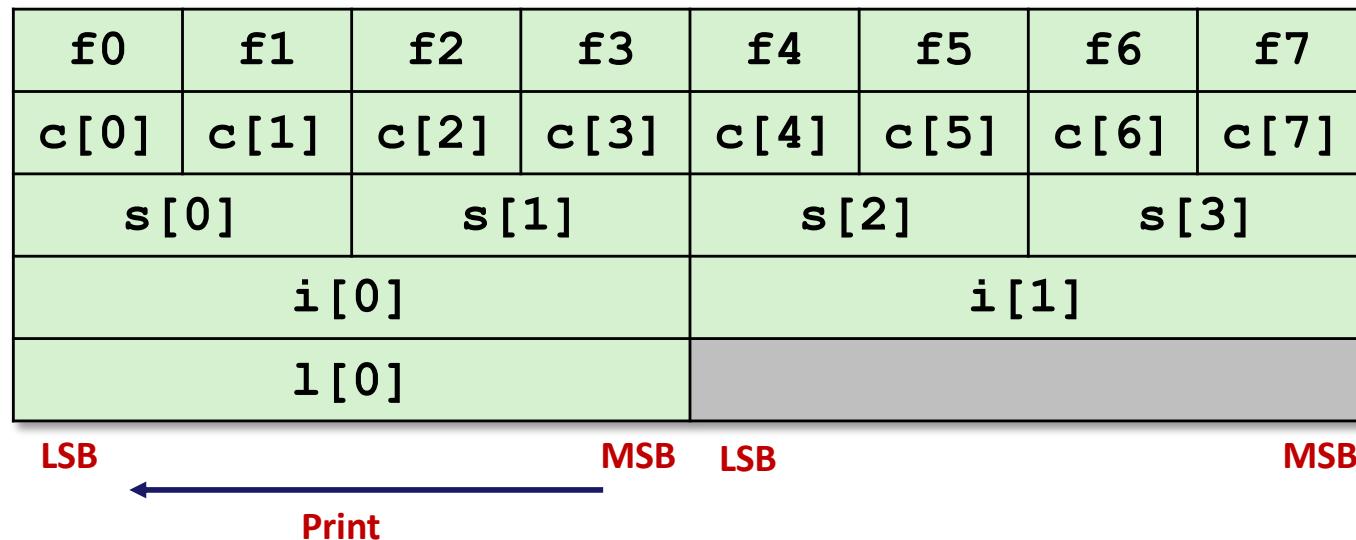
printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
       dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
       dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
       dw.l[0]);
```

# Byte Ordering on IA32

## Little Endian

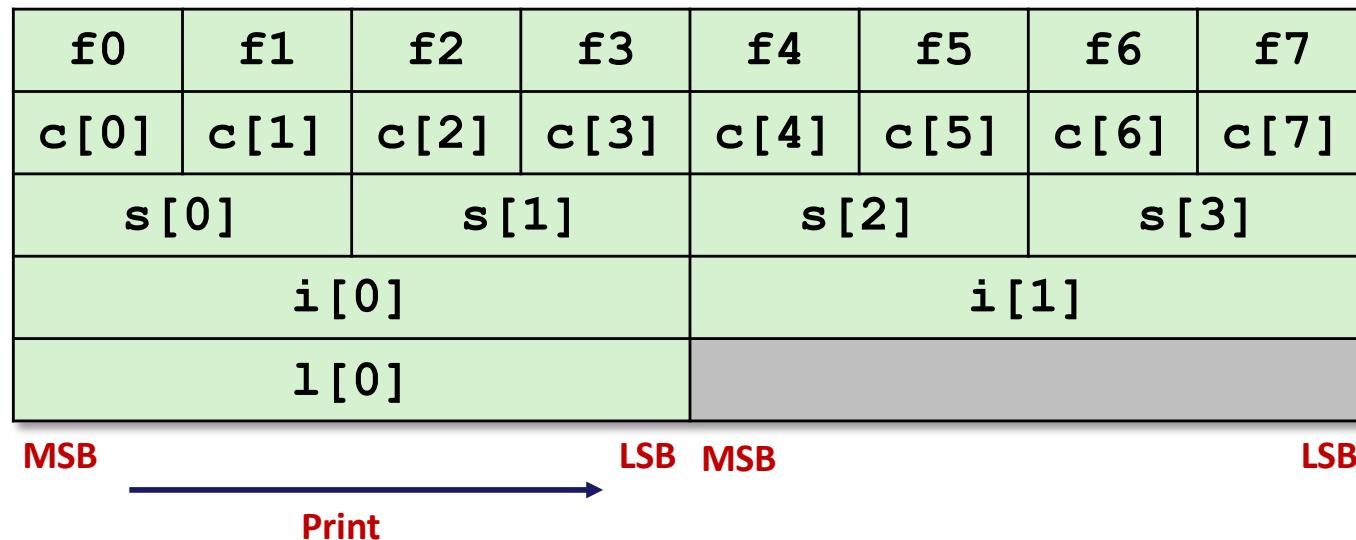


## Output:

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]  
Shorts 0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]  
Ints 0-1 == [0xf3f2f1f0,0xf7f6f5f4]  
Long 0 == [0xf3f2f1f0]

# Byte Ordering on Sun

BigEndian

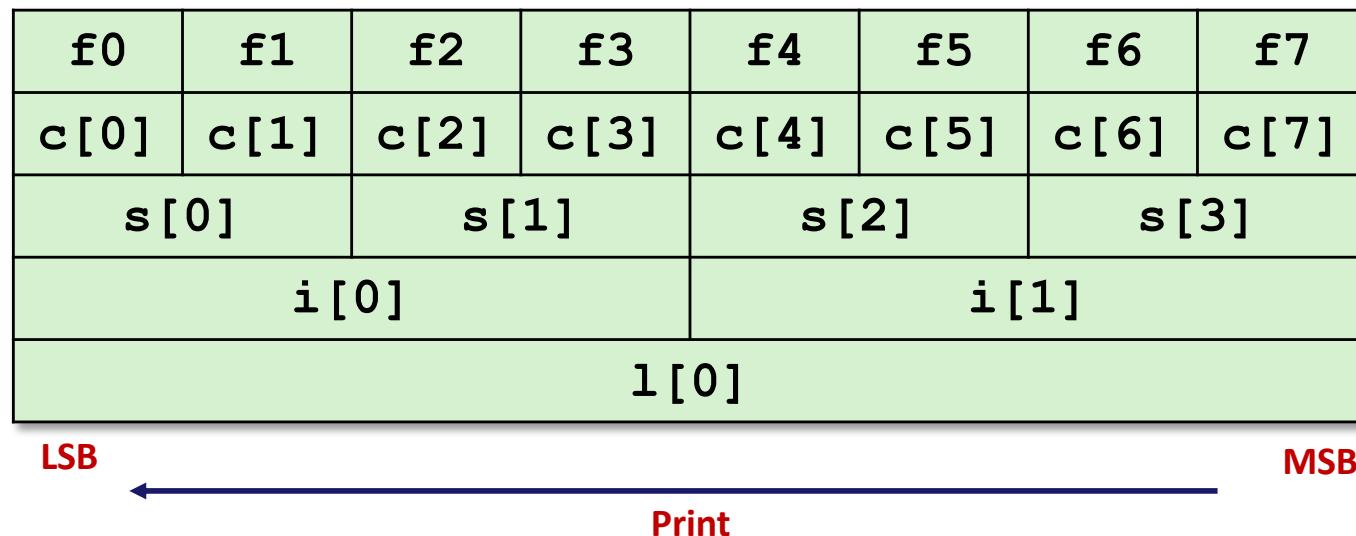


Output on Sun:

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]  
Shorts 0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]  
Ints 0-1 == [0xf0f1f2f3,0xf4f5f6f7]  
Long 0 == [0xf0f1f2f3]

# Byte Ordering on x86-64

LittleEndian



Output on x86-64:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts      0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints        0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long         0    == [0xf7f6f5f4f3f2f1f0]
```

# Summary of Compound Types in C

## ■ Arrays

- Contiguous allocation of memory
- Aligned to satisfy every element's alignment requirement
- Pointer to first element
- No bounds checking

## ■ Structures

- Allocate bytes in order declared
- Pad in middle and at end to satisfy alignment

## ■ Unions

- Overlay declarations
- Way to circumvent type system